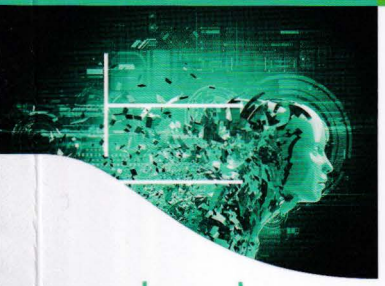
INFORMATIQUE
TECHNIQUE



Expert

Maîtrisez la Data Science avec Python

En téléchargement

Notebooks des exemples

Eric DEMANGEL

eni

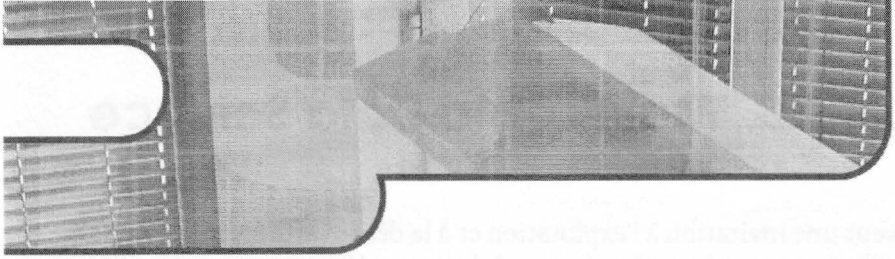
Expert IT

Maîtrisez la Data Science avec Python

En téléchargement

Notebooks des exemples

Eric DEMANGEL



Avant-propos

L'élaboration de ce livre est arrivée de manière fortuite, mais comme bien des événements dans la vie, ce qui semble être le fruit du hasard révèle parfois une logique sous-jacente inattendue.

Après avoir eu le privilège d'accompagner une cinquantaine d'étudiants en tant que mentor en data analyse et data science, j'ai pris conscience que ce projet éducatif revêtait une signification bien plus profonde. En réalité, il symbolisait une forme d'aboutissement naturel du volet pédagogique de mon engagement professionnel dans le but de consolider et de partager les connaissances acquises au fil du temps.

Conçu comme un outil efficace pour la pratique de la data science, il vise à fournir des solutions concrètes aux défis rencontrés dans ce domaine en constante évolution.

Il est important de souligner que chaque algorithme, chaque code et chaque méthode présentés ici sont le fruit d'une sélection minutieuse. Et plutôt que de noyer le lecteur sous un flot d'informations techniques, j'ai fait le choix délibéré de mettre en avant les éléments essentiels pour comprendre et maîtriser la data science. À de maintes reprises, j'ai appliqué les principes de la data science à des données issues de cette discipline elle-même en bâtissant des exemples sur des données réelles liées à la discipline, pour à la fois montrer et communiquer des informations.

Ce livre se veut une invitation à l'exploration et à la découverte sans prétendre être une feuille de route exhaustive. Les modules proposés ne sont pas détaillés de A à Z ; leur compréhension complète nécessite parfois une démarche active de recherche et d'approfondissement. C'est là que réside le véritable défi, mais aussi la véritable satisfaction : dans la quête du savoir et dans la capacité à résoudre des problèmes complexes de manière autonome.

En encourageant le lecteur à s'engager activement dans ce processus d'apprentissage, je suis convaincu que ce livre peut servir de tremplin vers d'innombrables opportunités professionnelles et intellectuelles dans le domaine fascinant de la data science. Je vous invite donc à plonger dans ces pages avec curiosité et enthousiasme, en souhaitant vous donner l'envie de pratiquer cette discipline en constante évolution.

Je tiens à exprimer ma profonde gratitude à ma famille et à mes amis pour leur soutien inestimable tout au long de ce projet. Un merci tout particulier à Kasia pour ses encouragements constants et sa capacité à détecter mes fautes d'orthographe et de formulation. Merci également à Thierry, qui a toujours été disponible pour relire mon travail, et à Alexandre, pour son immense implication à traquer mes nombreuses maladresses et ses judicieux conseils.

Bonne lecture, et que cette aventure vous soit aussi enrichissante qu'elle l'a été pour moi.

Les éléments à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence de l'ouvrage **EIPYTDATA** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

Chapitre 1
Introduction

Chapitre 2
Bases de Python et environnements

Chapitre 3
Préparer les données avec Pandas et Numpy

Chapitre 4
DataViz avec Matplotlib, Seaborn, Plotly

Chapitre 5
Analyse des données

Chapitre 6
Le Machine Learning avec Scikit-Learn

Chapitre 7
L'apprentissage supervisé

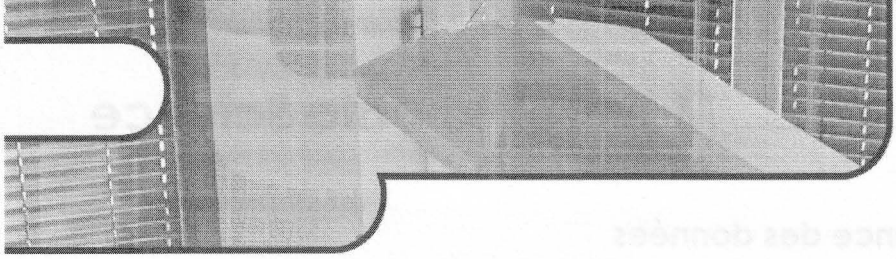
Chapitre 8
L'apprentissage non supervisé

Chapitre 9
Modéliser le texte et l'image

Chapitre 10

Mener un projet de data science avec Python

Conclusion



Chapitre 1
Introduction

1. Des données partout

Tout est donnée. La moindre information stockée sur un support numérique peut être récupérée et exploitée. Peu importe la nature du fichier : document texte, tableur, image, logs ou même un fichier vidéo, tout est exploitable. Ces fichiers ne se limitent pas à nos ordinateurs, nous les retrouvons partout : dans notre téléphone, notre voiture, sur le Web ou dans de simples capteurs. Et il n'est question ici que de données préexistantes. Cette immense bibliothèque de savoirs peut encore être enrichie par tout ce qui n'est pas encore enregistré : des photos papier, des livres ou des choses qui ne sont actuellement que dans notre tête comme une histoire personnelle ou une recette de cuisine.

Il n'y a pas de pénurie de données. Parfois, nous pourrions même avoir l'impression qu'il y en a trop. C'est pourquoi l'expression « infobésité » a été créée pour décrire ce phénomène.

Cette surabondance nous ouvre néanmoins les portes de possibilités infinies en termes de connaissances nouvelles et tout un champ des possibles qui en découle. Nous sommes dans une situation où ce n'est pas tant la matière première qu'est la donnée qui devrait nous préoccuper mais notre capacité à en évaluer sa qualité, la sélectionner, l'acquérir et la transformer pour en extraire des informations utiles. Tout processus de data science part des données et nous allons nous attacher pour commencer à étudier leur provenance.

1.1 Provenance des données

Ainsi, les données sont omniprésentes, en constante évolution et prennent diverses formes. Il suffit de les mettre à jour, et l'expression « data mining », ou extraction de données, illustre clairement ce processus consistant à extraire de l'information cachée et précieuse par analogie avec l'extraction de gemmes.

Les gisements d'informations ne sont pas équivalents en termes de qualité. Nous pouvons définir trois grandes familles de provenance des données.

1.1.1 Le Web

Le Web est à l'origine un gigantesque réservoir anarchique de données. Avec la prise de conscience de leur importance, nous avons assisté au développement de l'open data. Mais qu'est-ce que l'open data concrètement ? Nous pourrions la définir comme la volonté d'organiser les données accumulées pour en faciliter l'accès et l'utilisation. Ce mouvement a pris beaucoup d'ampleur et nous pouvons désormais avoir accès rapidement et simplement à tout type d'information. Toutes les organisations, au premier rang duquel les États, ont mis à disposition gratuitement leurs données au grand public dans des formats accessibles et réutilisables. Ces données vont ensuite pouvoir être utilisées par tout un chacun pour des projets de toute nature, qu'ils soient commerciaux ou non. Voici une liste non exhaustive de liens pour se familiariser avec :

- Données gouvernementales portant sur de multiples sujets :
  - https://www.data.gouv.fr/fr/datasets/
  - https://catalog.data.gov/dataset
- Données d'ONG :
  - https://www.who.int/fr/data
  - https://www.fao.org/faostat/fr/#home
- Données locales :
  - https://opendata.paris.fr/pages/home/
  - https://data.montpellier3m.fr/
  - https://opendata.bordeaux-metropole.fr/pages/accueil/

- Sites généralistes ou spécialisés :
  - https://fr.wikipedia.org/wiki/Wikipédia:Accueil_principal
  - https://fr.openfoodfacts.org/data
  - https://www.kaggle.com/datasets
  - https://data.opendatasoft.com/pages/home/
  - https://ourworldindata.org/

En dehors de l'open data, des milliards de données attendent encore de délivrer leurs secrets. Mais il faudra alors le faire nous-mêmes en veillant à respecter les conditions de récupération et d'utilisation des informations des sites. C'est le web scraping auquel nous nous initierons simplement un peu plus tard.

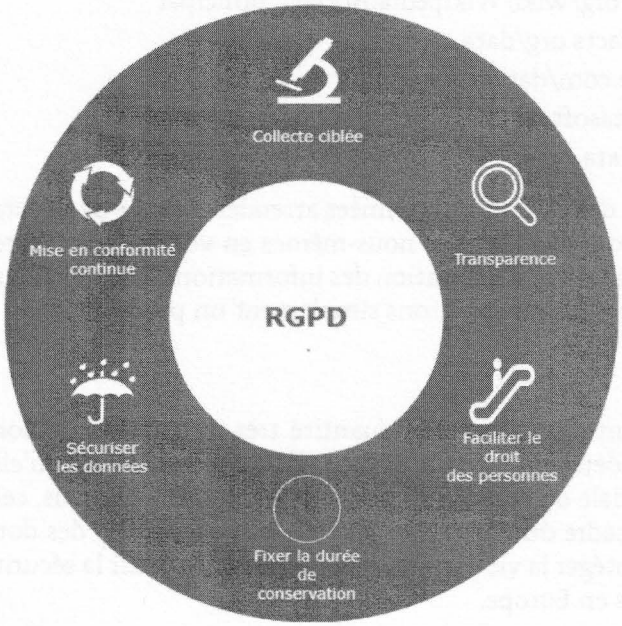
1.1.2 Les données privées

Les entreprises détiennent également une quantité très importante de données qui vont former la deuxième grande source. Toute information, qu'elle soit financière, commerciale ou technique, peut être analysée. Toutefois, cela doit être réalisé dans le cadre du règlement général de la protection des données (RGPD) visant à protéger la vie privée des individus et garantir la sécurité des données personnelles en Europe.

Remarque

Le RGPD est européen mais il a des équivalents dans le monde entier comme PIPEDA au Canada, LGPD au Brésil ou APPI au Japon. Tous ont pour but de protéger les données personnelles mais il existe des variations spécifiques à chaque juridiction.

Voici un diagramme présentant les six grandes règles du RGPD à respecter :



Au niveau de l'hébergement des données, il convient également de bien s'assurer qu'elles le sont dans une zone soumise à l'application du RGPD.

1.1.3 Créons nos propres données

Enfin, si toutes les informations disponibles publiques ou privées ne répondent pas à nos besoins, s'offre à nous la possibilité de les créer nous-mêmes. En s'imposant une certaine rigueur scientifique dans le recueil, il n'existe aucune restriction pour créer notre propre bibliothèque et c'est d'ailleurs l'unique possibilité si notre besoin consiste à mesurer le nombre de pas réalisés dans la journée ou notre temps passé sur une mission. Attention toutefois à respecter les règles d'utilisation des différents supports que nous pourrions utiliser pour nos expérimentations, comme les photos de lieux ou de personnes, la musique, les œuvres littéraires, etc.

Maintenant que nous avons identifié leur provenance, il convient de faire un point sur leurs formes.

1.2 Forme des données

Récupérer les données n'est qu'une première étape. Qu'elles soient stockées localement ou sur un cloud, ces données numériques se trouvent dans des fichiers ayant différentes natures : fichiers texte, Excel, HTML, JSON, JPEG, etc. Au sein de ces fichiers, l'information contenue peut se présenter sous plusieurs formes :

- structurée, comme dans le cas de fichiers Excel ou les bases de données relationnelles ;
- semi-structurée, comme JSON, XML ou NoSQL ;
- structurée, mais de manière inadéquate : tel est le cas si nous avons récupéré une page web via un web scraping. Le contenu sera alors structuré en HTML mais ne conviendra pas à une exploitation directe et nécessitera une remise en forme ;
- non structurée pour des fichiers texte, par exemple, où l'information n'est pas organisée et donc exploitable en l'état ;
- ajoutée à cela, la gestion des images nécessitant l'usage de techniques particulières afin d'extraire l'information contenue. Nous verrons dans le chapitre dédié comment y parvenir.

Les données vont, pour finir, se présenter sous la même forme qu'un tableau Excel avec des lignes et des colonnes. Il est utile à ce stade de faire un topo sur les terminologies employées et l'organisation de la table.

En ligne, nous trouverons les objets d'étude, qui peuvent être des clients, des produits, des oiseaux, etc. Nous parlons ici d'observations, mais nous rencontrerons différentes façons de les nommer selon le domaine ou le logiciel utilisé. De même, pour les colonnes présentant les caractéristiques telles que l'âge, le poids, la durée, le volume, etc., qui qualifient les observations. Voici un tableau des différentes appellations possibles renvoyant toutes à la même chose :

Ligne	Colonne
Enregistrement	Attribut
Entrée	Champ

Ligne	Colonne
Ligne	Colonne de donnée
Objet	Variable
Instance	Caractéristique
Élément	Propriété
	Champs

Maintenant que nous avons défini la forme la plus commune de présentation, finissons par un point sur le volume de ces données.

1.3 Volumétrie

L'analyse des données requiert un minimum de 50 observations pour démarrer. Ce seuil minimal est nécessaire pour engager le travail analytique. De plus, le nombre d'observations doit être soigneusement évalué en fonction du nombre de variables présentes : il est recommandé d'avoir un rapport d'au moins 100 observations pour une variable afin de garantir une analyse robuste et fiable. Et il n'y a pas de limite à la hausse : plus nous aurons d'observations, plus les modèles prédictifs seront fiables et robustes.

Cette volumétrie est un élément très important à prendre en compte car elle a des conséquences sur toute la suite des opérations : plus il y a de données, plus le stockage prend de la place, plus les calculs prennent du temps ce qui peut même aboutir à des blocages dans les cas de dépassements de mémoire. Il est donc primordial de s'organiser en conséquence. Nous choisirons de laisser de côté les architectures Big Data, conçues pour gérer des volumétries massives, car les défis posés par des volumes de données plus modestes demeurent déjà importants. Cette décision nous permettra de mieux concentrer nos efforts sur la compréhension de la data science dont nous allons nous attacher à définir les contours.

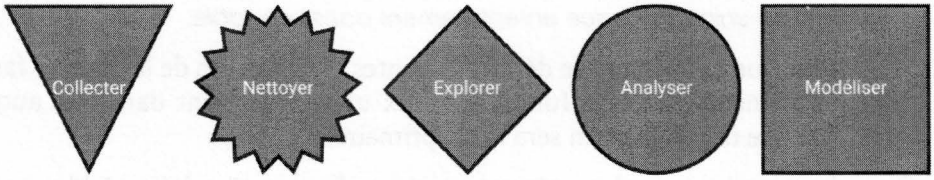
2. La data science

La data science est un domaine pluridisciplinaire qui mobilise une variété de techniques statistiques, mathématiques et informatiques.

Son objectif se résume en deux grandes étapes :

- **Feature engineering** : collecter, nettoyer, explorer et analyser les données.
- **Modélisation** : mettre en place, optimiser et maintenir les systèmes décisionnels.

Le schéma suivant nous servira de vue d'ensemble et facilitera la navigation dans ce livre :



2.1 Feature engineering

Le feature engineering, littéralement ingénierie des caractéristiques en français, représente la première grande étape de l'intervention d'un data scientist. Elle englobe toutes les opérations précédant la modélisation des systèmes décisionnels tels que l'acquisition, le nettoyage, l'exploration, l'analyse, la sélection et la préparation des données. Nous pourrions la définir comme la phase de préparation préalable à l'action. C'est une étape cruciale car la qualité de notre modélisation future dépendra largement du soin apporté à cette phase. Attachons-nous à détailler, dans l'ordre, ces différentes étapes en nous appuyant sur un exemple fictif.

2.1.1 La collecte des données

Un glacier nous a missionnés pour l'aider à comprendre et améliorer ses ventes. Il nous en fournit le détail. Nous allons commencer par vérifier si les données se présentent sous une forme exploitable. En effet, en fonction des analyses souhaitées, il sera souvent nécessaire de modifier la présentation des données pour répondre à nos besoins. De plus, ces données sont très souvent composées de plusieurs tables pour lesquelles il faudra identifier les clés primaires pour les lier.

Remarque

Rappelons qu'une clé primaire est une colonne ou un ensemble de colonnes dont les valeurs sont uniques pour chaque ligne et qui permettent d'identifier de manière unique chaque enregistrement dans une table.

Supposons que nous ayons le détail des ventes et le parfum de la glace, il faudra certainement lier ce parfum à son prix unitaire présent dans une autre table. Dans ce cas, le parfum sera la clé primaire.

Dans des cas plus complexes, le recours à un dictionnaire des variables peut s'avérer nécessaire. Ce document servira de guide détaillant la signification, le type des variables ou toute autre information propice à la compréhension des données. Il ne faut pas passer à l'étape suivante du nettoyage avant de s'être pleinement familiarisé avec l'ensemble des données qui vont être utilisées.

2.1.2 Le nettoyage

Les données nécessitent souvent une étape de nettoyage pour être prêtes à l'emploi. Elles nécessitent toujours des remises en forme ou des corrections. Un cas classique pour notre exemple serait d'afficher le prix avec le symbole € qui indique au consommateur la monnaie utilisée mais va nous empêcher de faire des calculs car la variable risque de ne pas être considérée comme numérique. Ces modifications sont légères et rapides à corriger mais d'autres, comme les valeurs manquantes, vont être plus difficiles à arbitrer. Dans ce cas, il faut d'abord se demander si l'absence de réponse est possible ou acceptable. Si ce n'est pas le cas, nous pouvons compléter en faisant ce qu'on appelle une imputation.

Cette opération consiste à remplacer les valeurs manquantes par la moyenne de la variable, sa médiane ou sa modale pour les variables numériques et simplement la modale pour les variables catégorielles. Il existe également des techniques plus sophistiquées que nous développerons dans la partie pratique dédiée. Dans les cas où l'imputation s'avère trop hasardeuse, nous pourrons envisager de ne pas prendre en compte les données trop peu remplies. Cet arbitrage est un vrai enjeu et nécessite de s'interroger sur les implications de chaque option.

Une fois le nettoyage réalisé, nous pouvons commencer à manipuler les données pour les évaluer.

2.1.3 L'exploration

Nous ne savons pas a priori quelles sont les caractéristiques de notre jeu de données. Commençons par découvrir ses dimensions, la qualité de remplissage puis, en manipulant et en visualisant, nous apprendrons combien de glaces sont vendues par jour en moyenne, est-ce qu'il y a des parfums qui ne se vendent pas, quelle fut la meilleure et la pire journée, est-ce que les ventes ont tendance à augmenter ou non. Nous allons alors prendre conscience de la réalité du terrain. La visualisation va devenir une alliée précieuse en mettant tout de suite en exergue les spécificités des données étudiées. Voici un exemple pour illustrer ce phénomène :

Parfum	Chiffre d'affaires
Pistache	3584
Cerise	2800
Fraise	3822
Vanille	4295
Chocolat	4834

Les barres parlent d'elles-mêmes et nous percevons tout de suite le chiffre d'affaires généré pour chaque parfum. Cette familiarisation avec le domaine d'étude va engendrer des constats et des questions que nous allons ensuite vérifier dans la partie suivante.

2.1.4 L'analyse

L'analyse suit directement notre exploration. Nous allons d'abord utiliser des tests statistiques pour déterminer la généralisation des observations constatées dans l'étape précédente. Cette phase se nomme inférence statistique et consiste à valider grâce à des méthodes de généralisation si ce que nous constatons lors de l'exploration de notre échantillon pourrait se généraliser à toute la population.

Concrètement, si nous étudions les ventes de glaces entre la semaine et le week-end, la différence devra atteindre un certain niveau en fonction de l'échantillon pour que nous puissions dire qu'il y a une différence significative et ainsi considérer ce type de découpage pertinent. En dessous de ce seuil, la différence pourrait provenir de l'effet de tirage et ne devrait pas être prise en compte. Les effectifs jouent ici un rôle crucial : plus nous avons d'observations, plus les marges d'incertitudes autour des valeurs constatées seront faibles et plus une différence, si minime soit-elle, pourrait se révéler significative.

Nous allons ensuite étudier les variables une à une, deux à deux puis globalement afin de parfaire complètement notre connaissance des données et de leur fonctionnement. Respectivement, ces études se nomment analyse univariée, bivariée et multivariée. Nous verrons dans le détail au chapitre Analyse des données comment mener à bien toutes ces analyses. À l'issue de cette étape, nous sommes quasiment prêts à l'action. Quasiment prêts car, forts de nos connaissances, nous avons désormais la capacité de créer ou d'ajouter une ou plusieurs variables qui pourraient jouer un rôle prépondérant dans notre modèle.

Remarque

Il est courant d'utiliser l'expression système décisionnel à la place de modèle.

2.2 La modélisation

Nous abordons donc cette deuxième grande partie avec un jeu de données nettoyé contenant des variables a priori pertinentes pour bâtir notre modélisation. Comme pour le feature engineering, nous allons suivre un processus qui devra là aussi s'inscrire dans une certaine dynamique faisant la part belle à l'expérimentation, le questionnement et la construction progressive de la meilleure solution possible en l'état de nos connaissances. Nous allons décrire toutes les étapes mais soulignons ici que toutes n'auront pas forcément lieu.

2.2.1 La sélection et la préparation des données

Forts des nouvelles connaissances issues de l'analyse, nous sommes désormais en mesure de sélectionner les variables pertinentes pour notre système décisionnel. Cette sélection n'est qu'une première étape car nous pourrions être amenés à la reconsidérer en fonction des résultats des modélisations. Il est important ici de souligner que le travail du data scientist est dynamique et ne se résume pas en une succession de tâches. Il faut en effet respecter un certain ordre mais ne pas hésiter à revenir en arrière si nous constatons des choses qui remettent en cause les étapes précédentes.

Par exemple, lors des modélisations, il peut apparaître que nous avons trop peu de variables car nos scores sont trop faibles ou bien, qu'il y a trop de variables car les résultats du jeu d'entraînement sont beaucoup trop éloignés du jeu de test. Dans le premier cas de figure où les scores sont trop faibles, il serait pertinent d'expérimenter des changements d'échelles, de rechercher et retirer des valeurs aberrantes ou d'augmenter le nombre de variables. Dans le deuxième cas où le système est trop complexe, il serait plutôt conseillé de réduire le nombre de variables. Toutes ces situations soulignent bien le côté dynamique de l'expérimentation où la liste des variables n'est jamais vraiment définitive. Mais en adoptant cette approche itérative et adaptable, nous renforçons la robustesse et la précision de notre travail sur les données.

Place maintenant à la préparation des données. Cette étape consiste à les préparer de manière à faciliter et ne pas biaiser les modélisations. Pour commencer, regardons la nature de nos différentes variables et interrogeons-nous sur leurs échelles respectives. Celles-ci sont rarement toutes de même nature et doivent être mises à l'échelle pour que certaines, du fait de leur nature, n'écrasent pas les autres. Ce pourrait être le cas de données contenant des âges allant, par exemple, de 20 à 60 ans, et des salaires qui peuvent s'étaler de quelques centaines à plusieurs dizaines de milliers. Une mise à l'échelle est ici nécessaire pour que l'un n'éclipse pas complètement l'autre. Voyons ceci comme un moyen de mettre les coureurs sur la même ligne de départ. Nous verrons en détail les différentes techniques à mettre en œuvre dans le chapitre Le machine learning avec Scikit-Learn.

Pour finir, rappelons que les données ne sont pas toutes de type numérique, certaines contiennent des informations textuelles. Il sera alors nécessaire de les mettre en binaires, c'est-à-dire de les transformer en un contenu chiffré de type 0 / 1 afin de pouvoir les intégrer à nos algorithmes qui, dans l'immense majorité, n'acceptent que des informations numériques.

2.2.2 La séparation des données

La modélisation consiste à prévoir ou classer une valeur. Cette valeur se nomme la variable cible. Les variables utilisées pour la trouver sont nommées les variables explicatives. Nous allons donc commencer par séparer la cible des autres en les mettant dans deux jeux de données différents.

Imaginons que nous cherchions à déterminer la consommation d'énergie d'un bâtiment par rapport à différents facteurs jugés pertinents comme la surface, le nombre d'occupants, la période et le type de chauffage. Ces facteurs explicatifs resteraient ensemble dans une même base de données et leur valeur de consommation correspondante connue serait mise de côté dans une autre table montrant à l'algorithme ce qu'il devrait trouver. Nous retrouvons ici le principe de machine learning, littéralement machine qui apprend, dans le fait que la machine se nourrit d'exemples pour s'entraîner à prédire.

Cet entraînement à partir des données va justement nécessiter une deuxième étape de séparation entraînant la création aléatoire d'un jeu d'entraînement et d'un jeu de test. Le jeu d'entraînement est l'exemple fourni à l'algorithme pour qu'il puisse faire les bons réglages en regardant ce qu'il doit trouver. Mais pour vraiment évaluer la pertinence de notre système, il convient de simuler une condition réelle en le mettant à l'épreuve sur un jeu de données qu'il n'a pas vu et qui a pour vocation de mesurer l'écart entre les valeurs prédites et les valeurs réelles. Ce jeu de test représente entre 20 et 30 % des données et il est absolument nécessaire pour évaluer correctement la performance du système. Cette étape est le minimum et nous verrons ensuite comment encore renforcer cet aléa lors de la mise en pratique. Nous sommes maintenant prêts pour expérimenter.

2.2.3 La phase d'expérimentation et d'évaluation

Il existe quantité d'algorithmes, chacun opérant de manière singulière. De ces algorithmes, on ne peut pas dire qu'il y en ait un qui domine les autres, tout dépend du domaine d'étude. L'expression anglaise consacrée, « *no free lunch* », vise justement à souligner qu'aucun ne fonctionne toujours mieux que les autres pour tous les types de problèmes. Pas de solution universelle, chaque cas est unique. La phase d'expérimentation et d'évaluation va ainsi consister à essayer les différents algorithmes et jouer sur les réglages propres à chacun pour choisir celui qui répondra le mieux à nos besoins. Nous prendrons soin de toujours utiliser les mêmes jeux d'entraînement et de test pour tous afin que certains ne soient pas lésés ou avantagés par un effet de tirage heureux. Il est important de préciser ici que le choix ne s'opère pas nécessairement et uniquement sur la performance. La performance pure obtenue de manière fortuite n'a pas de valeur ici et le candidat idéal devra faire montre de stabilité à ce niveau. Outre la recherche de hautes performances stables, certains critères comme la rapidité de temps de réponse ou le faible poids de stockage pourraient dans certains cas s'avérer cruciaux et emporter la mise. Nous sommes vraiment dans le sur-mesure. La technicité stricte ne suffit pas. L'expérimentation et la recherche constante de l'adéquation au cahier des charges sont primordiales ici. À l'issue de cette étape, nous sommes à même de proposer la solution optimale et de quantifier la différence avec ses concurrents.

2.2.4 La finalisation

À ce stade, nous avons arrêté un choix qui répond aux besoins sur le fond mais pas sur la forme. Nous n'avons pour le moment qu'un programme, fonctionnel certes, mais pas très accessible. Nous pourrions ainsi être amenés à créer une application. Cette application pourrait prendre la forme d'une simple « *application programming interface* » (API) qui renverrait juste un résultat lorsqu'elle serait sollicitée. Dans d'autres cas, il pourrait nous être demandé de développer un tableau de bord facilitant l'utilisation de la solution. Quelle que soit l'issue, soulignons ici que le recours à une API sert de facilitateur en permettant l'intercommunication de différents systèmes. Notre solution intégralement développée en Python pourrait être simplement utilisée par une personne codant dans n'importe quel autre langage.

2.2.5 La présentation des résultats

Nous pouvons à présent dévoiler la solution retenue au client. S'assurer qu'elle répond bien aux besoins initiaux. C'est le moment de faire preuve de pédagogie en expliquant la démarche et le fonctionnement dans le but d'échanger et que les personnes puissent s'approprier la solution. En somme, faire mûrir le projet. Ces échanges seront l'occasion de faire un premier bilan et d'ouvrir la voie à des interventions ou des évolutions ultérieures.

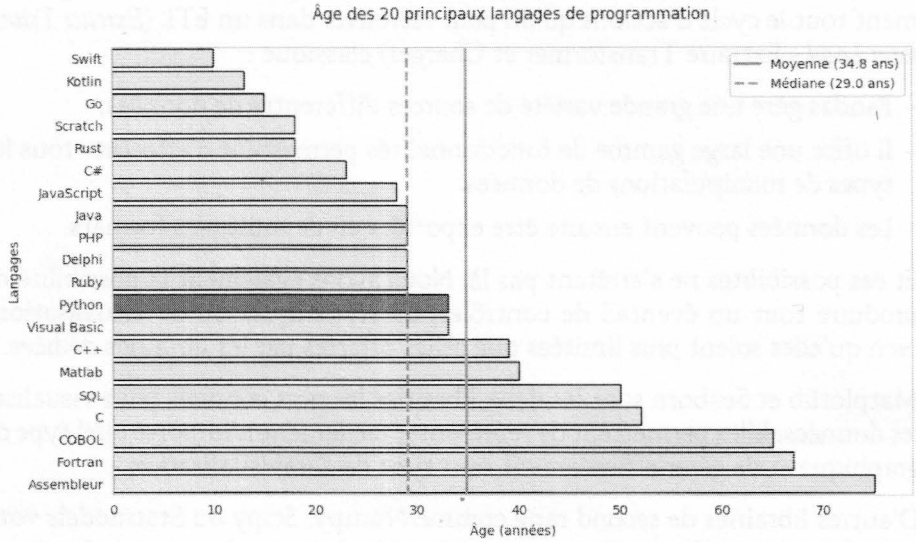
2.2.6 La maintenance

Les choses évoluent avec le temps et il peut être nécessaire de définir de futures interventions pour maintenir la solution à jour ou la renforcer. Différents facteurs peuvent justifier cette nécessité. Nous pourrions citer pour commencer le cas d'un jeu de données qui serait soumis à des variations saisonnières (fête ou autre), ce qui nécessiterait des ajustements ultérieurs face à des contextes changeants. Dans un autre cas de segmentation de clientèle où nous aurions constitué des groupes répondant à des profils de clients, des mises à jour pourraient être nécessaires pour refléter ces changements et maintenir la précision. Enfin, le programme pourrait souffrir de bugs de fonctionnement, ce qui amènerait à revoir le code pour renforcer sa robustesse.

Après avoir souligné l'importance des données et leur utilisation dans le cadre d'un processus de data science, nous allons maintenant nous intéresser au langage informatique qui constitue le troisième pilier essentiel : Python.

3. Python

Python a été créé en février 1991, par Guido van Russom, un développeur néerlandais. Ce langage, d'âge moyen par rapport à ses pairs, a connu depuis de nombreuses versions pour devenir progressivement le plus populaire au monde. Nous allons présenter ici les origines de son succès.



3.1 Les atouts naturels de Python

Python est reconnu pour sa simplicité et sa flexibilité, ce qui en fait un choix pertinent pour quiconque débute dans la programmation. Les utilisateurs plus expérimentés vont en outre apprécier sa polyvalence, sa puissance et sa capacité à favoriser le travail en équipe pour collaborer et maintenir des projets. Avec de tels atouts, de nombreuses communautés se sont formées au fil du temps et ont activement contribué à sa diffusion dans les milieux professionnels, renforçant encore sa popularité. Progressivement, il est devenu le langage le plus populaire et a su évoluer avec son temps. Il doit certainement son secret de longévité aux différentes et puissantes librairies mises en place par sa communauté.

3.2 Les librairies spécialisées

La polyvalence de Python n'a pas été laissée en jachère. Dans chaque domaine, des modules dédiés ont vu le jour et sont devenus au fil du temps des références.

Ainsi, dans le domaine de la data, Pandas s'est imposé car il facilite grandement tout le cycle d'actions qu'on peut retrouver dans un ETL (*Extract Transform Load* - Extraire Transformer et Charger) classique :

- Pandas gère une grande variété de sources différentes de données.
- Il offre une large gamme de fonctionnalités permettant d'effectuer tous les types de manipulations de données.
- Les données peuvent ensuite être exportées en de multiples formats.

Et ces possibilités ne s'arrêtent pas là. Nous avons également la possibilité de produire tout un éventail de contrôles, de statistiques ou de visualisations bien qu'elles soient plus limitées que celles offertes par les librairies dédiées.

Matplotlib et Seaborn sont les deux librairies les plus connues pour visualiser les données. Elles permettent de représenter facilement n'importe quel type de graphique et de gérer efficacement tout type de datavisualisation.

D'autres librairies de second rang comme Numpy, Scipy ou Statmodels vont nous fournir tous les outils et tests nécessaires à nos analyses statistiques.

Enfin, pour la partie modélisation, plusieurs outils majeurs sont disponibles : Scikit-Learn, qui couvre une large gamme de l'apprentissage automatique, ainsi que Tensorflow et Keras pour le deep learning. Pour le traitement du langage naturel (NLP), des solutions telles que SpaCy, NLTK ou Gensim sont également proposées.

Nous verrons plus tard leur fonctionnement mais retenons ici que nous avons absolument tout ce qu'il faut pour gérer tous les besoins du data scientist. Notons pour finir que cette liste de librairies n'est absolument pas exhaustive et qu'il existe de multiples trésors cachés à découvrir.

3.3 Plus encore

Tous les besoins du data scientist sont assouvis mais Python va encore au-delà. Avec lui, nous pouvons aussi gérer tout le reste comme la création d'une interface en local, d'une API ou d'une application hébergée, la gestion de l'hébergement que ce soit au niveau du frontend ou du backend. Les possibilités sont immenses. Il serait en réalité plus simple d'énumérer ses limites :

- les programmes nécessitant de très fortes contraintes de gestion et de puissance de mémoire ;
- les applications mobiles natives nécessitant une intégration étroite en Android ou iOS ;
- les jeux vidéo de très haute qualité.

C'est tout. Partout ailleurs, Python répondra à tous nos besoins.

Après cette présentation théorique des trois piliers que sont les données, la data science et Python, il est temps de passer à la pratique.

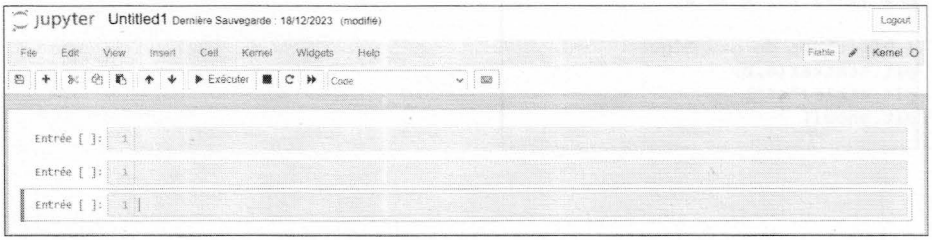
Chapitre 2
Bases de Python et environnements

1. Les notebooks

Ce chapitre n'a pas pour but d'initier à Python, mais plutôt de préparer un environnement de travail fonctionnel. Il est à considérer comme une mise au point avant de poursuivre. Pour ceux qui connaissent déjà les notebooks et sont utilisateurs réguliers de Python, le chapitre suivant peut être abordé directement. Pour tous les autres, nous allons commencer par présenter un support alternatif de programmation que nous utiliserons : le notebook.

1.1 Principe du notebook

Comparé à un programme classique contenant du code, le notebook offre une approche révolutionnaire de la programmation en apportant de l'interactivité au cadre austère habituel. Voici à quoi il ressemble :



1.1.1 Fonctionnement par cellule

Le notebook fonctionne par cellule. C'est la première marque car chaque cellule peut être lancée indépendamment des autres. Cette approche est très pertinente en termes de pédagogie car elle va permettre à l'apprenant de lancer spécifiquement la ou les cellules souhaitées et faciliter la manipulation. De plus, chaque cellule peut être de deux natures différentes et contenir soit du code soit du texte.

1.1.2 Possibilité d'annoter le code

Le texte va apporter cette deuxième couche d'interactivité en permettant à l'utilisateur d'annoter ce qu'il fait, de glisser des remarques ou tout simplement permettre d'ajouter des chapitres et des sous-chapitres. Nous passons ainsi d'un simple document contenant du code à un support de présentation contenant du code. D'autant que les cases contenant du texte peuvent accueillir et interpréter les langages Markdown, HTML et CSS et permettre ainsi une grande richesse de mise en forme ainsi que la possibilité d'ajouter un sommaire qui facilitera le parcours du document.

1.1.3 Affichage de contenu interactif

Enfin, le notebook offre la possibilité d'afficher tout type de contenu créé directement après le code : des tableaux, des graphiques interactifs ou pas, des images, etc. Afin de comparer, voici le même code dans un support classique et dans un notebook :

File Edit Format Run Options Window Help

```
# Import des librairies
import matplotlib.pyplot as plt

# Création des listes
a = [x for x in range(-10,11)]
b = [x**2 for x in a]

# Affichage du graphique
plt.scatter(a,b)
plt.title("x²")
plt.show()
```

Import des librairies

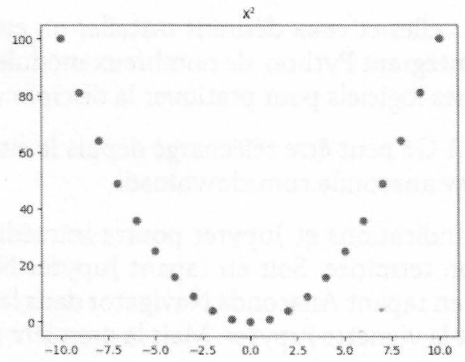
Entrée [2]:
```
import matplotlib.pyplot as plt
```

Création des listes

Entrée [3]:
```
a = [x for x in range(-10,11)]
b = [x**2 for x in a]
```

Affichage du graphique

Entrée [4]:
```
plt.scatter(a,b)
plt.title("x²")
plt.show()
```



Nous constatons en outre que le graphique est généré et reste présent, ce qui n'est pas le cas pour le programme classique.

Il est temps maintenant de voir ensemble les différentes façons d'accéder à un notebook.

1.2 Comment créer un notebook

1.2.1 Installation directe du module Jupyter

Si Python est déjà installé, la méthode la plus simple consiste à déployer directement le module Jupyter. Voici le code à entrer sur une invite de commande :

```
pip install jupyter
```

Une fois le module déployé, lancez la commande :

```
jupyter notebook
```

1.2.2 Installation de la suite Anaconda

Anaconda est la solution pour celles et ceux désirant installer un environnement complet de data science intégrant Python, de nombreux modules dédiés, ainsi que Jupyter et bien d'autres logiciels pour pratiquer la discipline.

Le package d'un peu moins de 1 Go peut être téléchargé depuis le site officiel dont voici le lien : https://www.anaconda.com/download.

Ensuite, il suffit de suivre les indications et Jupyter pourra immédiatement être lancé une fois l'installation terminée. Soit en tapant Jupyter Notebook dans la barre de recherche, soit en tapant Anaconda Navigator dans la barre de recherche puis en sélectionnant la vignette Jupyter. Mais la première méthode reste la plus rapide.

Remarque

Anaconda offre également la possibilité d'utiliser JupyterLab qui étend les fonctionnalités de Jupyter Notebook en offrant une interface utilisateur plus modulaire et avancée, permettant une expérience de développement plus riche et flexible.

1.2.3 Google Colaboratory

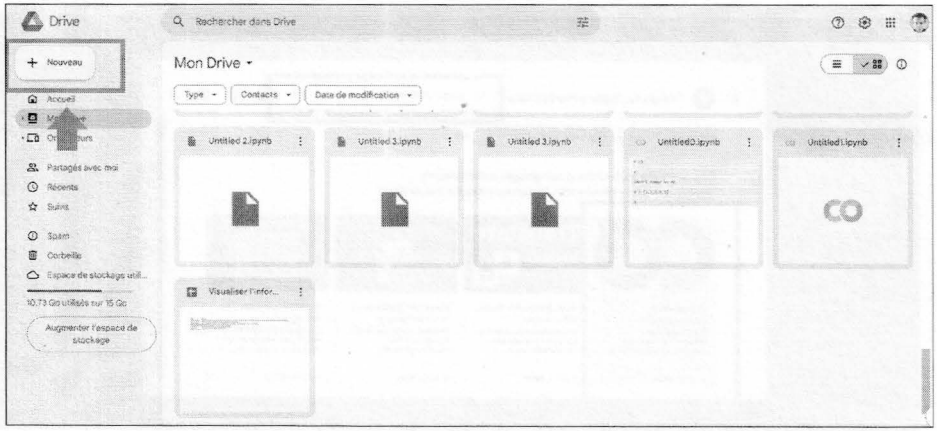
Google Colaboratory est une solution très séduisante pour créer des notebooks mais il faut au préalable être propriétaire d'un compte Google. Passée cette formalité, voici les avantages qui nous attendent :

- des notebooks basés sur une version récente de Python (3.10) pourvue de nombreux packages utiles ;
- des notebooks qui peuvent tourner plus rapidement grâce aux serveurs Google ;
- la possibilité offerte d'utiliser la mémoire GPU ou TPU de manière simple et immédiate ;
- un sommaire qui se construit automatiquement à partir des titres.

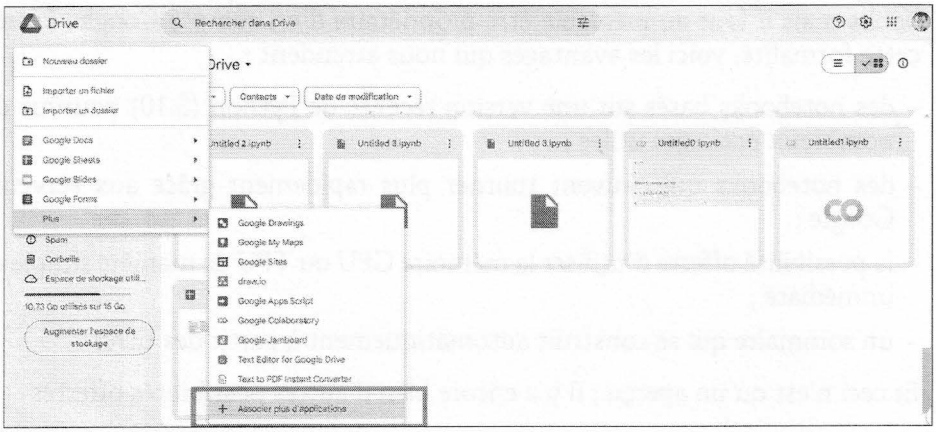
Et ceci n'est qu'un aperçu ; il y a encore bien d'autres possibilités offertes.

Voyons ensemble comment créer notre premier notebook Colab.

▶Rendez-vous sur le Drive Google et cliquez sur **+ Nouveau**.

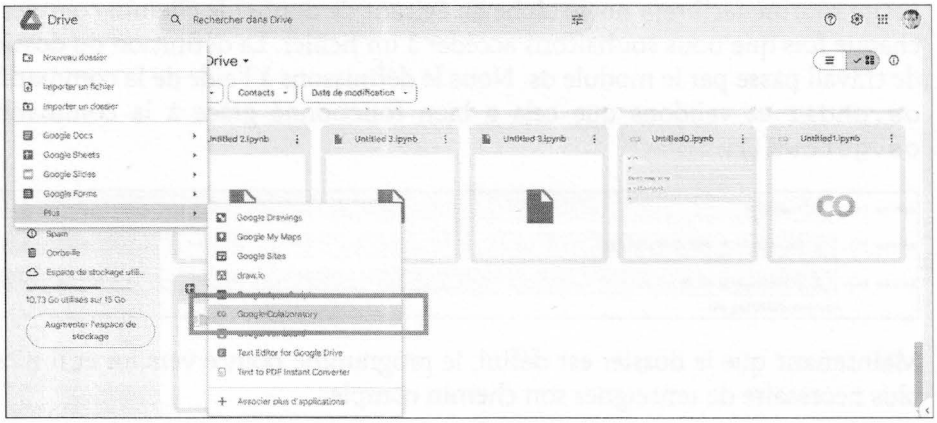


- Puis recherchez, en cliquant sur **Plus**, la mention **Associer plus d'applications** tout en bas de la liste :



En tapant « google colaboratory », le module est proposé, vous permettant de procéder à l'installation :

Une fois cela terminé, vous avez la possibilité de créer un notebook de cette façon :



Nous avons à présent la capacité de créer un notebook. Dès lors, nous allons en profiter pour rappeler succinctement les notions de base de Python utiles à notre domaine d'étude.

2. Commandes de base

Nous allons ici revisiter certains concepts fondamentaux de Python qui revêtent une importance cruciale dans le cadre de l'utilisation de Python en data science.

2.1 Acquisition des données

Avant de débuter toute analyse de données, il est primordial de les acquérir. Voici donc les mesures à prendre pour mener à bien cette première étape.

2.1.1 Définition du dossier de travail

Commençons déjà par définir le dossier sur lequel nous travaillerons. Cette petite routine facilitera notre tâche en évitant de donner le chemin complet à chaque fois que nous souhaitons accéder à un fichier. La définition du dossier de travail passe par le module os. Nous le définissons à l'aide de la commande `os.chdir` et validons que cela a bien fonctionné grâce à la commande `os.getcwd()`.

```
Entrée [4]:  1 import os

Entrée [5]:  1 os.chdir(r"D:\ENI\CHAPITRE_01")

Entrée [6]:  1 print(os.getcwd())
             D:\ENI\CHAPITRE_01
```

Maintenant que le dossier est défini, le programme pointe vers lui et il n'est plus nécessaire de renseigner son chemin complet.

Remarque

Tous les exemples de fichiers ont été stockés dans le sous-dossier DATA donc le chemin commencera toujours par DATA/nom_fichier.type.fichier.

2.1.2 Accès aux données

L'accès aux données peut s'effectuer de manière assez simple. Voici quelques exemples pour lire les formats de données les plus courants, en commençant par le format CSV, qui est un format texte avec un séparateur. Il est essentiel de connaître le séparateur pour que le fichier soit correctement lu. Par défaut, la virgule est utilisée pour les fichiers américains tandis que le point-virgule est utilisé pour les fichiers français. Pour faciliter ces opérations, nous utiliserons la bibliothèque Pandas, que nous étudierons en détail dans le chapitre Préparer les données avec Pandas et Numpy.

Lire des fichiers CSV

```
import pandas as pd
my_comma_csv = pd.read_csv("DATA/csv_anglais.csv")
my_semicolon_csv = pd.read_csv("DATA/csv_fr.csv",sep=";")
```

Lire des fichiers Excel

La lecture des fichiers Excel nécessite au préalable l'installation du module xlrd. Voici comment l'installer (à effectuer depuis l'invite de commande) :

```
pip install xlrd
```

Ensuite, nous pourrons lire les deux types d'extension de fichiers Excel grâce à la commande `read_excel` de pandas :

```
fic_xls =
pd.read_excel("DATA/File_xls.xls",sheet_name="nom_onglet")
```

```
fic_xlsx =
pd.read_excel("DATA/File_xlsx.xlsx",sheet_name="nom_onglet")
```

Lire des fichiers JSON

Le format JSON, abréviation de *JavaScript Object Notation*, tire ses origines du langage de programmation JavaScript. Il a été développé pour représenter des données de manière légère et lisible, tout en restant facilement interprétable par les ordinateurs. Nous aurons recours à la commande `read_json` de pandas pour ce faire :

```
fic_json = pd.read_json('DATA/iris.json')
```

Lire des fichiers SQL (exemple avec Sqlite3)

Nous sommes parfois amenés à traiter des fichiers SQL (*Structured Query Language* - langage de requête structurée) contenant des données stockées dans une base de données relationnelles. L'acquisition y est un peu moins aisée que pour les autres types vus précédemment et nécessite le recours au module sqlite3 :

```
import sqlite3
import pandas as pd

# Chemin vers le fichier de base de données SQLite
db_file_path = "DATA/my_db.db"

# Création d'une connexion à la base de données
conn = sqlite3.connect(db_file_path)

# Lecture de données à partir de la base de données en utilisant
```

```
# une requête SQL. Bien penser à remplacer table_name par
# le nom de votre table
query = "SELECT * FROM table_name"
df = pd.read_sql_query(query, conn)


# Fermeture de la connexion à la base de données
conn.close()
```

Lire un fichier texte

Enfin, dans le cas où vos données proviennent d'un simple fichier texte, voici le moyen d'acquérir son contenu :

```
with open("DATA/fichier_txt.txt","r")as file:
    contenu = file.read()
```

Notons ici que le contenu des fichiers bruts de ce type n'est pas organisé et nécessitera une remise en forme particulière.

2.2 Définition des données

Chaque variable a un type : numérique, décimal, texte, date, etc. Cette attribution se fait de manière automatique lors de la lecture des données par le programme mais n'est pas exempte d'erreurs. Il est courant de se rendre compte que le type attribué n'est pas le bon.

2.2.1 Changement du type

Le cas le plus courant concerne une variable numérique définie en texte. Dans ce cas, aucun calcul ne sera possible tant que le type ne sera pas numérique. Voici un simple exemple pour s'en convaincre :

```
a ="50"
b = "40"

print(a+b) # Ici l'addition va juste assembler les variables
# Output: 5040

print(int(a) + int(b)) # Ici, la conversion permet le calcul
# Output: 90
```

Ces erreurs de typage sont en général simples à résoudre hormis pour les dates dont la définition correcte peut s'avérer fastidieuse suivant la façon dont elles ont été écrites.

2.2.2 Gestion des dates

Les dates peuvent s'écrire de manière complètement différente tant dans la forme (format anglo-saxon ou français) que dans le détail (uniquement jour/mois/année, heures/minutes/secondes en plus, mois écrit en lettres, etc.). Il est souvent nécessaire d'indiquer précisément à Python la façon dont il convient de les lire.

Voici un exemple montrant comment gérer différents cas de figure en utilisant le module datetime :

```
from datetime import datetime

# Dates écrites de différentes façons
date_string_1 = "2023-05-15"
date_string_2 = "15/05/2023"
date_string_3 = "2023-05-15 12:30:45"

# Conversion des dates en objets datetime
date_1 = datetime.strptime(date_string_1, "%Y-%m-%d")
date_2 = datetime.strptime(date_string_2, "%d/%m/%Y")
date_3 = datetime.strptime(date_string_3, "%Y-%m-%d %H:%M:%S")

# Affichage des dates converties

print("Date 1:", date_1)
print("Date 2:", date_2)
print("Date 3:", date_3)
```

```
Date 1: 2023-05-15 00:00:00
Date 2: 2023-05-15 00:00:00
Date 3: 2023-05-15 12:30:45
```

Pour les dates avec le mois en texte, il est souvent nécessaire d'importer les paramètres locaux pour une gestion correcte :

```
from datetime import datetime
import locale

# Définition des paramètres locaux en français
locale.setlocale(locale.LC_TIME, "fr_FR.UTF-8")

# Date dont le mois est écrit en français
date_string_4 = "15 Mai 2023"

# Conversion de la date en objet datetime
date_4 = datetime.strptime(date_string_4, "%d %B %Y")

# Affichage de la date convertie
print("Date 4:", date_4)

# Output: Date 4: 2023-05-15 00:00:00
```

Comme pour les nombres, l'affectation correcte du type est un prérequis pour ensuite mener nos analyses. Maintenant que le type est correctement défini, un attribut qui passe plus inaperçu dans un premier temps va prendre une importance considérable : la taille du stockage.

2.2.3 Taille du stockage par type

Lors de l'acquisition des données, chaque variable se voit attribuer un type ainsi qu'une longueur en bits. Cette définition se fait souvent en prenant la longueur maximale de bits, soit 64 bits, ce qui n'est pas toujours pertinent par rapport au contenu de la variable. Prenons l'exemple d'une variable définissant l'âge d'une personne. En lui attribuant un type int64 (integer 64bist), nous pourrons potentiellement stocker un nombre compris entre -9223372036854775808 et 9223372036854775807. C'est un peu démesuré mais c'est pourtant ce qui sera défini par défaut.

Le problème, c'est que ce surcroît de poids va peser sur les performances car des bases de données plus lourdes vont prendre plus de temps à manipuler. Ceci pourrait même aboutir à un dépassement de mémoire qui ferait planter le programme. À titre illustratif, voici la différence constatée entre une base de données avant et après optimisation des types :

Avant optimisation

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    int64
 4   NX       506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    int64
 9   TAX      506 non-null    float64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    506 non-null    float64
 13  MEDV     506 non-null    float64
dtypes: float64(12), int64(2)
memory usage: 55.5 KB
```

Après optimisation

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   CRIM     506 non-null    float32
 1   ZN       506 non-null    int8
 2   INDUS    506 non-null    float32
 3   CHAS     506 non-null    bool
 4   NX       506 non-null    float32
 5   RM       506 non-null    float32
 6   AGE      506 non-null    float32
 7   DIS      506 non-null    float32
 8   RAD      506 non-null    int8
 9   TAX      506 non-null    int16
 10  PTRATIO  506 non-null    float32
 11  B        506 non-null    float32
 12  LSTAT    506 non-null    float32
 13  MEDV     506 non-null    int8
dtypes: bool(1), float32(9), int16(1), int8(3)
memory usage: 20.9 KB
```

L'usage en mémoire est ainsi passé de 55.5 kb à 20.9 kb, soit une réduction de 62 %, ce qui est un gain significatif.

De la même manière que la gestion des types dans une base de données optimise l'utilisation de la mémoire et accélère les calculs, une attention particulière aux structures de code et à leur organisation peut également jouer un rôle crucial dans l'efficacité globale du programme.

2.3 Structuration du code

Les structures de code claires et bien organisées garantissent une maintenance aisée, des performances optimales tout en facilitant la collaboration entre développeurs. C'est dans cette perspective qu'est né le Python Enhancement Proposal 8, dit le PEP8.

2.3.1 PEP8

Le PEP8 est un ensemble de règles permettant de bien coder en Python. Pour ce faire, il fournit un cadre de bonnes pratiques qui s'articulent autour de deux axes :

- En premier lieu, il faut organiser le code en fournissant tout un ensemble de règles destinées à le rendre plus lisible comme :
  - hiérarchisation des imports en commençant par les modules standards, puis les modules tiers et enfin les locaux,
  - indentation fixée à 4 espaces,
  - utilisation d'espaces autour des affectations et des opérateurs,
  - utilisation d'une ligne vide pour séparer les fonctions et les classes,
  - utilisation de plusieurs lignes vides consécutives à éviter,
  - espaces à l'intérieur des parenthèses, crochets et accolades à éviter ; seuls les espaces après les virgules d'une liste sont conseillés,
  - limitation de la longueur des lignes à 79 caractères pour le code et 72 pour les commentaires.
- En second lieu, il s'agit de faciliter la transmission du code entre les programmeurs. En ce sens, il est conseillé de :
  - utiliser des noms significatifs et explicites,
  - recourir à la convention snake_case pour les noms de variables, fonctions et méthodes consistant à utiliser des mots séparés par des underscores,
  - utiliser la convention CamelCase pour les noms de classes consistant à assembler des mots en mettant une majuscule au début de chacun,
  - commenter judicieusement le code avec des phrases complètes et en anglais,

- utiliser des docstrings servant à commenter le fonctionnement des modules, fonctions, classes et méthodes.

L'adoption du PEP8 est vivement encouragée. Elle apportera un vrai plus dans la lisibilité de notre code. Notons ici la possibilité de recourir à certains modules de Python comme black ou yapf pour reformater automatiquement le code. Voyons comment les mettre en œuvre :

- black :
  - Installation du module :

```
pip install black
```

  - Formater un fichier sans appliquer les modifications :

```
black -diff chemin_du_fichier/nom_fichier.py
```

  - Formater directement un fichier :

```
black chemin_du_fichier/nom_fichier.py
```

  - Formater directement tous les fichiers Python d'un dossier :

```
black chemin_des_fichiers/
```

- yapf :
  - Installation du module :

```
pip install yapf
```

  - Formater un fichier sans appliquer les modifications :

```
yapf chemin_du_fichier/nom_fichier.py
```

  - Formater directement un fichier :

```
yapf -i chemin_du_fichier/nom_du_fichier.py
```

En dehors des modules Python, il existe également des extensions sur les IDE comme PyCharm ou Visual Studio Code pour répondre à ce besoin.

Le PEP8 est un très bon début dans la recherche de clarification du code mais nous pouvons aller encore plus loin.

2.3.2 Optimisation du code

Nous allons explorer deux façons d'écrire le même algorithme pour illustrer la pertinence de l'optimisation du code. Supposons que nous devions écrire une fonction renvoyant le carré de chaque membre d'une liste. Voici pour commencer une façon basique de rédiger la fonction :

```
def square_calulation(liste):
    # Calculer le carré de chaque élément de la liste
    squares = []
    for i in range(len(liste)):
        squares.append(liste[i] * liste[i])

# Renvoyer le résultat
    return squares

my_list = [1, 2, 3, 4, 5]

square_calulation(my_list)

# Output: [1, 4, 9, 16, 25]
```

La fonction produit le résultat escompté mais voyons sa version optimisée en recourant à des compréhensions de listes, une technique pour créer des listes de manière concise :

```
def square_calulation_v2(liste):
    # Compréhension de liste pour calculer le carré
    Squares = [x ** 2 for x in liste]
    # Renvoi du résultat
    return squares
```

Ici la fonction a gagné en clarté, ce qui facilitera sa compréhension et la maintenance ultérieure du code. Voici un autre exemple probant d'optimisation concernant la création d'un dictionnaire en utilisant la fonction `zip`, qui associe des éléments de deux listes :

```
keys = ['nom', 'âge', 'ville']
values = ['Bob', 25, 'Paris']

# Méthode classique
my_dictionary = {}
for i in range(len(keys)):
    my_dictionary[keys[i]] = values[i]
```

```
# Méthode plus astucieuse
my_dictionary = {k: v for k, v in zip(keys, values)}
```

Remarque

Le site Codewars (https://www.codewars.com/) est un très bon moyen de faire évoluer son code. Il offre une plateforme interactive pour résoudre une variété de problèmes et se distingue par son système de classement des solutions permettant aux utilisateurs d'accéder aux meilleures approches et de perfectionner leurs compétences en programmation.

3. Utilisation avancée

Pratiquer la data science nécessite la maîtrise de certaines fonctionnalités plus avancées de Python que nous allons nous attacher à définir.

L'import des différentes librairies nécessaires à la réalisation de notre programme arrive en premier dans notre code. Il est important de commencer par voir comment les manipuler.

3.1 Gestion des librairies

Les librairies sont des ensembles de fonctions Python facilitant la gestion de problèmes complexes. Elles allègent considérablement l'écriture du code et nous permettent de produire des programmes que nous ne pourrions pas mettre en œuvre sans leur concours. Avant de voir leur fonctionnement, soulignons qu'une librairie évolue au cours du temps, qu'elle fait souvent appel à d'autres librairies qui évoluent aussi et il n'est pas rare de rencontrer des problèmes de compatibilité entre elles. C'est pourquoi savoir comprendre et maintenir cet écosystème est une étape peu compliquée mais obligatoire pour pouvoir mener à bien notre tâche.

Remarque

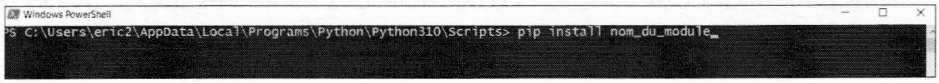
Les termes « bibliothèque », « librairie » et « module » seront utilisés de manière interchangeable pour désigner les librairies.

3.1.1 Installation

L'installation d'un module est une opération très courante. Il suffit d'ouvrir une invite de commande, quelle que soit la plateforme, et de taper (attention à bien mettre *pip* en minuscules) :

```
pip install nom_du_module
```

Voici une illustration en passant par Windows PowerShell sous Windows :



Il convient de préciser ici que sous Windows, la commande doit être lancée dans le sous-dossier Scripts du dossier du programme Python. Nous constaterons, pour nous en assurer, qu'il existe un fichier nommé pip.exe.

3.1.2 Mise à jour

Comme les librairies évoluent, il peut être nécessaire de les mettre à jour. Connaître la version d'une librairie peut se faire de deux façons :

– Soit en les affichant la version de toutes les librairies présentes :

```
pip freeze
```

– Soit en demandant les détails d'une librairie en particulier via la commande suivante :

```
pip show <nom_du_module>
```

Une fois la version connue, nous pouvons procéder à la mise à jour du package :

```
pip install --upgrade <nom_du_module>
```

3.1.3 Suppression

Une librairie peut parfaitement être supprimée si elle ne sert plus :

```
pip uninstall <nom_du_module>
```

Attention toutefois de bien s'assurer qu'elle n'est pas nécessaire à l'exécution d'une autre. Mais il suffira de l'installer à nouveau si tel est le cas.

Dans certaines situations, notamment pour celles et ceux qui ont installé énormément de librairies, le déploiement d'une nouvelle librairie peut s'avérer impossible à cause de conflits entre certains packages. Plutôt que de perdre du temps en recherches chronophages, nous pouvons créer un environnement virtuel.

3.2 L'environnement virtuel

3.2.1 Déploiement d'un environnement virtuel

L'environnement virtuel est l'assurance d'avoir un environnement Python n'utilisant que les librairies nécessaires à son bon fonctionnement évitant ainsi des problèmes d'incompatibilité et facilitant les mises à jour.

Au départ, un environnement virtuel ne contient que le programme Python sans aucun package. Ce sera à nous de les installer. Nous pouvons créer autant d'environnements que nous voulons au gré de nos besoins. Voyons ensemble les étapes pour ce faire.

L'idéal est de définir un dossier qui contiendra tous les environnements virtuels que nous créerons. Pour l'exemple, ce dossier sera nommé « Environnements virtuels ».

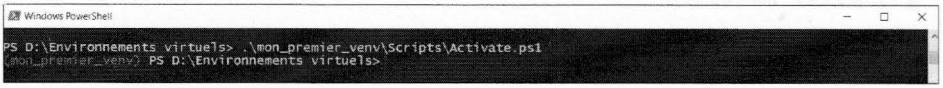
Ensuite, ouvrons une invite de commandes à l'intérieur de ce dossier et lançons la commande :

```
python -m venv mon_premier_venv
```

Après validation, l'environnement va être créé en quelques secondes. Il faut maintenant l'activer :

```
mon_premier_venv/Scripts/Activate.ps1
```

Si tout se passe bien, nous verrons apparaître le nom de l'environnement créé comme dans cette capture d'écran :



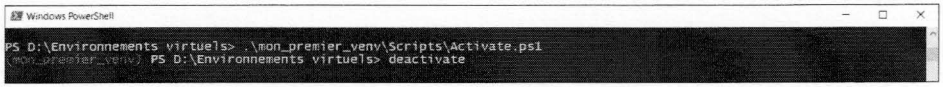
À ce stade, il est possible d'ajouter toutes les librairies nécessaires.

Remarque

Attention, si votre version de Python est antérieure à 3.6, il est nécessaire d'acquérir le fichier Python get-pip.py à l'adresse suivante : https://bootstrap.pypa.io/get-pip.py, de l'enregistrer et de l'exécuter dans votre invite de commande pour pouvoir bénéficier de pip.

Pour finir, lorsqu'il n'y en a plus besoin, l'environnement peut être désactivé de la façon suivante :

```
deactivate
```



3.2.2 Utilisation d'un environnement virtuel dans un notebook

Si nous souhaitons maintenant utiliser cet environnement virtuel dans un notebook, voici la marche à suivre.

Commençons par installer Jupyter :

```
python -m pip install jupyter
```

Il faut ensuite installer le module ipykernel permettant de déployer le noyau pour le langage Python dans Jupyter :

```
python -m ipykernel install --user --name=my_first_venv
```

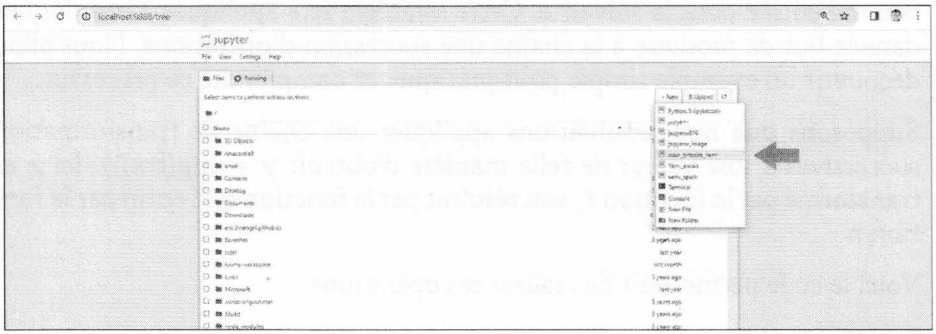
Remarque

Attention à bien mettre deux tirets du 6 à `user` et `name` et bien mettre le nom de l'environnement créé préalablement. Dans notre exemple, il se nommait my_first_venv.

Une fois ces commandes exécutées, lançons simplement Jupyter de cette façon :

```
jupyter notebook
```

Après quelques secondes, le navigateur va apparaître et nous pourrons alors créer un nouveau notebook basé sur notre environnement virtuel en le sélectionnant de la sorte :



Notons pour finir sur ce point qu'il est également possible d'installer directement un module depuis un notebook. Il faut se placer dans une cellule et entrer la commande :

```
!pip install nom_du_module
```

Après avoir étudié la gestion des bibliothèques et des environnements virtuels, il est temps de se familiariser avec les algorithmes qu'il est essentiel de comprendre sur le plan théorique avant de les appliquer dans le domaine de la science des données.

3.3 Les notions utiles pour la data science

Dans le vaste domaine de l'analyse et de la modélisation des données, la compréhension des différents algorithmes et méthodes est essentielle pour bien les mettre en pratique. Débutons cette exploration par le pipeline qui est couramment utilisé dans la modélisation.

3.3.1 Le pipeline

Le pipeline est une séquence ordonnée de fonctions ou d'opérations appliquées successivement à des données où le résultat de chaque étape est utilisé comme entrée pour la suivante. Cette stratégie sera appliquée à nos données dans le but de produire à la chaîne une succession d'opérations. Nous allons découvrir un exemple simple pour pratiquer et comprendre ce processus.

Supposons que nous souhaitions appliquer une chaîne de transformations successives à une valeur de telle manière d'obtenir y = h(g(f(x))). Ici x est transformé par la fonction f, son résultat par la fonction g et enfin par la fonction h.

Voici le code permettant de réaliser ces opérations :

```
pipeline = [
    lambda x: x ** 2 + 5,        # fonction f
    lambda x: x ** 3 + 3,        # fonction g
    lambda x: x ** 0.5 + 4       # fonction h (racine carrée : x doit
être positif)
]

result = 3
for f in pipeline:
    result = f(result)
    print(result)
```

```
14
2747
56.41183072551463
```

Cette approche nous permettra de créer une séquence d'opérations reproductibles où nous n'aurons qu'à ajuster la valeur d'entrée. Dans le cadre de la manipulation de données, un exemple de pipeline consisterait à récupérer les données, les diviser en un jeu d'entraînement et de test, et d'appliquer un algorithme spécifique dessus. Nous n'aurions plus ensuite qu'à changer l'algorithme décisionnel car grâce au pipeline toutes les autres opérations seraient automatiquement appliquées.

Le déploiement des algorithmes se fait justement en s'appuyant sur de la programmation orientée objet (POO) dont il est important, sans trop entrer dans les détails, d'en décrire le fonctionnement.

3.3.2 La programmation orientée objet (POO)

L'analogie avec l'automobile est une bonne façon de comprendre la programmation orientée objet. Supposons que nous ayons un moule permettant de créer une voiture. Ce moule correspond à la classe pour la POO et le fait de créer une voiture serait décrit comme une instance. Par ailleurs, la voiture en question possède des caractéristiques (sa couleur, sa forme, etc.) nommées attributs et des comportements (avancer, reculer, etc.) appelés méthodes. Ainsi, nous allons pouvoir créer autant de voitures que nécessaire, qui posséderont naturellement les attributs et méthodes liés à leur classe. Rien ne nous empêchera ensuite d'ajouter ou de retirer des attributs et des méthodes mais nous aurons la certitude de partir à chaque fois d'une voiture fonctionnelle.

Traduisons en code l'exemple utilisé :

```
# Définition de la classe Voiture
class car:
    def __init__(self, defined_color, car_type):
        self.defined_color = defined_color
        self.car_type = car_type

    def move_forward(self):
        print("La voiture avance.")

    def move_back(self):
        print("La voiture recule.")

# Création d'une instance de la classe Voiture
my_car = car(defined_color ="rouge", car_type ="berline")

# Accès aux attributs et appel des méthodes
print("Couleur de la voiture :", my_car.defined_color)
print("Forme de la voiture :", my_car.car_type)

# Accès aux méthodes
my_car.move_forward()
my_car.move_back()
```

Cette connaissance du fonctionnement de la POO est importante pour bien pratiquer, par exemple, les algorithmes du module Scikit-Learn dont le fonctionnement est basé dessus. Il sera alors plus simple de les prendre en main et d'interagir avec eux.

Passons maintenant aux décorateurs qui sont un peu moins connus du grand public mais qui vont s'avérer extrêmement utiles dans nos programmations.

3.3.3 Les décorateurs

Les décorateurs sont des fonctions prenant une autre fonction en argument qu'elles vont doter d'autres fonctionnalités sans la modifier. Cette technique est très pratique si nous souhaitons, par exemple, mesurer le temps d'exécution ou vérifier que les prérequis nécessaires à la bonne exécution d'une fonction sont bien respectés. Nous allons d'ailleurs fournir le code de ces deux exemples pour en mesurer l'intérêt.

Commençons par le chronométrage d'une fonction :

```
import time

# Décorateur de chronométrage de fonction
def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"La fonction {func.__name__} a mis
{end_time - start_time:.5f} secondes à s'exécuter.")
        return result
    return wrapper

# Fonctions à décorer
@timer_decorator
def slow_function():
    time.sleep(3)
    print("Fonction lente exécutée")

@timer_decorator
def fast_function():
    print("Fonction rapide exécutée")

# Appel des fonctions décorées
slow_function()
fast_function()
```

La fonction `timer_decorator` va s'avérer très utile dès lors que nous aurons besoin de mesurer le temps d'exécution d'une fonction. Il suffira de la disposer juste avant, précédée d'un `@` et le temps de calcul nous sera indiqué.

Dans un autre registre, explorons maintenant le code permettant de contrôler que les arguments fournis sont corrects. Ceci est une précaution bien utile pour s'assurer que nous n'aurons pas de bugs ou de résultats erronés :

```
# Décorateur pour vérifier les arguments d'une fonction
def validate_arguments(func):
    def wrapper(*args, **kwargs):
        # Vérification si tous les arguments sont positifs
        if all(isinstance(arg, (int, float)) and arg > 0 for
arg in args):
            print("Les types des arguments sont corrects.")
            return func(*args, **kwargs)
        else:
            print("Les types des arguments sont incorrects.")
            #raise ValueError("Tous les arguments doivent être
des nombres positifs.")
    return wrapper

# Fonction à décorer
@validate_arguments
def calculate_rectangle_area(length, width):
    return length * width

# Utilisation de la fonction décorée
area = calculate_rectangle_area(5, 4)
#area = calculate_rectangle_area("5", 4)
print("Aire du rectangle :", area)
```

Ici, nous avons deux définitions de la variable `area`. Nous constaterons que la première version renverra bien le message que tous les arguments sont corrects, au contraire de la seconde version dont le premier argument, « 5 », est une chaîne de caractères.

Toutes ces techniques de programmation participent à fiabiliser le code et éviter toute erreur qui pourrait remettre en cause toute la solution décisionnelle mise en place. En dépit de toutes les précautions, certaines erreurs peuvent persister. Il est opportun dans ce cas de prévoir une gestion des erreurs pour gérer les cas litigieux.

3.3.4 La gestion des erreurs

Gérer les erreurs participe à rendre le code plus robuste et éviter de perdre du temps à gérer les plantages. Les décorateurs, vus précédemment, peuvent jouer ce rôle. Nous avons également deux techniques utiles qu'il convient d'aborder : with et try-except.

Utiliser with

La technique with en Python est utilisée pour garantir une gestion propre des ressources en automatisant les actions de démarrage et de fermeture, telles que l'ouverture et la fermeture de fichiers, assurant ainsi leur clôture même en cas d'exception. Cette pratique est couramment utilisée pour l'ouverture d'un fichier comme ici :

```
# Ouverture du fichier en mode lecture avec la méthode "with"
with open("exemple.txt", "r") as f:
    # Lit le contenu du fichier et l'assigne à la variable "data"
    data = f.read()
    # Affiche le contenu lu
    print(data)

# À la fin du bloc "with", le fichier est automatiquement fermé,
même en cas d'exception.
```

Cette précaution est souvent présentée pour des fichiers TXT mais nous pouvons aussi l'utiliser pour des fichiers CSV, XLSX, etc.

Utiliser try-except

La seconde technique, try-except, va être plus largement utilisée. Elle consiste à indiquer au programme comment gérer les exceptions dans le code. De cette façon, nous initions une commande précédée de `try` et indiquons ensuite comment gérer l'exception en le précédent par `except`. La structure est la suivante :

```
try:
    # Code à essayer
except SomeException:
    # Code à exécuter en cas d'exception
```

Deux mots-clés supplémentaires peuvent suivre la routine except :

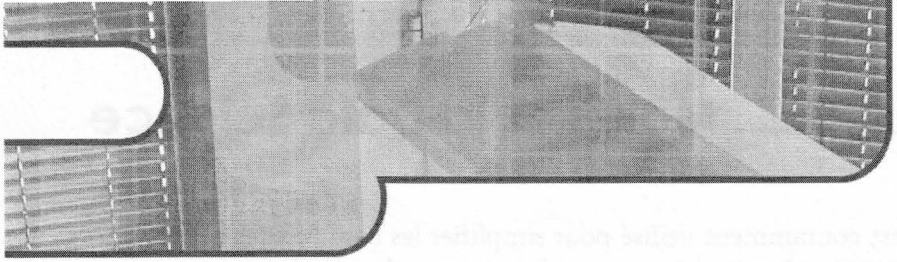
- `else` : si aucune exception n'a pu être levée. Cette option est facultative et doit être utilisée si le besoin s'en fait sentir.

```
try:
    # Opération pouvant générer une exception
    result = 10 / x
except ZeroDivisionError:
    # Code exécuté si une division par zéro est tentée
    print("Une division par zéro a été tentée !")
else:
  print("La division a pu être réalisée")
  print(result)
```

- `finally` : il sera exécuté indépendamment de l'exception. Cette commande trouve son utilité lorsque nous souhaitons exécuter certaines actions comme fermer le fichier par exemple pour libérer des ressources.

```
try:
    # Code à essayer
except SomeException:
    # Code à exécuter en cas d'exception
finally :
    # Code à exécuter quelle que soit l'issue de l'exception
```

Après ce rappel des fondamentaux de Python, nous sommes désormais prêts à mettre en pratique la data science de manière concrète.



Chapitre 3
Préparer les données avec Pandas et Numpy

1. Pandas, la bibliothèque Python incontournable pour manipuler les données

Nous sommes désormais prêts à explorer et analyser nos données. Pour ce faire, nous nous appuierons sur l'un des modules Python les plus essentiels : Pandas.

Pandas est la bibliothèque de Python permettant de manipuler et analyser les données. Elle a été créée en 2008 par Wes McKinney, un statisticien et développeur Python. Cette bibliothèque s'est progressivement imposée comme un élément incontournable pour gérer des données et a contribué à faire de Python une référence en la matière.

1.1 Installation

Pour commencer, assurons-nous qu'elle est bien installée. Si tel n'est pas le cas, validons la ligne suivante dans une invite de commande :

```
pip install pandas
```

Une fois installée, il suffit de l'importer :

```
import pandas as pd
```

L'alias `pd` est couramment utilisé pour simplifier les commandes liées à Pandas. Il permet ensuite de s'assurer que les commandes pandas que nous lançons s'y réfèrent bien.

1.2 Structure et type de données

Avant de l'utiliser, prenons le temps d'expliquer les différentes structures que peut prendre pandas. Celle que tout le monde a en tête est le DataFrame à deux dimensions se présentant comme un tableur Excel, mais il existe en réalité un type par dimension :

Nom de la structure	Nombre de dimensions	Principe
Séries	1	Données unidimensionnelles indexées
DataFrames	2	Feuille de calcul avec lignes et colonnes
Panels	3	Collection de DataFrames
DataFrames multi-index	4	DataFrames avec multi-index pour les lignes ou les colonnes

Remarque

Il existe aussi une structure `panel4D` de quatre dimensions mais elle est dépréciée et il est déconseillé de l'utiliser.

Dans l'immense majorité des cas, seuls les séries et les DataFrames seront rencontrés, mais il est utile de connaître l'existence des autres structures.

L'indexation est au cœur de ces structures, de manière beaucoup plus présente que dans les tableurs. Elle permet d'accéder facilement aux lignes, colonnes ou cases. Pandas propose d'ailleurs plusieurs façons en offrant la possibilité de s'y référer soit par leur indice soit par leur nom.

Petite précision utile : l'indexation commence à 0 et non à 1.

Le tableau suivant indique les quatre façons de faire :

Action	iloc	loc	at	query
Accès à une case	`df.iloc[2,1]`	`df.loc[2,'Col2']`	`df.at[2,'Col2']`	Inapproprié
Accès à une ligne	`df.iloc[2, :]`	`df.loc[2, :]`	Inapproprié	`df.query ("index==2")`
Accès à une colonne	`df.iloc[ :,1]`	`df.loc[ :, 'Col2']`	Inapproprié	Inapproprié
Filtrage avec condition	`masque = df['A'] > 2` `df.iloc[masque .values, 1]`	`masque = df['A'] > 2` `df.loc[masque .values, "B"]`	Inapproprié	`df.query ("A>2")`

■Remarque

La grande différence entre `iloc` et `loc`, qui sont les fonctions les plus utilisées, réside dans le fait que tout est numérique avec `iloc` alors que `loc` oblige à préciser le nom de la variable.

1.3 Possibilités offertes

Passé le côté structurel, regardons ensemble ce qui a fait le succès de Pandas : sa capacité à répondre à tous les besoins inhérents aux manipulations de données.

Dès le départ, Pandas offre l'assurance de pouvoir lire quasiment tous les types de fichiers. Outre les formes les plus classiques comme les fichiers texte, CSV, Excel ou JSON, nous est aussi offerte la possibilité de lire du SQL, des fichiers propriétaires comme SAS ou SPSS et d'autres formats que nous rencontrerons plus tard comme HDF5, Parquet ou Pickle.

Une fois les données acquises, nous pouvons accéder simplement à différentes informations comme les dimensions, le type des variables, le nombre d'observations non nulles par colonne ou des statistiques descriptives de base.

Pandas nous offre ensuite toute la panoplie de fonctions pour préparer les données : supprimer des observations ou des colonnes, imputer, remplacer, fusionner d'autres fichiers ou créer de nouvelles variables.

À cela s'ajoutent des capacités de visualisations graphiques, très pratiques pour appréhender les données. Cependant, pour des besoins plus avancés, nous pourrions préférer les fonctionnalités offertes par des modules dédiés tels que Matplotlib ou Seaborn.

Cette interopérabilité avec les autres modules est d'ailleurs un des autres piliers qui fait la force de Pandas. En plus des bibliothèques graphiques mentionnées précédemment, Pandas interagit parfaitement avec tous les modules Python dédiés à la data science comme Scikit-Learn ou Numpy. Nous allons justement présenter ce dernier qui agit dans l'ombre de Pandas.

2. Numpy, le pilier du calcul numérique

Numpy est une bibliothèque fondamentale pour le calcul scientifique en Python agissant comme une infrastructure de base pour de nombreuses autres bibliothèques. Prenons le temps de voir ensemble ce qui fait sa force et pourquoi elle est omniprésente dans les modules Python.

2.1 La structure ndarray

Le ndarray, abréviation de N-dimensionnal array signifiant tableau à n dimensions, est vraiment au cœur de la bibliothèque Numpy. Deux autres structures, les matrix et scalars, sont également proposées mais nous polariserons notre attention sur le ndarray tant son rôle est central.

Remarque

Attention, quantité de noms différents peuvent renvoyer à un ndarray. Ainsi, le terme tableau est la forme générique pour les qualifier mais nous pouvons rencontrer le terme vecteur pour des ndarray à une dimension ou matrice pour ceux en comprenant deux. Au-delà de deux dimensions, il est courant de rencontrer les appellations : array, NDArray, tenseur ou tensor.

Commençons par étudier les caractéristiques de cette structure fondamentale de Numpy.

2.1.1 Une structure homogène

Avant tout, c'est un tableau multidimensionnel homogène, c'est-à-dire qu'il peut prendre autant de dimensions que souhaité avec la contrainte que toutes les données soient du même type. Voici un moyen simple de créer un vecteur ndarray à partir d'une simple liste Python :

```
import numpy as np
a = [1, 2, 3, 4, 5]
array_numpy = np.array(a)
print(array_numpy.dtype)

# Output: int64
```

Quant à l'obligation d'homogénéité mentionnée plus haut, notons que l'introduction d'un membre de type float entraîne de facto le changement de type de l'ensemble du ndarray :

```
array_numpy_2 = np.array([1.0, 2, 3, 4, 5])
print(array_numpy_2.dtype)

# Output: float64
```

Cet exemple n'avait pour but que de créer un ndarray à partir d'une structure familière. Maintenant que nous avons pris nos marques, voici la façon de le créer directement :

```
debut = 0
fin = 10 #valeur non incluse
pas = 2
array_numpy = np.arange(debut, fin, pas) # Le pas peut être décimal

print(array_numpy)
# Output: [0 2 4 6 8]
```

Et l'éventail des possibilités ne s'arrête pas là. La commande `linspace`, par exemple, permet de créer un vecteur avec un certain nombre de valeurs d'intervalles identiques, très pratique lors de la création de graphiques :

```
array_linspace = np.linspace(0,100,15)
```

Cette simple ligne de code va ainsi générer un vecteur entre 0 et 100 inclus possédant 15 valeurs en tout à intervalles égaux.

Voyons maintenant par quels moyens créer des matrices à deux dimensions.

Cela peut se faire directement depuis une table Pandas de la façon la plus simple possible :

```
# df est un DataFrame pandas
array_from_pandas = df.values
```

Ici, nous avons récupéré tout le DataFrame mais nous n'aurions pu obtenir qu'une seule variable sous forme de vecteur de cette façon :

```
array_col_pandas = df['nom_colonne'].values
```

Ou plus :

```
array_cols_pandas = df[['nom_colonne1', 'nom_colonne2',
'nom_colonne3']].values
```

Il nous est aussi offert la possibilité de créer une matrice en fixant les dimensions et le contenu. Comme une matrice nulle, remplie de zéros, à l'aide de la commande `zeros` :

```
nb_rows = 10
nb_columns = 10

shape = (nb_rows, nb_columns)
matrice_zeros = np.zeros(shape)
```

Les commandes `ones` et `full` procèdent de la même façon mais avec respectivement des 1 ou une valeur spécifique définie pour toutes les cases :

```
# Matrice remplie de 1
matrice_1 = np.ones(shape)

# Matrice remplie de 100 (Tous les nombres sont possibles) :
matrice_100 = np.full(shape, 100)
```

Enfin, signalons les différentes possibilités offertes pour générer des nombres aléatoires, qu'ils soient entiers ou décimaux.

Pour les nombres décimaux, nous pouvons demander une matrice de nombres uniformes entre 0 et 1 grâce à `random.rand` :

```
# Exemple de matrice de 3 lignes sur 4 colonnes :
matrice_rand = np.random.rand(3,4)
```

L'utilisation nous permettra d'obtenir des nombres gaussiens, c'est-à-dire qui ont une moyenne de 0 et un écart-type de 1 :

```
# Exemple de matrice de 3 lignes sur 4 colonnes :
matrice_randn = np.random.randn(3,4)
```

La création des nombres entiers va nécessiter une opération supplémentaire : l'usage du mot-clé `reshape` car il n'est pas possible de définir directement les dimensions. Nous indiquerons ainsi dans `reshape` le nombre de lignes et de colonnes souhaitées :

```
# Création d'une liste de nombres entiers aléatoires
# random.randint(low, high=None, size=None)
randint_numpy = np.random.randint(0,50,100)

# Tranformation en matrice de 10 lignes sur 10 colonnes
randint_numpy = randint_numpy.reshape(10,10)
```

Cette matrice aurait pu être créée directement en une ligne en chaînant les commandes :

```
randint_numpy = np.random.randint(0,50,100).reshape(10,10)
```

Remarque

Attention à ce que le produit des dimensions de la nouvelle forme produite avec `reshape` corresponde exactement à la longueur de la liste de nombres aléatoires, sinon nous obtiendrons une erreur.

```
array([[ 0, 39, 24, 36, 35,  5,  6,  3, 34, 40],
       [33, 28,  4, 26, 32, 45,  9,  5, 33,  7],
       [30,  8, 20,  7,  3, 21, 27, 44,  3, 38],
       [20,  7, 19, 31,  0,  5, 27, 43, 30,  9],
       [19,  7, 21, 37, 28,  8, 43, 46,  0, 40],
       [38, 25, 10, 34, 23, 32, 19, 26, 14, 32],
       [ 6, 33, 44, 45, 41,  4, 29, 27, 17, 35],
       [ 2, 20, 45, 15, 36, 41,  4, 49, 13, 30],
       [45, 23, 34, 35,  9, 26, 26, 22, 12, 15],
       [34, 26, 38, 46, 16, 47, 40,  0, 10, 11]])
```

Avant de clôturer ce point, signalons une possibilité commune à toutes les structures Numpy : la possibilité de définir le dtype qui va impacter les performances. Ainsi, plus la place allouée en bits va être élevée, plus grande sera la précision mais au détriment du temps de calcul et vice versa.

Regardons ensemble la conséquence du changement de type sur une matrice :

```
# Définition de la graine d'aléa pour retrouver le même résultat
np.random.seed(0)

m1 = np.random.randn(3,4)
m2 = m1.astype(np.float16) # Modification du type de 64 à 16 bits

print("m1:\n",m1)
print("m2:\n",m2)
```

```
m1:
 [[ 1.76405235  0.40015721  0.97873798  2.2408932 ]
 [ 1.86755799 -0.97727788  0.95008842 -0.15135721]
 [-0.10321885  0.4105985   0.14404357  1.45427351]]
m2:
 [[ 1.764   0.4001  0.9785  2.24  ]
 [ 1.867  -0.977   0.95    -0.1514]
 [-0.1032  0.4106  0.144   1.454 ]]
```

Après cette exploration des différentes structures, regardons comment accéder aux données.

2.1.2 L'indexation

À l'instar des listes Python, les éléments des tableaux sont indexés. L'accès y est ainsi facilité. Mais la comparaison avec les simples listes Python s'arrête là car les ndarray offrent beaucoup plus de vitesse et d'options.

Commençons par découvrir comment accéder à nos données.

L'accès à un élément spécifique, dans le cadre d'une matrice à deux dimensions, consiste simplement à fournir le numéro de ligne et de colonne :

```
print(m2[0,1]) # Affichage de l'élément ligne 0 / colonne 1

# Output: 0.4001
```

Si nous souhaitons accéder à une ligne ou une colonne en particulier, il faut simplement recourir aux « : » indiquant d'inclure tous les éléments concernés :

```
print(m2[0, :]) # Affichage de la première ligne (d'index 0)

# Output: array([1.764 , 0.4001, 0.9785, 2.24  ], dtype=float16)
```

```
print(m2[ :, 1]) # Affichage de la deuxième colonne (d'index 1)

# Output: array([ 0.4001, -0.977 ,  0.4106], dtype=float16)
```

Nous pouvons aussi utiliser les « : » comme dans le cadre des listes pour afficher plusieurs lignes ou colonnes comme ici :

```
print(m2[ :, 1:3]) # Affichage des colonnes d'index 1 et 2


# Output:
array([[ 0.4001,  0.9785],
       [-0.977 ,  0.95  ],
       [ 0.4106,  0.144 ]], dtype=float16)
```

Pour finir, l'accès à des colonnes ou des lignes discontinues se fait en passant par une liste comme ici :

```
print(m2[:, [0,3]] # Affichage des colonnes d'index 0 et 3

# Output:
array([[ 1.764 ,  2.24  ],
       [ 1.867 , -0.1514],
       [-0.1032,  1.454 ]], dtype=float16)
```

Nous invitons celles et ceux qui ne sont pas encore très familiers avec ce type d'indexation à bien prendre le temps de les pratiquer car Numpy va revenir constamment au cours des différentes étapes.

Profitons de ce point sur les indexations pour aborder le sujet des manipulations des tableaux que nous allons rencontrer fréquemment.

2.1.3 La modification des structures

Il est courant de devoir agir sur la structure des données. En effet, certaines étapes nécessitent d'assembler des tables, d'autres de remettre le vecteur à plat ou de jouer sur ses dimensions. Une mise au point rapide s'impose en commençant par ce qu'il vaut mieux éviter : la fusion.

La fusion, également appelée merging, consistant à combiner deux tables en utilisant une clé primaire d'assemblage, est mieux réalisée avec Pandas, qui offre toutes les fonctionnalités nécessaires à cet effet. Nous verrons plus loin comment accomplir cette opération dans les meilleures conditions.

En revanche, la concaténation, qui consiste à juxtaposer des tables verticalement ou horizontalement, est très simple à réaliser avec Numpy. Elle peut se faire pour chaque orientation de deux manières.

Voici les deux moyens de la réaliser de manière horizontale :

```
# Création de deux tableaux Numpy
array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([[7, 8], [9, 10]])

# Concaténation horizontale avec numpy.concatenate
concatenated_array_concat = np.concatenate((array1, array2), axis=1)

# Concaténation horizontale avec numpy.hstack
concatenated_array_hstack = np.hstack((array1, array2))

print("Résultat de la concaténation avec numpy.concatenate :")
print(concatenated_array_concat)
print("\nRésultat de la concaténation avec numpy.hstack :")
print(concatenated_array_hstack)
```

```
Résultat de la concaténation avec numpy.concatenate :
[[ 1  2  3  7  8]
 [ 4  5  6  9 10]]

Résultat de la concaténation avec numpy.hstack :
[[ 1  2  3  7  8]
 [ 4  5  6  9 10]]
```

Réciproquement, voici comment fusionner verticalement :

```
# Création de deux tableaux Numpy
array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([[7, 8, 9], [10, 11, 12]])

# Concaténation verticale avec numpy.concatenate
concatenated_array_concat = np.concatenate((array1, array2), axis=0)

# Concaténation verticale avec numpy.vstack
concatenated_array_vstack = np.vstack((array1, array2))

print("Résultat de la concaténation avec numpy.concatenate :")
print(concatenated_array_concat)
print("\nRésultat de la concaténation avec numpy.vstack :")
print(concatenated_array_vstack)
```

```
Résultat de la concaténation avec numpy.concatenate :
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

Résultat de la concaténation avec numpy.vstack :
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Après cette mise au point sur les concaténations, nous allons aborder un aspect de la manipulation des tableaux qui est crucial en data science : le changement de structure des ndarray. Trois mots-clés vont être explorés pour résoudre trois situations :

Mettre à plat avec ravel

Au cours des manipulations ou des prétraitements de données, il peut être nécessaire d'aplatir une matrice en un vecteur unidimensionnel. La commande `ravel` est là pour nous y aider :

```
x = np.array([[1, 2, 3], [4, 5, 6]])
y = np.ravel(x)
print(y)
# Output: array([1, 2, 3, 4, 5, 6])
```

La matrice x de deux lignes sur trois colonnes va ainsi être transformée en un vecteur plat y de six membres. La forme (`shape`) passe ainsi de (2, 3) à (6,).

Notons que l'utilisation de `reshape(x,-1)` produit le même résultat :

```
y = np.reshape(x,-1)
```

C'est justement sur cette commande `reshape` que nous allons nous appuyer pour résoudre une petite subtilité qui peut bloquer ensuite notre code.

Passer d'un vecteur unidimensionnel à un vecteur colonne avec reshape

La récupération d'une colonne pandas dans un ndarray ne conserve pas la forme de colonne d'origine. Cette non-conservation de la forme entraîne des problèmes pour certains algorithmes qui réclament un vecteur colonne et ont à la place un vecteur unidimensionnel. Voici un exemple illustrant ce phénomène :

```
import pandas as pd
import numpy as np

# Création d'un dictionnaire de données
data = {
    'col1': [1, 2, 3, 4, 5],
    'col2': [0.1, 0.2, 0.3, 0.4, 0.5],
}

# Création du DataFrame
df = pd.DataFrame(data)
b = df['col1'].values

print(b)
# Output: array([1, 2, 3, 4, 5])

print(b.shape)
# Output: (5,)
```

Le vecteur unidimensionnel est reconnaissable à sa forme en (n,) pour laquelle nous constatons un vide après la virgule. C'est ici que nous allons avoir recours à `reshape` pour rétablir le 1 manquant après la virgule.

```
b = b.reshape(b.shape[0],1)
print(b)

# Output:
array([[1],
       [2],
       [3],
       [4],
       [5]])
```

Afin de ne pas commettre d'erreur et favoriser l'automatisation, le recours à `b.shape[0]` représentant le nombre de lignes est encouragé. L'adjonction du 1 après la virgule va ainsi rétablir le statut de vecteur colonne de b et sa forme sera bien désormais (5, 1).

Passer d'un vecteur colonne à un vecteur unidimensionnel avec squeeze

Dans d'autres cas comme l'affichage d'images ou la construction de certains graphiques, la forme de vecteur colonne peut être problématique. Squeeze va nous aider à revenir au vecteur unidimensionnel de la forme (n,) de cette façon :

```
c = b.squeeze()
print(c)
# Output: array([1, 2, 3, 4, 5])
```

Désormais le vecteur c est bien de la forme (5,).

Cette vérification dans un sens ou dans l'autre est vivement conseillée et permettra de venir à bout de fréquents petits blocages.

2.1.4 La vectorisation

La vectorisation est une des grandes forces de Numpy. Elle offre la possibilité de transformer tous les membres d'un ndarray sans devoir passer par une boucle. Voici un exemple simple illustrant cette possibilité. Notons que le type a également été modifié de manière homogène afin de s'adapter aux contraintes du calcul.

```
import numpy as np

# Création d'un tableau 2D
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

print(arr.dtype)
# Output: int64

print(arr)
# Output :
[[1 2 3]
 [4 5 6]
 [7 8 9]]

# Division de tous les éléments par 2
result = arr / 2
```

```
print(result.dtype)
# Output: float64

print(result)
# Output:
[[0.5 1.  1.5]
 [2.  2.5 3.]
 [3.5 4.  4.5]]
```

Cette possibilité va se révéler très pratique lors des nombreuses opérations de transformations de nos données comme la normalisation, standardisation ou toute autre opération nécessitant d'agir sur l'ensemble des données.

2.2 La puissance au service du calcul scientifique

Outre les avantages structurels et ceux liés à la vectorisation, des performances élevées alliées à une grande variété de calculs scientifiques ont largement contribué à la popularité de la bibliothèque Numpy.

L'usage du langage C, réputé plus rapide que Python, ainsi que les optimisations internes de la bibliothèque expliquent cette puissance. Ceci a des conséquences sur toute la chaîne des opérations car lorsqu'un calcul est plus rapide, plus d'expérimentations sont envisageables, ce qui est souvent un gage de succès en data science.

Afin de nous rendre compte du gain de temps, comparons les performances du calcul de l'inverse sur une grosse liste d'un million de chiffres entre une fonction classique et en passant directement par Numpy :

```
# Création d'une fonction pour inverser les éléments d'une liste
def inverse_calculation(values):
  output = np.empty(len(values))
  for i in range(len(values)):
    output[i]=1.0/values[i]
  return output
```

```
# Création d'une liste d'un million de chiffres aléatoires entre 1 et 9
Tab_large = np.random.randint(1,10,size=1000000)
```

Avec la méthode classique, cela prend 1.99 seconde pour réaliser le calcul (ce temps peut varier en fonction de l'ordinateur de chacun). Nous utilisons la fonction `%timeit` pour mesurer le temps d'exécution.

```
%timeit inverse_calculation(Tab_large)
# Output :  1.99 s ± 415 ms per loop (mean ± std. dev. of 7 runs,
1 loop each)
```

En passant directement, et de manière beaucoup plus directe par Numpy, le temps est de 2.06 millisecondes :

```
%timeit inverse_calculation(1.0/Tab_large)
# Output : 2.06 ms ± 79.7 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)
```

Le calcul a pris presque 1000 fois moins de temps avec Numpy !

2.3 Les possibilités offertes par Numpy

Numpy propose tous les types de calculs nécessaires aux data scientists. Et le tout en parfaite harmonie avec les autres modules.

Prenons le temps de parcourir les options offertes en termes de calcul.

2.3.1 Opérations mathématiques de base

Toutes les opérations mathématiques que nous pourrions retrouver sur une calculatrice scientifique sont présentes. Le recours à la vectorisation mentionné précédemment facilite en plus grandement l'usage des formules.

Tout se fait avec un code lisible et facile à maintenir, ce qui est aussi un point fort de Numpy, une fois que nous avons adopté sa logique de fonctionnement.

Justement, observons à quel point il est simple de produire la somme globale, par ligne et par colonne d'une matrice quelconque en jouant sur le paramètre `axis` :

```
# Création d'une matrice carrée de 5 sur 5
arr2d = np.arange(25).reshape(5,5)
print(arr2d)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

```
# Somme globale
print(arr2d.sum())
# Output : 300

# Somme par colonne
print(arr2d.sum(axis=0))
# Output:  [50 55 60 65 70]

# Somme par ligne
print(arr2d.sum(axis=1))
# Output: [ 10  35  60  85 110]
```

2.3.2 Algèbre linéaire et calculs statistiques

Pour aller plus loin dans la complexité, voici comment résoudre le système d'équations linéaires suivant :

$2x + 3y = 2$

$4x + y = 14$

À l'aide de Numpy, voici comment poser le système pour le résoudre :

```
# Coefficients du système d'équations
coefficients = np.array([[2, 3], [4, 1]])

# Termes constants
constants = np.array([2, 14])

# Résolution du système d'équations
solution = np.linalg.solve(coefficients, constants)

print("La solution du système d'équations est :", solution)
# Output:  La solution du système d'équations est : [ 4. -2.]
```

Dans le registre des statistiques, Numpy possède également tous les outils nécessaires à l'étude de nos variables. Voici un exemple créant un dictionnaire d'indicateurs statistiques grâce à une fonction. Nous pourrons ainsi appliquer cette fonction sur n'importe quelle variable et ainsi obtenir un état immédiat :

```
def describe_np(data):
    """
    Calcul des statistiques descriptives pour un tableau NumPy.
    """
```

```
    stats = {}
    stats['count'] = len(data)
    stats['mean'] = np.mean(data)
    stats['std'] = np.std(data)
    stats['min'] = np.min(data)
    stats['25%'] = np.percentile(data, 25)
    stats['50%'] = np.percentile(data, 50)
    stats['75%'] = np.percentile(data, 75)
    stats['max'] = np.max(data)
    return stats

# Exemple d'utilisation
sales_data = np.array([350, 420, 290, 480, 510, 380, 410])
sales_stats = describe_np(sales_data)

# Affichage des statistiques
print("Statistiques descriptives des ventes:")
for key, value in sales_stats.items():
    print("{}: {}".format(key, value))
```

```
Statistiques descriptives des ventes:
count: 7
mean: 405.7142857142857
std: 69.45854732081838
min: 290
25%: 365.0
50%: 410.0
75%: 450.0
max: 510
```

Notons également que Numpy propose toute la panoplie des fonctions, des interpolations, des transformées ou des optimisations dont l'exposé complet ne serait pas pertinent. Mais nous encourageons les plus curieux d'entre nous à y revenir ultérieurement pour mesurer la richesse de Numpy en outils mathématiques.

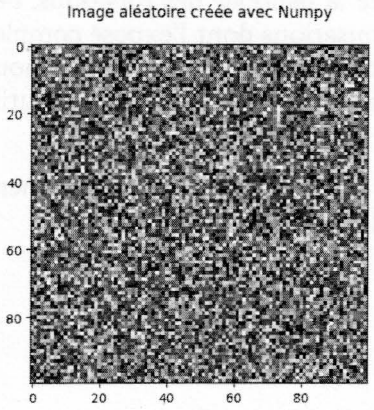
Pour finir sur une note plus inattendue, regardons ensemble comment Numpy peut être utilisé dans le domaine de l'image.

2.3.3 Création d'images

Numpy participe activement à de nombreuses bibliothèques, notamment celles dédiées aux manipulations d'images telles que OpenCV, imageio ou scikit-image. Cela est principalement dû au fait que NumPy offre une structure de données pertinente et efficace pour manipuler et stocker les données d'images sous forme de tableaux multidimensionnels.

Pour nous en convaincre, voici un exemple tout simple de création d'image aléatoire carrée de 100 pixels de côté en noir et blanc. Rappelons pour les personnes qui ne sont pas initiées aux images qu'un pixel a deux coordonnées x y et une valeur entre 0 et 255 permettant 256 nuances de gris entre le 0 marquant le noir et 255, le blanc. Ces valeurs sont aussi interprétables de manière décimale dans l'intervalle [0,1]. Ainsi, nous créons 10000 points ayant une valeur de gris aléatoire en une ligne de code :

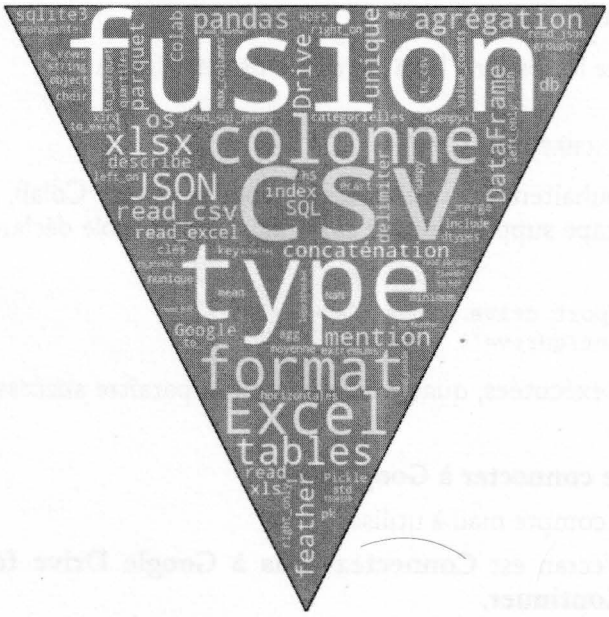
```
import numpy as np
import matplotlib.pyplot as plt

# Création d'une image avec NumPy
img = np.random.rand(100, 100)

# Titre
plt.title("Image aléatoire créée avec Numpy\n")

# Affichage de l'image
plt.imshow(img, cmap='gray')
plt.show()
```



Notre introduction à NumPy touche à sa fin. Bien que partielle, elle a permis de mettre en lumière la richesse des fonctionnalités de cette bibliothèque qu'il est fortement conseillé de maîtriser car elle sera omniprésente à toutes les étapes de notre travail sur les données.

Nous allons à présent mettre en pratique ces deux bibliothèques et entamer le processus de feature engineering consistant à collecter, nettoyer, explorer et analyser les données.

3. Collecte des données



Nuage de mots des termes techniques et informatiques utilisés pour la section Collecte des données

Définir son dossier de travail est un bon réflexe à prendre avant de commencer. En effet, les importations et exportations de fichiers en tout genre seront monnaie courante et figer, une bonne fois pour toutes, le chemin nous évitera de le répéter.

Nous réalisons cette opération grâce à la bibliothèque `os` dont l'acronyme correspond à Operating System. C'est en effet une bibliothèque standard mais essentielle permettant d'interagir avec le système d'exploitation, la manipulation des fichiers et des dossiers dont, dans notre cas, la gestion des chemins d'accès.

Définir le dossier de travail se passe de la façon suivante :

```
import os
os.chdir(r"D:\ENI\CHAPITRE_03")
```

Le « `r` » avant les guillemets est important car il indique au système que c'est une raw string empêchant ainsi l'interprétation de caractères comme `\n` qui renvoie à la ligne ou `\t` créant une tabulation.

Nous pouvons vérifier que le dossier a bien été défini de cette façon :

```
os.getcwd()
# Output : 'D:\\ENI\\CHAPITRE_03'
```

Pour celles et ceux qui souhaiteraient utiliser les notebooks Google Colab, la méthode nécessite une étape supplémentaire car il faut au préalable déclarer le dossier du Drive.

```
from google.colab import drive
drive.mount(r'/content/gdrive')
```

Une fois ces deux lignes exécutées, quatre écrans vont apparaître successivement :

- **Écran 1** : cliquer sur **Se connecter à Google Drive**.
- **Écran 2** : cliquer sur le compte mail à utiliser.
- **Écran 3** : le titre de l'écran est **Connectez-vous à Google Drive for desktop**. Cliquer sur **Continuer**.
- **Écran 4** : le titre de l'écran est **Google Drive veut un accès supplémentaire à votre compte Google**. Cliquer sur **Continuer**.

À l'issue de ces quatre étapes, l'accès au Drive est actif et nous pouvons utiliser la procédure précédente pour définir le dossier de travail.

3.1 Acquisition et contrôle des données

Voyons maintenant comment accéder à nos données. Attention de bien prendre soin au préalable de mettre nos fichiers de données sous le dossier de travail (ou dans un sous-dossier).

3.1.1 Les formats classiques des fichiers de données

Nous utiliserons Pandas pour acquérir nos données. La bibliothèque permet de lire quasiment tous les types de fichiers (Excel, CSV, TXT, JSON, SQL...). Voici un tableau récapitulatif des commandes qui commencent toutes par `read_` :

Type de fichier	Extension	Commande	Remarques	Options importantes
Excel	xls	`pd.read_excel('nom_du_fichier.xls')`	Module nécessaire : `pip install xlrd`	sheet_name
	xlsx	`pd.read_excel('nom_du_fichier.xlsx')`	Module nécessaire : `pip install openpyxl`	header usecols dtypes
CSV	CSV	`pd.read_csv('nom_du_fichier.csv, sep=';')`		sep
Texte	txt	`pd.read_csv ('nom_du_fichier.txt, delimiter= '\t')`		delimiter
JSON	json	`pd.read_json('nom_du_fichier.json)`		
		`pd.json_normalise ('nom_du_fichier.json)`	À favoriser pour les JSON contenant des structures imbriquées	

Outre ces formats de fichiers, il est courant de rencontrer des fichiers SQL.

Lire ce type de fichier nécessite une étape supplémentaire ainsi que l'utilisation du module gérant le SQL. Nous utiliserons `sqlite3` qui est la solution la plus simple pour lire directement un fichier de type db.

```
import sqlite3

# Adresse du fichier db
conn = sqlite3.connect(r"D:\...\fichier_db.db")

df = pd.read_sql_query('SELECT * FROM Ma_table', conn)
```

Remarque

Il est important de connaître le nom de la base de données à atteindre.

3.1.2 L'acquisition de données en pratique

Pour notre exemple, nous aurons recours au fichier CSV Restaurants.csv disponible ici :

https://raw.githubusercontent.com/eric2mangel/BDD/refs/heads/main/Restaurants.csv

Une fois le fichier récupéré, nous utiliserons la commande `read_csv()` pour le lire :

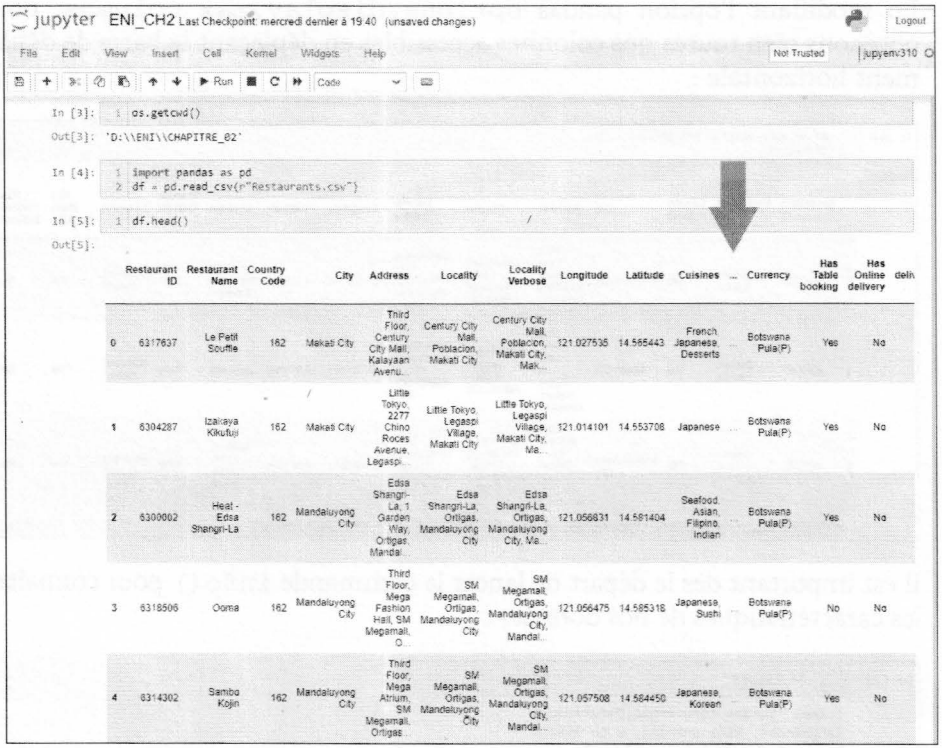
```
import pandas as pd
df = pd.read_csv(r"Restaurants.csv")
```

Première chose à faire après validation : vérifier que nos données sont bien lues en affichant les cinq premières lignes grâce à la commande `head()`. Cette commande affiche par défaut les cinq premières lignes mais il suffit de mettre le nombre souhaité entre parenthèses pour obtenir le nombre de lignes correspondant :

```
df.head()
```

Remarque

L'équivalent inverse de `head()` est `tail()` affichant les x dernières lignes du DataFrame.



	Restaurant ID	Restaurant Name	Country Code	City	Address	Locality	Locality Verbose	Longitude	Latitude	Cuisines	...	Currency	Has Table booking	Has Online delivery	deli
0	6317637	Le Petit Souffle	162	Makati City	Third Floor, Century City Mall, Kalayaan Avenu...	Century City Mall, Poblacion, Makati City	Century City Mall, Poblacion, Makati City, Mak...	121.027535	14.565443	French, Japanese, Desserts	...	Botswana Pula(P)	Yes	No	
1	6304287	Izakaya Kikufuji	162	Makati City	Little Tokyo, 2277 Chino Roces Avenue, Legaspi...	Little Tokyo, Legaspi Village, Makati City	Little Tokyo, Legaspi Village, Makati City, Ma...	121.014101	14.553708	Japanese	...	Botswana Pula(P)	Yes	No	
2	6300002	Heat - Edsa Shangri-La	162	Mandaluyong City	Edsa Shangri-La, 1 Garden Way, Ortigas, Mandal...	Edsa Shangri-La, Ortigas, Mandaluyong City	Edsa Shangri-La, Ortigas, Mandaluyong City, Ma...	121.056831	14.581404	Seafood, Asian, Filipino, Indian	...	Botswana Pula(P)	Yes	No	
3	6318506	Ooma	162	Mandaluyong City	Third Floor, Mega Fashion Hall, SM Megamall, O...	SM Megamall, Ortigas, Mandaluyong City	SM Megamall, Ortigas, Mandaluyong City, Mandal...	121.056475	14.585318	Japanese, Sushi	...	Botswana Pula(P)	No	No	
4	6314302	Sambo Kojin	162	Mandaluyong City	Third Floor, Mega Atrium, SM Megamall, Ortigas...	SM Megamall, Ortigas, Mandaluyong City	SM Megamall, Ortigas, Mandaluyong City, Mandal...	121.057508	14.584450	Japanese, Korean	...	Botswana Pula(P)	Yes	No	

Premier constat : nos données sont correctement lues mais nous remarquons la présence des « … » entre les noms des colonnes indiquant que toutes n'ont pas été affichées.

En modifiant l'option pandas `options.display.max_columns`, nous obtenons bien toutes nos colonnes accessibles en déplaçant la barre de défilement horizontale :

```
In [6]: 1 pd.options.display.max_columns = 999
        2 df.head(3)
```

Out[6]:

	Restaurant ID	Restaurant Name	Country Code	City	Address	Locality	Locality Verbose	Longitude	Latitude	Cuisines	Average Cost for two	Currency	Has Table booking	Has Online delivery
0	6317637	Le Petit Souffle	162	Makati City	Third Floor, Century City Mall, Kalayaan Avenu...	Century City Mall, Poblacion, Makati City	Century City Mall, Poblacion, Makati City, Mak...	121.027535	14.565443	French, Japanese, Desserts	1100	Botswana Pula(P)	Yes	No
1	6304287	Izakaya Kikufuji	162	Makati City	Little Tokyo, 2277 Chino Roces Avenue, Legaspi...	Little Tokyo, Legaspi Village, Makati City	Little Tokyo, Legaspi Village, Makati City, Ma...	121.014101	14.553708	Japanese	1200	Botswana Pula(P)	Yes	No
2	6300002	Heat - Edsa Shangri-La	162	Mandaluyong City	Edsa Shangri-La, 1 Garden Way, Ortigas, Mandal...	Edsa Shangri-La, Ortigas, Mandaluyong City	Edsa Shangri-La, Ortigas, Mandaluyong City, Ma...	121.056831	14.581404	Seafood, Asian, Filipino, Indian	4000	Botswana Pula(P)	Yes	No

Il est important dès le départ de lancer la commande `info()` pour connaître les caractéristiques de nos données :

```
In [7]: 1 df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9551 entries, 0 to 9550
Data columns (total 21 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Restaurant ID         9551 non-null   int64
 1   Restaurant Name       9551 non-null   object
 2   Country Code          9551 non-null   int64
 3   City                  9551 non-null   object
 4   Address               9551 non-null   object
 5   Locality              9551 non-null   object
 6   Locality Verbose      9551 non-null   object
 7   Longitude             9551 non-null   float64
 8   Latitude              9551 non-null   float64
 9   Cuisines              9542 non-null   object
 10  Average Cost for two  9551 non-null   int64
 11  Currency              9551 non-null   object
 12  Has Table booking     9551 non-null   object
 13  Has Online delivery   9551 non-null   object
 14  Is delivering now     9551 non-null   object
 15  Switch to order menu  9551 non-null   object
 16  Price range           9551 non-null   int64
 17  Aggregate rating      9551 non-null   float64
 18  Rating color          9551 non-null   object
 19  Rating text           9551 non-null   object
 20  Votes                 9551 non-null   int64
dtypes: float64(3), int64(5), object(13)
memory usage: 1.5+ MB
```

Le recours à cette commande est riche d'enseignements. Il nous informe sur le nombre de lignes et de colonnes, leur nom et leur type, la quantité de mémoire utilisée ainsi que du nombre de mentions non nulles. Ici, tout est bien rempli hormis la variable `Cuisines` présentant neuf mentions absentes. Nous verrons plus loin l'enjeu que représentent les valeurs manquantes et comment arbitrer ces situations.

Continuons notre prise de connaissances en produisant les statistiques des variables numériques grâce à `describe()`.

```
In [8]: df.describe()
```

Out[8]:

	Restaurant ID	Country Code	Longitude	Latitude	Average Cost for two	Price range	Aggregate rating	Votes
count	9.551000e+03	9551.000000	9551.000000	9551.000000	9551.000000	9551.000000	9551.000000	9551.000000
mean	9.051128e+06	18.365616	64.126574	25.854381	1199.210763	1.804837	2.666370	156.909748
std	8.791521e+06	56.750546	41.467058	11.007935	16121.183073	0.905609	1.516378	430.169145
min	5.300000e+01	1.000000	-157.948486	-41.330428	0.000000	1.000000	0.000000	0.000000
25%	3.019625e+05	1.000000	77.081343	28.478713	250.000000	1.000000	2.500000	5.000000
50%	6.004089e+06	1.000000	77.191964	28.570469	400.000000	2.000000	3.200000	31.000000
75%	1.835229e+07	1.000000	77.282006	28.642758	700.000000	2.000000	3.700000	131.000000
max	1.850065e+07	216.000000	174.832089	55.976980	800000.000000	4.000000	4.900000	10934.000000

Nous pouvons ainsi connaître le nombre de données par colonne, la moyenne (mean), l'écart-type (std), le minimum, les quartiles (25 %, 50 % et 75 %) et les extremums (min et max).

Pour les variables catégorielles, il est nécessaire de rajouter l'argument `include=["object","category"]` dans `describe()` afin d'obtenir un état sur les variables catégorielles.

```
In [9]: df.describe(include=["object"])
```

Out[9]:

	Restaurant Name	City	Address	Locality	Locality Verbose	Cuisines	Currency	Has Table booking	Has Online delivery	Is delivering now	Switch to order menu	Rating color	Rating text
count	9551	9551	9551	9551	9551	9542	9551	9551	9551	9551	9551	9551	9551
unique	7446	141	8918	1208	1265	1825	12	2	2	2	1	6	6
top	Cafe Coffee Day	New Delhi	Dilli Haat, INA, New Delhi	Connaught Place	Connaught Place, New Delhi	North Indian	Indian Rupees(Rs.)	No	No	No	No	Orange	Average
freq	83	5473	11	122	122	936	8652	8393	7100	9517	9551	3737	3737

Cela semble un peu succinct en termes d'informations mais nous savons désormais combien chaque variable a de mentions uniques, quelle est la mention la plus courante et sa fréquence.

Remarque

Si besoin, la commande `nunique()` permet de connaître le nombre de mentions uniques pour une variable en particulier.

Pour avoir un dénombrement précis par mention sur une variable en particulier, il faut utiliser la commande `value_counts()` :

```
In [10]:  1 df["Rating color"].value_counts()

Out[10]: Orange       3737
         White        2148
         Yellow       2100
         Green        1079
         Dark Green    301
         Red           186
         Name: Rating color, dtype: int64
```

Attention : pour les variables comportant un grand nombre de mentions, il est recommandé d'indiquer le nombre souhaité entre crochets. Sinon, seuls les cinq premiers et les cinq derniers seront affichés.

```
In [11]:  1 df["Restaurant Name"].value_counts()[:20]

Out[11]: Cafe Coffee Day        83
         Domino's Pizza         79
         Subway                 63
         Green Chick Chop       51
         McDonald's             48
         Keventers              34
         Pizza Hut              30
         Giani                  29
         Baskin Robbins         28
         Barbeque Nation        26
         Giani's                22
         Barista                22
         Dunkin' Donuts         22
         Costa Coffee           20
         Pind Balluchi          20
         Wah Ji Wah             19
         Twenty Four Seven      19
         Pizza Hut Delivery     19
         Sagar Ratna            19
         Republic of Chicken    18
         Name: Restaurant Name, dtype: int64
```

Mentionnons pour terminer la possibilité d'interagir facilement avec les noms de colonnes ou les mentions uniques d'une variable spécifique en utilisant respectivement les commandes `columns` et `unique()`.

```
In [12]: 1 mentions_rating = list(df["Rating text"].unique())
         2 mentions_rating

Out[12]: ['Excellent', 'Very Good', 'Good', 'Average', 'Not rated', 'Poor']

In [13]: 1 liste_variables = list(df.columns)
         2 liste_variables

Out[13]: ['Restaurant ID',
          'Restaurant Name',
          'Country Code',
          'City',
          'Address',
          'Locality',
          'Locality Verbose',
          'Longitude',
          'Latitude',
          'Cuisines',
          'Average Cost for two',
          'Currency',
          'Has Table booking',
          'Has Online delivery',
          'Is delivering now',
          'Switch to order menu',
          'Price range',
          'Aggregate rating',
          'Rating color',
          'Rating text',
          'Votes']
```

Après ces premiers contrôles, il peut être nécessaire, mais pas toujours, de modifier la structure des tables, concaténer ou fusionner d'autres données pour organiser l'information.

3.2 Manipulations avancées des données

Les données nécessitent très souvent une remise en forme pour répondre à nos besoins. Les cas de figure suivants sont courants dans la préparation des données :

- concaténer plusieurs tables identiques entre elles comme dans le cas de ventes mensuelles séparées dans plusieurs fichiers ;
- adjoindre des informations supplémentaires en fusionnant des tables ; par exemple, en rajoutant les infos sur des pays depuis un autre fichier ;
- agréger les données.

3.2.1 Concaténation

La concaténation peut se faire verticalement ou horizontalement. Il est néanmoins conseillé, pour les horizontales, de favoriser la fusion de type merging avec une clé primaire de fusion pour que les observations soient judicieusement assemblées.

Les fusions verticales de tables ayant les mêmes colonnes se réalisent sans problème :

```
# Création de deux DataFrames avec la même structure
df1 = pd.DataFrame({'A': [1, 2, 3],
                    'B': [4, 5, 6]})

df2 = pd.DataFrame({'A': [7, 8],
                    'B': [10, 11]})

# Concaténation verticale des deux DataFrames avec la même structure
concatenated_df = pd.concat([df1, df2])
concatenated_df
```

	A	B
0	1	4
1	2	5
2	3	6
0	7	10
1	8	11

Si certaines sont différentes, pas de panique ; les colonnes communes seront assemblées et des NaN seront utilisés pour combler les cases vides :

```
# Création d'un autre DataFrame avec des colonnes différentes
df3 = pd.DataFrame({'A': [13, 14],
                    'C': [16, 17]})

# Concaténation verticale avec un DataFrame ayant des colonnes
différentes
concatenated_df_with_diff_columns = pd.concat([df1, df3])
concatenated_df_with_diff_columns
```

	A	B	C
0	1	4.0	NaN
1	2	5.0	NaN
2	3	6.0	NaN
0	13	NaN	16.0
1	14	NaN	17.0

3.2.2 Fusion

La fusion est très courante et consiste à assembler deux tables grâce à une clé commune permettant de faire la liaison. Prenons un exemple simple commenté juste après pour bien maîtriser cet exercice.

```
import pandas as pd

# Définition de la table des informations sur les départements
departments = pd.DataFrame({
    "department_name": ["Ain", "Aisne", "Allier",
"Alpes-de-Haute-Provence", "Hautes-Alpes"],
    "surface_km2": [5762, 7362, 7340, 6925, 5549]
})

# Population des départements
departments_population = pd.DataFrame({
    "name": ["Ain", "Aisne", "Allier", "Alpes-de-Haute-Provence",
"Alpes-Maritimes"],
    "population": [663202, 527428, 334872, 166077, 1103941]
})

# Fusion des tables
departments_merged = pd.merge(departments, departments_population, \
                              left_on='department_name',
right_on='name', how='outer',indicator=True)

# Remplacement des valeurs NaN dans la colonne 'department_name' par
les valeurs de la colonne 'name'
departments_merged['department_name'].fillna(departments_merged['name'],
inplace=True)

# Suppression de la colonne redondante
departments_merged.drop(columns=['name'], inplace=True)

# Affichage des informations fusionnées
departments_merged
```

Out[16]:

	department_name	surface_km2	population	_merge
0	Ain	5762.0	663202.0	both
1	Aisne	7362.0	527428.0	both
2	Allier	7340.0	334872.0	both
3	Alpes-de-Haute-Provence	6925.0	166077.0	both
4	Hautes-Alpes	5549.0	NaN	left_only
5	Alpes-Maritimes	NaN	1103941.0	right_only

Il s'agit ici de fusionner deux tables contenant la surface de quelques départements pour la première et leur population pour la deuxième. Les tables ne possèdent pas exactement les mêmes départements. La fusion est lancée sur le nom du département dont la variable ne porte pas le même nom dans les deux DataFrames. À l'aide des mots-clés `left_on` et `right_on`, les noms respectifs de la table de gauche et de droite sont précisés. Avec le mot-clé `how`, le type de fusion est spécifié. Il correspond ici à une fusion totale gardant l'ensemble des deux afin de mettre en valeur l'intérêt de la mention `indicator`. Cette option permet d'indiquer, dans une variable nommée `_merge`, le type de fusion réalisé. C'est très pratique pour dissocier les fusions réussies (`both`) des observations sans équivalent (`left_only` et `right_only`). Il ne faut pas oublier de supprimer la variable `_merge` à chaque nouvelle fusion utilisant `indicator`.

Finalement, la ligne de code 20 permet de récupérer la mention absente de la variable `departement` à partir de la variable `nom` et cette dernière est supprimée en ligne 23, aboutissant à un DataFrame fusionné et exploitable.

Rappelons pour finir sur ce point que l'examen des fusions est indispensable pour garantir l'intégrité des données fusionnées. Il est essentiel de vérifier que chaque ligne a été fusionnée correctement afin d'éviter toute incohérence ou perte d'informations ultérieure. C'est une pratique saine et recommandée qui contribue à assurer la qualité des données et la fiabilité des analyses ultérieures.

3.2.3 Agrégation

L'agrégation consiste à regrouper des données en fonction de certaines caractéristiques, puis à calculer des statistiques ou des valeurs agrégées sur ces groupes.

Ce type de situation est rencontrée dès que nous avons différentes granularités dans nos données. Par exemple, dans le cas de données contenant le détail des ventes de différents magasins, il peut être nécessaire de faire un point sur les magasins et calculer leur chiffre d'affaires total. Et ensuite, faire un point par produit au global indépendamment du magasin. Ces deux approches nécessitent des agrégations.

Voici comment faire à l'aide des commandes `groupby` et `agg` :

```
import pandas as pd

# Création du DataFrame avec les données
data = {
    'Store': ['Paris', 'Paris', 'Lyon', 'Lyon', 'Paris'],
    'Product': ['A', 'B', 'C', 'A', 'C'],
    'Category': ['Food', 'Food', 'Books', 'Food', 'Books'],
    'Sales': [30, 200, 500, 80, 120]
}
df = pd.DataFrame(data)

# Statistiques des ventes par boutique et catégorie de produits
Sales_by_store_and_category = df.groupby(['Store', 'Category']).agg({
    'Sales': ['sum', 'mean', 'min', 'max']
}).reset_index()

# Affichage des résultats agrégés
print("\nSales by store and category:")
display(Sales_by_store_and_category)


# Statistiques des ventes par ville
print("Sales by store :")
Sales_by_store = df.groupby(['Store']).agg({
    'Sales': ['sum', 'mean', 'min', 'max']
}).reset_index()

# Affichage des résultats agrégés
display(Sales_by_store)

# Statistiques des ventes par catégorie de produits
print("Sales by category:")
Sales_by_category = df.groupby(['Category']).agg({
    'Sales': ['sum', 'mean', 'min', 'max']
}).reset_index()

# Affichage des résultats agrégés
display(Sales_by_category)
```

Sales by store and category :

	Store	Category	Sales			
			sum	mean	min	max
0	Lyon	Books	500	500.0	500	500
1	Lyon	Food	80	80.0	80	80
2	Paris	Books	120	120.0	120	120
3	Paris	Food	230	115.0	30	200

Sales by store :

	Store	Sales			
		sum	mean	min	max
0	Lyon	580	290.000000	80	500
1	Paris	350	116.666667	30	200

Sales by category :

	Category	Sales			
		sum	mean	min	max
0	Books	620	310.000000	120	500
1	Food	310	103.333333	30	200

Il était question ici uniquement d'agrégation sur une variable. Voici un exemple sur deux variables avec leurs indicateurs propres. Pour ce faire, le nombre de ventes a été ajouté à l'exemple précédent :

```
import pandas as pd

# Données
data = {
    'Store': ['Paris', 'Paris', 'Lyon', 'Lyon', 'Paris'],
    'Product': ['A', 'B', 'C', 'A', 'C'],
    'Category': ['Food', 'Food', 'Books', 'Food', 'Books'],
    'Sales': [30, 200, 500, 80, 120],
    'Quantity': [15, 20, 50, 20, 20]
}
df = pd.DataFrame(data)

# Regroupement par boutique
df2 = df.groupby(['Store']).agg({'Sales': ['sum', 'mean'],
'Quantity': ['sum']}).reset_index()
```

```
# Modification du nom des colonnes pour rajouter l'indicateur
df2.columns = ["_".join([str(index) for index in multi_index])
for multi_index in df2.columns.to_numpy()]

df2
```

Out[26]:

	Store_	Sales_sum	Sales_mean	Quantity_sum
0	Lyon	580	290.000000	70
1	Paris	350	116.666667	55

L'opération n'est pas très complexe et consiste simplement à ajouter des éléments en tant que clés de dictionnaire dans la procédure `agg`. Modifier les noms de colonnes grâce à la commande `"_".join` est conseillé pour éviter le multi-indexage des noms de variables, qui complique les manipulations.

Après toutes ces opérations sur les données, il peut être judicieux de sauvegarder les DataFrames pour éviter de tout relancer lors de la prochaine intervention.

3.2.4 Export des données

La nécessité de sauvegarder les données est en grande partie proportionnelle à leur taille. Le réflexe classique consiste à les exporter en CSV ou en Excel, mais il existe d'autres formats offrant des performances bien supérieures et, surtout, qui conservent le type des variables. Ce qui n'est pas le cas des sauvegardes en TXT, CSV et Excel.

Un jeu de données fictives d'un million de lignes et dix colonnes a été créé dans le but de mesurer les différences :

```
import pandas as pd
import numpy as np

# Création d'une grande table de données
n_rows = 1000000  # Nombre de lignes
n_cols = 10       # Nombre de colonnes

data = np.random.randn(n_rows, n_cols)
df = pd.DataFrame(data, columns=[f'col_{i}' for i in range(n_cols)])
```

Les fichiers ont été exportés dans six formats différents. Le temps nécessaire a été mesuré pour l'enregistrement et la lecture ainsi que le poids du fichier pour chaque format.

Le tableau suivant détaille le code pour enregistrer et lire les différents formats. Notons qu'avec Pandas, toutes les routines pour enregistrer commencent par `to_` suivi du nom du format de fichier. De la même manière, pour la lecture, tous débutent par `read_` :

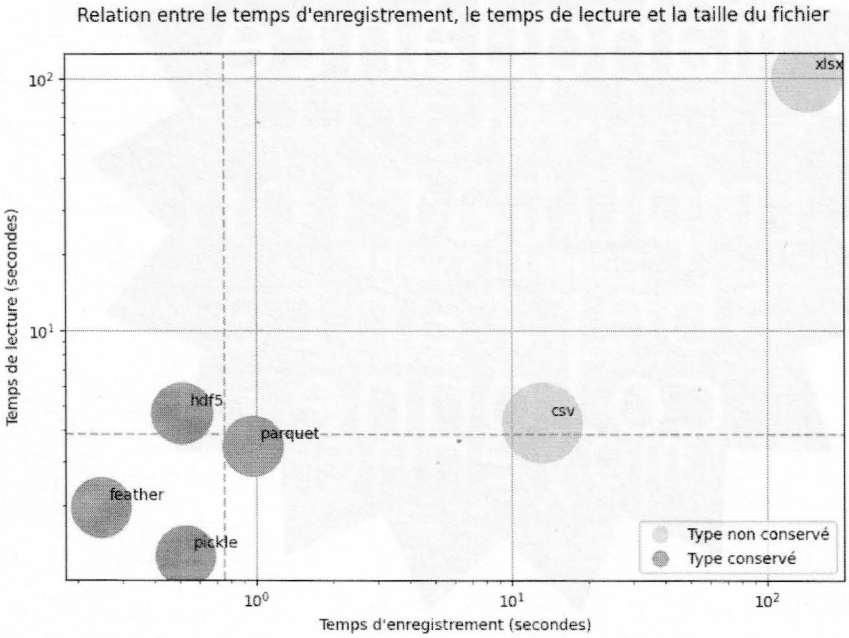
Format	Écriture	Lecture
csv	df.to_csv(r'D:\Fichiers\data.csv', index=False)	pd.read_csv(r'D:\Fichiers\data.csv')
Excel	df.to_excel(r'D:\Fichiers\data.xlsx', index=False)	pd.read_excel(r'D:\Fichiers\data.xlsx')
Parquet	df.to_parquet(r'D:\Fichiers\data.parquet', index=False)	pd.read_parquet (r'D:\Fichiers\data.parquet')
Pickle	df.to_pickle(r'D:\Fichiers\data.pkl')	pd.read_pickle(r'D:\Fichiers\data.pkl')
HDF5	df.to_hdf(r'D:\Fichiers\data.h5', key='df', mode='w')	pd.read_hdf (r'D:\Fichiers\data.h5', key='df')
Feather	df.to_feather(r'D:\Fichiers\data.feather')	pd.read_feather(r'D:\Fichiers\data.feather')

Pour se rendre compte, voici les mesures réalisées sur les différents formats :

Type	Temps d'enregistrement (en secondes)	Temps de lecture (en secondes)	Taille fichier (Ko)	Types conservés
csv	13.200	4.280	192686	Non
xlsx	145.000	101.000	128955	Non
parquet	0.975	3.450	80864	Oui
pickle	0.530	1.270	78156	Oui
hdf5	0.514	4.660	85945	Oui
feather	0.248	1.970	78148	Oui

Les fichiers Excel et CSV sont plus longs à écrire (écrire un fichier Excel prend 584 fois plus de temps que pour un Feather), plus longs à lire, leur poids de stockage est au moins 50 % plus lourd et ils ne conservent pas les types. Ainsi, changer le type de données, par exemple faire passer certaines variables de float64 à float16, ne sera pas conservé si les fichiers sont sauvegardés dans ces formats courants.

Le graphique suivant permet d'illustrer cette situation :



Remarque

Les échelles des axes ont dû être passées en logarithmique car la sous-performance du format XLSX écrasait tous les autres en base à gauche.

Il est maintenant temps d'aborder le sujet du nettoyage des données qui est un point essentiel du feature engineering.

4. Nettoyage des données

Nuage de mots des termes techniques et informatiques utilisés pour la section Nettoyage des données

Le nettoyage des données est quasiment systématique. Il est extrêmement rare de pouvoir l'éluder. Mais qu'entend-on par nettoyage ?

Le nettoyage des données consiste à sélectionner, corriger et organiser les données pour éliminer erreurs et incohérences. C'est essentiel pour garantir la qualité et la fiabilité des analyses ultérieures.

Regardons étape par étape comment mener à bien ce processus.

4.1 Sélection des données

Les données que nous utilisons ne sont pas forcément à prendre dans leur entièreté. En fonction de la mission à mener, toutes ne sont pas nécessaires. Par exemple, si nous souhaitons réaliser une analyse des produits vendus par une certaine enseigne, il ne sera d'aucune utilité de conserver les autres enseignes présentes dans la base. C'est primordial de commencer par là car nettoyer des données inutiles serait une perte de temps.

Voici différentes façons de sélectionner les lignes d'intérêt :

```
# Sélection de l'enseigne AAA
df_enseigne = df[df["Enseigne"] == "AAA"]

# Sélection des départements 14 et 78
df_departements = df[df["Département"].isin([14,78])]

# Sélection des départements 14 et 78 pour le mois de mai
df_departements2 = df[(df["Département"].isin([14,78])) &
(df["Mois"]=="mai")]

# Sélection des départements 14 et 78 pour le mois de mai avec query
df_departements3 = df.query("Département.isin([14, 78]) &
Mois=='mai'")
```

Remarque

Réduire le nombre de lignes par sélection peut aussi contribuer à améliorer le taux de remplissage des variables.

Réciproquement, toutes les variables ne sont pas forcément nécessaires. Commençons par les cas les plus simples : celles que nous pouvons supprimer sans nous poser de questions. C'est le cas des variables intégralement vides ou ne possédant qu'une seule modalité distincte, excluant ainsi les variables possédant une modalité et des valeurs manquantes qui pourraient potentiellement représenter une autre modalité. Ces variables, qui ne varient pas, peuvent être supprimées directement. Voici le moyen simple d'identifier les variables vides :

```
# Récupération du nom des variables vides
empty_variables = df.columns[df.isna().all()].tolist()
```

Et maintenant, étudions quelle solution pour identifier les variables n'ayant qu'une modalité. L'usage du mot-clé `dropna = False` force le système à considérer les valeurs manquantes comme une modalité à part entière :

```
# Récupération des variables n'ayant qu'une seule modalité
unique_variables = df.columns[(df.nunique(dropna=False) == 1)
.values].tolist()
```

Les variables n'ayant qu'une seule modalité et des valeurs manquantes nécessitent un examen plus approfondi afin de comprendre si les valeurs manquantes peuvent représenter une modalité par défaut. Cela pourrait être le cas d'une variable binaire pour laquelle seuls les 1 ont été notés, les vides représentant en fait 0. Ces variables seraient alors parfaitement valides et devraient être conservées, si possible en remplaçant les manquants par 0 de la façon suivante :

```
# Remplacement des manquants par 0
df['A'].fillna(0, inplace=True)
```

Pour le reste, il est important de déterminer à partir de quel seuil le pourcentage de non-réponses devient trop élevé, ce qui justifierait la suppression de la variable. Notons qu'il n'existe pas de seuil universellement défini à cet égard. Cette décision repose souvent sur le jugement du data scientist, qui doit prendre en compte divers facteurs contextuels.

Cependant, à titre indicatif, il est couramment accepté de considérer la suppression des variables comportant un taux de non-réponses supérieur à 40 %. Ce seuil n'est pas absolu et peut varier en fonction de la nature des données, du domaine d'application et des objectifs de l'analyse. Il est donc essentiel d'évaluer chaque situation pour adapter le seuil en conséquence.

Le taux de remplissage par variable est lui aussi assez simple à obtenir en appliquant la formule suivante :

```
# Taux de remplissage par variable
fill_rate = df.count() / len(df) * 100
```

Il sera ensuite pertinent de le représenter sous forme de graphique pour avoir immédiatement une bonne vision d'ensemble de la qualité de remplissage.

Remarque

La librairie Missingno est une solution intéressante qui met à notre disposition toute une panoplie de visualisations pour mieux comprendre et traiter les données manquantes dans nos datasets.

4.2 Contrôle de la qualité des données

4.2.1 Définition du bon type de données

Le jeu de données adapté à notre sujet est désormais prêt mais uniquement d'un point de vue quantitatif. Le contenu des variables doit maintenant être évalué d'un point de vue qualitatif. Les actions sont diverses et dépendent du sujet mais nous pouvons établir une liste des choses à vérifier :

- Pour les dates :
  - vérifier que les dates sont bien au format datetime et éventuellement harmoniser le format (français, américain...),
  - s'assurer qu'il n'y ait pas de dates aberrantes,
  - évaluer s'il est pertinent de séparer la partie heures-minutes-secondes de la date.
- Pour les variables numériques :
  - vérifier que le type est adapté aux données,
  - veiller à ce que toutes les données soient bien numériques.
- Pour les variables catégorielles :
  - établir la liste des mentions pour vérifier qu'elles sont toutes orthographiées de la même façon,
  - contrôler l'encodage dans le cas où les accents sont remplacés par des caractères bizarres. Cela peut constituer un vrai casse-tête dans certains cas. Le package `chardet` peut identifier l'encodage d'origine afin que les caractères s'affichent correctement. Il faudra ensuite le mentionner grâce à l'option `encoding` au moment de la lecture du fichier. Nous allons voir la marche à suivre.

4.2.2 Gestion des problèmes d'encodage

Il faut commencer par déployer le package Chardet :

```
pip install chardet
```

Le code suivant permet ensuite d'évaluer l'encodage :

```
1 import chardet
2 with open("Restaurants.csv", 'rb') as fichier_brut:
3     encodage = chardet.detect(fichier_brut.read(10000))
4
5 print(encodage)
```

```
{'encoding': 'UTF-8-SIG', 'confidence': 1.0, 'language': ''}
```

Il ne restera plus qu'à l'indiquer au moment de lire le fichier :

```
df = pd.read_csv(r"Restaurants.csv", encoding='utf-8-sig')
```

À l'issue de cette étape, les données sont normalement prêtes pour être étudiées. C'est le bon moment pour faire un point sur les valeurs aberrantes.

4.3 Identification des valeurs atypiques ou aberrantes

L'identification des valeurs aberrantes est une étape fondamentale. Bien qu'elle puisse être entreprise après le contrôle de la qualité des données, il est important de souligner que cette démarche peut également avoir lieu de manière flexible tout au long du processus, que ce soit lors de l'exploration initiale ou lors d'analyses plus approfondies.

Le terme anglophone *outlier* est couramment utilisé pour les qualifier. La traduction littérale, « aberrant », peut être parfois trompeuse pour les personnes qui se familiarisent avec car une valeur très éloignée des autres n'est pas forcément aberrante. Elle peut être juste atypique mais il est nécessaire d'investiguer. Une boutique qui vend une large gamme de produits possède des articles dont le prix est très bas ou très haut par rapport à la moyenne. Ces prix ne sont pas aberrants en soi, mais simplement moins courants. Il est important de faire la distinction entre les valeurs aberrantes et les valeurs simplement atypiques pour une compréhension précise des données statistiques.

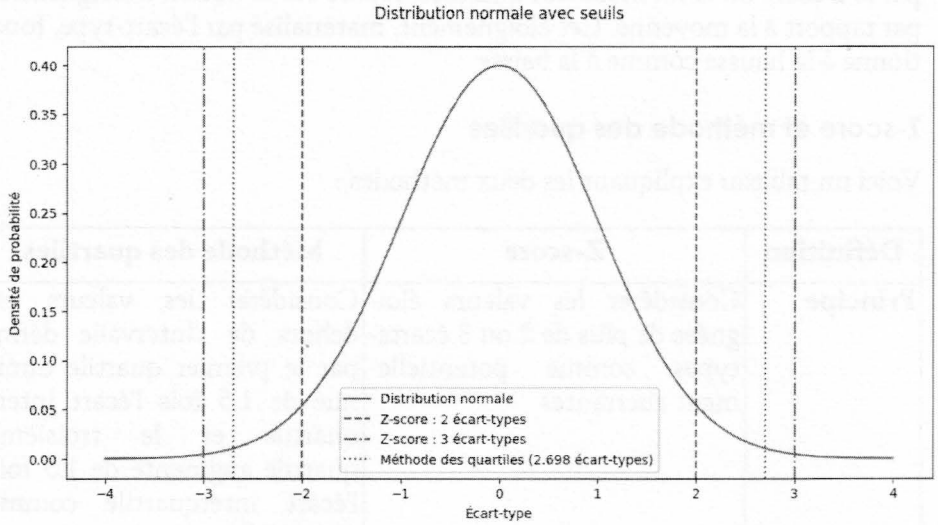
Le principe le plus couramment utilisé pour identifier les outliers, que ce soit par le z-score ou la méthode des quartiles, repose sur la notion d'éloignement par rapport à la moyenne. Cet éloignement, matérialisé par l'écart-type, fonctionne à la hausse comme à la baisse.

4.3.1 Z-score et méthode des quartiles

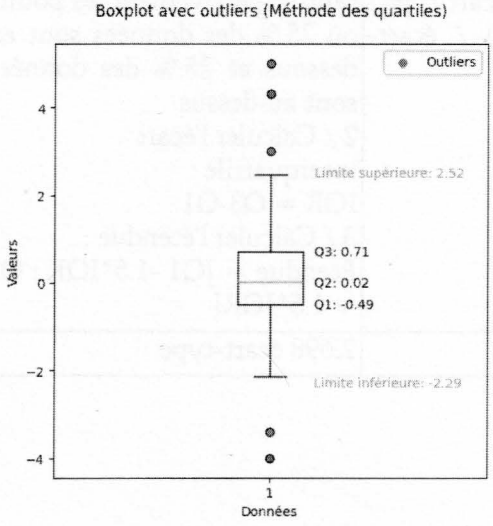
Voici un tableau expliquant les deux méthodes :

Définition	Z-score	Méthode des quartiles
Principe	Considérer les valeurs éloignées de plus de 2 ou 3 écarts-types comme potentiellement aberrantes	Considérer les valeurs en dehors de l'intervalle défini par le premier quartile diminué de 1.5 fois l'écart interquartile et le troisième quartile augmenté de 1.5 fois l'écart interquartile comme potentiellement aberrantes.
Calcul	Soustraire à chaque valeur la moyenne de l'échantillon puis diviser par l'écart-type : (valeur - moyenne) / écart-type	1 / Calculer le 1er quartile Q1 et le 3e quartile Q3 représentant respectivement les points où 25 % des données sont en dessous et 25 % des données sont au-dessus 2 / Calculer l'écart interquartile : IQR = Q3-Q1 3 / Calculer l'étendue : Etendue =]Q1 -1.5*IQR ; Q3 + 1.5*IQR[
Seuils	2 ou 3 écarts-types	2.698 écart-type

Le graphique suivant permet d'appréhender les niveaux de ces deux méthodes :



Le graphique boxplot que nous étudierons en détail plus loin identifie d'ailleurs directement les outliers sous forme de ronds comme dans ce visuel :



Afin de mettre en pratique ces deux méthodes, voici un code à conserver et à personnaliser :

```
import numpy as np
import pandas as pd

# Fonctions
# Détection des outliers avec z-score
def detect_outliers_z_score(data, threshold=3):
    z_scores = (data - data.mean()) / data.std()
    return np.abs(z_scores) > threshold

# Détection des outliers avec la méthode des quartiles
def detect_outliers_iqr(data):
    Q1 = data.quantile(0.25)
    Q3 = data.quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return (data < lower_bound) | (data > upper_bound)

# Création des données de test
np.random.seed(0)
data = np.random.normal(loc=0, scale=1, size=100)

# Ajout de quelques outliers
data[0] = 10
data[1] = -10

# Méthode du z-score
outliers_z_score = detect_outliers_z_score(data)
print("Outliers détectés avec la méthode du z-score:")
print(data[outliers_z_score])

# Méthode des quartiles
outliers_iqr = detect_outliers_iqr(pd.Series(data))
print("\nOutliers détectés avec la méthode des quartiles:")
print(data[outliers_iqr])
```

```
Outliers détectés avec la méthode du z-score:
[ 10. -10.]

Outliers détectés avec la méthode des quartiles:
[ 10. -10.]
```

D'autres approches alternatives existent comme Local Outlier Factor (LOF), Isolation Forest, DBSCAN ou Angle-Based Outlier Detection. Nous ne développerons que le LOF qui est, de surcroît, simple à comprendre mais il est vivement conseillé d'expérimenter les autres.

4.3.2 Local Outlier Factor

Le *Local Outlier Factor* (LOF) est basé sur le concept de densité locale. Contrairement aux approches reposant sur l'éloignement, le LOF considère comme outliers les valeurs ayant une faible densité locale. Ainsi, chaque point reçoit un score et l'algorithme entraîné prédira 1 s'il est normal et -1 dans le cas contraire.

Cette technique se révèle plus robuste que les deux précédentes, gérant mieux les grandes dimensions, les données complexes et hétérogènes. La seule difficulté, comme bien des algorithmes basés sur la densité, réside dans le choix du nombre de voisins.

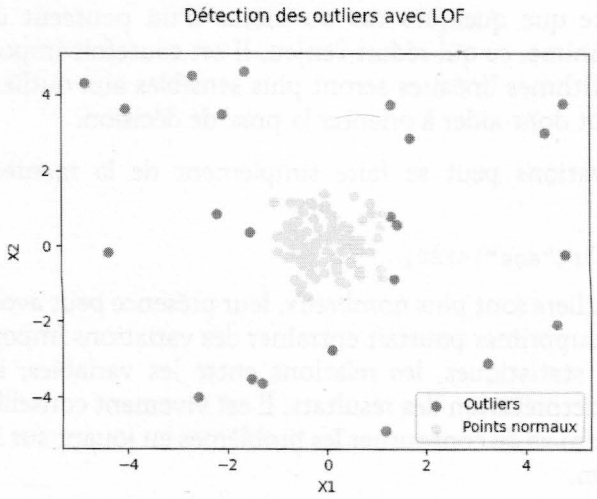
Rien de tel qu'un exemple pour comprendre le fonctionnement de l'algorithme.

Nous allons générer 100 points distribués normalement avec des coordonnées de deux dimensions pour lesquels nous appliquerons un coefficient de 0.6 pour réduire leur amplitude. Afin de bien souligner le phénomène, rajoutons 20 points avec un intervalle plus vaste [-5 ;5]. L'algorithme utilisé est importé depuis la bibliothèque Scikit-Learn :

```
from sklearn.neighbors import LocalOutlierFactor
import numpy as np

# Création des données de test
np.random.seed(0)

# Création d'une matrice de points
X = 0.6 * np.random.randn(100, 2)

# Introduction de points outliers
X_outliers = np.random.uniform(low=-5, high=5, size=(20, 2))

#Compilation verticale des 2 matrices
X = np.vstack([X, X_outliers])

# Entraînement du modèle LOF
clf = LocalOutlierFactor(n_neighbors=20)
y_pred = clf.fit_predict(X) # Entraînement et prédiction
```

À l'exécution de ce code, chaque point a été défini comme normal ou pas. Voici le code et la visualisation permettant de le constater :

```
import matplotlib.pyplot as plt

# Création d'un masque pour les outliers
outlier_mask = y_pred == -1

# Tracé du nuage de points en colorant les outliers en bleu
plt.scatter(X[outlier_mask, 0], X[outlier_mask, 1], color='#439cc8',
label='Outliers')
plt.scatter(X[~outlier_mask, 0], X[~outlier_mask, 1], color='#eee',
label='Points normaux')

# Ajout des étiquettes et un titre
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Détection des outliers avec LOF\n')

# Ajout d'une légende
plt.legend()

# Affichage du graphique
plt.show()
```



L'utilisation de ces différentes techniques va ainsi nous permettre de repérer les outliers. Mais que faut-il faire ensuite ?

4.4 Gestion des outliers

Une fois que les valeurs aberrantes ont été identifiées, il est crucial d'évaluer leur nature et leur impact sur l'analyse ou le modèle en cours. Si ces valeurs sont clairement le résultat d'erreurs de saisie ou de dysfonctionnements lors du recueil des données, il est généralement recommandé de les supprimer, car elles pourraient fausser les résultats des modélisations.

En revanche, lorsque ces valeurs sont légitimes, le choix devient plus délicat car leur impact peut varier considérablement. Trois types d'actions peuvent être entrepris alors : la suppression, le changement de distribution ou la décision de garder toutes les valeurs aberrantes. Explorons plus en détail ces différentes options.

4.4.1 Suppression des valeurs

La décision de supprimer des valeurs ne doit pas être prise à la légère. Il est recommandé de considérer cela comme une mesure de dernier recours, bien qu'il ne soit pas toujours possible de l'éviter. Avant tout, il est essentiel d'évaluer le nombre de cas concernés et de justifier la décision de manière appropriée.

Lorsque cela ne représente que quelques cas ou moins d'un pourcent de l'ensemble, l'impact est minime, ce qui réduit l'enjeu. Il est toutefois important de noter que les algorithmes linéaires seront plus sensibles aux outliers que d'autres. Ce critère peut donc aider à orienter la prise de décision.

Une suppression d'observations peut se faire simplement de la manière suivante :

```
df_wo_outliers = df[df["age"]<120]
```

En revanche, lorsque les outliers sont plus nombreux, leur présence peut avoir un impact significatif. Les supprimer pourrait entraîner des variations importantes sur les indicateurs statistiques, les relations entre les variables, et potentiellement biaiser l'interprétation des résultats. Il est vivement conseillé dans ce cas de les conserver mais de contourner les problèmes en jouant sur la distribution ou l'imputation.

4.4.2 Changement de la distribution

Transformer la distribution d'une variable est une méthode efficace pour réduire l'impact des valeurs extrêmes. En appliquant des transformations telles que la transformation logarithmique, la racine carrée ou des méthodes plus évoluées comme la transformation Box-Cox ou Yeo-Johnson, les données peuvent être ajustées pour mieux correspondre aux hypothèses des analyses statistiques ou des modèles, tout en atténuant l'influence des outliers.

Sans entrer pour le moment dans le détail des méthodes plus évoluées, la transformation d'une variable en log ou racine carrée se fait rapidement à l'aide des fonctions numpy dédiées :

```
# Transformation logarithmique
df['variable_log'] = np.log(df['variable'])

# Transformation racine carrée
df['variable_sqrt'] = np.sqrt(df['variable'])
```

4.4.3 Conservation des valeurs aberrantes

Le choix de conserver les valeurs aberrantes est parfaitement légitime mais à double tranchant.

D'un côté, leur préservation garantit l'intégrité et la complétude de l'ensemble des données, évitant ainsi toute perte d'informations potentiellement précieuses tout en conservant un juste reflet du réel. De l'autre côté, cela peut influencer de manière disproportionnée les résultats des analyses. Conserver ou non ces valeurs se résume souvent à un compromis entre le temps et la précision : le maintien des valeurs aberrantes permet de mesurer leur impact sur les résultats, renforçant ainsi la robustesse des analyses et des modèles. Cependant, cela peut nécessiter plus de temps et de ressources en raison de la complexité accrue due à la manipulation des données.

Enfin, une ultime solution pourrait consister à supprimer les valeurs extrêmes et à les traiter ensuite, de la même manière que les valeurs manquantes, en utilisant des techniques d'imputation. C'est cette dernière étape à laquelle nous allons nous atteler.

4.5 Imputations

L'imputation des données est une étape délicate consistant à remplacer les valeurs manquantes par des estimations pertinentes. Diverses techniques permettent de répondre à ce besoin. Dans cette exploration, nous irons des méthodes les plus simples et courantes pour vers des approches plus confidentielles et sophistiquées.

4.5.1 Imputation par la valeur la plus fréquente (modale)

L'imputation la plus simple consiste à remplacer les valeurs manquantes par la valeur la plus courante, nommée modale. Elle est considérée comme telle car elle nécessite juste de trouver la valeur la plus fréquente. Voici deux façons d'y recourir même si la première est de loin la plus simple :

- Méthode 1 en utilisant la fonction `mode()` de Pandas. Il est conseillé d'ajouter l'indice 0 pour les cas où il y a plusieurs modales :

```
# Imputation par la modale avec pandas
df['variable'] = df['variable'].fillna(df['variable'].mode()[0])
```

- Méthode 2 en utilisant la fonction `SimpleImputer` de Scikit-Learn :

```
# Imputation par la modale avec Scikit-learn
from sklearn.impute import SimpleImputer

# Création d'un imputeur avec stratégie « modale »
Imputer_mode = SimpleImputer(strategy='most_frequent')

# Imputation des valeurs manquantes
df['variable'] = imputer_mode.fit_transform(df[['variable']])
```

Les imputations peuvent évidemment se faire directement sur l'ensemble des variables mais ce n'est pas conseillé, il vaut mieux cibler car toutes les variables ne se prêtent pas à la même stratégie.

4.5.2 Imputation par la moyenne ou la médiane

Les imputations par moyenne ou médiane relèvent de la même approche que la précédente mais nécessitent une étape supplémentaire de calcul de la moyenne ou de la médiane. Ici, l'imputation par la médiane est à privilégier sur la moyenne si l'amplitude des valeurs est importante. Bien que simples à mettre en œuvre, ces trois méthodes que nous venons de voir présentent le défaut d'attribuer une même valeur à toutes les observations, ce qui les rend peu robustes. Avant d'explorer des méthodes plus sophistiquées, voici le code pour les mettre en pratique :

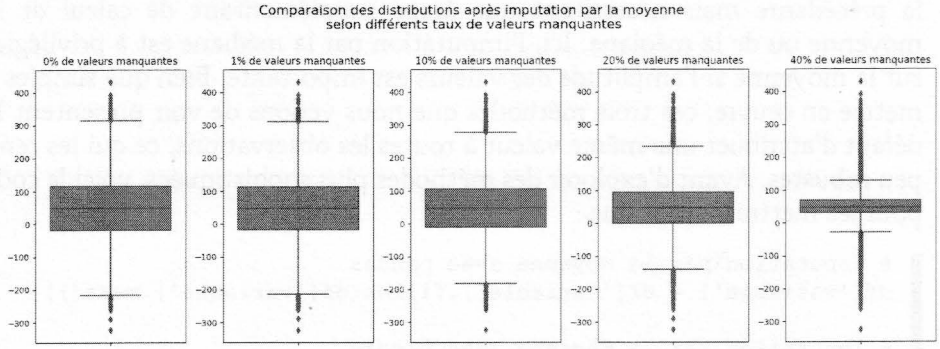
```
# Imputation par la moyenne avec pandas
df['variable'] = df['variable'].fillna(df['variable'].mean())

# Imputation par la médiane avec pandas
df['variable'] = df['variable'].fillna(df['variable'].median())
```

```
# Imputation par la modale avec Scikit-learn
from sklearn.impute import SimpleImputer

# Création d'un imputeur avec stratégie moyenne
imputer_mean = SimpleImputer(strategy='mean')

# Imputation des valeurs manquantes
df['variable'] = Imputer_mean.fit_transform(df[['variable']])


# Création d'un imputeur avec stratégie moyenne
imputer_median = SimpleImputer(strategy='median')

# Imputation des valeurs manquantes
df['variable'] = Imputer_median.fit_transform(df[['variable']])
```

À toutes fins utiles, signalons que l'usage de Scikit-Learn nécessite certes plus de code de prime abord mais permet de créer un objet qui pourra être réappliqué sur d'autres données et qu'il est conseillé lors de l'usage des pipelines que nous verrons ultérieurement.

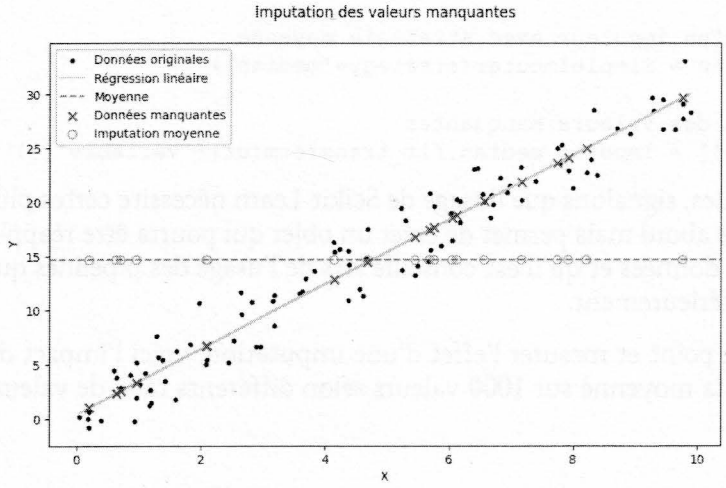
Pour finir sur ce point et mesurer l'effet d'une imputation, voici l'impact du remplissage par la moyenne sur 1000 valeurs selon différents taux de valeurs manquantes.

Le nombre d'outliers augmente avec le taux de manquants sous l'effet de la concentration au centre :

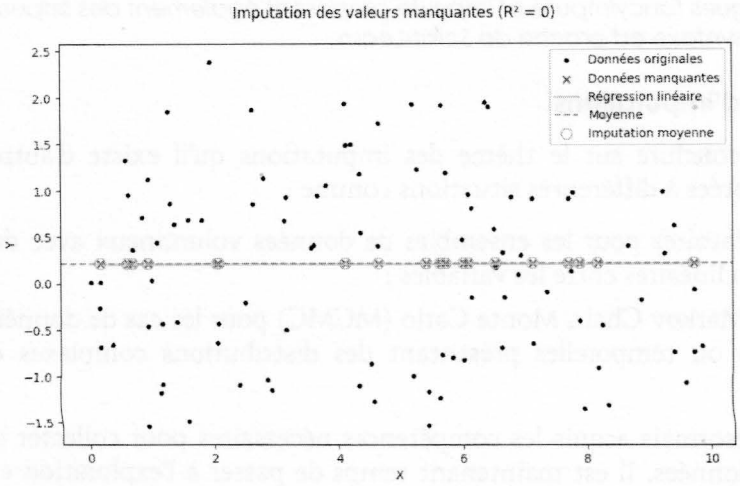


4.5.3 Imputation par régression

L'imputation par régression consiste à utiliser les relations entre les variables pour estimer les valeurs manquantes à l'aide d'une régression. Comparée aux précédentes méthodes, elle renvoie une valeur qui va dépendre d'une ou de plusieurs autres variables offrant ainsi un remplissage beaucoup plus précis. Afin d'illustrer l'intérêt de cette approche, voici un graphique montrant les différences entre l'imputation par moyenne et par régression :



Le constat est sans appel : les valeurs imputées par la moyenne sont très éloignées de celles obtenues par régression qui correspondent bien mieux à la tendance des autres points. La complétion par régression est donc préférable hormis dans les cas où nous ne disposerions pas de variables explicatives ($R^2=0$ signifiant qu'il n'y a aucune relation entre la variable à prédire et la ou les autres) comme dans l'exemple ci-dessous :



4.5.4 Imputation basée sur les plus proches voisins (KNN)

Une autre approche consiste à remplir les valeurs manquantes en se basant sur les points les plus proches. Cette méthode repose sur le postulat que les points similaires tendent à avoir des valeurs similaires, donc en trouvant les voisins les plus proches d'un point avec une valeur manquante, nous pouvons estimer cette valeur en utilisant les valeurs connues de ces voisins.

Ici, il est obligatoire de passer par Scikit-Learn car il n'existe pas de fonction dans pandas pour ce faire :

```
from sklearn.impute import KNNImputer

# Sélection des colonnes contenant des valeurs manquantes
cols_with_missing_values = ['col1', 'col2', 'col3']  # Remplacez
par les noms de vos colonnes
```

```
# Imputation KNN à ces colonnes
imputer = KNNImputer(n_neighbors=3) # Nombre de voisins à définir

df[cols_with_missing_values] =
imputer.fit_transform(df[cols_with_missing_values])
```

Remarque

Les bibliothèques fancyimpute et impyute proposent également des imputations dont la syntaxe est proche de Scikit-Learn.

4.5.5 Autres types d'imputations

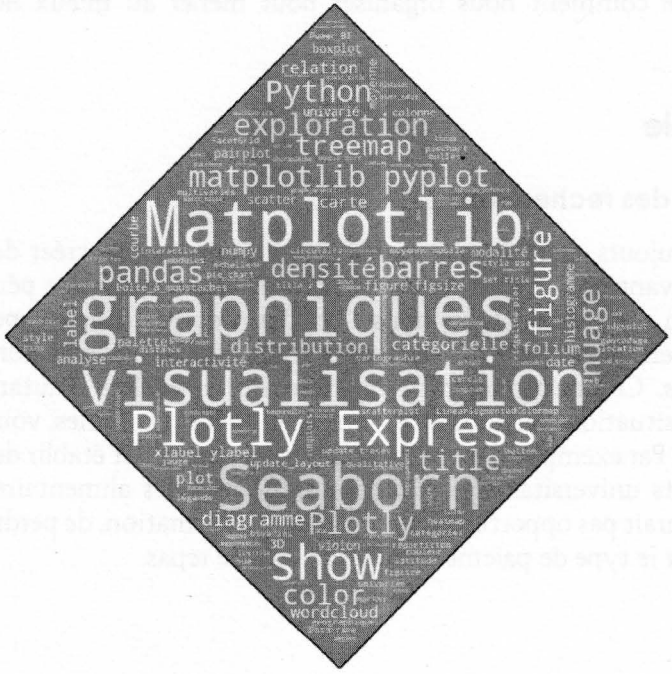
Notons pour conclure sur le thème des imputations qu'il existe d'autres méthodes adaptées à différentes situations comme :

- les forêts aléatoires pour les ensembles de données volumineux avec des relations non linéaires entre les variables ;
- la méthode Markov Chain Monte Carlo (MCMC) pour les cas de données séquentielles ou temporelles présentant des distributions complexes et inconnues.

Nous avons désormais acquis les compétences nécessaires pour collecter et nettoyer nos données. Il est maintenant temps de passer à l'exploration en utilisant les différentes bibliothèques graphiques disponibles en Python.

Chapitre 4
DataViz avec Matplotlib, Seaborn, Plotly

1. Introduction à la visualisation des données



Nuage de mots des termes techniques et informatiques utilisés pour ce chapitre

La visualisation est un élément fondamental dans l'analyse de données. Souvent qualifiée de data mining ou data exploration, cette démarche consiste à explorer les données dans le but de découvrir des modèles, des tendances, des relations voire des informations cachées permettant de comprendre pleinement le sujet d'étude.

1.1 La visualisation au service de la compréhension

À la manière d'un détective, nous allons chercher à comprendre les informations contenues en nous appuyant sur toute une panoplie de graphiques comme autant d'indices. Le choix est vaste et il est important de sélectionner les plus pertinents selon les cas. Tout l'art, finalement, consiste à représenter de manière claire et compréhensible des informations complexes. Cela demande quelques connaissances de base, de respecter certaines règles et surtout, de la pratique.

Regardons ensemble comment nous organiser pour mener au mieux nos recherches.

1.2 La méthodologie

1.2.1 Contextualisation des recherches

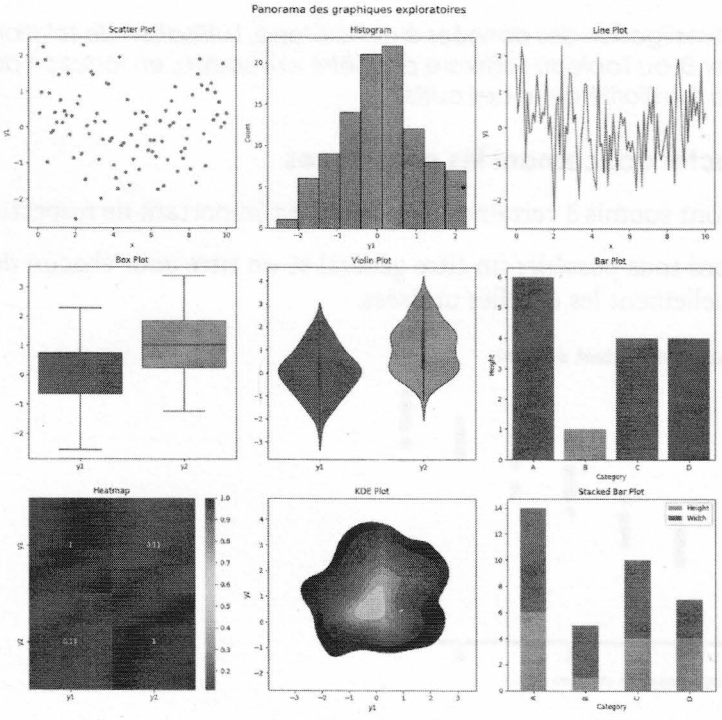
L'exploration est toujours menée dans un but. Il ne s'agit pas de créer des visuels au hasard. Avant de se lancer, il est important de définir notre périmètre d'intervention : que recherchons-nous ? Les données dont nous disposons dépassent souvent le cadre de nos besoins et toutes les variables ne sont pas forcément utiles. Ce questionnement préliminaire se révélera d'autant plus crucial dans les situations où nous disposons de plusieurs centaines, voire milliers d'entre elles. Par exemple, dans le cadre d'une mission pour établir des menus de restaurants universitaires en fonction des habitudes alimentaires des étudiants, il ne serait pas opportun, si nous avions l'information, de perdre du temps à visualiser le type de paiement ou les heures de repas.

1.2.2 Public concerné

L'exploration des données est souvent personnelle, visant à répondre à nos propres questionnements. Mais pour les cas où il y aurait d'autres destinataires, il est important de réfléchir à la façon de les présenter soit sur le fond, soit sur la forme, en fonction des autres personnes qui y auront accès. Cela relève de la même mécanique que d'adapter son discours au public concerné. Il convient également de s'interroger sur la notion de confidentialité et tout ce qui pourrait toucher au RGPD afin de déterminer ce qu'il est possible de représenter.

1.2.3 Les nombreuses possibilités de graphiques

Les graphiques sont fondamentaux pour améliorer la compréhension, il est donc primordial dans un premier temps de prendre connaissance de toute la gamme disponible, leur fonctionnement et leurs limites.



Le visuel ci-avant présente des exemples de graphiques courants avec lesquels nous allons nous familiariser dans les chapitres suivants. Il en existe bien sûr de nombreux autres, des plus simples aux plus confidentiels.

Les modules graphiques Python sont très différents tant au niveau de la diversité des possibilités que dans la facilité de mise en œuvre ou de personnalisation. C'est pourquoi, il faudra parfois se familiariser avec d'autres bibliothèques lorsque l'offre, pourtant importante de Matplotlib et Seaborn, fera défaut ou que la réalisation s'avérera trop complexe. Mais ces situations sont rares tant les deux solutions mentionnées couvrent déjà une large partie des besoins.

Dans cette perspective d'amélioration continue, il est recommandé de toujours rester ouvert aux solutions alternatives afin d'avoir la possibilité de pouvoir traiter n'importe quel type de visualisation.

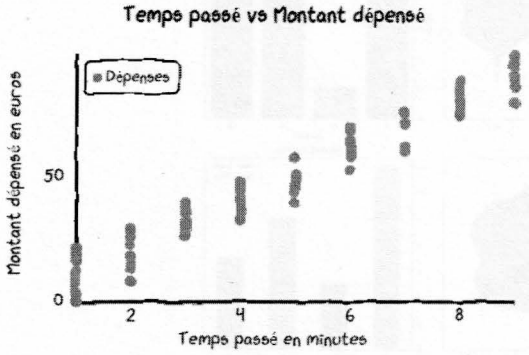
Remarque

Pour faciliter l'investigation des données à cette étape, l'utilisation de solutions telles que Power BI ou Tableau Software peut être envisagée, en fonction des besoins et de la familiarité avec ces outils.

1.2.4 Règles à respecter concernant les graphiques

Les graphiques sont soumis à certaines règles qu'il est important de respecter.

Ils doivent d'abord tous posséder un titre général et un titre pour chacun des axes avec éventuellement les échelles utilisées.



Ensuite, il convient de privilégier la visualisation la plus lisible et aérienne possible.

Ces deux précautions participent à l'objectif essentiel du graphique : la lisibilité et l'interprétabilité en moins de 30 secondes. Bien que cela puisse parfois sembler ambitieux, il devrait être considéré comme un horizon dans notre processus d'amélioration continue. En ce sens, les retours des divers interlocuteurs jouent un rôle essentiel. En prenant soin de les écouter et de les prendre en considération, nous avons l'opportunité d'affiner et d'améliorer constamment les visualisations que nous proposons.

Il est temps de faire connaissance avec les deux principales bibliothèques Python dédiées aux graphiques : Matplotlib et Seaborn.

2. Les principales bibliothèques pour la visualisation : Matplotlib, Seaborn et Plotly-Express

2.1 Matplotlib

2.1.1 Présentation de Matplotlib

Matplotlib est une bibliothèque de visualisation de données open source, développée à l'origine par John D. Hunter en 2003. Depuis ses débuts, elle est devenue l'une des plus populaires de l'écosystème Python pour la visualisation de données. Elle est souvent considérée comme primordiale pour ceux qui souhaitent s'initier à la visualisation en Python.

Matplotlib offre une grande variété de fonctionnalités et de possibilités de personnalisations pour créer différents types de graphiques comme des histogrammes, des nuages de points, des diagrammes en barres, des boîtes à moustaches, et bien d'autres.

Au cœur de Matplotlib se trouve `pyplot`, un sous-module qui fournit une interface simplifiée pour créer des graphiques et des visualisations avec Matplotlib. L'utilisation de `pyplot` est une étape fondamentale dans l'apprentissage de Matplotlib, car c'est à travers ce module que la plupart des graphiques sont créés et manipulés. Nous rencontrerons quasiment toujours la façon suivante de l'importer :

```
import matplotlib.pyplot as plt
```

L'alias `plt` est largement utilisé et il est vivement conseillé de l'utiliser.

Outre son large éventail de possibilités graphiques, la bibliothèque interagit parfaitement avec Numpy ou Pandas et elle fournit les fonctionnalités de base à Seaborn.

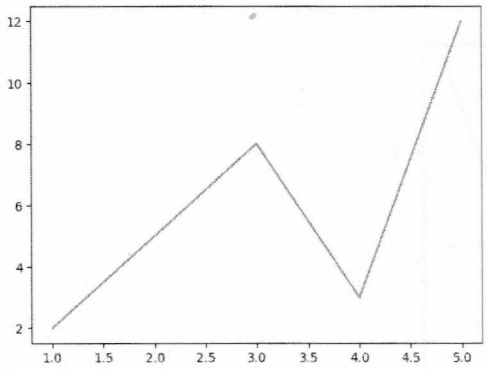
2.1.2 Premiers pas avec Matplotlib

Nous allons tracer un premier graphique tout simple avec Matplotlib nécessitant quatre étapes : import, données, tracé et affichage.

```
# 1 / Import de Matplotlib
import matplotlib.pyplot as plt # Attention à ne pas mettre de
majuscules

# 2 / Nos données
x = [1, 2, 3, 4, 5]
y = [2, 5, 8, 3, 12]

# 3 / Tracé du graphique
plt.plot(x, y)

# 4 / Affichage
plt.show() # Cette commande demande l'affichage
```

Remarque

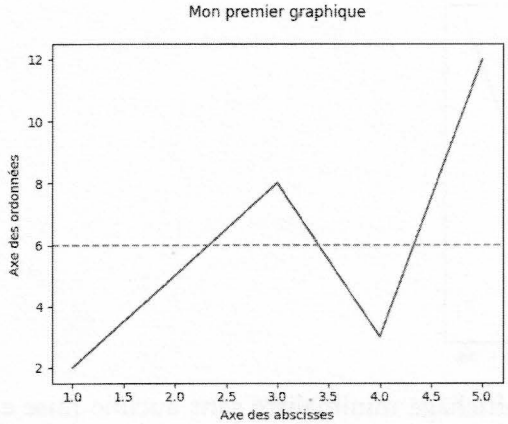
`plt.show()` peut être remplacé par « ; ». Il doit survenir à la fin du programme.



À ce stade, nous n'avons qu'un affichage minimaliste sans aucune mise en forme.

Mais nous allons pouvoir simplement rajouter des éléments en intercalant des lignes de commandes comme autant de nouvelles couches. Voici le code précédent intégrant en plus un titre, des noms aux axes, une ligne de couleur rouge et une ligne représentant la moyenne de l'axe vertical.

```
# Import de Matplotlib
import matplotlib.pyplot as plt # Attention à ne pas mettre de
majuscules

# Nos données
x = [1, 2, 3, 4, 5]
y = [2, 5, 8, 3, 12]

# Tracé du graphique avec personnalisation
plt.plot(x, y, color='red') # Tracé en rouge

# Ajout du titre et de celui des axes
plt.title('Mon premier graphique') # Ajout d'un titre
plt.xlabel('Axe des abscisses') # Nom de l'axe des abscisses
plt.ylabel('Axe des ordonnées') # Nom de l'axe des ordonnées

# Ajout d'une ligne horizontale pour la moyenne
plt.axhline(y=sum(y)/len(y), color='gray', linestyle='--')

# Affichage
plt.show() # Cette commande demande l'affichage
```



Sitôt la logique de construction des graphiques adoptée, il est assez intuitif de personnaliser le rendu en fonction des besoins. Voyons maintenant quelques options importantes de personnalisation qui nous aideront à faire face à de nombreux cas.

2.1.3 Personnalisation et options avancées

Savoir exploiter les options avancées permet de donner au graphique sa pleine puissance explicative. Voici quelques options qu'il est utile de connaître.

plt.figure()

Lors de la construction d'un graphique, il est souvent nécessaire, de devoir définir le cadre contenant le graphique en jouant sur sa taille ou sa définition. Tout se fait avec la commande `plt.figure()` qui doit figurer au début du code générant le graphique comme ici :

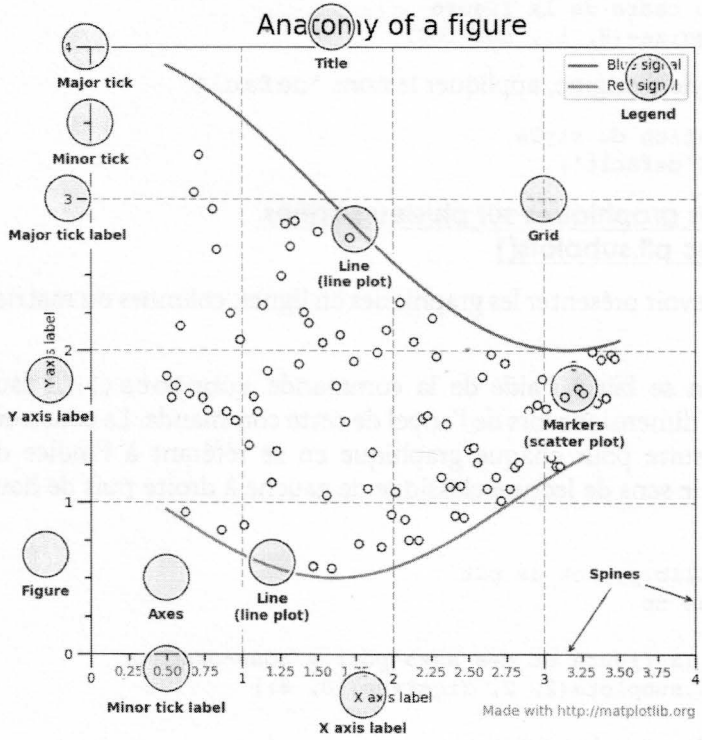
```
import matplotlib.pyplot as plt

# Création d'une nouvelle figure
plt.figure(figsize=(8, 6), dpi=100, facecolor='lightgray')
plt.plot([1, 2, 3], [4, 5, 6])
plt.title('Exemple de figure')
plt.show()
```

Voici des précisions sur les options utilisées :

- `figsize` permet de définir, dans l'ordre, la largeur et la hauteur du cadre.
- `dpi` spécifie la résolution de la figure en points par pouce.
- `facecolor` définit la couleur de fond du cadre.

Personnalisation des paramètres de la figure

Matplotlib offre la possibilité de modifier n'importe quelle partie du graphique : les titres, les axes, les graduations, etc. Le schéma suivant montre clairement les noms des fonctions permettant d'agir sur chaque poste :



Ce schéma est disponible directement sur le site à l'adresse suivante :

https://matplotlib.org/stable/tutorials/introductory/usage.html#parts-of-a-figure

plt.style.use()

Côté mise en forme, Matplotlib possède des styles prédéfinis.

La liste et le rendu sont accessibles directement sur le site à cette adresse :

https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html

Pour les utiliser, il suffit de définir le nom du style juste avant `plt.figure()`, en utilisant la commande `plt.style.use('Nom du style')` :

```
# Définition du style
plt.style.use('dark_background')
```

```
#Définition du cadre de la figure
plt.figure(figsize=(8, 6), dpi=100)
```

Pour revenir au style d'origine, appliquer le nom `'default'`.

```
# Réinitialisation du style
plt.style.use('default')
```

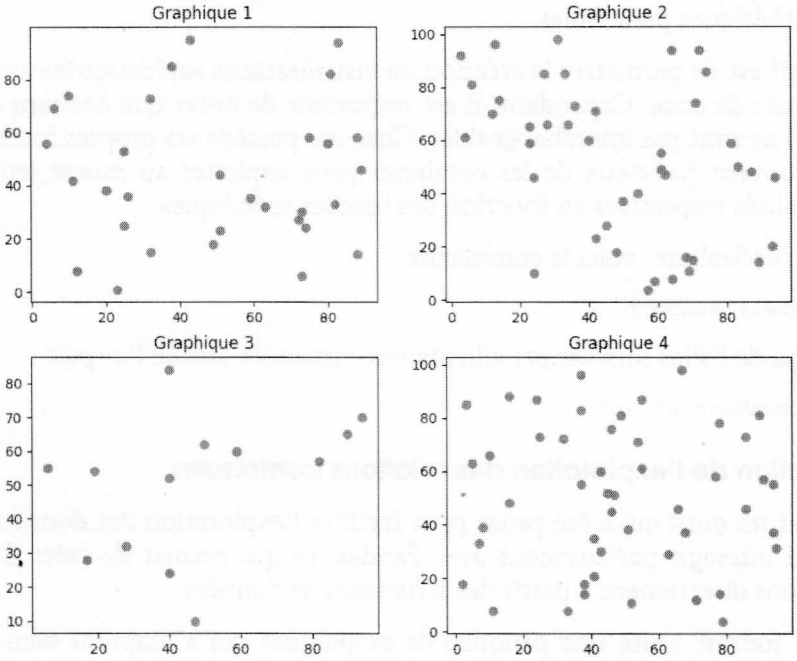
Organisation des graphiques sur plusieurs lignes et colonnes avec plt.subplots()

Il est courant de devoir présenter les graphiques en lignes, colonnes ou matrice selon les besoins.

Cette organisation se fait à l'aide de la commande `subplots()`. Il faut d'abord définir les dimensions lors de l'appel de cette commande. La construction aura lieu ensuite pour chaque graphique en se référant à l'indice de chacun défini par le sens de lecture classique de gauche à droite puis de haut en bas.

```
import matplotlib.pyplot as plt
import numpy as np

# Création de la figure et des axes pour 4 sous-graphes
fig, axs = plt.subplots(2, 2, figsize=(10, 8))

# Génération des graphiques
for i, ax in enumerate(axs.flat):
    # Création de 20 valeurs pour x
    x = np.random.randint(1, 100, 20)
    # Création de 20 valeurs pour y
```

```
        y = np.random.randint(1, 100, 20)
        ax.scatter(x, y) # Dessin du graphique
        ax.set_title(f'Graphique {i+1}')  # Titre du graphique


# Affichage des graphiques
plt.show()
```



Pour changer la disposition, il suffit de jouer sur les paramètres `nrows` et `ncols`.

Il est vivement conseillé d'apprendre à bien maîtriser Matplotlib avant de passer aux autres librairies graphiques. Cette bibliothèque permet d'obtenir la plupart des visualisations souhaitées et de se familiariser de manière simple avec tous les éléments des graphiques.

2.2 Seaborn

2.2.1 Présentation de Seaborn

Seaborn est une bibliothèque de visualisations graphiques fondée sur Matplotlib. Conçue sur une interface de plus haut niveau, elle simplifie la création de graphiques attrayants en offrant une syntaxe concise et des options esthétiques prédéfinies.

Son objectif est de permettre la création de visualisations sophistiquées avec un minimum de code. Cependant, il est important de noter que Seaborn et Matplotlib ne sont pas interchangeables. Chacune possède ses propres forces, et il est souvent judicieux de les combiner pour exploiter au mieux leurs fonctionnalités respectives en fonction des besoins spécifiques.

Pour installer Seaborn, voici la commande :

```
pip install seaborn
```

L'utilisation de l'alias `sns` est par ailleurs recommandée lors de l'import :

```
import seaborn as sns
```

2.2.2 Simplification de l'exploration des relations complexes

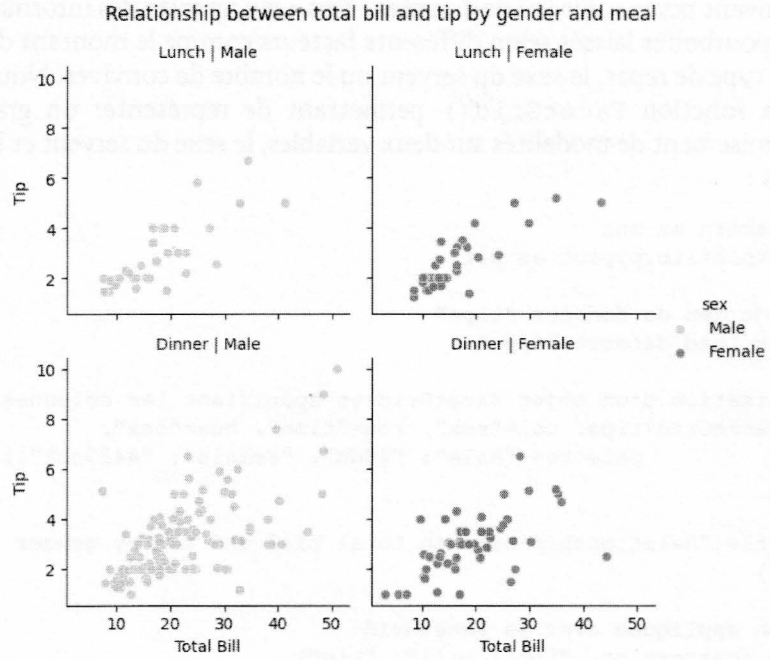
Seaborn est un outil qui a été pensé pour faciliter l'exploration des données. D'abord, il interagit parfaitement avec Pandas, ce qui permet de créer des visualisations directement à partir des structures de données.

Ensuite, il fournit toute une panoplie de graphiques qui s'adaptent bien à l'étude des relations entre les variables, il est ainsi très simple de passer de l'exploration univariée à bivariée.

Enfin, certaines fonctions comme `lmplot()`, `pairplot()` ou `FacetGrid()` facilitent grandement l'exploration multivariée pour une syntaxe très concise.

L'exemple suivant porte sur le jeu de données « tips » qui apporte des informations sur les pourboires laissés selon différents facteurs comme le montant de l'addition, le type de repas, le sexe du serveur ou le nombre de convives. Nous utiliserons la fonction `FacetGrid()` permettant de représenter un graphique par croisement de modalités sur deux variables, le sexe du serveur et le type de repas :

```
import seaborn as sns
import matplotlib.pyplot as plt

# Import du jeu de données "tips"
tips = sns.load_dataset("tips")

# Initialisation d'un objet FacetGrid en spécifiant les colonnes
g = sns.FacetGrid(tips, col="sex", row="time", hue="sex",
                  palette={"Male": "#ddd", "Female": "#439cc8"})

# Titre
plt.suptitle("Relationship between total bill and tip by gender
and meal")

# Fonction appliquée avec le FacetGrid
g.map(sns.scatterplot, "total_bill", "tip")

# Ajout d'une légende et mise en forme
g.add_legend()
g.set_axis_labels("Total Bill", "Tip")
g.set_titles(col_template="{col_name}",
row_template="{row_name}")

# Affichage
plt.show()
```



Remarque

La démonstration du `pairplot()` qui offre le meilleur rapport entre le nombre de lignes de code et la quantité d'informations sera abordée à la section Graphiques multivariés.

Après avoir présenté les deux grandes bibliothèques de visualisation, il est temps de faire connaissance avec une solution un peu plus récente et interactive : Plotly-express.

2.3 Plotly.express

2.3.1 La version simplifiée de Plotly

Dans l'écosystème Python, de nombreuses bibliothèques graphiques sont disponibles, mais ces derniers temps, Plotly-Express semble gagner en importance, méritant que nous nous y attardions.

Plotly-Express est construit sur la bibliothèque Plotly, qui permet de créer une grande variété de visualisations en 2D comme en 3D. Cependant, Plotly-Express simplifie le processus en fournissant des fonctions prédéfinies pour les tâches courantes de visualisation, ce qui réduit considérablement la quantité de code nécessaire pour créer des graphiques interactifs.

Voici comment procéder à l'installation :

```
pip install plotly_express # Attention à bien mettre un « _ »
```

Et maintenant, à titre de comparaison, voici le même graphique, un nuage de points, réalisé avec Plotly et Plotly-Express :

– Version Plotly

```
import plotly.graph_objs as go

# Données
x = [1, 2, 3, 4, 5]
y = [10, 11, 12, 13, 14]

# Création la figure
fig = go.Figure()

# Création du nuage de points avec Plotly
fig.add_trace(go.Scatter(x=x, y=y, mode='markers', name='Données'))

# Ajout des titres et des étiquettes
fig.update_layout(title='Nuage de points avec Plotly',
                  xaxis_title='Axe X',
                  yaxis_title='Axe Y')

# Affichage de la figure
fig.show()
```

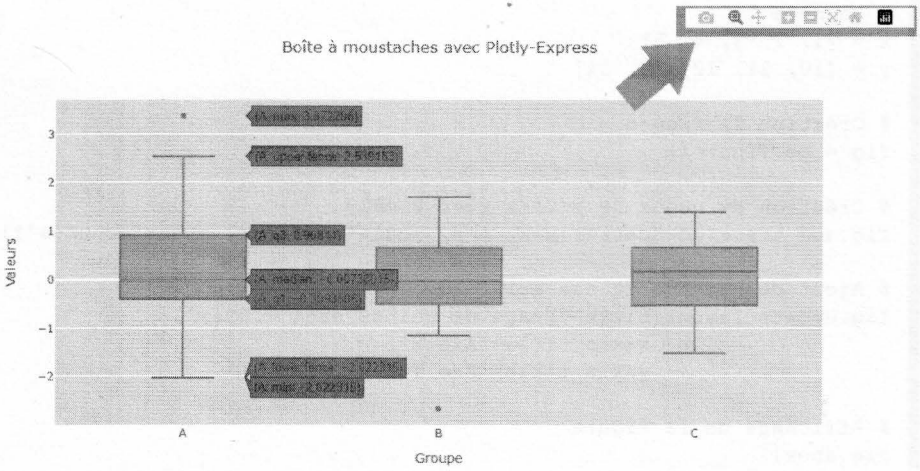
– Version Plotly-Express

```
import plotly.express as px

# Données
x = [1, 2, 3, 4, 5]
y = [10, 11, 12, 13, 14]

# Création du nuage de points avec Plotly-Express
fig = px.scatter(x=x, y=y, title='Nuage de points avec Plotly-Express',
labels={'x': 'Axe X', 'y': 'Axe Y'})

# Affichage de la figure
fig.show()
```

Le code de Plotly-Express est nettement plus court.

2.3.2 L'interactivité de Plotly-Express

L'autre force de Plotly-Express, dans le cas présent par rapport à Matplotlib et Seaborn, réside dans son interactivité. Cette capacité à créer des visualisations interactives est un vrai plus qui facilite l'exploration tout en la rendant plus dynamique. Elle permet en effet d'accéder facilement, par exemple, aux coordonnées de certains points d'un graphique ou aux mesures statistiques d'une boîte à moustaches, comme dans le cas suivant :



Ici, en passant la souris sur les boxplots, les caractéristiques s'affichent, ce qui est très pratique pour investiguer. Mentionnons également les possibilités offertes par le menu en haut à droite permettant de zoomer, recentrer et surtout d'exporter la figure en un seul clic.

2.3.3 L'avenir de Plotly-Express

Plotly-Express a été introduite en 2019, ce qui en fait une bibliothèque bien plus récente que Matplotlib et Seaborn, créées respectivement en 2003 et 2012. Cette différence d'ancienneté a joué en faveur de ces deux dernières, qui sont largement utilisées dans d'autres librairies et bénéficient d'une vaste communauté d'utilisateurs et de développeurs. De plus, Matplotlib et Seaborn continuent d'être activement développées, ce qui renforce leur position en tant que références dans le domaine de la visualisation de données.

Plutôt que de considérer Plotly-Express comme une solution de remplacement, il serait plus judicieux de la voir comme une troisième corde à notre arc dans notre boîte à outils de visualisation de données. La combinaison des trois bibliothèques permettra de toujours mieux répondre aux différents besoins en termes de visualisations.

3. Les différents types de graphiques

3.1 Les enjeux

3.1.1 Le cheminement vers le bon graphique

Le choix du bon type de graphique pour représenter un ensemble de données n'est pas toujours évident. Il est essentiel de commencer par définir clairement l'objectif du graphique et de s'assurer que nous avons une compréhension approfondie du sujet à traiter. Cette étape de recherche et de réflexion participe souvent à renforcer notre connaissance du problème. Ensuite, il est crucial de sélectionner le type de graphique le plus approprié parmi de multiples options disponibles, en veillant à ce qu'il réponde efficacement à nos questions. Enfin, une attention particulière doit être accordée à la finalisation et à la mise en forme du graphique, car ces éléments viennent renforcer le message qu'il véhicule.

3.1.2 Les postes importants

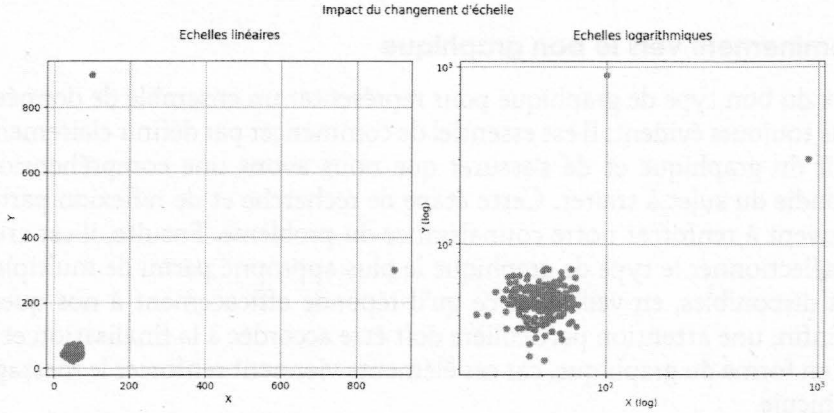
La création d'un graphique efficace repose avant tout sur la simplicité et la clarté. Il est primordial que l'information soit accessible immédiatement, révélant ainsi tous les aspects du sujet abordé. Au-delà du simple choix du type de graphique, il convient de valoriser la recherche des bonnes couleurs, d'une police d'écriture bien lisible, ainsi que toute autre démarche visant à garantir une mise en forme claire et aérée. Le graphique ne doit pas être considéré isolément, mais comme un tout cohérent où chaque élément, du fond à la forme, contribue à la transmission efficace de l'information.

3.1.3 Les contraintes

Lors de la création des graphiques, il sera nécessaire de composer avec certaines contraintes.

Contraintes d'échelle

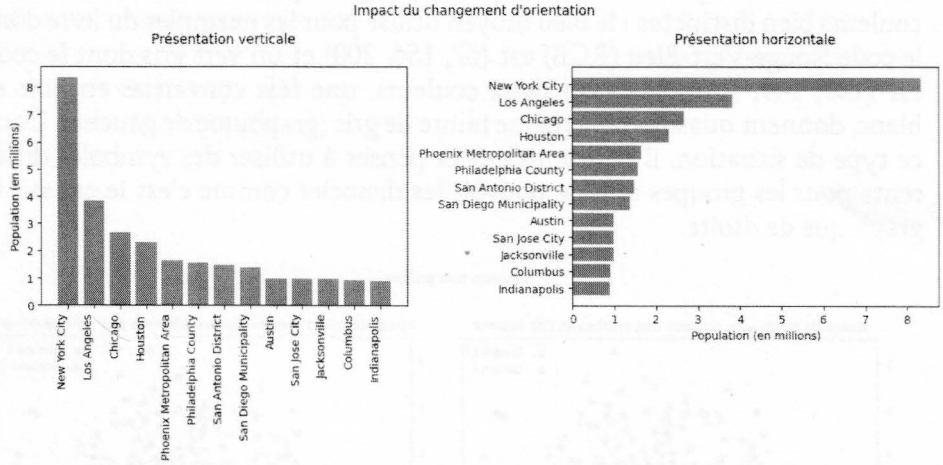
Parmi celles-ci, les problèmes d'échelles sont fréquents, notamment lorsque des valeurs extrêmes viennent perturber la disposition des autres points, les écrasant dans un coin du graphique. Pour remédier au problème, l'augmentation de la taille du graphique n'est pas toujours possible en raison de contraintes d'espace, ce qui peut nous amener à modifier l'échelle des axes. Ce changement n'est cependant pas sans conséquence sur la représentation fidèle des distributions, pouvant entraîner des distorsions. L'illustration suivante met en évidence l'impact de cette modification.



Contraintes de lisibilité

Un soin tout particulier devra également être apporté à la lisibilité des textes. Cette contrainte est très fréquente lors de la construction de plusieurs graphiques sur une même page, ce qui peut nécessiter de nombreux essais pour aboutir au juste équilibre. D'une manière générale, il est important de toujours s'assurer que les textes sont suffisamment grands pour être lus aisément, qu'ils ne cachent pas d'autres éléments graphiques, et que la couleur du texte contraste correctement avec le fond.

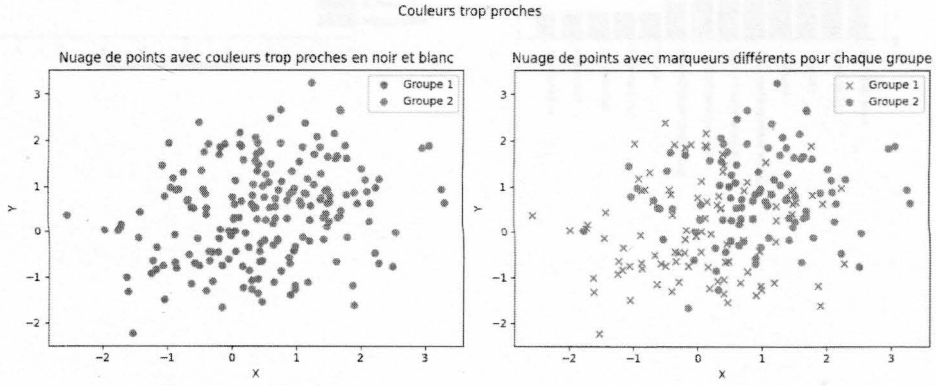
N'oublions pas, pour terminer sur ce point, le confort de lecture. L'orientation du graphique peut, en effet, jouer un rôle important à cet égard. Par exemple, dans le cas suivant, une présentation horizontale est préférable car elle évite de devoir incliner la tête pour lire les noms des villes.



Contraintes liées aux couleurs

La gestion des couleurs constitue souvent une contrainte délicate lors de la création de graphiques. Il est essentiel de rechercher une harmonie visuelle en utilisant une palette de couleurs restreinte, cohérente et commune à tous les visuels. L'utilisation de palettes trop variées peut entraîner une dissonance visuelle et rendre la comparaison des données difficile pour le lecteur. De plus, dans certains cas, des contraintes spécifiques sur la palette de couleurs peuvent être imposées, ce qui peut poser des problèmes, notamment si les nuances sont trop proches ou si les graphiques doivent être imprimés en noir et blanc, comme c'est le cas pour cet ouvrage. Cette dernière remarque est également valable pour prendre en compte, parmi notre public, les gens atteints de daltonisme.

Le visuel suivant produit un exemple de nuage de points coloriés selon deux couleurs bien distinctes : le bleu moyen utilisé pour les exemples du livre dont le code Rouge-Vert-Bleu (RGB) est (67, 156, 200) et un vert gris dont le code est (121, 149, 150). Mais ces deux couleurs, une fois converties en noir et blanc, donnent quasiment la même teinte de gris (graphique de gauche). Dans ce type de situation, il faut anticiper et penser à utiliser des symboles différents pour les groupes afin de pouvoir les dissocier comme c'est le cas sur le graphique de droite.



3.2 Les graphiques univariés

Nous allons commencer notre exploration des différents graphiques par les univariés, à savoir ceux ne décrivant qu'une seule variable. Pour des raisons de commodités de présentation, les graphiques seront regroupés par type de données : numériques (également appelées quantitatives) et catégorielles (également appelées qualitatives).

3.2.1 Graphiques univariés pour les données numériques

Histogramme

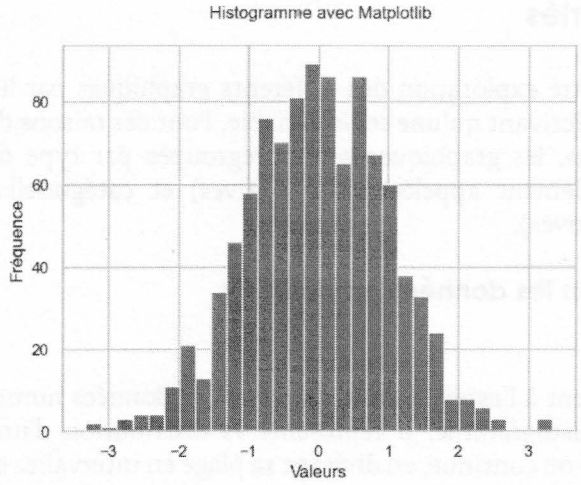
Le premier graphique qui vient à l'esprit dans le domaine des données numériques est certainement l'histogramme. Il représente la distribution d'une variable numérique, discrète ou continue, en divisant sa plage en intervalles et en affichant le nombre d'occurrences dans chaque intervalle sous forme de barres. Il permet de visualiser rapidement la forme de la distribution d'une variable.

Remarque

Attention à ne pas confondre l'histogramme avec le graphique en barres (cf. sous-section Graphiques univariés pour les données catégorielles de ce chapitre) qui concerne les variables catégorielles.

L'histogramme se construit simplement de la façon suivante avec Matplotlib avec la fonction `hist` :

```
import matplotlib.pyplot as plt
import numpy as np

# Génération de données aléatoires
data = np.random.normal(loc=0, scale=1, size=1000)

# Création de l'histogramme avec Matplotlib
plt.hist(data, bins=30, color='#439cc8', edgecolor='black')
plt.title('Histogramme avec Matplotlib')
plt.xlabel('Valeurs')
plt.ylabel('Fréquence')
plt.show()
```



Notons qu'avec l'option `bins`, il est possible de faire varier le nombre de regroupements. Ces regroupements sont de taille égale.

La réalisation du même graphique en Seaborn utilise la fonction `hisplot` en reprenant les mêmes arguments. Elle permet en plus de tracer la courbe de densité.

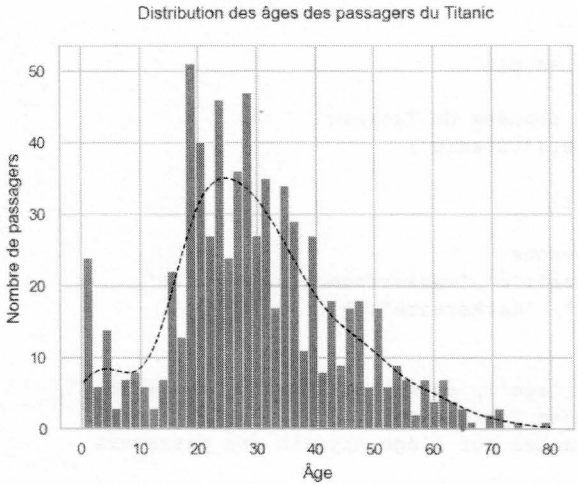
Courbe de densité

Ajouter une courbe de densité permet une représentation lissée et continue de la distribution des données. Elle complète ainsi les informations discrètes de l'histogramme, parfois sujet aux variations brusques, par une vue plus fluide et probabiliste de la densité des valeurs.

L'utilisation de Seaborn est ici recommandée car la syntaxe est bien plus concise et le rendu plus esthétique qu'avec Matplotlib.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Import du jeu de données
titanic = sns.load_dataset('titanic')

# Tracé du graphique
sns.histplot(data=titanic, bins=50, x='age', kde=True,
color='#439cc8', alpha=1, line_kws =
```

```
{'linestyle':'dashed','linewidth':'1'}).lines[0].set_color('black')

# Titre
plt.title('Distribution des âges des passagers du Titanic\n')

# Définition du nom des axes
plt.xlabel('Âge')
plt.ylabel('Nombre de passagers')

plt.show()
```



Cet exemple est l'occasion de montrer avec quelle facilité il est possible d'acquérir un jeu de données, le Titanic dans le cas présent, grâce à la commande `load_dataset()`. L'option `bins` a volontairement été fixée à 50 pour accentuer l'irrégularité des barres et, par contraste, souligner l'effet lissant de la courbe de densité, ici représentée en pointillés.

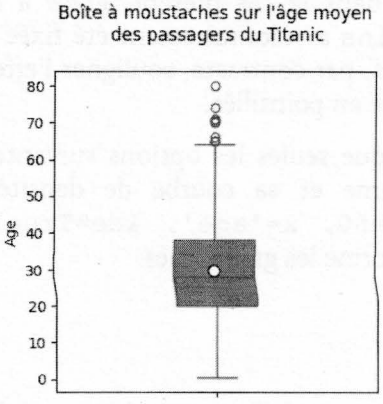
Précisons pour terminer sur ce graphique que seules les options suivantes sont nécessaires pour générer l'histogramme et sa courbe de densité : `sns.histplot(data=titanic, bins=50, x='age', kde=True)`, les autres servent uniquement à mettre en forme les graphiques.

Boîte à moustaches

La boîte à moustaches, également appelée boxplot ou diagramme en boîte, est l'un des graphiques les plus importants. Elle offre une quantité significative d'informations sur la distribution des données, notamment les quartiles (dont la médiane), les seuils de détection des valeurs aberrantes, la dispersion des données, ainsi que la présence éventuelle de valeurs extrêmes.

Reprenons l'exemple du Titanic pour présenter cet indicateur :

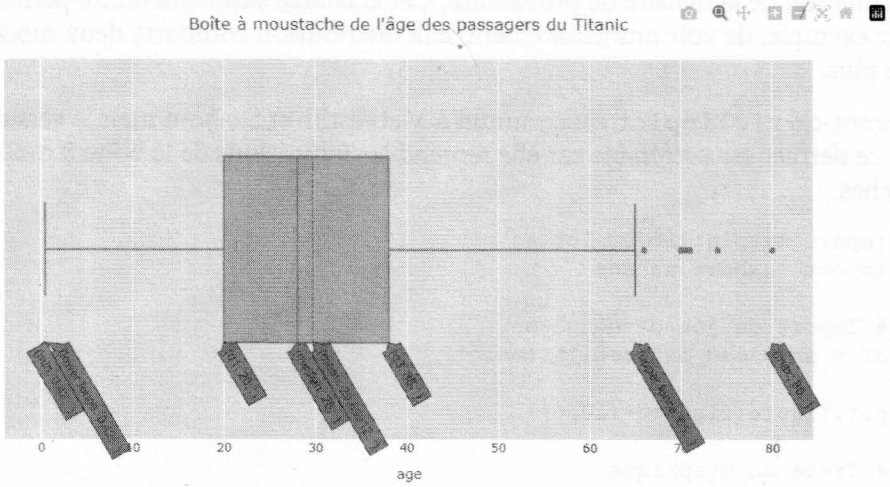
```
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

# Récupération du jeu de données du Titanic
titanic = sns.load_dataset('titanic')

plt.figure(figsize=(4, 4))

# Mise en forme de la moyenne
mean_formatting = {'marker':'o','markerfacecolor':'white',
'markeredgecolor':'black', 'markersize':'8'}

# Tracé du boxplot
sns.boxplot(data=titanic['age'], showmeans=True,color='#439cc8',
width=0.25, meanprops= mean_formatting)
plt.title("Boîte à moustaches sur l'âge moyen\n des passagers
du Titanic\n")
plt.ylabel('Age')
plt.show()
```



La construction du graphique ne présente guère de problème. Il n'y a que l'ajout de la moyenne qui demande quelques lignes de code supplémentaires, notamment pour la mise en forme. Le graphique est clair et bien lisible mais va manquer de précision si nous souhaitons connaître les valeurs exactes des seuils.

C'est ici que l'usage de Plotly-Express s'avère pertinent. Les commandes sont plus concises et l'interactivité nous apporte les valeurs recherchées lorsque nous passons la souris sur le graphique :

```
import plotly.express as px

# Récupération du jeu de données du Titanic
df = sns.load_dataset('titanic')

# Tracé du boxplot
fig = px.box(df, x = "age",  points="outliers",
color_discrete_sequence = ["#439cc8"])

# Ajout de la moyenne
fig.update_traces(boxmean=True)
fig.show()
```



L'interactivité offre ainsi une vraie valeur ajoutée pour analyser les données.

Dans notre cas, nous apprenons immédiatement que :

- l'âge moyen était de 29.69 ans ;
- l'âge médian était de 28 ans ;
- les âges s'étendent de 0.42 à 80 ans ;
- 25 % des passagers avaient moins de 20 ans ;
- 50 % des passagers avaient entre 20 et 38 ans ;
- 25 % avaient plus de 38 ans ;
- les passagers de plus de 65 ans, matérialisés par des points, étaient relativement rares (l'âge de chacun est affiché lors du survol de la souris).

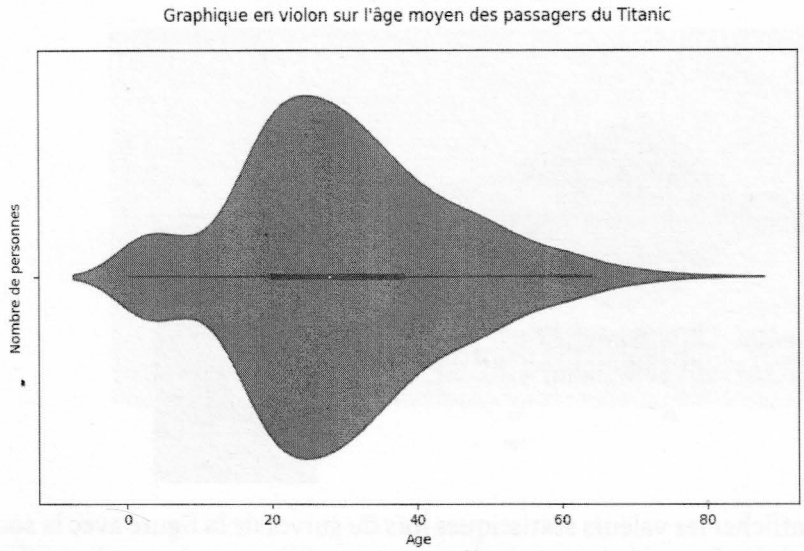
Diagramme en violon

Le diagramme en violon opère la synthèse entre la boîte à moustaches et le diagramme de densité. Alors que la boîte à moustaches offre une vue synthétique des quartiles et des valeurs aberrantes, le violon vient apporter l'information de densité de répartition des points qui lui fait défaut. Cette combinaison permet une représentation plus complète de la distribution des données, offrant à la fois une vue globale de la dispersion des données et une estimation de la densité de probabilité. Cet éclairage supplémentaire permet, par exemple, de voir immédiatement si la distribution comporte deux modes ou plus.

Le mot-clé `violinplot` est commun à Matplotlib et Seaborn mais la version de ce dernier est préférable car elle reprend les indications de la boîte à moustaches.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Import du jeu de données
df = sns.load_dataset('titanic')

plt.figure(figsize=(10,6))

# Tracé du graphique
sns.violinplot(x="age", data=df, color="#439cc8")

# Ajout des titres et des labels
```

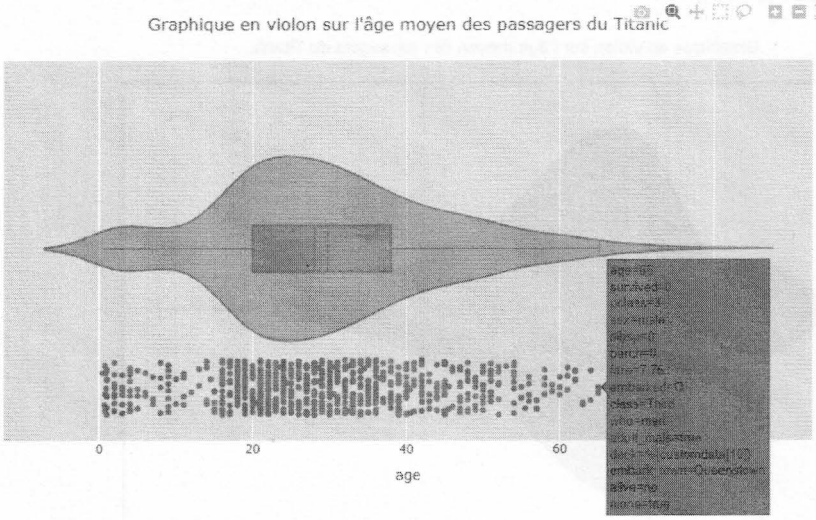
```
plt.title("Graphique en violon sur l'âge moyen des passagers
du Titanic\n")
plt.xlabel("Age")
plt.ylabel("Nombre de personnes")
plt.show()
```



Notons qu'une fois encore, Plotly-Express apporte une version plus adaptée à l'exploration grâce à son interactivité. Elle matérialise en plus les valeurs sous forme de points pour avoir une vision plus précise de la distribution en comparaison de la courbe de densité.

```
import plotly.express as px
import pandas as pd

# Import du jeu de données
df = sns.load_dataset('titanic')

# Création du violon plot avec Plotly
fig = px.violin(df, x="age",  box=True, points="all",
hover_data=df.columns, color_discrete_sequence=["#439cc8"])
fig.update_traces(meanline_visible=True)

# Ajout du titre
```

```
fig.update_layout(title="Graphique en violon sur l'âge moyen
des passagers du Titanic", title_x=0.5)

fig.show()
```



En plus d'afficher les valeurs statistiques lors du survol de la figure avec la souris, le détail de chaque point est également accessible, ce qui est très confortable pour étudier précisément certaines observations.

Après avoir examiné diverses approches pour analyser les distributions, voyons maintenant quels types de graphiques sont appropriés pour étudier la répartition des données dans le contexte des variables catégorielles.

3.2.2 Graphiques univariés pour les données catégorielles

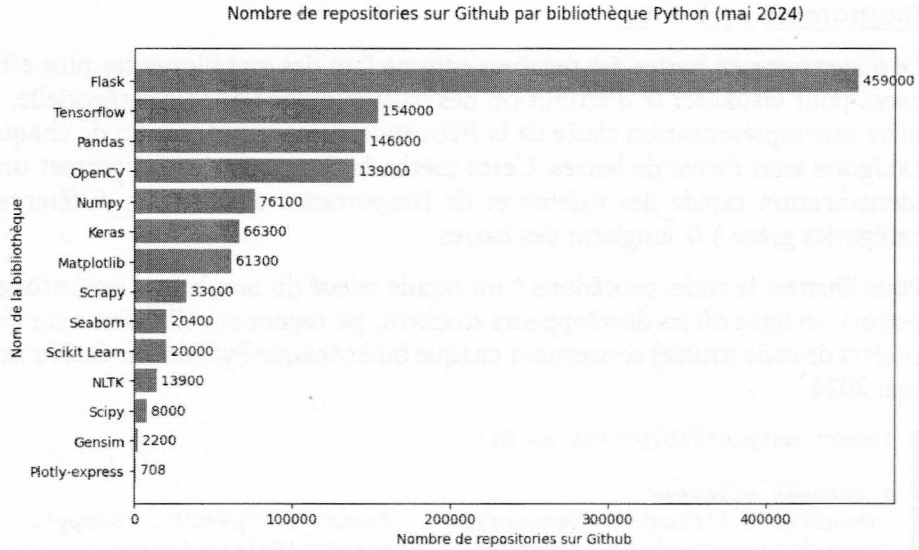
Dans le cadre des variables qualitatives, nous allons à présent nous familiariser avec les outils destinés à comprendre la répartition et la fréquence des différentes catégories présentes dans nos données.

Diagramme en barres

Le diagramme en barres est reconnu comme l'un des graphiques les plus efficaces pour visualiser la distribution des valeurs d'une variable catégorielle. Il offre une représentation claire de la fréquence ou de la proportion de chaque catégorie sous forme de barres. Cette méthode, bien que simple, permet une identification rapide des valeurs et de l'importance relative des différentes catégories grâce à la longueur des barres.

Pour illustrer le code, procédons à un rapide relevé du nombre de repositories (espace en ligne où les développeurs stockent, partagent et collaborent sur des projets de code source) concernant chaque bibliothèque Python sur GitHub en mai 2024.

```
import matplotlib.pyplot as plt

# Données relevées
libraries = ['Flask', 'Tensorflow', 'Pandas', 'OpenCV', 'Numpy',
'Keras', 'Matplotlib', 'Scrapy', 'Seaborn', 'Scikit Learn',
'NLTK', 'Scipy', 'Gensim', 'Plotly-express']

repository_count = [459000, 154000, 146000, 139000, 76100, 66300,
61300, 33000, 20400, 20000, 13900, 8000, 2200, 708]

# Tri des données par ordre décroissant
libraries = [x for _, x in sorted(zip(repository_count, libraries),
reverse=False)]
repository_count.sort(reverse=False)

# Création du graphique en barres
plt.figure(figsize=(10, 6))
plt.barh(libraries, repository_count, color='#439cc8')

# Ajout des titres et des labels
plt.xlabel('Nombre de repositories sur Github')
plt.ylabel('Nom de la bibliothèque')
plt.title('Nombre de repositories sur Github par bibliothèque
Python\n')

# Affichage des valeurs
for i, valeur in enumerate(repository_count):
    plt.text(valeur, i, f' {valeur}', ha='left',va='center')

plt.show()
```

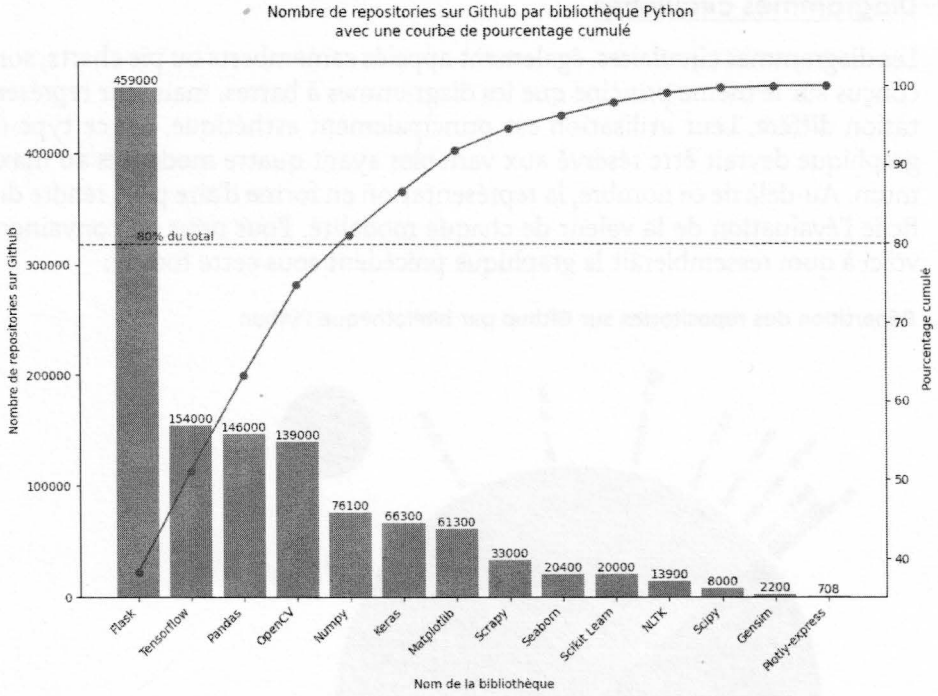


Cette petite étude ad hoc nous permet de constater que la librairie Flask, utilisée pour développer des applications web en Python, domine très largement les autres en termes de repositories sur GitHub. En queue de liste, nous retrouvons Plotly-Express avec un nombre très faible de 708 repositories.

Remarque

Le graphique a été intentionnellement trié par ordre décroissant, en commençant par le haut, afin de mettre en avant les librairies les plus populaires et faciliter l'accès à l'information.

Il ne faut jamais hésiter à aller plus loin et à ajouter des indicateurs pour approfondir nos connaissances. Ici, l'ajout d'un pourcentage cumulé permet de constater que les cinq bibliothèques les plus populaires représentent 80 % de l'ensemble.

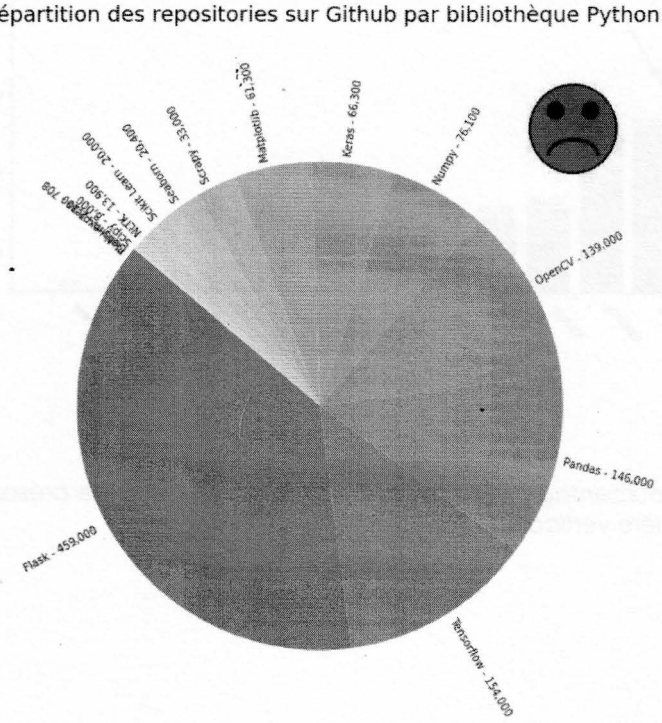


Remarque

Pour intégrer le pourcentage cumulé, nous sommes contraints de présenter les barres de manière verticale.

Diagrammes circulaires

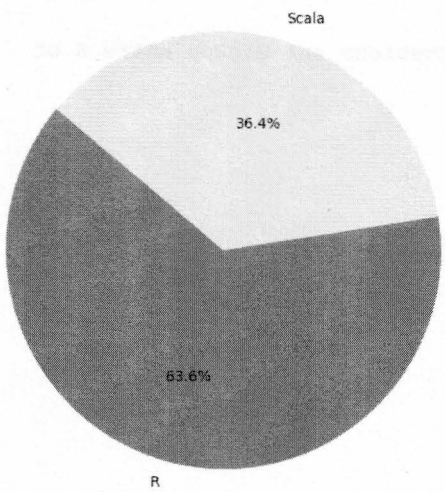
Les diagrammes circulaires, également appelés camemberts ou pie charts, sont conçus sur le même principe que les diagrammes à barres, mais leur représentation diffère. Leur utilisation est principalement esthétique, car ce type de graphique devrait être réservé aux variables ayant quatre modalités au maximum. Au-delà de ce nombre, la représentation en forme d'aire peut rendre difficile l'évaluation de la valeur de chaque modalité. Pour nous en convaincre, voici à quoi ressemblerait le graphique précédent sous cette forme :



Cette représentation est nettement moins lisible que la précédente. De plus, il est très difficile de discerner visuellement la différence, par exemple, entre Matplotlib et Keras.

Mais lorsqu'il est utilisé correctement, le pie chart est un outil de visualisation efficace. Si nous souhaitons, par exemple, représenter la part des discussions entre R et Scala sur GitHub en mai 2024, le pie chart délivre immédiatement l'information sous forme de pourcentages à partir des données fournies grâce au paramètre `autopct` :

```
import matplotlib.pyplot as plt

# Données issues de Github
num_discussions = [7000, 4000]
languages = ['R', 'Scala']

plt.figure(figsize=(8, 6))

# Création du pie chart
plt.pie(num_discussions, labels=languages, autopct='%1.1f%%',
startangle=140, colors=["#439cc8","#f2f2f2"])
plt.axis('equal')  # Aspect ratio égal pour que le pie chart soit
circulaire

# Titre
plt.title('Répartition des discussions sur GitHub entre R et Scala\n')

# Affichage du pie chart
plt.show()
```



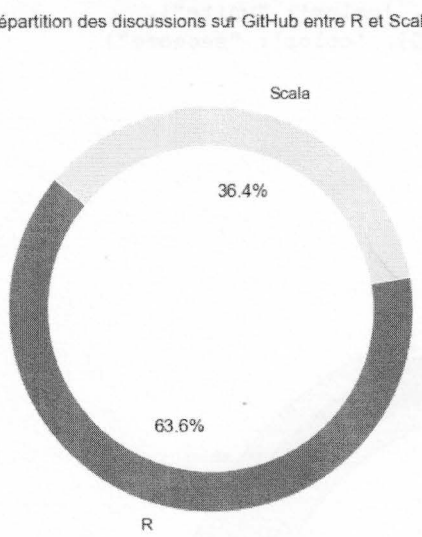
Notons qu'il est possible de moduler l'angle de départ du graphique grâce à l'option `startangle`.

Remarque

Les pie charts en 3D sont à proscrire, car ils peuvent fausser la perception de l'importance des catégories.

Selon les goûts ou les besoins, il est très simple de passer à un donut en insérant un cercle au milieu du pie chart :

```
import matplotlib.pyplot as plt

# Données
num_discussions = [7000, 4000]
languages = ['R', 'Scala']

plt.figure(figsize=(8, 6))

# Création du donut chart
plt.pie(num_discussions, labels=languages, autopct='%1.1f%%',
startangle=140, colors=["#439cc8","#f2f2f2"],
wedgeprops=dict(width=0.4))

# Ajout d'un cercle blanc au centre pour créer le trou
plt.gca().add_artist(plt.Circle((0,0),0.8,color='white'))

# Titre
plt.title('Répartition des discussions sur GitHub entre R et
Scala\n')

# Affichage du donut
plt.show()
```

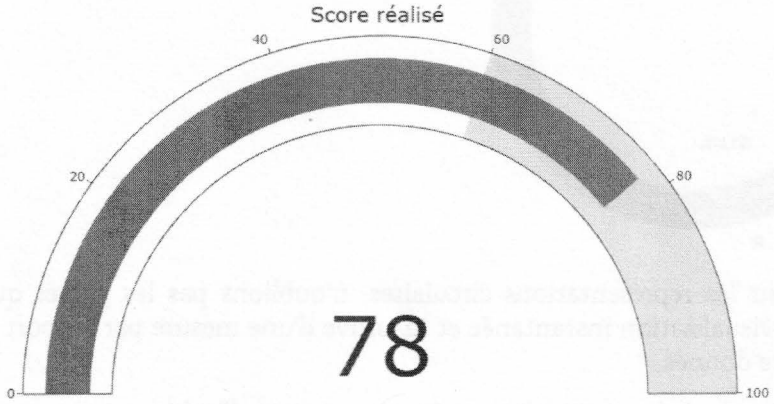


Pour finir sur les représentations circulaires, n'oublions pas les jauges qui offrent une visualisation instantanée et intuitive d'une mesure par rapport à une référence donnée.

Une façon simple de les construire consiste à passer par Plotly :

```
import plotly.graph_objects as go

#Paramètres
seuil = 60
pourcentage = 78

# Création de la jauge avec Plotly Graph Objects
fig = go.Figure()

# Ajout de la jauge
fig.add_trace(go.Indicator(
    mode="gauge+number", value=pourcentage,
    title={"text": "Score réalisé"},
    domain = {'x': [0, 1], 'y': [0, 1]},
    gauge = {
        'axis': {'range': [None, 100]},
        'bar': {'color': "#439cc8"},
        'steps': [
```

```
            {'range': [0, seuil], 'color': "white"},
            {'range': [seuil, 100], 'color': "#eeeeee"}
        ]
    }
))


# Affichage de la jauge
fig.show()
```



Remarque

Nous avons dû recourir directement à Plotly, car Plotly Express, comme Matplotlib et Seaborn, ne possède pas de fonction intégrée pour les jauges et nécessite beaucoup de lignes de code.

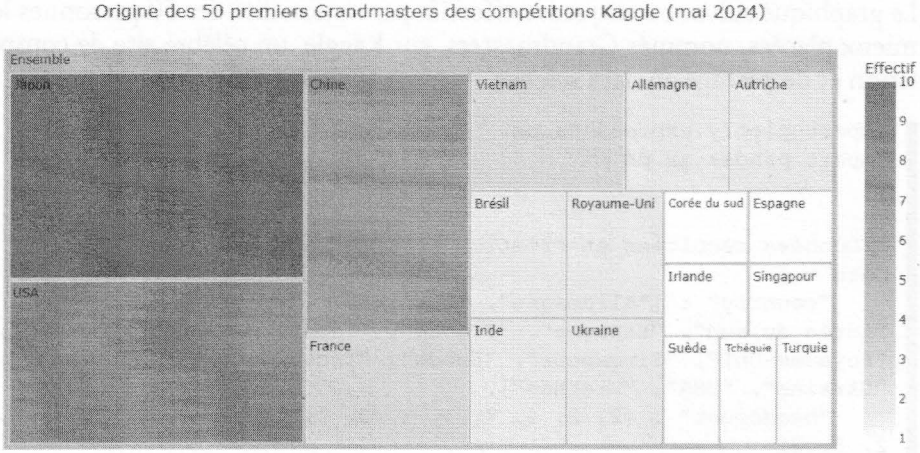
Treemap

Les treemaps sont une méthode de représentation visuelle d'une variable catégorielle, où chaque catégorie est représentée par un quadrilatère dont la taille est proportionnelle à son importance. Contrairement aux diagrammes circulaires, les treemaps sont recommandés lorsque le nombre de catégories est élevé.

Pour créer un treemap avec Matplotlib, l'installation du module squarify est nécessaire. Cependant, une approche plus simple consiste à utiliser Plotly-express, qui permet de créer des treemaps directement sans nécessiter d'installation supplémentaire.

Le graphique suivant indique les effectifs par nationalité des 50 personnes les mieux placées, nommés Grandmasters, sur Kaggle, un célèbre site de compétition et de partage en data science.

```
import plotly.express as px
import pandas as pd


# Données récoltées et création du DataFrame
data = {
    "country" : ["Allemagne", "Autriche", "Brésil", "Chine",
"Corée du sud", "Espagne", "France", "Inde", "Irlande", "Japon",
"Royaume-Uni", "Singapour", "Suède", "Tchéquie", "Turquie",
"Ukraine", "USA", "Vietnam"],
    "headcount" : [2, 2, 2, 7, 1, 1, 3, 2, 1, 10, 2, 1, 1, 1, 1,
2, 8, 3]
}

df = pd.DataFrame(data)

# Création d'une palette personnalisée
custom_color_list = [
    [0, "rgb(255,255,255)"],
    [1, "rgb(67,156,200)"]
]

# Tracé de la figure
fig = px.treemap(df, path=[px.Constant('Ensemble'),'country'],
values='headcount', color=" headcount ",
color_continuous_scale=custom_color_list)

# Ajout du titre centré
fig.update_layout(title_text='Origine des 50 premiers
Grandmasters des compétitions Kaggle', title_x=0.5)

# Affichage du treemap
fig.show()
```



Nuage de mots

Le nuage de mots est une représentation visuelle où les mots sont affichés proportionnellement à leur fréquence dans un corpus de texte. Bien que conçu initialement pour analyser des corpus textuels, il peut également être appliqué à des variables catégorielles avec de nombreuses modalités de réponses. Il convient néanmoins de noter que cette forme de visualisation est plus esthétique que scientifique, car la différence d'importance basée sur la taille des mots peut être plus difficile à apprécier.

Les nuages de mots nécessitent l'installation du module wordcloud :

```
pip install wordcloud
```

Et voici le code permettant de le générer basé sur les patronymes des passagers du Titanic :

```
import pandas as pd
from wordcloud import WordCloud
import matplotlib.pyplot as plt

# Chargement du dataset Titanic
titanic_df = pd.read_csv("https://raw.githubusercontent.com/
datasciencedojo/datasets/master/titanic.csv")

# Sélection uniquement de la colonne 'Name'
names = titanic_df['Name']
```

```
# Concaténation de tous les noms en une seule chaîne de caractères
text_names = ' '.join(names)

# Création du nuage de mots
wordcloud = WordCloud(width=800, height=400,
background_color='white') # Une palette peut être définie grâce
à l'option colormap, par exemple colormap='Blues'

# Code supplémentaire nécessaire pour définir une couleur
wordcloud.color_func = lambda *args, **kwargs: (67, 156, 200)
wordcloud = wordcloud.generate_from_text(text_names)

# Affichage du nuage de mots
plt.figure(figsize=(10, 5))
plt.title("Nuage de mots des noms et prénoms des passagers
du Titanic\n", color="#333333")
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

Nuage de mots des noms et prénoms des passagers du Titanic

3.2.3 Récapitulatif

Avant de passer aux visualisations bivariées et multivariées, voici un tableau récapitulatif de tous les grands types de graphiques univariés :

Nom	Histogramme	Boxplot	Violon	Barres	Graphiques circulaires	Treemap	Nuage de mots
Forces	Simple	Distribution par quartiles et valeurs aberrantes	Combinaison de boxplot et KDE	Simple et précis	Simple à comprendre	Précis et autorise plusieurs niveaux de détail	Visuel et simple à comprendre mais moins précis
Données	Tout type de volume	Volume moyen à grand	Volume moyen à grand	2 à 30 mentions	4 mentions maximum	Pour tout nombre de mentions	Analyse de texte
Variables	Numériques			Catégorielles			

LES GRAPHIQUES UNIVARIÉS

3.3 Les graphiques bivariés et multivariés

Les graphiques multivariés constituent un aspect essentiel de l'analyse exploratoire des données en offrant une représentation visuelle riche et complexe des relations entre plusieurs variables simultanément. Contrairement aux graphiques univariés qui se concentrent sur la distribution ou la composition de chaque variable, les graphiques multivariés permettent d'explorer les interactions et de les comparer. Pour présenter ces différents outils, nous commencerons par étudier ceux portant sur un même type de variable, puis les graphiques portant sur deux variables de nature différente pour finir sur les visualisations multivariées.

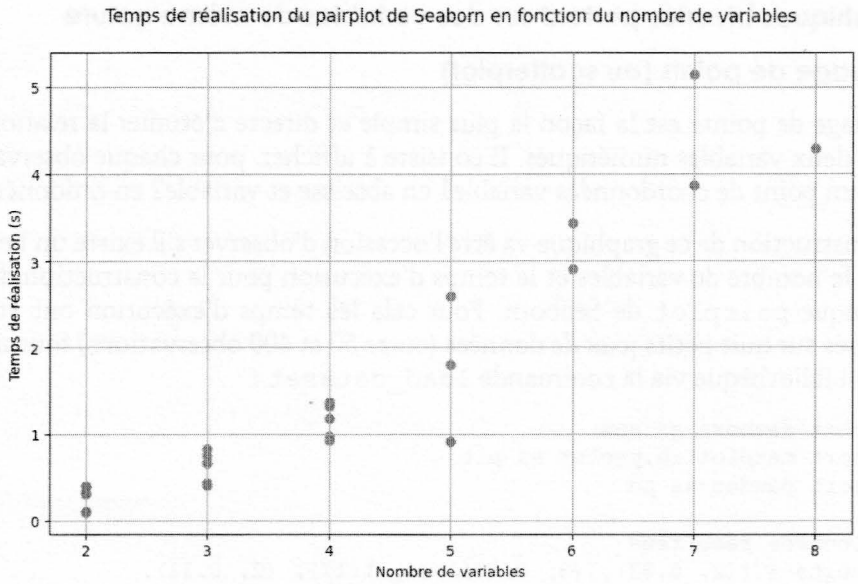
3.3.1 Graphiques bivariés portant sur des variables de même nature

Le nuage de points (ou scatterplot)

Le nuage de points est la façon la plus simple et directe d'étudier la relation entre deux variables numériques. Il consiste à afficher, pour chaque observation, un point de coordonnées variable1 en abscisse et variable2 en ordonnée.

La construction de ce graphique va être l'occasion d'observer s'il existe un lien entre le nombre de variables et le temps d'exécution pour la construction du graphique `pairplot` de Seaborn. Pour cela les temps d'exécution ont été mesurés sur huit petits jeux de données (entre 50 et 400 observations) fournis par la bibliothèque via la commande `load_dataset` :

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Données récoltées
results = [(2, 0.32), (3, 0.66), (4, 1.17), (2, 0.31),
           (3, 0.66), (4, 1.37), (5, 1.8), (6, 2.91),
           (7, 3.87), (8, 4.28), (2, 0.33), (3, 0.74),
           (4, 1.32), (5, 2.59), (6, 3.43), (7, 5.14),
           (2, 0.39), (3, 0.83), (4, 0.92), (5, 0.92),
           (2, 0.4), (2, 0.1), (3, 0.4), (2, 0.4),
           (3, 0.4), (4, 0.98), (2, 0.11)]

df = pd.DataFrame(results, columns=['nbvar', 'execution_time'])

plt.figure(figsize=(10, 6))

# Tracé du scatterplot
plt.scatter(df["nbvar"], df["execution_time"], c='#439cc8')

# Mise en forme
plt.xlabel('Nombre de variables')
plt.ylabel('Temps de réalisation (s)')
plt.title('Temps de réalisation du pairplot de Seaborn en fonction
du nombre de variables\n')
plt.grid(True)
plt.show()
```



Le graphique est très simple à construire en Matplotlib grâce à la commande `scatter` qui nécessite juste d'indiquer les variables à représenter.

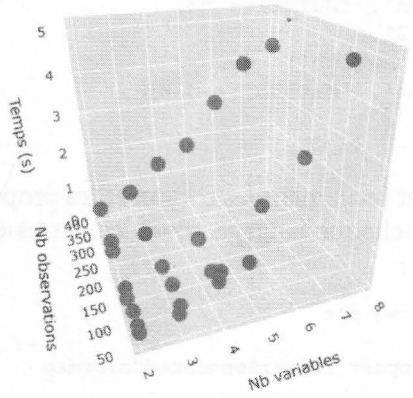
Le scatter3d (la version 3D du scatterplot)

Le surface plot est la version en trois dimensions du scatter plot offrant la possibilité de constater les liens entre trois variables numériques. C'est l'occasion de reprendre les données précédentes et de rajouter le nombre d'observations de chaque jeu de données. Ici le recours à Plotly-express est indispensable car la version 3D de Matplotlib n'offre pas la possibilité de faire pivoter le graphique. Et un graphique en trois dimensions sans cette possibilité ne sert à rien car la perspective imposée ne permet pas l'étude correcte des relations.

```
import pandas as pd
import plotly.express as px

# Données récoltées et construction du DataFrame
results = [(2, 0.32, 150), (3, 0.66, 150), (4, 1.17, 150),
(2, 0.31, 398), (3, 0.66, 398), (4, 1.37, 398), (5, 1.8, 398),
(6, 2.91, 398), (7, 3.87, 398), (8, 4.28, 398), (2, 0.33, 51),
(3, 0.74, 51), (4, 1.32, 51), (5, 2.59, 51), (6, 3.43, 51),
(7, 5.14, 51), (2, 0.39, 90), (3, 0.83, 90), (4, 0.92, 90),
```

```
(5, 0.92, 90),
(2, 0.4, 272), (2, 0.1, 274), (3, 0.41, 274), (2, 0.4, 60),
(3, 0.43, 60), (4, 0.98, 60), (2, 0.11, 144)]

df = pd.DataFrame(results, columns=['nbvar', 'execution_time',
"n_observations"])


# Création du scatterplot en 3D avec Plotly-express
fig = px.scatter_3d(df, x='nbvar', y='n_observations',
                    z='execution_time',
                    color_discrete_sequence=['#439cc8'],
                    labels={'nbvar':'Nb variables',
                            'n_observations':'Nb observations',
                            'execution_time':'Temps (s)'})
fig.update_layout(title_text="Temps de réalisation du pairplot \
en fonction du nombre de variables et d'observations",
title_x=0.5)

# Affichage
fig.show()
```

Temps de réalisation du pairplot en fonction du nombre de variables et d'observations



À l'issue de quelques tentatives de rotations pour trouver la perspective idéale, la figure permet de constater que le nombre d'observations de chaque jeu de données ne semble pas influer sur le temps de réalisation du graphique.

Les diagrammes groupés et empilés

Les barres, qu'elles soient groupées ou empilées, sont le moyen le plus direct pour étudier la composition d'une variable qualitative en fonction d'une autre.

Le choix de l'un ou de l'autre dépend des besoins en termes d'analyse. Pour illustrer ces différences, nous allons à nouveau utiliser GitHub pour relever le nombre de repositories liés à des algorithmes de machine learning pour des langages de programmation autres que Python et R.

Voici les données relevées sur GitHub que nous utiliserons :

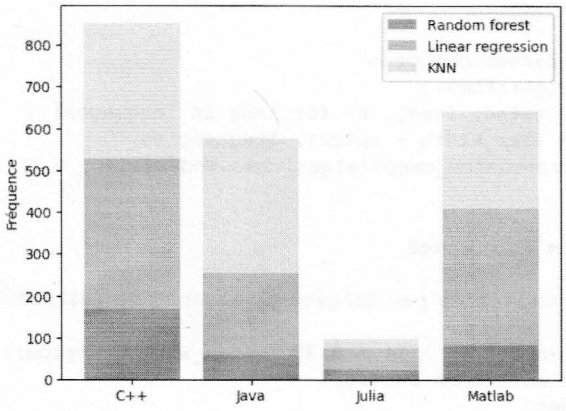
```
# Données
algorithms = ['Random forest', 'Linear regression', 'KNN']
languages = ['C++', 'Java', 'Julia', 'Matlab']
dic = {
    ('Random forest', 'C++'): 171,
    ('Linear regression', 'C++'): 358,
    ('KNN', 'C++'): 324,
    ('Random forest', 'Java'): 58,
    ('Linear regression', 'Java'): 197,
    ('KNN', 'Java'): 323,
    ('Random forest', 'Julia'): 24,
    ('Linear regression', 'Julia'): 53,
    ('KNN', 'Julia'): 21,
    ('Random forest', 'Matlab'): 83,
    ('Linear regression', 'Matlab'): 328,
    ('KNN', 'Matlab'): 231
}
```

Si nous souhaitons visualiser facilement les totaux et les proportions relatives de chaque algorithme pour chaque langage, alors les barres empilées sont la solution à mettre en œuvre :

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import LinearSegmentedColormap

# Calcul des totaux par algorithme
totals = {algo: sum(dic.get((algo, lang), 0) for lang in languages) for
algo in algorithms}

# Création d'une palette dégradée personnalisée
```

```
start_color = '#439cc8'
end_color = '#eeeeee'
custom_colors = [start_color, end_color]
custom_cmap = LinearSegmentedColormap.from_list('custom',
custom_colors)

# Création du cadre graphique
fig, ax = plt.subplots()

# Création des barres empilées
bottom = [0] * len(languages)
for algo in algorithms:
    frequencies = [dic.get((algo, lang), 0) for lang in languages]
    bars = ax.bar(languages,
                  frequencies,
                  bottom=bottom,
                  label=algo,
                  color=custom_cmap((algorithms.index(algo) + 1)/
len(algorithms)))
    bottom = [bottom[i] + frequencies[i] for i in range(len(languages))]

# Ajout des légendes et des étiquettes
ax.set_ylabel('Fréquence')
ax.set_title('Nombre de repositories par langage et algorithme (Github /
mai 2024)\n')
ax.legend()

# Affichage du barchart
plt.show()
```

Nombre de repositories par langage et algorithme (Github / mai 2024)



C++ arrive en tête des outsiders avec un peu plus de 800 repositories liés à ces trois algorithmes, soit huit fois plus que Julia.

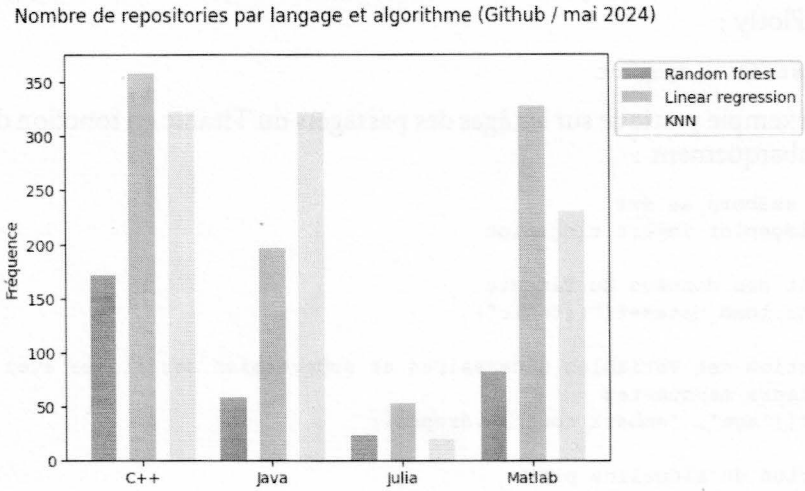
Remarque

Pour information, Python, qui n'est pas représenté, compte 13 300 repositories pour ces trois algorithmes.

Si, en revanche, nous souhaitons comparer directement les valeurs absolues des différents algorithmes au sein de chaque langage, alors il faut choisir les barres groupées :

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import LinearSegmentedColormap

# Création d'une palette dégradée personnalisée
start_color = '#439cc8'
end_color = '#eeeeee'
custom_colors = [start_color, end_color]
custom_cmap = LinearSegmentedColormap.from_list('custom',
custom_colors)

# Calcul des positions des barres
bar_width = 0.2
space = 0.1
num_bars = len(languages)
positions = np.arange(num_bars)

# Création du cadre graphique
fig, ax = plt.subplots()

# Création du graphique à barres groupées
for i, algo in enumerate(algorithms):
    frequencies = [dic.get((algo, lang), 0) for lang in languages]
    ax.bar(positions + i * (bar_width + space), frequencies,
bar_width, label=algo, color=custom_cmap((algorithms.index(algo) + 1) /
len(algorithms)))

# Ajout des légendes et des étiquettes
ax.set_ylabel('Fréquence')
ax.set_title('Nombre de repositories par langage et algorithme (Github /
mai 2024)\n')
ax.set_xticks(positions + (num_bars - 1) * 0.33 * (bar_width + space))
# 0.33 pour que le langage soit bien central
ax.set_xticklabels(languages)
```

```
# Déplacement de la légende en dehors du graphique
ax.legend(loc='upper left', bbox_to_anchor=(1, 1))

# Affichage
plt.show()
```



La visualisation précédente offre ainsi une meilleure perception du nombre de dépôts entre les différents langages.

3.3.2 Graphiques bivariés portant sur des variables de natures différentes

Étudier les relations entre des variables de nature différente nécessite d'autres graphiques. Dans ce contexte, nous allons réutiliser certains graphiques déjà étudiés dans la partie univariée, mais cette fois-ci en les dupliquant par modalité. Il est à noter que certains graphiques conserveront le même nom, tandis que d'autres, tels que le ridgeline plot, auront des noms différents mais seront en réalité des estimations de densité de noyau (KDE) pour chaque modalité.

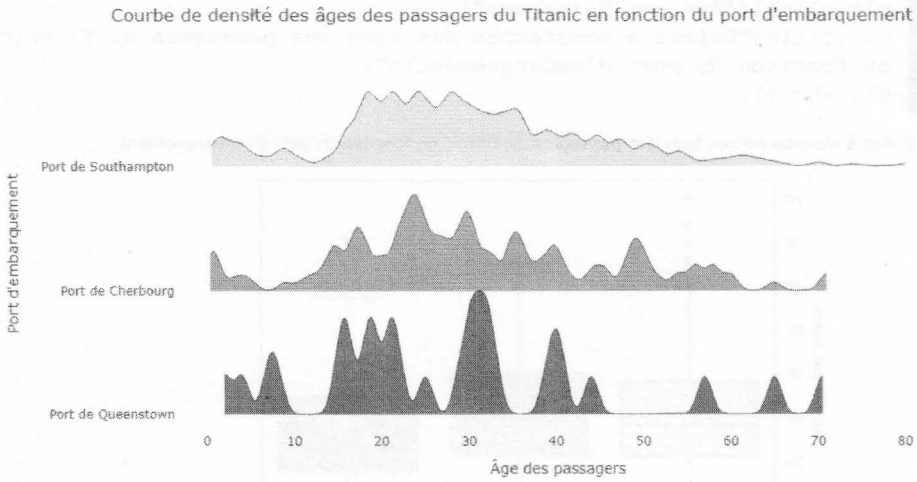
Le ridgeline plot

Le ridgeline plot est une solution pertinente pour visualiser les courbes de densité de probabilité d'une variable numérique pour chaque modalité d'une variable catégorielle. L'utilisation du module ridgeplot est ici conseillée car la syntaxe est concise et propose l'interactivité en tant que module réalisé à partir de Plotly :

```
pip install ridgeplot
```

Voici un exemple pratique sur les âges des passagers du Titanic en fonction du port d'embarquement :

```
import seaborn as sns
from ridgeplot import ridgeplot

# Import des données du Titanic
df = sns.load_dataset("titanic")

# Sélection des variables nécessaires et suppression des lignes avec
des valeurs manquantes
df = df[['age', 'embark_town']].dropna()

# Création du ridgeline plot
fig = ridgeplot(
    samples=[df[df['embark_town'] == port]['age'].tolist() for port in
df['embark_town'].unique()],
    colormode="row-index",
    colorscale=[(0.0, '#439cc8'), (0.5, '#98c5db'), (1.0, '#eeeeee')],
    labels=[f"Port de {port}" for port in df['embark_town'].unique()],
    bandwidth=0.8, # Souplesse du lissage
    spacing=1 # Espacement entre les courbes
)

# Mise en forme
fig.update_layout(
    plot_bgcolor="white",
    yaxis_title="Port d'embarquement",
    xaxis_title="Âge des passagers",
    showlegend=False,
    title_text="Courbes de densité des âges des passagers du Titanic en
fonction du port d'embarquement", title_x=0.5
)

# Affichage
fig.show()
```



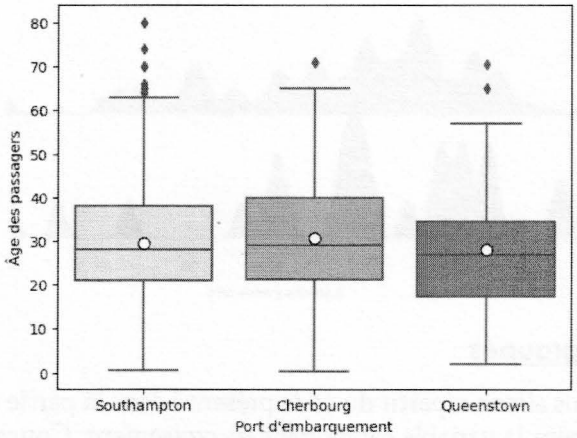
Boxplot et violons groupés

Pour les boxplots, nous allons repartir du code présenté dans la partie univariée en rajoutant simplement la variable catégorielle de croisement. Concernant les graphiques bivariés, Seaborn s'avère très pratique pour les générer.

```
import seaborn as sns

# Définition des couleurs
colors = ['#eeeeee', '#98c5db', '#439cc8']

# Import des données du Titanic et suppression des valeurs
manquantes
df = sns.load_dataset("titanic").dropna(subset=['age',
'embark_town'])

# Mise en forme de la moyenne
mise_en_forme_moyenne = {'marker':'o','markerfacecolor':
'white','markeredgecolor':'black', 'markersize':'8'}

# Création du boxplot avec la palette spécifiée
sns.boxplot(data=df, x='embark_town', y='age', palette=colors,
showmeans=True,meanprops=mise_en_forme_moyenne)

# Mise en forme
plt.xlabel("Port d'embarquement")
```

```
plt.ylabel("Âge des passagers")
plt.title("Boîtes à moustaches des âges des passagers du Titanic
en fonction du port d'embarquement\n")
plt.show()
```

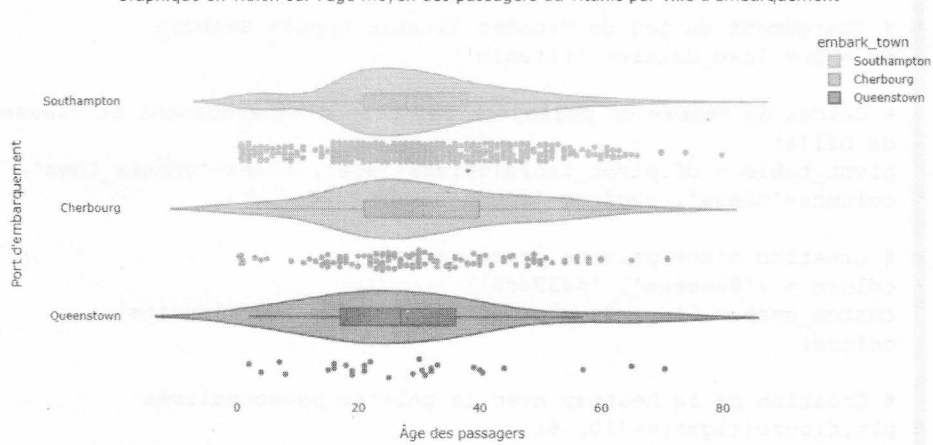
Boîtes à moustache des âges des passagers du Titanic en fonction du port d'embarquement



De la même manière que pour la partie univarié, nous repartons de l'exemple en Plotly-express pour réaliser le diagramme en violon :

```
import plotly.express as px
import seaborn as sns

# Import du jeu de données
df = sns.load_dataset('titanic')

# Définition de la palette personnalisée
colors = ['#cccccc', '#98c5db', '#439cc8']

# Tracé du graphique en violon
fig = px.violin(
    df, x="age", y="embark_town", box=True, points="all",
hover_data=df.columns,
    color="embark_town", color_discrete_sequence=colors
)

# Ajout de la moyenne
fig.update_traces(meanline_visible=True)
```

```
# Mise en forme
fig.update_layout(
    title="Graphique en violon sur l'âge moyen des passagers du
Titanic par ville d'embarquement",
    title_x=0.5,
    paper_bgcolor='white',  # Couleur de fond de la zone
extérieure
    plot_bgcolor='white',     # Couleur de fond de la zone de
traçage
    violingap=0.3,           # Espacement entre les violons
    violingroupgap=0.1,       # Espacement entre les groupes de
violons
    yaxis_title="Port d'embarquement", # Titre axe y
    xaxis_title="Âge des passagers", #Titre axe x
)

fig.show()
```



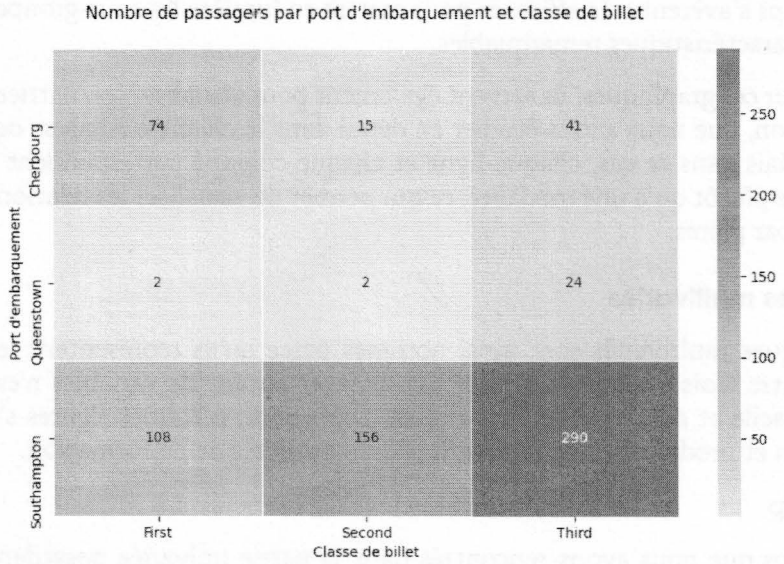
Ici encore, l'interactivité couplée à la quantité d'informations proposée offre un réel confort pour étudier en détail la distribution par catégorie.

Heatmaps

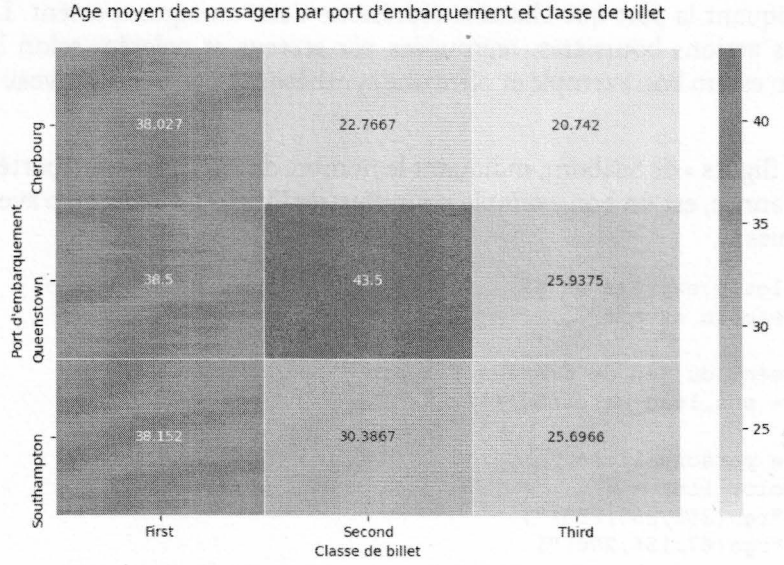
Les heatmaps sont un autre pilier de la visualisation. Elles permettent, grâce à l'usage de couleurs, de montrer clairement où les croisements entre les catégories sont plus forts ou plus faibles. Bien que la majorité des variables analysées soient de même nature, l'utilisation d'une variable numérique comme variable d'analyse est également possible, ce qui justifie sa place dans cette catégorie.

La construction de la matrice demande peu de lignes de codes et offre une vision immédiate de la répartition des données. L'usage de Seaborn est ici recommandé pour sa concision. Voici le code nécessaire à l'affichage du nombre passagers par port et par classe de billet :

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from matplotlib.colors import LinearSegmentedColormap

# Chargement du jeu de données Titanic depuis Seaborn
df = sns.load_dataset('titanic')

# Calcul du nombre de passagers par port d'embarquement et classe
de billet
pivot_table = df.pivot_table(values='age', index='embark_town',
columns='class', aggfunc='count')

# Création d'une palette personnalisée
colors = ['#eeeeee', '#439cc8']
custom_cmap = LinearSegmentedColormap.from_list('custom',
colors)

# Création de la heatmap avec la palette personnalisée
plt.figure(figsize=(10, 6))
sns.heatmap(pivot_table, annot=True, cmap=custom_cmap,
linewidths=.5, fmt='g')
plt.title("Nombre de passagers par port d'embarquement et classe
de billet\n")
plt.xlabel("Classe de billet")
plt.ylabel("Port d'embarquement")
plt.show()
```



En remplaçant count par mean dans `aggfunc`, nous obtenons directement une autre visualisation sur l'âge moyen :



Les heatmaps s'avèrent très efficaces pour mettre en lumière les sous-groupes ayant des caractéristiques remarquables.

Pour finir sur ces graphiques, ils servent également pour visualiser les matrices de corrélation, que nous allons étudier en détail dans le chapitre Analyse des données. Mais dans ce cas, chaque ligne et chaque colonne correspondent à une variable plutôt qu'à une modalité, ce qui permet de visualiser les relations entre elles par paires.

3.3.3 Graphiques multivariés

Les graphiques multivariés sont ainsi nommés parce qu'ils représentent les relations entre trois variables ou plus. Représenter autant de variables n'est pas chose facile et demande de la pratique. Cependant, certaines figures s'y prêtent bien et produisent une représentation intelligible de l'information.

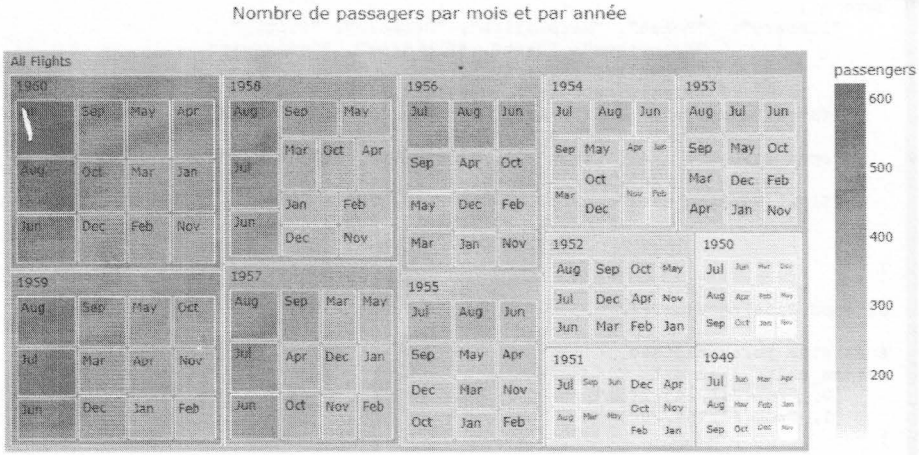
Le treemap

Les treemaps que nous avons rencontrés dans la partie univariée possèdent une caractéristique supplémentaire : la hiérarchie, qui permet de gérer différents niveaux d'imbrication et donc de représenter plusieurs variables. Ce type de visualisation est très efficace pour afficher un grand nombre de modalités tout en indiquant la part que chacune représente dans un regroupement. Le treemap des actions boursières, regroupées par secteur et colorées selon la tendance, en est un bon exemple et offre une synthèse avec différents niveaux de lecture.

Le dataset « flights » de Seaborn, indiquant le nombre de passagers transportés par mois et année, est un bon exemple pour illustrer l'intérêt du treemap avec des hiérarchies :

```
import plotly.express as px
import seaborn as sns

# Chargement du jeu de données flights
flights = sns.load_dataset('flights')

# Palette personnalisée
custom_color_list = [
    [0, "rgb(255,255,255)"],
    [1, "rgb(67,156,200)"]
```

```
]

# Création du treemap
fig = px.treemap(
    flights,
    path=[px.Constant('All Flights'), 'year', 'month'],
# Hiérarchie: 'All Flights' > year > month
    values='passengers',
    color='passengers',  # Echelle de couleurs basée sur le
nombre de passagers
    color_continuous_scale=custom_color_list
)

# Titre
fig.update_layout(
    title_text='Treemap du nombre de passagers par mois et
par année',
    title_x=0.5
)

# Afficher le treemap
fig.show()
```



Le graphique précédent offre ainsi une bonne synthèse permettant d'identifier et de comparer le nombre de passagers par mois ou par année.

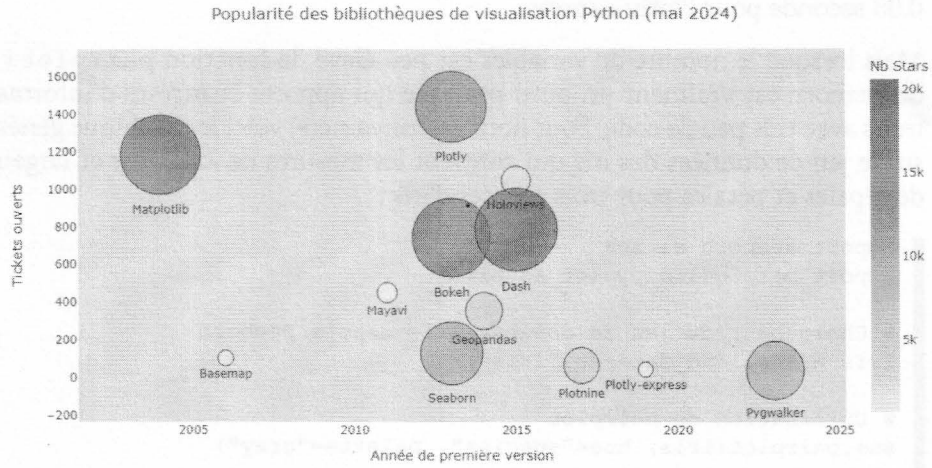
Bubble chart

Le bubble chart, ou graphique à bulles, est une extension du nuage de points. En ajustant la taille des points, il permet d'ajouter une troisième dimension quantitative. Il est même possible d'incorporer une quatrième dimension en coloriant les bulles en fonction d'une variable qualitative. Les quatre dimensions offertes sont d'une grande aide pour synthétiser des situations à multiples facteurs. Il est, cependant, préférable de l'utiliser lorsqu'il n'y a pas trop de points, car un trop grand nombre peut nuire à la lecture du graphique.

Dans le but d'illustrer ce graphique tout en restant dans la thématique des bibliothèques Python liées à la visualisation, nous allons les représenter en fonction de leur date de première version en abscisse et du nombre de tickets ouverts sur GitHub en ordonnée (un ticket est une demande d'amélioration ou le signalement d'un bug). La taille des bulles, elle, sera proportionnelle au nombre d'étoiles attribuées sur GitHub. Les données ont été recueillies sur le site https://pypi.org/ qui recense les packages Python.

```
import plotly.express as px
import pandas as pd

# Données récoltées
data = {
    "library": ["Bokeh", "Matplotlib", "Seaborn", "Plotly",
                "Holoviews", "Dash", "Plotnine", "Geopandas",
                "Pygwalker", "Mayavi",
                "Plotly-express", "Basemap"],
    "favorite_stars": [18893, 19410, 12011, 15406, 2634, 20627,
                       3847, 4238, 10303, 1256, 700, 765],
    "open_issues": [743, 1183, 132, 1438, 1045, 785, 67,
                    355, 40, 452, 46, 105],
    "first_version_date": [2013, 2004, 2013, 2013, 2015,
                           2015, 2017, 2014, 2023, 2011,
                           2019, 2006]
}

df = pd.DataFrame(data)

# Palette personnalisée
custom_color_list = [
    [0, "rgb(245,245,245)"],
    [1, "rgb(67,156,200)"]
]

# Création du bubble chart
fig = px.scatter(df,
    x="first_version_date",
    size="favorite_stars",
    y="open_issues",
    hover_name="library",
```

```
    text="library",
    labels={"first_version_date": "Année de première version",
            "favorite_stars": "Nb Stars",
          "open_issues": "Tickets ouverts"},
    size_max=60,
    color="favorite_stars", # Légende pour le nombre d'étoiles
    color_continuous_scale=custom_color_list) # Dégradé de couleurs

# Disposition du texte et cerclage de noir des bulles.
fig.update_traces(textposition='bottom center',marker_line_color='black')

# Titre
fig.update_layout(
    title_text='Popularité des bibliothèques de visualisation Python (mai 2024)',
    title_x=0.5
)

# Suppression des grilles horizontales et verticales
fig.update_xaxes(showgrid=False, zeroline=False)
fig.update_yaxes(showgrid=False, zeroline=False)

# Affichage du graphique
fig.show()
```



En jouant à la fois sur la couleur et la taille des bulles, les bibliothèques les plus populaires s'imposent visuellement comme de grosses bulles foncées. Leurs coordonnées nous renseignent immédiatement sur leur ancienneté en abscisses et le nombre de tickets ouverts en ordonnées. Nous obtenons ainsi un bon résumé de leurs caractéristiques.

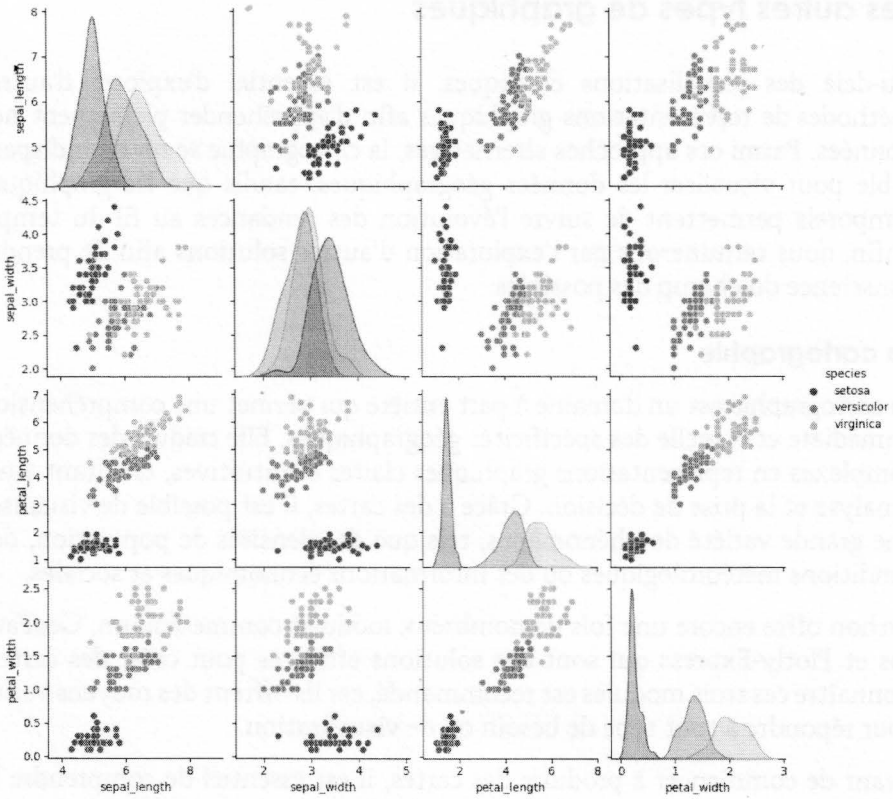
La matrice de nuages de points

La matrice de nuages de points génère un graphique global représentant les relations entre les variables par paires, tout en permettant de colorier les points en fonction d'une variable catégorielle. C'est un outil essentiel à utiliser dès le début de l'analyse, car il offre une vue d'ensemble des relations entre les variables et permet de saisir rapidement les tendances et les corrélations présentes dans les données.

Les fonctions proposées par Seaborn et Plotly-Express, respectivement `plairplot()` et `scatter_matrix`, proposent ce type de graphique en une ligne de commande. La seule limite à l'usage de cette visualisation réside dans le nombre de variables, car au-delà d'une douzaine, le temps nécessaire augmente considérablement. Un test a été réalisé sur le dataset Wine disponible à partir de la librairie Scikit-learn. Pour afficher la matrice de nuages de points de 13 variables, cela prend 24.68 secondes sur Seaborn contre 0.03 seconde pour Plotly-express.

Mais lorsque le nombre de variables est peu élevé, la fonction `pairplot()` de Seaborn est vraiment un outil pratique qui apporte beaucoup d'informations avec très peu de code. Pour nous en convaincre, voici le graphique généré sur le jeu de données des iris qui contient les mesures de longueur et largeur de sépales et pétales pour trois espèces d'iris :

```
import seaborn as sns
import matplotlib.pyplot as plt

# Chargement du jeu de données Iris depuis Seaborn
iris = sns.load_dataset("iris")

# Utilisation de pairplot
sns.pairplot(iris, hue="species", palette="gray")

# Affichage du graphique
plt.show()
```



Dans le code précédent, une seule ligne de commande suffit pour produire un point complet sur l'ensemble des variables numériques de manière individuelle (en diagonale) et par couple de variables (hors diagonale). L'option `hue` est là pour colorier les points selon une variable définie, facilitant ainsi l'exploration et la connaissance des données.

Grâce au pairplot, nous remarquons immédiatement, par exemple, que :

- Les espèces semblent facilement identifiables selon la longueur ou la largeur des pétales.
- Il existe une relation linéaire entre la longueur et la largeur des pétales.
- L'espèce Setosa se détache bien des deux autres, quelles que soient les caractéristiques.

3.4 Les autres types de graphiques

Au-delà des visualisations classiques, il est essentiel d'explorer d'autres méthodes de représentations graphiques afin d'appréhender pleinement nos données. Parmi ces approches alternatives, la cartographie se révèle indispensable pour visualiser les données géographiques, tandis que les graphiques temporels permettent de suivre l'évolution des tendances au fil du temps. Enfin, nous terminerons par l'exploration d'autres solutions afin de prendre conscience du champ des possibles.

3.4.1 La cartographie

La cartographie est un domaine à part entière qui permet une compréhension immédiate et visuelle des spécificités géographiques. Elle traduit des données complexes en représentations graphiques claires et intuitives, facilitant ainsi l'analyse et la prise de décision. Grâce à des cartes, il est possible de visualiser une grande variété de phénomènes, tels que des densités de population, des conditions météorologiques ou des informations économiques et sociales.

Python offre encore une fois de nombreux modules comme Folium, GeoPandas et Plotly-Express qui sont des solutions efficaces pour créer des cartes. Connaître ces trois modules est recommandé, car ils offrent des moyens variés pour répondre à tout type de besoin ou de visualisation.

Avant de commencer à produire des cartes, il est essentiel de comprendre le processus de géocodage et l'obtention des coordonnées géographiques des points d'intérêt.

Géocodage

Le géocodage est le processus de conversion des adresses physiques en coordonnées géographiques représentées par la latitude et de la longitude. Voici un tableau pour revoir la définition de ces concepts :

Paramètre	Définition	Caractéristiques
Latitude	Mesure de la position nord-sud sur Terre par rapport à l'équateur	Varie de -90° (sud) à +90° (nord)

Paramètre	Définition	Caractéristiques
Longitude	Mesure de la position est-ouest sur Terre par rapport au méridien de Greenwich	Varie de -180° (ouest) à +180° (est)

Pour obtenir ces coordonnées, nous pouvons utiliser le module Geopy qu'il convient au préalable d'installer :

```
pip install geopy
```

Une fois déployé, voici comment obtenir, par exemple, les coordonnées du Musée du Louvre :

```
from geopy.geocoders import Nominatim

# Nominatim est la fonction de géocodage
geolocator = Nominatim(user_agent="recherche-ville")

# Indication de l'adresse, ville et pays
loc = geolocator.geocode("99 rue de Rivoli, Paris, France")

# Affichage des coordonnées
print("latitude :" ,loc.latitude,"\nlongtitude :" ,loc.longitude)

# Output: latitude : 48.8627337
# Output: longtitude : 2.3348359
```

Remarque

Il n'est pas obligatoire d'indiquer une adresse précise, rechercher la ville et le pays ou le pays uniquement fonctionnent parfaitement.

Geopy propose également une fonction qui peut s'avérer intéressante : la distance géodésique entre deux points. Voici comment procéder pour estimer la distance Paris-New York :

```
from geopy.distance import geodesic

# Coordonnées de Paris et New York
Paris = (48.86, 2.32)
NYC = (40.71, -74)

# Affichage de la distance en kilomètres
```

```
print(geodesic(Paris, NYC).km)

# Output: 5850.376365
```

Les fonds de cartes

Un autre aspect essentiel de la cartographie concerne les fonds de cartes. Il s'agit ici de polygones représentant des entités géographiques telles que des pays, des régions, des quartiers, ou tout autre type de découpage géographique. Pour représenter ces figures, il est nécessaire de récupérer leurs coordonnées, souvent stockées dans des fichiers de type GeoJSON.

À l'exception des cartes des pays ou des États américains, il faudra généralement effectuer des recherches pour les obtenir. Il est conseillé ici d'aller explorer les grands sites d'open data gouvernementaux, GitHub ou des plateformes spécialisées comme gadm.org pour y parvenir. Voici, par exemple, un lien pour se procurer les départements français :

https://www.data.gouv.fr/fr/datasets/carte-des-departements-2-1/

Il est temps maintenant de s'initier à la cartographie en présentant quelques exemples simples.

Affichage d'une carte avec des indicateurs

Folium est un bon moyen de démarrer. C'est une bibliothèque Python pour la création de cartes interactives basées sur Leaflet.js qui est elle-même une bibliothèque JavaScript. Il présente une syntaxe concise et compréhensible permettant d'afficher très rapidement la carte désirée.

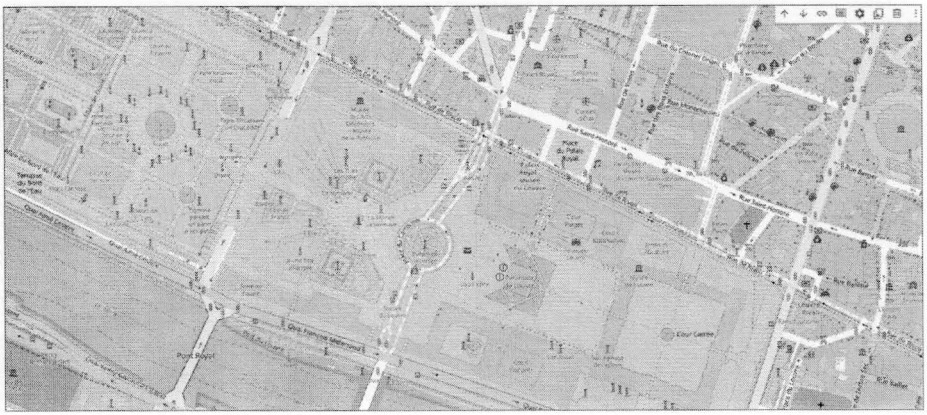
D'abord, procédons à l'installation :

```
pip install folium
```

Assurons-nous à présent que les coordonnées fournies pour le Louvre sont correctes :

```
import folium

# Création de la map. La valeur de zoom est élevée pour afficher
précisément

my_map = folium.Map(
    location=[48.8627337,2.3348359],
```

```
    zoom_start=17
)

# Affichage de la carte
my_map
```

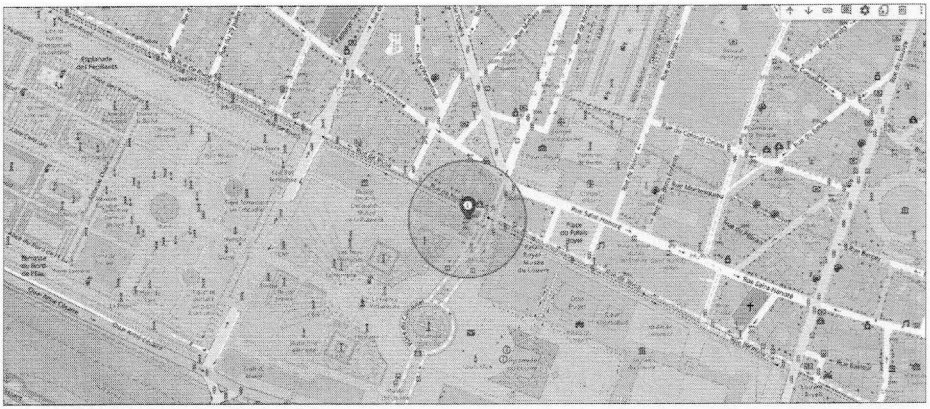


Un document HTML est ainsi créé correspondant précisément à la localisation du musée.

Nous pouvons ensuite ajouter tout type d'indicateurs sur cette carte. Voici un exemple avec un marqueur rouge pour indiquer précisément l'entrée et un cercle dont le diamètre peut être ajusté grâce au paramètre `radius` :

```
import folium

# Latitude et longitude de l'entrée
Louvre_entry = [48.8627337,2.3348359]

# Création de la carte
my_map = folium.Map(location = Louvre_entry, zoom_start=17)

# Ajout d'un marqueur rouge pour indiquer précisément l'entrée
folium.Marker(Louvre_entry, popup='99 rue de Rivoli 75001 Paris',
icon=folium.Icon(color='red', icon='info-sign')).add_to(my_map)

# Ajout d'un cercle de rayon 100 (radius=100)
folium.CircleMarker(
    location= Louvre_entry,
    radius=100,
```

```
    color='#439cc8',
    fill=True,
    fill_color='#439cc8'
).add_to(my_map)
my_map
```



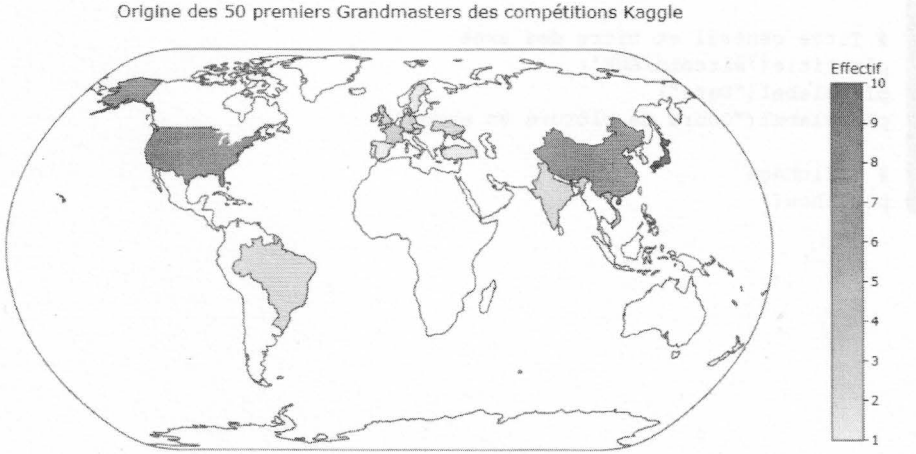
Pour finir sur Folium, précisons que les cartes sont interactives et qu'il est possible de changer le thème de la carte avec l'option `tiles`.

Carte choroplèthe

Les cartes choroplèthes sont des cartes découpées en régions et colorées selon une certaine variable. Le découpage nécessite de recourir à un fichier GeoJson. Pour notre exemple, nous utiliserons Plotly-express en repartant des données sur le nombre de Grandmasters de Kaggle par pays. Il est possible ensuite, si le nom des pays est écrit en anglais, d'utiliser le découpage par pays pré-fourni avec la commande `locationmode` :

```
import plotly.express as px
import pandas as pd

# Données récoltées et création du DataFrame
data = {
    "country": ["Germany", "Austria", "Brazil", "China",
                "South Korea", "Spain", "France", "India",
                "Ireland",. "Japan", "United Kingdom", "Singapore",
                "Sweden", "Czech Republic", "Turkey",
                "Ukraine", "United States", "Vietnam"],
    "headcount": [2, 2, 2, 7, 1, 1, 3, 2, 1, 10,
```

```
                2, 1, 1, 1, 1, 2, 8, 3]
}
df = pd.DataFrame(data)

# Palette personnalisée
custom_color_list = [
    [0, "rgb(245,245,245)"],
    [1, "rgb(67,156,200)"]
]

# Création de la carte choroplèthe avec Plotly Express
fig = px.choropleth(df,
                    locations="country",
                    locationmode="country names",
                    color="headcount",
                    projection="natural earth",
                    color_discrete_sequence=["white"],
                    color_continuous_scale=custom_color_list,
                    template="simple_white")



# Ajout du titre centré
fig.update_layout(title_text='Origine des 50 premiers Grandmasters
des compétitions Kaggle', title_x=0.5)

# Affichage de la carte
fig.show()
```



3.4.2 Les données temporelles

Les données temporelles sont des séries de données collectées à différents moments, permettant d'observer les évolutions et les cycles. Ce type de données offre ainsi une perspective dynamique ouvrant la voie aux prévisions.

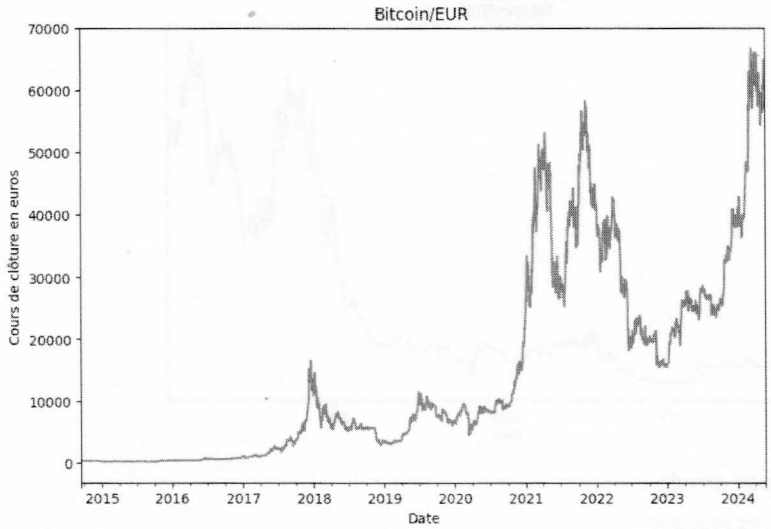
Indexation par date

Construire un graphique temporel avec Python est aisé, mais il faut faire attention à un détail important : il est recommandé d'utiliser les dates comme index plutôt que comme une simple variable. Cette opération ouvre différentes possibilités sur les transformations des dates que nous allons détailler.

Possibilités de tracés offertes

Prenons l'exemple de l'évolution du bitcoin. Nous utiliserons Matplotlib car, bien que ne bénéficiant pas de l'interactivité de Plotly-express, la bibliothèque s'avère bien plus simple à utiliser pour les opérations portant sur les dates.

```
import matplotlib.pyplot as plt
import pandas as pd

# Import des données
bitcoin = pd.read_csv("https://raw.githubusercontent.com/eric2mangel/
BDD/main/BTC-EUR.csv",index_col="Date",parse_dates=True)

# Tracé du graphique
bitcoin['Close'].plot(figsize=(9,6), color="#439cc8")

# Titre général et titre des axes
plt.title('Bitcoin/EUR')
plt.xlabel("Date")
plt.ylabel("Cours de clôture en euros")

# Affichage
plt.show()
```



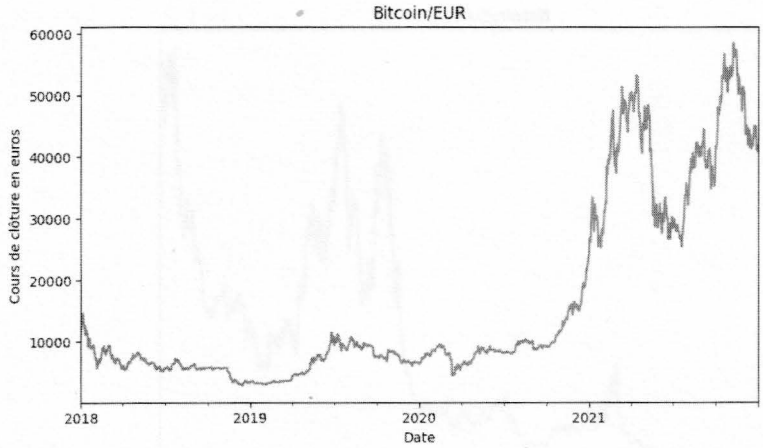
Il est nécessaire, ici, d'apporter quelques précisions sur le code lors de l'acquisition du CSV (`read_csv`) :

- La variable Date est considérée comme un index grâce au recours à la commande `index_col`.
- La commande `parse_dates=True` permet de gérer efficacement les dates en les transformant en objets datetime.

L'acquisition correcte des données temporelles nous ouvre les possibilités suivantes en modifiant les lignes suivantes :

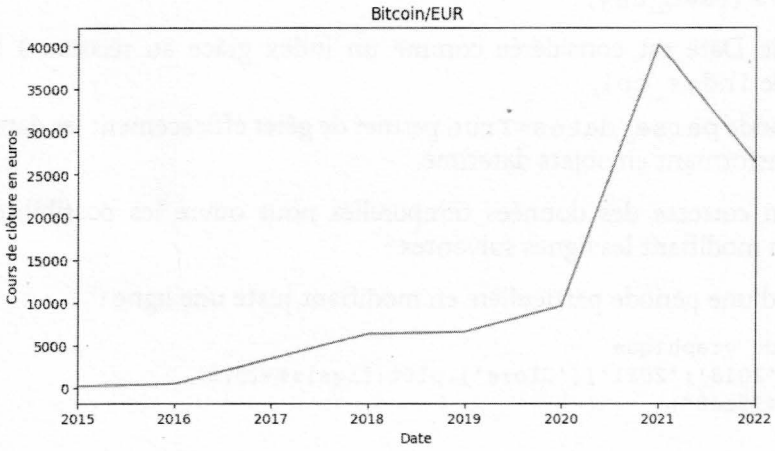
- Sélection d'une période particulière en modifiant juste une ligne :

```
# Tracé du graphique
bitcoin['2018':'2021']['Close'].plot(figsize=(9,5),
color="#439cc8")
```



– Affichage par année :

```
# Tracé du graphique
bitcoin['2015':'2022']['Close'].resample('YE').mean().plot(figsize=(9,5),
color="#439cc8")
```



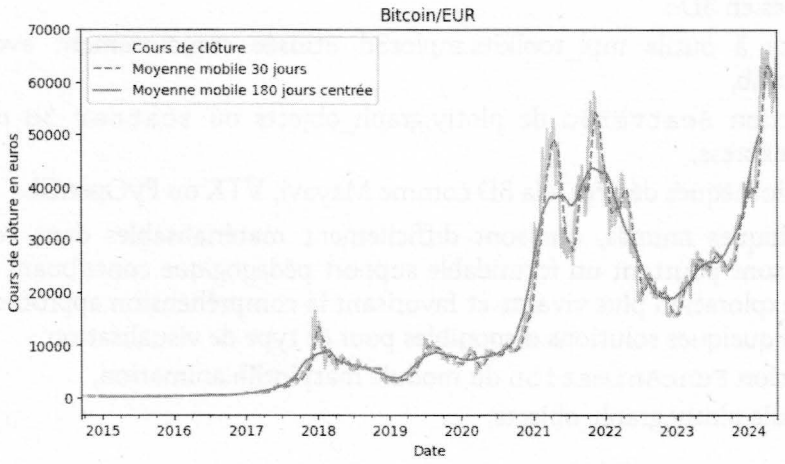
Remarque

L'option `resample` autorise une grande variété de rééchantillonnages de périodes qui vont bien au-delà de la semaine ou du mois. Voir la page concernée sur le site de Pandas : https://pandas.pydata.org/pandas-docs/dev/reference/api/pandas.DataFrame.resample.html.

Ajout de moyennes mobiles

La moyenne mobile permet de lisser la fluctuation des données et de faciliter l'identification de tendances sous-jacentes. Elle est calculée en prenant la moyenne successive sur une fenêtre glissante dont la durée est à définir selon les besoins.

```
# Tracé du cours de clôture
bitcoin['Close'].plot(figsize=(9,5),
color="lightgrey",label='Cours de clôture')

# Tracé de la moyenne mobile à 30 jours
bitcoin['Close'].rolling(window=30).mean().plot(c='#439cc8',
linestyle='--',label='Moyenne mobile 30 jours')

# Tracé de la moyenne mobile à 180 jours recentrée pour éviter
le décalage
bitcoin['Close'].rolling(window=180,
center=True).mean().plot(c='#439cc8',label='Moyenne mobile 180
jours centrée')

# Mise en forme
plt.title('Bitcoin/EUR')
plt.xlabel("Date")
plt.ylabel("Cours de clôture en euros")
plt.legend()
plt.show()
```



Remarque

D'autres types d'indicateurs peuvent être ajoutés pour identifier les tendances, comme des régressions linéaires ou polynomiales. Cependant, il est prématuré de les tracer ici, car cela nécessite de définir et de comprendre ces concepts au préalable. Nous aborderons ce sujet au chapitre L'apprentissage non supervisé.

3.4.3 Les autres solutions graphiques

Les graphiques présentés précédemment ont été sélectionnés car ils sont les plus couramment utilisés. Cependant, il existe de nombreuses autres solutions graphiques, tant sur le fond que sur la forme.

Visualisations alternatives

Les graphiques sont majoritairement représentés en deux dimensions et de manière statique. Voici une liste de solutions alternatives qu'il est opportun d'expérimenter pour peu qu'elles apportent un gain dans l'exploration :

- La représentation en trois dimensions doit être envisagée uniquement si l'affichage est dynamique. Il faut en effet pouvoir révéler tous les aspects et ne pas rester prisonnier d'un seul point de vue. La 3D, longtemps restée statique, a certainement été un frein à l'essor de ce type de visualisation mais les possibilités offertes aujourd'hui devraient pousser les utilisateurs à l'expérimenter. Voici quelques bibliothèques permettant de créer des graphiques en 3D :
  - la boîte à outils mpl_toolkits.mplot3d utilisée conjointement avec Matplotlib,
  - la fonction `Scatter3d` de plotly.graph_objects ou `scatter_3d` de plotly-express,
  - les bibliothèques dédiées à la 3D comme Mayavi, VTK ou PyOpenGL.
- Les graphiques animés, qui sont difficilement matérialisables dans cet ouvrage, sont pourtant un formidable support pédagogique contribuant à rendre l'exploration plus vivante et favorisant la compréhension approfondie. Voici quelques solutions disponibles pour ce type de visualisation :
  - la fonction `FuncAnimation` du module matplotlib.animation,
  - le module plotly.graph_objects,

- la bibliothèque Manim,
- la bibliothèque Pygame connue pour le développement de jeux, mais qui peut également être utilisée pour créer des animations graphiques.

Pour finir sur ce point, toute mesure facilitant la manipulation des données doit être valorisée car c'est dans l'expérimentation que nous découvrons l'information.

Les autres bibliothèques graphiques

Python offre de nombreuses autres solutions de visualisations dont voici une liste non exhaustive :

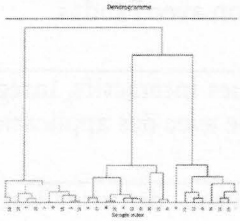
Module	Description rapide	Points forts
Altair	Visualisation exploratoire de données, visualisation de données statistiques.	Syntaxe simple et intuitive, intégration avec Pandas.
Bokeh	Visualisation interactive pour le web et tableaux de bord pour le web.	Graphiques interactifs, intégration facile avec des applications web.
Dash	Applications de données interactives, tableaux de bord complexes.	Combinaison de visualisation et d'interactivité, basé sur Plotly.
GeoPandas	Extension de Pandas pour la manipulation et l'analyse de données géospatiales.	Manipulation facile des données géospatiales, intégration avec Pandas.
Mayavi	Bibliothèque de visualisation scientifique en 3D.	Visualisation 3D puissante, support de visualisation de données complexes.
Plotnine	Implémentation de la bibliothèque ggplot2 de R pour Python.	Basé sur la grammaire des graphiques, syntaxe déclarative.
Pygwalker	Visualisation de données pour les non-programmeurs, exploration rapide de données.	Simplifie la visualisation de données sans besoin de codage complexe.

Chaque bibliothèque possède ses forces et ses faiblesses, et chaque personne a des affinités particulières. Il est donc important de tester pour trouver celles qui répondent au mieux à nos besoins.

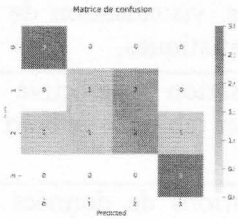
Les visualisations liées au Machine Learning

En plus de tous les graphiques destinés à la visualisation, le machine learning propose des visualisations spécifiques pour évaluer la performance des modèles et interpréter les résultats obtenus lors de l'entraînement et du test des algorithmes. Voici quelques exemples très fréquemment rencontrés en machine learning disponibles dans Scikit-Learn :

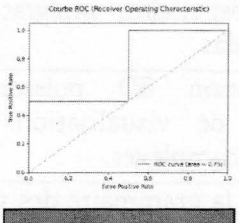
LES GRAPHIQUES DU MACHINE LEARNING



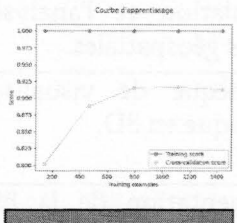
Dendrogramme



Matrice de confusion



Courbe ROC

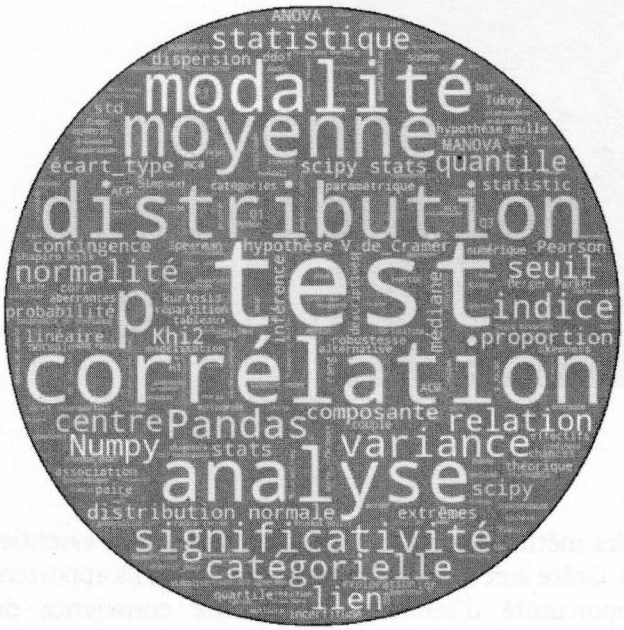


Courbe d'apprentissage

Ce tour d'horizon des visualisations touche à sa fin. Nous allons voir dans le chapitre Analyse des données comment valider et approfondir les premières informations issues des graphiques.

Chapitre 5
Analyse des données

1. Introduction à l'analyse des données

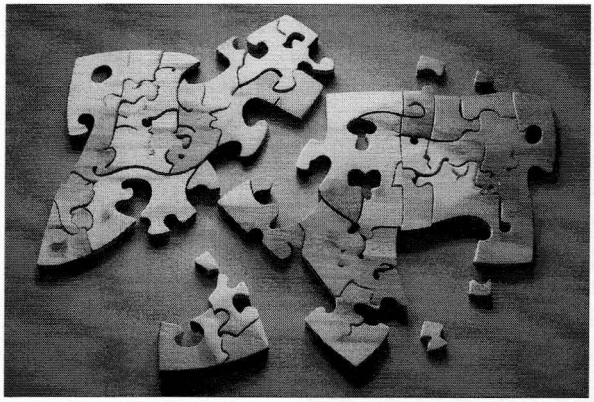


Nuage de mots des termes techniques et informatiques utilisés pour ce chapitre

1.1 Définition et rôle de l'analyse de données

Dans le cadre de l'analyse exploratoire de données (couramment appelée EDA pour *Exploratory Data Analysis*), exploration et analyse fonctionnent en tandem pour approfondir notre compréhension des informations, visibles ou cachées, que révèlent les données. Tandis que l'exploration nous a permis de porter un premier regard et d'identifier des schémas initiaux, l'analyse nous permet de parfaire cette connaissance par un examen plus minutieux, grâce notamment à des outils statistiques qui permettent de mesurer les phénomènes observés et de valider s'ils peuvent être considérés comme significatifs ou non.

Prenons l'exemple d'un puzzle : là où l'exploration nous permettrait d'entrevoir, à travers l'examen des formes et des couleurs, la possibilité d'assembler certaines pièces, l'analyse représenterait l'examen minutieux qui confirmerait la faisabilité des différents assemblages.



1.2 Enjeux

Les tests statistiques et les méthodes d'analyses exploratoires sont essentiels dans l'étude des données. Grâce aux connaissances nouvelles qu'ils apportent, nous allons avoir l'opportunité d'innover, de prendre conscience des contraintes et d'améliorer la prise de décision.

1.2.1 Innovation et créativité

L'apport de nouvelles informations, illustrées de trois façons différentes et décrites ci-dessous, favorisera l'émergence de l'innovation et de la créativité. Cette compréhension approfondie des données nous permettra ainsi de concevoir des solutions originales.

Identification des caractéristiques importantes

En premier lieu, l'analyse nous permet d'identifier les éléments ayant un impact significatif sur le domaine étudié. Nous pourrons ainsi déterminer les variables d'intérêt, comprendre les liens existants entre elles et mettre en place des mesures prioritaires les concernant. Dans le cadre de l'examen des ventes d'un restaurant, par exemple, le jour de la semaine, le moment de la journée, le type de plat ou la présence d'offres promotionnelles pourraient se révéler être des facteurs clés de l'analyse. A contrario, la couleur des uniformes du personnel pourrait ne pas avoir d'impact significatif sur les ventes.

Détection et gestion des problèmes

En second lieu, nous avons la possibilité de découvrir des anomalies ou des écarts importants dans les données, qui peuvent indiquer la présence de problèmes. Paradoxalement, ces problèmes peuvent constituer des opportunités inattendues pour agir. Par exemple, pour l'analyse du restaurant, nous pourrions constater une baisse significative des commandes les mercredis soir. Cette anomalie pourrait indiquer un problème spécifique à ce jour, comme un menu moins attrayant ou une concurrence accrue. Cependant, en identifiant cette baisse, le restaurant pourrait exploiter cette opportunité pour lancer une promotion spéciale le mercredi soir, attirant ainsi plus de clients, et transformer un point faible en avantage.

Les schémas et les tendances de fond

En dernier lieu, des tendances et des modèles plus globaux émergeront, constituant une synthèse de notre examen approfondi des données. Pour notre restaurant, cela pourrait révéler des tendances globales comme le fait que les plats à base de tel ingrédient se vendent mieux en semaine ou que telle boisson est particulièrement demandée le samedi.

Ces multiples informations, qu'elles soient locales ou globales, renforceront notre connaissance fine du domaine et faciliteront la conception de stratégies créatives et innovantes sur lesquelles reposent nos modélisations.

1.2.2 Prise de conscience des contraintes spécifiques

En évaluant correctement l'information contenue dans les données, nous allons pouvoir prendre la mesure des contraintes qui vont limiter nos possibilités.

Nombre de variables à gérer

Lors de l'analyse, il peut advenir qu'un grand nombre de variables jouent un rôle significatif pour un problème donné. Cette profusion de caractéristiques va avoir un impact direct sur notre travail en termes de temps et de puissance de calcul : plus la base de données est importante ou complexe, plus les calculs sont longs et réclament de la puissance.

Mais plutôt que de considérer cette abondance de variables comme un obstacle, nous pouvons la transformer en une opportunité.

D'abord, en profitant du nombre de variables disponibles pour affiner les questions posées, nous pouvons ne garder que celles qui répondent spécifiquement à un sujet mieux défini. Par exemple, en décidant de prédire la satisfaction des clients par catégories d'âges plutôt que globalement, nous pouvons cibler notre analyse. En raisonnant par groupe, il devient plus simple de trouver les variables d'importance pour chaque catégorie d'âge plutôt que pour l'ensemble des clients. En procédant ainsi, nous réduisons la complexité du système.

Ensuite, cela peut être l'occasion de recourir à des techniques de réduction dimensionnelle pour encore parfaire nos connaissances et affiner la sélection. Dans le chapitre Le Machine Learning avec Scikit-Learn, nous examinerons comment la réduction dimensionnelle fonctionne en recombinant les variables pour créer de nouvelles combinaisons qui apportent plus d'information avec moins de variables.

Enfin, dans les cas extrêmes où le nombre de caractéristiques est énorme, il est possible de trouver une issue en gardant uniquement les variables qui fluctuent le plus. Ce filtrage par faible variance permet d'éliminer les caractéristiques peu informatives, en ne conservant que les variables dont la variance est la plus élevée.

Échantillon significatif

Nos possibilités d'analyses vont également dépendre du nombre d'observations disponibles dans la base de données. C'est l'occasion de rappeler qu'en dessous de cinquante observations, il n'y a pas de data science possible. Ce nombre est le seuil minimal pour commencer à travailler et cela nous permet d'introduire la notion de significativité.

La significativité nous aide à distinguer les résultats réels des coïncidences en indiquant la probabilité que le phénomène observé soit dû au hasard. Le nombre des observations joue dans ce cas un rôle primordial, car plus il y a de données, plus il est facile de détecter des effets réels et de réduire l'influence des variations aléatoires. Ainsi, les phénomènes constatés sur de faibles volumes auront de fortes chances de ne pas être pris en compte, ce qui réduira forcément notre champ des possibles dans l'interprétation et la généralisation des résultats. Cela pourrait aussi être l'occasion de voir s'il est possible d'obtenir plus de données.

1.2.3 Amélioration de la prise de décision

Toutes ces opérations ont pour finalité d'améliorer la prise de décision reposant sur deux piliers : la validation des hypothèses et la réduction des risques liés à l'incertitude.

Validation des hypothèses

La validation des hypothèses suit naturellement l'EDA. En manipulant et en observant les données sous différents angles, nous identifions des phénomènes qui suscitent des hypothèses. En travaillant sur leur validation ou leur rejet, nous allons progressivement améliorer notre connaissance des phénomènes et faciliter la prise de décision.

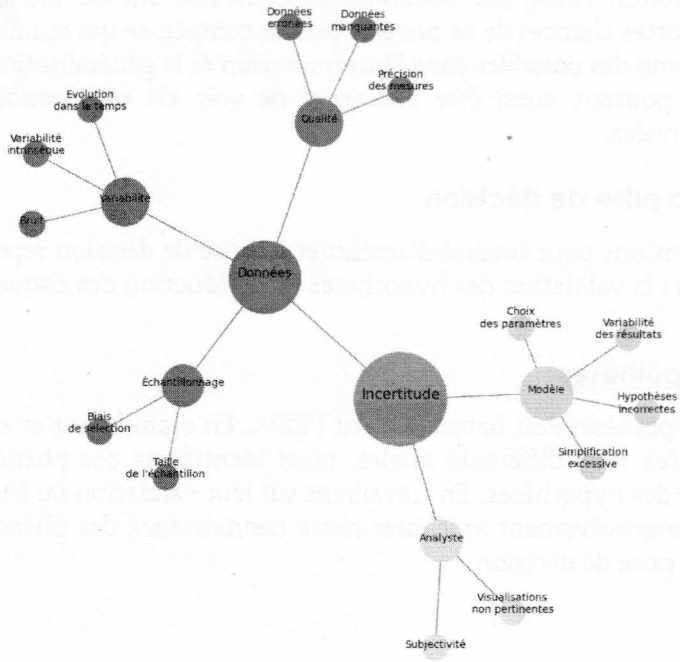
Par exemple, supposons que nous analysions les ventes de glaces et que nous remarquons une augmentation des ventes durant l'été. Nous pouvons formuler l'hypothèse que les ventes de glaces augmentent avec la hausse des températures. Grâce à l'EDA, nous explorons les données pour vérifier cette relation. Si les données montrent effectivement une corrélation entre les ventes de glaces et les températures élevées, nous validons notre hypothèse. Sinon, nous la rejetons et cherchons d'autres facteurs explicatifs. Cette convergence progressive va finalement considérablement faciliter la prise de décision.

Réduction des risques liés à l'incertitude

L'incertitude est un facteur omniprésent qui pèse fortement sur la prise de décision.

Pour nous donner une bonne vision d'ensemble de toutes les sources possibles d'incertitudes, nous allons les représenter à l'aide du graphe suivant :

Graphe des sources d'incertitude dans une EDA



Il ressort de l'étude du graphique précédent que lorsque nous sommes au stade de l'analyse, il faut aller traquer les sources d'incertitude d'abord dans les données, puis en nous-mêmes. Un jeu de données de qualité sera un gage de certitude. Passée cette vérification, il ne nous restera alors qu'à nous interroger sur notre subjectivité et sur la pertinence de nos visualisations. Plus tard, lors de la modélisation, nous serons de nouveau confrontés à d'autres manifestations de l'incertitude, mais nous aborderons ce problème en détail dans le chapitre Le Machine Learning avec Scikit-Learn.

À présent que les contours d'une bonne prise de décision sont définis, nous allons étudier les outils statistiques dont nous disposons pour mener à bien nos investigations. Ce sera aussi l'occasion de revoir les notions fondamentales de statistique.

2. Statistiques descriptives et inférentielles

L'analyse des variables nécessite de revoir certaines notions de statistiques. Il n'est nullement question ici de se noyer dans les détails mais simplement de comprendre les principaux outils nécessaires à l'analyse.

L'analyse statistique repose sur deux piliers que sont les statistiques descriptives et les statistiques inférentielles, chacune jouant un rôle bien spécifique.

Les statistiques descriptives ont pour objectif de décrire et résumer au mieux un ensemble de données. Pour faciliter la compréhension et l'application des concepts, nous allons étudier séparément les mesures pour les variables quantitatives et les variables catégorielles. Cette approche permettra de bien distinguer les méthodes spécifiques à chaque type de variable.

Les statistiques inférentielles permettent de généraliser les caractéristiques d'un échantillon de données à une population générale, dans le but de faire des prédictions. Nous utiliserons des tests spécifiques à chaque typologie de cas qui valideront ou non nos hypothèses.

Avant de faire connaissance avec les différentes statistiques, il est utile de définir la notion de robustesse, qui va être largement abordée. Un test ou une mesure est dit robuste s'il fournit des résultats fiables et précis même lorsque les conditions idéales ne sont pas toutes réunies. Voici les situations qu'un test ou une mesure robuste doivent pouvoir gérer :

- la présence de valeurs aberrantes ;
- la non-normalité de la distribution ;
- les données ordinales ;
- les ex aequo.

Cette robustesse est essentielle pour garantir des conclusions valides dans des conditions variées.

2.1 Description des variables quantitatives

L'examen des variables quantitatives, également appelées continues, s'opère en trois étapes : identifier le centre, décrire la variabilité autour de ce centre, et analyser plus généralement la distribution des valeurs de la variable.

2.1.1 Mesures de tendance centrale

La mesure de la tendance centrale repose principalement sur trois outils : la moyenne, la médiane et le mode. Ils ont en commun d'être simples à construire et à comprendre.

La moyenne

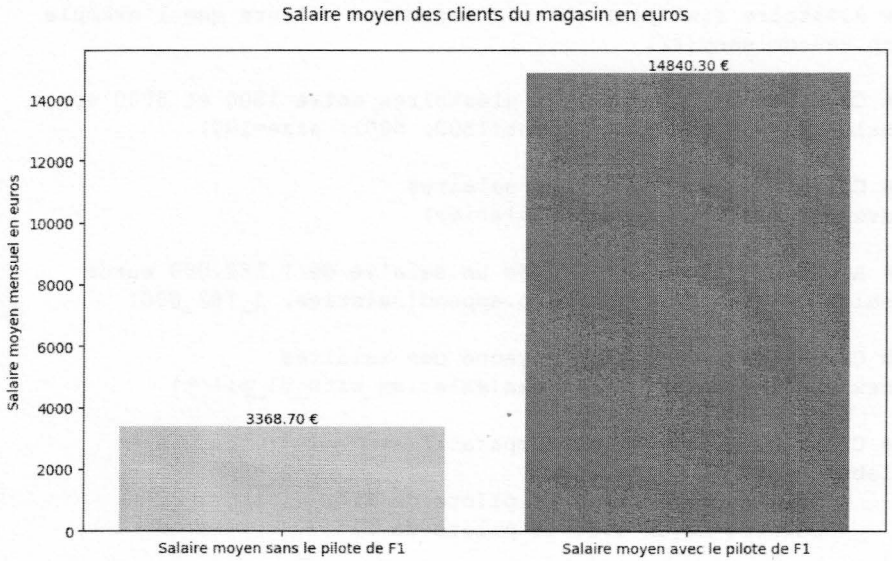
La moyenne arithmétique est l'un des outils les plus couramment utilisés. Elle consiste à faire la somme de toutes les valeurs et à les diviser par leur nombre.

C'est le premier réflexe pour trouver le centre des données mais il souffre d'un handicap majeur : l'influence des valeurs extrêmes.

Prenons un exemple pour illustrer la façon de calculer la moyenne et souligner sa faiblesse : imaginons un magasin contenant 100 clients dont les salaires varient de 1500 et 5000 euros. Comparons maintenant le salaire moyen du client de ce magasin avant et après l'entrée d'un pilote de F1 dont le salaire annuel moyen en 2023 était de 13 950 000 euros, soit 1 162 500 euros mensuels.

```
import numpy as np
import matplotlib.pyplot as plt

# Aléatoire fixé pour obtenir les mêmes valeurs que l'exemple
np.random.seed(42)

# Création de 100 salaires aléatoires entre 1500 et 5000 euros
salaries = np.random.randint(1500, 5001, size=100)

# Calcul de la moyenne des salaires
average_salary = np.mean(salaries)

# Ajout du pilote de F1 avec un salaire de 1,162,000 euros
salaries_with_f1_pilot = np.append(salaries, 1_162_000)

# Calcul de la nouvelle moyenne des salaires
new_average_salary = np.mean(salaries_with_f1_pilot)

# Création du graphique comparatif
labels = [
    'Salaire moyen sans le pilote de F1',
    'Salaire moyen avec le pilote de F1'
]
values = [average_salary, new_average_salary]

plt.figure(figsize=(10, 6))
bars = plt.bar(
    labels,
    values,
    color=['lightgrey', '#439cc8']
)

# Ajout des valeurs sur les barres
for bar in bars:
    height = bar.get_height()
    plt.text(
```

```
        bar.get_x() + bar.get_width() / 2.0,
        height,
        f'{height:.2f} €',
        ha='center',
        va='bottom'
    )

plt.ylabel('Salaire moyen mensuel en euros')
plt.title('Salaire moyen des clients du magasin en euros\n')
plt.show()
```



Le salaire moyen est ainsi passé de 3368.70 euros à 14840.30 euros soit une multiplication du salaire moyen par plus de quatre pour une personne supplémentaire.

Remarque

La commande `random.seed` de numpy a été utilisée afin que les tirages aléatoires soient reproductibles et que quiconque exécute ce programme obtienne les mêmes résultats.

À toutes fins utiles, il existe d'autres types de moyennes. Voici un tableau les présentant. Le même exemple des clients du magasin a été repris pour souligner les différences :

Type de moyenne	Explication	Type de données	Forces	Faiblesses	Salaire moyen avec pilote de F1
Géométrique	Moyenne multiplicative	Conseillée pour les données qui varient de manière multiplicative	Robuste aux valeurs extrêmes	Non définie pour des valeurs négatives ou nulles	3394.98 euros
Harmonique	Moyenne des inverses	Conseillée pour les taux et les ratios	Très sensible aux faibles valeurs	Très influencée par les faibles valeurs proches de zéro	3056.90 euros
Quadratique	Moyenne des carrés	Pour des données qui concernent l'énergie ou la puissance	Utile pour l'analyse de variance	Sensible aux valeurs extrêmes	5323.05 euros

Pour les trois autres types de moyennes, l'impact de l'ajout du pilote de F1 est moins prononcé que celui observé avec la moyenne arithmétique. Voici le code nécessaire pour les calculer :

Type de moyenne	Module	Commande
Géométrique	Scipy	`import scipy.stats as stats` `stats.gmean(variable)`
Harmonique	Scipy	`import scipy.stats as stats` `stats.hmean(variable)`
Quadratique	Numpy	`import numpy as np` `np.sqrt(np.mean(np.square(variable)))`

La médiane

La médiane est une mesure de tendance centrale qui représente la valeur du milieu dans un ensemble de données ordonnées, séparant les données en deux parties égales.

Remarque

Petite subtilité : dans le cas des jeux de données dont le nombre est pair, il faut faire la moyenne des deux valeurs centrales pour la trouver.

La médiane possède deux gros avantages par rapport à la moyenne :

- sa grande robustesse aux valeurs extrêmes, ce qui en fait une option à privilégier dans ce type de cas ;
- sa capacité à mieux représenter la tendance centrale dans les distributions asymétriques, c'est-à-dire lorsque les valeurs sont réparties de manière inégale autour de la moyenne.

Néanmoins, la prise en compte unique de la valeur centrale rend la médiane moins informative que la moyenne, en particulier lorsque les données sont groupées ou présentent une structure spécifique. De plus, il n'existe pas de déclinaisons équivalentes aux moyennes harmonique, géométrique et quadratique pour la médiane.

Voici quatre façons d'obtenir la médiane à partir de modules différents :

Module	Commande pour calculer la médiane
Numpy	`numpy.median(data)`
Pandas	`dataframe['column'].median()`
Scipy	`scipy.stats.scoreatpercentile(data, 50)`
Statistics	`statistics.median(data)`

Le mode

Le mode est la valeur la plus présente dans un ensemble de données. Cette mesure de tendance centrale peut s'appliquer aussi bien aux variables numériques qu'aux variables catégorielles, ce qui la différencie de la moyenne et de la médiane, qui ne s'appliquent qu'aux données numériques.

Un autre avantage par rapport à ses deux homologues est que dans des distributions asymétriques ou multimodales, le mode peut fournir des informations sur les valeurs les plus courantes, ce que la moyenne et la médiane ne peuvent pas capturer.

Cette mesure possède néanmoins quelques faiblesses :

- Elle est moins informative que la moyenne et la médiane surtout si les données sont continues ou n'ont pas de valeur dominante claire.
- Elle est plus instable lors des petits changements dans les données.
- Comme il peut y avoir plusieurs modes, il peut être difficile de déterminer la tendance centrale.

Remarque

C'est grâce au mode que nous utilisons les termes unimodale, bimodale ou multimodale pour décrire respectivement les distributions de données ayant un mode, deux modes ou plus.

Avec Pandas, le résultat est renvoyé sous forme de liste indiquant les différentes valeurs. Le code ne sera pas fourni pour Scipy, car il ne renvoie pas plusieurs valeurs comme cela devrait être le cas et il n'existe pas de fonction mode pour Numpy.

Avec Pandas, le résultat est renvoyé sous forme de liste indiquant les différentes valeurs. Nous mentionnerons uniquement ce module ici, car contrairement à Scipy, qui par défaut ne renvoie qu'une seule valeur même en présence de plusieurs modes, Pandas offre une sortie plus détaillée avec toutes les valeurs modales. De plus, Numpy ne dispose pas d'une fonction mode intégrée.

```
import pandas as pd

data = {'column': [1, 3, 3, 6, 7, 8, 9, 3, 7, 7]}

df = pd.DataFrame(data)

modes = df['column'].mode().tolist()

print("Modes:", modes)  # Output: [3, 7]
```

Après avoir exploré les indicateurs mesurant la tendance centrale, découvrons ceux qui mesurent la dispersion autour du centre.

2.1.2 Mesures de dispersion

Un indicateur de dispersion a pour but de mesurer la variabilité des valeurs donc évaluer si elles sont regroupées ou éparpillées. Cette information est indispensable et complémentaire de la mesure de la tendance centrale. Un simple exemple permet de comprendre son importance : prenons deux classes ayant la même moyenne de 10/20. Nous pouvons obtenir cette même moyenne avec une classe où tous les élèves ont 10/20 et une autre où la moitié à 0/20 et l'autre moitié, 20/20. Même moyenne mais deux réalités complètement différentes.

Pour évaluer la dispersion, nous avons à disposition plusieurs outils que nous allons découvrir par ordre croissant de complexité : l'étendue, les quartiles et l'écart interquartile, la variance et l'écart-type.

L'étendue

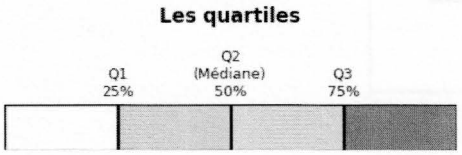
L'étendue est la mesure la plus simple à comprendre et à calculer. Elle correspond à la différence entre la valeur maximale et la valeur minimale d'un jeu de données. Cette mesure simple reste imparfaite car elle ne reflète pas la distribution des valeurs entre les deux extrêmes, mais elle permet de se faire rapidement une première impression. L'étendue apporte une part d'explication sur la dispersion d'un jeu de données, bien qu'il soit nécessaire d'utiliser des mesures complémentaires pour appréhender complètement la dispersion.

Nous pouvons calculer directement le range en faisant `max(data) - min(data)` ou utiliser Numpy qui propose la fonction `ptp` (*peak to peak*) dédiée :

```
Import numpy as np
data = [1, 3, 3, 6, 7, 8, 9, 3, 7, 7]
data_range = np.ptp(data)
print(data_range)
# Output: 8
```

Les quartiles et l'écart interquartile

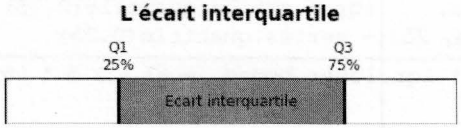
Les quartiles divisent un ensemble de données en quatre parties égales. Il y a trois quartiles principaux : Q1, Q2 et Q3. Le premier quartile (Q1) est la valeur en dessous de laquelle se trouvent 25 % des données, le deuxième quartile (Q2) est la médiane qui divise les données en deux moitiés égales, et le troisième quartile (Q3) est la valeur au-dessus de laquelle se trouvent 25 % des données.



Remarque

Les quartiles font partie de la famille des quantiles, incluant également les déciles ou les centiles, qui divisent un ensemble de données en respectivement 10 et 100 parties égales.

Sitôt connus Q1 et Q3, il est possible de calculer l'écart interquartile qui est la différence entre Q3 et Q1 et qui contient 50 % des valeurs du jeu de données.



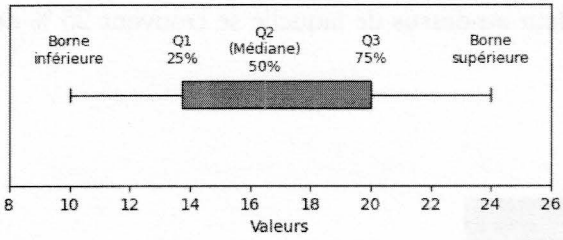
Enfin, la dernière étape consiste à établir les bornes inférieures et supérieures acceptables au-delà desquelles les valeurs seront considérées comme des outliers, c'est-à-dire des observations atypiques qui s'écartent significativement du reste des données et peuvent influencer négativement nos analyses. Ces bornes sont calculées de la façon suivante en se basant sur l'écart interquartile :

Borne inférieure = Q1 – 1.5 * Écart interquartile

Borne supérieure = Q3 + 1.5 * Écart interquartile

Nous retrouvons ainsi tous ces indicateurs sur le graphique de la boîte à moustache :

Les indicateurs de la boîte à moustache



Voici un tableau récapitulatif indiquant les commandes pour calculer les quartiles et écarts interquartiles :

Quartile/ IQR	Numpy	Pandas
Q1	`q1 = np.percentile(data, 25)`	`q1 = series.quantile(0.25)`
Q3	`q3 = np.percentile(data, 75)`	`q3 = series.quantile(0.75)`
IQR	`iqr = np.percentile(data, 75) - np.percentile(data, 25)`	`iqr = series.quantile(0.75) - series.quantile(0.25)`
Borne inférieure	`lower_bound = q1 - 1.5 * iqr`	`lower_bound = q1 - 1.5 * iqr`
Borne supérieure	`upper_bound = q3 + 1.5 * iqr`	`upper_bound = q3 + 1.5 * iqr`

La variance

La variance est une mesure quantitative de la dispersion des valeurs autour de la moyenne. Son calcul nécessite quatre étapes :

- calculer la moyenne des données ;
- soustraire la moyenne à chaque valeur pour obtenir les écarts ;
- élever chaque écart au carré ;
- calculer la moyenne de ces carrés.

Ainsi, plus la variance est élevée, plus les données sont dispersées, et inversement.

Cet indicateur souffre par ailleurs de quelques faiblesses comme :

- l'influence significative des valeurs aberrantes, qui peut entraîner des estimations biaisées ;
- son interprétation, qui est difficile car elle est exprimée en unités au carré.

Avant d'afficher les codes pour calculer la variance, il est essentiel de parler de la correction de Bessel qui concerne les populations et les échantillons. Cette méthode vise à corriger le biais dans l'estimation de la variance de la population en divisant par n−1, car nous travaillons avec un échantillon plutôt qu'avec la population entière. Cette correction est essentielle car elle explique pourquoi le résultat obtenu peut différer de celui que nous anticipions. En effet, diviser par n−1 au lieu de n permet d'obtenir une estimation plus précise de la variance de la population à partir de l'échantillon. Les deux façons de calculer sont indiquées dans le tableau suivant :

Module	Code variance population	Code variance échantillon
Numpy	`var = np.var(data, ddof=0)`	`var = np.var(data, ddof=1)`
Pandas	`var = df['column'].var (ddof=0)`	`var = df['column'].var (ddof=1)`
Statistics	`var = statistics.pvariance (data)`	`var = statistics.variance (data)`

Remarque

Attention : pour Numpy et Pandas, le paramètre `ddof` indique s'il s'agit de la population ou de l'échantillon, en utilisant respectivement 0 pour la population et 1 pour l'échantillon. En revanche, pour le module statistics, nous utilisons les fonctions `pvariance` pour la population et `variance` pour l'échantillon.

Nous retrouverons le même phénomène pour l'écart-type qui est préféré à la variance car il atténue les faiblesses de cette dernière.

L'écart-type

L'écart-type représente la dispersion des données autour de la moyenne. Il est calculé à partir de la variance, dont il est la racine carrée. Ce recours à la racine carrée corrige deux défauts de la variance.

Premièrement, bien que l'écart-type soit toujours sensible aux valeurs aberrantes (outliers), il ne les amplifie pas autant que la variance, car l'unité est rétablie grâce à la racine carrée.

Deuxièmement, et surtout, l'écart-type est exprimé dans la même unité que les données, ce qui facilite grandement les comparaisons et l'interprétation des résultats.

Cela fait de lui l'un des indicateurs les plus importants de la dispersion. Il est utilisé très largement à toutes les étapes du travail sur les données.

Voici les moyens de le mettre en œuvre en tenant en compte, là encore, de la loi de Bessel :

Module	Code écart-type population	Code écart-type échantillon	Attention
Numpy	`std = np.std(data, ddof=0)`	`std = np.std(data, ddof=1)`	Réglé par défaut à ddof=0
Pandas	`std = df['column'] .std(ddof=0)`	`std = df['column'] .std(ddof=1)`	Réglé par défaut à ddof=1
Statistics	`std = statistics .pstdev(data)`	`std = statistics .stdev(data)`	

Maintenant que nous savons comment étudier la répartition des valeurs autour d'un point central, il est temps de découvrir la distribution qui va nous apporter cette image globale indispensable pour comprendre la nature d'une variable numérique.

2.1.3 La distribution

La distribution statistique décrit la manière dont les valeurs d'une variable sont réparties. L'étude de sa forme couplée aux enseignements de tendance centrale et de dispersion va nous permettre de comprendre pleinement le comportement de la variable. L'étude de la distribution va nous permettre d'apporter les éclairages suivants :

- chiffrer la probabilité qu'une valeur se situe dans une certaine plage ;
- déterminer si la distribution se rapproche d'une distribution théorique connue ;
- identifier les caractéristiques des valeurs extrêmes ou aberrantes dans le jeu de données.

La distribution normale

La distribution normale, également connue sous le nom de courbe de Gauss ou courbe en cloche, est un concept qui va jouer un rôle crucial dans l'analyse des données. Elle est nommée ainsi car c'est une distribution très fréquemment observée dans les phénomènes naturels et sociaux. Dans une distribution normale, la majorité des valeurs sont autour de la moyenne et leur fréquence baisse au fur et à mesure que nous nous en éloignons de part et d'autre de manière symétrique.

Pourquoi joue-t-elle un rôle si important dans l'analyse ?

D'abord, parce que sa fréquence d'apparition élevée dans beaucoup de phénomènes en fait une approximation pertinente.

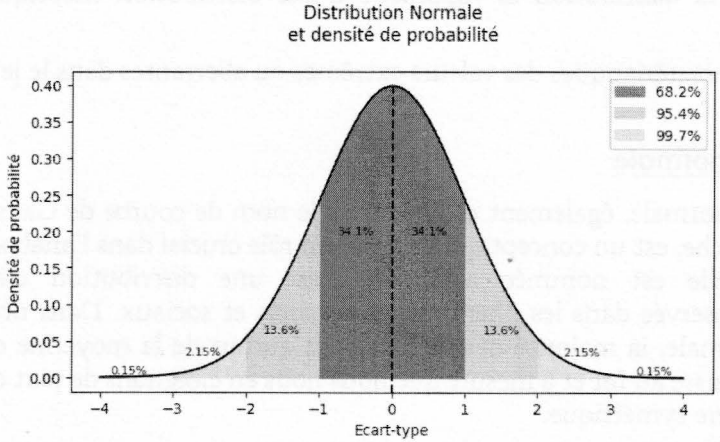
Ensuite, grâce au théorème central limite, même si les données initiales ne sont pas normalement distribuées, les moyennes d'échantillons de ces données tendent vers une distribution normale, permettant d'appliquer divers méthodes et tests statistiques de manière fiable.

Enfin, ses propriétés mathématiques simples facilitent les calculs et l'interprétation des résultats.

Voyons dans le détail ces propriétés intéressantes.

Caractéristiques et propriétés de la distribution normale

- **Symétrie** : la distribution est symétrique par rapport à la moyenne donc il y a autant de valeurs de chaque côté.
- **Égalité des indicateurs de tendance centrale** : dans une distribution normale, moyenne, médiane et mode sont égaux.
- **Règle des 68-95-99** : dans une distribution normale, environ 68.2 % des données se trouvent à moins d'un écart-type de la moyenne, 95.4 % à moins de deux écarts-types, et 99.7 % à moins de trois écarts-types.



Le graphique précédent mérite quelques commentaires pour bien le comprendre :

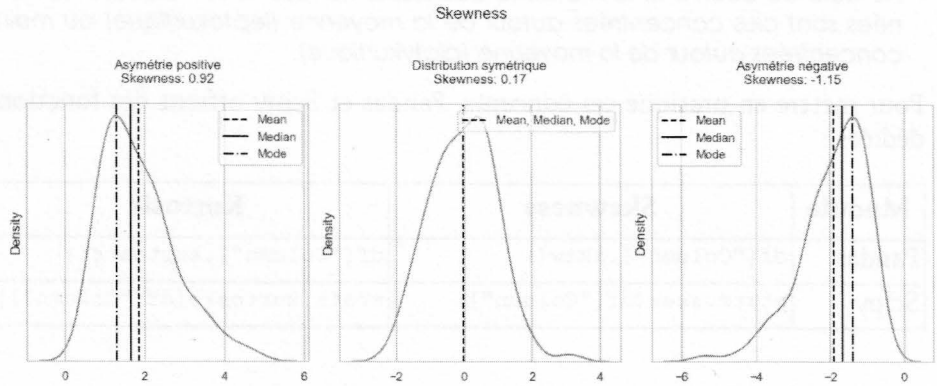
- En théorie, dans une distribution normale, la moyenne, la médiane et le mode ont la même valeur et correspondent à la ligne centrale en pointillés.

- Cette courbe sert d'estimation de probabilité. Ainsi, par exemple, les valeurs situées entre +2 et +3 écarts-types sont censées représenter 2.15 % de l'ensemble. De la même manière, 34.1 % des valeurs sont censées être entre -1 et 0 écart-type. Ces probabilités sont cumulables, c'est pourquoi 68.2 % des valeurs sont situées entre -1 et 1 écarts-types, 95.4 % entre -2 et 2 et pour finir 99.7 % entre -3 et 3 écarts-types. Ce cumul ne fonctionne pas uniquement de manière bilatérale. Nous pouvons par exemple estimer que 15.75 % des observations seront situées entre +1 et +3 écarts-types (13.6 % + 2.15 %).

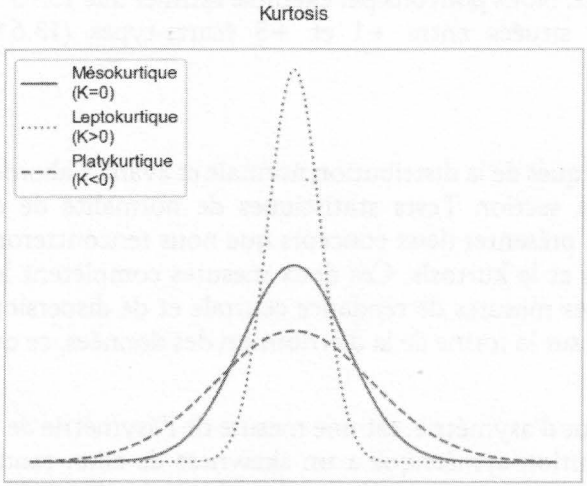
Skewness et kurtosis

Pour finir sur les caractéristiques de la distribution normale et avant d'aborder les tests de normalité à la section Tests statistiques de normalité de ce chapitre, il est opportun de présenter deux concepts que nous rencontrerons fréquemment : le skewness et le kurtosis. Ces deux mesures complètent les informations fournies par les mesures de tendance centrale et de dispersion, en donnant des indications sur la forme de la distribution des données, ce qui facilite leur compréhension :

- Le skewness, ou coefficient d'asymétrie, est une mesure de l'asymétrie de la distribution. Une distribution symétrique a un skewness de zéro, tandis qu'une distribution avec une queue plus longue sur la droite (asymétrie positive) ou sur la gauche (asymétrie négative) aura un skewness différent de zéro.



– Le kurtosis, ou coefficient d'aplatissement, quant à lui, mesure la concentration des données autour de la moyenne. Une distribution normale a un kurtosis de 0 indiquant une distribution avec une forme standard en termes d'aplatissement. Un kurtosis supérieur indique une distribution plus pointue (leptokurtique), tandis qu'un kurtosis inférieur indique une distribution plus aplatie (platykurtique).



Remarque

Au-delà de décrire la forme de la courbe, le kurtosis nous indique si les données sont plus concentrées autour de la moyenne (leptokurtique) ou moins concentrées autour de la moyenne (platykurtique).

Pour mettre en pratique ces concepts, Pandas et Scipy offrent des fonctions dédiées :

Module	Skewness	Kurtosis
Pandas	`df["Column"].skew()`	`df["Column"].kurtosis()`
Scipy	`stats.skew(df["Column"])`	`stats.kurtosis(df["Column"])`

2.2 Description des variables catégorielles

Les statistiques descriptives s'appliquent également aux variables catégorielles.

2.2.1 Fréquence, proportion et gestion des modalités rares

Pour décrire une variable catégorielle, nous allons nous attarder sur trois aspects importants : la fréquence, la proportion et la gestion des modalités rares.

Fréquence

Une variable catégorielle est constituée de modalités et il va s'agir, dans un premier temps, de dénombrer les occurrences de chaque modalité. Cette information est essentielle pour comprendre la répartition des données et identifier les modalités rares ou dominantes.

La notion de mode, déjà abordée dans le chapitre DataViz avec Matplotlib, Seaborn, Plotly, intervient ici : le mode correspond à la modalité la plus fréquente. Identifier le mode nous permet de déterminer quelle modalité est la plus répandue dans notre jeu de données.

Pandas propose la fonction `value_counts()` qui affiche directement le décompte des fréquences d'une variable :

```
df["Variable_name"].value_counts()
```

Proportion

La proportion est la fréquence relative de chaque modalité, exprimée en pourcentage du total des observations, ce qui nous permet d'évaluer l'importance relative de chaque modalité. Cette information est indispensable et complémentaire de la fréquence pour évaluer correctement l'importance de chaque mention.

En ajoutant simplement l'option `normalize=True` à la fonction `value_counts()` de Pandas, nous obtenons les proportions souhaitées :

```
df["Variable_name"].value_counts(normalize=True)
```

Gestion des modalités rares

Les modalités avec un très faible effectif peuvent poser des problèmes dans l'analyse des données. Avec moins de cinq occurrences, nous risquons de rendre l'analyse instable ou non représentative. D'ailleurs, le test du chi-carré (Khi2), que nous allons aborder à la section Tests statistiques bivariés, ne se prononce pas pour les cas ayant moins de cinq occurrences, car des effectifs trop faibles peuvent invalider les résultats du test. Dans ce type de situation, deux options s'offrent à nous : regrouper les mentions sous-représentées dans une modalité « autres », par exemple, ou les ajouter à la mention la plus proche, si cela est possible.

Voici une petite fonction simple qui nous aidera à réunir directement les mentions marginales d'un DataFrame :

```
def regroup_rare_categories(df, column, threshold=5,
new_label='Other'):

    """
      Pour un DataFrame spécifique (df),
      la fonction regroupe les catégories rares,
      fixées par un seuil (threshold) d'une variable catégorielle
      à définir (column) sous une nouvelle étiquette (new_label).

    """
    # Calcul de la fréquence des modalités
    frequency = df[column].value_counts()

    # Identification des catégories rares
    rare_categories = frequency[frequency < threshold].index

    # Remplacement des catégories rares par l'étiquette 'Other'
    df[column] = df[column].replace(rare_categories, new_label)

    return df
```

Comprendre la distribution d'une variable catégorielle est important, mais explorer les associations entre deux variables apporte des connaissances encore plus significatives. C'est ce que nous allons voir avec la pratique du tableau de contingence.

2.2.2 Tableau de contingence

Les tableaux de contingence, également connus sous le nom de tableaux croisés, sont des outils spécialisés dans l'analyse des relations entre deux variables catégorielles. Ils permettent d'observer les effectifs de chaque paire de modalités entre deux variables.

Ces tableaux interviennent à trois moments clés de l'analyse de données : lors de l'exploration initiale, à l'occasion des tests statistiques et au moment de la modélisation. Dans cette section, nous nous concentrerons principalement sur la présentation des tableaux de contingence, tandis que les détails concernant les tests statistiques seront abordés en profondeur dans la section Tests statistiques bivariés de ce chapitre.

La pratique des tableaux de contingence est l'occasion d'en apprendre un peu plus sur le milieu des développeurs de logiciels. Pour cela, nous avons récupéré les données de l'enquête de Stack Overflow de 2023 disponibles à cette adresse : https://survey.stackoverflow.co

Nous allons produire, à partir de ces données, le tableau de contingence entre la tranche d'âge et la variable indiquant si les personnes travaillent en présentiel, en télétravail ou de manière hybride. Voici le code nécessaire pour produire le tableau en utilisant la fonction *crosstab* de Pandas :

```
import pandas as pd

# Remplacer « your_path » par le chemin complet de votre dossier
df = pd.read_csv(r"your_path\survey_results_public.csv")

# Commande pour afficher toutes les colonnes
pd.set_option('display.max_columns', 999)

contingency_table = pd.crosstab(df['Age'], df['RemoteWork'])
contingency_table
```

RemoteWork / Age	Hybrid (some remote, some in-person)	In-person	Remote
18-24 years old	4525	2872	3456
25-34 years old	13601	5024	12394
35-44 years old	8116	2317	9159
45-54 years old	3308	1109	3476
55-64 years old	1198	516	1333
65 years or older	192	118	371
Prefer not to say	44	23	64
Under 18 years old	147	134	313

Nous reviendrons sur ce tableau dans la sous-section dédiée au test du Khi2 pour vérifier la significativité des résultats. Cependant, nous constatons dès à présent que les 18-34 ans travaillent majoritairement en mode hybride, tandis que le télétravail prédomine pour toutes les autres classes d'âge.

Après avoir examiné les tableaux de contingence pour comprendre les relations entre différentes variables, nous pouvons maintenant nous tourner vers un autre outil qui va nous offrir une autre information utile : les indices de diversité.

2.2.3 Indices de diversité

Les indices de diversité quantifient la répartition et l'équilibre des catégories au sein d'une variable ou de couples de variables, offrant ainsi des mesures essentielles pour évaluer la variété ou l'uniformité des données. En se basant sur ces indices, il est possible de comprendre plus précisément la structure et la richesse de l'information contenue, ce qui facilite la compréhension et la prise de conscience des éventuels défauts de répartition des variables catégorielles.

Nous allons faire connaissance avec trois indices de diversité différents : Berger-Parker, Simpson et Shannon-Weaver.

Indice de diversité de Berger-Parker

L'indice de Berger-Parker renvoie la proportion de la catégorie la plus importante.

Ainsi, plus l'indice est proche de 1, plus faible est la diversité et inversement lorsque l'indice tend vers 0.

Il n'existe pas de fonction dédiée pour calculer cet indice mais une fonction récupérant la colonne affichant la plus forte proportion après avoir normalisé chaque colonne fera parfaitement l'affaire. Nous allons repartir de la table de contingence calculée précédemment :

```
# Fonction pour calculer l'indice de Berger-Parker
def berger_parker_index(df):
    # Normalisation par colonne
    proportions = df.div(df.sum(axis=1), axis=0)

    # Recherche de la plus grande valeur par ligne
    berger_index = proportions.max(axis=1)

    # Affichage sous forme de DataFrame
    return pd.DataFrame(
        berger_index,
        columns=['Berger-Parker Index']
    )

# Calcul de l'indice de Berger-Parker
berger_parker_index(contingency_table)
```

Age	Berger-Parker Index
18-24 years old	0.416935
25-34 years old	0.438473
35-44 years old	0.467487
45-54 years old	0.440390
55-64 years old	0.437479
65 years or older	0.544787
Prefer not to say	0.488550
Under 18 years old	0.526936

Dans cet exemple, l'indice de Berger-Parker est renseigné pour chaque tranche d'âge. Les 65 ans et plus forment le groupe présentant la plus faible diversité car la proportion la plus répandue est de 54.47 %. Inversement, ce sont les 18-24 ans qui offrent la plus faible valeur de 41.69 %, représentant ainsi la classe offrant le plus de diversité. Il est important de souligner que cet indice est très sensible à la catégorie dominante.

Indice de diversité de Simpson

L'indice de Simpson présente une autre façon d'étudier la diversité. Il mesure la probabilité que deux individus sélectionnés au hasard dans une population appartiennent à la même catégorie. Comme pour l'indice de Berger-Parker : plus la valeur est proche de 0, plus il y a de diversité, et vice versa.

En repartant de la même table de contingence, voici la fonction nécessaire à la réalisation de l'indice :

```
# Fonction pour calculer l'indice de Simpson
def simpson_index(df):
    # Normalisation par colonne
    proportions = df.div(df.sum(axis=1), axis=0)

    # Somme par ligne des proportions au carré
    simpson_index = (proportions ** 2).sum(axis=1)

    # Création du DataFrame avec l'indice de Simpson
    return pd.DataFrame(
        simpson_index,
        columns=['Simpson Index']
    )
# Calcul de l'indice de Simpson
simpson_index(contingency_table)
```

Age	Simpson Index
18-24 years old	0.345265
25-34 years old	0.378141
35-44 years old	0.404134
45-54 years old	0.389334
55-64 years old	0.374652
65 years or older	0.406306
Prefer not to say	0.382320
Under 18 years old	0.389796

Nous retrouvons les mêmes résultats que pour l'indice de Berger-Parker, mais l'indice de Simpson peut diverger car il est sensible à la distribution globale et non juste à une seule.

Indice de diversité de Shannon-Weaver

Terminons par l'indice de Shannon-Weaver qui est le plus complexe des trois à comprendre. Ici, c'est la diversité des préférences qui est mesurée en tenant compte du nombre de modalités et de la façon dont ces catégories sont réparties. Au lieu de faire la somme de chaque proportion montée au carré comme pour l'indice de Simpson, chaque proportion est ici multipliée par son logarithme donnant ainsi plus de poids aux catégories moins communes.

Voyons les résultats affichés en se basant toujours sur le même exemple pour comparer :

```
import pandas as pd
import numpy as np

# Fonction pour calculer l'indice de Shannon
def shannon_index(df) :
    # Normalisation par ligne
    proportions = df.div(df.sum(axis=1), axis=0)

    # Calcul de l'indice de Shannon
    shannon_index = - (proportions * np.log(proportions))
.sum(axis=1)

    # Retourner le DataFrame avec l'indice de Shannon
    return pd.DataFrame(
        Shannon_index,
        columns=['Shannon Index']
    )

# Calcul de l'indice de Shannon
shannon_index(contingency_table)
```

Age	Shannon Index
18-24 years old	1.080947
25-34 years old	1.022892
35-44 years old	0.973015
45-54 years old	1.001371
55-64 years old	1.029432
65 years or older	0.991564
Prefer not to say	1.021844
Under 18 years old	1.019092

Remarque

L'indice de Shannon vaut 0 s'il n'y a qu'une catégorie mais il peut dépasser 1, contrairement aux deux autres indices, car sa valeur maximale dépend du nombre de catégories observées. Concrètement, plus il y a de catégories, plus cet indice peut être élevé.

Le résultat diverge ici par rapport aux deux autres indices. Si nous retrouvons toujours la catégorie des 18-24 ans comme présentant le plus de diversité, c'est désormais la tranche des 35-44 ans qui présente le moins de diversité. En renforçant l'importance de la catégorie la plus sous-représentée grâce au concours du logarithme, le classement a été modifié. Cette divergence n'est pas à considérer comme un problème mais plutôt comme une opportunité de saisir des détails subtils et ainsi renforcer notre compréhension du jeu de données.

C'est justement l'occasion, après avoir découvert les vertus explicatives des statistiques descriptives, de faire connaissance avec les statistiques inférentielles et les règles qui les régissent.

2.3 Statistiques inférentielles

2.3.1 Concepts de base

Les statistiques inférentielles permettent de généraliser des propriétés à une population à partir d'un échantillon de données. Concrètement, nos données ne représentent presque jamais la population entière d'un domaine d'étude. Nous travaillons la plupart du temps sur une fraction de cet ensemble, appelée échantillon de données. Ainsi, nous avons besoin de savoir si les phénomènes observés dans notre échantillon peuvent se généraliser à l'ensemble de la population. C'est ici qu'interviennent les statistiques inférentielles et leurs outils, capables de nous fournir cette estimation.

Nous allons étudier ici, avant de les détailler, le fonctionnement général des tests d'hypothèses et leur interprétation.

2.3.2 Hypothèses nulles et alternatives

Tous les tests reposent sur la formulation de deux hypothèses :

- l'hypothèse nulle (H0) qui représente la situation de base ;
- l'hypothèse alternative (H1) qui représente le changement.

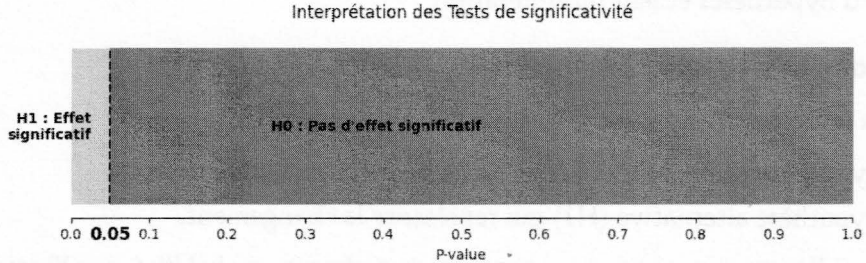
Le but d'un test statistique va consister à évaluer la probabilité que l'hypothèse nulle, représentant l'absence d'effet, soit vraie. Si cette probabilité est trop faible, l'hypothèse nulle (H0) sera rejetée en faveur de l'hypothèse H1, suggérant la présence d'un effet ou d'un changement.

Typiquement, si nous étudions l'influence de l'âge sur la dépense des personnes d'un magasin, nous allons évaluer la probabilité que l'âge n'ait pas d'incidence sur les dépenses. Cette probabilité correspond à l'hypothèse nulle H0. Si cette probabilité est inférieure à un seuil spécifique, nous rejetons H0. Cela signifie que l'absence d'effet est improbable donc nous concluons que l'âge a un impact sur le niveau des dépenses (H1 adoptée).

2.3.3 P-value

La probabilité utilisée pour invalider ou non l'hypothèse nulle se nomme la p-value. C'est le point commun entre tous les tests : ils renvoient tous une p-value. Nous allons ensuite comparer cette p-value à un seuil défini pour décider de conserver ou de rejeter l'hypothèse nulle. Le seuil le plus couramment utilisé est de 5 % (0.05), mais selon les secteurs ou les besoins, nous pourrions utiliser des seuils de 1 % (0.01), 2 % (0.02) ou 10 % (0.10).

Reprenons notre exemple sur l'influence de l'âge sur la dépense des personnes dans un magasin. Supposons que nous obtenons une p-value de 0,03 après avoir effectué notre test statistique. Comparée au seuil de 0,05 le plus couramment utilisé, cette p-value est inférieure, ce qui nous conduit à rejeter l'hypothèse nulle (H0) et conclure que l'âge joue un rôle sur le niveau des dépenses.



2.3.4 Significativité

La significativité statistique nous indique si la p-value observée est inférieure ou égale au seuil choisi, justifiant ainsi le rejet de l'hypothèse nulle. Le résultat est alors statistiquement significatif. Il est d'ailleurs courant de rencontrer des formules de ce type : « Le résultat est significatif au seuil des 5 % ».

Cette possibilité offerte de changer le seuil de significativité n'est pas sans conséquence. Si nous reprenons notre exemple précédent, la p-value de 0.03 au seuil des 5 % nous incite à conclure que l'âge joue un rôle dans le niveau des dépenses mais au seuil des 2 %, par exemple, la conclusion aurait été que l'âge ne joue pas dans le niveau des dépenses car 0.03 est supérieur à 0.02.

Cet exemple met en lumière une des faiblesses des tests statistiques : le p-hacking. Il est ainsi possible, si les résultats ne conviennent pas, de changer le seuil et d'aboutir à l'assertion inverse. Cette possibilité n'est évidemment offerte que pour les p-values comprises entre 1 et 5 % mais c'est important de souligner cette faiblesse du système.

Remarque

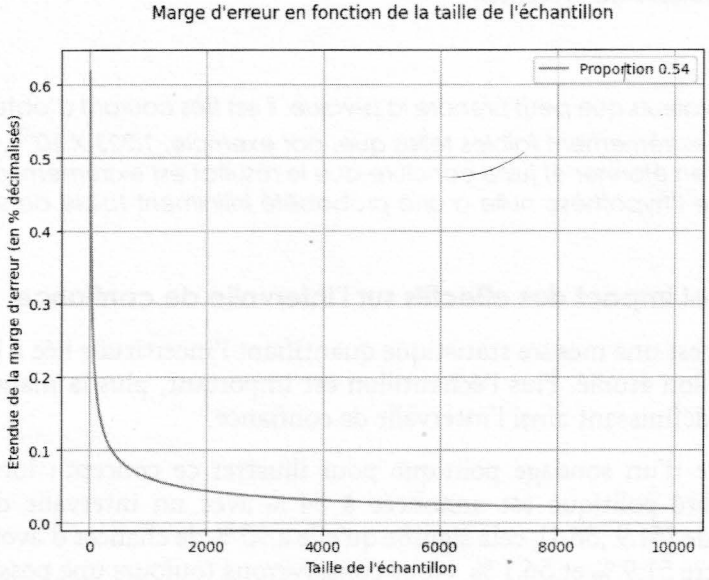
Concernant les valeurs que peut prendre la p-value, il est très courant d'obtenir des p-values extrêmement faibles telles que, par exemple, 1.323×10^{-23}. Il ne faudra pas s'en étonner et juste conclure que le résultat est extrêmement significatif et que l'hypothèse nulle a une probabilité infiniment faible de se produire.

2.3.5 Marge d'erreur et impact des effectifs sur l'intervalle de confiance

La marge d'erreur est une mesure statistique quantifiant l'incertitude liée à la taille de l'échantillon étudié. Plus l'échantillon est important, plus la marge d'erreur diminue définissant ainsi l'intervalle de confiance.

Prenons l'exemple d'un sondage politique pour illustrer ce concept : lorsqu'une personnalité politique est annoncée à 54 % avec un intervalle de confiance à 95 % de [51.9 ;56.1], cela signifie qu'elle a 95 % de chances d'avoir un score situé entre 51.9 % et 56.1 %. Nous conserverons toujours une possibilité de 5 % que son score soit en dehors de cet intervalle de confiance. Bien que tout soit fait pour la minimiser, les statistiques comportent toujours une part d'incertitude qu'il faut accepter.

Le graphique suivant illustre l'impact des effectifs sur l'étendue de l'intervalle de confiance dont l'échelle est exprimée ici en pourcentage décimalisé. Nous avons affaire à une courbe logarithmique décroissante avec une forte diminution initiale qui ralentit rapidement :



Le tableau suivant détaille précisément les valeurs des bornes inférieures et supérieures de l'intervalle de confiance en fonction de l'échantillon interrogé pour une proportion annoncée de 54 % :

Effectif		Borne inférieure	Borne supérieure
100		47.0 %	61.0 %
1000		51.8 %	56.2 %
10000		53.3 %	54.7 %

Remarque

L'article Wikipédia suivant détaille les formules de calcul de l'intervalle pour une proportion ou une moyenne :
https://fr.wikipedia.org/wiki/Intervalle_de_confiance

3. Modules Python pour l'analyse de données

Avant de choisir et de mettre en œuvre les différents tests statistiques, il est opportun de faire un point sur les possibilités offertes par les bibliothèques Python en la matière.

3.1 Les capacités limitées des modules classiques

Les librairies classiques telles que Pandas ou Numpy proposent, en plus de leurs fonctions pour manipuler les données, quelques fonctions pour calculer des indicateurs statistiques. Ces capacités se bornent aux statistiques descriptives ainsi qu'à la possibilité de calculer la corrélation et la covariance. Bien que certaines aient déjà été abordées précédemment, voici un tableau synthétique permettant de récapituler leurs possibilités :

Fonction	Pandas	NumPy
Moyenne	`df[« Column »].mean()`	`np.mean(array)`
Médiane	`df[« Column »].median()`	`np.median(array)`
Mode	`df[« Column »].mode()`	**Non disponible**
Écart-type	`df[« Column »].std()`	`np.std(array)`
Variance	`df[« Column »].var()`	`np.var(array)`
Minimum	`df[« Column »].min()`	`np.min(array)`
Maximum	`df[« Column »].max()`	`np.max(array)`
Résumé Statistique	`df[« Column »].describe()`	**Non disponible**
Percentiles	`df[« Column »].quantile(q)`	`np.percentile(array, q)`
Quantiles	`df[« Column »].quantile(q)`	`np.quantile(array, q)`
Asymétrie	`df[« Column »].skew()`	**Non disponible**
Aplatissement	`df[« Column »].kurt()`	**Non disponible**
Rang	`df[« Column »].rank()`	`np.argsort(array)`

Fonction	Pandas	NumPy
Matrice de Corrélation	`df[« Column »].corr()`	`np.corrcoef(array)`
Matrice de Covariance	`df[« Column »].cov()`	`np.cov(array)`

Remarque

Les corrélations seront abordées à la sous-section Corrélations entre variables numériques de ce chapitre. La covariance, qui évalue la manière dont deux variables varient ensemble, sera, en revanche, laissée de côté.

3.2 Les modules spécialisés en statistiques

Des modules Python ont été spécialement créés pour répondre aux besoins en termes de statistiques inférentielles. Nous aurons recours à deux d'entre eux qui couvriront largement en la matière : Scipy et Statmodels.

Remarque

Il existe un module nommé statistics, mais ses fonctions couvrent seulement les statistiques descriptives.

3.2.1 Scipy

Scipy est une bibliothèque Python proposant un vaste choix de fonctions scientifiques et techniques. Ici, nous allons utiliser le module stats qui est dédié aux statistiques descriptives et inférentielles.

Voici comment l'installer :

```
pip install scipy
```

Nous importerons ensuite le module stats qui répondra à nos besoins :

```
from scipy import stats
```

3.2.2 Statmodels

Statmodels est une bibliothèque dédiée aux statistiques notamment pour les estimations de modèles et la réalisation de tests. Cette librairie possède également des tableaux de synthèse très pratiques comme celui dédié à la régression linéaire que nous découvrirons dans le chapitre L'apprentissage supervisé.

Pour le moment, voici la commande pour l'installer :

```
pip install statmodels
```

Et la manière, un peu spécifique, de l'appeler :

```
import statsmodels.api as sm
```

Il est temps maintenant de mettre en pratique ces modules et de voir concrètement comment se pratique l'inférence statistique.

4. Tests statistiques de normalité

4.1 Contexte et objectif

Les tests de normalité de distribution constituent une étape fondamentale dans l'analyse des données, en particulier pour les variables numériques. Ils ont pour objectif de déterminer si la distribution d'une variable suit une loi normale. Cette vérification est essentielle, car elle influence le choix des méthodes statistiques à appliquer par la suite, afin de déterminer s'il existe un lien entre les variables. En effet, la normalité d'une variable conditionne le recours à des tests paramétriques, qui supposent une distribution normale des données, ou à des tests non paramétriques, mieux adaptés lorsque cette condition n'est pas remplie.

Avant de passer au test à proprement parler, il est intéressant de présenter les Quantile-Quantile plots (Q-Q Plots) qui offrent une manière intuitive et visuelle de représenter la normalité des données.

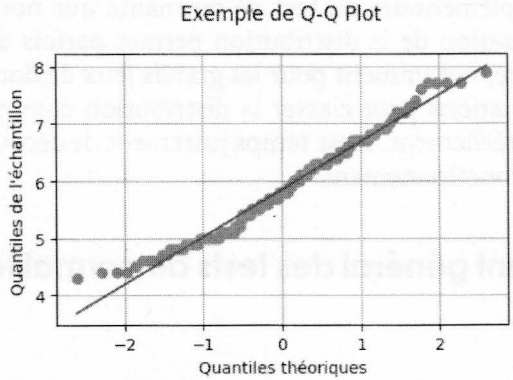
4.2 Les Q-Q plots

4.2.1 Définition et tracé du graphique

Les Q-Q plots (QQ pour Quantile-Quantile) sont des graphiques qui comparent les quantiles d'un jeu de données avec les quantiles d'une distribution théorique, souvent la distribution normale. En traçant les quantiles observés et les quantiles théoriques, nous pouvons aisément voir si nos données semblent normalement distribuées ou non.

Prenons un exemple que nous allons commenter pour comprendre comment utiliser ce type de visualisation :

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import scipy.stats as stats

# Import du jeu de données des iris
iris = sns.load_dataset("iris")

# Sélection d'une colonne numérique
data = iris['sepal_length']

# Création d'un Q-Q plot
plt.figure(figsize=(5, 3))
stats.probplot(data, dist= »norm », plot=plt)

# Modification de la couleur des points
dots = plt.gca().get_lines()[0]
dots.set_markerfacecolor('#439cc8') # Couleur des points
dots.set_markeredgecolor('#439cc8') # Couleur des bords des points

# Titres
plt.title(« Exemple de Q-Q Plot\n »)
plt.xlabel(« Quantiles théoriques »)
plt.ylabel(« Quantiles de l'échantillon »)
plt.grid(True)
plt.show()
```



Remarque

Il est parfaitement normal que les valeurs des quantiles en x et y soient différentes, car l'axe des x montre une distribution théorique centrée autour de 0 à plus ou moins deux écarts-types, tandis que l'axe des y représente nos données réelles, qui s'étendent de 4.3 à 7.9 avec une moyenne d'environ 5.84.

4.2.2 Interprétation

Regardons comment interpréter le graphique précédent :

- La ligne droite diagonale représente la distribution normale où chaque quantile théorique en x correspond à son quantile théorique en y, d'où la diagonale.
- Les valeurs de la variable sont représentées par des points de coordonnées « valeur théorique du quantile » en x et « valeur observée du quantile » en y.
- Si les points sont à peu près alignés sur la diagonale, la distribution a des chances d'être normale.
- Plus la forme générale des points diverge par rapport à la diagonale, plus l'anormalité de distribution est présente.
- Le graphique met bien en lumière les déviations et permet de détecter les outliers.

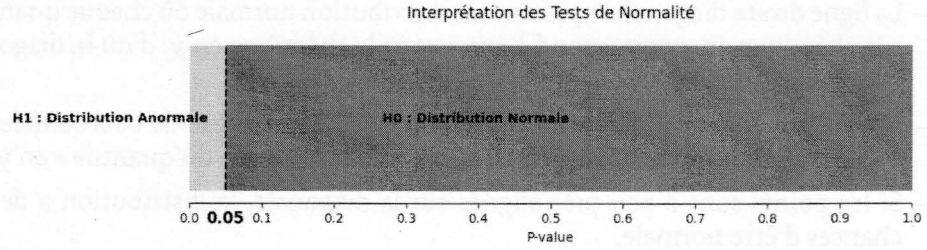
Ce type de visualisation est complémentaire du test de normalité que nous étudierons juste après. La visualisation de la distribution permet parfois de conserver l'hypothèse de normalité, notamment pour les grands jeux de données où la somme de petites déviations peut classer la distribution comme anormale alors qu'elle ne l'est pas réellement. Il est temps justement de découvrir les tests de normalité et leur fonctionnement.

4.3 Principe de fonctionnement général des tests de normalité

4.3.1 Principe de fonctionnement

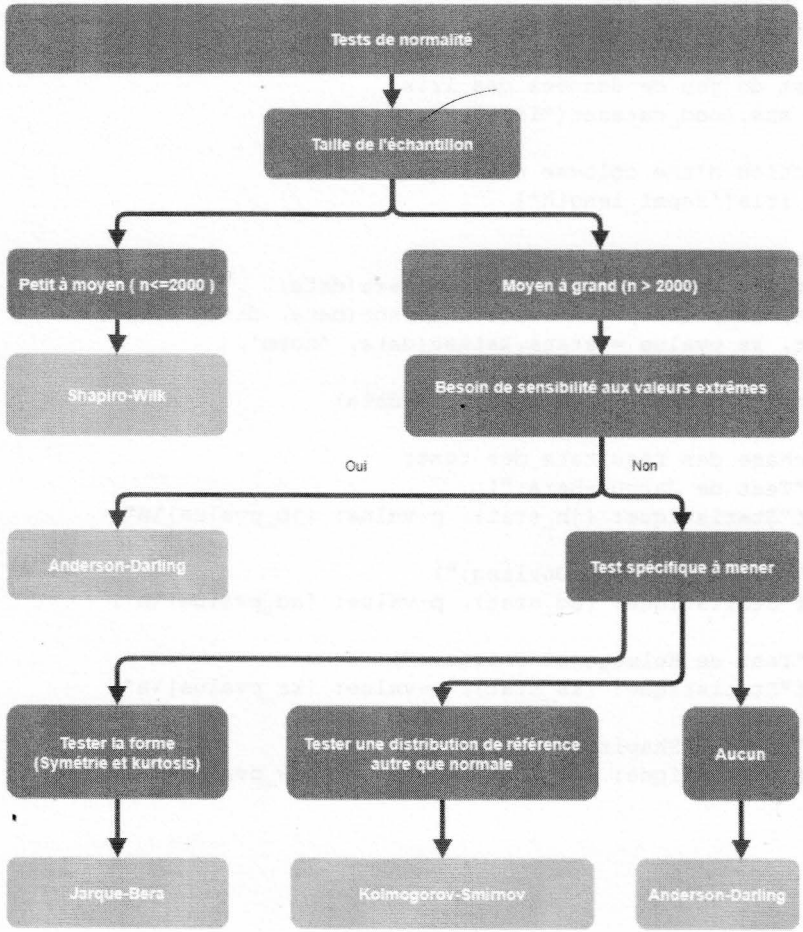
Le principe de fonctionnement général des tests de normalité repose sur les concepts de l'inférence statistique vue précédemment. Nous allons à nouveau formuler un test d'hypothèse basé sur H0 et H1 qui prennent ici les significations suivantes :

- L'hypothèse nulle H0 de normalité est retenue si la p-value renvoyée est strictement supérieure au seuil de significativité, souvent le seuil de 0.05. La distribution de la variable est alors considérée comme normale.
- Si la p-value est inférieure au seuil de significativité, c'est l'hypothèse H1 qui est retenue impliquant que la distribution de la variable n'est pas normale.



4.3.2 Les différents tests de normalité

Les tests de normalité permettent de déterminer si une distribution est normale. Différents tests sont disponibles pour répondre à diverses situations, tenant compte de différents facteurs comme la taille de l'échantillon ou la présence de valeurs extrêmes. Dans cette étude, nous examinerons les tests de Shapiro-Wilk, Anderson-Darling, Kolmogorov-Smirnov et Jarque-Bera, et fournirons un schéma illustrant leur utilisation recommandée en fonction des cas de figure spécifiques.



Le schéma précédent nous aidera à sélectionner le bon test de normalité à utiliser selon la situation. Il est utile de préciser qu'il est tout à fait possible de mettre en œuvre plusieurs tests. Par exemple, si notre échantillon est faible avec des valeurs extrêmes, il serait intéressant d'expérimenter le test de Shapiro-Wilk et d'Anderson-Darling.

Repartons du graphique du Q-Qplot précédent et calculons tous les tests de normalités à l'aide la bibliothèque Scipy :

```
import pandas as pd
import seaborn as sns
import scipy.stats as stats

# Import du jeu de données des iris
iris = sns.load_dataset("iris")

# Sélection d'une colonne numérique
data = iris['sepal_length']

# Tests de normalité
jb_stat, jb_pvalue = stats.jarque_bera(data)
ad_stat, ad_pvalue, _ = stats.anderson(data, dist='norm')
ks_stat, ks_pvalue = stats.kstest(data, 'norm',
args=(data.mean(), data.std()))
sw_stat, sw_pvalue = stats.shapiro(data)

# Affichage des résultats des tests
print("Test de Jarque-Bera:")
print(f"Statistique: {jb_stat}, p-value: {jb_pvalue}\n")

print("Test d'Anderson-Darling:")
print(f"Statistique: {ad_stat}, p-value: {ad_pvalue}\n")

print("Test de Kolmogorov-Smirnov:")
print(f"Statistique: {ks_stat}, p-value: {ks_pvalue}\n")

print("Test de Shapiro-Wilk:")
print(f"Statistique: {sw_stat}, p-value: {sw_pvalue}\n")
```

```
Test de Jarque-Bera:
Statistique: 4.485875437350938, p-value: 0.10614621817187797

Test d'Anderson-Darling:
Statistique: 0.8891994860134105, p-value: [0.562 0.64  0.767 0.895 1.065]

Test de Kolmogorov-Smirnov:
Statistique: 0.08865361377316228, p-value: 0.17813737848592026

Test de Shapiro-Wilk:
Statistique: 0.9760899543762207, p-value: 0.01018026564270258
```

Les résultats précédents méritent quelques explications :

- Pour les tests de Jarque-Bera et Kolmogorov-Smirnov, la distribution est considérée comme normale car les p-values respectives d'environ 0.106 et 0.178 sont supérieures à 0.05.
- Pour le test d'Anderson-Darling, l'interprétation est différente. Nous obtenons une liste des seuils de significativité à 15 %, 10 %, 5 %, 2.5 % et 1 %. Ces seuils de significativité représentent les niveaux de confiance à partir desquels l'hypothèse nulle selon laquelle les données suivent une distribution spécifique (normale dans notre cas) est rejetée. Dans notre cas, la statistique d'environ 0.889 est supérieure au seuil de 0.767 relatif aux 5 %, ce qui indique que la distribution est considérée comme anormale au seuil des 5 %.
- Enfin, le test de Shapiro-Wilk, avec une p-value d'environ 0.01, est inférieur à 0.05 et indique également que la distribution est anormale. Selon le schéma précédent, ce test est le plus approprié dans notre cas car nous avons un petit échantillon de 150 valeurs.

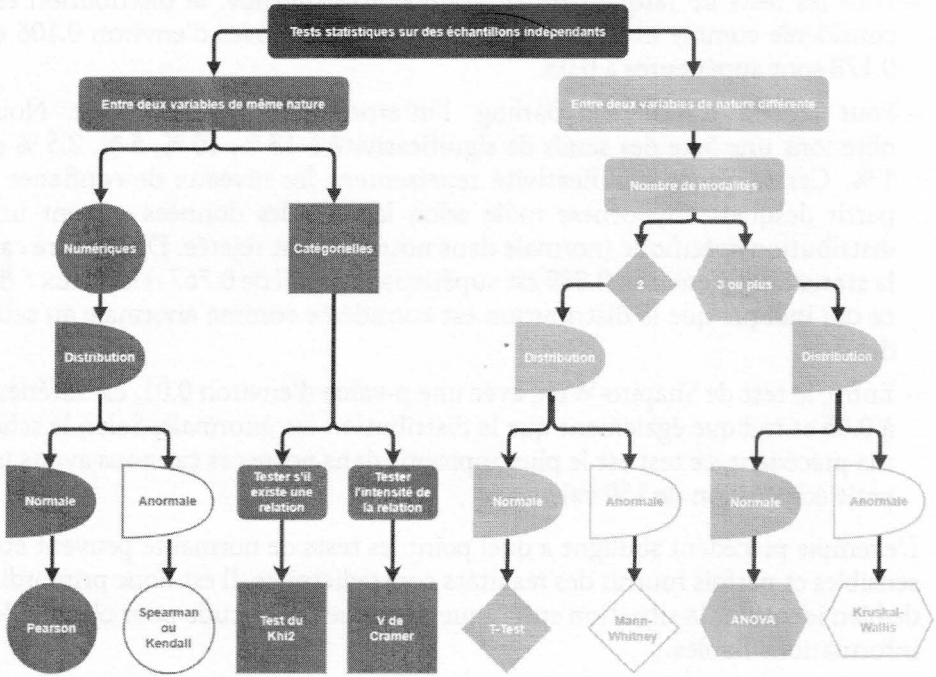
L'exemple précédent souligne à quel point les tests de normalité peuvent être sensibles et parfois fournir des résultats contradictoires. Il est donc primordial de bien identifier la situation spécifique à chaque cas d'étude pour obtenir des informations fiables.

Déterminer avec précision la normalité de la distribution est une étape essentielle, car elle influence le choix du type de test à mener pour évaluer les liens entre deux variables : paramétrique si la distribution est normale, non paramétrique dans le cas contraire. Nous allons voir dans la section suivante quels tests doivent être utilisés selon les cas.

5. Tests statistiques bivariés

Les tests statistiques bivariés examinent la relation entre deux variables pour identifier des associations ou des différences significatives, aidant ainsi à comprendre les liens potentiels et à formuler des hypothèses de recherche pour faciliter la prise de décision. Ils fournissent également une base pour des analyses plus complexes, telles que la régression et les modèles multivariés.

Nous allons suivre le schéma suivant pour les étudier en commençant par les tests portant sur deux variables de même nature puis ceux sur deux variables de nature différente.



Remarque

Nous nous concentrerons sur les tests portant sur des échantillons indépendants, c'est-à-dire des groupes qui sont totalement distincts les uns des autres. Il faut signaler qu'il existe également des tests sur les groupes appariés qui portent sur un même groupe où chaque observation est associée à une observation correspondante dans un autre groupe. C'est le cas, par exemple, pour un groupe avant et après un traitement médical. Afin de ne pas nous noyer sous les informations, ce cas de figure sera laissé de côté mais une simple recherche permettra de facilement trouver le test adéquat.

5.1 Tests bivariés entre des variables de même nature

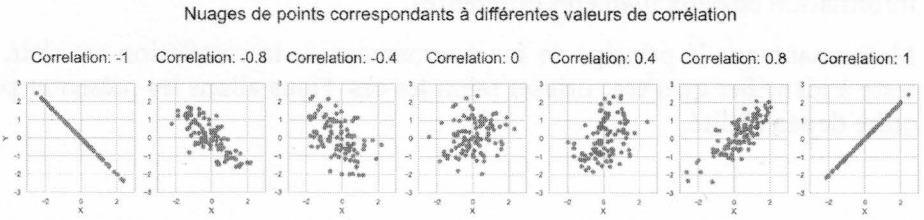
5.1.1 Corrélations entre variables numériques

Principe et enjeux de la corrélation

Débutons cette familiarisation avec les tests bivariés par les corrélations de type linéaire, c'est-à-dire lorsque la relation entre deux variables peut être décrite par une ligne droite.

Ces tests entre deux variables numériques sont primordiaux, car ils nous permettent de mesurer entre -1 et 1, la force du lien existant entre les deux variables. Une corrélation de 1 indique une relation parfaitement positive, -1 une relation parfaitement négative, et 0 aucune relation. Les corrélations sont donc essentielles pour identifier et quantifier les relations linéaires potentielles entre des variables, et sont souvent visualisées à l'aide de matrices de corrélation ou de nuages de points.

Voici un schéma illustratif montrant le coefficient de corrélation pour différentes relations entre deux variables :



Remarque

Il existe également des tests pour les relations non linéaires que nous aborderons plus bas avec la corrélation de Spearman et le tau de Kendall.

Le recours à la corrélation joue un rôle majeur dans la sélection des variables lors de l'analyse de données. Il est communément accepté qu'une corrélation inférieure à -0.9 ou supérieure à 0.9 indique des liens très forts entre les variables. Ces seuils de corrélation ont été sélectionnés en raison de leur capacité à mettre en évidence des relations linéaires ou inversement linéaires significatives entre les variables. Cependant, il est important de noter que ces seuils peuvent varier en fonction du contexte de l'analyse et des normes spécifiques à chaque domaine.

Identifier des corrélations trop fortes permet ainsi d'éviter la redondance d'informations et de réduire le bruit dans le modèle, ce qui a pour effet d'améliorer sa performance. Ainsi, lorsqu'une corrélation supérieure à 0.9 en valeur absolue est observée entre deux variables, il est souvent recommandé de les examiner de près et de décider si l'une des variables peut être supprimée ou si elles peuvent être fusionnées en une seule variable représentative.

Prenons un exemple simple pour illustrer : supposons que nous disposions d'un jeu de données comportant des centaines de variables, mais que nous manquions de temps pour les examiner individuellement de manière détaillée. En utilisant la corrélation, nous pourrions immédiatement identifier des relations trop fortes entre certaines d'entre elles. Par exemple, nous pourrions détecter facilement la présence de la température en degrés Celsius et en degrés Fahrenheit. La corrélation entre ces deux variables serait, en effet, presque égale à 1, étant donné qu'elles sont liées par l'équation linéaire : Fahrenheit = 9/5 * Celsius + 32. Identifier ces liens nous permettrait de supprimer l'une de ces deux variables pour éviter de délivrer la même information de deux manières différentes.

Maintenant que le principe de fonctionnement de la corrélation est clair, il reste à identifier quel test utiliser selon les cas. Nous allons les présenter par ordre de sensibilité.

La corrélation de Pearson

La corrélation de Pearson est le test qui vient naturellement à l'esprit lorsqu'il est question de corrélation. Il est conseillé lorsque la distribution est normale avec des valeurs continues et qu'il existe un lien linéaire entre les variables. Des trois, c'est le plus sensible des tests aux valeurs aberrantes et à la linéarité. Attention toutefois car il peut renvoyer des résultats trompeurs si la relation est non linéaire ou si des valeurs aberrantes importantes sont présentes.

Le tableau suivant présente les trois façons d'obtenir les corrélations de Pearson. Il est à noter que Pearson est le test par défaut dans Pandas et Numpy :

Module	Code pour un couple de variables	Code pour générer la matrice de corrélation	Commentaires

Pandas	`import pandas as pd`
	
	`# Variables en listes`
	`pd.Series(x).corr(pd.Series(y))`
	
	`# Variables d'un DataFrame`
	`correlation = df['col1'].corr(df['col2'])`
		`import pandas as pd`
		
		`# DataFrame nommé df`
		`df.corr(numeric_only=True)`
			La plus simple à utiliser

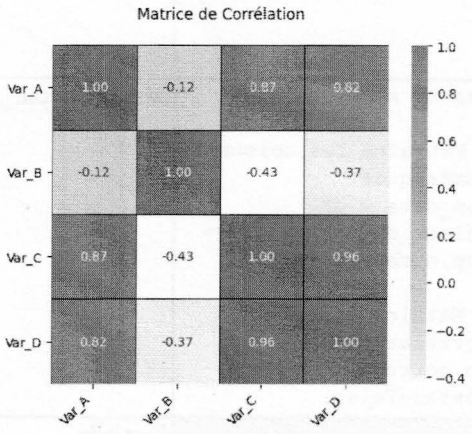
NumPy	`import numpy as np`
	
	`np.corrcoef(x, y)[0, 1]`
		`import numpy as np`
		
		`# Extraire les colonnes numériques`
		`num_data = df.select_dtypes(include=[np.number]).values`
		
		`# Matrice de corrélation`
		`np.corrcoef(num_data, rowvar=False)`
			Moins directe

SciPy	`from scipy.stats import pearsonr`
	
	`pearsonr(x, y)[0]`
		Non disponible directement	Partielle

L'usage de Pandas est ici vivement encouragé pour sa simplicité par rapport aux autres modules.

Dans le cas d'une matrice, une mise en forme apporte un vrai plus et facilite la détection des corrélations fortes. Voici un exemple de code partant d'un DataFrame quelconque :

```
# Calcul de la matrice de corrélation en spécifiant
numeric_only=True
correlation_matrix = df.corr(numeric_only=True)

# Création d'une heatmap
sns.heatmap(correlation_matrix,
            annot=True, fmt=".2f", cmap='Blues', cbar=True,
            square=True, linewidths=0.5, linecolor='black')

# Ajustement des étiquettes et du titre
plt.title('Matrice de Corrélation\n', fontsize=12)
plt.xticks(rotation=45, ha='right')
plt.yticks(rotation=0)

# Affichage de la figure
plt.show()
```



Remarque

La diagonale est toujours égale à 1 car elle correspond à la corrélation d'une variable par rapport à elle-même.

La corrélation de Spearman

La corrélation de Spearman consiste à calculer les corrélations à partir du rang des valeurs plutôt que des valeurs elles-mêmes comme dans le cas de Pearson.

C'est la solution à employer lorsque les liens entre les variables ne sont pas linéaires ou que les données ne sont pas normalement distribuées ou qu'il existe des valeurs aberrantes. En se basant sur les rangs, ce test est moins sensible que la corrélation de Pearson mais plus robuste. Il peut encore être influencé par les valeurs extrêmes mais moins que pour Pearson.

Ici encore, Pandas propose la façon la plus simple de le calculer, nécessitant simplement de rajouter `method = 'spearman'` dans la fonction `corr`. En revanche, soulignons que Numpy ne propose aucune solution pour ce test. Voici le tableau récapitulatif :

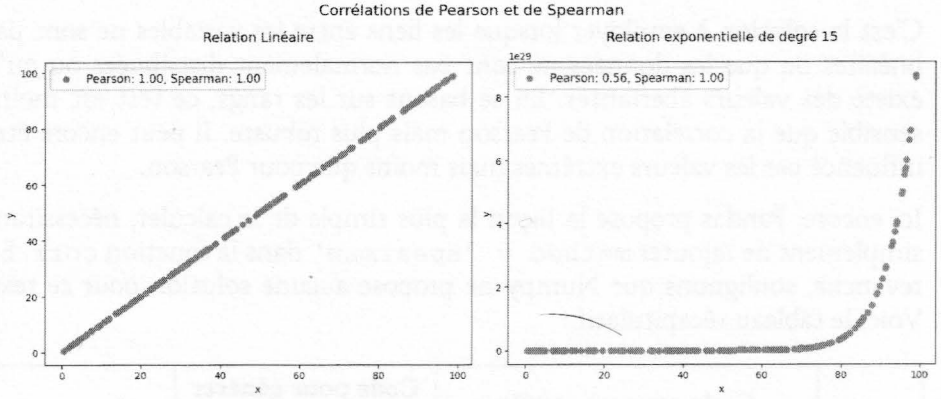
Module	Code pour un couple de variables	Code pour générer la matrice de corrélation	Commentaires

Pandas	`import pandas as pd`
	
	`# Variables en listes`
	`pd.Series(x).corr(pd.Series(y),method='spearman')`
	
	`# Variables d'un DataFrame`
	`correlation = df['col1'].corr(df['col2'], method='spearman')`
		`import pandas as pd`
		
		`# DataFrame nommé df`
		`df.corr(method='spearman', numeric_only=True)`
			La plus simple à utiliser

NumPy		**Non disponible**	

SciPy	`from scipy.stats import spearmanr`
	
	`spearmanr(x, y)[0]`
		Non disponible directement	Partielle

Afin de mesurer la force de Spearman sur Pearson, nous allons regarder la valeur des corrélations dans une situation linéaire comparée à un cas non linéaire (un exposant 15 pour bien accentuer le caractère exponentiel) :



Si les deux corrélations affichent la même valeur de 1 sur une relation strictement linéaire, dans le cas d'un lien exponentiel, le coefficient de Pearson décroche complètement. En utilisant uniquement cet indicateur, nous n'aurions pas identifié la corrélation trop forte entre les deux variables car la valeur de 0.56 pour Pearson est raisonnable. En revanche, l'utilisation du coefficient de Spearman, plus adapté aux relations non linéaires, nous aurait permis d'identifier la relation forte entre les variables. Conserver uniquement la version du coefficient de Pearson aurait très certainement eu un impact négatif sur notre modélisation en sous-estimant la véritable nature de la relation entre les variables.

Dernier point concernant la corrélation de Spearman : si les variables numériques d'un jeu de données étudié présentent des distributions normales pour certaines et anormales pour d'autres, il est plus raisonnable de demander une matrice de confusion avec la méthode Spearman plutôt que Pearson.

Le tau de Kendall

Le tau de Kendall est un test de corrélation moins connu qui mesure, comme les coefficients de Pearson et de Spearman, la force et la direction de l'association entre deux variables, avec une valeur comprise entre -1 et 1.

Il repose sur la comparaison de toutes les paires de données, classées comme concordantes ou discordantes. Une paire de points est dite concordante si les rangs des deux variables augmentent ou diminuent ensemble. En revanche, elle est considérée comme discordante si le rang d'une variable augmente tandis que celui de l'autre diminue.

Cette approche par rang de paires rend le test encore plus robuste que le coefficient de Spearman aux valeurs aberrantes, à la présence d'ex aequo, et devient particulièrement adaptée lorsque le nombre d'observations est très faible.

Pour l'utiliser avec Pandas, il faut simplement mettre `method='kendall'`.

Pour Scipy, il faut importer la fonction spécifique. Voici un exemple tout simple portant sur un très faible volume de données qui se révèle parfaitement significatif :

```
from scipy.stats import kendalltau

# Exemple de données
x = [10, 20, 15, 30, 25]
y = [5, 15, 10, 25, 20]

# Calcul du coefficient de corrélation de Kendall
tau, p_value = kendalltau(x, y)

print(f"Coefficient de corrélation de Kendall : {tau}")
print(f"Valeur de p associée : {p_value}")

# Output:
# Coefficient de corrélation de Kendall : 0.9999999999999999
# Valeur de p associée : 0.016666666666666666
```

5.1.2 Tests d'indépendance entre variables catégorielles

Principes des tests d'indépendance

Les tests d'indépendance entre variables catégorielles représentent l'autre grand domaine des tests statistiques bivariés appliqués aux variables de même nature. Ici, c'est l'association entre deux variables catégorielles qui va être évaluée. Contrairement aux tests de corrélation, il n'est pas nécessaire d'évaluer la normalité des données pour ces tests.

Nous étudierons ces tests en commençant par le test du Khi2 et le test exact de Fisher, qui visent à évaluer l'existence d'un lien entre deux variables catégorielles. Ensuite, nous aborderons le V de Cramer et le coefficient Phi, qui permettent de mesurer la force de ce lien de manière analogue aux corrélations.

Le test du Khi2

Le test du Khi2 vise à évaluer l'existence d'un lien entre deux variables qualitatives. Il compare la distribution observée des données à une distribution théorique attendue sous l'hypothèse d'indépendance des variables. Si la distance entre ces deux distributions est suffisamment importante, l'hypothèse nulle (H0) d'indépendance est rejetée en faveur de l'hypothèse alternative (H1) de dépendance.

Remarque

Il faut au moins 30 observations pour appliquer le test du Khi2.

Nous allons mettre en pratique le Khi2 sur l'étude réalisée par Stack Overflow pour vérifier si l'âge et l'utilisation de l'IA dans le processus de développement sont liés. Voici comment réaliser le test étape par étape :

Commençons par récupérer les données :

```
#Lien du site pour récupérer les données en csv
#https://survey.stackoverflow.co/

import pandas as pd

# Remplacer your_path par le chemin complet de votre dossier
df = pd.read_csv(r"your_path\survey_results_public.csv")

# Affichage de toutes les colonnes
pd.set_option('display.max_columns', 999)
```

Nous allons ensuite réaliser la table de contingence entre l'âge et l'utilisation de l'IA grâce à la fonction `crosstab` de Pandas :

```
contingency_table = pd.crosstab(df['Age'], df['AISelect'])
contingency_table
```

AISelect Age	No, and I don't plan to	No, but I plan to soon	Yes
18-24 years old	4119	3570	9983
25-34 years old	9079	8747	15137
35-44 years old	6588	5816	7910
45-54 years old	3071	2462	2676
55-64 years old	1493	994	840
65 years or older	617	306	207
Prefer not to say	155	73	101
Under 18 years old	1099	742	2188

Enfin, nous produisons, à l'aide de la fonction `chi2_contingency` de scipy.stats, tous les indicateurs du Khi2 :

```
from scipy.stats import chi2_contingency

# Calcul du test du Khi2
chi2_stat, p_val, dof, expected =
chi2_contingency(contingency_table)

# Affichage des résultats
print("\nValeur du Khi2 :", chi2_stat)
print("Degrés de liberté :", dof)
print("P-value :", p_val)

# Output:
# Valeur du Khi2 : 3035.354600788779
# Degrés de liberté : 14
# P-value : 0.0
```

Détaillons les résultats obtenus :

- La valeur du Khi2 d'environ 3035 correspondant à la somme des écarts par case entre les effectifs observés et les effectifs théoriques.
- Les degrés de liberté correspondent au produit du nombre de modalités en ligne -1 par le nombre de modalités en colonne -1.

  Soit, dans notre cas (8-1) * (3-1) = 14. Ils permettent d'établir les seuils en fonction de la taille de la matrice.

- Enfin, la p-value nous indique qu'au seuil des 5 %, la probabilité que les deux variables soient indépendantes est nulle (0.0 %) donc l'âge et l'usage de l'IA sont bien dépendants.

Il est possible, pour finir, d'obtenir les valeurs attendues qui sont stockées dans la variable *expected*.

Le test de Fisher

Le test de Fisher est recommandé pour les matrices 2X2 avec moins de 30 observations au total. Il fonctionne de manière similaire au test du Khi2 mais est plus fiable pour les petits effectifs en raison de sa précision avec les faibles fréquences. Il est accessible grâce au module Scipy :

```
from scipy.stats import fisher_exact

# Création d'un tableau de contingence (2x2)
# Exemple de données: [ [A oui, A non], [B oui, B non] ]
tableau_contingence = [[10, 2],
                       [1, 5]]

# Test de Fisher
oddsratio, p_value = fisher_exact(tableau_contingence)

# Affichage des résultats
print(f"Rapport de cotes (odds ratio) : {oddsratio}")
print(f"Valeur p : {p_value}")

# Output:
# Rapport de cotes (odds ratio) : 25.0
# Valeur p : 0.012820512820512822
```

Dans cet exemple fictif, deux informations sont produites :

- Le rapport de cotes qui permet de comparer les chances (les cotes) qu'un événement se produise entre deux groupes. C'est le rapport des rapports des modalités des groupes. Ici, la valeur de 25 indique que l'événement « oui » a 25 fois plus de chances de se produire dans le groupe A que dans le B. Le odds ratio fonctionne de la façon suivante :
  - Un odds ratio de 1 signifie que l'événement « oui » a autant de chances de se produire en A qu'en B.

  - Un odds ratio supérieur à 1 indique que l'événement est plus probable dans le premier groupe.
  - Un odds ratio inférieur à 1 indique que l'événement est plus probable dans le deuxième groupe.
- La p-value d'environ 0.0128, inférieure à 0.05, indique qu'il existe une association significative entre la variable `Groupe` et la variable `Oui_Non`.

Ces deux tests indiquent la présence d'un lien mais ne fournissent aucune information sur la force de ce lien, contrairement aux corrélations. Pour répondre à ce besoin, nous pouvons utiliser le V de Cramer.

Le V de Cramer

Le V de Cramer est une mesure utilisée pour évaluer la force de l'association entre deux variables catégorielles à partir de tableaux de contingence de toute taille. Il permet de quantifier la force de l'association déjà établie par le test du Khi2. Cependant, contrairement au coefficient de corrélation, il n'indique pas de direction dans cette association. Le V de Cramer varie ainsi de 0 à 1, où une valeur plus élevée indique une association plus forte, mais il ne peut pas être négatif comme dans le cas des coefficients de corrélation. Voici un tableau permettant d'établir, de manière empirique, la force du lien selon la valeur du V de Cramer :

Valeur du V de Cramer	**Intensité de la relation entre les deux variables**
< 0.10	Nulle à très faible
>= 0.10 et < 0.30	Faible
>= 0.30 et < 0.50	Modéré
>= 0.50	Fort

Il n'existe pas de fonction dédiée au V de Cramer mais son calcul, relativement simple, ne nécessite que trois lignes de codes et correspond à la racine carrée du Khi2 divisé par le Khi2 maximal. Le Khi2 maximal est égal au produit de l'effectif par la valeur de la plus petite dimension du tableau de contingence -1. Nous allons reprendre l'exemple de l'enquête de Stack Overflow au début de la sous-section Tests d'indépendance entre variables catégorielles, pour mesurer la force du lien entre la tranche d'âge et l'usage de l'IA et détailler pas à pas le calcul :

```
# Calcul du V de Cramer

# Table de contingence sur les variables Age et AISelect
contingency_table = pd.crosstab(df['Age'], df['AISelect'])

# 1 / Somme totale de la table de contingence
n = contingency_table.values.sum()

# 2 / Calcul du Khi2 théorique maximal
khi2max = n * (min(contingency_table.shape) -1)

# 2 / Calcul du V de Cramer
v_cramer = np.sqrt(chi2 / khi2max)

# Affichage des résultats
print(f"V de Cramer : {v_cramer}")

# Output: V de Cramer : 0.13134544556843927
```

Le lien entre âge et usage de l'IA existe mais le V de Cramer nous aide à tempérer la portée de ce résultat en soulignant qu'il est faible. C'est très important de pouvoir évaluer la force d'un phénomène car l'existence seule du lien n'apporte que peu d'information.

Passons maintenant à la deuxième partie des tests bivariés portant sur deux variables de natures différentes.

5.2 Tests bivariés entre des variables de nature différente

Les tests bivariés portant sur des variables de natures différentes consistent majoritairement à évaluer comment évolue une variable numérique en fonction des différentes modalités d'une variable catégorielle. Deux principaux paramètres vont jouer dans le choix du test : le nombre de modalités et la distribution de la variable numérique.

5.2.1 Tests de comparaison à deux modalités

Débutons par le cas le plus simple entre une variable numérique et une variable catégorielle contenant deux modalités de réponses.

Le t-test

Le t-test, également appelé test de Student, évalue s'il existe une différence significative entre les moyennes de deux échantillons d'une variable distribuée normalement. Il est donc essentiel de tester la normalité avant de choisir le test approprié.

Dans le cas du t-test où la normalité est confirmée, nous sommes en présence d'un test paramétrique. Cela signifie que le test repose sur des hypothèses concernant la façon dont les données sont réparties, comme le fait qu'elles suivent une courbe en cloche (distribution normale), ce qui permet ainsi d'estimer les paramètres de la moyenne et de l'écart-type pour évaluer la significativité.

Regardons l'application d'un t-test sur le jeu des iris pour lequel nous ne prendrons que les observations appartenant aux deux premières espèces :

```
import seaborn as sns
import pandas as pd
from scipy.stats import shapiro, ttest_ind

# Chargement des données
df0 = sns.load_dataset("iris")

first_two_species = df0['species'].unique()[:2]
df = df0[df0['species'].isin(first_two_species)]

# Test de normalité sur sepal_width avec Shapiro-Wilk
stat, p_value_shapiro = shapiro(df["sepal_width"])
```

```
print(f"Statistique de test : {stat}")
print(f"p-value : {p_value_shapiro}")

# Output:
# Statistique de test : 0.9897729754447937
# p-value : 0.6462556719779968
```

Avec une p-value d'environ 0.646, la distribution est normale. Nous allons pouvoir appliquer le t-test après avoir scindé les données par modalité :

```
# Séparation des valeurs par modalité
g1 = df[df["species"] == "setosa"]["sepal_width"]
g2 = df[df["species"] == "versicolor"]["sepal_width"]

# Test d'indépendance entre les deux modalités
t_stat, p_value_ttest = ttest_ind(g1, g2)

print(f"Statistique de test : {t_stat}")
print(f"p-value : {p_value_ttest}")

# Output:
# Statistique de test : 9.454975848128596
# p-value : 1.8452599454769322e-15
```

Avec une p-value très inférieure à 0.05, nous pouvons conclure qu'il y a une différence significative entre les moyennes des deux espèces d'iris. Cela indique que les deux espèces présentent des largeurs de sépales significativement différentes.

Le test de Mann-Whitney

Le test de Mann-Whitney est l'alternative au t-test lorsque la distribution n'est pas normale. En tant que test non paramétrique basé sur des rangs, il ne nécessite pas de distribution normale, il est plus robuste aux outliers et aussi lorsque les échantillons évalués sont de taille différente, ce qui peut être un critère important dans certains cas.

Reprenons l'exemple précédent en sélectionnant une autre variable numérique anormalement distribuée comme `petal_length` :

```
# Test de normalité sur petal_length avec Shapiro-Wilk
stat, p_value_shapiro = shapiro(df["petal_length"])

print(f"Statistique de test : {stat}")
```

```
print(f"p-value : {p_value_shapiro}")

# Output:
# Statistique de test : 0.8018537759780884
# p-value : 2.823854483580135e-10
```

Comme nous le confirme le test de Shapiro-Wilk, la distribution de `petal_length` est anormale avec une p-value largement inférieure à 0.05. Essayons maintenant le test de Mann-Whitney :

```
from scipy.stats import mannwhitneyu

# Séparation des valeurs par modalité
g1 = df[df["species"] == "setosa"]["petal_length"]
g2 = df[df["species"] == "versicolor"]["petal_length"]

# Test d'indépendance de Mann-Whitney entre les deux modalités
u_stat, p_value_mannwhitney = mannwhitneyu(g1, g2,
alternative='two-sided')  # alternative='two-sided' pour un
test bilatéral

print(f"Statistique de test U : {u_stat}")
print(f"p-value : {p_value_mannwhitney}")

# Output:
# Statistique de test U : 0.0
# p-value : 5.651011584290147e-18
```

Le test confirme que les deux groupes setosa et versicolor présentent des longueurs de pétales significativement différentes. Nous avons utilisé ici l'option de test bilatéral `two-sided` pour tester toute différence significative entre les deux groupes. Mais il serait également envisageable d'évaluer l'infériorité ou la supériorité d'un groupe par rapport à l'autre avec respectivement les options `less` et `greater`.

5.2.2 Tests de comparaison à trois modalités ou plus

Les tests de comparaison à trois modalités ou plus s'inscrivent dans la même démarche que ceux à deux modalités en présentant d'une part un versant paramétrique, le test d'ANOVA, et d'autre part un versant non paramétrique, le test de Kruskal-Wallis.

Le test d'ANOVA

Dans le cadre d'une variable numérique distribuée normalement, le test d'ANOVA (Analyse de la Variance) permet de comparer les moyennes de plusieurs groupes définis par une variable qualitative avec trois modalités ou plus, afin de déterminer s'il existe des différences significatives entre ces groupes.

Pour ce faire, la statistique F est calculée. Elle correspond au rapport de la variance entre les groupes à la variance à l'intérieur des groupes.

Une statistique F élevée suggère que la variabilité entre les groupes est beaucoup plus grande que la variabilité à l'intérieur des groupes, ce qui indique des différences significatives entre les moyennes des groupes.

Inversement, une statistique F faible est le signe d'une variance entre les groupes plus faible que la variance à l'intérieur des groupes, soit de faibles différences entre les moyennes des groupes.

Il est important de préciser que le test n'apporte qu'une réponse globale sur l'incidence de la variable catégorielle testée sur les moyennes des sous-groupes correspondants.

Reprenons notre exemple sur les iris, mais cette fois sur l'ensemble du jeu de données, et comparons la largeur des sépales des trois espèces :

```
import seaborn as sns
import pandas as pd
from scipy.stats import shapiro, ttest_ind

# Chargement des données
df = sns.load_dataset("iris")

# Test de normalité sur sepal_width avec Shapiro-Wilk
stat, p_value_shapiro = shapiro(df["sepal_width"])

print(f"Statistique de test : {stat}")
print(f"p-value : {p_value_shapiro}")

Output:
# Statistique de test : 0.9849168062210083
# p-value : 0.10112646222114563
```

La distribution de `sepal_width` reste normale sur l'ensemble des données ouvrant la voie à l'usage de l'ANOVA. Nous allons proposer ici une façon plus automatique de récupérer les valeurs par mention qui fera gagner de nombreuses lignes de codes pour les variables catégorielles avec de nombreuses modalités :

```
from scipy.stats import f_oneway

# Liste des modalités uniques de la variable
modalites = df["species"].unique()

# Liste vide pour stocker les séries de données
data_groups = []

# Boucle sur chaque modalité
for modalite in modalites:
    # Filtrage pour chaque modalité
    filtered_df = df[df["species"] == modalite]

    # Extraction de la colonne "sepal_width"
    filtered_data = filtered_df["sepal_width"]

    # Ajout des données filtrées à la liste
    data_groups.append(filtered_data)

# ANOVA sur les données groupées
anova_result = f_oneway(*data_groups)
print(anova_result)

# Output: F_onewayResult(statistic=49.160040089612075,
pvalue=4.492017133309115e-17)
```

Le test se révèle significatif indiquant des différences notoires entre les moyennes de `sepal_width` par espèce.

Signalons également la possibilité de réaliser un ANOVA avec Statmodels qui propose un code plus concis et des résultats plus complets :

```
from statsmodels.formula.api import ols
import statsmodels.api as sm

# ANOVA
model = ols('sepal_width ~ species', data=df).fit()

anova_table = sm.stats.anova_lm(model, typ=2)
```

```
print(anova_table)

# Output:
#                sum_sq     df          F        PR(>F)
# species     11.344933    2.0   49.16004  4.492017e-17
# Residual    16.962000  147.0        NaN           NaN
```

Nous retrouvons les mêmes résultats qu'avec Scipy mais l'information sur les résidus offre la possibilité d'évaluer la part de variance expliquée par le système en divisant la variable `species` par la somme de species + Residual, soit :

11.344933 / (11.344933 + 16.962) = 0.4008 soit 40.08 % de variance expliquée

Remarque

L'option `typ=2` qui produit uniquement les informations les plus importantes est recommandée pour sa simplicité d'interprétation.

Le test d'ANOVA nous apporte finalement une réponse globale, à savoir qu'il existe au moins une différence significative entre les groupes comparés, mais sans préciser la ou les paires concernées. Pour obtenir une réponse plus détaillée, nous allons faire connaissance avec le test de Tukey.

Le test de Tukey

Le test de Tukey est une méthode utilisée dans l'ANOVA pour effectuer des comparaisons par paires de moyennes et cibler celles qui sont significativement différentes.

De manière similaire au Khi2, une valeur critique de différence est établie en fonction du nombre de groupes, du nombre d'observations et du niveau de confiance arrêté. Les paires dépassant ce seuil sont ainsi identifiées.

Reprenons notre exemple des iris pour mettre en œuvre ce test :

```
from statsmodels.stats.multicomp import pairwise_tukeyhsd

# Test de Tukey pour les comparaisons multiples
tukey = pairwise_tukeyhsd(
    endog=df['sepal_width'],
    groups=df['species'],
    alpha=0.05
```

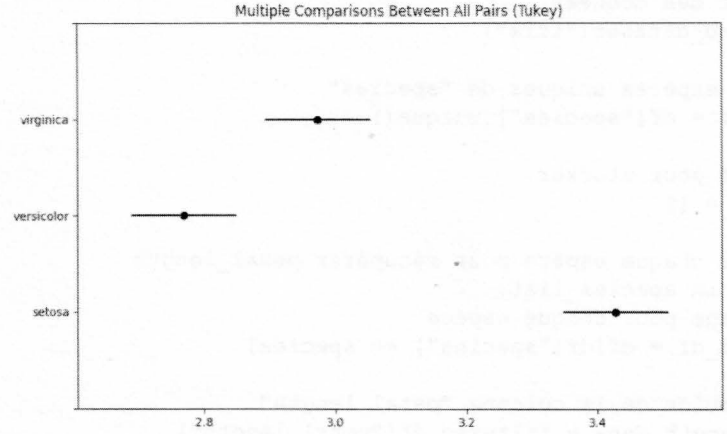
```
)

print(tukey)

# Visualisation des résultats du test de Tukey
tukey.plot_simultaneous();
```

Remarque

Ne pas oublier le point-virgule après la dernière commande graphique pour éviter d'obtenir un affichage en double dans certains systèmes.

Multiple Comparison of Means - Tukey HSD, FWER=0.05

group1	group2	meandiff	p-adj	lower	upper	reject
setosa	versicolor	-0.658	0.0	-0.8189	-0.4971	True
setosa	virginica	-0.454	0.0	-0.6149	-0.2931	True
versicolor	virginica	0.204	0.0088	0.0431	0.3649	True



La fonction `pairwise_tukeyhsd` nous délivre un tableau par paires de modalités. Ici toutes les différences entre les paires sont significatives (p-adj<0.05) et marquées `reject = True` indiquant que l'hypothèse H0 d'égalité des moyennes est rejetée au profit de l'hypothèse alternative H1.

La fonction `plot_simultaneous()` nous permet en plus de visualiser les moyennes de chaque modalité avec leurs marges d'erreur.

L'alliance du test d'ANOVA et de Tukey va ainsi devenir un puissant indicateur pour détecter quelles variables catégorielles ont une incidence sur une variable numérique spécifique et surtout, quelles modalités sont à considérer.

Le test de Kruskal-Wallis

Le test de Kruskal-Wallis est l'alternative non paramétrique au test d'ANOVA lorsque la variable numérique étudiée est anormalement distribuée. Il se base sur les rangs des observations pour évaluer si les médianes des groupes sont égales ou non.

Il présente l'avantage d'être robuste aux conditions de distribution et d'être utilisable aussi sur des échantillons de petite taille. En revanche, il s'avère moins précis que l'ANOVA et surtout, ne propose pas d'équivalent au test de Tukey qui permet un examen précis des couples de modalités.

```
from scipy.stats import kruskal
import seaborn as sns

# Chargement des données
df = sns.load_dataset("iris")

# Liste des espèces uniques de "species"
species_list = df["species"].unique()

# Liste vide pour stocker
data_groups = []

# Boucle sur chaque espèce pour récupérer petal_length
for species in species_list:
    # Filtrage pour chaque espèce
    filtered_df = df[df["species"] == species]

    # Extraction de la colonne "petal_length"
    petal_length_data = filtered_df["petal_length"]

    # Ajout des données filtrées à la liste
    data_groups.append(petal_length_data)

# Test de Kruskal-Wallis
kruskal_result = kruskal(*data_groups)

print(kruskal_result)

# Output: KruskalResult(statistic=130.41104857977163,
pvalue=4.803973591157605e-29)
```

De manière analogue au test ANOVA, le test de Kruskal-Wallis nous retourne la statistique et la p-value qui, ici, indiquent au moins une différence significative entre deux paires des moyennes.

5.2.3 Conclusions sur les tests bivariés

À l'issue de ce point sur les tests bivariés, nous sommes désormais en capacité de sélectionner le test adapté en fonction des données. Cette analyse est subtile mais en se référant au schéma fourni au début de ce chapitre, obtenir la bonne information nécessite uniquement de la rigueur.

Cette section est essentielle, car identifier les variables liées à notre objet d'étude maximise nos chances de réussir une modélisation correcte. Pour faciliter la compréhension, nous avons volontairement écarté les formules des algorithmes, mais il sera important de les revisiter ultérieurement pour approfondir notre expertise sur le sujet.

Après avoir vu comment précisément identifier les liens entre deux variables, nous allons voir comment acquérir une vision du fonctionnement global des données avec l'analyse multivariée.

6. Analyse multivariée

L'analyse multivariée a pour objectif d'examiner simultanément plusieurs variables pour comprendre les relations complexes et les interactions qui existent entre elles. Elle permet de révéler des patterns cachés et de fournir une vue d'ensemble cohérente sur le fonctionnement global d'un jeu de données. C'est une étape essentielle qui finalise l'analyse des données et permet d'identifier avec certitude les variables les plus importantes et les lois qui régissent le jeu de données étudié.

Avant d'aborder l'Analyse en Composantes Principales (ACP), qui constitue un pilier fondamental de l'analyse multivariée, nous explorerons deux autres techniques qui répondent à des besoins spécifiques : l'analyse de la variance multivariée (MANOVA) et l'Analyse des Correspondances Multiples (ACM).

6.1 Analyse de la variance multivariée (MANOVA)

6.1.1 Présentation et champs d'applications

La MANOVA est une généralisation de l'ANOVA qui vise à déterminer si une ou des variables catégorielles ont des liens avec les moyennes de plusieurs variables numériques dépendantes. Elle est particulièrement utile pour évaluer les effets combinés de plusieurs facteurs et mieux comprendre les relations complexes entre plusieurs variables.

6.1.2 Cas pratique d'utilisation

Prenons le jeu de données des diamants fourni par Seaborn :

```
import seaborn as sns

df = sns.load_dataset("diamonds")
df.head()
```

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

Il serait intéressant de savoir si les variables qualitatives décrivant la coupe et la couleur (`cut` et `color`) ont un impact sur des variables numériques comme les carats et le prix (`carat` et `price`). C'est l'occasion de mettre en œuvre une MANOVA en utilisant le module statsmodels :

```
from statsmodels.multivariate.manova import MANOVA

# Formule pour MANOVA
formula = 'carat + price ~ cut + color'

# Ajustement du modèle MANOVA
manova = MANOVA.from_formula(formula, data=df)
result = manova.mv_test()
```

```
# Affichage des résultats
print(result)
```

```
                 Multivariate linear model
================================================================

----------------------------------------------------------------
       Intercept         Value  Num DF   Den DF    F Value   Pr > F
----------------------------------------------------------------
          Wilks' lambda 0.7266 2.0000 53928.0000 10146.0514 0.0000
          Pillai's trace 0.2734 2.0000 53928.0000 10146.0514 0.0000
  Hotelling-Lawley trace 0.3763 2.0000 53928.0000 10146.0514 0.0000
     Roy's greatest root 0.3763 2.0000 53928.0000 10146.0514 0.0000
----------------------------------------------------------------

----------------------------------------------------------------
          cut            Value  Num DF   Den DF    F Value   Pr > F
----------------------------------------------------------------
          Wilks' lambda 0.9157 8.0000 107856.0000  606.8951 0.0000
          Pillai's trace 0.0848 8.0000 107858.0000  596.9893 0.0000
  Hotelling-Lawley trace 0.0915 8.0000  77037.6735  616.8121 0.0000
     Roy's greatest root 0.0850 4.0000  53929.0000 1145.7359 0.0000
----------------------------------------------------------------


----------------------------------------------------------------
         color           Value   Num DF   Den DF    F Value   Pr > F
----------------------------------------------------------------
          Wilks' lambda 0.8157 12.0000 107856.0000  963.9377 0.0000
          Pillai's trace 0.1853 12.0000 107858.0000  917.5670 0.0000
  Hotelling-Lawley trace 0.2249 12.0000  83884.9136 1010.5293 0.0000
     Roy's greatest root 0.2197  6.0000  53929.0000 1974.8367 0.0000
================================================================
```

L'algorithme MANOVA utilise différents tests pour valider ou non les liens. Dans notre cas, la constante (Intercept) ainsi que les deux variables catégorielles ont bien un lien significatif avec les variables numériques `price` et `carat` car elles présentent toutes une p-value inférieure à 0.05. L'effet combiné des deux variables catégorielles est ainsi validé.

6.2 Analyse en composantes multiples (ACM)

La cartographie des variables catégorielles

L'analyse en composantes multiples (ACM) est une méthode à privilégier lorsque les données sont composées de variables catégorielles. À partir d'un tableau de contingence qui croise les différentes modalités, les axes principaux sont calculés pour établir de nouvelles coordonnées. Cela permet de représenter graphiquement les mentions des différentes variables et des observations en deux ou trois dimensions, créant une sorte de cartographie où les modalités les plus proches sur le graphique sont celles qui ont le plus d'affinités. Notons qu'il est également envisageable d'insérer des variables numériques mais elles devront au préalable être regroupées en catégories.

Avant de représenter les mentions, il est important de contrôler le niveau de variance cumulée expliquée afin de s'assurer d'interpréter correctement les proximités entre les points.

Pour démontrer la puissance de l'ACM, nous allons l'utiliser sur le jeu des iris dans le but d'obtenir une vision d'ensemble des caractéristiques des espèces par rapport à la longueur des pétales et des sépales. Nous aurons pris soin, au préalable, de regrouper les variables numériques en trois parties égales nommées Courte, Moyenne et Longue.

Pour déployer l'ACM, nous allons utiliser le module Prince :

```
pip install prince
```

```
import pandas as pd
import prince
import matplotlib.pyplot as plt

# Fonction qui découpe une numérique en 3 parties égales
def cut_three(dataframe, var):
    # Création du nouveau nom de colonne
    new_name = var + "_cat"

    # Découpage de la colonne en 3 catégories
    dataframe[new_name] = pd.cut(
        dataframe[var],
        bins=3,
        labels=['Courte', 'Moyenne', 'Longue']
    )
```

```
    # Suppression de l'ancienne colonne
    dataframe = dataframe.drop(var, axis=1)

    return dataframe

# Import des données
df = sns.load_dataset("iris")

# Transformation en données catégorielles
df = cut_three(df,"sepal_length")
df = cut_three(df,"petal_length")

# Seules les variables catégorielles sont conservées
df = df.select_dtypes(include=['object', 'category'])

# Initialisation de la MCA
mca = prince.MCA(n_components=2, random_state=42)

# Entraînement sur les données
mca = mca.fit(df)

# Application sur les données
mca_data = mca.transform(df)

# Coordonnées des modalités
mod_coords = mca.column_coordinates(df)
```

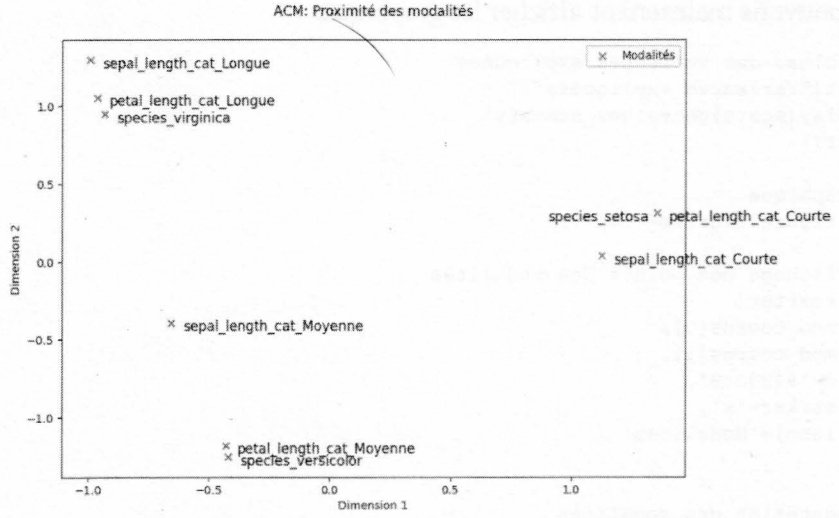
Nous pouvons maintenant afficher les graphiques :

```
# Tableau des variances expliquées
print("Variances expliquées")
display(mca.eigenvalues_summary)
print()

# Graphique
plt.figure(figsize=(10, 7))

# Affichage des points des modalités
plt.scatter(
    mod_coords[0],
    mod_coords[1],
    c='#439cc8',
    marker='x',
    label='Modalités'
)

# Annotation des modalités
```

```
for i, (x, y) in enumerate(zip(mod_coords[0], mod_coords[1])):
    # Légère mise en forme pour que les noms soient bien lisibles
    if mod_coords.index[i] == "species_setosa":
        plt.text(
            x - 0.45,
            y - 0.05,
            f'{mod_coords.index[i]}',
            fontsize=12
        )
    else:
        plt.text(
            x + 0.05,
            y - 0.05,
            f'{mod_coords.index[i]}',
            fontsize=12
        )

plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.title('ACM: Proximité des modalités\n')
plt.legend()
plt.show()
```

Variances expliquées

component	eigenvalue	% of variance	% of variance (cumulative)
0	0.918	45.89%	45.89%
1	0.676	33.80%	79.69%



Quelques explications pour bien comprendre l'ACM réalisée ici :

- Les variances cumulées atteignent 79.69 % de variance cumulée, ce qui signifie que les distances entre les modalités sont bien restituées.
- Par le jeu des proximités, nous constatons que :
  - L'espèce Virginica est caractérisée par des sépales et pétales longs.
  - À l'inverse, les pétales et sépales de l'espèce Setosa sont courts.
  - Au milieu, nous retrouvons l'espèce Versicolor qui présente plutôt des longueurs moyennes pour ces deux caractéristiques.

Ainsi, après s'être assuré que le niveau de variance expliquée est suffisamment élevé, l'ACM offre un résumé très pertinent des variables catégorielles et de leurs proximités.

6.3 Analyse en composantes principales (ACP)

6.3.1 Un des piliers de la data science

Nous terminerons ce chapitre dédié à l'analyse par un algorithme fondamental dans l'analyse de données : l'Analyse en Composantes Principales (ACP) qui apporte énormément d'informations sur les données.

Nous nous limiterons dans cette partie à mettre en lumière sa contribution dans l'analyse avant d'y revenir dans les chapitres Le Machine Learning avec Scikit Learn et L'apprentissage non supervisé, pour expliquer plus en détail comment il est construit et le rôle qu'il peut jouer également dans les modélisations.

Voici une liste des apports de l'ACP à l'étude des données :

- indiquer si le jeu de données peut se résumer facilement ;
- montrer les variables les plus importantes ;
- illustrer les interactions entre les variables ;
- identifier les outliers ;
- repérer les observations qui jouent un rôle important ;
- favoriser la compréhension des informations cachées.

6.3.2 Utilisation sur un cas pratique

Rien ne vaut un exemple concret pour apprécier l'intérêt de cet algorithme et la façon de l'utiliser. Nous allons travailler sur un petit jeu de données comprenant 19 recettes de pâtisseries et desserts. Pour chaque recette, les quantités nécessaires en grammes pour produire 8 portions sont indiquées pour le lait, le sucre, les œufs, la farine, le chocolat et les corps gras (confondant beurre, huile et crème fraîche).

Les données, stockées sur GitHub, se présentent de la façon suivante :

```
import pandas as pd

df = pd.read_csv("https://github.com/eric2mangel/BDD/raw/main/
Patisseries_et_desserts.csv",sep=";", encoding="ISO-8859-1",
index_col=0)

df.head()
```

Recette	Corps_gras_g	Lait_g	Sucre_g	Oeufs_g	Farine_g	Chocolat_g
Gâteau au chocolat	120	0	150.0	150	80	200
Crêpes froment	128	1000	40.0	200	400	0
Cookies	53	0	93.0	50	107	53
Madeleines	320	0	400.0	600	480	0
Gaufres	64	640	64.0	150	400	0

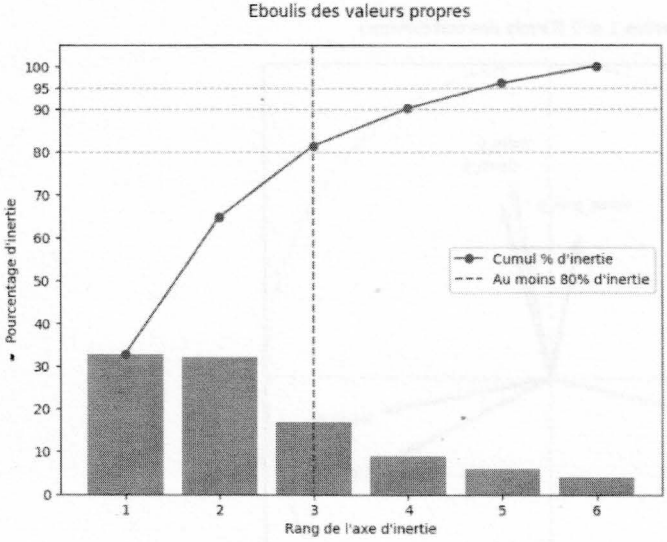
Nous sommes ici dans une approche pratique pour apprendre à interpréter une ACP. Le code Python nécessaire à sa construction sera expliqué au chapitre L'apprentissage non supervisé.

Trois graphiques sont ainsi nécessaires à l'interprétation de l'ACP. Ils se présentent dans cet ordre et se nomment respectivement : éboulis des valeurs propres, cercle des corrélations et le graphique des individus.

6.3.3 L'éboulis des valeurs propres

La mise en œuvre de l'ACP va entraîner la construction de nouvelles variables composées de toutes les anciennes dans différentes proportions. Elles se nomment composantes pour bien les dissocier des anciennes variables et sont rangées par ordre décroissant d'importance, basé sur la part de variance expliquée.

Dans le cas de nos desserts, l'éboulis de valeurs propres est le suivant :



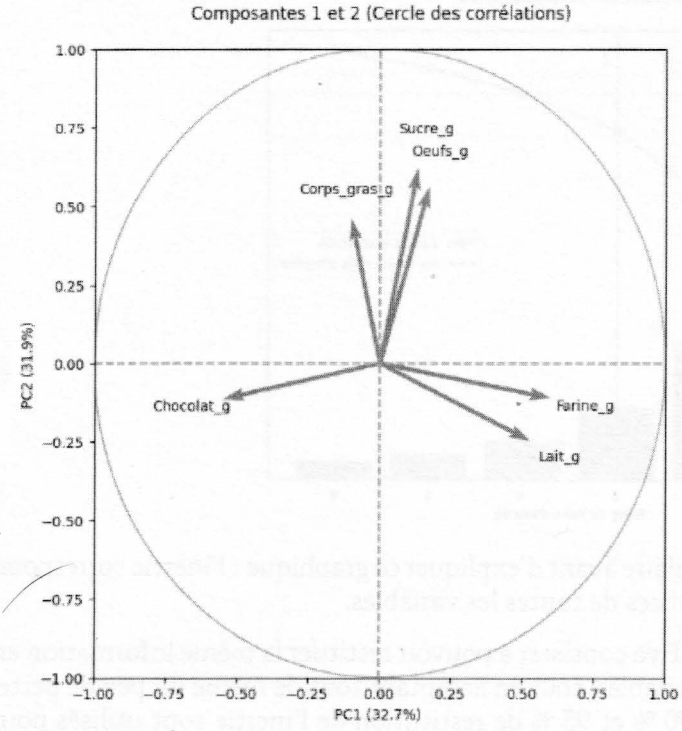
Petit point de vocabulaire avant d'expliquer ce graphique : l'inertie correspond à la somme des variances de toutes les variables.

L'enjeu dans une ACP va consister à pouvoir restituer la même information en utilisant moins de variables, tout en acceptant tout de même un peu de perte. Les seuils de 80 %, 90 % et 95 % de restitution de l'inertie sont utilisés pour identifier, en théorie, le nombre de composantes à prendre en compte pour travailler. Dans notre cas, nous pourrions restituer un peu plus de 80 % de l'information globale avec seulement trois composantes. Voilà, pour la théorie. En pratique, nous allons nous satisfaire de pouvoir représenter les données en deux dimensions en gardant tout de même autour de 64 % de l'inertie globale.

Nous allons nous appuyer sur un second graphique pour déterminer la signification des nouvelles composantes : le cercle des corrélations.

6.3.4 Le cercle des corrélations

Le cercle des corrélations représente la manière dont les anciennes variables s'illustrent dans le nouveau système de composantes. La représentation en deux dimensions de couples successifs de composantes permet ainsi de comprendre ce nouveau système.



La représentation des couples de variables se nomme plan factoriel. L'analyse débute toujours par le premier plan qui contient les deux premières composantes. Lorsque la quantité d'inertie expliquée est trop faible, il est nécessaire de représenter les autres plans factoriels pour s'assurer de comprendre pleinement le nouveau système. Ici il est indiqué dans le bas du graphique que la composante 1 restitue 32.7 % de l'inertie globale et à gauche que la deuxième composante nous apporte 31.9 % supplémentaires.

Commentons maintenant le graphique en commençant par l'axe des abscisses puis les ordonnées :

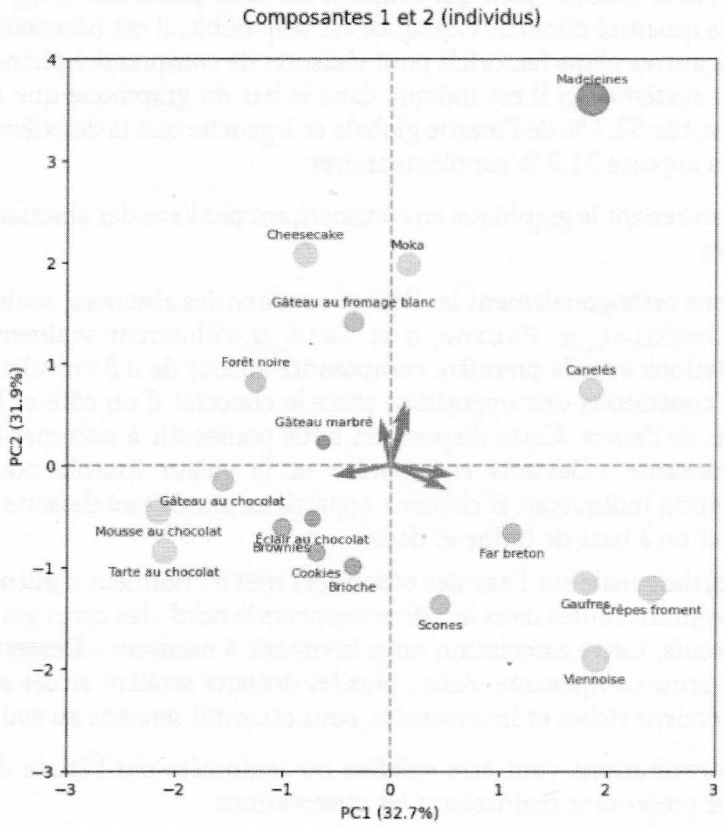
- Si nous projetons orthogonalement les flèches sur l'axe des abscisses, seules les variables `Chocolat_g`, `Farine_g` et `Lait_g` s'illustrent vraiment avec des corrélations avec la première composante autour de 0.5 en valeur absolue. Nous constatons une opposition entre le chocolat d'un côté et la farine et le lait, de l'autre. Cette disposition nous pousserait à nommer la première composante « Desserts chocolatés » et la valeur fournie pour chaque observation indiquerait si chacune appartient plutôt aux desserts à base de chocolat ou à base de farine et de lait.
- La projection orthogonale sur l'axe des ordonnées met à l'honneur d'autres variables qui pointent toutes dans le même sens vers le nord : les corps gras, le sucre et les œufs. Cette association nous inviterait à nommer « Desserts riches » la deuxième composante. Ainsi, plus les desserts seraient situés au nord, plus ils seraient riches et inversement pour ceux qui seraient au sud.

Ces premières constatations vont être validées ou renforcées par l'étude du dernier graphique présentant maintenant les observations.

6.3.5 Le graphique des individus

Le graphique des individus représente les observations dans le même référentiel que le cercle des corrélations. Il va nous permettre de mieux comprendre le graphique précédent en indiquant le positionnement de chaque observation (chaque dessert dans notre cas) dans le système des composantes.

Regardons ensemble comment se présente ce graphique :



Ainsi s'assemblent les différentes pièces du puzzle. Les desserts sont disposés en fonction du nouveau système des composantes. Conformément au cercle des corrélations, nous retrouvons à l'ouest les desserts chocolatés, à l'est ceux qui sont plutôt composés de farine et de lait. Leur ordonnée va aussi nous renseigner sur leur degré de richesse en allant des desserts les moins riches au sud vers les plus chargés au nord. Les viennoises et les madeleines jouent ici respectivement les observations extrêmes pour les produits riches, la taille du cercle de chacun indiquant le poids qu'elles représentent dans notre système. Cet éloignement du centre va ainsi désigner les chefs de groupes qui incarnent le plus les nouvelles dimensions.

L'apport de l'ACP est très important pour la compréhension d'un jeu de données. Il clôture le feature engineering en nous offrant le meilleur résumé possible de notre système de variables tout en révélant les articulations cachées que nous ne pourrions pas découvrir autrement. À l'issue de cette étape, nous sommes censés avoir acquis les informations essentielles contenues dans les données et ainsi être en mesure de sélectionner ou de créer les variables qui feront le succès de nos modélisations.

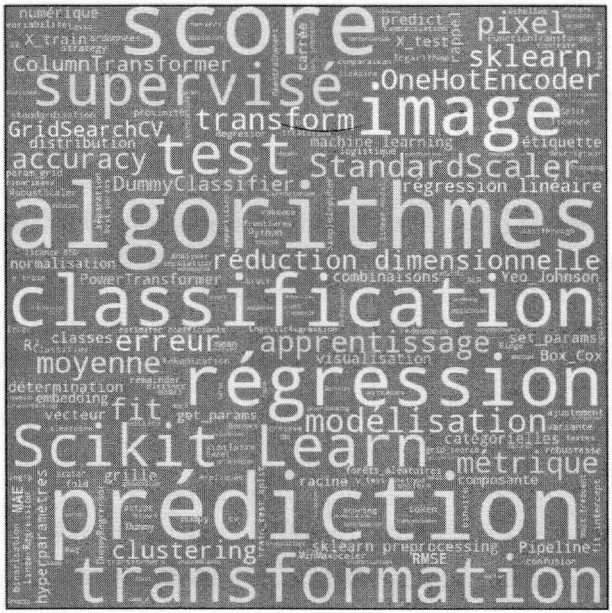
Remarque

L'ACP est un algorithme linéaire, il peut s'avérer inutile dans certains cas de non-linéarité mais il existe des alternatives que nous aborderons au chapitre L'apprentissage non supervisé.

Du chapitre Le Machine Learning avec Scikit-Learn au chapitre Modéliser le texte et l'image, nous allons découvrir ce que sont les modélisations, les différentes formes qu'elles peuvent prendre et comment en tirer le meilleur parti.

Chapitre 6
Le Machine Learning avec Scikit-Learn

1. Introduction au Machine Learning : concepts et types de modèles



Nuage de mots des termes techniques et informatiques utilisés pour ce chapitre

Le Machine Learning vise à développer des systèmes informatiques capables d'apprendre à partir de données pour comprendre et prendre des décisions. Pour ce faire, nous allons utiliser des algorithmes et des modèles statistiques de différentes natures.

Il existe toute une panoplie d'outils permettant de répondre à tout type de problème.

Dans le cadre de cet ouvrage, nous restreindrons le Machine Learning à trois grands domaines :

- l'apprentissage non supervisé ;
- l'apprentissage supervisé ;
- le traitement du texte et des images.

Il est important de signaler que le traitement du langage naturel et des images ne sera abordé, dans le cadre de cet ouvrage, qu'en superficie, pour rester accessible à tous. En revanche, d'autres domaines, comme l'apprentissage par renforcement, qui ont un champ applicatif plus confidentiel, seront volontairement mis de côté.

1.1 L'apprentissage non supervisé

1.1.1 Définition

L'apprentissage non supervisé va nous permettre de comprendre le fonctionnement caché de nos données. Il est appelé non supervisé car les informations qui vont émerger n'ont pas d'étiquette. Nous sommes donc dans une situation de découverte, ne sachant pas à l'avance ce que nous allons trouver. Ce type d'apprentissage va nous permettre de répondre aux questions suivantes :

- Quelles sont les caractéristiques dans mes données qui sont les plus importantes ?
- Lesquelles sont de moindre intérêt ?
- Existe-t-il des liens ou des oppositions entre certaines caractéristiques ?
- Comment pourrions-nous mieux comprendre les composantes de notre champ d'études ?

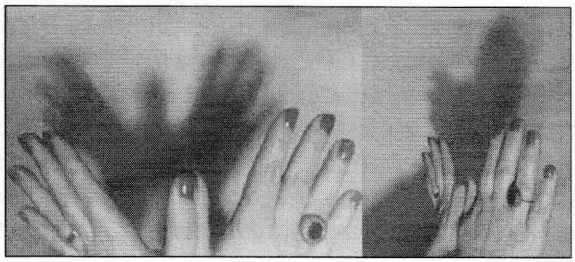
– Comment regrouper certaines observations pour résumer les grandes tendances ou raisonner en termes de groupes ?

Concrètement, si nous nous représentons nos données comme un tableau Excel, les deux grandes opérations vont consister à agir sur les colonnes ou sur les lignes. Nous pourrions, pour illustrer, imaginer un tableau contenant les clients d'une entreprise où chaque ligne représenterait un client et différentes caractéristiques de ces clients en colonnes comme leurs revenus, le montant total dépensé ou leur âge (une caractéristique par colonne). Nos outils pourraient nous permettre de découvrir s'il existe un lien entre l'âge et le montant dépensé par exemple, ou de regrouper ensemble les clients par proximité des profils et pouvoir ainsi former des groupes de clients : jeunes et dépensiers, personnes âgées économes, etc.

Ces deux grands types d'actions portent un nom : la réduction dimensionnelle et le clustering.

1.1.2 La réduction dimensionnelle

Avant d'aborder la description de la réduction dimensionnelle appliquée aux données, prenons un exemple simple pour illustrer son fonctionnement : les ombres chinoises. Elles consistent à produire sur un mur une ombre représentant une chose ou un animal en positionnant ses mains de manière adéquate. Outre le positionnement des mains, un autre paramètre est primordial : l'orientation de la lumière créant l'ombre. Sans une inclinaison optimale, l'ombre recherchée n'aura pas de sens. Dans cette analogie, nos mains seraient nos données, la réduction dimensionnelle serait la lumière idéalement inclinée et l'ombre, la représentation symbolique résultante.



Passons maintenant à la réduction dimensionnelle appliquée aux données. Cette technique consiste à diminuer le nombre des caractéristiques pour simplifier la représentation des données tout en préservant au mieux les informations importantes. Nous allons donc produire un bon résumé appliqué à une base de données. Il est important de souligner, pour bien comprendre ici, que dimension renvoie à variable.

En pratique, nous allons créer de nouvelles variables, appelées composantes, qui sont en fait une recombinaison des anciennes, établissant une nouvelle grille de lecture. Maîtriser cette technique est fondamental pour quiconque souhaite bien comprendre les subtilités des données, révéler les informations sous-jacentes et percevoir les relations entre les différentes variables de manière claire et accessible. Contrairement à certains algorithmes qui nous renvoient directement un résultat « prêt à interpréter », la réduction dimensionnelle nécessite d'analyser les résultats et laisse une large place à la capacité de l'interprétation humaine. En récompense de cet effort, nous récoltons de nombreux avantages que voici :

En premier lieu, en concentrant l'information importante, la réduction dimensionnelle va nous permettre de visualiser nos données sous forme d'un bon résumé jusqu'à trois dimensions. Pour reprendre l'exemple de notre base de données de clients, nous serions en mesure de géolocaliser chaque client, mais aussi chacune de leurs caractéristiques nous permettant ainsi par le jeu des proximités-éloignements de voir s'il existe des groupes, s'ils ont un profil plus ou moins homogène, si certains se détachent nettement ou non. C'est dans ce genre de cas que nous mesurons l'importance cruciale de la data visualisation permettant une compréhension immédiate de l'information contenue dans notre base de données.

En deuxième lieu, un autre avantage de la réduction dimensionnelle est de permettre de réduire le bruit et la redondance. Cet avantage a un bénéfice qui est double. Primo, il permet de mieux comprendre nos données sans être pollué par des détails sans importance. Cela est fondamental car la connaissance la plus profonde de l'information contenue jouera pleinement sur le reste des opérations. Secundo, cet allègement des données, à considérer comme une optimisation de l'allocation des ressources, permet également d'alléger le poids des données.

L'avantage ici se situe plus sur un versant technique : si le fichier contenant les données est plus léger, le programme tourne plus vite, ce qui contribue à réduire les temps de calcul et d'utilisation des ressources informatiques.

Pour finir, l'étude des composantes présentées par ordre décroissant d'importance va nous permettre de mesurer l'importance de chaque ancienne variable, en discernant leur contribution relative à la composition globale. Cela permet de hiérarchiser nos variables et in fine de mieux les comprendre.

Nous allons maintenant nous intéresser à la deuxième partie des algorithmes non supervisés qui va nous aider à raisonner en termes de groupes.

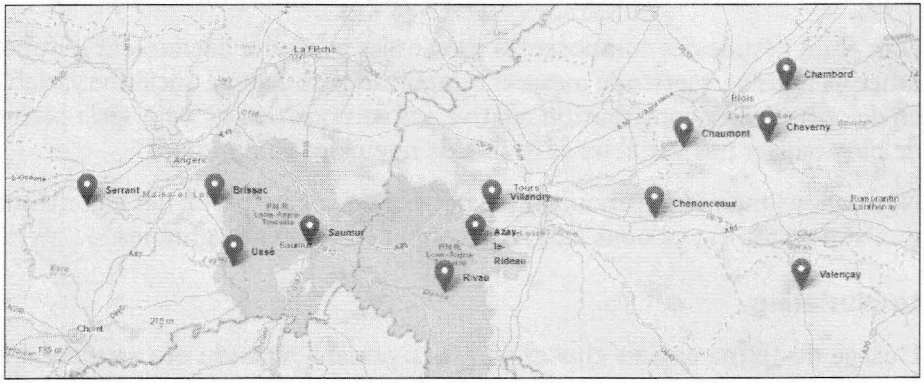
1.1.3 Le clustering

L'usage du terme anglais clustering est largement répandu et correspond en français à la notion de partitionnement. Il est important ici de souligner qu'il ne doit pas être confondu avec le terme de classification que nous allons aborder ensuite, car il existe une différence importante entre les deux notions :

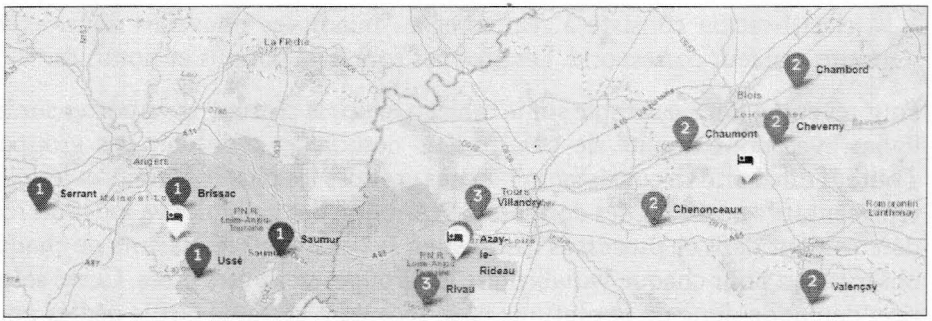
- Le clustering consiste à créer des groupes par proximité ou vraisemblance sans savoir à l'avance comment les données vont être assemblées. Il conviendra ensuite, comme pour la réduction dimensionnelle, d'analyser les résultats pour identifier les groupes ainsi créés et leur donner un nom.
- La classification consiste à réaffecter au mieux les nouvelles données au groupe qui leur correspond. Les groupes sont déjà connus et nommés.

Pour revenir à notre exemple sur le tableau Excel, le clustering va porter sur les lignes avec pour finalité de réduire leur nombre et d'obtenir des groupes. Toute la difficulté va consister à créer des groupes les plus cohérents et les plus représentatifs possibles des données. La sélection des variables va jouer un rôle déterminant dans ce processus car c'est en se basant sur les valeurs de chaque observation pour chaque variable que les groupes vont être créés. Cette sélection demande donc de la pratique et le recours à certains outils dédiés pour réussir. Une sélection fine des variables devra être menée et des tests indiquant des options pertinentes de clustering seront nécessaires ici. Nous aborderons ensemble toutes ces actions dans le chapitre L'apprentissage non supervisé.

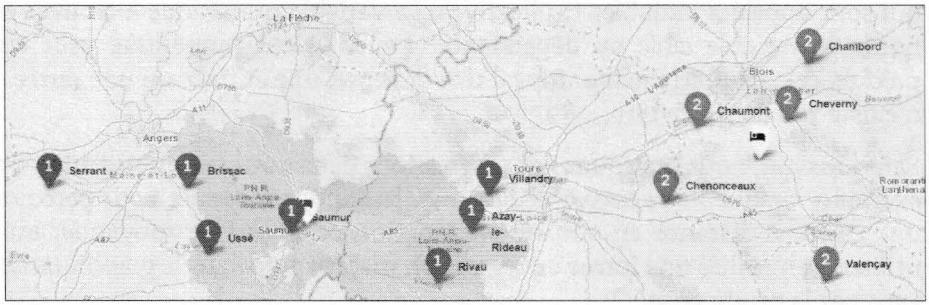
Pour illustrer l'intérêt d'un clustering, supposons que nous souhaitions visiter les 12 châteaux de la Loire suivants :



Nous planifions un séjour de trois jours pour les visiter, et grâce à leurs coordonnées géographiques, nous allons pouvoir facilement déterminer à l'aide d'un clustering à 3 groupes (ici, nous utilisons l'algorithme du K-means que nous pratiquerons dans le chapitre L'apprentissage non supervisé) le lieu géographique idéal pour trouver un hôtel (les centres des groupes appelés centroïdes sont représentés par les balises représentant un lit) :



Changement de programme, nous ne disposons plus que de deux jours à cause d'un rendez-vous imprévu : pas de problème, nous n'avons plus qu'à demander de faire deux groupes :



Nous allons employer la même méthodologie présentée ici, quelles que soient nos données. Gardons notre exemple de bases de données clients, il faudra juste se représenter nos clients comme les villes et les valeurs des caractéristiques (âge, montant des dépenses, salaire, etc.) comme des coordonnées géographiques.

Après avoir défini les possibilités et le champ d'application du non supervisé, intéressons-nous à son homologue supervisé qui va nous ouvrir un autre champ des possibles.

1.2 L'apprentissage supervisé

1.2.1 Introduction

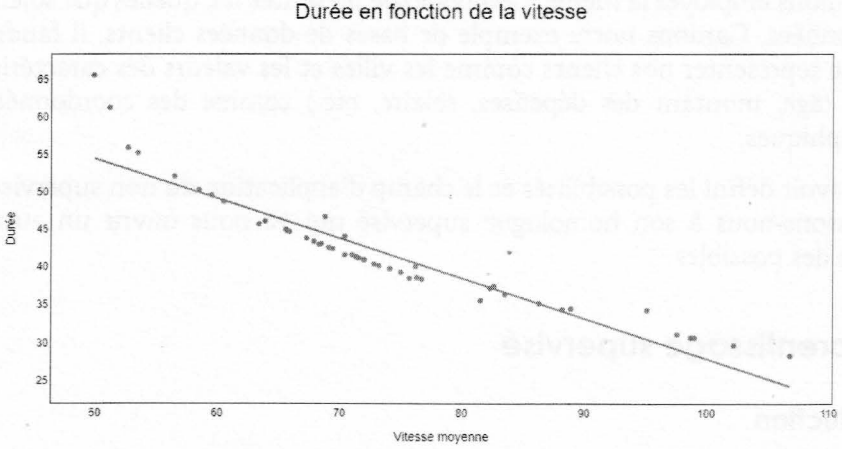
Le périmètre d'intervention de l'apprentissage supervisé peut se définir par deux verbes : prédire et classer.

Contrairement à l'apprentissage non supervisé, où les données ne sont pas étiquetées et où le modèle doit découvrir des structures ou des schémas par lui-même, l'apprentissage supervisé repose sur des données annotées, où chaque exemple est associé à une étiquette prédéfinie. En étudiant les étiquettes, il va être possible d'entraîner des algorithmes à trouver des moyens de prédire une valeur ou classer des observations. Ces deux grandes techniques sont nommées respectivement régression et classification.

1.2.2 Régression

La régression consiste à mettre en place un algorithme permettant de prévoir une valeur numérique (discrète ou continue) en fonction d'autres variables qui sont appelées variables explicatives. La variable numérique à trouver est nommée variable cible ou dépendante. Toute valeur numérique peut être prédite, cela peut être le montant d'une dépense, une surface ou une durée. Il n'existe pas de restriction.

Sans le savoir, nous pratiquons des régressions dans notre vie quotidienne. Par exemple, lorsque nous essayons d'estimer le temps nécessaire pour parcourir une certaine distance en voiture en fonction de la vitesse moyenne, nous utilisons en réalité une forme de régression matérialisée dans le graphique suivant par la droite grise.



Toutefois, à partir de maintenant, nous allons apprendre à aller plus loin en gérant des systèmes plus complexes avec plusieurs variables explicatives.

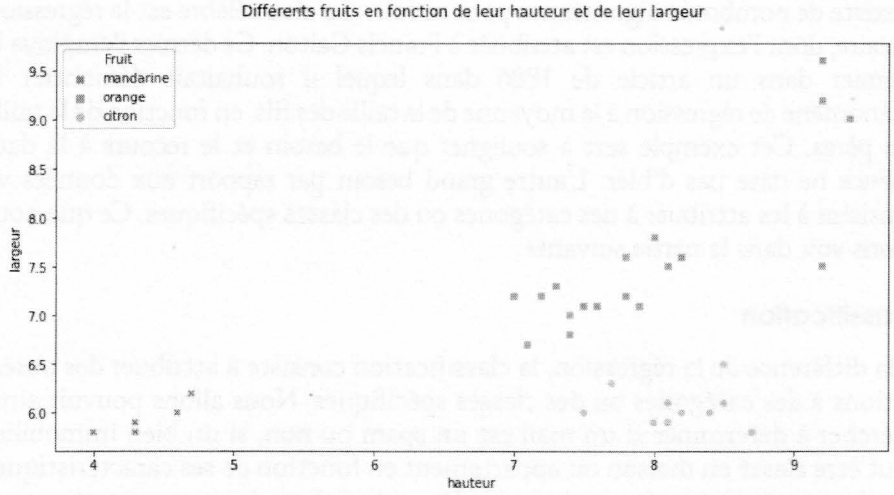
Une condition importante concernant ces multiples variables explicatives est qu'elles soient indépendantes entre elles. Cela permet d'éviter de produire un phénomène d'emballement. En revanche, il est important que la dépendance de chacune soit établie avec la variable cible pour éviter un effet de hasard heureux.

Il existe de nombreux algorithmes pour ce faire. Le plus célèbre est la régression linéaire, dont l'expression est attribuée à Francis Galton. Ce dernier l'employa le premier dans un article de 1886 dans lequel il souhaitait démontrer le phénomène de régression à la moyenne de la taille des fils, en fonction de la taille des pères. Cet exemple sert à souligner que le besoin et le recours à la data science ne date pas d'hier. L'autre grand besoin par rapport aux données va consister à les attribuer à des catégories ou des classes spécifiques. Ce que nous allons voir dans la partie suivante.

1.2.3 Classification

À la différence de la régression, la classification consiste à attribuer des observations à des catégories ou des classes spécifiques. Nous allons pouvoir ainsi chercher à déterminer si un mail est un spam ou non, si un bien immobilier peut être classé en maison ou appartement en fonction de ses caractéristiques ou, dans un registre plus technique, déterminer à quel auteur appartient un texte ou à quel peintre attribuer telle peinture. Tout peut être classifié. Là encore, nous mettons en pratique des classifications sans même nous en rendre compte dans notre vie quotidienne.

Prenons l'exemple de fruits, en nous concentrant uniquement sur deux caractéristiques : la hauteur et la largeur, dans le but de les représenter en deux dimensions et de faciliter la compréhension. Il faut noter cependant qu'il y en a normalement beaucoup plus. Observons maintenant comment les différents membres de chaque famille de fruits se regroupent dans des zones similaires du graphique. À partir de cette observation, nous pourrions envisager de les classifier en fonction de leurs dimensions respectives. Par exemple, un fruit mesurant moins de 5 cm de hauteur et moins de 6,5 cm de largeur aurait de fortes chances d'être une mandarine.



Nous retrouvons donc, comme pour la régression, le recours à des variables explicatives. Cependant, dans le cadre de la classification, nous avons la possibilité de travailler avec une ou plusieurs variables cibles (ici nous avons trois fruits donc trois variables cibles).

Une autre différence importante avec la régression réside dans le fait qu'au lieu de chercher à prédire une valeur, dans le cadre de la classification, nous recherchons des frontières de décision qui nous permettent de classer les observations dans des catégories spécifiques. Cette notion de détermination probabiliste des frontières de décision nous permet également d'obtenir la probabilité d'appartenance à chaque catégorie, ce qui peut s'avérer utile dans des situations telles qu'un diagnostic médical par exemple en offrant au patient la possibilité de prendre une décision plus éclairée en fonction du degré de certitude.

Nous avons à ce stade défini les deux grandes techniques pour comprendre, prédire ou classer les données. Mais les données ne se limitent pas à des tableaux : tout est donnée. Tant qu'une information est stockée sur un support numérique, que ce soit un tableau, un texte, une image, un son ou une vidéo, nous pouvons le transformer et l'exploiter pour le prédire, l'expliquer ou le classer. Ainsi, il convient d'envisager un chapitre sur le texte et l'image pour illustrer cette possibilité.

1.3 Le texte et l'image

1.3.1 Définitions des concepts

L'étude du langage (*Natural Language Processing* ou NLP) et des images forme deux domaines vastes et distincts. Il est cependant crucial de souligner que ce sont les étapes nécessaires à la transformation de ces données en vecteurs (ou embeddings) qui constituent le point de départ essentiel. Une fois que les données textuelles ou visuelles ont été converties en représentations numériques, des opérations classiques de classification ou de régression peuvent être appliquées avec efficacité. Nous traiterons volontairement ce domaine de manière superficielle pour permettre au lecteur d'en distinguer les contours et permettre les premières expérimentations. Mais nous n'irons pas plus loin car chaque thème pourrait faire l'objet d'un ouvrage entier.

1.3.2 Le texte et le NLP

Un texte brut n'est pas exploitable en l'état. Il est nécessaire de le préparer afin de le rendre utilisable pour les algorithmes. Cette préparation peut être effectuée de différentes manières. Nous pouvons la réaliser manuellement pour les premières en respectant certaines règles de bonne conduite ou avoir recours à des librairies spécialisées de Python. Nous allons nous attacher à définir ces grandes étapes de transformation afin de bien comprendre les manipulations effectuées lorsque nous utiliserons des modules spécialisés.

Nettoyage et normalisation : la première étape consiste à nettoyer le texte en supprimant les éléments indésirables tels que la ponctuation, les caractères spéciaux ou les balises HTML, le cas échéant. Ensuite, nous normalisons les caractères en les mettant en minuscules et en standardisant les nombres et les dates selon les besoins.

Tokenization : la deuxième étape, appelée tokenization, consiste à transformer le texte en une série d'unités significatives appelées tokens. Ces tokens peuvent être des mots individuels, des groupes de mots ou des phrases entières. Lors de ce processus, le texte est découpé et chaque élément est considéré comme un token distinct. Les listes résultantes de la tokenization sont couramment rencontrées sous le nom de bag of words.

Suppression des mots inutiles : dans la troisième phase, nous éliminons les mots inutiles tels que les mots courants ou peu informatifs, soit en utilisant des listes prédéfinies de stopwords, soit en créant notre propre liste en fonction du contexte.

Pour les deux dernières phases, le recours à des modules est quasiment obligatoire.

Stemmatisation et lemmatisation : la quatrième partie consiste à réduire les mots à leur racine ou leur forme canonique pour éviter la multiplication des formes conjuguées des verbes, par exemple. Cette étape est réalisée soit par stemming, qui consiste à trancher pour ne garder que la racine du mot, soit par lemmatisation, qui vise à retourner à l'origine du mot pour obtenir des termes plus compréhensibles et ouvrir la voie à la dernière étape.

Vectorisation : enfin, une fois que le texte est prétraité, nous pouvons le transformer en vecteurs numériques à l'aide d'algorithmes spécifiques. Nous obtenons alors des bags of words (littéralement des sacs de mots), contenant tous les termes et leur fréquence, pondérée ou non, de manière non ordonnée. L'application du `CountVectorizer` et du `TF-IDF` sera justement abordée en pratique dans le chapitre Modéliser le texte et l'image.

1.3.3 Le traitement des images

Comment l'image numérique est-elle définie ?

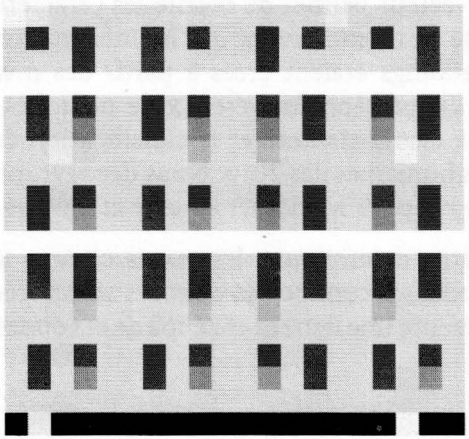
L'image numérique est constituée de pixels ayant chacun une teinte.

Commençons par le cas le plus simple : l'image en noir et blanc. Elle est définie par x pixels en hauteur et y pixels en largeur. Chaque pixel a une valeur comprise entre 0 et 255 où 0 correspond au noir absolu, 255 au blanc absolu et tout le dégradé des gris entre ces deux bornes.

Voici l'exemple d'une façade d'immeuble en noir et blanc à gauche et les valeurs respectives de chaque pixel à droite :

Façade d'immeuble pixellisée

Image en Niveaux de Gris



Valeurs des niveaux de gris

250	236	231	229	231	231	229	231	229	231	231	229	231	229	241	231	229	231	229	231
231	42	231	42	231	231	42	231	42	231	231	42	231	43	231	231	43	231	42	231
231	231	231	42	231	231	45	231	231	231	231	42	231	231	231	231	231	231	44	231
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
231	42	231	42	231	232	42	230	42	231	231	42	231	43	230	231	42	234	42	231
231	42	231	134	231	231	42	230	148	231	231	136	231	43	231	231	145	234	42	231
231	231	245	224	231	231	231	231	197	231	231	197	231	231	231	231	228	247	231	231
231	231	231	231	231	231	231	231	231	231	231	231	231	231	231	231	231	231	231	231
231	42	227	41	231	230	42	231	42	229	227	42	229	42	231	231	43	231	42	231
232	39	231	153	231	231	45	231	153	231	231	153	229	44	230	231	153	231	44	231
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
231	42	231	42	231	231	42	230	42	231	231	42	231	43	231	231	42	234	42	231
231	46	231	38	231	231	42	230	38	231	231	38	231	43	230	231	36	234	42	231
231	231	231	197	231	231	231	231	197	231	231	197	231	231	231	231	193	231	231	231
231	231	231	231	231	231	231	231	231	231	231	231	231	231	231	231	231	231	231	231
231	42	227	41	231	231	42	231	42	231	231	42	229	42	231	231	42	234	42	231
232	39	235	155	231	231	45	231	155	231	231	158	233	44	230	231	44	229	158	231
230	230	230	230	230	230	230	230	230	230	230	230	230	230	230	230	230	230	230	230
1	241	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	250	1	1

L'image en couleur fonctionne également avec des pixels mais ceux-ci ont désormais trois valeurs entre 0 et 255, une pour chaque canal : rouge, vert et bleu. En combinant ces trois valeurs, un peu plus de 16 700 000 de teintes différentes peuvent être produites.

Ainsi stockée sous formes de matrices de pixels, l'image ne va pas pouvoir être utilisée directement sous cette forme en data science et ce, pour deux raisons :

- Les images d'un dossier ont rarement la même taille et la même définition, rendant caduque toute comparaison pixel à pixel entre deux images.
- La moindre rotation ou le moindre zoom décale tous les pixels.

L'extraction de caractéristiques des images

La solution pour contourner ce problème consiste à repérer des points d'intérêt dans l'image, à les regrouper par familles et à les quantifier pour aboutir à un vecteur descripteur propre à chaque image. Nous le rencontrons couramment sous le nom d'embedding. Ce vecteur permet de décrire les caractéristiques de l'image de manière compacte en ne conservant que les informations importantes. Jusqu'à 2012, les embeddings étaient créés à partir des invariants. Mais, à partir de cette date, l'apparition des réseaux de neurones à convolution (CNN) a révolutionné la reconnaissance et la classification des images, dépassant même les capacités humaines dès 2015. Nous découvrirons ensemble ces deux techniques dans le chapitre Modéliser le texte et l'image.

Ainsi, une fois transformés en vecteurs descripteurs, les textes comme les images peuvent être manipulés et modélisés comme des données structurées classiques où chaque embedding représente une ligne et chacune de ses dimensions, une colonne.

2. Présentation de Scikit-Learn, la bibliothèque Python pour la data science

2.1 Une offre simple et complète de fonctionnalités

La librairie Scikit-Learn, également appelée Sklearn, est une bibliothèque de Machine Learning créée par des membres de l'INRIA en 2010. Elle s'est progressivement imposée pour devenir une référence en la matière.

Scikit-Learn ne se contente pas de proposer une large gamme de modèles de régression et de classification, elle offre en plus toutes les fonctionnalités pour préparer les données, optimiser les algorithmes et mesurer les résultats. Elle permet ainsi de gérer seule toute la chaîne des opérations d'une modélisation classique.

Sur la forme, Scikit-Learn a été pensée pour être simple et satisfaire aussi bien les débutants que les praticiens les plus expérimentés. Certains points bien pensés facilitent notamment la prise en main :

- La bibliothèque possède une documentation fournie mettant toujours un exemple d'application pour chaque algorithme.
- Hormis quatre exceptions (LinearRegression, LogisticRegression, SVR et SVC), les algorithmes de régression et de classification ont, respectivement, tous un nom qui se termine par `Regressor` ou `Classifier` pour indiquer à quel type d'apprentissage ils sont destinés.
- Les algorithmes sont assez simples à utiliser et présentent des méthodes communes à un large nombre de fonctions permettant de facilement de passer de l'un à l'autre en retrouvant des termes communs.

C'est justement l'occasion de découvrir ensemble ces méthodes communes qui facilitent grandement la prise en main de Scikit-Learn. Mais juste avant, il est temps de l'installer si ce n'est pas encore fait :

```
pip install -U scikit-learn
```

Remarque

L'option `-U` nous assurera en plus de mettre la librairie à jour, si elle est déjà présente, vers la dernière version.

2.2 Des méthodes communes aux différentes fonctions

Lors de la mise en œuvre de Scikit-Learn, nous utilisons un socle commun de méthodes que nous allons retrouver dans la plupart des algorithmes. La finalité de cette bibliothèque consiste à appliquer diverses transformations sur les données, de jouer sur les paramètres ainsi que d'évaluer ces transformations. Pour ce faire, les méthodes suivantes ont été largement intégrées à toutes les fonctions.

2.2.1 La méthode fit()

La méthode `fit()` est fondamentale dans Scikit-Learn. Cette méthode permet d'entraîner un algorithme sur un jeu de données spécifique en ajustant les paramètres du modèle par rapport aux données. Une fois entraîné, ces paramètres sont conservés en mémoire et l'algorithme pourra être déployé sur d'autres jeux de données.

La méthode `fit()` est utilisée à la fois pour entraîner un algorithme d'apprentissage mais également pour appliquer diverses transformations sur les données.

Notons qu'il s'agit juste d'un entraînement et qu'il va être nécessaire de recourir aux méthodes `transform()` ou `predict()` pour l'appliquer.

Voici, pour illustrer, comment entraîner l'algorithme `StandardScaler()` qui transforme les données numériques dans le but d'avoir une moyenne de 0 et un écart-type de 1 :

```
# Import de la fonction StandardScaler
from sklearn.preprocessing import StandardScaler

# Nos données numériques (ne pas appliquer sur des données catégorielles)
data = [[1, 2], [3, 4], [5, 6], [7, 8]]

# Instanciation de l'objet StandardScaler
my_standardscaler = StandardScaler()

# Entraînement sur le jeu de données data
my_standardscaler.fit(data)
```

L'objet `my_standardscaler` est désormais prêt à l'emploi et peut être directement appliqué à *data* ou à n'importe quel autre dataset tant que la structure des données est la même. Les paramètres d'entraînement de `data` seront ainsi appliqués à d'autres données.

Dans le cadre de l'entraînement d'un algorithme supervisé destiné à prédire une valeur ou une classe, il sera nécessaire d'ajouter un deuxième paramètre dans la parenthèse de `fit()`.

Voici un exemple d'entraînement d'une régression linéaire destinée à prédire le prix de vente d'un appartement à partir de sa surface :

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Prédiction du prix d'un logement en fonction de sa surface
X = np.array([55, 60, 72, 80, 90]).reshape(-1, 1) # Surface
y = np.array([158, 180, 207, 220, 250])  # Prix en milliers d'euros

# Création et ajustement du modèle de régression linéaire
model = LinearRegression()

# Entraînement
model.fit(X, y)
```

Remarque

`reshape(-1,1)` est utilisé pour que X soit bien considéré comme un vecteur en colonnes. Cette étape est nécessaire pour que les données en entrée soient correctement traitées.

2.2.2 Les méthodes transform et fit_transform

À présent que l'algorithme a été entraîné, il peut être déployé sur les données d'entraînement ou sur n'importe quel autre dataset tant que celui-ci présente les mêmes colonnes et dans le même ordre que le jeu d'entraînement. La méthode `transform()` va ainsi appliquer les transformations. Pour les cas où nous souhaitons directement entraîner et transformer, nous pouvons recourir directement à la méthode `fit_transform()`.

Mettons en pratique grâce à `transform()` l'objet `my_standardscaler` entraîné précédemment :

```
# Transformation des données
data_scaled = my_standardscaler.transform(data)

print(data_scaled)

# Output:
#[[-1.34164079 -1.34164079]
# [-0.4472136  -0.4472136 ]
# [ 0.4472136   0.4472136 ]
# [ 1.34164079  1.34164079]]
```

2.2.3 La méthode predict

La méthode `predict()` va jouer un rôle similaire à la méthode `transform()` mais dans le cadre des prédictions des algorithmes supervisés et non supervisés. Elle va nous permettre d'appliquer l'algorithme entraîné sur une ou des nouvelles observations et de prédire leur valeur.

Reprenons l'exemple de notre régression linéaire sur le prix des appartements. Si un nouveau mandat rentre pour un appartement de 98 m^2, nous pourrons estimer sa valeur :

```
# Prédiction sur le bien
model.predict(np.array([[98]]))

# Output: array([268.68823242])
```

Le résultat nous est restitué sous forme de vecteur. Le prix est estimé à environ 269 000 euros.

2.2.4 La méthode score()

La méthode `score()` est également très répandue. Elle est destinée à fournir le coefficient de détermination (R^2) qui indique la proportion de la variance expliquée par la ou les variables explicatives. Plus elle est proche de 1, plus le modèle est bon. Attention car le R^2 peut aussi être négatif indiquant que l'algorithme fait moins bien que de prédire simplement la moyenne.

Vérifions le score de notre régression linéaire :

```
# Calcul du coefficient de détermination
model.score(X,y)

# Output: 0.985746436356181
```

Cet exemple est uniquement destiné à montrer comment utiliser le score. Nous verrons un peu plus loin quelles précautions sont indispensables à prendre pour le considérer comme acceptable.

2.2.5 Les méthodes get_params et set_params

Les méthodes `get_params()` et `set_params()` permettent respectivement d'obtenir et d'initialiser les paramètres d'un algorithme. Précisons tout de même que l'initialisation des paramètres se fait souvent au moment de la création de l'instance de l'algorithme comme ici :

```
# Création et ajustement du modèle de régression linéaire
model = LinearRegression(fit_intercept=False)
```

Mais l'usage de `set_params()` permet de le modifier après coup. Il ne faudra pas oublier de réentraîner le modèle pour que les changements soient effectifs :

```
# Création et ajustement du modèle de régression linéaire
model.set_params(fit_intercept=False)

# Réentraînement nécessaire
model.fit(X,y)

# Affichage du nouveau score
model.score(X,y)

# Output: 0.9638810763453214 # Score sans inclure l'ordonnée
à l'origine
```

Sans surprise, `get_params()` renverra un état des paramètres incluant également les modifications opérées avec `set_params()` si cette méthode a été utilisée :

```
# Affichage des paramètres de l'algorithme
model.get_params()

# Output:
# {'copy_X' : True, 'fit_intercept': False, 'n_jobs': None,
'positive' : True}
```

2.3 Le soutien de la licence BSD et d'une communauté active

En plus d'offrir une bibliothèque simple, complète et bien pensée, Scikit-Learn bénéficie du soutien de la licence BSD et d'une communauté active de développeurs qui constituent une autre composante de sa popularité.

La licence BSD (*Berkeley Software Distribution Licence*) est une licence libre qui autorise une grande flexibilité au niveau de l'utilisation, de la modification et de la redistribution du code. La grande liberté offerte par ce type de licence n'est certainement pas étrangère à l'implication de la communauté d'utilisateurs et de contributeurs qui participent à renforcer la pertinence et la qualité de ce projet.

Ainsi, Scikit-Learn s'est imposée comme une référence incontournable tant pour les chercheurs et professionnels des données que pour les débutants souhaitant s'initier à la data science avec un outil simple, robuste et complet.

3. Les grandes étapes d'un projet de Machine Learning

Scikit-Learn va nous accompagner dans toutes les grandes étapes d'une modélisation que nous allons découvrir maintenant d'un point de vue théorique. Nous fournirons un exemple pratique de chaque cas, régression et classification, au chapitre L'apprentissage supervisé.

Cette découverte de la modélisation sera séparée en deux parties en commençant par la préparation des données puis l'expérimentation des algorithmes et leur évaluation.

3.1 La préparation des données

3.1.1 La séparation des variables explicatives de la variable cible

La modélisation est une expérience scientifique consistant à prédire une valeur ou l'appartenance à une classe à partir des données disponibles.

Après avoir importé les données d'un jeu fictif, que nous nommerons `df`, la première mesure va consister à séparer les variables explicatives de la variable à expliquer, dite variable cible :

```
X = df[["Variable_1","Variable_2","Variable_3"]] # Variables explicatives
y = df["Variable_cible"]                         # Variable cible
```

3.1.2 La séparation entre données d'entraînement et données de test

Afin de nous rapprocher des conditions réelles et d'être en mesure d'évaluer correctement les performances de notre expérimentation, nous allons réaliser une deuxième séparation consistant à mettre de côté une partie des données qui serviront à simuler les conditions réelles.

Ainsi, nous isolerons, dès le départ, entre 10 % et 30 % des observations selon les cas. Ce jeu sera nommé le jeu de test et il comprendra deux tables : `X_test` et `y_test`. La partie restante sera le jeu d'entraînement, comprenant lui aussi deux tables : `X_train` et `y_train`. Ce dernier servira à entraîner les algorithmes qui auront connaissance de ses paramètres. Le jeu de test, quant à lui, sera inconnu des algorithmes et n'interviendra qu'à la fin du processus pour évaluer le score en conditions réelles, qui est le seul à prendre en compte.

```
# Import de la fonction train_test_split
from sklearn.model_selection import train_test_split


# Split des données avec un jeu de test représentant 20 % de l'ensemble
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)
```

Il est primordial de retenir que le jeu d'entraînement, en tant que tel, est soumis aux fonctions `fit()` ou `fit_transform()` selon qu'il serve juste d'entraînement ou qu'il soit aussi nécessaire de transformer le jeu de données.

En revanche, seules les méthodes `transform()` ou `predict()` devront être appliquées sur le jeu de test car l'utilisation de `fit()` entraînerait une fuite de données (data leakage) qui remettrait complètement en cause la modélisation.

3.1.3 Les transformations des variables

La transformation des variables consiste à appliquer des algorithmes qui vont modifier leur nature pour améliorer les résultats de la modélisation. Explorons ces possibilités en commençant par les variables numériques.

Les variables numériques peuvent subir deux types de transformations, qui ne sont pas obligatoires mais doivent être appliquées dans cet ordre : d'abord une transformation de distribution, puis une mise à l'échelle.

Voyons en détail ces transformations :

Les transformations de distribution

Les transformations de distribution consistent à appliquer des algorithmes sur les variables numériques pour rendre leur distribution plus normale. Ces actions ont un effet parfois bénéfique, notamment pour les algorithmes linéaires, mais ce n'est jamais garanti. Il faut essayer.

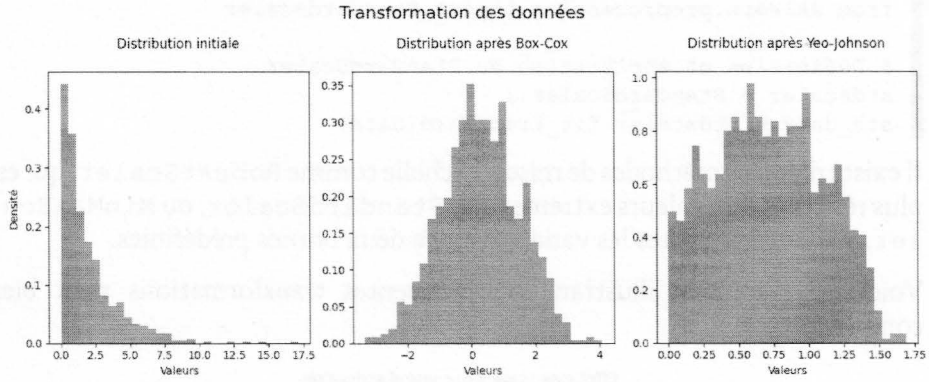
Différents types de transformations peuvent être appliquées en fonction du problème de distribution à corriger : log-transformation, transformation carrée, cubique, racine carrée ou des versions plus évoluées comme Yeo-Johnson ou Box-Cox.

Pour ce faire, Scikit-Learn propose deux fonctions du module preprocessing : `PowerTransformer` et `FunctionTransformer`.

Voici comment appliquer le `PowerTransformer` proposant deux méthodes différentes `yeo-johnson` et `box-cox` :

```
from sklearn.preprocessing import PowerTransformer

pt = PowerTransformer(method='yeo-johnson')
transformed_data = pt.fit_transform(data)
```

Un exemple visuel pour comprendre l'action de ces transformations :



La méthode `FunctionTransformer` permet de déployer tout type de transformation comme les fonctions logarithme, inverse, carrée ou cubique, entre autres. Voici un exemple de mise en œuvre de la fonction logarithme :

```
from sklearn.preprocessing import FunctionTransformer
import numpy as np

# Log transformation
log_transformer = FunctionTransformer(np.log1 p, validate=True)
log_transformed_data = log_transformer.fit_transform(data)
```

Remarque

La fonction `log1p` de Numpy est nommée ainsi car elle ajoute 1 à chaque valeur avant de calculer le logarithme naturel. Cela permet d'éviter les problèmes liés au calcul du logarithme de 0, qui est impossible et entraîne une valeur infinie.

Les normalisations et mises à l'échelle

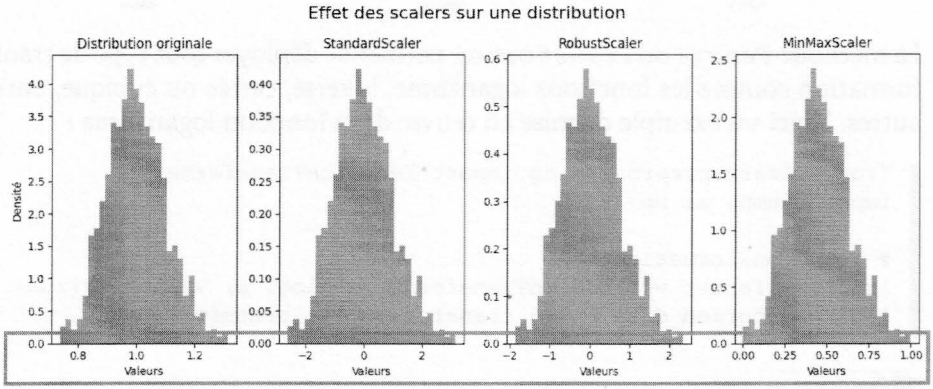
Les normalisations et mises à l'échelle, contrairement aux transformations du point précédent, sont beaucoup plus courantes. Elles ont pour but de remettre toutes les variables à la même échelle et éviter ainsi que les variables ayant des échelles de variabilité plus importantes invisibilisent celles dont la variabilité est faible. Elles consistent, pour donner une image, à remettre tous les coureurs sur la même ligne de départ.

Voici, par exemple, le code nécessaire pour standardiser les données :

```
from sklearn.preprocessing import StandardScaler

# Définition et application du StandardScaler
stdscaler = StandardScaler()
std_data = stdscaler.fit_transform(data)
```

Il existe d'autres méthodes de mise à l'échelle comme `RobustScaler`, qui est plus résistant aux valeurs extrêmes que `StandardScaler`, ou `MinMaxScaler`, qui ramène toutes les variables entre deux bornes prédéfinies.

Voici un graphique illustrant ces différentes transformations pour bien comprendre :



Ce sont les échelles qui vont être modifiées. Avec `StandardScaler` et `RobustScaler`, les distributions sont centrées autour de 0 et normalisées, respectivement, par la moyenne et l'écart-type pour le `StandardScaler`, et par la médiane et l'écart interquartile pour le `RobustScaler`. Pour le `MinMaxScaler`, toutes les valeurs sont recalibrées entre 0 et 1.

La liste complète est accessible à cette adresse :

https://scikit-learn.org/stable/api/sklearn.preprocessing.html

La binarisation des variables catégorielles

Les possibilités de transformation des variables catégorielles, contrairement aux variables numériques, se résument à quelques rares options qui sont de plus obligatoires car le contenu en texte n'est pas géré par les algorithmes, sauf par de rares exceptions comme l'algorithme de l'arbre de décision ou des forêts aléatoires.

La binarisation (fonction `OneHotEncoder`) consistant à transformer chaque modalité sous forme de binaire (1/0) est la transformation la plus couramment utilisée et recommandée.

Signalons toutefois que d'autres types de transformations, comme le `LabelEncoder` ou `OrdinalEncoder`, peuvent être invoqués dans quelques cas bien spécifiques comme les échelles de satisfaction. Mais gardons bien en tête que la recode de mentions sans hiérarchie via des numéros est à proscrire car elle induit, de facto, un ordre qui pourrait perturber les modélisations.

```
from sklearn.preprocessing import OneHotEncoder

# Définition et application du OneHotEncoder
ohe = OneHotEncoder ()
ohe_data = ohe.fit_transform(data)
```

3.1.4 La mise en œuvre ciblée des transformations

Comme nous l'avons vu dans la sous-section Les transformations des variables, les variables numériques et catégorielles subissent des transformations spécifiques selon leur type. Il est donc important de pouvoir le préciser afin d'éviter, par exemple, de standardiser des variables binaires qui perdraient ainsi leur intérêt.

Scikit-Learn propose la fonction `ColumnTransformer` qui est particulièrement efficace pour appliquer des transformations différentes selon le type.

ColumnTransformer avec une seule action par type de variable

Débutons l'utilisation du `ColumnTransformer` par le cas de figure le plus simple : appliquer une seule transformation par variable. Prenons pour illustrer un dataset fictif avec trois variables numériques fictives nommées `num1`, `num2` et `num3` ; et deux variables catégorielles fictives nommées `categ1` et `carteg2`. Nous souhaitons appliquer un `StandardScaler` sur les deux premières numériques et un `MinMaxScaler` sur la dernière. Concernant les variables qualitatives, elles subiront une transformation de binarisation en utilisant le `OneHotEncoder` :

```
# Import des fonctions nécessaires
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder,
MinMaxScaler


# Création de l'instance de transformation
column_transformer = ColumnTransformer(transformers=[
        ('num_standard', StandardScaler(), ["num1", "num2"]),
        ('num_minmax', MinMaxScaler(), ["num3"]),
        ('cat', OneHotEncoder(drop='first'), ["categ1", "categ2"])
    ],
    remainder='passthrough'  # Pour passer les colonnes non spécifiées
)

# Application sur les données
# fit_transform pour le jeu d'entraînement
X_train_scaled = column_transformer.fit_transform(X_train)

# Juste transform pour le jeu de test
X_test_scaled = column_transformer.transform(X_test)
```

Deux remarques importantes concernant cet extrait de code :

- L'utilisation de `drop='first'` dans `OneHotEncoder(drop='first')` sert à éviter la colinéarité. En supprimant la première catégorie de chaque variable, cela élimine la redondance puisque la catégorie restante peut être déduite des autres.
- L'option `remainder='passthrough'` indique de ne pas tenir compte des autres variables pour lesquelles aucune transformation ne serait souhaitée. Il serait aussi possible, dans ce cas, de les supprimer en utilisant la commande `remainder='drop'`.

ColumnTransformer avec plusieurs actions ordonnées par variable

L'application de transformations multiples et ordonnées est l'occasion d'introduire la fonction `Pipeline` de Scikit-Learn. Cette dernière permet de chaîner plusieurs étapes de traitement, telles que la preprocessing des données et l'entraînement du modèle, tout en garantissant que ces actions sont exécutées dans un ordre précis. Grâce à l'option `steps`, on peut structurer clairement chaque transformation, ce qui rend le code plus lisible et reproductible. Reprenons le dataset fictif précédent et appliquons plusieurs transformations aux deux premières variables numériques de cette façon :

```
# Import des fonctions nécessaires
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import RobustScaler, OneHotEncoder,
PowerTransformer, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline


# Création de l'instance de transformation incluant des Pipelines
column_transformer = ColumnTransformer(
    transformers=[
        ('numeric1', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='median')),
            ('power', PowerTransformer(method='yeo-johnson')),
            ('scaler', RobustScaler())
        ]), ["num1"]),
        ('numeric2', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='mean')),
            ('scaler', StandardScaler())
        ]), ["num2"]),
        ('cat', OneHotEncoder(drop='first'), ["categ1", "categ2"])
    ],
    remainder='drop'  # suppression des autres variables non
spécifiées
)

# Application sur les données
X_train_scaled = column_transformer.fit_transform(X_train)
X_test_scaled = column_transformer.transform(X_test)
```

Cette fonction `Pipeline` est très pratique et pourra être réutilisée ensuite dans le cadre des tests des algorithmes nécessitant eux aussi des actions ordonnées.

3.1.5 Finalisation de la préparation des données

Avant de passer à l'expérimentation, certaines actions sont encouragées comme, par exemple :

- Renommer les colonnes des variables binarisées pour obtenir leurs noms plutôt qu'un numéro d'ordre dans le but de faciliter la compréhension ultérieure des résultats. L'opération consiste à récupérer les noms des variables binarisées et à les assembler avec les autres qui n'ont pas subi ce type de transformation :

```
# Récupération des noms des variables transformées
transformer =
 column_transformer.named_transformers_['transformer_name']
transformed_feature_names = list(
    transformer.get_feature_names_out(["original_column"])
)

# Création d'une liste globale des noms de variables
all_feature_names = ["non_transformed_feature"] +
transformed_feature_names

# Ajout des noms aux matrices
X_train_processed =
pd.DataFrame(X_train, columns=all_feature_names)
X_test_processed =
pd.DataFrame(X_test, columns=all_feature_names)
```

- Modifier le type des variables. Il est, en effet, courant de retrouver des variables binaires codées en `float64` alors qu'elles ne contiennent que des 0 ou des 1. Modifier leur type en `int8`, par exemple, permet de réduire l'usage de la mémoire pour la table concernée. Tous les calculs s'en trouvent ainsi accélérés :

```
# Calcul de l'utilisation de la mémoire avant modification du type
initial_memory_usage =
X_train_processed.memory_usage(deep=True).sum()

memory_start = initial_memory_usage / 1024 ** 2
print(f"Memory usage avant modification du type :
{memory_start:.2f} MB")

# Identification des colonnes commençant par un certain préfixe
prefix = 'prefix'  # Remplacer par le préfixe spécifique
```

```
cols = [col for col in X_train_processed.columns if
col.startswith(prefix)]

# Conversion des colonnes en int8
X_train_processed[cols] = X_train_processed[cols].astype('int8')
X_test_processed[cols] = X_test_processed[cols].astype('int8')

# Calcul de l'utilisation de la mémoire après modification du type
final_memory_usage =
X_train_processed.memory_usage(deep=True).sum()

memory_end = final_memory_usage / 1024 ** 2
print(f"Memory usage après modification du type :
{memory_end:.2f} MB")
```

Les données sont désormais prêtes pour l'expérimentation.

3.2 L'expérimentation

L'expérimentation consiste à trouver l'algorithme qui délivre les performances les plus élevées et les plus stables possibles. Regardons ensemble les grandes étapes à mener pour y parvenir :

- définition des métriques pour l'évaluation ;
- détermination des outils nécessaires pour tester les hyperparamètres ;
- mise en place d'un système de base qui servira de référence ;
- tests des divers algorithmes avec différentes combinaisons d'hyperparamètres ;
- évaluation globale des algorithmes pour déterminer le plus performant.

Remarque

Un hyperparamètre est un paramètre d'un algorithme défini avant l'entraînement du modèle et qui contrôle le fonctionnement de l'algorithme.

3.2.1 Définition des métriques pour l'évaluation

Dans les modélisations, nous départageons les algorithmes grâce aux métriques. Le choix de tel ou tel algorithme sera réalisé en fonction de ces mesures. Il est donc important de définir quelles sont les métriques à utiliser pour les régressions ou pour les classifications.

Les métriques des régressions

Trois métriques sont importantes pour évaluer les prédictions d'une valeur numérique : le coefficient de détermination, la RMSE et la MAE.

- Le **coefficient de détermination** (R^2) indique la part de variance expliquée par le modèle par rapport à la variance totale des données. Nous l'avons déjà rencontré en définissant la méthode `score()` à la sous-section La méthode score() de ce chapitre. C'est le premier indicateur à regarder pour évaluer si le système de variables sélectionnées est correct ou non.

Les deux indicateurs suivants représentent deux façons de mesurer l'erreur moyenne :

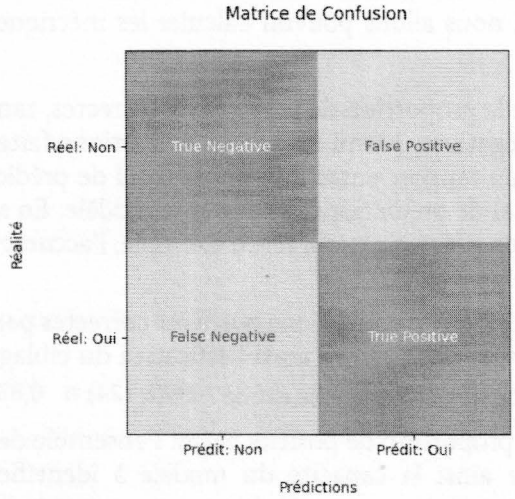
- La **RMSE** (*Root Mean Square Error*) ou erreur moyenne quadratique : cette métrique mesure la racine carrée de la moyenne des carrés des erreurs entre les prédictions du modèle et les valeurs réelles. Elle est particulièrement sensible aux grandes erreurs qu'elle pénalise plus fortement.
- La **MAE** (*Mean Absolute Error*) ou erreur absolue moyenne : cette métrique mesure la moyenne des valeurs absolues des erreurs entre les prédictions du modèle et les valeurs réelles. Contrairement à la RMSE, la MAE est moins sensible aux grandes erreurs donc elle reflète une erreur moyenne non influencée par des valeurs extrêmes.

L'usage complémentaire de ces trois mesures est recommandé car il permet d'avoir une vision globale de la performance des prédictions.

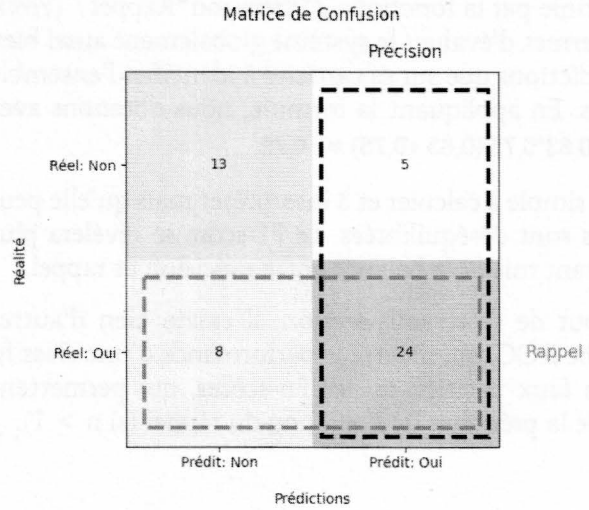
Les métriques des classifications

Dans le cadre des classifications, quatre mesures principales peuvent être utilisées selon le domaine d'étude. Ces mesures, parmi d'autres possibles, proviennent toutes de la matrice de confusion qui est indispensable pour contrôler la répartition entre les prédictions et les véritables catégories.

Voici à quoi ressemble une matrice de confusion classique indiquant les emplacements des prédictions correctes (*True Negative* et *True Positive*) et des incorrectes (*False Negative* et *False Positive*) :



Et à titre illustratif, voici une matrice de confusion avec des effectifs fictifs qui servira d'exemple pour calculer les différentes métriques :



Cette matrice fournit un point précis sur la répartition des prédictions par rapport à la réalité. Dans une situation idéale, seule la diagonale correspondant aux prédictions correctes devrait être remplie indiquant une exactitude parfaite. En pratique, certaines prédictions sont fausses et la matrice va nous aider à les identifier. Grâce à elle, nous allons pouvoir calculer les métriques suivantes :

- L'**exactitude** (Accuracy) donne la proportion de prédictions correctes, tant pour les classes positives que négatives, parmi toutes les prédictions faites par le modèle. Elle correspond au rapport entre le nombre total de prédictions correctes et le nombre total de prédictions faites par le modèle. En se basant sur la matrice de confusion précédente qui sert d'exemple, l'accuracy est de : (13 +24)/(13 +24 +5 +8) = 0,74.
- La **précision** représente la proportion de prédictions positives correctes parmi toutes les prédictions positives. Elle mesure ainsi l'efficacité du ciblage des prédictions positives. Dans notre exemple, elle est de : 24/(5 +24) ≈ 0,83.
- Le **rappel** (ou sensibilité) est la proportion de positifs parmi l'ensemble des échantillons positifs. Il mesure ainsi la capacité du modèle à identifier correctement les cas positifs. Le calcul correspond dans notre exemple à : 24/(8 +24) = 0,75.
- Le **F1-score** prend en compte les deux mesures précédentes et renvoie leur moyenne harmonique définie par la fonction : 2*Précision*Rappel / (Précision+Rappel). Ce score permet d'évaluer le système globalement aussi bien sur la précision de ses prédictions que sur sa capacité à identifier l'ensemble des observations positives. En appliquant la formule, nous obtenons avec nos données fictives : 2 * 0,83*0,75/(0,83 +0,75) ≈ 0,79.

Notons que l'exactitude est simple à calculer et à interpréter mais qu'elle peut être trompeuse si les classes sont déséquilibrées. Le F1-score se révélera plus robuste dans ce cas en reflétant mieux la balance entre précision et rappel.

Comme mentionné au début de cette sous-section, il existe bien d'autres indicateurs, tels que la courbe ROC, qui illustre la performance d'une classification entre les vrais et les faux positifs, ou les Fn-scores, qui permettent d'augmenter l'importance de la précision (si n < 1) ou du rappel (si n > 1).

À présent que nous savons évaluer une modélisation selon le cas, nous allons faire connaissance avec les fonctions permettant de tester différents hyperparamètres d'un même algorithme.

3.2.2 Les algorithmes d'optimisation d'hyperparamètres

Les algorithmes de Machine Learning possèdent tous des hyperparamètres sur lesquels nous pouvons jouer pour améliorer leurs performances.

Dans ce but, nous allons utiliser un algorithme d'optimisation d'hyperparamètres pour trouver les meilleures combinaisons. Ce dernier évalue chaque combinaison en utilisant la validation croisée qui est une méthode où les données sont divisées en plusieurs sous-ensembles (folds). Le modèle est ainsi entraîné sur certains folds et testé sur le fold restant, ce processus étant répété plusieurs fois pour assurer des évaluations fiables et généralisables.

Il existe de nombreuses solutions d'optimisation mais nous nous restreindrons à deux fonctions de Scikit-Learn :

- **GridSearchCV** est une fonction bien connue d'optimisation. Elle explore de manière exhaustive toutes les combinaisons, ce qui la rend coûteuse en temps. Nous la recommandons donc lorsque le nombre de combinaisons à tester n'est pas trop important, bien qu'elle puisse être optimisée en réduisant le nombre de variables ou en parallélisant les calculs.
- **RandomizedSearchCV** est destinée à des optimisations comportant de nombreuses combinaisons. Son exploration aléatoire est bien plus rapide mais ne garantit pas de trouver la meilleure combinaison.

Il est temps de mettre en œuvre ces fonctions et de débuter la modélisation.

3.2.3 Le modèle de base (DummyRegressor et DummyClassifier)

Nous allons débuter la modélisation par la mise en place d'un modèle de base (Dummy) qui a pour but de révéler la performance minimale attendue. Il va servir de point de comparaison pour évaluer les résultats des modèles plus complexes.

Deux fonctions sont fournies par Scikit-Learn pour ce faire :

- Le `DummyRegressor` permet d'établir la performance de base pour une régression. Ce n'est pas le R^2, ici, qui sera intéressant car il tournera autour de zéro. En revanche, les valeurs fournies de RMSE et la MAE seront une information précieuse pour prendre la mesure des deux types d'erreur moyenne.
- Le `DummyClassifier` jouera un rôle similaire pour la classification mais l'information la plus pertinente proviendra du choix de la stratégie :
  - Si la stratégie « `most_frequent` » est sélectionnée, cela signifie que prédire la classe majoritaire engendre le meilleur score.
  - Si c'est la stratégie « `stratified` », cela signifie que c'est la distribution des classes qui prime.
  - La stratégie « `uniform` » effectuera des prédictions totalement aléatoires sans respecter la distribution réelle des classes. S'il y a n classes, chacune aura 1/n chance d'être prédite.

Voici un exemple de mise en place d'un `DummyClassifier` sur le jeu des iris pour savoir quelle stratégie est la plus pertinente et afficher la valeur d'accuracy. L'entraînement d'un algorithme avec grille de recherche fait toujours appel à ces étapes dans cet ordre :

- import de l'algorithme ;
- création d'une instance de l'algorithme ;
- mise en œuvre du dictionnaire des hyperparamètres à expérimenter ;
- définition de la grille de recherche ;
- entraînement de la grille de recherche ;
- récupération des meilleurs paramètres et des meilleurs scores.

Le code suivant reprend les étapes précédemment énoncées :

```
from sklearn.datasets import load_iris
from sklearn.model_selection import GridSearchCV
from sklearn.dummy import DummyClassifier

# Chargement des données
iris = load_iris()
X = iris.data
```

```
y = iris.target

# Définition du DummyClassifier
dummy = DummyClassifier()

# Paramètres pour GridSearchCV
strategies = ["most_frequent", "prior", "stratified", "uniform"]
param_grid = {'strategy' : strategies}

# Création de l'objet GridSearchCV
grid_search = GridSearchCV(estimator=dummy,
                           param_grid=param_grid,
                           cv=5,
                           scoring='accuracy')

# Exécution de la recherche sur grille
grid_search.fit(X, y)

# Résultats
print("Meilleure stratégie :", grid_search.best_params_)
print("Meilleur score d'accuracy :", grid_search.best_score_)
```

```
Meilleure stratégie : {'strategy': 'most_frequent'}
Meilleur score d'accuracy : 0.3333333333333333
```

3.2.4 Tests des divers algorithmes avec différentes combinaisons de paramètres

Après avoir défini le modèle de base, nous allons expérimenter d'autres algorithmes dans le but d'obtenir de meilleures performances que les modèles `Dummy`.

L'entraînement de chaque algorithme va se dérouler de la même façon que pour le `DummyClassifier` en reprenant les mêmes étapes que celles définies à la sous-section Le modèle de base (DummyRegressor et DummyClassifier). Voici, pour illustrer, l'application d'une régression logistique sur le l'exemple précédent :

```
from sklearn.linear_model import LogisticRegression

# Définition du LogisticRegression
log_reg = LogisticRegression(max_iter=2000,solver='saga')
```

```
# Paramètres pour GridSearchCV
param_grid = {
    'C' : [1, 10],  # Exemples de valeurs pour le paramètre de
régularisation
    'penalty' : ['l2']  # Utilisation de la régularisation L2
}

# Création de l'objet GridSearchCV
grid_search = GridSearchCV(estimator=log_reg,
                           param_grid=param_grid,
                           cv=5,
                           scoring='accuracy')

# Exécution de la recherche sur grille
grid_search.fit(X, y)

# Résultats
print("Meilleurs paramètres :", grid_search.best_params_)
print("Meilleur score d'accuracy :", grid_search.best_score_)
```

```
Meilleurs paramètres : {'C': 10, 'penalty': 'l2'}
Meilleur score d'accuracy : 0.9800000000000001
```

Nous obtenons ici une excellente accuracy de 0,98 indiquant que l'algorithme retrouve l'espèce de quasiment tous les iris.

Pour les algorithmes ayant une différence trop importante entre le score d'entraînement et le score de test, il est possible de répondre à ce phénomène, nommé surapprentissage ou overfitting, en mettant en place des méthodes de généralisation telles que Ridge, Lasso et ElasticNet. Ces méthodes, qui s'appliquent aux régressions linéaire et logistique, fonctionnent de la façon suivante :

- **Ridge** réduit l'importance des coefficients pour minimiser le surapprentissage.
- **Lasso** sélectionne les variables les plus importantes en annulant les coefficients des moins importantes.
- **ElasticNet** combine les avantages de Ridge et Lasso en pénalisant à la fois la somme des carrés et la somme des valeurs absolues des coefficients.

3.2.5 L'évaluation et le choix final

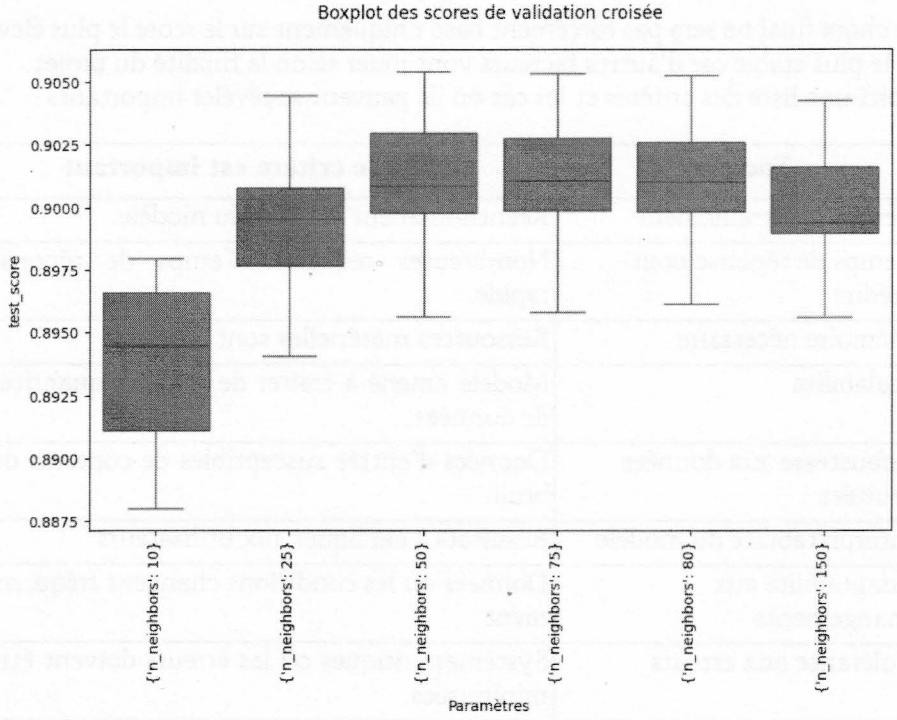
À l'issue de la modélisation, différents algorithmes vont ainsi être comparés dans le but de trouver le champion.

Le choix final ne sera pas forcément basé uniquement sur le score le plus élevé et le plus stable car d'autres facteurs vont jouer selon la finalité du projet. Voici une liste des critères et les cas où ils peuvent se révéler importants :

Facteur	Cas où le critère est important
Temps d'entraînement	Réentraînement régulier du modèle.
Temps de réponse pour prédire	Nombreuses requêtes. Temps de réponse rapide.
Mémoire nécessaire	Ressources matérielles sont limitées.
Scalabilité	Modèle amené à traiter de grandes quantités de données.
Robustesse aux données bruitées	Données d'entrée susceptibles de contenir du bruit.
Interprétabilité du modèle	Résultats à expliquer aux utilisateurs.
Adaptabilité aux changements	Données où les conditions changent fréquemment.
Tolérance aux erreurs	Systèmes critiques où les erreurs doivent être minimisées.
Poids du fichier de l'algorithme	Espace de stockage limité.

En fonction du contexte, les algorithmes devront être confrontés globalement sur les critères d'intérêt pour désigner la solution idéale. La visualisation est ici grandement encouragée pour y parvenir.

Voici un exemple fictif illustrant, en validation croisée, les différents scores réalisés par un KNN Regressor, qui prédit une valeur à partir des K voisins (paramètre à moduler) les plus proches :



La visualisation nous montre tout de suite qu'un optimum, en termes de score et de variabilité du score, semble être touché autour de 75-80 voisins et que les performances semblent se dégrader lorsque nous nous en éloignons. Nous pourrions ainsi adapter de manière plus précise le nombre de voisins pour toujours maximiser nos chances de tendre vers la modalisation optimale.

4. Conclusions sur la modélisation

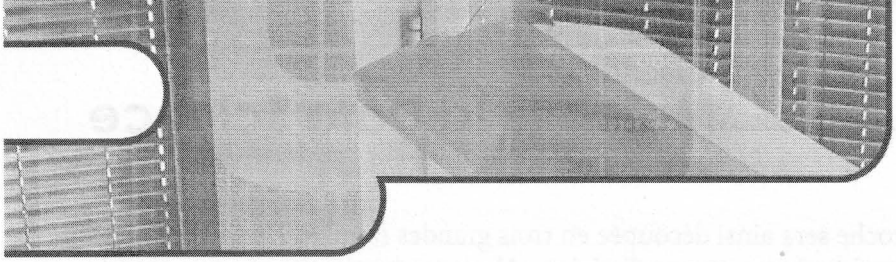
En guise de synthèse sur ce sujet, voici un récapitulatif des grandes étapes à mener pour réaliser cette partie oh combien importante. Les colonnes du tableau, Régression et Classification, indiquent les possibilités spécifiques offertes pour chaque étape :

Ordre	Étape	Régression	Classification
1	Sélection des variables descriptives et cible		
2	Split des données (train et test)		Stratify*
3	Transformation des colonnes	Binarisation des variables catégorielles Transformations et/ou mises à l'échelle des variables numériques	
4	Remise en forme et modification des types		
5	Entraînement d'un modèle de base	DummyRegressor	DummyClassifier
6	Entraînement de différents algorithmes	Régression linéaire et algorithmes dont le nom se termine par Regressor	Régression logistique et les algorithmes dont le nom se termine par Classifier
7	Évaluation des métriques	R^2 RMSE MAE	Accuracy Précision Recall F1-score Matrice de confusion

Ordre	Étape	Régression	Classification
8	Évaluation multicritères	Scores Variabilité des scores Temps d'entraînement Temps de calcul Poids des fichiers des algorithmes	

*`Stratify` : option utilisée pour que la répartition des classes dans le jeu d'entraînement et de test respecte la même proportion que dans l'ensemble du jeu de données.

Cette liste n'est, bien entendu, pas exhaustive, tant Python regorge de modules divers et variés pour nous aider à améliorer continuellement nos résultats. Elle contient cependant tout ce qui est nécessaire pour mener à bien une modélisation.



Chapitre 7
L'apprentissage supervisé

1. Introduction

L'apprentissage supervisé vise à prédire une valeur ou l'appartenance à une catégorie en s'appuyant sur un large choix d'algorithmes dédiés. L'objet de ce chapitre est de faire connaissance avec ces différentes familles d'algorithmes et de les mettre, ensuite, en pratique dans un exemple complet de régression puis de classification.

2. Les familles d'algorithmes

Nous avons vu précédemment que les algorithmes de Scikit-Learn sont bien identifiés en fonction de leur finalité : les algorithmes dédiés à la régression ont tous un nom qui se termine par `Regressor` et ceux dédiés à la classification ont tous un nom se finissant par `Classifier`.

Nous allons ici dépasser cette notion de destination pour nous intéresser à l'esprit de chaque algorithme, indépendamment de sa finalité. Ce sera ainsi l'occasion de mieux connaître les caractéristiques de chacun et de les employer à meilleur escient.

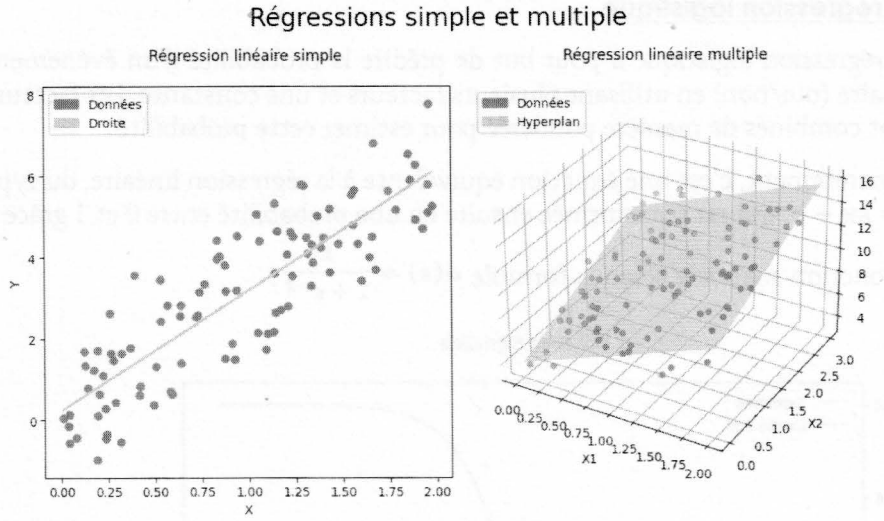
Notre approche sera ainsi découpée en trois grandes familles : les algorithmes linéaires, semi-linéaires et non linéaires. Nous y aborderons chaque grande famille d'algorithmes, comment les installer et les paramètres externes à ajuster pour optimiser leurs performances. Ces paramètres externes sont qualifiés d'hyperparamètres.

2.1 Les algorithmes linéaires

Les algorithmes linéaires cherchent basiquement à établir une relation linéaire entre des caractéristiques et une variable cible. Ce type de lien, plutôt simple, les destine à expliquer des systèmes peu complexes. Ce sont donc des options à privilégier dès le début qui refléteront, suivant les résultats, la complexité du système et serviront de référence pour comparer les performances d'algorithmes gérant des relations plus complexes. Découvrons sans tarder les membres de cette famille.

2.1.1 Les régressions

La régression linéaire et la régression logistique sont des techniques linéaires où l'effet des variables indépendantes sur la variable cible est supposé être additif et proportionnel. Ce sont les seuls algorithmes de Scikit-learn dont le nom ne se termine pas par `Regressor` ou `Classifier` bien qu'elles représentent respectivement chacun d'entre eux. Avant de passer à l'étude particulière de chacune d'elle, notons qu'il peut y avoir une ou plusieurs variables explicatives. Dans le cas d'une seule, nous parlons de régression simple où une droite sera ajustée aux points. S'il y en a plusieurs, il sera question de régression multiple où un hyperplan sera ajusté aux données.



La régression linéaire

La régression linéaire, illustrée dans ses versions simple et multiple dans le graphique précédent, permet de prédire une valeur numérique en utilisant plusieurs facteurs et une constante. Les facteurs sont combinés de manière pondérée pour obtenir une prédiction.

Il est possible, par exemple, de prédire le prix d'une maison en utilisant une somme pondérée de sa surface, de son emplacement et de son âge.

La régression linéaire est importée de cette manière :

```
from sklearn.linear_model import LinearRegression
```

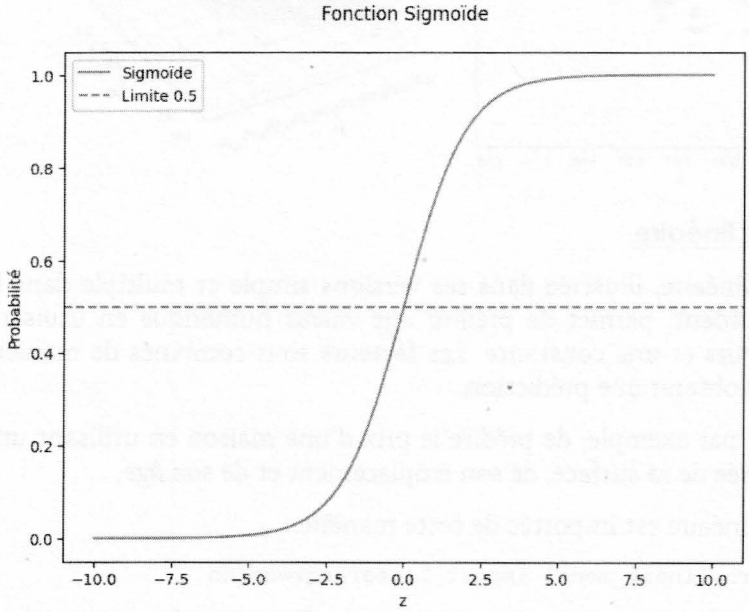
La liste de ses hyperparamètres n'est pas très étoffée. Voici les deux options intéressantes à expérimenter :

Hyperparamètres pertinents	Signification
`fit_intercept`: Booléen (par défaut=`True`)	Indique si l'interception (constante) du modèle doit être calculée
`positive`: Booléen (par défaut=`False`)	Force les coefficients à être positifs

La régression logistique

La régression logistique a pour but de prédire la probabilité d'un événement binaire (oui/non) en utilisant plusieurs facteurs et une constante. Les facteurs sont combinés de manière pondérée pour estimer cette probabilité.

Concrètement, c'est une équation équivalente à la régression linéaire, du type $z = ax + b$, qui est transformée ensuite en une probabilité entre 0 et 1 grâce à la fonction sigmoïde selon la formule $\sigma(z) = \frac{1}{1 + e^{-z}}$.



Grâce à cette transformation, il est possible d'attribuer à chaque observation une valeur entre 0 et 1 et ainsi procéder à la classification binaire en fonction de cette probabilité.

Nous pourrons, par exemple, évaluer la probabilité qu'un patient soit atteint d'une maladie en deux étapes : d'abord en utilisant une combinaison pondérée de son âge, de sa consommation journalière de certains produits, du nombre de pas réalisés par jour et d'une constante. Ensuite, en appliquant cette valeur à la courbe sigmoïde, nous obtiendrons la probabilité qu'il soit atteint de la maladie ou non.

L'appel à la régression logistique se fait de la manière suivante :

```
from sklearn.linear_model import LogisticRegression
```

Du point de vue des hyperparamètres, elle conserve la constante mais introduit en plus la possibilité d'agir sur le type et la force de la régularisation. Rappelons que la régularisation, abordée dans le chapitre Analyse des données, permet d'amoindrir et/ou de supprimer les coefficients pour améliorer la généralisation de la prédiction. Nous allons justement revenir dessus dans la sous-section suivante Les régressions régularisées :

Hyperparamètres Pertinents	Signification
`fit_intercept`: Booléen (par défaut=`True`)	Indique si l'interception (constante) du modèle doit être calculée.
`penalty`: Chaîne ('l1', 'l2', 'elasticnet', None, par défaut='l2')	Type de régularisation à appliquer.
`C`: Float (par défaut=1.0)	Inverse de la force de régularisation. Plus `C` est petit, plus la régularisation est forte.

2.1.2 Les régressions régularisées

Les méthodes comme Lasso, Ridge et ElasticNet ne sont pas uniquement des hyperparamètres de la régression logistique. Ce sont aussi des fonctions à part entière intégrées dans l'arsenal des techniques de modélisation et destinées à réduire la complexité du modèle en agissant sur les coefficients.

Elles vont permettre non seulement d'améliorer la généralisation, mais également d'offrir, par dévoiements, d'autres rôles que nous allons étudier.

La régression Lasso

La régression Lasso utilise la régularisation de type L1 pour réduire le nombre de variables et simplifier le modèle, améliorant ainsi sa capacité à se généraliser. En pratique, si un modèle de régression linéaire montre une performance d'entraînement nettement meilleure que celle du test, l'utilisation de la régression Lasso permet de réduire cette disparité en éliminant les variables moins importantes. Cela favorise la stabilité du modèle et améliore sa précision sur de nouvelles données.

La fonction est accessible de cette façon :

```
from sklearn.linear_model import Lasso
```

Elle reprend les hyperparamètres de la régression linéaire en ajoutant l'`alpha` qui permet d'ajuster l'importance de la régularisation Lasso.

Hyperparamètres pertinents	Signification
`fit_intercept`: Booléen (par défaut=`True`)	Indique si l'interception (constante) du modèle doit être calculée.
`alpha`: Float (par défaut=1.0)	Définit la force de la régularisation. Doit être compris entre 0 (aucune régularisation) et + l'infini.

Remarque

Il ne faut pas hésiter à expérimenter avec des valeurs d'alpha inférieures à 1 d'une part et grandement supérieures à 1 d'autre part pour renforcer l'effet de la régularisation. Les effets de ces variations ne se font pas ressentir de manière uniforme sur toutes les métriques de performance du modèle. Ainsi, il est crucial de mener des expérimentations afin de déterminer l'impact le plus bénéfique sur les métriques d'importance, en fonction du contexte spécifique.

Au-delà de son rôle de régularisation, la fonction Lasso peut être dévoyée comme un outil de réduction dimensionnelle destinée à la sélection des variables les plus importantes.

La régression Ridge

La régression Ridge illustre une autre approche de la régularisation en utilisant la norme L2, qui vise à réduire la valeur des coefficients. Contrairement à la norme L1 qui peut supprimer complètement certains, l'approche de la norme L2 est moins radicale mais contribue de manière similaire à réduire l'importance des variables les moins significatives.

Comme toutes les régressions, l'algorithme provient du module `sklearn.linear_model` :

```
from sklearn.linear_model import Ridge
```

Nous retrouvons aussi exactement les mêmes hyperparamètres que pour la régression Lasso.

En revanche, les détournements de la fonction Ridge sont bien différents de ceux de Lasso. En pénalisant les coefficients des variables, Ridge réduit les effets négatifs de la multi-colinéarité et stabilise les estimations des coefficients. Cette stabilité améliore la robustesse du modèle particulièrement lorsque l'échantillon est faible ou face à des données bruitées.

La régression ElasticNet

La régression ElasticNet combine les avantages de Lasso et Ridge, réduisant ainsi la complexité du modèle et la colinéarité des variables. Cet algorithme est particulièrement performant lorsque les variables explicatives sont nombreuses et corrélées.

Nous retrouvons cet algorithme dans le même module que tous les autres :

```
from sklearn.linear_model import ElasticNet
```

Au niveau des hyperparamètres, nous retrouvons `alpha` et `fit_intercept` qui sont communs aux deux autres régularisations mais ElasticNet hérite en plus de `l1_ratio` qui permet de définir la part de Lasso et réciproquement Ridge dans la régularisation.

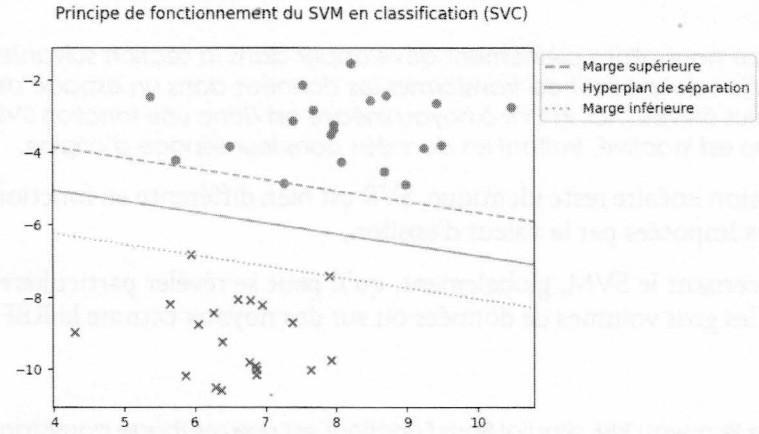
Hyperparamètres pertinents	Signification
`fit_intercept`: Booléen (par défaut=`True`)	Indique si l'interception (constante) du modèle doit être calculée.
`alpha`: Float (par défaut=1.0)	Définit la force de la régularisation. Doit être compris entre 0 (aucune régularisation) et + l'infini.
`l1_ratio`: Float (par défaut=0.5)	Définit la part de régularisation L1 et réciproquement, L2.

2.1.3 Les machines à vecteur de support (SVM)

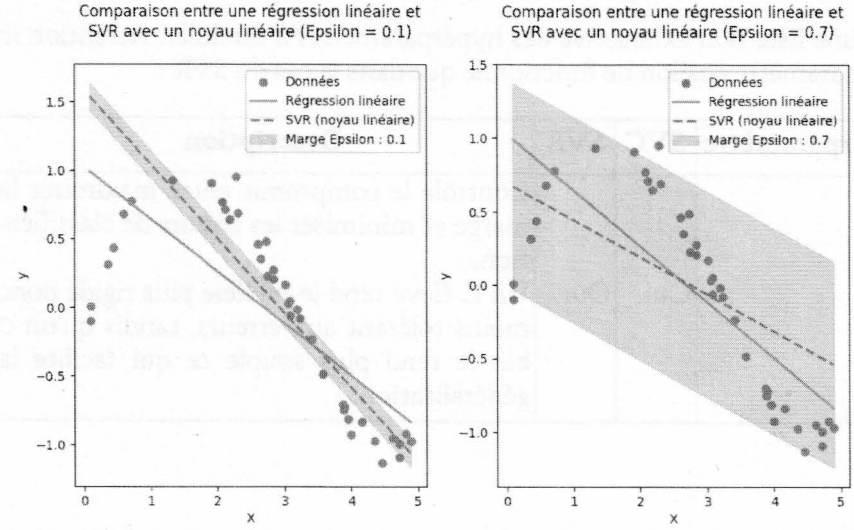
L'algorithme des machines à vecteur de support (SVM) est une méthode d'apprentissage supervisé utilisée à la fois pour la classification (SVC) et la régression (SVR).

En classification, l'objectif principal du SVM est de trouver une droite ou un hyperplan qui sépare au mieux les différentes classes de données, en maximisant la marge de séparation entre elles tout en tolérant quelques points mal prédits. Pour ce faire, SVM utilise une marge souple (*soft margin*) qui tolère certaines erreurs de classification grâce au paramètre. Ainsi, un *C* élevé favorise des marges plus rigides tandis qu'un *C* bas est plus souple et facilite la généralisation.

Contrairement à d'autres algorithmes, SVM se concentre principalement sur les points de données proches de la frontière de décision, appelés vecteurs de support, car ce sont ces points qui déterminent la position de l'hyperplan. Cette spécificité renforce son efficacité et sa robustesse aux données bruitées ou aux points éloignés.



Dans sa version dédiée à la régression (SVR), la logique de fonctionnement est un peu différente. En fonction de la valeur d'epsilon, l'algorithme va chercher à maximiser le nombre de points à l'intérieur de la marge tout en limitant le nombre et la distance des points en dehors. En fonction de la valeur d'epsilon, l'inclinaison et l'ordonnée de la courbe peuvent beaucoup varier. Voici un exemple présentant une régression linéaire et une SVR à noyau linéaire pour deux valeurs d'epsilon :



Remarque

Le noyau, que nous allons pleinement développer dans la section suivante, est une fonction qui permet de transformer les données dans un espace de dimensions plus élevées. Ici, la SVR à noyau linéaire est donc une fonction SVR dont le noyau est inactivé, traitant les données dans leur espace d'origine.

Là où la régression linéaire reste identique, SVR est bien différente en fonction des contraintes imposées par la valeur d'epsilon.

Signalons concernant le SVM, globalement, qu'il peut se révéler particulièrement lent sur les gros volumes de données ou sur des noyaux comme le RBF.

Remarque

Précisons que le noyau RBF (Radial Basis Function) est une méthode consistant à transformer les données d'entrée en un espace de caractéristiques de plus hautes dimensions pour les rendre linéairement séparables.

L'importation des fonctions se réalise de la façon suivante :

```
# SVR version régression
from sklearn.svm import SVR

# SVC version classification
from sklearn.svm import SVC
```

Voici une liste non exhaustive des hyperparamètres à moduler. Attention ici car le paramètre epsilon ne fonctionne que dans le cas du SVR :

Hyperparamètre	SVC	SVR	Description
C	Oui	Oui	Contrôle le compromis entre maximiser la marge et minimiser les erreurs de classification. Un C élevé rend le modèle plus rigide donc moins tolérant aux erreurs, tandis qu'un C bas le rend plus souple ce qui facilite la généralisation.

Hyperparamètre	SVC	SVR	Description
`Kernel`	Oui	Oui	Spécifie le type de noyau à utiliser pour transformer les données. Les options courantes incluent `'linear'`, `'poly'`, `'rbf'`, `'sigmoid'` et `'precomputed'`.
`Epsilon`	Non	Oui	Détermine une zone de tolérance dans laquelle les erreurs de prédiction ne sont pas pénalisées. Plus `epsilon` est grand, plus le modèle est tolérant aux erreurs de prédiction.

Le changement de noyau occupe une place très importante pour l'algorithme SVM. C'est d'ailleurs ce qui va faire de ces algorithmes linéaires, des algorithmes semi-linéaires soulignant ainsi leur nouvelle capacité à gérer des phénomènes non linéaires suite à la modification des données en amont. Nous allons justement développer ces possibilités dans la section suivante.

2.2 Les algorithmes semi-linéaires (modèles à noyau)

Les algorithmes à noyau (semi-linéaires) désignent des algorithmes linéaires rendus efficients sur des problèmes non linéaires grâce à l'application d'un noyau (linéaire, polynomial, RBF) ou à des transformations explicites (carré, racine carrée, log) des données d'entrée.

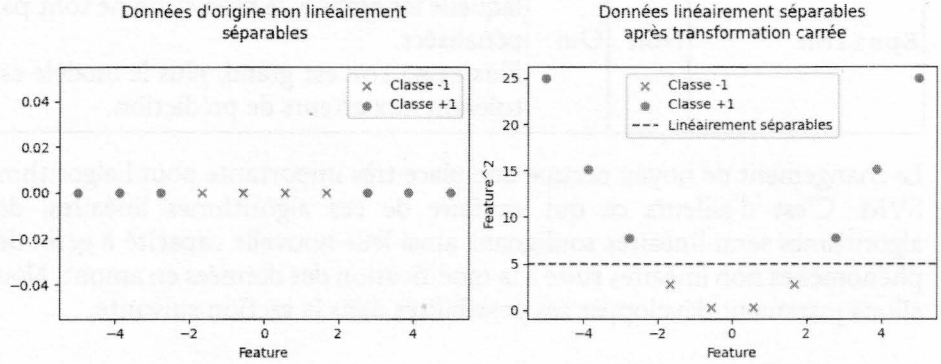
La différence entre une transformation et un noyau réside dans leur effet sur la représentation des données :

- **Transformation des variables** : chaque variable est modifiée individuellement, par exemple, en appliquant des fonctions comme le carré, la racine carrée ou le logarithme.
- **Utilisation d'un noyau** : modifie la représentation des données pour inclure des interactions et des relations non linéaires entre toutes les variables simultanément.

Remarque

Ces techniques ne sont pas limitées uniquement aux seuls algorithmes linéaires et peuvent être largement utilisées dans divers contextes d'apprentissage automatique.

Voici un exemple simple de transformation d'une variable pour illustrer l'intérêt de ces actions. Les points deviennent ici linéairement séparables après le passage des données au carré :



Nous allons calculer les différents scores obtenus pour un même jeu de données fictif et les comparer sans transformation, avec transformation de la variable et avec l'application de la technique du noyau.

Commençons par importer les modules nécessaires et créer un jeu de données aléatoire avec un lien logarithmique entre X et y :

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.kernel_ridge import KernelRidge
from sklearn.metrics import r2_score
from sklearn.linear_model import Ridge

# Création de données non linéaires
np.random.seed(0)
X = np.random.uniform(1, 10, 100).reshape(-1, 1)
y = 2 * np.log(X).flatten() + np.random.normal(0, 0.3, 100)
```

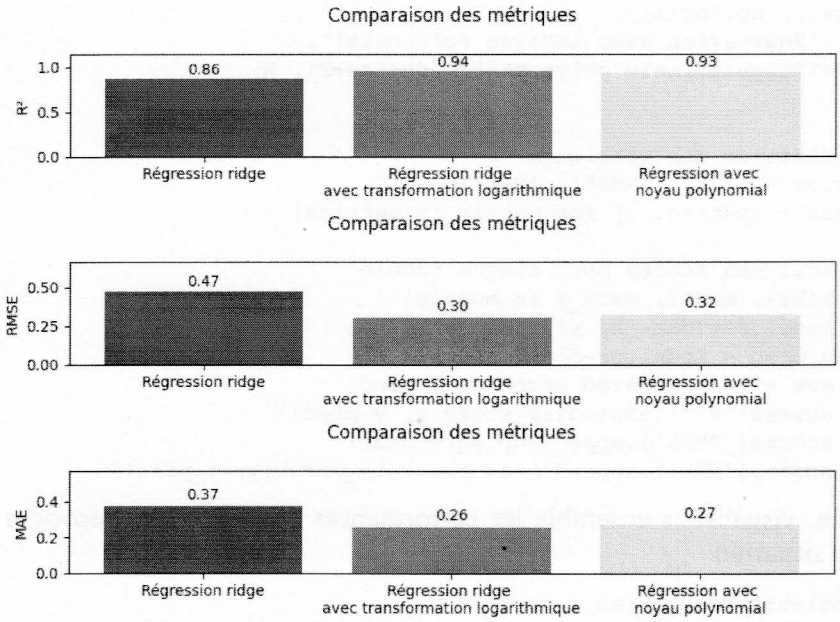
Modélisons maintenant les trois situations pour une régression Ridge :

```
# Définition des modèles
models = [
    ('Régression Ridge', Ridge(), X),
    ('Régression Ridge \navec Transformation Logarithmique',
Ridge(), np.log(X)),
    ('Régression avec \nNoyau Polynomial',
KernelRidge(kernel='polynomial', degree=2), X)
]

# Définition des métriques
metrics = ['R²', 'RMSE', 'MAE']
scores = {metric: [] for metric in metrics}

# Calcul des scores pour chaque modèle
for label, model, data_X in models:
    model.fit(data_X, y)
    y_pred = model.predict(data_X)
    mse = mean_squared_error(y, y_pred)
    scores['R²'].append(r2_score(y, y_pred))
    scores['RMSE'].append(np.sqrt(mse))
    scores['MAE'].append(mean_absolute_error(y, y_pred))
```

Et enfin, visualisons ensemble les performances des métriques associées à chaque situation :

```
# Couleurs des barres
colors = ['#285E78', '#439CC8', '#EEEEEE']

# Création du graphique en barres pour les scores
plt.figure(figsize=(8, 6))

for i, metric in enumerate(metrics):
    plt.subplot(len(metrics), 1, i+1)
    bars = plt.bar([label for label, _, _ in models],
                   scores[metric],color=colors)
    plt.ylabel(metric)
    plt.title(f'Comparaison des métriques\n')
    plt.ylim(0, max(scores[metric]) + 0.2)

    # Ajout des valeurs numériques au-dessus des barres
    for bar, score in zip(bars, scores[metric]):
        plt.text(bar.get_x() + bar.get_width()/2,
```

```
                bar.get_height() + 0.02, f'{score:.2f}',
                ha='center', va='bottom')

plt.tight_layout()
plt.show()
```



Dans cet exemple, les transformations et changements de noyau améliorent indéniablement toutes les métriques. Il sera donc judicieux de considérer ces options dans nos futures expérimentations.

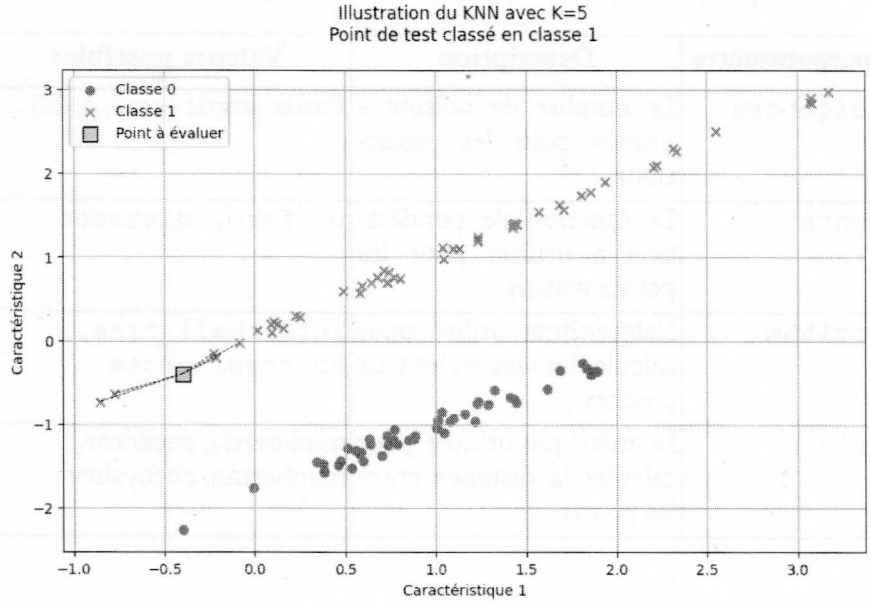
2.3 Les algorithmes non linéaires

Nous terminerons notre tour d'horizon par les algorithmes non linéaires. Ces méthodes sont conçues pour capturer des relations non linéaires, souvent plus complexes, ce qui leur permet d'offrir des performances potentiellement supérieures. Cependant, cette capacité à modéliser la complexité des données apporte son lot de contraintes supplémentaires : une interprétation plus difficile des modèles, des besoins accrus en termes de données et une sensibilité plus élevée à l'overfitting et aux réglages des hyperparamètres.

Comme pour les autres algorithmes, nous allons favoriser une approche par familles en commençant par les plus proches voisins, puis l'arbre de décision, les méthodes ensemblistes et les réseaux de neurones pour terminer.

2.3.1 Les plus proches voisins (KNN)

L'algorithme des plus proches voisins repose sur le principe simple que des points qui sont proches dans l'espace des caractéristiques soient susceptibles d'appartenir à la même classe ou d'avoir des valeurs similaires.



Cet algorithme est très simple à comprendre, à implémenter et il est relativement robuste car il ne suppose aucune distribution sous-jacente des données. Cela lui permet de capturer des relations complexes et non linéaires.

Il faut cependant signaler qu'il n'est pas trop destiné aux grands ensembles de données car il souffre de deux défauts majeurs :

- une faible efficacité computationnelle car il doit calculer la distance entre tous les points ;
- le piège de la dimension signifiant que son efficacité diminue avec le nombre de variables qui rendent les distances moins discriminantes.

Le KNN fonctionne aussi bien en régression qu'en classification, voici donc les deux manières d'importer ses deux versions :

```
# KNN version régression
from sklearn.neighbors import KNeighborsRegressor

# KNN version classification
from sklearn.neighbors import KNeighborsClassifier
```

Voici, enfin, quelques hyperparamètres pertinents à évaluer :

Hyperparamètre	Description	Valeurs possibles
`n_neighbors`	Le nombre de voisins à utiliser pour les prédictions.	Entier positif (ex: 3, 5, 10)
`weights`	La fonction de pondération à utiliser pour les points voisins.	`uniform, distance`
`algorithm`	L'algorithme utilisé pour calculer les voisins les plus proches.	`auto, ball_tree, kd_tree, brute`
`metric`	La métrique utilisée pour calculer la distance entre les points.	minkowski, euclidean, manhattan, chebyshev

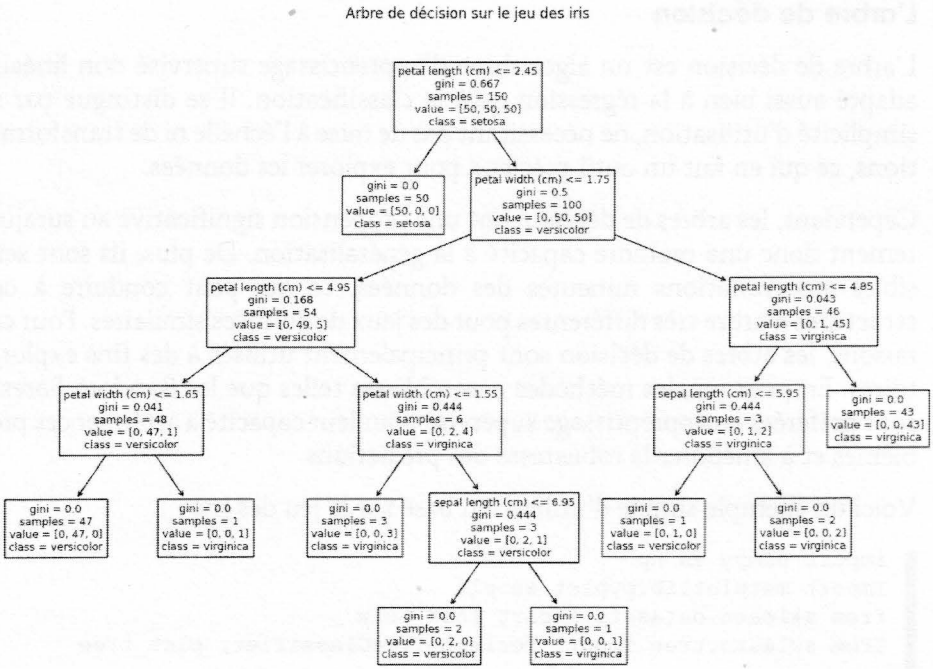
2.3.2 L'arbre de décision

L'arbre de décision est un algorithme d'apprentissage supervisé non linéaire adapté aussi bien à la régression qu'à la classification. Il se distingue par sa simplicité d'utilisation, ne nécessitant pas de mise à l'échelle ni de transformations, ce qui en fait un outil précieux pour explorer les données.

Cependant, les arbres de décision ont une propension significative au surajustement donc une moindre capacité à la généralisation. De plus, ils sont sensibles aux variations mineures des données, ce qui peut conduire à des structures d'arbre très différentes pour des jeux de données similaires. Pour ces raisons, les arbres de décision sont principalement utilisés à des fins exploratoires. En pratique, les méthodes ensemblistes telles que les Random Forests sont préférées en apprentissage supervisé pour leur capacité à atténuer ces problèmes et à améliorer la robustesse des prédictions.

Voici un exemple simple d'utilisation basé sur le jeu des iris :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree

# Import des données
iris = load_iris()
X = iris.data
y = iris.target

# Entraînement de l'arbre
clf = DecisionTreeClassifier(random_state=123)
clf.fit(X, y)

# Tracé de l'arbre de décision
plt.figure(figsize=(15, 10))
plot_tree(clf, filled=False,
          feature_names=iris.feature_names,
          class_names=iris.target_names)
plt.title('Arbre de Décision sur les Données Iris\n')
plt.show()
```



Le parcours de l'arbre permet d'identifier les caractéristiques les plus discriminantes, qui apparaissent en premier en haut de l'arbre. À chaque nœud, les échantillons positifs sont généralement situés à gauche et les échantillons négatifs à droite, en fonction de la condition définie par la caractéristique sélectionnée. La pureté d'un nœud est évaluée à l'aide de l'indice de Gini, qui varie de 0 à 1 : il atteint 0 lorsque tous les échantillons appartiennent à une seule classe, et 1 lorsque les échantillons sont répartis également entre toutes les classes possibles.

Ainsi, dans notre exemple, une longueur de pétale inférieure ou égale à 2.45 cm nous permettra d'identifier l'espèce setosa. Il faudra ensuite examiner d'autres variables pour pouvoir isoler complètement les observations des deux autres espèces.

2.3.3 Les méthodes ensemblistes

Les méthodes ensemblistes sont un maillon essentiel des algorithmes d'apprentissage automatique, non seulement pour les modèles non linéaires mais aussi pour les algorithmes en général. Elles se distinguent par leur capacité à combiner plusieurs modèles individuels pour améliorer la performance prédictive globale du système. Les principales techniques ensemblistes incluent le bagging, le boosting, le stacking et le voting, offrant chacune des avantages spécifiques selon le contexte et les données.

Bagging : le fondement de Random Forest

La technique du bagging consiste à entraîner en parallèle plusieurs modèles sur des sous-ensembles d'observations tirées au hasard avec remise. Chaque modèle individuel est ensuite combiné pour prédire soit une moyenne des scores (dans le cas d'une régression) ou un vote à la majorité (dans le cas d'une classification).

En procédant ainsi, nous récoltons de nombreux avantages comme :

- amélioration de la généralisation grâce à la réduction de l'overfitting ;
- réduction de l'impact des valeurs extrêmes ou manquantes ;
- la combinaison de modèles individuels favorise la stabilité ;
- ce type d'entraînement favorise la parallélisation des calculs.

Le célèbre Random Forest est le digne représentant de cette méthode. En plus de tous les avantages dont il hérite du bagging, il y ajoute une sélection aléatoire des caractéristiques pour encore renforcer sa robustesse ainsi qu'un feature importance permettant de mesurer l'importance des variables.

Voici la façon d'importer les deux versions de cet algorithme :

```
# Random forest version régression
from sklearn.ensemble import RandomForestRegressor

# Random forest version classification
from sklearn.ensemble import RandomForestClassifier
```

Random Forest possède de nombreux hyperparamètres mais voici ceux considérés comme les plus influents pour faciliter notre initiation :

Hyperparamètre	Description
`n_estimators`	Nombre d'arbres. Augmenter ce nombre peut améliorer la performance mais augmentera également le temps d'entraînement.
`max_depth`	Profondeur maximale de chaque arbre. Contrôle la complexité des arbres.
`max_features`	Nombre de caractéristiques à considérer pour la meilleure division. Une valeur plus faible réduit la diversité des arbres et peut améliorer la performance.

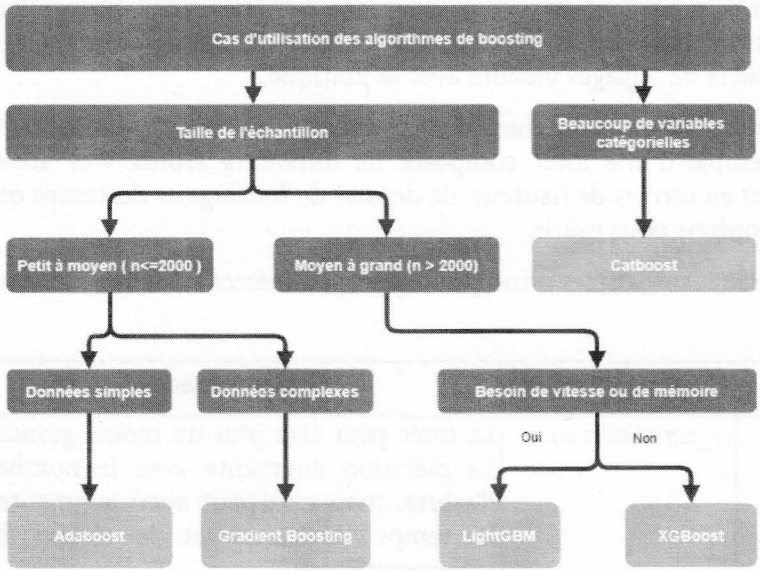
Boosting

La méthode de boosting propose un fonctionnement bien différent du bagging en favorisant des modèles faibles (des arbres peu profonds) améliorés de manière séquentielle par l'augmentation progressive du poids des mauvaises prédictions. Pour faire simple : le but est d'obtenir un modèle fort en combinant séquentiellement des modèles faibles.

Cette méthode apporte de nombreux avantages parmi lesquels :

- des scores de prédiction très élevés ;
- une réduction du biais et de la variance produisant des modèles plus stables ;
- une bonne gestion des données déséquilibrées, des valeurs aberrantes et des valeurs manquantes.

De nombreux algorithmes composent cette famille, c'est pourquoi nous allons utiliser le schéma suivant pour nous guider dans le choix de l'algorithme approprié en fonction du contexte.



Voyons maintenant comment importer ces algorithmes sachant que seuls les deux premiers proviennent de Scikit-Learn :

```
# AdaBoost
from sklearn.ensemble import AdaBoostClassifier,
from sklearn.ensemble import AdaBoostRegressor

# Gradient Boosting
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import GradientBoostingRegressor

# XGBoost
from xgboost import XGBClassifier
from xgboost import XGBRegressor

# LightGBM
from lightgbm import LGBMClassifier
from lightgbm import LGBMRegressor

# CatBoost
from catboost import CatBoostClassifier
from catboost import CatBoostRegressor
```

Les algorithmes possèdent beaucoup d'hyperparamètres donc nous allons nous concentrer sur ceux qui sont communs à tous. La maîtrise des réglages et des possibilités de réglages viendra avec la pratique.

Pour comprendre le fonctionnement d'un boosting et de ses hyperparamètres, prenons l'exemple d'une forêt composée de différents arbres. Ces arbres peuvent varier en termes de hauteur, de densité de feuillage et du temps que nous leur accordons pour mûrir.

Voici un tableau résumant les principaux hyperparamètres et leur fonctionnement :

Image	Hyperparamètre	Conséquence
Nombre d'arbres	`n_estimators`	La forêt peut être plus ou moins grande. La précision augmente avec le nombre d'arbres, mais cela peut aussi augmenter le temps de calcul et le risque de surapprentissage.
Hauteur des arbres	`max_depth`	Plus les arbres sont hauts, plus ils peuvent effectuer de décisions successives pour diviser les données. Cela représente le nombre de niveaux de décisions possibles dans chaque arbre.
Densité du feuillage	`num_leaves`	Plus les arbres ont de feuilles, plus ils peuvent prendre de décisions finales différentes. Cela représente la diversité des résultats finaux que chaque arbre peut produire.
Temps de maturation	`learning_rate`	Plus le temps de maturation est long, plus les arbres peuvent apprendre des informations détaillées et complexes. En boosting, cela se traduit par le taux d'apprentissage, qui contrôle à quel point chaque nouvel arbre ajuste les erreurs des précédents.

Stacking

Le stacking illustre une autre facette singulière des méthodes ensemblistes. Son objectif principal est de construire un métamodèle en combinant les prédictions de plusieurs modèles de base initialement sélectionnés. Le succès du stacking dépendra donc, non seulement des algorithmes de base sélectionnés, mais aussi de la diversité et de la complémentarité de ces modèles.

Bien que les besoins computationnels soient plus élevés, la combinaison d'algorithmes hétérogènes peut aboutir à des résultats supérieurs à ceux obtenus par chacun pris indépendamment.

Regardons ensemble comment mettre en place un stacking à travers le code suivant :

```
# 1 / Import des bibliothèques nécessaires
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import StackingClassifier


# 2 / Initialisation des modèles de base
linear_model = LogisticRegression(max_iter=1000,
                                  random_state=42)
knn_model = KNeighborsClassifier(n_neighbors=5)
rf_model = RandomForestClassifier(n_estimators=100,
                                  random_state=42)

# 3 / Initialisation du modèle de stacking
stacking_models = [
    ('logistic', linear_model),
    ('knn', knn_model),
    ('random_forest', rf_model)
]

# 4 / Instanciation du stacking
stacking_model = StackingClassifier(estimators=stacking_models,
final_estimator=LogisticRegression())

# 5 / Entraînement du modèle de stacking
stacking_model.fit(X_train, y_train)
```

La mise en place est relativement simple à mettre œuvre :

- import et initialisation des modèles de base (points 1 et 2 du code) ;
- création d'une liste des noms et instances de chaque modèle (point 3) ;
- instanciation de l'algorithme de stacking au point 4 en précisant la liste des algorithmes utilisés dans le paramètre `estimators` et l'algorithme utilisé pour les combiner tous ensemble dans `final_estimator` ;
- entraînement classique de l'algorithme ainsi créé (point 5).

Il est vivement conseillé ici de bien prendre des algorithmes hétérogènes afin d'exploiter les forces de chacun et ainsi renforcer le stacking résultant.

Voting

La méthode du voting consiste également à mettre à profit la combinaison de différents algorithmes de base mais le vote se fera ensuite soit à la majorité (hard voting), soit en prenant la moyenne pondérée des scores (soft voting).

À l'inverse du stacking, la sélection d'algorithmes homogènes, voire du même algorithme avec des hyperparamètres différents, est encouragée ici car ils ont tendance à produire des prédictions plus cohérentes qui peuvent améliorer l'efficacité du voting. La finalité recherchée ici consiste à s'appuyer sur la combinaison de différents algorithmes et hyperparamètres pour améliorer la robustesse et la généralisation de l'algorithme résultant.

La mise en place du voting est équivalente au stacking. Nous allons présenter un exemple basé sur le même algorithme :

```
# Import des bibliothèques nécessaires
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier

# Définition des modèles individuels
clf1 = RandomForestClassifier(n_estimators=10, random_state=1)
clf2 = RandomForestClassifier(n_estimators=50, random_state=1)
clf3 = RandomForestClassifier(n_estimators=100, random_state=1)

# Création du classificateur de vote avec le type de vote
eclf = VotingClassifier(
    estimators=[('rf_10', clf1), ('rf_50', clf2), ('rf_100', clf3)],
    voting='hard')
```

```
# Entraînement du voting
eclf.fit(X_train, y_train)
```

Dans le cas du voting, les deux hyperparamètres importants propres au système sont `estimators` et `voting` qui acceptent les options `'hard'` ou `'soft'`.

2.3.4 Les réseaux de neurones

Les réseaux de neurones occupent une place importante dans les algorithmes non linéaires car ils sont particulièrement efficaces pour traiter les problèmes complexes et multidimensionnels.

Inspirés par les neurones du cerveau humain, ils forment un réseau organisé en trois types de couches : la couche d'entrée qui reçoit les données, les couches intermédiaires, dites cachées, où s'opèrent les calculs et la couche de sortie qui renvoie la prédiction ou la représentation vectorielle des données appelées embedding.

Dans le cadre de l'apprentissage supervisé dans Scikit-Learn, nous pouvons utiliser les *Multi-Layer Perceptron* (MLP) en mode régression et classification. Il existe, cependant, bien d'autres types de réseaux de neurones que nous allons présenter succinctement car nous expérimenterons certains dans le chapitre Modéliser le texte et l'image.

Multi-Layer Perceptron

La méthode *Multi-Layer Perceptron* (MLP) est une implémentation d'un perceptron multicouche qui s'entraîne en utilisant la rétropropagation. La rétropropagation corrige l'erreur entre la prédiction et la réalité en ajustant les poids des connexions, en remontant progressivement de la couche de sortie vers la couche d'entrée, pour améliorer les prédictions du réseau.

Le MLP se déploie de la même manière que les autres algorithmes de Scikit-Learn en l'important de la façon suivante :

```
# Version régression
from sklearn.neural_network import MLPRegressor

# Version classification
from sklearn.neural_network import MLPClassifier
```

Comme tout réseau de neurones, le MLP possède de nombreux hyperparamètres ouvrant des possibilités immenses de réglages.

Hyperparamètre	Explication
`hidden_layer_sizes`	Détermine la structure des couches cachées. Par exemple `(100, 50)` signifie deux couches cachées avec 100 et 50 neurones respectivement.
`activation`	Spécifie la fonction d'activation utilisée pour les neurones. Les choix courants incluent `'relu'`, `'tanh'`, et `'logistic'`.
`solver`	L'algorithme utilisé pour l'optimisation. Les options incluent `'adam'`, `'sgd'` (gradient stochastique), et `'lbfgs'`.
`learning_rate`	Contrôle la vitesse à laquelle le modèle ajuste les poids. Peut être une valeur fixe (`'constant'`), adaptable (`'adaptive'`), ou décroissante (`'invscaling'`).
`alpha`	Valeur de la régularisation définie par défaut à 0.0001

Les différents réseaux de neurones

Les réseaux de neurones occupent depuis 2012 une place centrale dans la data science. Avec l'arrivée des *Large Language Model* (LLM) comme ChatGPT, cette influence a dépassé le cadre de cette discipline pour se diffuser largement dans le grand public.

Il est donc important de nommer et décrire le rôle de chaque type de réseaux pour avoir une vision générale des possibilités offertes par cette technologie. Voici un tableau récapitulatif pour ce faire :

Type de Réseau de neurones	Applications	Notes
Convolutional Neural Networks (CNN)	Vision par ordinateur Traitement d'images	Scikit-learn ne supporte pas directement les CNN. Utiliser Keras via KerasClassifier et KerasRegressor.
Region-based Convolutional Neural Networks (RCNN)	Détection d'objets Segmentation d'images	Utilisés pour des tâches plus spécifiques en vision par ordinateur. Implémentation plus complexe.
Recurrent Neural Networks (RNN)	Traitement du langage naturel Séries temporelles	Implémentés principalement dans des frameworks spécialisés.
Autoencoders	Apprentissage non supervisé Réduction de dimension Détection d'anomalies	Nécessitent des frameworks spécialisés pour une implémentation avancée.
Generative Adversarial Networks (GAN)	Génération d'images Amélioration de la qualité d'image Synthèse de données	Implémentation complexe nécessitant des frameworks spécialisés.
Transformers	Traitement du langage naturel Traduction automatique Génération de texte	Utilisés avec des bibliothèques dédiées comme Hugging Face.

Type de Réseau de neurones	Applications	Notes
Large Language Models (LLM)	Traitement du langage naturel Génération de texte Compréhension de texte	ChatGPT est un LLM basé sur l'architecture des transformers, alimenté par un vaste ensemble de données textuelles. Il est utilisé en conjonction avec des bibliothèques telles que Hugging Face pour faciliter son déploiement et son utilisation.

Il est maintenant temps de passer à la pratique et d'explorer le fonctionnement de ces algorithmes. Nous allons procéder à une application concrète à travers un exemple de régression et de classification.

3. La régression en pratique

C'est le jeu de données des diamants qui sera utilisé ici pour expérimenter les étapes d'une modélisation de régression. Ce jeu possède 53 940 observations sur différentes caractéristiques des diamants comme les carats, la taille, la couleur, les dimensions ou le prix. Nous souhaiterions prédire le prix d'un diamant à partir de la table (largeur du sommet du diamant par rapport au point le plus large), du nombre de carats et de sa couleur.

Remarque

Un travail préalable de feature engineering a été effectué, ce qui explique pourquoi certaines variables seront directement sélectionnées ou ignorées dans notre exemple. Le but étant de se concentrer ici uniquement sur la modélisation.

3.1 Préparation des données

3.1.1 Import des données

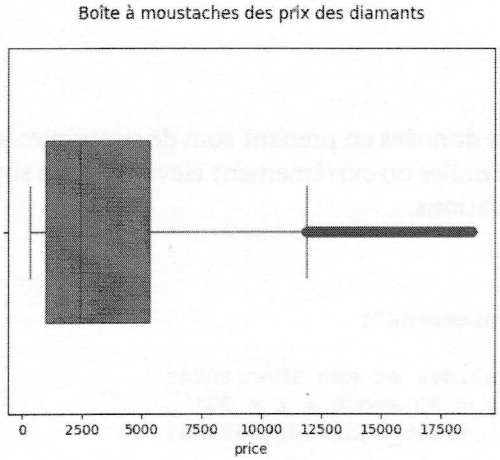
Nous commençons par importer les données en prenant soin de supprimer les observations ayant des dimensions nulles ou extrêmement élevées. Cette suppression ne concerne que 23 observations.

```
import seaborn as sns

df_brut = sns.load_dataset("diamonds")

# Conservation des mesures valides et non aberrantes
condition = 'x != 0 and 0 < y < 30 and 0 < z < 30'
df = df_brut.query(condition).reset_index(drop=True)

df.head()
```

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

Avant d'aller plus loin, regardons la distribution de la variable `price` :

```
# Boxplot de price
sns.boxplot(x=df['price'],color="#439CC8",width=0.5)
plt.title("Boîte à moustaches des prix des diamants\n")
plt.show()
```



Les prix des diamants s'échelonnent de 326 à 18 823 euros avec un prix médian à 2401 et un prix moyen d'environ 3931 euros.

3.1.2 Séparation des variables explicatives de la variable cible

La première mesure après l'importation va consister à séparer les variables explicatives de la variable à expliquer en les plaçant dans deux tables différentes X et y.

```
X = df[["table","carat","color"]] # Variables explicatives
y = df["price"]                   # Variable cible
```

3.1.3 Séparation entre données d'entraînement et de test

Une nouvelle séparation a lieu pour scinder le jeu entre les données d'entraînement et les données de test.

```
# Import de la fonction train_test_split
from sklearn.model_selection import train_test_split

# Split des données
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42
)
```

3.1.4 Les transformations des variables

Nous avons sélectionné deux variables numériques (`table` et `carat`) et une variable catégorielle nommée `color`. Nous allons ici appliquer une transformation de mise à l'échelle `RobustScaler()` sur les variables numériques `table` et `carat` et une binarisation de la variable `color` grâce à la fonction `OneHotEncoder()`. La fonction `ColumnTransformer()` est utilisée ici pour cibler les transformations sur les variables concernées :

```
# Import des fonctions nécessaires
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import RobustScaler, OneHotEncoder

# Création de l'instance de transformation
column_transformer = ColumnTransformer(transformers=[
        ('num', RobustScaler(), ["table", "carat"]),
        ('cat', OneHotEncoder(drop='first'), ["color"])
    ]
)

# Application sur les données
# fit_transform pour le jeu d'entraînement
X_train_scaled = column_transformer.fit_transform(X_train)

# Juste transform pour le jeu de test
X_test_scaled = column_transformer.transform(X_test)
```

3.1.5 Finalisation de la préparation des données

Avant de passer à l'expérimentation, nous allons maintenant mettre en œuvre deux mesures pour finaliser :

- Renommer les colonnes des variables binarisées pour obtenir leurs noms plutôt qu'un numéro d'ordre. Cela facilitera la compréhension ultérieure des résultats :

```
# Récupération des noms des variables binarisées
names_bin_columns =list(column_transformer
.named_transformers_['cat'] .get_feature_names_out(["color"]))

# Création d'une liste globale des noms de variables
all_columns = ["table", "carat"] + names_bin_columns
```

```
# Ajout des noms aux matrices
X_train_processed = pd.DataFrame(X_train_scaled,
                                 columns=all_columns)
X_test_processed = pd.DataFrame(X_test_scaled,
                                columns=all_columns)
```

Modifier le type des variables. Dans notre cas, transformer le type des variables binaires de `float64` à `int8` permet de réduire l'usage de la mémoire pour la table d'entraînement de 2,63 Mb à 0.91 Mb. Tous les calculs s'en trouvent ainsi accélérés :

```
memory_usage = X_train_processed.memory_usage(deep=True).sum()

print(f"Memory usage avant modification du type:
{memory_usage / 1024 ** 2:.2f} MB")

# Identification des colonnes commençant par 'color'
color_col = [col for col in X_train_processed.columns if
col.startswith('color')]

# Conversion des colonnes en int8
X_train_processed[color_col] =
X_train_processed[color_col].astype('int8')
X_test_processed[color_col] =
X_test_processed[color_col].astype('int8')

memory_usage = X_train_processed.memory_usage(deep=True).sum()

print(f"Memory usage après modification du type :
{memory_usage / 1024 ** 2:.2f} MB")
```

```
Memory usage avant modification du type: 2.63 MB
Memory usage après modification du type : 0.91 MB
```

Les données sont désormais préparées. Place à l'expérimentation.

3.2 Fonction de calcul et d'affichage des régressions

Avant de passer à l'évaluation des modèles, voici une fonction bien pratique permettant de tester n'importe quel algorithme avec ou sans hyperparamètres. La fonction affiche, à la fin, les meilleurs paramètres et les performances de l'algorithme. Elle récupère aussi toutes les données d'expérimentation qui pourront être exploitées ensuite pour faire un bilan des solutions testées.

```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV,
cross_val_score
from sklearn.metrics import r2_score, mean_squared_error,
mean_absolute_error

def evaluate_model(algorithm, param_grid, X_train, y_train, X_test,
y_test, search_type='grid', scoring='r2', cv=5):
    """
    Entraîne un modèle en utilisant GridSearchCV ou RandomizedSearchCV,
    prédit les valeurs de test et calcule les métriques.

    :param algorithm: L'instance de l'algorithme à utiliser.
    :param param_grid: Dictionnaire des paramètres à tester(None si vide)
    :param X_train: Les données d'entraînement.
    :param y_train: Les cibles d'entraînement.
    :param X_test: Les données de test.
    :param y_test: Les cibles de test.
    :param search_type: Le type de recherche ('grid' pour GridSearchCV ou
    'random' pour RandomizedSearchCV).
    :param scoring: Métrique pour GridSearchCV ou RandomizedSearchCV.
    :param cv: Nombre de folds pour la validation croisée.
    :return: Un dictionnaire des meilleurs paramètres, R², RMSE, MAE,
    les résultats et le modèle.
    """

    # Si le param_grid est vide, entraînement sans optimisation
    if param_grid is None:
        algorithm.fit(X_train, y_train)
        best_model = algorithm
        best_params = algorithm.get_params()
        cv_results = cross_val_score(best_model,
                                     X_train, y_train,
                                     cv=cv,
                                     scoring=scoring)
    else:
        # Choisir le type de recherche d'hyperparamètres
        if search_type == 'grid':
```

```
            search = GridSearchCV(algorithm,
                                  param_grid,
                                  cv=cv,
                                  scoring=scoring)
        elif search_type == 'random':
            search = RandomizedSearchCV(algorithm,
                                        param_grid,
                                        cv=cv,
                                        scoring=scoring)
        else:
            raise ValueError("search_type must be 'grid' or 'random'")

        # Entraînement et optimisation
        search.fit(X_train, y_train)
        best_model = search.best_estimator_
        best_params = search.best_params_
        cv_results = search.cv_results_

    # Prédiction avec le meilleur modèle
    y_pred = best_model.predict(X_test)

    # Calcul des indicateurs
    r2 = r2_score(y_test, y_pred)
    rmse = mean_squared_error(y_test, y_pred) ** 0.5
    mae = mean_absolute_error(y_test, y_pred)

    # Affichage des résultats
    print(f"Modèle : {algorithm.__class__.__name__}")
    print(f"Meilleurs paramètres : {best_params}")
    print(f"R² : {r2}")
    print(f"RMSE : {rmse}")
    print(f"MAE : {mae}")

    return {
        'best_params': best_params,
        'r2': r2,
        'rmse': rmse,
        'mae': mae,
        'cv_results': cv_results,
        'best_model' : best_model
    }
```

Cette fonction va grandement nous faciliter l'expérimentation des différents algorithmes en nous permettant de récupérer les données d'expérimentation et d'afficher les résultats nécessaires à l'évaluation.

3.3 La modélisation d'une régression

3.3.1 Modèle de base (DummyRegressor)

Nous débutons la modalisation par le calcul du modèle de base qui servira de référence pour relativiser la performance des autres solutions testées.

```
from sklearn.dummy import DummyRegressor

dummy_regr = DummyRegressor()
param_grid_dr = {
    'strategy': ["mean","median"]
}
results_dummy = evaluate_model_reg(dummy_regr, param_grid_dr,
X_train_processed, y_train, X_test_processed, y_test)
```

```
Modèle : DummyRegressor
Meilleurs paramètres : {'strategy': 'mean'}
R² : -4.514190489257608e-05
RMSE : 3970.0877794570974
MAE : 3008.402689259864
```

Les résultats précédents méritent quelques commentaires :

- La stratégie « moyenne » a été sélectionnée au profit de la stratégie « médiane » car elle fournit le meilleur score.
- Le R^2 tourne, sans surprise, autour de zéro, puisque l'algorithme ne capture aucune variation dans les données.
- La RMSE indique une erreur moyenne quadratique de 3970 euros signifiant que le système prédit le prix du diamant à 3970 euros près, ce qui est mauvais.
- La MAE est moins élevée à environ 3008 euros confirmant le fait que les valeurs extrêmes, qui font monter la RMSE à 3970, sont bien présentes et pèsent sur les prédictions.

3.3.2 Test des algorithmes concurrents

Une fois le modèle de référence évalué, nous allons tester différents algorithmes qui ont leurs propres modes de fonctionnement et paramétrages. Nous procéderons étape par étape afin de bien commenter les résultats. Toutefois, en conditions réelles, il serait recommandé d'utiliser un pipeline pour explorer tous les aspects en une seule fois. Nous verrons comment procéder à la sous-section Le pipeline de ce chapitre.

Débutons cette expérimentation par l'un des algorithmes les plus connus : la régression linéaire.

Régression linéaire

La régression linéaire est une bonne option à tester dès le départ, car elle est simple à mettre en œuvre et à interpréter. Et en tant qu'algorithme linéaire, elle constitue une base solide pour évaluer les performances et déterminer si des approches plus complexes sont nécessaires.

Grâce à notre fonction, il suffit d'importer l'algorithme souhaité et de définir ses paramètres propres et nous obtenons le même type de rapport que pour le `DummyRegressor`.

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
param_grid_lr = {
    'fit_intercept': [True, False],
    'positive': [True, False]
}

results_lr = evaluate_model_reg(lr,
                                param_grid_lr,
                                X_train_processed,
                                y_train,
                                X_test_processed,
                                y_test)
```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits
Modèle : LinearRegression
Meilleurs paramètres : {'fit_intercept': True, 'positive': False}
R² : 0.8689209091643584
RMSE : 1437.3317439376701
MAE : 960.0696361893933
```

Apportons quelques commentaires à ces résultats :

- La présence de l'ordonnée à l'origine a été retenue car elle apporte de meilleurs résultats avec que sans.
- Forcer les coefficients à être positifs n'était pas pertinent.
- Le coefficient de détermination est très bon à environ 0.87.
- La RMSE et la MAE sont significativement réduites à respectivement 1437 et 960. Cela représente des baisses respectives de 63.8 % et 68 %, ce qui est très important.

La régression linéaire permet d'évaluer l'importance des variables dans le modèle en étudiant leurs coefficients respectifs. Le code pour l'obtenir est le suivant :

```
for i in
zip(X_train_processed.columns,results_lr["best_model"].coef_):
    print(i)
```

```
('table', -223.1036208384283)
('carat', 5199.984079393048)
('color_E', -94.59087184119448)
('color_F', -95.65712767271005)
('color_G', -113.05514007545906)
('color_H', -735.0998381434866)
('color_I', -1079.361591408647)
('color_J', -1929.9683294453112)
```

Sans surprise, c'est le carat qui pèse le plus dans le prix. La table joue négativement. Comme la table, les couleurs impactent de manière négative et croissante dans l'ordre alphabétique. Nous retrouvons ainsi la logique des lettres de la couleur qui vont de D, qui est la meilleure, à J, qui est la pire.

Test d'algorithmes semi-linéaires

Un SVR avec noyau polynomial de degré 2 et 3 a été testé mais ne s'est pas révélé concluant car les performances étaient dégradées pour toutes les métriques et le temps d'entraînement dépassait les 20 minutes comparées aux quelques secondes que nécessitent les autres algorithmes. Voici le code pour cet algorithme, ainsi que les résultats :

```
from sklearn.svm import SVR


# Définition du modèle SVR
svr = SVR()

# Grille des hyperparamètres à tester
param_grid_svr = {
    'kernel': ['poly'],  # Différents noyaux à tester
    'degree' : [2,3]
}
results_svr = evaluate_model_reg(svr,
                                 param_grid_svr,
                                 X_train_processed,
                                 y_train,
                                 X_test_processed,
                                 y_test,
                                 search_type='grid')
```

```
Fitting 5 folds for each of 2 candidates, totalling 10 fits
Modèle : SVR
Meilleurs paramètres : {'degree': 2, 'kernel': 'poly'}
R² : 0.6869134216651844
RMSE : 2221.3783362553363
MAE : 1430.2664130289882
```

Les piètres performances nous incitent à écarter cette solution.

KNN Regressor

Passons maintenant aux algorithmes non linéaires en commençant par le KNN Regressor qui est une solution intéressante à explorer, car il est facilement compréhensible et offre généralement de bonnes performances. Il faut juste veiller à ne pas l'utiliser sur des données contenant beaucoup de variables car il souffre de faiblesses computationnelles dans les cas de grandes dimensions.

```
from sklearn.neighbors import KNeighborsRegressor
knnr = KNeighborsRegressor()
param_grid_knnr = {
    'n_neighbors': [x for x in range(5,10)],
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'metric': ['euclidean', 'manhattan', 'chebyshev',
'minkowski']
}

results_knnr = evaluate_model_reg(knnr,
                                  param_grid_knnr,
                                  X_train_processed,
                                  y_train,
                                  X_test_processed,
                                  y_test,
                                  search_type='random')
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Modèle : KNeighborsRegressor
Meilleurs paramètres : {'weights': 'uniform', 'n_neighbors': 9, 'metric': 'manhattan', 'algorithm': 'auto'}
R² : 0.9008747096851734
RMSE : 1249.9209378301086
MAE : 693.2768710847345
```

Remarque

Dans la mesure où `seach_type` est réglé sur `random`, la grille de recherche est une `RandomizedSearchCV`, donc les résultats peuvent être différents de ceux affichés.

À nouveau, nous constatons une amélioration globale de toutes les métriques mais la baisse de l'erreur moyenne est plus marquée sur la MAE que sur la RMSE indiquant que les grands écarts continuent de peser de manière négative.

XGBoost Regressor

Nous finirons sur un algorithme ensembliste bien connu pour son efficacité et sa robustesse sur les données complexes : le XGBoost Regressor.

```
from xgboost import XGBRegressor

# Définition du XGBRegressor
xgbr = XGBRegressor()

# Grille de paramètres pour la recherche
param_grid_xgbr = {
    'n_estimators': [100, 150, 200],
    'learning_rate': [0.01, 0.1, 0.2,0.3],
    'max_depth': [3, 4, 5, 6],
    'subsample': [0.6, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0]
}

# Évaluation du modèle avec la fonction evaluate_model_reg
results_xgbr = evaluate_model_reg(xgbr,
                                  param_grid_xgbr,
                                  X_train_processed,
                                  y_train,
                                  X_test_processed,
                                  y_test,
                                  search_type='random')

# Affichage des résultats
print(results_xgbr)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Modèle : XGBRegressor
Meilleurs paramètres : {'subsample': 1.0, 'n_estimators': 100, 'max_depth': 3, 'learning_rate': 0.2, 'colsample_bytree': 1.0}
R² : 0.907150387763977
RMSE : 1209.7075348868716
MAE : 673.263752212864
```

Les performances sont encore améliorées. `XGBoostRegressor` apparaît, dans notre expérimentation, comme le meilleur choix faisant passer la RMSE de 3970 à 1209.7 soit une baisse de 69.5 % par rapport au modèle de base. De manière encore plus marquée, la MAE passe de 3008 à 673 soit une diminution de 77.2 %.

De tous les algorithmes expérimentés ici, il est clairement la meilleure proposition par rapport aux métriques. Nous avons procédé ici à une présentation détaillée afin de bien commenter chaque algorithme. En pratique, l'usage du pipeline va faciliter toutes ces expérimentations. Nous allons voir dans la sous-section suivante comment l'exploiter.

3.3.3 Le pipeline

Le pipeline joue un rôle central dans la modélisation en aidant à automatiser, à simplifier le flux de travail et ainsi garantir a reproductibilité et l'absence de fuite de données entre les phases d'entraînement et de test.

Pour illustrer son fonctionnement, nous allons reprendre en une seule fois toutes les expérimentations d'algorithmes et mettre l'ensemble sous forme de pipeline.

Nous allons commencer par importer les modules nécessaires et définir nos algorithmes en définissant leurs nom, fonction et dictionnaire d'hyperparamètres :

```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from xgboost import XGBRegressor

from sklearn.preprocessing import StandardScaler

# Définition de nos régressors :
regressors = {
    'LinearRegression': (LinearRegression(),
     {'regressor__fit_intercept': [True, False],
      'regressor__positive': [True, False],}),
    'KNeighborsRegressor': (KNeighborsRegressor(),
     {'regressor__n_neighbors': [x for x in range(5,10)],
```

```
        'regressor__weights': ['uniform', 'distance'],
        'regressor__algorithm': ['auto','ball_tree', 'kd_tree',
'brute'],
        'regressor__metric': ['euclidean', 'manhattan',
'chebyshev','minkowski']}),
    'XGBRegressor': (XGBRegressor(),
     {'regressor__n_estimators': [100, 150, 200],
      'regressor__learning_rate': [0.01, 0.1, 0.2,0.3],
      'regressor__max_depth': [3, 4, 5, 6],
      'regressor__subsample': [0.6, 0.8, 1.0],
      'regressor__colsample_bytree': [0.6, 0.8, 1.0]})
}
```

Remarque

Attention à bien faire précéder le nom de chaque hyperparamètre par le nom que nous emploierons pour définir la classe des algorithmes testés. Ici, nous les appelons `regressor` donc tous les noms de fonctions du dictionnaire seront précédés par `regressor`. Et attention à bien mettre deux underscores successifs.

Une fois le pipeline défini, voici la façon de le mettre en œuvre :

```
# Collecte des scores
results = {}

# Boucle sur chaque algorithme avec pipeline
for name, (reg, param_grid) in regressors.items():
    pipeline = Pipeline([
        ('scaler', StandardScaler()),  # Standarscaler
        ('regressor', reg) # Puis test de l'algorithme
    ])

    # Définition et entraînement de la GridsearchCV
    grid_search = GridSearchCV(estimator=pipeline,
                               param_grid=param_grid,
                               cv=5, scoring='r2')

    grid_search.fit(X_train_processed, y_train)

    #Récupération des meilleurs paramètres
    best_pipeline = grid_search.best_estimator_

    # Calcul du R² sur l'ensemble de test
```

```
    y_pred_test = best_pipeline.predict(X_test_processed)
    r2_test = r2_score(y_test, y_pred_test)

    # Calcul de RMSE et MAE sur l'ensemble de test
    mse_test = mean_squared_error(y_test, y_pred_test)
    rmse_test = np.sqrt(mse_test)
    mae_test = mean_absolute_error(y_test, y_pred_test)

    # Incrémentation du dictionnaire de tous les résultats
    results[name] = {
        'R²_train': cross_val_score(best_pipeline,
                                    X_train_processed,
                                    y_train, cv=5,
                                    scoring='r2'),
        'R²_test': r2_test,
        'RMSE_test': rmse_test,
        'MAE_test': mae_test
    }
```

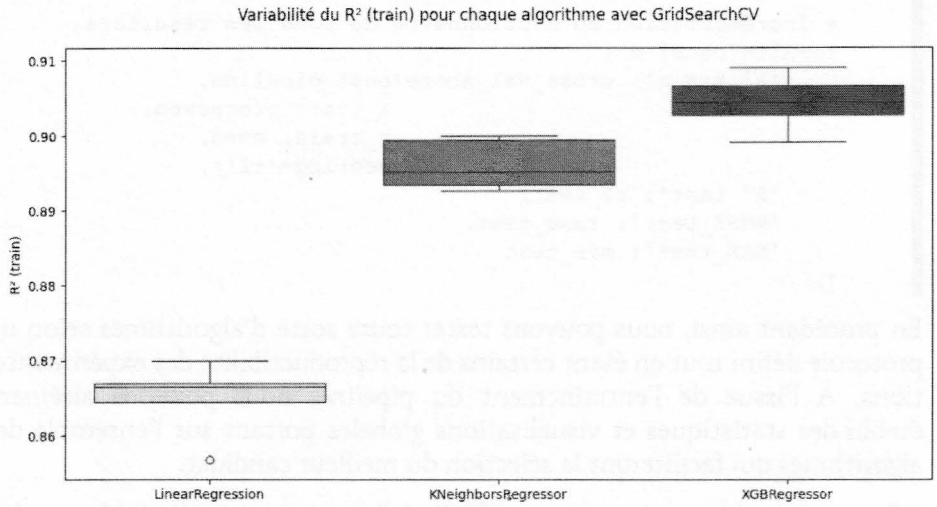
En procédant ainsi, nous pouvons tester toute sorte d'algorithmes selon un protocole défini tout en étant certains de la reproductibilité des expérimentations. À l'issue de l'entraînement du pipeline, nous pouvons aisément établir des statistiques et visualisations globales portant sur l'ensemble des algorithmes qui faciliteront la sélection du meilleur candidat.

Afin que le code ne soit pas trop difficile à lire, nous avons limité le nombre d'informations dans le dictionnaire des résultats. Mais en conditions réelles, il serait intéressant d'ajouter, par exemple, le temps d'exécution ou le poids du fichier d'algorithme ou toute sorte d'indicateurs nous aidant à faire notre choix.

À titre d'exemple, le graphique suivant indique la variabilité des scores d'entraînement pour les meilleurs paramètres de chaque algorithme :

```
colors = ['#EEE','#439CC8','#285E78']  # Exemple de couleurs
palette = dict(zip(regressors.keys(), colors))

# Visualisation des résultats
sns.boxplot(
    data=[results[name]['R²_train']
          for name in regressors.keys()],
    ax=ax,
    palette=colors
```

```
)
ax.set_xticks(range(len(regressors)))
ax.set_xticklabels(regressors.keys())
ax.set_title('Variabilité du R² (train) pour chaque algorithme
avec GridSearchCV\n')
ax.set_ylabel('R² (train)')
plt.show()
```



Dans notre cas, `XGBRegressor` confirme sa place de leader en affichant une variabilité faible tout en ayant un score plus élevé que ses concurrents.

L'emploi du pipeline ne vient pas forcément remplacer les tests successifs des algorithmes car, s'il permet de réaliser tous les tests d'un seul coup, le temps d'exécution peut devenir un handicap lors des expérimentations d'hyperparamètres.

4. La classification en pratique

Nous allons maintenant pratiquer une classification. Pour cela, nous reprendrons l'exemple des « diamonds » en essayant maintenant de déduire la qualité de la taille du diamant à partir d'autres variables.

4.1 Préparation des données

4.1.1 Import des données

Commençons par importer les données et supprimer directement les mesures inexistantes ou aberrantes :

```
import seaborn as sns
import pandas as pd
import numpy as np

# Import du jeu de données
df = sns.load_dataset("diamonds")

# Suppression des mesures inexistantes ou aberrantes
df = df_brut.query('x != 0 and 0 < y < 30 and 0 < z < 30')
.reset_index(drop=True)
```

4.1.2 Séparation entre les variables explicatives et la variable cible

Nous allons ensuite préparer les matrices X et y. Attention ici de bien passer la matrice y en type `category` :

```
# Préparation des matrices X et y
X = df[["price","table","depth","color","clarity"]]
y = pd.Series(df['cut'], dtype='category')
```

4.1.3 Séparation entre données d'entraînement et de test

La séparation entre les données d'entraînement et de test s'effectue ensuite de la manière suivante :

```
from sklearn.model_selection import train_test_split

# Division des données entre train et test
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42, stratify=y)
```

Remarque

L'usage de `stratify` dans la fonction `train_test_split` permet de maintenir les proportions des classes entre les ensembles d'entraînement et de test

4.1.4 Transformation des colonnes

Une fois les données séparées, nous allons les transformer de manière spécifique en fonction de leur type à l'aide de la fonction *ColumnTransformer*. Après avoir désigné les variables numériques et catégorielles, nous appliquerons respectivement une standardisation pour les unes et une binarisation pour les autres :

```
from sklearn.compose import ColumnTransformer

# Identification des colonnes numériques et binaires
numeric_columns = X.select_dtypes(include=['number']).columns
categorical_columns =
X.select_dtypes(exclude=['number']).columns

# Transformation des données
column_transformer  = ColumnTransformer(
    transformers=[
        ('num', RobustScaler(), numeric_columns),
        ('cat', OneHotEncoder(drop='first'),
categorical_columns)
    ]
)

# Application de ColumnTransformer
X_train_scaled = column_transformer.fit_transform(X_train)
X_test_scaled = column_transformer.transform(X_test)
```

4.1.5 Remise en forme des noms

La binarisation nécessite de renommer chaque nouvelle variable par sa mention d'origine :

```
# Création de la liste des noms incluant les noms des binaires
names_bin_columns =
list(column_transformer.named_transformers_['cat']
.get_feature_names_out(categorical_columns))
all_columns = list(numeric_columns) +  names_bin_columns
```

```
# Ajout des noms dans les tables
X_train_processed = pd.DataFrame(X_train_scaled,
columns=all_columns)
X_test_processed = pd.DataFrame(X_test_scaled,
columns=all_columns)
```

4.1.6 Ajustement du type des variables

Pour finir, ces nouvelles variables binaires vont être identifiées dans une liste et converties en `int8` dans le but de réduire leur empreinte sur la mémoire :

```
# Identification des noms des anciennes variables catégorielles
categ_cols = [
    col for col in X_train_processed.columns
    if any(col.startswith(categ_col)
    for categ_col in categorical_columns)
]

# Conversion de ces colonnes en int8 pour gagner en mémoire
X_train_processed[categ_cols] =
X_train_processed[categ_cols].astype('int8')
X_test_processed[categ_cols] =
X_test_processed[categ_cols].astype('int8')
```

4.2 Fonction de calcul et d'affichage des classifications

Nos données sont désormais prêtes pour l'expérimentation.

Comme pour la régression, voici la fonction modifiée destinée aux classifications. Celle-ci est destinée à entraîner un modèle de classification avec divers hyperparamètres, évaluer ses performances grâce à l'accuracy et afficher une classification report et une matrice de confusion pour bien mesurer les résultats de l'algorithme testé :

```
from sklearn.model_selection import GridSearchCV,
RandomizedSearchCV
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report,
confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
```

```
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder

def eval_classification(algorithm, param_grid, X_train,
                        y_train, X_test, y_test,
                        search_type='grid',
                        scoring='accuracy', cv=5):
    """
    Entraîne un modèle de classification en utilisant
    GridSearchCV ou RandomizedSearchCV,
    prédit les valeurs de test et
    calcule les métriques de performance.
    :param algorithm: L'instance de l'algorithme à utiliser.
    :param param_grid: Dictionnaire des paramètres à tester
    dans GridSearchCV ou RandomizedSearchCV. (Mettre None si
vide)
    :param X_train: Les données d'entraînement.
    :param y_train: Les cibles d'entraînement.
    :param X_test: Les données de test.
    :param y_test: Les cibles de test.
    :param search_type: Le type de recherche
    ('grid' pour GridSearchCV ou 'random' pour
RandomizedSearchCV).
    :param scoring: Métrique de scoring pour GridSearchCV ou
RandomizedSearchCV.
    :param cv: Nombre de folds pour la validation croisée.
    :return: Un dictionnaire contenant les meilleurs paramètres,
     accuracy, precision, recall, f1, confusion matrix et
classification report.
    """

    # Encodage des étiquettes
    label_encoder = LabelEncoder()
    y_train_encoded = label_encoder.fit_transform(y_train)
    y_test_encoded = label_encoder.transform(y_test)

    # Récupération des catégories originales
    unique_categories = label_encoder.classes_

    # Entraîne sans optimisation d'hyperparamètres si param_grid
est vide
    if param_grid is None:
        algorithm.fit(X_train, y_train_encoded)
        best_model = algorithm
```

```
        best_params = algorithm.get_params()
        cv_results = cross_val_score(
            best_model, X_train,
            y_train_encoded,cv=cv,
            scoring=scoring
        )
    else:
        # Choisir le type de recherche d'hyperparamètres
        if search_type == 'grid':
            search = GridSearchCV(
                algorithm, param_grid, cv=cv,
                scoring=scoring
            )
        elif search_type == 'random':
            search = RandomizedSearchCV(
                algorithm, param_grid, cv=cv,
                scoring=scoring
            )
        else:
            raise ValueError(
                "search_type doit être 'grid' ou 'random'"
            )
        # Entraînement et optimisation
        search.fit(X_train, y_train_encoded)
        best_model = search.best_estimator_
        best_params = search.best_params_
        cv_results = search.cv_results_

    # Prédiction avec le meilleur modèle
    y_pred = best_model.predict(X_test)

    # Calcul des indicateurs
    accuracy = accuracy_score(y_test_encoded, y_pred)
    class_report = classification_report(
        y_test_encoded, y_pred,
        target_names=unique_categories
    )
    conf_matrix = confusion_matrix(y_test_encoded, y_pred)

    # Affichage des résultats
    print()
    print(f"Modèle : {algorithm.__class__.__name__}")
    print(f"Meilleurs paramètres : {best_params}")
    print(f"Accuracy : {accuracy}")
```

```
    # Affichage du rapport de classification
    print("\nRapport de classification:\n")
    print(class_report)

    # Affichage de la matrice de confusion avec Seaborn
    plt.figure(figsize=(6, 4))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
                cbar=False, xticklabels=unique_categories,
                yticklabels=unique_categories)
    plt.xlabel('Etiquettes prédites')
    plt.ylabel('Etiquettes réelles')
    plt.title('Matrice de confusion')
    plt.show()

    return {
        'best_params': best_params,
        'accuracy': accuracy,
        'confusion_matrix': conf_matrix,
        'classification_report': class_report,
        'cv_results': cv_results,
        'best_model': best_model
    }
```

4.3 Expérimentations

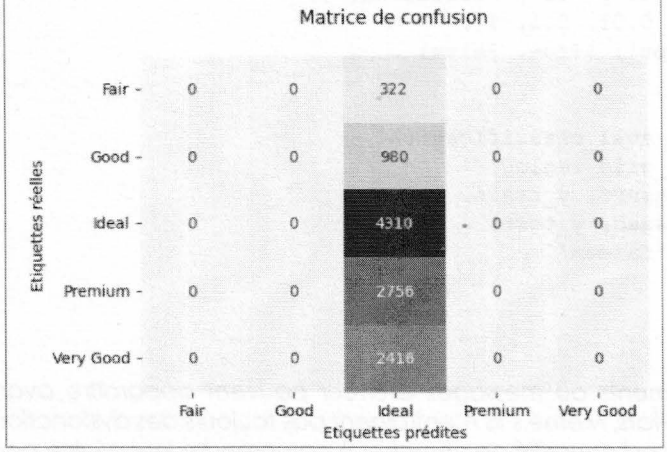
4.3.1 Modèle de base (DummyClassifier)

Le `DummyClassifier` va nous permettre de connaître le score de base.

```
from sklearn.dummy import DummyClassifier

dc = DummyClassifier()

param_grid_dc = {
    'strategy': [
        "most_frequent",
        "prior",
        "stratified","uniform"
    ]
}

results_dc = eval_classification(
```

```
    dc, param_grid_dc,
    X_train_processed, y_train,
    X_test_processed, y_test
)
```

```
Modèle : DummyClassifier
Meilleurs paramètres : {'strategy': 'most_frequent'}
Accuracy : 0.3996661721068249

Rapport de classification:
              precision    recall  f1-score   support

        Fair       0.00      0.00      0.00       322
        Good       0.00      0.00      0.00       980
       Ideal       0.40      1.00      0.57      4310
     Premium       0.00      0.00      0.00      2756
   Very Good       0.00      0.00      0.00      2416

    accuracy                           0.40     10784
   macro avg       0.08      0.20      0.11     10784
weighted avg       0.16      0.40      0.23     10784
```



L'usage du `DummyClassifier` est instructif car il nous apprend que :

- En utilisant la stratégie du plus fréquent, le modèle obtient une accuracy d'environ 0.4 qui est la fréquence de la classe « ideal », la plus présente.
- Les classes sont donc déséquilibrées. Il existe des modules, comme SMOTE, qui permettent de corriger ce problème en augmentant artificiellement les observations de la classe sous-représentée. Nous laisserons ce problème de côté pour ne pas surcharger davantage le code.

– Commenter le rapport de classification et la matrice de confusion n'a guère d'intérêt car seule la colonne « ideal » est complétée, en accord avec la stratégie retenue.

4.3.2 Algorithmes concurrents

Régression logistique

Débutons ces expérimentations par la régression logistique qui est un bon point de départ. Le choix a été fait de tester plusieurs combinaisons, donc l'option `RandomizedSearchCV` semble plus pertinente :

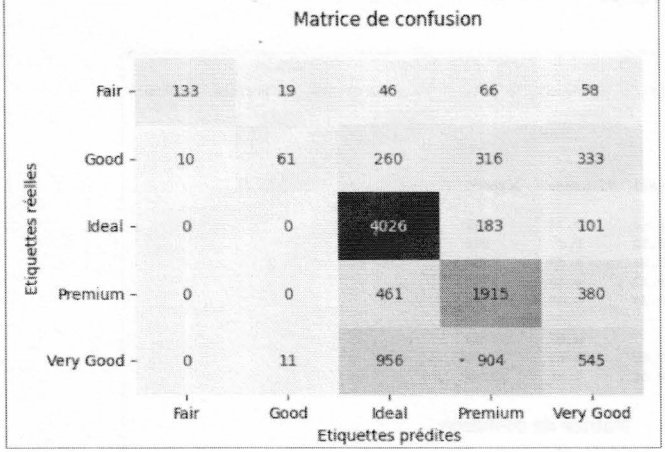
```
from sklearn.linear_model import LogisticRegression

reglog = LogisticRegression(max_iter=500,solver='saga')
param_grid_reglog = {
    'penalty': ["l1", "l2", "elasticnet"],
    'C': [0.005, 0.01, 0.1, 1],
    'fit_intercept': [True, False]
}

results_reglog = eval_classification(
    reglog,param_grid_reglog,
    X_train_processed, y_train,
    X_test_processed, y_test,
    search_type='random'
)
```

Remarque

Quelques avertissements ou messages d'erreur peuvent apparaître avant l'affichage des résultats. Même s'ils n'entraînent pas toujours des dysfonctionnements profonds, il est conseillé de chercher à comprendre leur origine pour améliorer la qualité des expérimentations.

```
Modèle : LogisticRegression
Meilleurs paramètres : {'penalty': 'l1', 'fit_intercept': True, 'C': 0.005}
Accuracy : 0.6194362017804155

Rapport de classification:
```

	precision	recall	f1-score	support
Fair	0.93	0.41	0.57	322
Good	0.67	0.06	0.11	980
Ideal	0.70	0.93	0.80	4310
Premium	0.57	0.69	0.62	2756
Very Good	0.38	0.23	0.28	2416
accuracy			0.62	10784
macro avg	0.65	0.47	0.48	10784
weighted avg	0.60	0.62	0.57	10784



La régression logistique offre des performances moyennes avec un score de quasiment 0.62. Néanmoins, nous avons un début de diagonalisation de la matrice de confusion et quelques bonnes performances sur les classes « Fair » et « Ideal » pour la précision et un très bon taux de détection global des « Ideal » à 0.93.

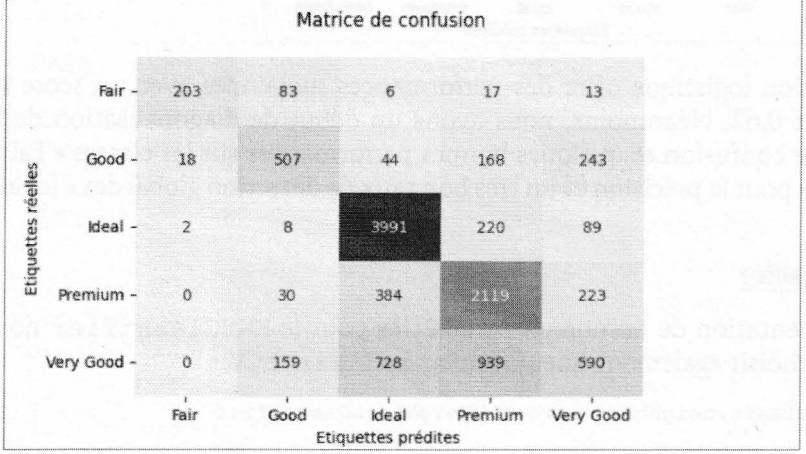
KNNClassifier

L'expérimentation de nombreux paramètres pour le `KNNClassifier` nous pousse à choisir également une `RandomizedSearchCV`.

```
from sklearn.neighbors import KNeighborsClassifier

knnc_classif = KNeighborsClassifier()
```

```
param_grid_knnc = {
    'n_neighbors': [13, 14, 15, 16],
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'p': [1, 2]
}

results_knnc = eval_classification(
    knnc_classif,
    param_grid_knnc,
    X_train_processed,
    y_train, X_test_processed,
    y_test, search_type='random'
)
```

```
Modèle : KNeighborsClassifier
Meilleurs paramètres : {'weights': 'uniform', 'p': 1, 'n_neighbors': 14, 'algorithm': 'auto'}
Accuracy : 0.6871290801186943

Rapport de classification:

              precision    recall  f1-score   support

        Fair       0.91      0.63      0.74       322
        Good       0.64      0.52      0.57       980
       Ideal       0.77      0.93      0.84      4310
     Premium       0.61      0.77      0.68      2756
   Very Good       0.51      0.24      0.33      2416

    accuracy                           0.69     10784
   macro avg       0.69      0.62      0.63     10784
weighted avg       0.67      0.69      0.66     10784
```



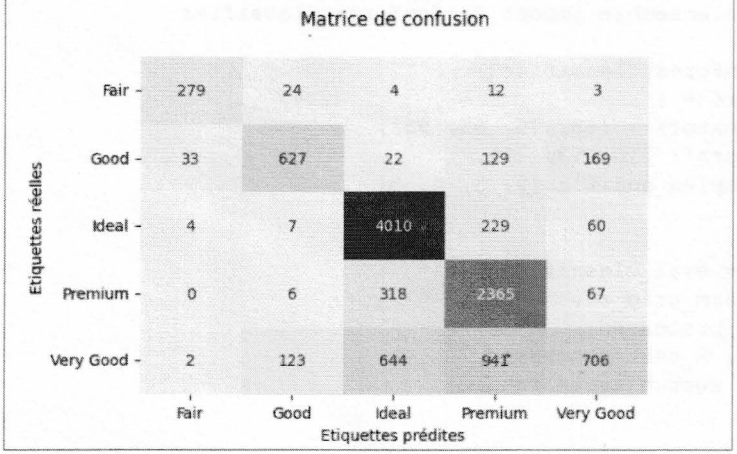
Les résultats s'améliorent encore à quasiment 0.69 et la diagonalisation de la matrice est bien plus marquée dénotant de bonnes performances sur l'ensemble des classes bien qu'il subsiste des disparités. Nous atteignons ici une bonne précision en général mais la capacité du système à retrouver la catégorie « Very Good » reste faible à 0.24.

RandomForestClassifier

Nous allons maintenant comparer les performances du modèle de base au *RandomForestClassifier*. Il nécessite environ 2 minutes de temps d'entraînement mais les résultats sont à la hauteur des attentes :

```
from sklearn.ensemble import RandomForestClassifier

rfc = RandomForestClassifier()
param_grid_rf = {
    'n_estimators': [65, 75, 85, 95],
    'max_depth': [10, 13, 15],
    'min_samples_split': [2, 3, 4, 5]
}

results_rf = eval_classification(
    rfc, param_grid_rf,
    X_train_processed,
    y_train, X_test_processed,
    y_test, search_type='random'
)
```

```
Modèle : RandomForestClassifier
Meilleurs paramètres : {'n_estimators': 85, 'min_samples_split': 5, 'max_depth': 13}
Accuracy : 0.7406342729970327

Rapport de classification:

              precision    recall  f1-score   support

        Fair       0.88      0.87      0.87       322
        Good       0.80      0.64      0.71       980
       Ideal       0.80      0.93      0.86      4310
     Premium       0.64      0.86      0.74      2756
   Very Good       0.70      0.29      0.41      2416

    accuracy                           0.74     10784
   macro avg       0.76      0.72      0.72     10784
weighted avg       0.74      0.74      0.72     10784
```



Le `RandomForestClassifier` produit ici des résultats plutôt élevés qui appellent des commentaires :

- Le score d'accuracy de 0.74 est bon, globalement la performance est au rendez-vous.
- Nous retrouvons le même problème que pour les concurrents concernant la capacité à retrouver la classe « Very Good ». Le recall reste très faible ici à 0.29.
- La précision de ses prédictions, comme pour les autres classes d'ailleurs, est plutôt élevée oscillant entre 0.64 et 0.88.

Globalement, nous parvenons à obtenir de bonnes performances grâce au Random Forest. D'autres algorithmes de la famille des méthodes ensemblistes ont été testés et les scores tournaient tous autour de 0.73-0.74. Nous retrouvons dans cette famille, une bonne capacité à gérer des systèmes complexes.

Ici, la matrice de confusion se révèle être un outil précieux pour détecter les classes problématiques, au premier rang desquelles nous retrouvons la classe « Very Good ». Quel que soit l'algorithme, cette classe affiche un faible recall traduisant une faible capacité à retrouver les diamants appartenant à cette catégorie. Ce type de constat invite à étudier minutieusement cette catégorie, en particulier, pour trouver les raisons de cette contre-performance. En l'absence d'améliorations, il pourrait s'avérer nécessaire de réévaluer la catégorisation initiale, et de considérer d'éventuelles modifications dans le processus de classification pour améliorer la détection de cette classe.

5. Conclusion

En conclusion, ce chapitre a introduit les principes fondamentaux de la modélisation supervisée en régression et en classification. Nous avons couvert les concepts essentiels tout en omettant certaines notions pour éviter une surcharge d'information. Il est important de souligner, à ce propos, que la gamme d'outils et de techniques disponibles est à la fois abondante et parfois complexe.

Pour véritablement maîtriser ces méthodes, une pratique régulière est indispensable. En approfondissant notre compréhension et en appliquant ces techniques dans divers domaines, nous pourrons affiner nos analyses, élargir notre boîte à outils, et développer des compétences plus avancées.

Le tableau ci-joint est un résumé des expérimentations abordées :

Aspect	Régression	Classification
Variable cible	Variable continue (quantitative)	Variable discrète (catégorielle)
Évaluation des performances	Erreur quadratique moyenne (MSE), Erreur absolue moyenne (MAE), R^2	Matrice de confusion, Accuracy, Précision, Rappel, F-score

Aspect	Régression	Classification
Hypothèses du modèle	Relation linéaire ou non, indépendance des observations et des variables	
Prétraitement des données	Normalisation/Standardisation des variables Détection des valeurs aberrantes Encodage des variables catégorielles Modification du type	
Choix des variables	Sélection basée sur l'importance des variables et la non colinéarité	Sélection basée sur la pertinence pour la classification
Validation croisée	Utilisation de cross-validation pour éviter le surapprentissage	Utilisation de cross-validation avec stratification
Analyse des erreurs		Analyse des erreurs de classification pour identifier des biais
Complexité du modèle	Impact de la complexité sur le surapprentissage et le sous-apprentissage	Impact des paramètres du modèle sur la performance (e.g., profondeur d'arbre, nombre de voisins)
Interprétabilité	Coefficients du modèle, Importance des variables	Importance des caractéristiques, Matrice de confusion
Optimisation des hyperparamètres	Ajustement des hyperparamètres comme la régularisation	Optimisation des hyperparamètres comme le nombre de voisins, la profondeur des arbres

Pour finir, voici quelques pistes supplémentaires à explorer, non abordées dans les exemples, qui nous orienterons vers des actions supplémentaires à entreprendre pour affiner encore davantage nos modélisations :

Aspect	Régression	Classification
Prétraitement des données		Gestion des classes déséquilibrées : oversampling (SMOTE) ou undersampling
Gestion des valeurs manquantes	Imputation (moyenne, médiane, interpolation)	
Analyse des résidus	Vérification de l'homoscédasticité, de la normalité des résidus et tests de colinéarité	
Détection des anomalies	Analyse des résidus pour détecter les points influents	Analyse des erreurs pour identifier les points mal classés

Chapitre 8
L'apprentissage non supervisé

1. Introduction

L'apprentissage non supervisé est une méthode de Machine Learning visant à découvrir des motifs, des groupes ou des structures dans des données non étiquetées. Contrairement à l'apprentissage supervisé, où la présence de données étiquetées permet d'évaluer un algorithme au moyen d'un score de performance, l'approche non supervisée est plus subtile et nécessite une analyse approfondie des résultats obtenus. Ces résultats, livrés sans mode d'emploi, peuvent révéler des informations cachées dans les données, telles que des regroupements naturels ou des caractéristiques saillantes.

Dans ce chapitre, nous allons mettre en pratique la réduction dimensionnelle puis le clustering. Nous découvrirons également, pour chacun, comment les évaluer et quels sont les différents algorithmes disponibles.

2. La réduction dimensionnelle

L'analyse en composantes principales (ACP) est un algorithme fondamental en data science qui permet de réduire la dimensionnalité des données tout en conservant autant que possible leur variance. Cette technique peut être utilisée à deux moments spécifiques d'un projet de data science :

- **À la fin du feature engineering** : pour comprendre et visualiser les relations multidimensionnelles entre les variables.
- **Parfois au moment de la modélisation** : soit pour lisser les données, soit pour permettre à certains algorithmes de fonctionner plus efficacement lorsque les besoins en mémoire sont trop importants.

Nous allons donc étudier comment le mettre en pratique dans ces deux cas de figure.

2.1 L'ACP en pratique pour analyser

L'Analyse en Composantes Principales (ACP) est un outil essentiel de l'analyse multidimensionnelle. Elle permet de révéler les relations complexes et cachées entre les variables, tout en identifiant leur influence dans un jeu de données. De plus, l'ACP isole les observations influentes, offrant ainsi une vision plus précise des données.

Ces informations sont accessibles à travers trois graphiques que nous allons apprendre à construire successivement sur le jeu de données du Titanic qui présente des informations variées et soulignera bien les forces de l'ACP.

2.1.1 Préparation des données

La préparation des données est nécessaire avant la mise en œuvre d'une ACP. Pour que l'ACP fonctionne correctement, il est nécessaire de :

- Transformer les variables catégorielles en binaires si certaines ont été sélectionnées. Le recours à des fonctions comme *get_dummies* de *Pandas* permet de réaliser ces transformations sur les variables ciblées dans l'option *columns*.

- Définir une stratégie pour supprimer les valeurs manquantes : nous imputons par la médiane pour simplifier, bien que 20 % des âges (177) manquent et nécessiteraient une analyse plus approfondie.
- Standardiser les données : l'utilisation de `StandardScaler` est recommandée, mais `RobustScaler` (si présence d'outliers) ou `MinMaxScaler` (pour mettre le même poids à toutes les variables) peuvent être utilisés selon les cas.

Nous allons, pour illustrer notre exemple, réaliser une petite étude sociologique des passagers du Titanic. Pour ce faire, nous utiliserons les variables `age`, `survived` (survie ou non), `alone` (voyage seul ou non), `fare` (le prix du billet), `pclass` (reflétant le statut socio-économique du passager, avec 1 étant la classe la plus élevée et 3 la plus basse) et la variable catégorielle `sex`.

```
import numpy as np
import pandas as pd

import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Import des données
datadeb = sns.load_dataset("titanic")


# Imputation des âges manquants par la médiane
datadeb['age'].fillna(datadeb['age'].median(), inplace=True)

# Binarisation de la variable sex
datadeb = pd.get_dummies(datadeb,
                         columns=["sex"],
                         drop_first=True)

# Sélection des variables d'intérêt après transformation
features = ["age", "alone", "survived", "fare", "pclass", "sex_male"]

# Création du DataFrame final avec les variables sélectionnées
data = datadeb[features].copy()


# Standardisation des variables numériques
num_features= ["age","survived","fare","pclass"]
```

```
scaler = StandardScaler()
data[num_features] = scaler.fit_transform(data[num_features])

data.head()
```

	age	alone	survived	fare	pclass	sex_male
0	-0.565736	False	-0.789272	-0.502445	0.827377	True
1	0.663861	False	1.266990	0.786845	-1.566107	False
2	-0.258337	True	1.266990	-0.488854	0.827377	False
3	0.433312	False	1.266990	0.420730	-1.566107	False
4	0.433312	True	-0.789272	-0.486337	0.827377	True

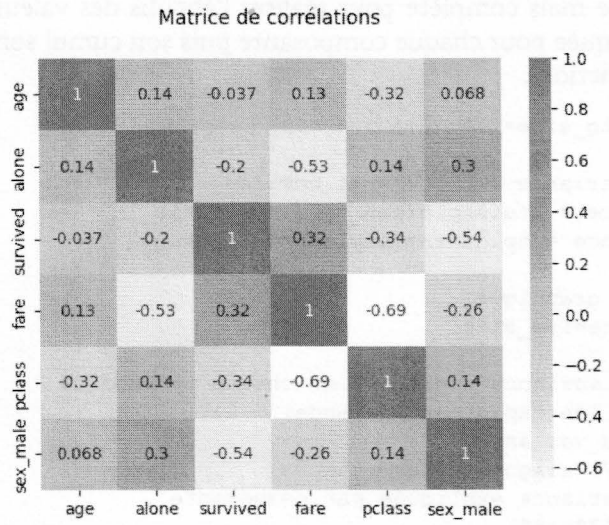
Nous avons sélectionné ici un `StandardScaler` qui harmonise les variables numériques en leur donnant une moyenne de 0 et un écart-type de 1, affectant ainsi leur importance relative. Pour illustrer l'impact de ces transformations, voici un tableau comparatif des variances de chaque variable en fonction du type de transformation appliqué aux données :

Variance des variables en fonction de la transformation

	StandardScaler	RobustScaler	MinMaxScaler	Original
age	1.001124	1.003033	0.026767	169.512498
survived	1.001124	0.236772	0.236772	0.236772
fare	1.001124	4.631963	0.009408	2469.436846
pclass	1.001124	0.699015	0.174754	0.699015
sex_male	0.228475	0.228475	0.228475	0.228475
alone	0.239723	0.239723	0.239723	0.239723

Désormais, nos données, hormis les variables binaires, sont standardisées. Nous allons réaliser une matrice de corrélation (de Spearman pour nous émanciper des problèmes de distribution) pour vérifier l'absence de corrélations trop fortes :

```
plt.title("Matrice de corrélations\n")
sns.heatmap(
    data.corr(method='spearman'),
    annot=True, cmap='Blues'
)
plt.show()
```



En l'absence de corrélations trop fortes, l'ACP peut débuter.

2.1.2 L'éboulis des valeurs propres

La première étape de l'ACP consiste à mettre en place l'éboulis des valeurs propres. Ce graphique va nous indiquer la part de variance expliquée par chaque composante, c'est-à-dire la part d'information des données initiales que cette composante capture, ainsi que le cumul de ces variances. Nous pourrons ainsi savoir si notre jeu peut être correctement résumé et combien de composantes sont nécessaires.

Commençons par instancier l'ACP :

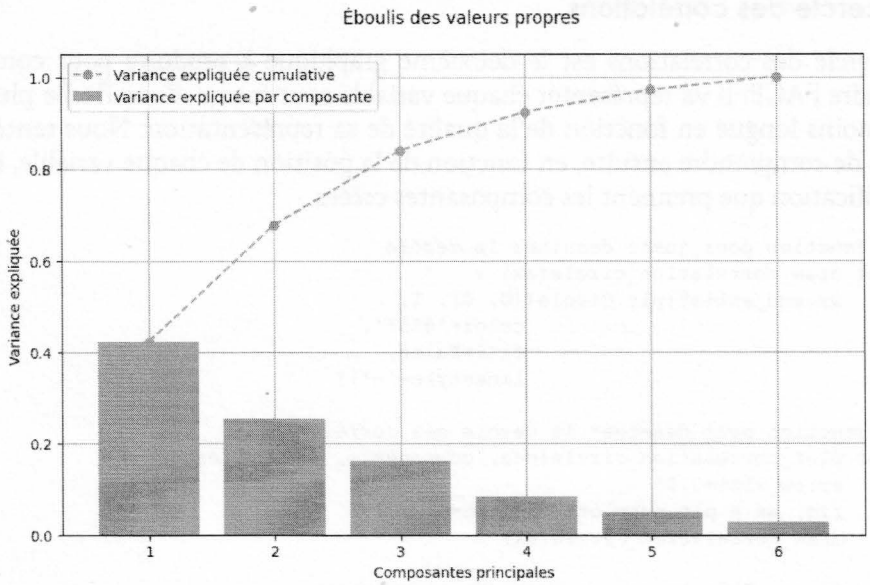
```
# Initialisation de l'ACP
pca = PCA()
pca.fit(data)
```

Remarque

Au début, il ne faut jamais fixer le nombre de composantes de l'ACP car l'éboulis des valeurs propres pourrait être incomplet et ne pas refléter toute l'information disponible.

Voici une fonction simple mais complète pour réaliser l'éboulis des valeurs propres. La variance expliquée pour chaque composante puis son cumul sont calculés au début de la fonction :

```
def scree_plot(pca,fig_size=(10, 6)) :

    # Calcul de la variance expliquée et cumulée
    explained_variance = pca.explained_variance_ratio_
    cumulative_variance = np.cumsum(explained_variance)

    # Déclaration du graphique
    plt.figure(figsize=fig_size)

    # Barres pour la variance expliquée de chaque composante
    plt.bar(range(1, len(explained_variance) + 1),
            explained_variance,
            alpha=0.7, align='center',
            label='Variance expliquée par composante',
            color="#439cc8")

    # Courbe de la variance expliquée cumulative
    plt.plot(range(1, len(cumulative_variance) + 1),
             cumulative_variance,
             marker='o',
             linestyle='--',
             color='darkorange',
             label='Variance expliquée cumulative')

    plt.xlabel('Composantes principales')
    plt.ylabel('Variance expliquée')
    plt.title('Éboulis des valeurs propres\n')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()

# Lancement de la fonction
scree_plot(pca)
```



L'inertie est la somme des variances de toutes les variables. Dans notre graphique, chaque barre représente la part expliquée de cette inertie par chaque composante. Les composantes sont les nouvelles variables créées par l'ACP et portent ce nom car elles sont composées des anciennes variables.

La courbe cumulative représente le cumul de la part expliquée. Il nous permet de connaître le nombre de composantes à considérer pour pouvoir résumer au mieux nos données. Les seuils de 80 %, 90 % et 95 % sont souvent considérés représentant, dans notre cas, respectivement 3, 4 et 5 variables.

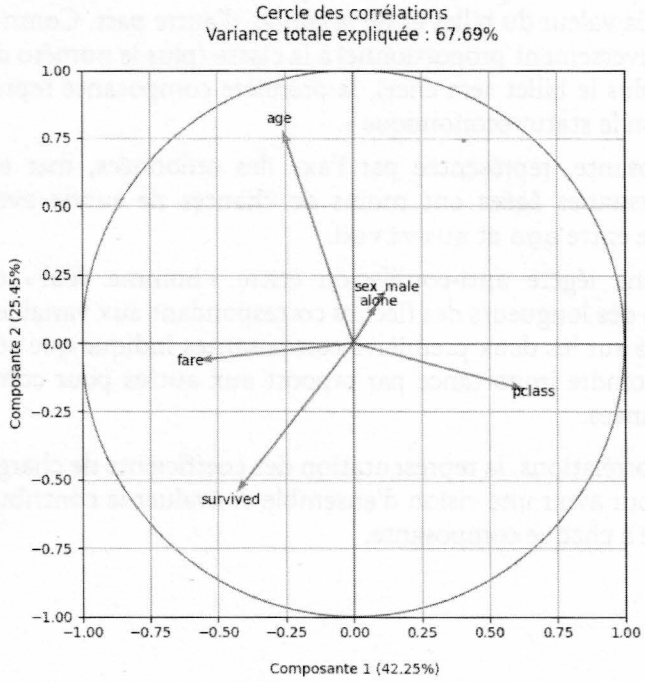
Dans la pratique, ce sont souvent uniquement les deux premières composantes qui sont utilisées pour représenter les données et se faire une idée de l'organisation des variables. Nous sommes donc très souvent en dessous des 80 % (67,69 % dans notre cas).

2.1.3 Le cercle des corrélations

Le cercle des corrélations est le deuxième graphique à produire pour comprendre l'ACP. Il va représenter chaque variable sous forme d'une flèche plus ou moins longue en fonction de la qualité de sa représentation. Nous tenterons de comprendre ensuite, en fonction de la position de chaque variable, la signification que prennent les composantes créées.

```
# Fonction pour juste dessiner le cercle
def draw_correlation_circle(ax) :
    ax.add_artist(plt.Circle((0, 0), 1,
                             color='#555',
                             fill=False,
                             linestyle='-'))

# Fonction pour dessiner le cercle des corrélations
def plot_correlation_circle(pca, components, feature_names) :
    arrow_size=0.05
    fig, ax = plt.subplots(figsize=(7, 7))
    draw_correlation_circle(ax)

    for i, feature in enumerate(feature_names):
        x, y = pca.components_[components, i]
        norm = np.linalg.norm([x, y])

        if norm != 0:
            x_base = x - x / norm * arrow_size
            y_base = y - y / norm * arrow_size
            ax.plot(
                [0, x_base],
                [0, y_base],
                color='#439cc8'
            )
            ax.quiver(
                x_base,
                y_base,
                x / norm * arrow_size,
                y / norm * arrow_size,
                angles='xy',
                scale_units='xy',
                scale=1,
                color='#439cc8'
            )

        ax.text(x * 1.05, y * 1.05, feature,
                color='black', ha='center', va='center')
```

```
    var_ratio = pca.explained_variance_ratio_[components] * 100

    x_label = f'\nComp. {components[0] + 1} ({var_ratio[0]:.2f}%)'
    y_label = f'Comp. {components[1] + 1} ({var_ratio[1]:.2f}%)'
    title = (f'Cercle des corrélations\n'
             f'Variance expliquée : {var_ratio.sum():.2f}%\n')

    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    ax.set_title(title)
    ax.grid()
    ax.axhline(0, color='#555',  linewidth=1)
    ax.axvline(0, color='#555',  linewidth=1)
    ax.set_xlim([-1, 1])
    ax.set_ylim([-1, 1])
    ax.set_aspect('equal', adjustable='box')
    plt.show()

# Exemple sur composante 1 et 2 correspondant à [0,1]
plot_correlation_circle(pca, [0,1], data.columns)
```



Décrivons succinctement ce graphique avant de le commenter :

- Le graphique concerne les composantes 1 et 2 qui sont les plus importantes et représentent 67,69 % de l'inertie globale. Nous pourrions demander la visualisation d'autres composantes afin de parfaire nos connaissances.
- Le graphique indique, pour chaque axe, le pourcentage d'inertie expliquée : 42,25 % pour la première composante et 25,45 % pour la seconde.
- Les graduations des axes sont les corrélations. En projetant orthogonalement la pointe de la flèche de chaque variable sur l'axe étudié, nous obtenons la corrélation de chaque variable relative.

Maintenant, nous pouvons faire un bilan des informations fournies par ce graphique, en gardant à l'esprit que chaque composante est la somme pondérée des anciennes variables :

- En projetant orthogonalement les flèches sur l'axe des abscisses, nous découvrons, en considérant uniquement les projections supérieures à |0.3|, que la première composante représente une opposition entre la classe du billet, d'une part, et la valeur du billet et de la survie, d'autre part. Comme le prix du billet est inversement proportionnel à la classe (plus le numéro de la classe est faible, plus le billet sera cher), la première composante représente « La survie selon le statut économique ».
- La deuxième composante, représentée par l'axe des ordonnées, met en évidence que les personnes âgées ont moins de chances de survie avec l'opposition marquée entre `age` et `survived`.
- Même s'il existe une légère anti-corrélation entre « homme seul » et « survie », la faiblesse des longueurs des flèches correspondant aux variables `sex_male` et `alone` sur les deux premières composantes indique que ces variables ont une moindre importance par rapport aux autres pour comprendre le jeu de données.

En plus du cercle des corrélations, la représentation des coefficients de charge peut être opportune pour avoir une vision d'ensemble et évaluer la contribution de chaque variable à chaque composante.

```
# Calcul des coefficients de charge
loadings = pca.components_.T * np.sqrt(pca.explained_variance_)

# Mise sous forme de DataFrame
loadings_df = pd.DataFrame(
    loadings,
    columns=[
        f'Composante {i+1}' for i in range(pca.n_components_)
    ],
    index=data.columns
)

# Affichage avec mise en forme
display(loadings_df.style.background_gradient(cmap='Blues'))
```

	Composante 1	Composante 2	Composante 3	Composante 4	Composante 5	Composante 6
age	-0.355758	0.832282	-0.358260	0.227076	-0.039692	0.019434
alone	0.122902	0.155803	-0.084977	-0.057873	0.407895	-0.152954
survived	-0.594104	-0.583556	-0.531786	0.091599	0.092975	0.088289
fare	-0.783324	-0.067701	0.532375	0.302759	0.088629	-0.001710
pclass	0.868627	-0.180687	-0.044663	0.457959	0.036606	0.030013
sex_male	0.163990	0.208348	0.139469	-0.111724	0.174488	0.309505

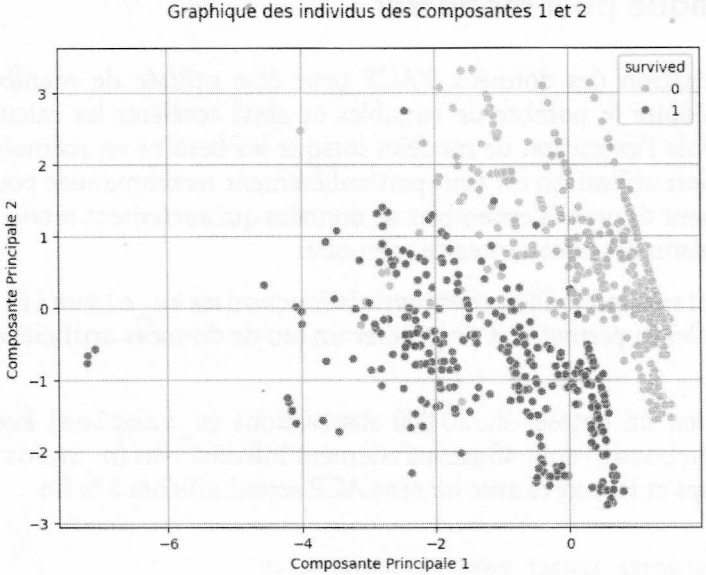
En étudiant ces deux supports, nous allons parfaire nos connaissances des variables et découvrir leur fonctionnement caché.

Mais pour comprendre totalement les données et exploiter au mieux la réduction dimensionnelle, nous allons avoir recours à un dernier graphique indispensable : le graphique des individus.

2.1.4 Le graphique des individus

Le graphique des individus place les observations dans le même système de coordonnées que les variables, permettant de visualiser leurs relations en deux ou trois dimensions. Cette projection qui révèle des groupes, des proximités, et des outliers, offre une meilleure compréhension des liens entre les observations tout en clarifiant le cercle des corrélations.

```
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

def plot_individuals(data, pca, components=(0, 1),
hue_variable=None):
  """Trace le graphique des individus en utilisant
  les composantes principales spécifiées."""

  pca_df = pd.DataFrame(
      pca.transform(data),
      columns=[f'PC{i+1}'
               for i in range(data.shape[1])]
  )

  # datadeb est le nom du fichier d'origine
  if hue_variable:
    pca_df[hue_variable] = datadeb[hue_variable]

  plt.figure(figsize=(8, 6))
  sns.scatterplot(x=f'PC{components[0] + 1}',
                  y=f'PC{components[1] + 1}',
                  hue=hue_variable, data=pca_df,
                  palette="Blues")

  plt.title(f"Graphique des individus \
  (composantes {components[0] + 1} et {components[1] + 1})\n")


  plt.xlabel(f'Composante Principale {components[0] + 1}')
  plt.ylabel(f'Composante Principale {components[1] + 1}')
  plt.legend(title=hue_variable) if hue_variable else None
  plt.grid(True)
  plt.show()

plot_individuals(
    data, pca, components=(0, 1), hue_variable="survived"
)
```



La fonction précédente génère le graphique des individus. Pour illustrer ses fonctionnalités, les observations sont colorées en fonction de la variable `survived`. La tendance de l'ancienne variable `survived` sur le cercle des corrélations est bien cohérente et souligne que ce sont bien les gens plus jeunes et/ou riches qui ont davantage survécu. Une ligne de démarcation très claire apparaît ici entre ces deux segments de populations.

En explorant ces différents graphiques et en ajustant les options pour les différentes composantes, nous pouvons obtenir une compréhension subtile et approfondie de nos jeux de données. L'ACP démontre ainsi son rôle crucial dans l'analyse multidimensionnelle.

Sa capacité à réduire la dimensionnalité tout en préservant les informations principales va pouvoir être mise à profit pour les modélisations. Nous allons justement étudier dans la section suivante comment exploiter cette fonctionnalité.

2.2 L'ACP en pratique pour modéliser

Lors de la modélisation des données, l'ACP peut être utilisée de manière judicieuse pour réduire le nombre de variables et ainsi accélérer les calculs voire rendre possible l'exécution de modèles lorsque les besoins en mémoire sont trop élevés. Son utilisation est tout particulièrement recommandée pour faciliter le traitement de grands ensembles de données qui autrement seraient inaccessibles en raison des contraintes de mémoire.

Dans l'exemple qui suit, nous allons découvrir la fonction `make_classification` de Scikit-learn permettant de générer un jeu de données artificielles paramétrable.

Il sera demandé, ici, un dataset de 20 000 observations (`n_samples`) avec 50 variables (`n_features`) dont 48 seront vraiment informatives (`n_informative`). Les temps et les scores avec ou sans ACP seront affichés à la fin.

```
import time
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Création d'un jeu de données volumineux
X, y = make_classification(n_samples=20000,
                           n_features=50,
                           n_informative=48,
                           random_state=42)

# Split des données
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
# Standardisation des données
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Entraînement et évaluation sans ACP
clf = RandomForestClassifier(random_state=42)
start_time = time.time()
```

```
clf.fit(X_train_scaled, y_train)

score_wo_acp = accuracy_score(y_test, clf.predict(X_test_scaled))

print(f"Score sans ACP : {score_wo_acp:.2f}")

time_wo_acp = time.time() - start_time
print(f"Temps d'exécution sans ACP : {time_wo_acp:.4f} secondes")


# Application d'une ACP
pca = PCA(n_components=0.99) #99 % de variance originale restituée
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

# Entraînement et évaluation avec ACP
clf_pca = RandomForestClassifier(random_state=42)
start_time = time.time()
clf_pca.fit(X_train_pca, y_train)

score_acp = accuracy_score(y_test, clf_pca.predict(X_test_pca))
print(f"Score avec ACP : {score_acp:.2f}")

time_acp = time.time() - start_time
print(f"Temps d'exécution avec ACP : {time_acp:.4f} secondes")

print(f"Nombre de composantes principales : {pca.n_components_}")
```

```
Score sans ACP : 0.93
Temps d'exécution sans ACP : 14.0525 secondes
Score avec ACP : 0.94
Temps d'exécution avec ACP : 12.6216 secondes
Nombre de composantes principales : 47
```

Remarque

Attention, les temps d'exécution peuvent varier considérablement en fonction du processeur utilisé.

En exécutant le code sur Google Colaboratory, l'application de l'ACP a réduit le temps d'exécution de 14,05 secondes à 12,62 secondes tout en améliorant le score. Cette amélioration est probablement la conséquence de la réduction du bruit dans les données après application de l'ACP.

2.3 Les autres algorithmes de réduction dimensionnelle

Lorsque les données présentent des relations non linéaires complexes, l'ACP peut se révéler inefficace. Différentes alternatives sont alors disponibles selon les situations mais notons qu'elles ont toutes pour but de préserver les distances là où l'ACP chercher à préserver la variance.

Avant de présenter ces solutions, il est important de préciser que la notion de « global » fait référence aux tendances et relations générales dans l'ensemble des données, tandis que la notion de « local » concerne les relations et structures spécifiques observées dans des sous-ensembles de données. Cette distinction est cruciale pour comprendre les approches de réduction dimensionnelle qui suivront.

Le tableau suivant présente les cas d'utilisation et désavantages de quatre algorithmes de réduction dimensionnelle :

Algorithme	Cas d'utilisation	Inconvénient
ACP	Préserver la variance globale donc la structure globale des données. Besoin de stabilité et de rapidité pour des données de grandes dimensions.	Ne capture pas les structures non linéaires.
t-SNE	Préserver la structure locale pour visualiser des clusters.	Instabilité due à l'initialisation aléatoire Peut déformer les distances globales. Long temps de calcul pour grandes données.
MDS	Préserver les distances globales.	Sensible au bruit. Limité pour des structures non linéaires complexes.
Isomap	Préserver les structures locales et globales. Capturer des structures non linéaires	Sensible aux choix des paramètres. Long temps de calcul pour grandes données.

Remarque

Il existe d'autres algorithmes tels que le Kernel PCA, UMAP ou LLE mais ils ont été volontairement mis de côté car ils sont plus complexes à mettre en œuvre ou à analyser.

Afin d'évaluer ces algorithmes, nous allons les mettre en œuvre et mesurer leurs performances sur le jeu des iris. Nous choisirons deux types de mesures :

- La capacité de chaque algorithme à reproduire les distances entre les données originales et les données réduites en utilisant le coefficient de Spearman, ce qui permet d'évaluer l'efficacité de la réduction dimensionnelle en termes de préservation des relations entre les points.
- La capacité de chaque algorithme à préserver les clusters présents dans les données en utilisant le coefficient de silhouette qui indique dans quelle mesure les points de données au sein de chaque cluster sont similaires entre eux et distincts des autres clusters.

Commençons par importer les modules et acquérir les données :

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.manifold import TSNE, MDS, Isomap
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris
from sklearn.metrics import pairwise_distances, silhouette_score
from scipy.stats import spearmanr

# Import des données iris
iris = load_iris()
data = iris.data
target = iris.target
target_names = iris.target_names
```

Préparons ensuite les différents algorithmes à tester :

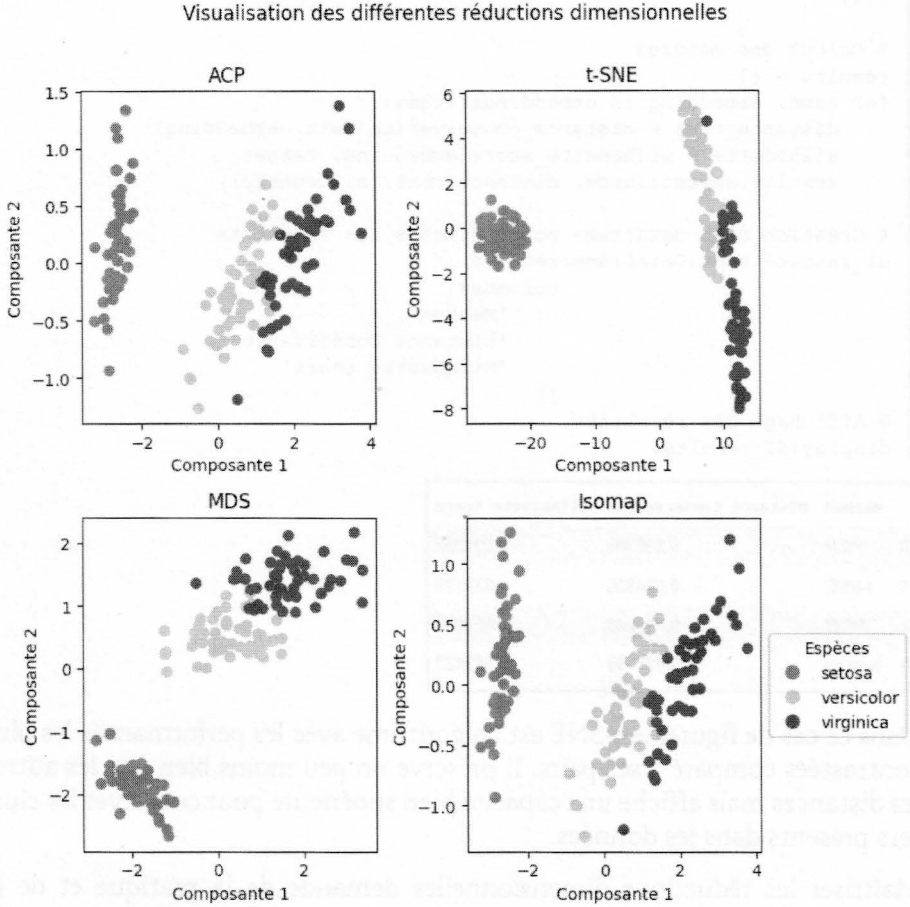
```
# Dictionnaire des techniques de réduction dimensionnelle
methods = {
    'ACP': PCA(n_components=2),
    't-SNE': TSNE(n_components=2, random_state=42),
    'MDS': MDS(n_components=2, random_state=42,
               normalized_stress='auto'),
    'Isomap': Isomap(n_components=2,n_neighbors=100)
}

# Application des méthodes
embeddings = {
    name: model.fit_transform(data)
    for name, model in methods.items()
}
```

Il est intéressant de commencer par visualiser la projection des données en deux dimensions. Nous les colorierons par espèce.

```
# Définition de la colormap personnalisée
custom_colors = ['#439CC8', 'lightgrey', '#285E78']
color_map = np.array([custom_colors[i] for i in target])
```

```
# Visualisation des résultats
plt.figure(figsize=(8, 8))
plt.suptitle(
    'Visualisation des différentes réductions dimensionnelles\n'
)

for i, (name, embedding) in enumerate(embeddings.items(), 1):
    plt.subplot(2, 2, i)
    plt.scatter(embedding[:, 0], embedding[:, 1], c=color_map)
    plt.title(name)
    plt.xlabel('Composante 1')
    plt.ylabel('Composante 2')

# Ajout de la légende
handles = [plt.Line2D([0], [0], marker='o', color='w',
label=species,
                      markerfacecolor=custom_colors[j],
markersize=10)
           for j, species in enumerate(target_names)]

plt.legend(title='Espèces',
```

```
            handles=handles,
            loc='center left',
            bbox_to_anchor=(1, 0.5))

plt.tight_layout()
plt.show()
```



En fonction des algorithmes, les projections divergent. Il sera donc pertinent d'étudier les deux mesures évoquées pour avoir une meilleure idée de la capacité de chaque algorithme à réduire correctement les données.

```
# Fonction pour calculer la conservation des distances
def distance_conservation(original_data, reduced_data):
    original_distances = pairwise_distances(original_data)
    reduced_distances = pairwise_distances(reduced_data)
    return spearmanr(
        original_distances.flatten(),
        reduced_distances.flatten()
)[0]

# Calcul des mesures
results = []
for name, embedding in embeddings.items():
    distance_cons = distance_conservation(data, embedding)
    silhouette = silhouette_score(embedding, target)
    results.append([name, distance_cons, silhouette])

# Création d'un dataframe pour afficher les résultats
df_results = pd.DataFrame(results,
                          columns=[
                              'Method',
                              'Distance Conservation',
                              'Silhouette Score'
                          ])
# Affichage des résultats
display(df_results)
```

	Method	Distance Conservation	Silhouette Score
0	ACP	0.995880	0.534393
1	t-SNE	0.964939	0.632176
2	MDS	0.995248	0.526687
3	Isomap	0.995893	0.539123

Dans ce cas de figure, le t-SNE est l'algorithme avec les performances les plus contrastées comparé à ses pairs. Il préserve un peu moins bien que les autres les distances mais affiche une capacité bien supérieure pour conserver les clusters présents dans les données.

Maîtriser les réductions dimensionnelles demande de la pratique et de la réflexion. C'est ainsi que nous réussirons à simplifier notre système de variables tout en préservant les informations essentielles qu'elles contiennent.

Nous allons maintenant pratiquer une autre forme de simplification mais sur les observations à travers le clustering.

3. Le clustering

3.1 La pratique du clustering avec le K-means

Le K-means est le plus célèbre des algorithmes de clustering. Son but est de regrouper les observations proches en clusters. Il est apprécié pour sa simplicité d'explication et de paramétrage. L'une de ses principales caractéristiques est la possibilité de définir le nombre de clusters souhaités, ce qui n'est pas toujours le cas de ses concurrents.

Cependant, déterminer le nombre optimal de clusters ne va pas de soi et constitue un enjeu. Plusieurs méthodes et tests, comme la méthode du coude (*elbow method*) ou le coefficient de silhouette, sont disponibles pour nous aider à trouver le partitionnement idéal des données.

Commençons, sans plus tarder, la pratique du clustering sur le jeu des pingouins qui se prête bien à une démonstration sur le clustering.

3.1.1 Acquisition et préparation des données

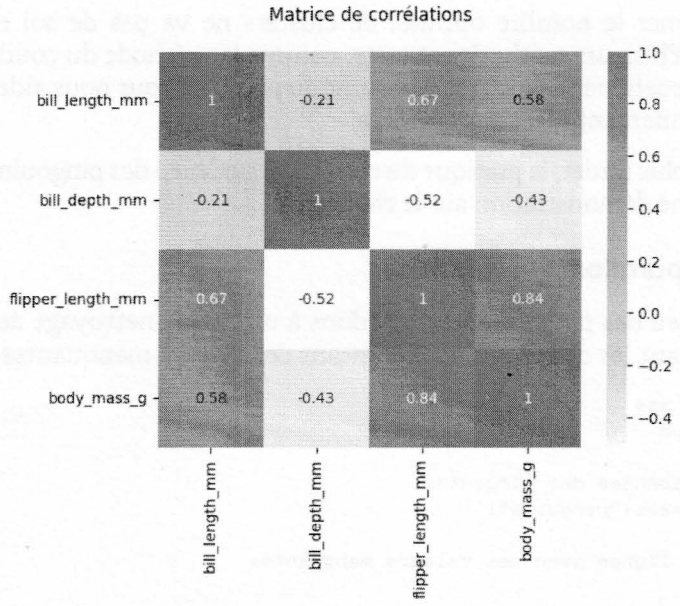
Nous importons le jeu des pingouins et procédons à un rapide nettoyage des données en supprimant les observations présentant des valeurs manquantes :

```
import seaborn as sns
import pandas as pd

# Chargement des données des pingouins
df = sns.load_dataset("penguins")

# Suppression des lignes avec des valeurs manquantes
df = df.dropna()

display(df.head())
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Female
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	Female
5	Adelie	Torgersen	39.3	20.6	190.0	3650.0	Male

Nous construirons notre clustering uniquement sur les variables numériques pour ne pas rendre cet exemple trop complexe mais, dans d'autres cas, l'usage des variables catégorielles est encouragé si cela s'avère nécessaire.

Dans le cas présent, les quatre variables correspondant à la longueur du bec, sa profondeur, la longueur de la nageoire et la masse corporelle semblent suffisantes pour opérer des regroupements. Après nous être préalablement assurés qu'ils avaient tous une distribution anormale, nous produisons une matrice de corrélation de Spearman pour contrôler qu'il n'existe pas de corrélation trop forte :



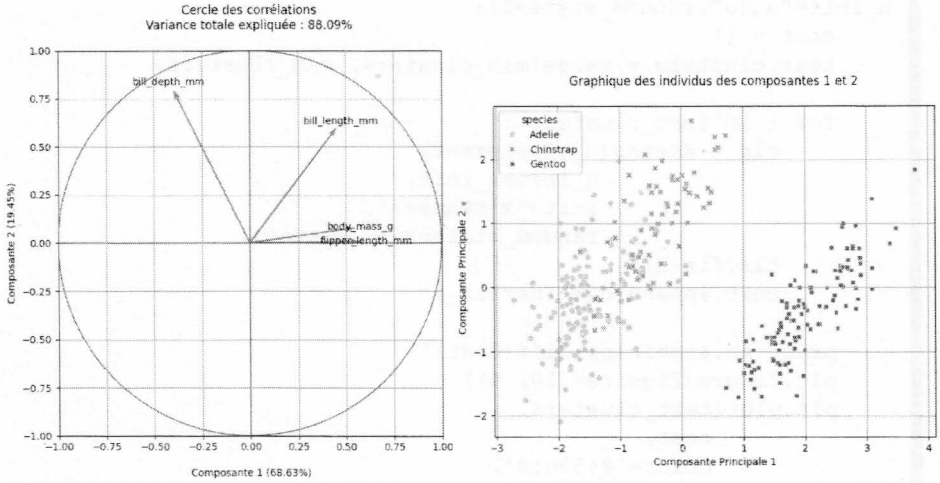
La longueur de la nageoire et la masse corporelle sont fortement corrélées (0,84) mais ne sont pas identiques et ont des natures très différentes. Par conséquent, toutes les variables seront conservées.

Nous pouvons procéder à la sélection définitive des caractéristiques qui seront standardisées afin de garantir une échelle homogène pour toutes.

```
# Sélection des caractéristiques pour le clustering
features = [
    'bill_length_mm',
```

```
    'bill_depth_mm',
    'flipper_length_mm',
    'body_mass_g'
]

X = df[features].values

# Standardisation des caractéristiques
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Récupération des étiquettes réelles des espèces
y = df["species"].astype('category').cat.codes
```

Avant de passer au clustering, avoir recours à une l'ACP est souvent pertinent, notamment pour déterminer l'importance des variables, leurs articulations et la présence d'outliers.



Avec un niveau d'inertie expliquée de 88,09 % sur les deux premières composantes, ce qui est excellent, ce dataset ne présente aucune variable mal représentée. Par ailleurs, aucun point ne se détache grandement des autres donc pas d'outliers problématiques à gérer.

Nous pouvons passer aux tests de clustering.

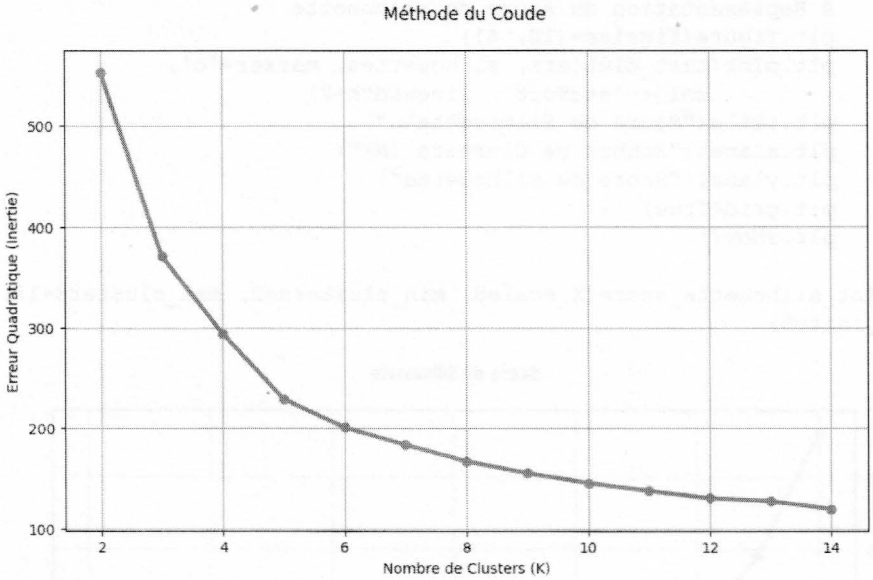
3.1.2 Les tests pour déterminer le nombre de clusters

Différents tests peuvent être utilisés pour déterminer le nombre optimal de clusters. Chacun de ces tests peut produire des résultats variés, qu'il conviendra d'évaluer pour choisir la meilleure option. Nous allons pratiquer la méthode du coude et le score silhouette mais d'autres options comme l'indice Davis-Bouldin, l'indice de Calinski-Harabasz ou le coefficient de Dunn peuvent être mis en œuvre.

La méthode du coude

La méthode du coude consiste à tracer la somme des erreurs quadratiques en fonction du nombre de clusters et à identifier le « coude » du graphique, où l'ajout de nouveaux clusters n'améliore plus significativement la qualité du clustering.

```
def plot_elbow(X, min_clusters=2, max_clusters=15,
n_init="auto",random_state=0):
    cost = []
    test_clusters = range(min_clusters, max_clusters)

    for i in test_clusters :
        cls = KMeans(n_clusters=i,
                     n_init=n_init,
                     init='k-means++',
                     random_state=random_state)
        cls.fit(X)
        cost.append(cls.inertia_)

    print(np.argmin(np.diff(cost)))
    plt.figure(figsize=(10, 6))
    plt.plot(test_clusters,
             cost,
             color='#439cc8',
             linewidth=3,
             marker='o')
    plt.title("Méthode du Coude\n")
    plt.xlabel("Nombre de Clusters (K)")
    plt.ylabel("Erreur Quadratique (Inertie)")
    plt.grid(True)
    plt.show()

plot_elbow(X_scaled, min_clusters=2, max_clusters=15, n_init=25)
```



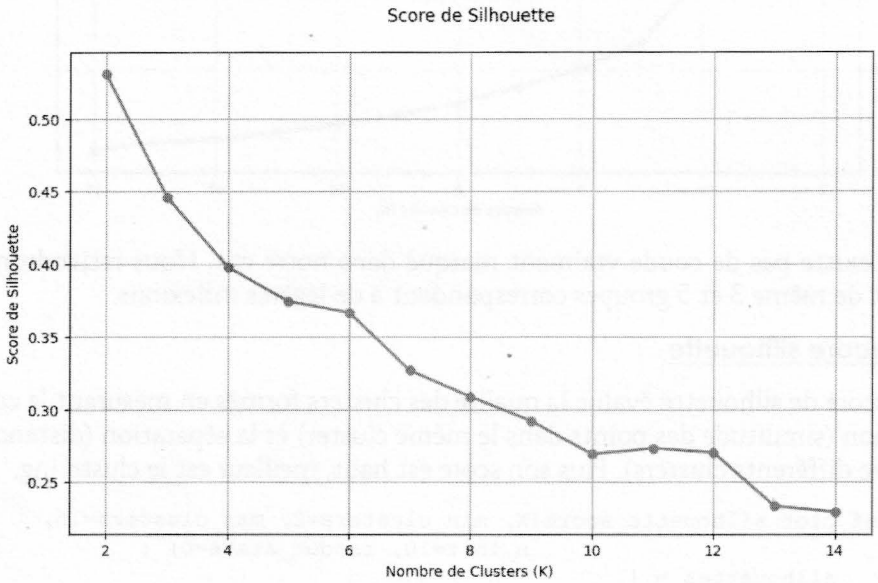
Il n'existe pas de coude vraiment marqué dans notre cas. Nous retiendrons tout de même 3 et 5 groupes correspondant à de légères inflexions.

Le score silhouette

Le score de silhouette évalue la qualité des clusters formés en mesurant la cohésion (similitude des points dans le même cluster) et la séparation (distance entre différents clusters). Plus son score est haut, meilleur est le clustering.

```
def plot_silhouette_score(X, min_clusters=2, max_clusters=15,
                          n_init=10, random_state=0) :
    silhouettes = []
    test_clusters = range(min_clusters, max_clusters)

    for num_clusters in test_clusters :
        cls = KMeans(n_clusters=num_clusters, n_init=n_init,
                     init='k-means++',
random_state=random_state)
        cls.fit(X)
        silh = metrics.silhouette_score(X, cls.labels_)
        silhouettes.append(silh)
```

```
    # Représentation du score de silhouette
    plt.figure(figsize=(10, 6))
    plt.plot(test_clusters, silhouettes, marker='o',
             color='#439cc8', linewidth=2)
    plt.title("Score de Silhouette\n")
    plt.xlabel("Nombre de Clusters (K)")
    plt.ylabel("Score de Silhouette")
    plt.grid(True)
    plt.show()

plot_silhouette_score(X_scaled, min_clusters=2, max_clusters=15,
n_init=5)
```



Le score silhouette est maximal pour deux groupes à 0,53, ce qui est une performance moyenne. L'ajout de groupes supplémentaires n'entraîne pas d'amélioration.

Nous allons ainsi tester un K-means à 2, 3 et 5 groupes sur le jeu de données et évaluer ensuite l'option la plus pertinente.

3.1.3 Choix du clustering

Choisir un clustering consiste à essayer différentes options, les visualiser et déterminer le meilleur choix par rapport au but recherché. Dans notre cas, nous cherchons juste à regrouper les pingouins en groupes cohérents.

Nous allons ainsi calculer le K-means pour chaque option puis réattribuer à chaque ligne son cluster et enfin, afficher un tableau de synthèse des valeurs moyennes de chaque cluster pour chaque variable.

Avant de procéder aux tests, voici la fonction qui va permettre de créer les tableaux de synthèse :

```
from sklearn.preprocessing import MinMaxScaler

def df_cluster_means(X, labels, column_names) :

    # Création d'un DataFrame aux dimensions des données
    df = pd.DataFrame(X, columns=column_names)

    # Ajout des clusters
    df['Cluster'] = labels

    # Regroupement par cluster
    cluster_means = df.groupby('Cluster').mean()

    # Normalisation
    scaler = MinMaxScaler(feature_range=(0, 10))
    scaled_means = scaler.fit_transform(cluster_means)

    # Finalisation du tableau
    df_fin =  pd.DataFrame(scaled_means,
                           columns=cluster_means.columns,
                           index=cluster_means.index)
    return

df_fin.style.background_gradient(cmap='Blues').format('{:.1f}')
```

Nous pouvons maintenant lancer les tests :

```
from IPython.display import display, HTML

# Définition des clusters
n_clusters_list = [2, 3, 5]

# Dictionnaire de stockage des résultats
cluster_dfs = {}
```

```
for n_clusters in n_clusters_list:
    kmeans = KMeans(
        n_clusters=n_clusters,
        n_init=10,
        init='k-means++',
        random_state=0
    )
    labels = kmeans.fit_predict(X_scaled)
    cluster_dfs[n_clusters] = df_cluster_means(X_scaled, labels, features)

# Affichage des DataFrames pour chaque clustering
for n_clusters in n_clusters_list :
    display(HTML(
        f'<h3 style="font-family : Arial; color: #439cc8;">'
        f'Clustering à {n_clusters} clusters:</h3>'
    ))
    display(cluster_dfs[n_clusters])
    print()
```

Clustering à 2 clusters:

Cluster	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
0	0.0	10.0	0.0	0.0
1	10.0	0.0	10.0	10.0

Clustering à 3 clusters:

Cluster	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
0	10.0	10.0	2.9	2.0
1	9.9	0.0	10.0	10.0
2	0.0	8.3	0.0	0.0

Clustering à 5 clusters:

Cluster	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
0	9.5	8.4	2.8	1.8
1	10.0	2.8	10.0	10.0
2	0.0	6.5	0.0	0.0
3	6.4	0.0	7.4	6.1
4	2.1	10.0	1.9	3.4

Les tableaux suivants décrivent les trois regroupements que nous allons commenter :

- Le clustering à 2 groupes oppose très clairement les petits pingouins au bec court (0) aux gros pingouins au long bec.
- Le clustering à 3 groupes conserve les deux catégories précédentes (1 et 2) mais isole un nouveau groupe de pingouins plutôt moyens avec un bec long et haut.
- Le clustering à 5 groupes présente des différences plus subtiles entre les groupes et mériterait de plus amples investigations pour définir correctement ces groupes.

En définitive, nous retrouvons le grand clivage naturel entre les deux grands groupes avec le premier cas et la solution à trois groupes qui se rapprocherait peut-être de la catégorisation par espèce fournie par la variable `species`.

C'est justement l'occasion de vérifier si les trois groupes issus du clustering sont similaires aux trois espèces de pingouins.

3.1.4 Le score ARI

L'*Adjusted Rand Index* (ARI) est une mesure de similarité entre deux partitions qui retourne une valeur entre -0,5 et 1 permettant d'évaluer la similarité de la manière suivante :

- Les partitions sont identiques pour une valeur de 1.
- Les partitions correspondent à un regroupement aléatoire pas meilleur que le hasard pour une valeur de 0.
- En passant en dessous de 0, les partitions sont considérées comme moins similaires qu'un regroupement aléatoire.

Voyons en pratique le résultat du score ARI pour notre exemple :

```
from sklearn.metrics import adjusted_rand_score

# Calcul du clustering K-means avec 3 groupes
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X_scaled)
labels = kmeans.labels_
```

```
# Les étiquettes réelles des espèces
y = df["species"].astype('category').cat.codes

# Calcul du score ARI
ari_score = adjusted_rand_score(y, labels)

print(f"\nScore ARI : {ari_score:.3f}")


# Output:
# Score ARI : 0.799
```

Le score ARI de 0,799 confirme une forte similarité entre la catégorisation des espèces et les regroupements issus du clustering et démontre la puissance des algorithmes non supervisés de clustering.

3.2 Les autres algorithmes de clustering

Il existe de nombreux algorithmes de clustering au fonctionnement divergent, capables de répondre à divers besoins. Nous allons faire connaissance avec trois d'entre eux : le GMM, le Meanshift et le DBSCAN.

3.2.1 GMM

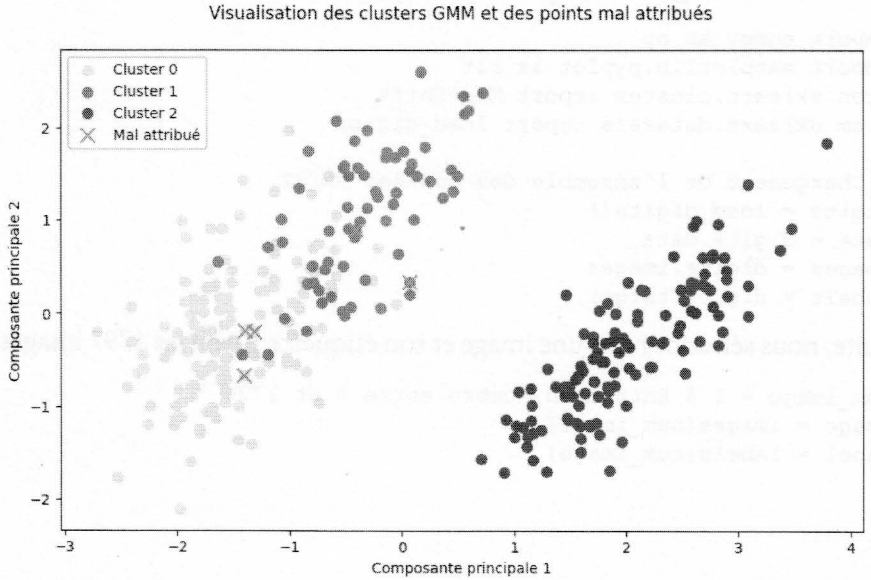
L'algorithme du *Gaussian Mixture Model* (GMM) crée des clusters en recherchant des distributions gaussiennes, donc normales, dans les données. Il peut ainsi détecter des clusters qui se chevauchent, à la différence du K-means. Pour ce faire, il utilise l'algorithme d'*Expectation-Maximization* (EM) afin d'estimer les paramètres de ces distributions. Par ailleurs, le nombre de groupes souhaité peut être défini, ce qui facilite son utilisation.

Appliquons cet algorithme à nos données précédentes pour comparer ses performances :

```
from sklearn.mixture import GaussianMixture

from sklearn.metrics import adjusted_rand_score

# Initialisation d'un GMM à 3 groupes
gmm = GaussianMixture(n_components=3, random_state=42)
gmm.fit(X_scaled)
```

```
labels = gmm.predict(X_scaled)

# Les étiquettes réelles des espèces
y = df["species"].astype('category').cat.codes

# Calcul du score ARI
ari_score = adjusted_rand_score(y, labels)

print(f"\nScore ARI : {ari_score:.3f}")

# Output:
# Score ARI : 0.959
```

Le GMM affiche un score impressionnant de 0,959 soit plus de 16 points que le K-means. Seuls 5 points sur 333 se sont avérés appartenir au mauvais groupe comme le montre le graphique ci-dessous :



En décomposant une distribution complexe en plusieurs distributions normales, le GMM peut être appliqué dans divers domaines où la compréhension des sous-structures est cruciale comme le marketing, la détection de fraudes ou la reconnaissance de formes.

3.2.2 Meanshift

L'algorithme de `MeanShift` est une méthode de clustering basée sur la recherche des zones où les points de données sont les plus concentrés, formant ainsi des groupes naturels dans les données. Cette approche le rend particulièrement adapté à la vision par ordinateur et au traitement d'images. Cependant, à la différence d'autres méthodes de clustering, il n'est pas possible de fixer à l'avance le nombre de groupes, ce qui peut représenter une contrainte pour certaines tâches.

Plutôt que de l'expérimenter sur le jeu des pingouins (pour lequel il ne trouve que deux groupes avec un ARI de 0,639), nous allons le mettre en œuvre sur le jeu des MNIST (*Modified National Institute of Standards and Technology*), qui est un ensemble de chiffres manuscrits.

Il faut d'abord charger les images et les étiquettes :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import MeanShift
from sklearn.datasets import load_digits

# Chargement de l'ensemble des données MNIST
digits = load_digits()
data = digits.data
images = digits.images
labels = digits.target
```

Ensuite, nous sélectionnons une image et son étiquette parmi les 1797 images :

```
num_image = 1 # Entrer un nombre entre 0 et 1796
image = images[num_image]
label = labels[num_image]
```

Il faut ensuite convertir l'image en coordonnées de pixels. À l'aide de la fonction `meshgrid`, nous allons récupérer les coordonnées x et y et créer une matrice à trois entrées avec la coordonnée x en première colonne, y en seconde et l'intensité pour la dernière colonne. Avoir les coordonnées x, y et leur intensité permet ensuite à l'algorithme de regrouper les teintes selon les zones :

```
# Conversion en coordonnées de pixels (x, y) avec intensités
x, y = np.meshgrid(
    np.arange(image.shape[1]),
    np.arange(image.shape[0])
)

pixel_coordinates = np.stack(
    (x.flatten(),
     y.flatten(),
     image.flatten()),
    axis=-1
)
```

Application du Meanshift :

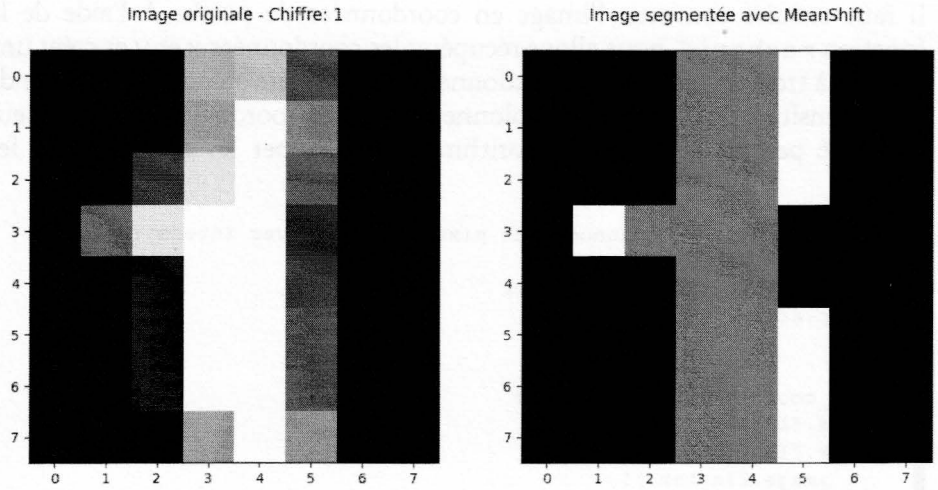
```
mean_shift = MeanShift()
mean_shift.fit(pixel_coordinates)
labels_meanshift = mean_shift.labels_
```

Ensuite, il est nécessaire de retransformer les labels obtenus au format de l'image :

```
segmented_image = labels_meanshift.reshape(image.shape)
```

Enfin, les deux images, normale et segmentée, sont affichées grâce au code suivant :

```
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title(f'Image originale - Chiffre: {label}\n')

plt.subplot(1, 2, 2)
plt.imshow(segmented_image, cmap='gray')
plt.title('Image segmentée avec MeanShift\n')

plt.show()
```



La segmentation des images avec `MeanShift` va ainsi faciliter la séparation des chiffres du fond, la reconnaissance individuelle de chaque chiffre ou l'amélioration de la qualité des images, ce qui aura un impact positif sur une classification ultérieure, par exemple.

3.2.3 DBSCAN

Le DBSCAN (*Density-Based Spatial Clustering of Applications with Noise*) est une méthode de clustering qui identifie les régions de haute densité de points en fonction d'une distance et d'un nombre de voisins définis préalablement (paramètres nommés respectivement `eps` et `min_samples`). C'est d'ailleurs le paramétrage de ces deux critères qui rend l'utilisation du DBSCAN plus complexe que celle des autres méthodes vues précédemment.

En contrepartie de cette complexité, DBSCAN permet de détecter efficacement des formes de clusters arbitraires, d'isoler les outliers (groupe numéroté -1) et de traiter des ensembles de données avec des densités variables, ce qui le rend particulièrement efficace pour des applications telles que la détection d'anomalies ou la segmentation d'images.

Remarque

Attention toutefois à ce que les densités des clusters ne soient pas trop différentes.

Prenons l'exemple de tournées fictives d'un taxi où la destination de chaque course est représentée par une coordonnée GPS. C'est l'occasion d'utiliser la fonction `make_blobs` de Scikit-Learn qui permet de créer des clusters aléatoires autour de centroïdes :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs

# Création des données GPS synthétiques
centers = [(4, 4), (2, 2), (-1, -1)]
taxi_ride, _ = make_blobs(n_samples=200, centers=centers,
                          cluster_std=0.4, random_state=42)
```

En plus des 200 courses concentrées sur 3 zones, nous allons ajouter 20 courses aléatoires avec une plus forte variabilité :

```
# Ajout de 20 courses exceptionnelles
n_noise = 20
noise = np.random.uniform(low=-3, high=6, size=(n_noise, 2))
taxi_ride = np.concatenate([taxi_ride, noise], axis=0)
```

Ensuite, nous appliquons le `DBSCAN` :

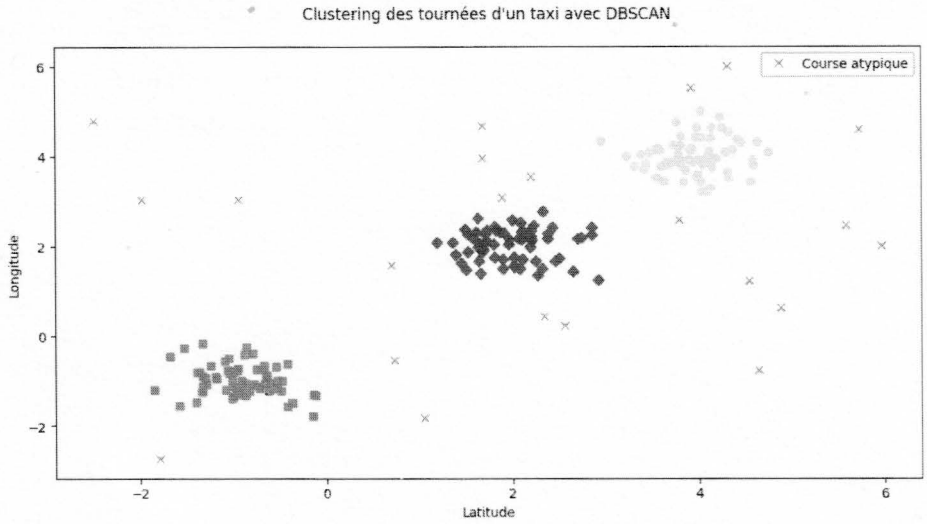
```
# Application du DBSCAN
dbscan = DBSCAN(eps=0.5, min_samples=5)
labels_dbscan = dbscan.fit_predict(taxi_ride)
```

Définition des couleurs et marqueurs pour rendre le graphique plus lisible :

```
# Préparer les couleurs et les marqueurs
cluster_colors = ['#EEE', '#439CC8', '#285E78']
cluster_markers = ['o', 's', 'D']
anomaly_color = 'darkorange'
anomaly_marker = 'x'
```

Nous pouvons enfin visualiser le clustering réalisé grâce à ce code :

```
# Visualisation du DBSCAN
plt.figure(figsize=(12, 6))
unique_labels = set(labels_dbscan)

# Visualisation des clusters et des courses atypiques
for k in unique_labels :
    class_member_mask = (labels_dbscan == k)
    xy = taxi_ride[class_member_mask]

    if k == -1:
        # Courses atypiques identifiées (-1)
        col = anomaly_color
        marker = anomaly_marker
        label = 'Course atypique'
    else:
        # Clusters de courses classiques
        col = cluster_colors[k % len(cluster_colors)]
        marker = cluster_markers[k % len(cluster_markers)]
        label = None

    plt.plot(
        xy[:, 0],
        xy[:, 1],
        marker,
        c=col,
        markersize=6,
        alpha=0.9,
        label=label
    )

plt.title('Clustering des données GPS avec DBSCAN\n')
plt.xlabel('Latitude')
plt.ylabel('Longitude')
plt.legend()
plt.show()
```



En utilisant le clustering par densité, le DBSCAN a identifié trois groupes de tournées dites classiques. Les points situés en dehors de ces zones denses ont été classés comme groupe -1 et sont représentés par des croix.

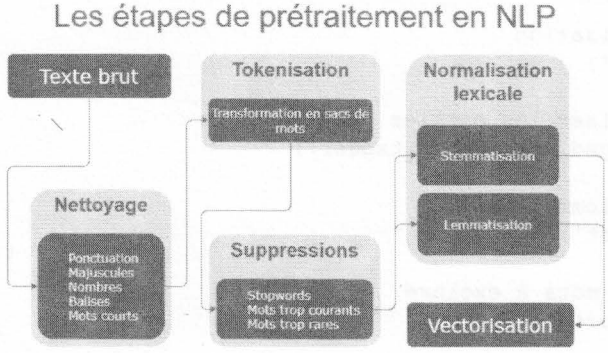
Ce chapitre sur l'apprentissage non supervisé touche à sa fin. Les algorithmes que nous avons explorés demandent de la pratique mais seront d'une grande aide pour résumer les données de manière efficace et distinguer les informations importantes de celles qui sont accessoires.

Chapitre 9
Modéliser le texte et l'image

1. La modélisation du texte

Le traitement du langage naturel (NLP, pour *Natural Language Processing*) est un domaine essentiel de la data science, visant à permettre aux machines d'interagir efficacement avec le langage humain. Comme évoqué dans le chapitre Analyse des données, les phases de prétraitement et de vectorisation sont cruciales pour préparer les données textuelles et réaliser une modélisation efficace.

Pour celles et ceux qui n'ont pas parcouru le chapitre Analyse des données, voici un schéma récapitulatif des étapes de prétraitement :



Python propose plusieurs modules pour cette préparation, parmi lesquels NLTK, TextBlob et spaCy.

1.1 Les modules du NLP

Les modules Python de NLP facilitent grandement la préparation des textes en offrant un vaste éventail de fonctions pour transformer et enrichir les données textuelles. Cependant, ils varient de manière assez importante en termes de fonctionnalités, d'ergonomie et de performances.

1.1.1 NLTK

Conçu en 2005, NLTK, pour *Natural Language Toolkit*, est l'un des premiers outils développés pour le langage naturel. Il offre une grande variété de fonctionnalités et constitue un choix robuste pour les tâches linguistiques complexes.

Voici la commande pour l'installer :

```
pip install nltk
```

Lors de l'importation, il est important de télécharger les ressources nécessaires de NLTK pour le rendre pleinement opérationnel. Cela peut se faire en chargeant les ressources les plus populaires de la manière suivante :

```
import nltk
nltk.download('popular')
```

Il peut également être opportun de télécharger des ressources spécifiques en fonction de nos besoins, telles que les corpus pour l'analyse de sentiment ou les modèles pour le balisage des parties du discours. Voici quelques exemples :

```
# Ressource de tokenisation
nltk.download('punkt')

# Ressource pour baliser les parties du discours
nltk.download('averaged_perceptron_tagger')

# Ressource pour la lemmatisation
nltk.download('wordnet')

# Ressource pour les mots à exclure
nltk.download('stopwords')
```

Bien que nous privilégierons le module spaCy pour faciliter l'initiation au NLP, voici un court exemple de mise en pratique de NLTK :

```
#Import des modules et ressources nécessaires
import nltk

nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
```

```
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk import pos_tag

# Exemple de texte
text = "Give me liberty, or give me death! For without liberty,
life is not worth living."

# Tokenisation en phrases
sentences = sent_tokenize(text)
print(f"Tokenisation en phrases : \n{sentences}\n")

# Tokenisation en mots
words = word_tokenize(text)
print(f"Tokenisation en mots : \n{words}\n")

# Balisage des parties du discours
tags = pos_tag(words)
print(f"Balisage des parties du discours :  \n{tags}")
```

```
Tokenisation en phrases :
['Give me liberty, or give me death!', 'For without liberty, life is not worth living.']

Tokenisation en mots :
['Give', 'me', 'liberty', ',', 'or', 'give', 'me', 'death', '!', 'For', 'without', 'liberty', ',', 'life', 'is', 'not', 'worth', 'living', '.']

Balisage des parties du discours :
[('Give', 'VB'), ('me', 'PRP'), ('liberty', 'JJ'), (',', ','), ('or', 'CC'), ('give', 'VB'),
('me', 'PRP'), ('death', 'NN'), ('!', '.'), ('For', 'IN'), ('without', 'IN'), ('liberty', 'NN'),
(',', ','), ('life', 'NN'), ('is', 'VBZ'), ('not', 'RB'), ('worth', 'JJ'), ('living', 'NN'), ('.', '.')]
```

Remarque

L'exemple est en anglais car c'est la langue par défaut de NLTK. Il est important de vérifier la prise en charge et la gestion correcte des langues pour chaque module.

1.1.2 TextBlob

TextBlob est une version simplifiée de NLTK, conçue pour rendre le NLP plus accessible. Il se distingue par ses capacités d'analyse des sentiments ou ses capacités de corrections orthographiques. De plus, il peut être enrichi avec de nouveaux modèles ou langues grâce à des extensions, ce qui le rend particulièrement polyvalent.

L'installation a lieu de la façon suivante :

```
pip install textblob
```

Remarque

Attention : certaines fonctionnalités de TextBlob dépendent de NLTK. Il sera donc nécessaire de l'installer séparément pour pouvoir utiliser ces fonctionnalités.

Voici un court exemple pour illustrer les capacités de ce module. Un o a été volontairement ajouté après living pour simuler une faute d'orthographe.

Nous commençons par importer NLTK et TextBlob :

```
import nltk
nltk.download('popular')

from textblob import TextBlob
```

```
# Analyser un texte
text = "Give me liberty, or give me death! For without liberty,
life is not worth livingo."
blob = TextBlob(text)


# Tokenisation en phrases
print(f"Tokenization en mots :\n{blob.words}\n")

# Balisage du discours
print(f"Balisage :\n{blob.tags}\n")

# Correction de la phrase
print(f"Balisage :\n{blob.correct()}\n")

# Analyse de sentiment
print(f"Analyse de sentiments :\n{blob.sentiment}\n")
```

```
Tokenization en mots :
['Give', 'me', 'liberty', 'or', 'give', 'me', 'death', 'For', 'without', 'liberty', 'life', 'is', 'not', 'worth', 'livingo']
Balisage :
[('Give', 'VB'), ('me', 'PRP'), ('liberty', 'JJ'), ('or', 'CC'), ('give', 'VB'), ('me', 'PRP'), ('death', 'NN'),
('For', 'IN'), ('without', 'IN'), ('liberty', 'NN'), ('life', 'NN'), ('is', 'VBZ'), ('not', 'RB'), ('worth', 'JJ'), ('livingo', 'NN')]
Correction de la phrase :
Give me liberty, or give me death! For without liberty, life is not worth living.
Analyse de sentiments :
Sentiment(polarity=-0.15, subjectivity=0.1)
```

La fonction de correction a bien repéré le « o » et rendu la phrase correcte.

L'autre fonction intéressante d'analyse de sentiments attribue une note de -1 à 1 pour la `polarity` allant respectivement de très négatif à très positif. Cette fonction retourne également l'indicateur `subjectivity` qui indique de 0 si le contenu est parfaitement objectif et jusqu'à 1 pour ce qui est complètement subjectif.

1.1.3 spaCy

spaCy est un module de NLP qui offre un équilibre intéressant entre la simplicité et les fonctions limitées de TextBlob, et l'exhaustivité, bien que moins accessible, de NLTK. Il est réputé pour sa meilleure optimisation, offrant ainsi une plus grande vitesse, ainsi que pour ses modèles de haute qualité qui facilitent l'utilisation du NLP.

Lors de l'installation, il ne faut pas oublier de télécharger le modèle de langue utilisé. Pour notre exemple, nous emploierons le petit modèle français entraîné sur des textes d'actualité et de médias :

```
# Installation de spaCy
pip install spacy

# Installation du modèle de langue française
python -m spacy download fr_core_news_sm
```

Remarque

Pour le français, il existe trois tailles de modèles de poids croissant : sm, md et lg, allant de 15 Mo à 545 Mo.

Précisons que les modèles anglais et chinois, à la différence des autres, sont entraînés sur le Web. L'import du modèle anglais est donc légèrement différent (`core_web` à la place de `core_news`) :

```
# Installation du modèle de langue anglaise
python -m spacy download en_core_web_sm
```

Avant de passer à une mise en pratique sur un dataset, voici un simple exemple d'utilisation de spaCy sur une phrase :

```
import spacy

# Chargement du modèle préentraîné pour le français
nlp = spacy.load("fr_core_news_sm")

# Exemple de phrase
text = "Nous partîmes cinq cents ; mais par un prompt renfort,
Nous nous vîmes trois mille en arrivant au port,"

# Nettoyage de la ponctuation
doc = nlp(text)
tokens_no_punct = [token for token in doc if not token.is_punct]

# Tokenisation (déjà effectuée lors du passage du texte à nlp)
tokens = [token.text for token in tokens_no_punct]
print(f"Tokens (sans ponctuation) :\n{tokens}\n")

# Suppression des stopwords
stopwords = spacy.lang.fr.stop_words.STOP_WORDS
filtered_tokens = [token for token in tokens_no_punct
                   if token.text.lower() not in stopwords]

ft_texts= [token.text for token in filtered_tokens]
print(f"Filtered Tokens (sans stopwords) :\n{ft_texts}\n")

# Lemmatisation
lemmas = [token.lemma_ for token in filtered_tokens]
print(f"Lemmatisation :\n{lemmas}\n")
```

```
Tokens (sans ponctuation) :
['Nous', 'partîmes', 'cinq', 'cents', 'mais', 'par', 'un', 'prompt',
'renfort', 'Nous', 'nous', 'vîmes', 'trois', 'mille', 'en', 'arrivant', 'au', 'port']
Filtered Tokens (sans stopwords) :
['partîmes', 'cents', 'prompt', 'renfort', 'vîmes', 'arrivant', 'port']
Lemmatisation :
['partir', 'cent', 'prompt', 'renfort', 'voir', 'arriver', 'port']
```

Remarque

spaCy ne possède pas de fonction de stemmatisation. Il sera nécessaire de recourir à NLTK pour ce faire.

Une dernière remarque globale commune à tous les modules : la transformation des mots en minuscules doit être guidée par l'objectif recherché. Dans certains contextes, il peut être judicieux de conserver les majuscules. Pour les cas les plus litigieux, il ne faut pas hésiter à tester les deux approches.

1.2 Mise en pratique de la NLP

Nous allons mettre en pratique le NLP sur un sujet qui ne contient pas trop de subtilités linguistiques : le Répertoire opérationnel des Métiers et des Emplois (ROME). Ce répertoire sert à identifier le plus précisément possible chaque emploi.

Nous allons ainsi étudier s'il est possible de retrouver le grand domaine d'activité à partir de l'intitulé exact de la profession.

1.2.1 Prétraitement des données

Le prétraitement des données sera réalisé avec spaCy. Il faudra donc au préalable importer le modèle français :

```
python -m spacy download fr_core_news_sm
```

Puis acquérir les données :

```
import pandas as pd

df_deb =
pd.read_csv("https://raw.githubusercontent.com/eric2mangel/BDD/
main/ROME_metiers.csv", sep=",",encoding='utf_8', low_memory=False)

print(df_deb.shape)
df_deb.head()
```

(31977, 13)

	Grand domaine	Domaine professionnel	Fiche ROME	Libellé	Code OGR	Grand Domaine	domaine Professionnel Code
0	H	29	2	Chaudronnier-tuyauteur / Chaudronnière-tuyaute...	11980	Industrie	H29
1	H	29	2	Chaudronnier-tuyauteur / Chaudronnière-tuyaute...	11980	Industrie	H29
2	H	29	2	Chef d'équipe traceur / traceuse en chaudronnerie	12493	Industrie	H29
3	H	29	2	Opérateur / Opératrice en tôlerie industrielle	17222	Industrie	H29
4	H	29	2	Serrurier métallier industriel / Serrurière mé...	19418	Industrie	H29

	Domaine Professionnel	Code ROME	Code NAF	Label NAF	code_rome_2	domaine_pro_2
	Mécanique, travail des métaux et outillage	H2902	33	Réparation et installation de machines et d'éq...	H2902	Mécanique, travail des métaux et outillage
	Mécanique, travail des métaux et outillage	H2902	28	Fabrication de machines et équipements n.c.a.	H2902	Mécanique, travail des métaux et outillage
	Mécanique, travail des métaux et outillage	H2902	23	Fabrication d'autres produits minéraux non mét...	H2902	Mécanique, travail des métaux et outillage
	Mécanique, travail des métaux et outillage	H2902	25	Fabrication de produits métalliques, à l'excep...	H2902	Mécanique, travail des métaux et outillage
	Mécanique, travail des métaux et outillage	H2902	23	Fabrication d'autres produits minéraux non mét...	H2902	Mécanique, travail des métaux et outillage

Le fichier contient 31977 observations et 13 colonnes. Nous allons préparer les données dans le but de prédire la variable `Grand Domaine` pour chaque profession contenue dans la variable `Libellé`.

Comme c'est une première approche destinée à vérifier la viabilité de cette classification, nous allons sélectionner au hasard 5000 observations afin de limiter les temps de calcul :

```
df = df_deb.sample(n=5000, random_state=42)
```

Nous pouvons maintenant procéder au prétraitement de la variable `Libellé` pour laquelle nous allons utiliser la fonction de prétraitement suivante :

```
import pandas as pd
import re
import spacy
from nltk.stem import SnowballStemmer
import nltk


# Chargement du modèle spaCy pour le français
nlp = spacy.load("fr_core_news_sm")

# Initialisation du Stemmer NLTK
stemmer = SnowballStemmer("french")

# Charger les stopwords de spaCy et NLTK
stopwords_spacy = nlp.Defaults.stop_words

# Fonction de prétraitement générique
def preprocess_text(text, use_stemming=False):
    doc = nlp(text)
    tokens = [
        stemmer.stem(token.lemma_.lower())
        if use_stemming else token.lemma_.lower()
        for token in doc
        if not token.is_punct and
        token.lemma_.lower() not in stopwords_spacy and
        token.text.lower() not in stopwords_spacy
    ]
    return ' '.join(tokens)
```

La fonction `preprocess_text` transforme le contenu textuel de chaque observation de la manière suivante :

- La ligne `doc = nlp(text)` est une fonction spaCy qui prétransforme le texte brut en une représentation structurée et annotée ;
- La variable `tokens` est créée. C'est une liste constituée des mots du doc à condition qu'ils ne soient pas de la ponctuation et qu'ils n'appartiennent pas aux `stopwords` de spaCy.

Nous pouvons ensuite créer deux nouvelles variables : une pour la version lemmatisée et une pour la version stemmatisée :

```
# Lemmatisation
df['label_lemma'] = df['Libellé'].apply(
    lambda x: preprocess_text(x)
)

# Stemmatisation
df['label_stem'] = df['Libellé'].apply(
    lambda x: preprocess_text(x, use_stemming=True)
)
```

Remarque

Tous les entraînements et classifications qui vont suivre s'appuieront sur ces deux variables pour traiter les données textuelles.

Avant de commencer la modélisation, il est important de faire un point sur la méthodologie. Nous allons utiliser ici deux méthodes différentes pour extraire les caractéristiques des textes consistant à dénombrer les mots de deux manières distinctes : le `CountVectorizer` et le TF-IDF.

1.2.2 Les extracteurs de caractéristiques

CountVectorizer

Le CountVectorizer est un algorithme permettant de convertir un corpus de texte en une matrice dénombrant les occurrences de chaque mot ou groupe de mots. Il est possible de moduler différents paramètres comme le nombre d'apparitions minimales, maximales des mots ou la possibilité d'utiliser des groupes de mots. Voici un petit tableau non exhaustif des possibilités :

Option	Rôle
`max_df`	Ignore les termes ayant une fréquence de document strictement supérieure à ce seuil (proportion ou nombre absolu).
`min_df`	Ignore les termes ayant une fréquence de document strictement inférieure à ce seuil (proportion ou nombre absolu).
`max_features`	Garde uniquement les termes les plus fréquents. Utile pour limiter la taille du vocabulaire.
`ngram_range`	Tuple (min_n, max_n) spécifiant la plage de n-grammes à extraire. Par exemple (1, 2) pour extraire des unigrammes et des bigrammes.
`stop_words`	Liste de mots à ignorer.

TF-IDF

Le TF-IDF apporte une pondération au dénombrement des termes. Comme le `CountVectorizer`, il compte les termes dans un premier temps (TF pour *Term Frequency*) puis pondère ces comptages pour réduire l'importance des termes trop fréquents en mettant en avant les mots qui sont bien distinctifs (IDF pour *Inverse Document Frequency*).

Remarque

LE TF-IDF utilise les mêmes paramètres que le CountVectorizer.

1.2.3 La modélisation

La modélisation peut débuter. Nous allons tester pour les deux méthodes de vectorisation (CountVectorizer et TF-DF) différents algorithmes sur les données stemmatisées et lemmatisées. Nous sélectionnerons à la fin la combinaison la plus performante.

Nous allons commencer par importer les modules nécessaires et préparer les dictionnaires d'algorithmes et d'hyperparamètres :

```
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split,
GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer,
TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Définition des jeux de données
X_lemma = df['label_lemma']
X_stem = df['label_stem']
y = df['Grand Domaine']

# Splits
X_train_lemma, X_test_lemma, y_train, y_test = train_test_split(
    X_lemma,
    y,
    test_size=0.2,
    random_state=42,
    stratify=y
)

X_train_stem, X_test_stem, y_train, y_test = train_test_split(
    X_stem,
    y,
    test_size=0.2,
    random_state=42,
```

```
    stratify=y
)
# Définition des algorithmes
models = {
    'logreg': LogisticRegression(max_iter=500,solver='saga',),
    'knn': KNeighborsClassifier(),
    'rf': RandomForestClassifier()
}

# Définition des hyperparamètres
params = {
    'logreg': {'clf__C': [0.1, 1, 10],
               'clf__penalty': ['l1','l2']},
    'knn': {'clf__n_neighbors': [3, 5, 7],
            'clf__weights': ['uniform', 'distance']},
    'rf': {'clf__n_estimators': [100, 200],
           'clf__max_depth': [None, 10, 20]}
}

# Vectorizers utilisés avec paramètres
vectorizers = {
    'count': CountVectorizer(max_df=0.85, ngram_range=(1, 2)),
    'tfidf': TfidfVectorizer(max_df=0.85, ngram_range=(1, 2))
}
```

Ensuite, nous pouvons lancer la modélisation en testant toutes les combinaisons :

```
# Variables pour stocker les résultats
best_model = None
best_score = 0
best_model_name = ''
best_params = {}

# Boucle de tests des combinaisons
for vectorizer_name, vectorizer in vectorizers.items():
    for model_name, model in models.items():
        for use_lemma in [True, False]:
            # Définition du pipeline
            pipeline = Pipeline([
                ('vect', vectorizer),
                ('clf', model)
            ])

            # Définition des hyperparamètres
```

```
            param_grid = params[model_name]
            grid_search = GridSearchCV(
                pipeline,
                param_grid,
                cv=5, n_jobs=-1,
                scoring='accuracy'
            )

            # Sélection des données
            X_train, X_test = (
                (X_train_lemma, X_test_lemma) if use_lemma
                else (X_train_stem, X_test_stem)
            )

            # Entraînement du modèle
            grid_search.fit(X_train, y_train)

            # Mise à jour du meilleur modèle si nécessaire
            current_score = grid_search.best_score_
            if current_score > best_score:
                best_score = current_score
                best_model = grid_search.best_estimator_
                best_model_name = (
                    f"{model_name}_{vectorizer_name}_"
                    f"{'lemma' if use_lemma else 'stem'}"
                )
                best_params = grid_search.best_params_

            current_model = (
                f"{model_name}_{vectorizer_name}_"
                f"{'lemma' if use_lemma else 'stem'}"
            )
            print(f"{current_model} / score: {current_score}")

print(f"\nMeilleur modèle: {best_model_name}")
print(f"Score: {best_score}")
print(f"Meilleurs hyperparamètres: {best_params}")
```

```
logreg_count_lemma / score: 0.8055
logreg_count_stem / score: 0.8105
knn_count_lemma / score: 0.607
knn_count_stem / score: 0.60875
rf_count_lemma / score: 0.7440000000000001
rf_count_stem / score: 0.74725
logreg_tfidf_lemma / score: 0.80175
logreg_tfidf_stem / score: 0.8030000000000002
knn_tfidf_lemma / score: 0.72875
knn_tfidf_stem / score: 0.71875
rf_tfidf_lemma / score: 0.7372500000000001
rf_tfidf_stem / score: 0.7385

Meilleur modèle: logreg_count_stem
Score: 0.8105
Meilleurs hyperparamètres: {'clf__C': 10, 'clf__penalty': 'l2'}
```

La réalisation des tests dure environ 15 minutes sur Google Colab.

La régression logistique, basée sur des données stemmatisées vectorisées par un `CountVectorizer`, s'avère être la solution la plus performante avec un accuracy score d'environ 0,81.

Le rapport de classification est globalement bon affichant des `precision` et des `recall` allant de bons à excellents pour toutes les catégories :

```
y_pred = best_model.predict(X_test_stem)

print("Rapport de classification:")
print(classification_report(y_test, y_pred))
```

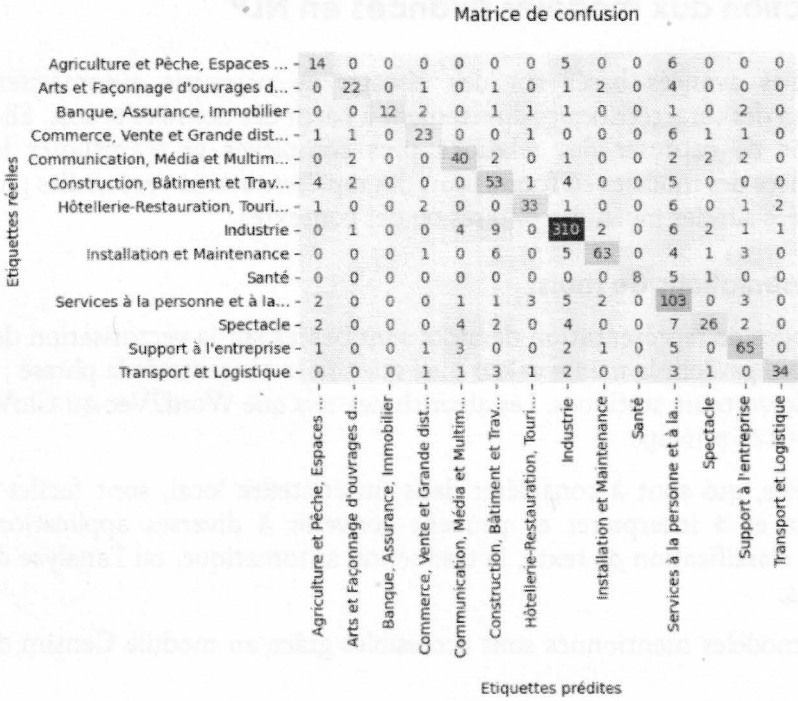
```
Rapport de classification:
                                                    precision    recall  f1-score   support

Agriculture et Pêche, Espaces naturels et Espaces verts, Soins aux animaux       0.56      0.56      0.56        25
                        Arts et Façonnage d'ouvrages d'art       0.79      0.71      0.75        31
                                  Banque, Assurance, Immobilier       1.00      0.60      0.75        20
                             Commerce, Vente et Grande distribution       0.74      0.68      0.71        34
                                Communication, Média et Multimédia       0.75      0.78      0.77        51
                             Construction, Bâtiment et Travaux publics       0.68      0.77      0.72        69
                  Hôtellerie-Restauration, Tourisme, Loisirs et Animation       0.82      0.72      0.77        46
                                                        Industrie       0.91      0.92      0.92       336
                                  Installation et Maintenance       0.90      0.76      0.82        83
                                                          Santé       1.00      0.57      0.73        14
                  Services à la personne et à la collectivité       0.63      0.86      0.73       120
                                                     Spectacle       0.76      0.55      0.64        47
                                  Support à l'entreprise       0.81      0.83      0.82        78
                                  Transport et Logistique       0.92      0.74      0.82        46

                                                    accuracy                           0.81      1000
                                                   macro avg       0.81      0.72      0.75      1000
                                                weighted avg       0.82      0.81      0.81      1000
```

Pour finir, la matrice de confusion affiche très clairement une diagonalisation marquée où toutes les catégories sont globalement bien identifiées.

```
# Obtention des catégories uniques
unique_categories = df['Grand Domaine'].astype("category")
.cat.categories

# Matrice de confusion
conf_matrix = confusion_matrix(
    y_test, y_pred, labels=unique_categories
)

# Fonction pour abréger les noms de catégories
def abbreviate_names(names, max_length=30):
    return [name[:max_length] + '...'
            if len(name) > max_length
            else name for name in names
    ]

# Application de la fonction
abbreviated_categories = abbreviate_names(unique_categories)

# Affichage de la matrice de confusion
plt.figure(figsize=(6, 4))
sns.heatmap(
    conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False,
    xticklabels=abbreviated_categories,
    yticklabels=abbreviated_categories
)
plt.xlabel('\nEtiquettes prédites')
plt.ylabel('Etiquettes réelles')
plt.title('Matrice de confusion\n')
plt.show()
```

Remarque

Une fonction d'abréviation a été ajoutée pour faciliter l'affichage des noms de catégories qui sont très longs.



Avec `CountVectorizer` et TF-IDF, nous avons utilisé ici une extraction manuelle des caractéristiques produisant des embeddings simples. Rappelons qu'un embedding est une représentation vectorielle qui capture les relations sémantiques des mots dans un espace de dimension réduite. Bien que ces méthodes soient efficaces pour certaines tâches, elles présentent des limitations en matière de capture des relations contextuelles et sémantiques entre les mots.

Pour surmonter ces limites, il est nécessaire d'explorer des approches plus avancées, basées sur des réseaux de neurones.

1.3 Introduction aux modèles avancés en NLP

Les modèles avancés basés sur des réseaux de neurones automatisent l'extraction des caractéristiques directement à partir des données brutes. Elles permettent de capturer des relations plus complexes et d'améliorer les performances des modèles en fournissant des représentations vectorielles plus riches basées sur des mots, des phrases ou des contextes.

1.3.1 Les représentations de mots

Les méthodes de représentation de mots sont basées sur la vectorisation des vocables, qui gardent la même valeur quel que soit le contexte de la phrase ; il s'agit ici de vecteurs statiques. Les algorithmes tels que Word2Vec ou GloVe reposent sur ce principe.

Ces modèles, qui sont à considérer dans un contexte local, sont faciles à comprendre et à interpréter et peuvent convenir à diverses applications comme la classification de texte, la traduction automatique, ou l'analyse de sentiments.

Les deux modèles mentionnés sont accessibles grâce au module Gensim de Python :

```
pip install gensim
```

Voici un petit exemple pour pratiquer Word2vec. Nous importerons depuis l'API de Gensim le fichier « text8 » qui est une version réduite de Wikipédia. Il contient des articles de l'encyclopédie présentés sous forme de listes de sacs de mots dont voici des exemples abrégés :

```
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse',
'first', 'used', 'against'...]
['reciprocity', 'qualitative', 'impairments', 'in',
'communication', 'as', 'manifested'...]
```

Ce jeu de données servira à entraîner Word2Vec qui permet de trouver des mots similaires en fonction de leur contexte d'utilisation dans les textes :

```
# Import des bibliothèques nécessaires
from gensim.models import Word2Vec
import gensim.downloader as api
```

```
# Chargement de text8 (sous-ensemble de Wikipédia)
corpus = api.load('text8')


# Entraînement du modèle Word2Vec sur text8
model = Word2Vec(corpus,
                 vector_size=100,
                 window=5,
                 min_count=5, workers=4)
```

Quelques compléments d'information sur les arguments utilisés au-dessus :

- `corpus` : jeu de données utilisé pour l'entraînement ;
- `vector_size = 100` : nombre de dimensions des vecteurs de mots. Plus la valeur est élevée, plus le modèle est précis mais les temps d'entraînement seront plus longs ;
- `window=5` : nombre de mots pris en compte avant et après pour construire le contexte du mot ;
- `min_count=5` : nombre d'apparitions minimum d'un mot pour être retenu dans le modèle ;
- `workers=4` : nombre de threads à utiliser. Les threads sont des unités d'exécution permettant de réaliser plusieurs tâches en même temps. Plus il y en a, plus les temps d'entraînement sont réduits. L'option `workers=-1` utilisera tous les threads disponibles.

Le modèle est désormais prêt à l'emploi. Voici deux exemples consistant à demander les cinq mots les plus proches d'un vocable et la similarité entre deux termes :

```
# Trouver les 5 mots les plus similaires à...
word = 'king'

similar_words = model.wv.most_similar(word, topn=5)
print(f"Mots similaires à {word} :")
for word, score in similar_words:
    print(f"{word}: {score:.4f}")

# Output:
#Mots similaires à king :
#prince: 0.7391
#throne: 0.6975
```

```
#emperor: 0.6878
#kings: 0.6853
#queen: 0.6831
```

```
# Calcul de la similarité entre deux mots
sim_2w = model.wv.similarity('king', 'queen')

print(f"\nSimilarité entre 'king' et 'queen' : { sim_2w:.4f}")

# Output: Similarité entre 'king' et 'queen' : 0.6831
```

1.3.2 L'encodage des phrases

L'encodage de phrases entières ou de morceaux de texte constitue un pas supplémentaire dans la contextualisation en adoptant une approche globale. Cette méthode est plus dynamique que l'approche par mots, ce qui favorise un traitement plus précis du langage naturel, en tenant compte des interactions et du contexte global du texte.

Les algorithmes comme *Universal Sentence Encoder* (USE) ou Sent2Vec sont des algorithmes appliquant ce principe.

1.3.3 Transformers et modèles contextuels

Les transformers réunissent le meilleur des deux mondes en capturant à la fois des relations globales et locales. Cette approche valorise la contextualisation dynamique de chaque mot en tenant compte de sa position dans la phrase (local) et dans le texte complet (global). Grâce au mécanisme, chaque mot peut interagir avec tous les autres mots dans la phrase, permettant une compréhension approfondie des dépendances complexes en ouvrant la voie à des tâches telles que la génération de texte ou les réponses aux questions.

Les modules comme BERT ou GPT (*Generative Pre-trained Transformer*) reposent sur ce principe.

1.3.4 Les Larges Languages Models (LLM)

La technique des transformers a rendu possible la création des *Large Language Models* (LLM), qui sont, de fait, des transformers entraînés sur des milliards de textes. Grâce à cet entraînement massif, ces modèles acquièrent des capacités phénoménales qui leur permettent de comprendre et de générer du texte avec une précision et une fluidité impressionnantes. L'interaction avec un LLM passe par un prompt permettant de formuler des questions et de récupérer les réponses. Les applications potentielles des LLM sont immenses et pourraient entraîner des changements significatifs dans divers secteurs de la société.

Ces modèles, puissants et volumineux, peuvent être spécialisés grâce au transfer learning. Cette technique, également employée dans le traitement de l'image, consiste à récupérer un modèle préentraîné et à l'adapter à des tâches spécifiques en utilisant, séparément ou conjointement :

- Fine-tuning : ajustement d'un modèle préentraîné sur un ensemble de données spécifique pour améliorer ses performances sur une tâche particulière.
- RAG (*Retrieval-Augmented Generation*) : combinaison de génération de texte et de récupération d'informations. Le modèle utilise des documents externes pour enrichir ses réponses.

Attention toutefois car cela nécessite une grande puissance de calcul, souvent hors de portée des ordinateurs personnels.

2. La modélisation des images

Dans un monde inondé par les images, les besoins en termes de modélisation sont très importants. Comme pour le NLP, c'est une discipline à part entière en data science qui est très riche en termes de possibilités. Les opérations de modélisation sont communes aux autres domaines mais nécessitent un prétraitement qui peut s'avérer extrêmement complexe dans certains cas. En effet, pour extraire correctement l'information contenue dans les images, il est souvent nécessaire d'appliquer diverses transformations, telles que le redimensionnement, la conversion de formats, et l'application de filtres ou d'algorithmes spécifiques.

L'introduction des *Convolutional Neural Networks* (CNN) a marqué un tournant décisif dans ce domaine. Conceptualisés dans les années 80, les CNN ont connu un essor majeur à partir de 2012, révolutionnant le domaine de la vision par ordinateur. Ils automatisent une grande partie des étapes complexes de prétraitement des images, permettant ainsi d'extraire directement les caractéristiques à partir des données brutes. Cette automatisation simplifie considérablement le processus d'analyse des données visuelles et améliore significativement les performances de classification.

Néanmoins, tout le monde ne cherche pas forcément à automatiser complètement le traitement d'images et les modules classiques restent encore largement utilisés car elles permettent, dans certains domaines applicatifs, de gérer finement chaque étape de transformation.

Nous allons donc commencer par un tour d'horizon des modules Python dédiés au traitement d'images, en explorant ensuite leurs fonctionnalités pratiques à travers des exemples concrets. Enfin, nous aborderons les *Convolutional Neural Networks* (CNN), en montrant comment s'initier facilement à cette technologie.

2.1 Les solutions de Machine Learning destinées aux images

Parmi les nombreux modules dédiés à l'image, nous allons explorer trois solutions qui permettent de s'initier de manière progressive au traitement d'images : Pillow, Scikit-Image et OpenCV.

2.1.1 Pillow pour s'initier au prétraitement

Pillow est le module idéal pour s'initier à la manipulation des images. Il est simple à utiliser et possède toutes les fonctionnalités pour lire, créer, sauvegarder, convertir, redimensionner et appliquer divers filtres sur les images.

Lors de l'installation, il faut bien penser à mettre un P majuscule à Pillow :

```
pip install Pillow
```

La manipulation des images est ensuite très simple à effectuer.

Commençons par ouvrir et afficher une image :

```
from PIL import Image, ImageFilter

# Ouverture de l'image
image = Image.open(r"adresse_image\nom_image.jpg")

# Affichage
image.show(title="Image originale")
```

Remarque

Pillow est une extension de l'ancienne bibliothèque PIL. L'utilisation de PIL dans les imports permet de conserver la compatibilité avec les fonctionnalités existantes de PIL.

Convertissons-la en niveaux de gris :

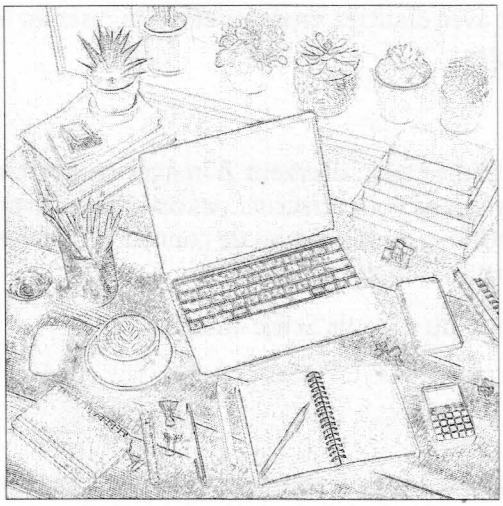
```
# Conversion en noir et blanc
bw_image = image.convert("L")

# Affichage
bw_image.show(title="Image Noir et Blanc")
```

Voici un exemple de retournement de l'image :

```
# Retournement horizontal de l'image
flipped_image = bw_image.transpose(Image.FLIP_LEFT_RIGHT)

# Affichage
flipped_image.show(title="Image Retournée")
```

Et un exemple d'application d'un filtre pour dessiner les contours :

```
# Application du filtre de contour
contour_image = flipped_image.filter(ImageFilter.CONTOUR)

# Affichage
contour_image.show(title="Image avec Filtre Contour")
```



Finalement, nous pouvons enregistrer notre image de cette façon :

```
# Sauvegarde
contour_image.save(r"adresse_image\nom_image_modifiee.jpg"
```

Lors du prétraitement des images, il est courant de les convertir en arrays Numpy pour faciliter les opérations de transformations. Cette conversion, comme toutes les autres actions, nécessite une ligne de code :

```
import numpy as np

# Conversion de l'image en un tableau NumPy
image_array = np.array(image)

# Affichage de la forme du tableau
print(image_array.shape)

# Output : (768, 768, 3)
```

Le code précédent affiche les caractéristiques de l'image originale sous la forme : (hauteur, largeur, nombre de canaux de couleurs). Dans notre cas, le nombre de canaux est de 3 car chaque pixel est défini par un tuple de trois valeurs : une pour le rouge, une pour le vert et une pour le bleu.

Pillow apporte beaucoup de simplicité dans la manipulation des images. Bien qu'il n'offre pas de fonctions de modélisations, il est en revanche tout à fait possible de l'utiliser conjointement avec d'autres modules et de lui réserver les étapes de transformations des images.

2.1.2 Scikit-image

Scikit-image est une bibliothèque dédiée au traitement d'images en Python. Elle propose un large éventail de fonctions pour effectuer des opérations avancées sur les images telles que la détection de formes ou de contours, l'analyse des caractéristiques ou la segmentation d'images.

Avant tout, procédons à l'installation du module Scikit-image :

```
pip install scikit-image
```

Remarque

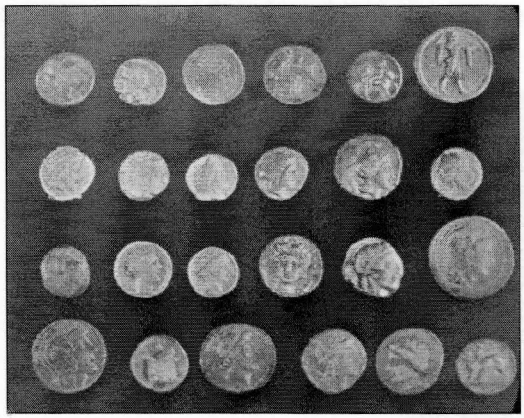
Le package est installé sous un dossier nommé skimage. C'est ce nom qu'il faut utiliser ensuite pour l'importer.

Pour évaluer les capacités de segmentation et de dénombrement de ce module, nous allons, à titre d'exemple, détailler étape par étape la méthode permettant de compter le nombre de pièces présentes sur une photo.

La première étape consiste à récupérer l'image des pièces faisant partie du module comme une image d'entraînement sous le nom de `data.coins` :

```
import numpy as np
import matplotlib.pyplot as plt

# Import des modules scikit-image nécessaires
from skimage import data, segmentation, filters, color,
morphology, graph, measure

# Chargement de l'image des pièces
image = data.coins()
```

```
plt.imshow(image, cmap='gray')
plt.axis('off')  # Commande pour masquer les axes
plt.show()
```



Dans un deuxième temps, nous allons créer un masque aux dimensions de l'image. Sachant que l'intensité d'un pixel dans une image en noir et blanc varie de 0 (pixel noir) à 255 (pixel blanc), nous allons appliquer la transformation suivante :

- Tous les pixels dont la valeur est inférieure à 100 seront labellisés 1.
- Tous les pixels dont la valeur est supérieure à 149 seront labellisés 2.

```
# Création d'une matrice de zéros aux dimensions de l'image
labels = np.zeros(image.shape, dtype=int)

# Opération de labellisation
labels[image < 100] = 1
labels[image > 150] = 2
```

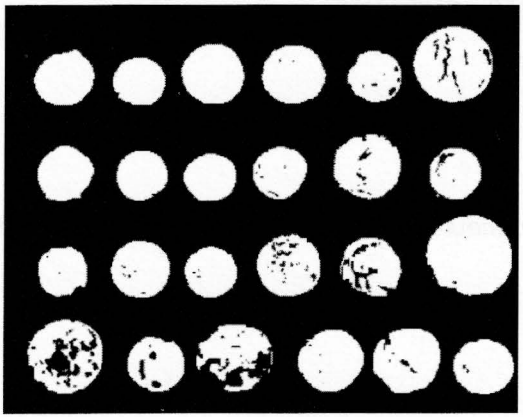
Remarque

Définir la valeur inférieure et supérieure n'est pas automatique et demande quelques tâtonnements.

À ce stade, la matrice contient des 1 pour les pixels sombres, des 2 pour les pixels clairs et des 0 pour les pixels dont la valeur oscille entre 100 et 149, représentant la zone d'indécision. Pour illustrer cette situation, le visuel suivant représente en noir les pixels sombres, en blanc les clairs et en gris ceux qui restent indécis :

Dans un troisième temps, l'algorithme de segmentation `random_walker` va être utilisé pour rattacher les pixels 0 soit au groupe des sombres soit au groupe des clairs en fonction de leur proximité et de la structure de l'image. Cette étape nécessite une seule ligne de code :

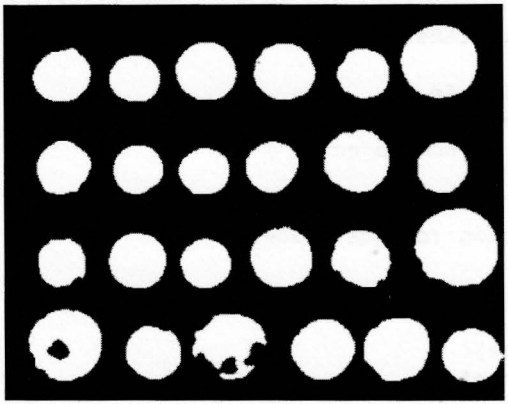
```
image_rw = segmentation.random_walker(image, labels)
```



La segmentation a relativement bien fonctionné mais certaines régions des pièces restent morcelées, ce qui peut perturber le dénombrement.

Une opération de traitement d'image appelée fermeture morphologique, qui dilate puis rétrécit les zones blanches (ici, les pixels blancs marqués par le chiffre 2), sera appliquée en quatrième étape pour combler les trous :

```
image_cleaned = morphology.binary_closing(
    image_rw == 2,
    morphology.disk(3) # Taille du disque de 3 pixels
).astype(int)
```



Remarque

Déterminer la taille du disque (3 dans notre cas) demande quelques tâtonnements.

À la suite de la fermeture morphologique des pixels blancs, il est conseillé d'appliquer également une fermeture morphologique sur les pixels noirs pour éviter la présence de pixels blancs isolés pouvant être considérés, à tort, comme des groupes. Nous allons cibler ici les pixels noirs mais garder la logique de la première fermeture en ajoutant un tilde ~ devant la fonction :

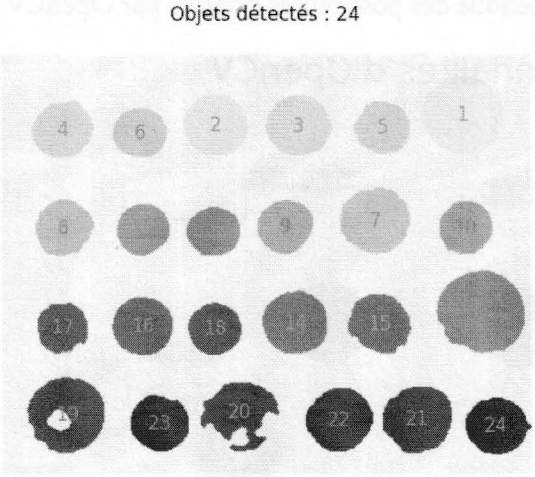
```
image_cleaned = ~morphology.binary_closing(
    image_cleaned == 0,
    morphology.disk(1) # Petit disque de 1 pixel
)
```

Remarque

Comme précédemment, quelques tâtonnements ont été nécessaires pour déterminer la valeur du disque à 1.

Enfin, à l'aide de la fonction `measure.label`, nous allons pouvoir identifier chaque pièce et, ce faisant, les dénombrer :

```
# Attribution d'un label aux pixels connectés (les pièces)
label_objects, num_objects = measure.label(image_cleaned,
return_num=True)

# Commande pour obtenir les propriétés des régions labellisées
regions = measure.regionprops(label_objects)

# Affichage de l'image labellisée
plt.imshow(label_objects, cmap='Blues')
plt.title(f"Objets détectés : {num_objects}")
plt.axis('off')

# Numérotation de chaque région (chaque pièce)
for region in regions:
    y, x = region.centroid  # Centre de chaque région
    plt.text(x, y,
             str(region.label),
             color='darkorange',
             fontsize=12,
             ha='center', va='center')

plt.show()
```



En définitive, chaque pièce est indexée et peut être localisée et mesurée sur l'image, ce qui ouvre de nombreuses possibilités. La procédure n'est certes pas instantanée et met en évidence la difficulté du prétraitement manuel des images, mais avec un peu de pratique, les possibilités sont immenses.

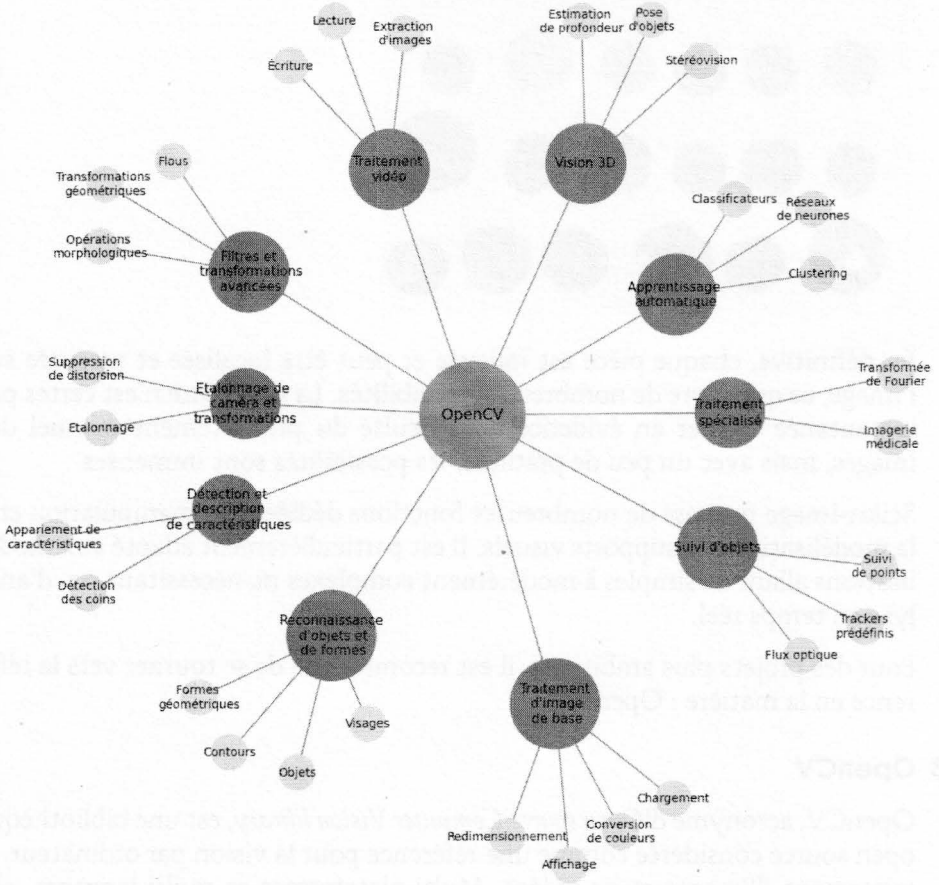
Scikit-image propose de nombreuses fonctions dédiées à la manipulation et à la modélisation des supports visuels. Il est particulièrement adapté aux modélisations allant de simples à modérément complexes ne nécessitant pas d'analyse en temps réel.

Pour des projets plus ambitieux, il est recommandé de se tourner vers la référence en la matière : OpenCV.

2.1.3 OpenCV

OpenCV, acronyme d'*Open source Computer Vision library*, est une bibliothèque open source considérée comme une référence pour la vision par ordinateur, le traitement d'images et de vidéos. Multi-plateformes et multi-langages, elle offre de très hautes performances et propose une multitude de fonctions.

Voici, pour s'en convaincre, l'étendue des possibilités offertes par OpenCV :



De nombreux sites sont consacrés au traitement de contenu visuel, mais un site se démarque particulièrement : PyImageSearch (https://pyimagesearch.com). Ce site aborde de manière didactique une grande variété de technologies liées à l'image. OpenCV y occupe une place importante, mais l'auteur explore également des modules tels que TensorFlow ou des matériels comme le Raspberry Pi afin de permettre aux utilisateurs d'approfondir leurs compétences dans ce domaine. Que ce soit pour améliorer des images, les classifier avec des réseaux de neurones, s'initier à la détection d'objets avec une webcam ou même résoudre les problèmes d'installation de certains modules, PyImageSearch propose des tutoriels clairs et détaillés pour tout public.

Nous allons justement nous inspirer de ce site et, dans la sous-section La modélisation des images, explorer trois solutions pratiques liées aux images.

Avant d'aborder ces solutions, nous allons commencer par installer OpenCV. Bien qu'il puisse être préinstallé sur certaines distributions de Python, il est recommandé de vérifier et, si nécessaire, de l'installer manuellement. Pour profiter des fonctionnalités étendues, nous utiliserons le module de base ainsi que `opencv-contrib-python`, qui inclut des modules supplémentaires. Installer les deux versions garantit le bon fonctionnement global en identifiant correctement le chemin du module ainsi que l'accès aux fonctions supplémentaires :

```
pip install opencv-python opencv-contrib-python
```

Précisons pour terminer qu'il faut utiliser l'alias `cv2` pour importer OpenCV :

```
import cv2
```

2.2 Méthodes de modélisation des images

Nous allons mettre en œuvre trois cas pratiques de modélisations liées à l'image pour illustrer différentes fonctionnalités possibles.

2.2.1 Segmenter

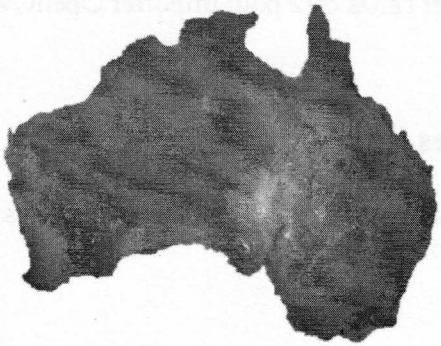
La segmentation consiste à regrouper des éléments qui sont proches. Dans le cas d'une image, nous pouvons regrouper par formes, couleurs ou textures, par exemple. Il peut aussi s'agir de créer des groupes de photos partageant des caractéristiques communes.

Pour illustrer un cas pratique de segmentation, nous allons étudier les types d'occupation du sol en Australie (déserts, végétations...). Pour ce faire, nous utiliserons une vue satellite détourée sur fond blanc disponible à l'adresse suivante :

https://github.com/eric2mangel/Images/blob/main/AUSTRALIE.png

Sitôt l'image récupérée, il faut remplacer `CHEMIN_DU_FICHIER` dans le code suivant par le chemin où l'image est stockée :

```
import cv2
import numpy as np

# Chargement de l'image
image = cv2.imread(r'CHEMIN_DU_FICHIER\AUSTRALIE.png')

# Affichage de l'image
cv2.imshow('Image Originale', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```



À partir de cette vue, il serait intéressant de pouvoir regrouper les couleurs entre elles dans le but de mettre en lumière les différents types d'occupation du sol et pouvoir les quantifier.

Avant d'appliquer le K-means pour regrouper les pixels par couleurs, nous allons préparer les données et exclure des calculs les pixels blancs autour du pays :

```
# Conversion de l'image en un tableau de données 2D (pixels)
pixels = image.reshape((-1, 3))

# Conservation des pixels non blancs (255, 255, 255)
non_white_mask = np.any(pixels != [255, 255, 255], axis=1)
non_white_pixels = pixels[non_white_mask]
```

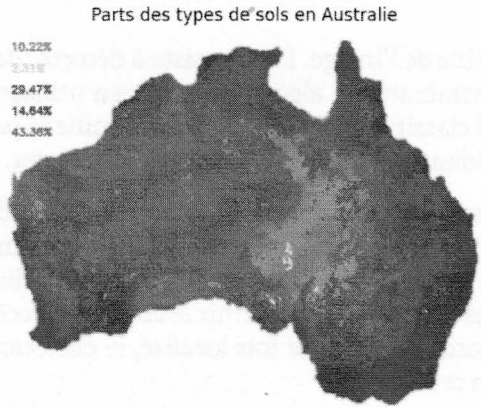
Nous allons pouvoir lancer le calcul du K-means. Attention car l'algorithme dédié d'OpenCV exige des données en `float32`. Il ne faut pas oublier cette conversion préalable sous peine de voir le programme planter :

```
# Conversion en float32 pour K-means
non_white_pixels = np.float32(non_white_pixels)

# Critères pour K-means (précision et nombre d'itérations max)
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
            100, 0.2)

# Nombre de clusters (segments) à utiliser pour la segmentation
k = 5  # A ajuster pour obtenir plus ou moins de segments

# Clustering K-means sur les pixels qui ne sont pas blancs
_, labels, centers = cv2.kmeans(non_white_pixels,
                                k, None, criteria,
                                10, cv2.KMEANS_RANDOM_CENTERS)

# Conversion des centroïdes en entiers
centers = np.uint8(centers)

# Chaque centroïde représente une couleur

# Remplacer chaque pixel non blanc par le centre correspondant
segmented_non_white_image = centers[labels.flatten()]

# Création d'une image blanche de la taille de l'image originale
segmented_image = np.full_like(pixels, [255, 255, 255])

# Grâce au masque, nous remplissons les valeurs des segments
segmented_image[non_white_mask] = segmented_non_white_image
```

```
# Retour aux dimensions de « image »
segmented_image = segmented_image.reshape(image.shape)
```

À ce stade, chaque pixel non blanc a été rattaché à son groupe et nous pouvons, après la mise en forme suivante afficher et quantifier chaque groupe :

```
# Affichage des pourcentages sur l'image segmentée
font = cv2.FONT_HERSHEY_SIMPLEX
for i, percent in enumerate(percentages):
    # Choix d'une position pour chaque pourcentage
    text_pos = (10, 30 + i * 30)

    # Texte à afficher
    text = f"{percent:.2f}%"

    # Taille du texte et la ligne de base
    (text_width, text_height), baseline = cv2.getTextSize(
        text, font, 0.5, 2
        )

    # Coordonnées du rectangle de fond ajustés selon le texte
    rect_start = (text_pos[0] - 5,
                  text_pos[1] - text_height - 5)
    rect_end = (text_pos[0] + text_width + 5,
                text_pos[1] + baseline)

    # Dessiner un rectangle blanc derrière le texte
    cv2.rectangle(segmented_image, rect_start,
                  rect_end, (255, 255, 255), thickness=-1)

    # Couleur du texte correspondant à la couleur du segment
    color = tuple(int(c) for c in centers[i])

    # Ajouter le texte sur l'image
    cv2.putText(segmented_image, text,
                text_pos, font, 0.5, color, 2)


# Affichage l'image segmentée avec pourcentages
cv2.imshow('Parts des types de sols en Australie',
           segmented_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```



Grâce au regroupement des couleurs des sols, la part de chaque couleur est quantifiée et nous pouvons en déduire les estimations suivantes. Nous préciserons entre parenthèses les valeurs trouvées après de rapides recherches :

- Un peu plus de 53 % de l'Australie est constituée de déserts dont 43,36 % ont un sol riche en oxydes de fer (couleur rouge) et 10,22 % ont un sol sableux ou argileux (proportions évoluant entre 18 et 45 % selon les sources).
- Les zones de végétations peu denses et de forêts représentent respectivement 29,47 % et 14,64 % (pas d'information précise trouvée sur la végétation mais les forêts représentent 17 % du territoire selon les sources officielles du pays : https://www.agriculture.gov.au/abares/forestsaustralia/australias-forests/profiles/australias-forests-2019).
- La présence de lacs de sel asséchés, en gris clair sur la carte, qui représentent environ 2,31 % du territoire (entre 1 et 2 % selon les sources).

Nos estimations, basées uniquement sur des regroupements de couleurs, sont certes imparfaites, mais elles permettent de fournir un ordre de grandeur cohérent et surtout d'offrir une information rapidement accessible qu'il est difficile d'obtenir autrement.

Cette segmentation est donc à adapter en fonction des pays et de la diversité de leurs sols. Mais avec relativement peu de lignes de codes, nous disposons d'une solution pour quantifier les sols des pays et étudier leur évolution dans le temps.

2.2.2 Détecter

La détection est un autre grand domaine de l'image. Elle consiste à détecter des formes, couleurs ou objets soit en entraînant un algorithme, soit en utilisant des modèles préentraînés comme des classifieurs en cascade, qui éliminent par étapes successives les zones ne possédant pas les formes qu'il peut détecter.

Ainsi, nous allons faire leur connaissance en utilisant la version capable de détecter et de situer les visages de face sur une image. Ce fichier, nommé `haarcascade_frontalface_default.xml`, est situé dans la bibliothèque d'OpenCV. Pour l'utiliser, il est nécessaire de fournir le chemin d'accès en effectuant une recherche sur son ordinateur. Une fois localisé, la détection de visages est très simple à mettre en œuvre.

Dans notre exemple, nous allons utiliser une photo de groupe générée par IA pour éviter tout problème de RGPD. Cette image est disponible à l'adresse suivante :

https://github.com/eric2mangel/Images/blob/main/Groupe.jpg

Remarque

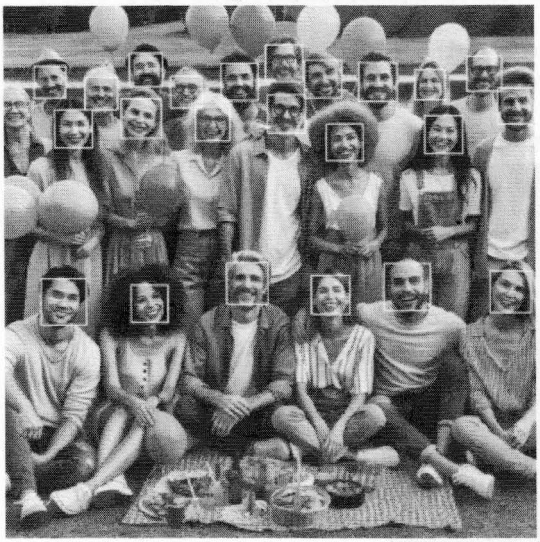
Les visages générés par IA présentent de grosses imperfections mais ce sera l'occasion de mettre à l'épreuve le classifieur. Des ballons ont été ajoutés pour tester s'ils ne dont pas confondus avec des visages.

Voici le code nécessaire à la détection de visages. Il est relativement court :

```
import cv2

# Chargement du classifieur
# Remplacer PathToFile par l'adresse du fichier
path_to_file = r'PathToFile\haarcascade_frontalface_default.xml'
face_cascade = cv2.CascadeClassifier(path_to_file)

# Lecture de l'image
# Remplacer PathToFile2 par l'adresse du fichier
image = cv2.imread(r'PathToFile2\Groupe.jpg')

# Transformation en noir et blanc
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Détection des visages
```

```
faces = face_cascade.detectMultiScale(gray,
                                      scaleFactor=1.1,
                                      minNeighbors=5,
                                      minSize=(30, 30))

print(f"{len(faces)} visages identifiés")

# Tracé des rectangles autour des visages détectés
for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x+w, y+h), (255, 255, 255), 2)

# Afficher l'image avec les visages détectés
cv2.imshow('Image', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

23 visages identifiés



L'exemple précédent est éloquent car il montre les limites de la détection : la seule personne dont le visage n'est pas détecté est celle dont le visage est coupé. En effet, comme il n'est pas total, le classifieur ne peut pas se mettre en place et jouer son rôle. Il faudra prendre de soin de s'assurer que les visages soient complets pour que la détection fonctionne correctement.

Ces classifieurs ouvrent la voie à de nombreuses possibilités de dénombrement, de localisation ou, par exemple, permettre de flouter automatiquement les personnes hormis la personne concernée. Il suffit de remplacer dans le code précédent la boucle « Tracé des rectangles autour des visages détectés » par les instructions suivantes :

```
# Index du visage à ne pas flouter
index_visage_a_garder = 0 # Index à modifier

# Parcours des visages détectés
for i, (x, y, w, h) in enumerate(faces):
    if i != index_visage_a_garder:
        # Extraction du visage
        face = image[y:y+h, x:x+w]

        # Application d'un flou gaussien
        blurred_face = cv2.GaussianBlur(face, (19, 19), 30)

        # Remplacement par le visage flouté
        image[y:y+h, x:x+w] = blurred_face
```

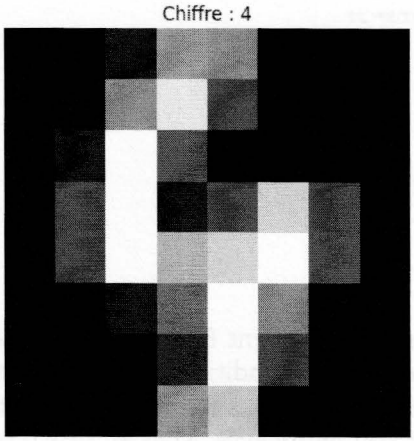
23 visages identifiés

2.2.3 Classifier

À la différence de la détection, la classification attribue une étiquette globale à l'image. Nous allons, pour terminer, aborder un exemple simple en nous appuyant sur le jeu des MNIST (*Modified National Institute of Standards and Technology*) qui est une célèbre base de données de chiffres manuscrits, entre 0 et 9, régulièrement mis en avant dans des exemples de reconnaissance de caractères.

Les 1797 chiffres manuscrits sont directement accessibles de la façon suivante :

```
from sklearn import datasets
import matplotlib.pyplot as plt

# Chargement du dataset MNIST
digits = datasets.load_digits()
X = digits.images
y = digits.target

plt.imshow(X[111], cmap='gray') #111 est un index aléatoire
plt.title(f'Chiffre : {y[111]}')
plt.axis('off')
plt.show()
```



Nous allons maintenant entraîner un algorithme à reconnaître les chiffres. Il faut juste penser à mettre à plat les pixels de chaque image avant de lancer l'entraînement. L'opération nécessite peu de lignes de codes et affiche un score quasiment parfait (98,67 %) :

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Redimensionnement des images (les pixels sont mis en ligne)
n_samples = len(X)
X = X.reshape((n_samples, -1))
print(X.shape)

# Train test
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.5,
    random_state=42
)

# Entraînement d'un KNN
# 3 voisins donnait les meilleurs scores
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Prédictions
y_pred = knn.predict(X_test)

# Calcul et affichage de la précision
accuracy = accuracy_score(y_test, y_pred)
print(f"Précision : {accuracy * 100:.2f}%")

# Output: Précision : 98.67%
```

Les performances élevées et la rapidité du calcul doivent être relativisées car les images de MNIST mesurent 8 pixels de côté. En conditions réelles, sur des images d'au moins 32 pixels de côté, voire beaucoup plus selon les besoins en termes de précision, les enjeux seraient un peu plus élevés. Mais au prix d'un jeu d'entraînement suffisamment important, la reconnaissance de caractères est à la portée du plus grand nombre et peut être personnalisée à notre propre graphie.

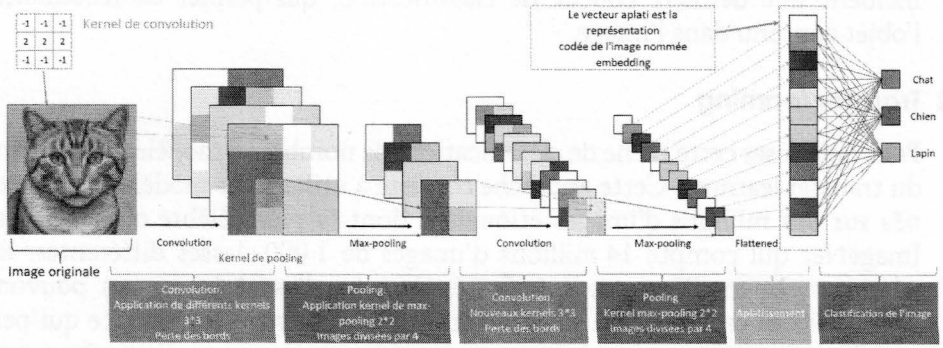
À travers ces trois exemples, nous ne découvrons qu'une infime partie des possibilités offertes concernant les images. Nous encourageons tout un chacun à expérimenter différentes approches et pratiquer car les réglages sont multiples et souvent subtils. Nous allons justement terminer ce chapitre sur les réseaux de neurones à convolution qui ont révolutionné et grandement simplifié la reconnaissance et la classification des images.

2.3 Aller plus loin avec les CNN

2.3.1 Principe de fonctionnement du CNN

Les réseaux de neurones à convolution (CNN) sont des modèles d'intelligence artificielle qui isolent automatiquement les caractéristiques des images. Avant leur apparition, il était nécessaire de les extraire manuellement à travers des méthodes comme les invariants, un processus souvent fastidieux et aux résultats limités que nous laisserons volontairement de côté. Aujourd'hui, les CNN simplifient cette tâche en « balayant » automatiquement les images à l'aide de petits filtres, appelés kernels ou noyaux, qui repèrent des éléments importants tels que les formes, les couleurs ou les motifs. Lors de l'entraînement, les poids des kernels sont ajustés progressivement par rétropropagation, améliorant progressivement les performances de classification.

Pour bien comprendre le CNN, voici un schéma illustrant son fonctionnement que nous allons commenter :



Au début, chaque image est présentée au réseau quasiment sous sa forme brute. Le seul prétraitement appliqué consiste à redimensionner l'image et à normaliser les valeurs des pixels entre 0 et 1, afin d'uniformiser les données et faciliter l'exécution.

Les kernels de **convolution**, généralement des carrés de 3 ou 5 pixels de côté, se déplacent ensuite progressivement sur chaque image, de gauche à droite et de haut en bas, générant au fur et à mesure des ensembles d'images transformées qui sont très variées et souvent conceptuelles. Le nombre de convolutions appliquées, ainsi que le nombre d'images résultantes, peut varier considérablement et résulte souvent d'un processus d'expérimentation.

Après les convolutions, intervient une étape clé : le **pooling**. Cette opération consiste à réduire la taille des images tout en conservant les informations essentielles. L'option `max-pooling` est la plus couramment utilisée. Basée généralement sur un carré de 2 pixels de côté, elle va récupérer la valeur maximale des quatre. Au terme de son déplacement sur toute l'image, une image quatre fois plus petite en résulte.

Le nombre d'images transformées va ainsi aller croissant tout en étant toujours plus petites. Après ces phases successives de convolution-pooling, l'ensemble des petites images vont être aplaties en un vecteur colonne unique.

Ce vecteur est en fait la représentation codée de l'image, appelée **embedding**. Il peut être récupéré directement comme une version utilisable de l'image pour tout type d'opération de modélisation. Cependant, la plupart des CNN incluent une dernière couche de classification, qui permet de reconnaître l'objet contenu dans l'image.

2.3.2 Transfer learning

Pour optimiser cette tâche de classification, de nombreux modèles tirent parti du transfer learning. Cette approche consiste à utiliser des modèles préentraînés sur des millions d'images étiquetées dont la plus célèbre collection est ImageNet qui compte 14 millions d'images de 1 000 classes différentes. En adaptant ces modèles aux spécificités de nouvelles tâches, nous pouvons bénéficier des caractéristiques déjà apprises (textures ou formes), ce qui permet d'améliorer les performances tout en réduisant le temps d'entraînement.

Ces modèles sont accessibles gratuitement sur des modules comme TensorFlow ou Keras. Nous pouvons les utiliser tels quels ou les modifier afin de les spécialiser pour une tâche particulière. Nous n'aborderons pas le réentraînement d'un modèle, car cela serait trop ambitieux par rapport au cadre de l'initiation aux CNN. En revanche, nous allons apprendre à les acquérir et à les utiliser pour effectuer des classifications d'images, en profitant de leurs fonctionnalités déjà intégrées.

2.3.3 Initiation à Tensorflow et Keras

Deux modules liés sont incontournables dès lors que nous travaillons avec les CNN : Tensorflow et Keras.

TensorFlow est une bibliothèque open source développée par Google pour l'apprentissage automatique. C'est l'un des outils les plus utilisés en IA pour construire et entraîner des modèles de deep learning. Keras, quant à elle, est une API de haut niveau intégrée à TensorFlow qui facilite la création et l'entraînement de modèles de réseaux de neurones.

Dans la mesure où Keras est intégrée à Tensorflow depuis la version 2, la commande suivante permet d'installer les deux d'un seul coup :

```
pip install tensorflow
```

Assurons-nous que l'installation a bien eu lieu à l'aide de la commande suivante :

```
import tensorflow as tf
print(tf.__version__)

# Output: 2.17.0
```

La liste et les spécificités des CNN disponibles sont sur le site de Keras à l'adresse URL suivante :

https://keras.io/api/applications/

Voici un visuel du début de cette liste résumant les caractéristiques de chaque modèle. En cliquant sur chacun des noms s'ouvre une page fournissant toutes les informations nécessaires pour l'utilisation du CNN :

Available models

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	79.0%	94.5%	22.9M	81	109.4	8.1
VGG16	528	71.3%	90.1%	138.4M	16	69.5	4.2
VGG19	549	71.3%	90.0%	143.7M	19	84.8	4.4
ResNet50	98	74.9%	92.1%	25.6M	107	58.2	4.6
ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
ResNet101	171	76.4%	92.8%	44.7M	209	89.6	5.2
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
ResNet152	232	76.6%	93.1%	60.4M	311	127.4	6.5
ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6
InceptionV3	92	77.9%	93.7%	23.9M	189	42.2	6.9
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0
MobileNet	16	70.4%	89.5%	4.3M	55	22.6	3.4
MobileNetV2	14	71.3%	90.1%	3.5M	105	25.9	3.8
DenseNet121	33	75.0%	92.3%	8.1M	242	77.1	5.4
DenseNet169	57	76.2%	93.2%	14.3M	338	96.4	6.3
DenseNet201	80	77.3%	93.6%	20.2M	402	127.2	6.7

Nous allons justement terminer ce chapitre sur des exemples très simples d'utilisation des CNN.

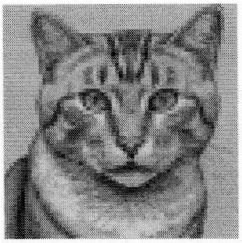
2.3.4 Exemples d'utilisation des CNN

Les exemples suivants illustrent la facilité avec laquelle nous pouvons utiliser des réseaux de neurones. Il existe cependant un versant plus complexe, impliquant la récupération, la modification et le réentraînement d'un CNN. Ce processus viendra naturellement avec la pratique et l'évolution de nos besoins. À ce stade, voici quelques exemples pratiques.

Acquisition et identification

Commençons par un usage basique : récupérer un CNN et l'utiliser pour décrire une image. L'image de chat de cet exemple est disponible ici :

https://github.com/eric2mangel/Images/blob/main/Chat.png



Importons les modules et le CNN, MobileNet pour cet exemple :

```
import tensorflow as tf
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.mobilenet import MobileNet,
preprocess_input, decode_predictions
import numpy as np

# Chargement du modèle préentraîné
model = MobileNet(weights='imagenet')
```

Après l'affichage d'une barre de progression, le modèle est prêt à l'emploi. Nous allons récupérer l'image et la prétraiter :

```
# Indiquer le chemin d'accès de l'image
img_path = r'YOUR_PATH\chat.png'

# Prétraitement
img = image.load_img(img_path, target_size=(224, 224))
img_array = image.img_to_array(img)
img_array = np.expand_dims(img_array, axis=0)
img_array = preprocess_input(img_array)
```

La prédiction ne nécessite ensuite que quelques lignes de code :

```
# Prédiction
predictions = model.predict(img_array)
predictions_ok = decode_predictions(predictions, top=3)[0]
```

```
# Affichage des résultats
for i, (imagenet_id, label, score) in enumerate(predictions_ok):
    print(f"{i + 1}: {label} ({score:.2f})")

# Output:
# 1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 567 ms/step
# 1: tabby (0.66)
# 2: Egyptian_cat (0.14)
# 3: lynx (0.09)
```

Le modèle estime qu'il existe 66 % de chances qu'il s'agisse d'un Tabby (qui est une race de chat), 14 % de chances que ce soit un chat égyptien et 9 % de chances que ce soit un lynx.

Remarque

Les résultats seront très différents en fonction du modèle utilisé. Pour le VGG16, la prédiction la plus fiable, avec 11 % de certitude, indique qu'il s'agit d'un bonnet de bain... Attention donc à bien tester avant d'utiliser.

Création des embeddings pour tout un dossier d'images

Nous allons à présent travailler sur un dossier d'images. Ce dossier, « Pictures », est disponible à l'adresse suivante :

https://github.com/eric2mangel/Images/tree/main/Pictures

Une fois les images enregistrées en local, nous allons commencer par déclarer notre CNN et le chemin d'accès aux images :

```
import os
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.mobilenet import MobileNet,
preprocess_input

# Chargement du modèle préentraîné
model = MobileNet(weights='imagenet')

# Adresse du dossier contenant toutes les images
images_folder = r'PATH_TO_FOLDER\Pictures'
```

Nous allons ensuite charger et prétraiter chaque image du dossier :

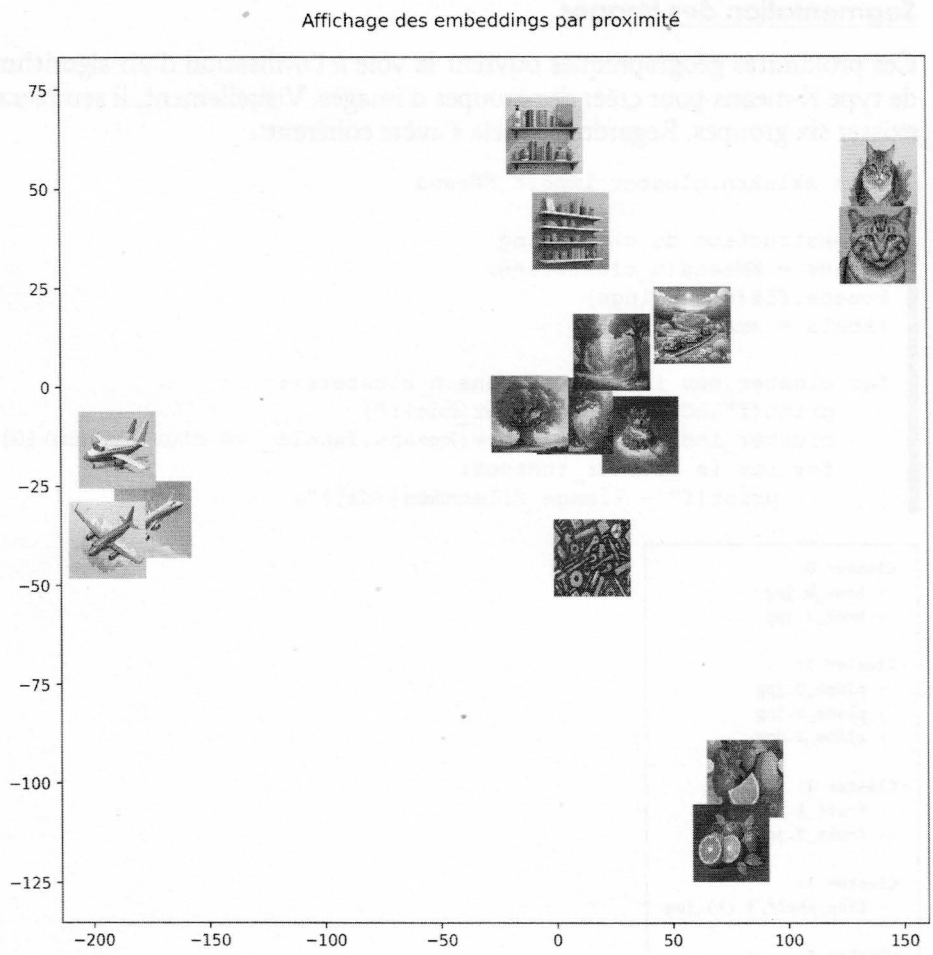
```
# Charger et prétraiter toutes les images
image_arrays = []
image_filenames = []
images_database = []

# Boucle pour chaque image du dossier
for img_file in os.listdir(images_folder):
    img_path = os.path.join(images_folder, img_file)

    # Chargement et redimensionnement
    img = image.load_img(img_path, target_size=(224, 224))
    img_array = image.img_to_array(img)

    # Normalisation de l'image pour l'affichage
    image_arrays.append(img_array / 255.0)

    # Prétraitement pour MobileNet
    img_array_preprocessed = preprocess_input(
        np.expand_dims(img_array, axis=0)
    )

    # Ajout à la base de données
    images_database.append(img_array_preprocessed)

    # Ajout du nom du fichier image
    image_filenames.append(img_file)

# Convertir la liste des images prétraitées en un tableau numpy
images_database = np.vstack(images_database)

# Prédiction pour chaque image
embeddings = model.predict(images_database)
```

À ce stade, toutes les images ont été encodées et sont désormais utilisables pour n'importe quel type de modélisation.

Visualisation des proximités entre images

À présent que les images sont vectorisées, il devient relativement aisé de les représenter en les projetant dans un espace à deux dimensions pour évaluer si l'encodage était pertinent ou non. Nous utiliserons pour ce faire l'algorithme de T-SNE :

```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
from matplotlib.offsetbox import OffsetImage, AnnotationBbox

# Réduire la dimensionnalité avec t-SNE
tsne = TSNE(n_components=2, perplexity=3)
embeddings_2d = tsne.fit_transform(embeddings)

# Création de la figure
plt.figure(figsize=(10, 10))

# Fonction pour ajouter des miniatures sur le graphique
def imscatter(x, y, images, zoom=0.22):
    for i in range(len(images)):
        image = OffsetImage(images[i], zoom=zoom)
        ab = AnnotationBbox(image, (x[i], y[i]), frameon=False)
        plt.gca().add_artist(ab)

# Affichage des miniatures des images
imscatter(embeddings_2d[:, 0], embeddings_2d[:, 1], image_arrays)

# Configurer les axes et le titre
plt.title('Affichage des embeddings par proximité\n')
plt.xlim(np.min(embeddings_2d[:, 0]) - 20,
         np.max(embeddings_2d[:, 0]) + 20)
plt.ylim(np.min(embeddings_2d[:, 1]) - 20,
         np.max(embeddings_2d[:, 1]) + 20)
plt.show()
```



L'encodage des images s'avère plutôt performant. Les images les plus proches le sont également sur le plan géographique. Il serait ainsi possible de réorganiser un dossier de photos non triées tout en acquérant une bonne vision d'ensemble.

Segmentation des images

Ces proximités géographiques ouvrent la voie à l'utilisation d'un algorithme de type K-means pour créer des groupes d'images. Visuellement, il semblerait exister six groupes. Regardons si cela s'avère cohérent :

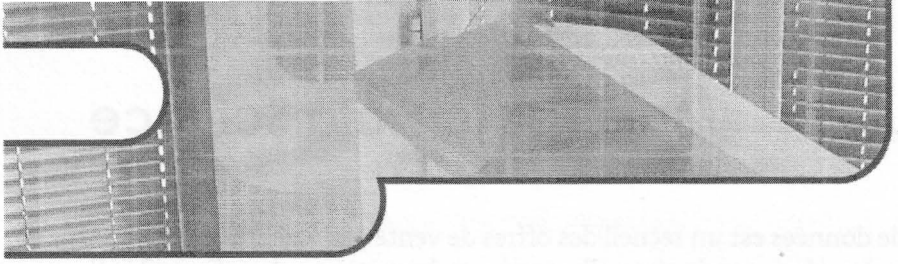
```
from sklearn.cluster import KMeans

# Construction du clustering
kmeans = KMeans(n_clusters=6)
kmeans.fit(embeddings)
labels = kmeans.labels_

for cluster_num in range(kmeans.n_clusters):
    print(f"\nCluster {cluster_num}:")
    cluster_indices = np.where(kmeans.labels_ == cluster_num)[0]
    for idx in cluster_indices:
        print(f" - {image_filenames[idx]}")
```

```
Cluster 0:
 - tree_0.jpg
 - tree_3.jpg

Cluster 1:
 - plane_0.jpg
 - plane_1.jpg
 - plane_2.jpg

Cluster 2:
 - fruit_1.jpg
 - fruit_3.jpg

Cluster 3:
 - tidy_shelf_3 (1).jpg

Cluster 4:
 - cat_0.jpg
 - cat_3.jpg

Cluster 5:
 - island_1.jpg
 - tidy_shelf_3 (4).jpg
 - tools_2.jpg
 - trafic3.jpg
 - tree_1.jpg
```

Hormis le Cluster 5 qui rassemble des images plus disparates, les arbres, chats, fruits ou avions sont naturellement regroupés ensemble validant ainsi la pertinence de l'encodage.

Ces quelques exemples présentent des moyens d'action intéressants mais avec la pratique du réentraînement des CNN pour les spécialiser sur un domaine particulier, les capacités de classification deviennent vite époustouflantes pour peu que nous ayons suffisamment d'exemples à fournir. La donnée demeure ce qu'il y a de plus précieux en data science, quel que soit le domaine. Sans elle, toutes les techniques, même les plus sophistiquées, sont sans effet.

Ce chapitre s'achève sur ces deux mondes vastes et riches en possibilités qu'est l'étude du langage et de l'image, qui, malgré les avancées significatives des dernières années, offrent encore de nombreuses opportunités à explorer.



Chapitre 10
Mener un projet de data science avec Python

1. Introduction

Ce chapitre offre l'occasion de mettre en pratique sur un projet réel une grande partie des concepts et techniques abordés dans cet ouvrage. Les notebooks de ce projet sont disponibles à l'adresse suivante :

https://github.com/eric2mangel/UsedCarPricePredictor

Afin de rendre la démonstration plus fluide, nous ne reprendrons que les parties les plus importantes. L'intégralité des expérimentations sera toutefois disponible dans les notebooks.

2. Le sujet : déterminer le prix des véhicules d'occasion

2.1 Les données

Les données du sujet sont issues du site Kaggle et disponibles à cette adresse :

https://www.kaggle.com/datasets/wspirat/germany-used-cars-dataset-2023

La licence est « CC0 : Domain Public », ce qui signifie que les données sont placées dans le domaine public et peuvent être librement utilisées. Attention à toujours bien vérifier et respecter les conditions d'utilisation d'un dataset.

Notre jeu de données est un recueil des offres de ventes de voitures d'occasion issues d'un des plus grands sites allemands en la matière : AutoScout24. Il offre un large éventail de modèles et de versions de véhicules. En tout, il y a 251 079 véhicules.

2.2 Les étapes du projet

Nous allons mettre en pratique une démarche classique de traitement et de modélisation des données en prenant soin de séparer la partie feature engineering de la partie modélisation. Cette approche est pertinente car elle nous permet de nous consacrer à chaque poste indépendamment. De plus, lors de la modélisation, nous pouvons repartir des données transformées et nous éviter de devoir recharger tout depuis le début.

2.2.1 Le notebook de l'EDA

Le notebook de l'EDA va couvrir l'acquisition, le nettoyage, l'exploration, l'analyse et la finalisation des données. Notre jeu de données ne présente que 15 variables mais certaines comme la marque, le modèle ou la version possèdent énormément de modalités. Nous sommes donc dans un bon entre-deux en termes de complexité. Ce sera l'occasion de découvrir en pratique que certaines phases, comme la remise en forme de certains contenus, ne sont pas toujours triviales.

Le notebook sera annoté et structuré pour aider les utilisateurs à bien comprendre et se repérer. À la fin de ce notebook, nous enregistrerons les données enrichies dans un nouveau fichier adapté (Parquet en l'occurrence). Cela nous permettra de repartir de ce support pour le notebook suivant.

2.2.2 Le notebook de modélisation

Le notebook de modélisation couvrira toutes les phases de préparation des données, d'expérimentations et d'évaluation des résultats. Nous verrons, à la fin, comment exporter l'algorithme pour pouvoir le réutiliser dans le cadre d'une API, par exemple.

2.2.3 Les aléas des données

L'exemple pratique devait porter, à l'origine, sur les prix des locations Airbnb à Paris lors des Jeux olympiques de 2024. Malheureusement, la variabilité des tarifs proposés était tellement importante que les modélisations n'ont pas pu aboutir. Elles enregistraient des coefficients de détermination autour de 0,4, ce qui était largement insuffisant pour une démonstration pertinente. Mais cette situation permet de mettre en lumière le fait que tout ne peut pas être modélisable. Dans le cas présent, les variables disponibles étaient insuffisantes pour retrouver le prix réel de chaque logement. Cette capacité à trouver des variables d'intérêt pour chaque situation est d'ailleurs précieuse et nous encourageons tout un chacun à développer cette capacité.

3. La modélisation en pratique

3.1 Notebook 1 : EDA

3.1.1 Acquisition et premiers contrôles des données

Avant de débuter, il est nécessaire de récupérer les données dont le lien a été fourni en début de chapitre et de les enregistrer dans un dossier de notre choix.

Procédons maintenant à l'acquisition :

```
import pandas as pd
import numpy as np

pd.set_option("display.max_columns", 999)
raw = pd.read_csv(r"YOUR_PATH\data.csv",low_memory=False,sep=",")

brut.head()
```

	Unnamed: 0	brand	model	color	registration_date	year	price_in_euro	power_kw	power_ps	transmission_type	fuel_type	fuel_consumption_l_100km
0	0	alfa-romeo	Alfa Romeo GTV	red	10/1995	1995	1300	148	201	Manual	Petrol	10,9 l/100 km
1	1	alfa-romeo	Alfa Romeo 164	black	02/1995	1995	24900	191	260	Manual	Petrol	NaN
2	2	alfa-romeo	Alfa Romeo Spider	black	02/1995	1995	5900	110	150	Unknown	Petrol	NaN
3	3	alfa-romeo	Alfa Romeo Spider	black	07/1995	1995	4900	110	150	Manual	Petrol	9,5 l/100 km
4	4	alfa-romeo	Alfa Romeo 164	red	11/1996	1996	17950	132	179	Manual	Petrol	7,2 l/100 km

fuel_consumption_g_km	mileage_in_km	offer_description
260 g/km	160500.0	2.0 V6 TB
- (g/km)	190000.0	Q4 Allrad, 3.2L GTA
- (g/km)	129000.0	ALFA ROME 916
225 g/km	189500.0	2.0 16V Twin Spark L
- (g/km)	96127.0	3.0i Super V6, absoluter Topzustand !

Le fichier, présenté en deux parties pour des contraintes de lisibilité, comporte 15 variables et 251 079 observations.

La première variable est un index incrémentiel qui peut être immédiatement supprimé :

```
raw = raw.drop(["Unnamed: 0"],axis=1)
```

Ensuite, nous renommerons la variable `offer_description` en `version` pour des raisons de commodités :

```
raw.rename(columns={'offer_description': 'version'},
           inplace=True)
```

Nous pouvons désormais prendre connaissance des caractéristiques globales du dataset :

```
raw.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 251079 entries, 0 to 251078
Data columns (total 14 columns):
 #   Column                    Non-Null Count   Dtype
---  ------                    --------------   -----
 0   brand                     251079 non-null  object
 1   model                     251079 non-null  object
 2   color                     250913 non-null  object
 3   registration_date         251075 non-null  object
 4   year                      251079 non-null  object
 5   price_in_euro             251079 non-null  object
 6   power_kw                  250945 non-null  object
 7   power_ps                  250950 non-null  object
 8   transmission_type         251079 non-null  object
 9   fuel_type                 251079 non-null  object
 10  fuel_consumption_l_100km  224206 non-null  object
 11  fuel_consumption_g_km     251079 non-null  object
 12  mileage_in_km             250927 non-null  float64
 13  version                   251078 non-null  object
dtypes: float64(1), object(13)
memory usage: 26.8+ MB
```

La qualité de remplissage est globalement très bonne avec un taux de complétion global moyen de 99,22 %. La variable `fuel_comsumption_l_100km` a le taux de remplissage le plus bas avec 89,3 %.

En parcourant les données, certains prix sont remplacés par un nom de société. Dans la mesure où notre modélisation va porter sur la prédiction des prix, nous pouvons dès maintenant supprimer toutes les observations n'ayant pas de prix numérique. Nous allons récupérer dans une nouvelle variable `price` les valeurs numériques et quantifier le nombre d'observations non numériques :

```
raw['price'] = pd.to_numeric(raw["price_in_euro"],
                             errors='coerce')
non_num_price = raw[raw['price'].isna() &
                    raw["price_in_euro"].notna()]
non_num_price.shape

#Output: (199,15)
```

Nous disposons de 199 observations ne possédant pas un prix numérique. Ces observations vont être supprimées ainsi que l'ancienne variable `price_in_euro` :

```
raw = raw.drop(list(non_num_price.index),axis=0)
raw = raw.drop(["price_in_euro"],axis=1)
```

Nous pouvons désormais passer au nettoyage.

3.1.2 Nettoyage des données

Le poste qui va nécessiter le plus d'attention concerne le type des variables car seul le kilométrage (`mileage_in_km`) et `price` sont en numérique alors que des variables comme `year`, les `power` ou les `fuel_comsumption` qui devraient l'être, sont pour l'instant en type `object`.

Gestion de power_kw et power_ps

Avant de réaliser les contrôles, consulter le dictionnaire des variables disponible sur le site nous permet de comprendre que les variables `power_kw` et `power_ps` sont équivalentes, mais avec un coefficient de conversion.

Pour des raisons de familiarisation, nous conserverons `power_ps` qui exprime la puissance en chevaux plutôt que son équivalent en kilowatts.

```
raw = raw.drop(["power_kw"],axis=1)
```

Gestion de fuel_consumption_g_km et fuel_consumption_l_100km

Concernant les variables `fuel_comsumption`, nous allons tester si l'un est rempli et pas l'autre et vice-versa. Notre test indique qu'il n'y a pas de cas où `fuel_consumption_g_km` est vide et `fuel_consumption_l_100km` ne l'est pas. En revanche, il y a 26873 observations en sens inverse :

```
missing_cons1 = raw[raw['fuel_consumption_l_100km'].isna() &
                    raw['fuel_consumption_g_km'].notna()]
missing_cons1
```

	brand	model	color	registration_date	year	power_ps	transmission_type	fuel_type	fuel_consumption_l_100km	fuel_consumption_g_km	mileage_in_
1	alfa-romeo	Alfa Romeo 164	black	02/1995	1995	260	Manual	Petrol	NaN	- (g/km)	19000
2	alfa-romeo	Alfa Romeo Spider	black	02/1995	1995	150	Unknown	Petrol	NaN	- (g/km)	12900
18	alfa-romeo	Alfa Romeo GTV	red	03/1997	1997	201	Manual	Petrol	NaN	- (g/km)	16879
22	alfa-romeo	Alfa Romeo Spider	black	07/1997	1997	192	Manual	Petrol	NaN	- (g/km)	19400
27	alfa-romeo	Alfa Romeo	black	09/1997	1997	207	Manual	Petrol	NaN	- (g/km)	7226
...	...	...	...	...	...	...	...	...	...	...	
251048	volvo	Volvo C40	black	01/2023	2023	231	Automatic	Electric		0 g/km	210
251056	volvo	Volvo C40	black	05/2023	2023	231	Automatic	Electric	NaN	400 km Reichweite	300
251074	volvo	Volvo XC40	white	04/2023	2023	261	Automatic	Hybrid	NaN	43 km Reichweite	122
251077	volvo	Volvo XC40	white	05/2023	2023	179	Automatic	Hybrid	NaN	45 km Reichweite	150
251078	volvo	Volvo XC40	gold	03/2023	2023	218	Automatic	Electric	NaN	438 km Reichweite	5

26873 rows × 13 columns

Avec ce contrôle, hormis les cas de non-remplissage, nous découvrons de nouvelles subtilités du jeu de données :

- Pour certains véhicules, la valeur d'émissions de grammes au kilomètre est indiquée à 0, ce qui semble logique pour des véhicules électriques.
- Pour d'autres véhicules, il est indiqué l'autonomie en mode électrique pour cette même colonne grâce au mot `Reichweite` qui signifie, après recherches, « rayon d'action » en français.

Le contenu de la variable `fuel_consumption_g_km` est donc mixte, il va falloir remédier au problème.

Regardons combien de mentions contenant `Reichweite` il existe dans ces variables :

```
# Liste des colonnes à vérifier
cols_to_check = ['fuel_consumption_l_100km',
                 'fuel_consumption_g_km']

# Comptage des occurrences de "(Ort)" dans les colonnes spécifiées
count_ort = raw[cols_to_check].\
apply(lambda x: x.str.contains(r'Reichweite',
na=False).sum()).sum()
```

```
print(count_ort)

# Output: 4324
```

Nous trouvons 4324 mentions, ce qui pourrait être intéressant pour une modélisation dédiée aux véhicules électriques, mais cela restera trop marginal sur l'ensemble des motorisations donc les mentions seront écartées.

Nous allons donc écrire « 0 g/km » dans `fuel_consumption_g_km` pour les voitures électriques :

```
raw.loc[raw['fuel_type'] == "Electric", 'fuel_consumption_g_km'] =
"0 g/km"
```

Pour les voitures hybrides qui contenaient des valeurs comportant `Reichweite`, nous remettrons la valeur de `fuel_consumption_g_km` en valeur manquante.

```
raw.loc[(raw['fuel_type'].str.contains('Hybrid', na=False)) &
        (raw['fuel_consumption_g_km'].str.\
         contains('Reichweite', na=False)),
        'fuel_consumption_g_km'] = np.nan
```

À l'issue de ces corrections, il reste 49 véhicules avec la mention `Reichweite` mais non répertoriés en hybride ou électrique. Nous les écarterons pour éviter des recherches trop chronophages.

```
raw = raw.loc[~raw["fuel_consumption_g_km"].str.\
              contains('Reichweite', na=False)]
```

Avant de procéder à la transformation numérique de la variable, vérifions s'il reste des mentions ne contenant pas g/km :

```
lig_wo_gkm = raw.loc[(~raw["fuel_consumption_g_km"].str.\
              contains('g/km', na=False)) &
                    (raw["fuel_consumption_g_km"].notna())]

lig_wo_gkm["fuel_consumption_g_km"].value_counts()

# Output:
# -/-              882
# 44 km (Ort)        1
```

Ce contrôle nous informe de l'existence d'une autre information : 44 km (ORT) qui, après une rapide vérification, s'avère être l'autonomie électrique en cycle urbain. Avant de continuer, contrôlons la fréquence de ces mentions dans les variables concernées :

```
# Liste des colonnes à vérifier
cols_to_check = ['fuel_consumption_l_100km',
                 'fuel_consumption_g_km']

# Comptage des occurrences de "(Ort)" dans les colonnes spécifiées
count_ort = raw[cols_to_check].\
apply(lambda x: x.str.contains(r'\(Ort\)', na=False).sum()).sum()

print(count_ort)

# Output: 483
```

La faible occurrence nous pousse à ne pas créer une variable pour cette information.

Nous allons pouvoir procéder au nettoyage de la variable `fuel_consumption_g_km` grâce au recours à une fonction dédiée qui va gérer de façon précise la transformation du contenu :

```
import re
def clean_variable(df, col_name, new_col_name, replacements,
suffix='g/km'):
    # Création d'une regex à partir des remplacements
    replacement_regex = '|'.join(map(re.escape, replacements))

    # Nettoyage de la colonne
    df[col_name] = (
        df[col_name]
        .str.replace(replacement_regex, '', regex=True)
        .str.replace(',', '.')  # « , » remplacées par des « . »
        .where(df[col_name].str.endswith(suffix), '')
        .replace('', np.nan)  # Chaînes vides remplacées par NaN
        .astype(float)  # Conversion en float
    )

    # Conversion en numérique avec gestion des erreurs
    df[new_col_name] = pd.to_numeric(df[col_name],
                                     errors='coerce')
    # Suppression de l'ancienne colonne
```

```
    df = df.drop(col_name, axis=1)

    return df
```

Ainsi, pour les variables intégrant l'échelle pour chaque observation, il est possible de préciser cette échelle pour la remplacer ainsi que toutes les mentions hors sujet par rapport au contenu de la variable. À l'issue des remplacements, les virgules sont remplacées par des points pour que le contenu décimal soit bien considéré comme tel. Voici la commande et les paramètres à lancer pour la variable `fuel_consumption_g_km` :

```
replacements = ['- (g/km)', '-/-', '9.999.999',' g/km']
raw = clean_variable(raw, 'fuel_consumption_g_km',
                     'fuel_consumption_g_km_num',
                     replacements, suffix='g/km')
```

La variable est désormais numérique et tout ce qui empêchait la mise en place du format numérique a été supprimé.

Nous pouvons adopter la même démarche avec la variable `fuel_consumption_l_100km` en commençant par inscrire `0 l/100 km` pour les véhicules électriques :

```
raw.loc[raw['fuel_type'] == "Electric",
        'fuel_consumption_l_100km'] = "0 l/100 km"
```

Puis utiliser à nouveau la fonction définie précédemment pour nettoyer cette variable :

```
replacements = ['- (l/100 km)', '-/-',' l/100 km']
raw = clean_variable(raw, 'fuel_consumption_l_100km',
                     'fuel_consumption_l_100km_num',
                     replacements, suffix='l/100 km')
```

Pour finir avec ces deux variables, nous allons imputer les valeurs manquantes en récupérant, lorsqu'elles existent ou sont les plus courantes, les valeurs définies par marque, modèle, version, carburant et année de production.

Voici la fonction utilisée pour l'occasion :

```
from collections import Counter

# Fonction pour trouver la valeur unique ou majoritaire
def get_majority_or_unique(series):
    non_null_values = series.dropna()
    if len(non_null_values) == 0:
        return pd.Series(dtype='float64')
    counter = Counter(non_null_values)
    if len(counter) == 1:
        # Si une seule valeur unique existe
        return non_null_values.iloc[0]
    else:
        # Sinon la valeur la plus fréquente est retournée
        return counter.most_common(1)[0][0]
```

La fonction va être maintenant utilisée dans la boucle suivante :

```
# Colonnes à traiter
columns_to_fill = ['fuel_consumption_g_km_num',
                   'fuel_consumption_l_100km_num']

# Pour chaque caractéristique à remplir
for col in columns_to_fill:
    # Colonne temporaire pour stocker les valeurs calculées
    raw['temp_' + col] = raw.groupby(['brand',
                                      'model',
                                      'version',
                                      'fuel_type',
                                      'year'])[col]\
                                .transform(lambda x:
get_majority_or_unique(x))

    # Colonne d'origine remplie avec la colonne temporaire
    raw[col].fillna(raw['temp_' + col], inplace=True)

    # Suppression la colonne temporaire
    raw.drop(columns=['temp_' + col], inplace=True)
```

Finalement, grâce à ce type d'imputation, 3698 valeurs ont pu être récupérées pour la première variable et 773 pour la deuxième :

```
fuel_consumption_g_km_num      212872 non-null  float64
fuel_consumption_l_100km_num   227830 non-null  float64
```

```
fuel_consumption_g_km_num      216570 non-null  float64
fuel_consumption_l_100km_num   228603 non-null  float64
```

Gestion de la variable year

La variable `year` peut être convertie en numérique :

```
raw['year'] = pd.to_numeric(raw['year'], errors='coerce')
```

Gestion de la variable month

Nous allons récupérer pour terminer, le mois à partir de la variable `registration_date` :

```
raw['registration_date'] =
pd.to_datetime(raw['registration_date'],
                              format='%m/%Y', errors='coerce')
raw['month'] = raw['registration_date'].dt.month
```

Et pour finir la partie nettoyage, au vu du volume des données, nous ne garderons que les observations correctement remplies :

```
df = raw.dropna().copy()
```

Il est désormais temps d'explorer les données.

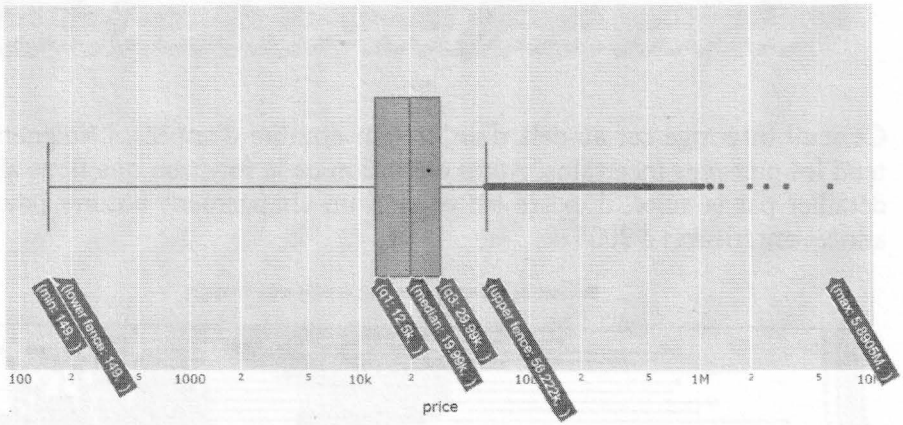
3.1.3 Exploration et analyse

Analyse univariée du prix

Dans la mesure où la modélisation portera sur le prix, nous débuterons cette exploration par la distribution de la valeur `price` :

```
import plotly.express as px

# Créer le boxplot horizontal avec échelle logarithmique
fig = px.box(df, x='price',
             title='Boxplot des prix (échelle logarithmique)',
             log_x=True)

# Affichage
fig.show()
```

Boxplot des prix (échelle logarithmique)

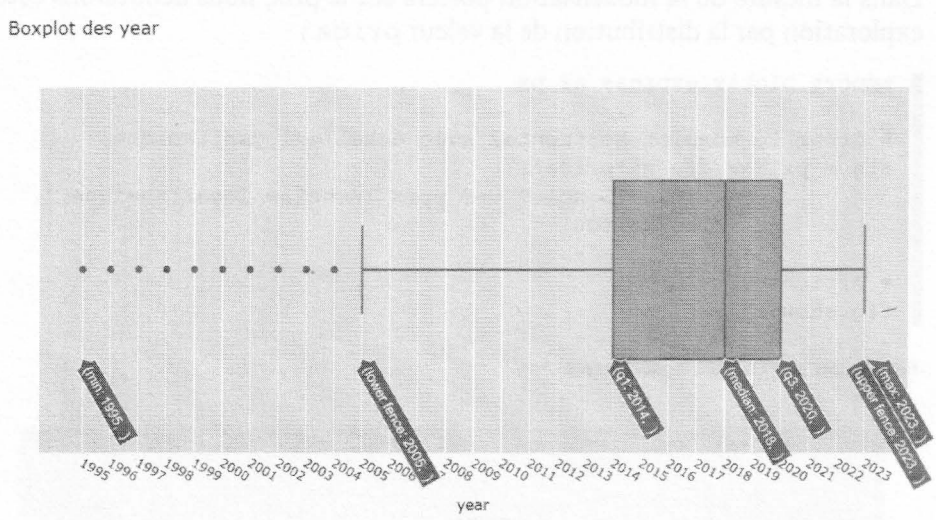


La distribution des prix est étendue, allant de 149 euros à 5 890 500 euros. Dans l'optique de ne pas trop nous éparpiller et éviter de biaiser les prédictions avec des voitures haut de gamme et rares, nous nous limiterons au seuil des valeurs aberrantes, soit 56 222 euros.

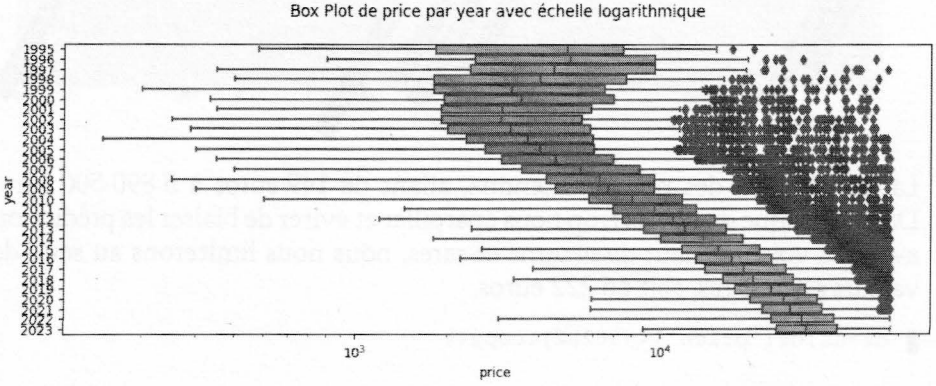
```
df=df[df['price']<=56222].copy()
```

Analyse univariée de l'année de production

L'étude de la distribution des années présente également des valeurs identifiées comme aberrantes pour les années antérieures à 2005 :



Ce seuil interroge car au-delà d'un certain nombre d'années, l'éloignement rend les prix plus incertains. Après exécution de la fonction que nous allons détailler par la suite, il existe bel et bien un changement notoire pour les années antérieures à 2005 :



Dans le graphique précédent, le corps des boxplots a tendance à s'allonger de manière notoire au fur et à mesure que nous remontons dans le temps et de manière encore plus criante avant 2005. Ce double constat nous invite à circonscrire notre borne inférieure à 2005.

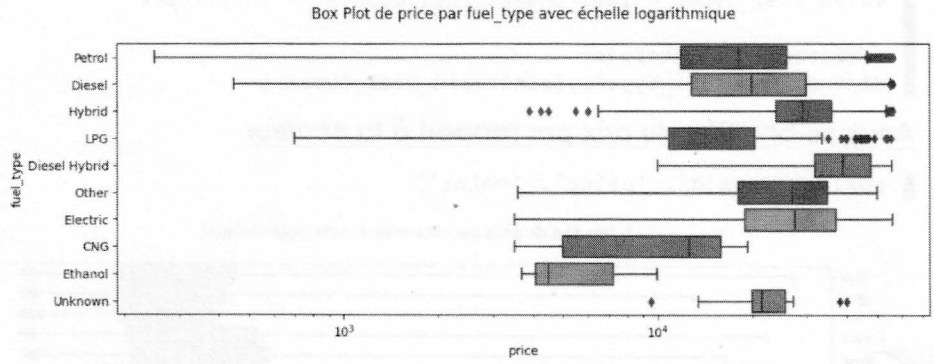
```
df=df[df['year']>=2005].copy()
```

Pour la suite de nos explorations, nous utiliserons la fonction `analyze_data` (dans le premier notebook) qui nous permettra de directement utiliser le bon test statistique après avoir testé la normalité de la variable numérique.

Analyse bivariée du prix par rapport au type de motorisation

L'exécution de la commande suivante va générer un graphique et les résultats du test statistique :

```
analyze_data(df, 'price', 'fuel_type')
```




```
--- Résultats du test d'Anderson-Darling ---
Statistique d'Anderson-Darling: 1620.1137933206628
Valeurs critiques: [0.576 0.656 0.787 0.918 1.092]
Niveaux de signification: [15.  10.   5.   2.5  1. ]
Les données ne suivent pas une distribution normale. Effectuer le test de Kruskal-Wallis.
--- Résultats du test de Kruskal-Wallis ---
Statistique de Kruskal-Wallis: 11173.109976262902
P-value: 0.0
```

La variable `fuel_type` se révèle parfaitement significative mais quelques mentions isolées attirent notre attention. Nous allons vérifier les effectifs à l'aide de la fonction `value_counts()` :

```
df.fuel_type.value_counts()
```

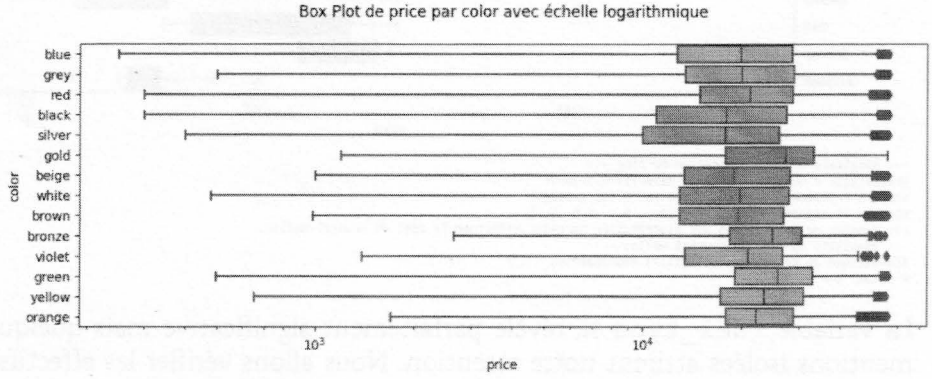
```
Petrol           108050
Diesel            66151
Hybrid             9181
Electric           4913
LPG                 714
Diesel Hybrid       303
Other                66
CNG                  25
Unknown              22
Ethanol               3
Name: fuel_type, dtype: int64
```

Certaines mentions ont de faibles effectifs (<100) et nous allons procéder à leur suppression :

```
# Calcul des effectifs de chaque groupe
fuel_counts = df['fuel_type'].value_counts()

# Sélection des types de carburant ayant au moins 100 occurrences
valid_fuel_types = fuel_counts[fuel_counts >= 100].index

# Filtrage du DataFrame
df = df[df['fuel_type'].isin(valid_fuel_types)]
```

Analyse bivariée du prix par rapport à la couleur

```
analyze_data(df, 'price', 'color')
```




```
--- Résultats du test d'Anderson-Darling ---
Statistique d'Anderson-Darling: 1619.7220040719549
Valeurs critiques: [0.576 0.656 0.787 0.918 1.092]
Niveaux de signification: [15.  10.   5.   2.5  1. ]
Les données ne suivent pas une distribution normale. Effectuer le test de Kruskal-Wallis.
--- Résultats du test de Kruskal-Wallis ---
Statistique de Kruskal-Wallis: 2934.0915258636774
P-value: 0.0
```

```
1 df.color.value_counts()
```

```
black     43205
grey      35894
white     31826
silver    24620
blue      23645
red       17152
brown      3510
orange     2820
green      2287
beige      1875
yellow     1319
gold        431
bronze      425
violet      303
Name: color, dtype: int64
```

La couleur et le prix partagent un lien significatif et les effectifs par couleur sont tous supérieurs à 300. Nous conserverons donc cette variable ainsi que toutes les mentions pour notre modélisation.

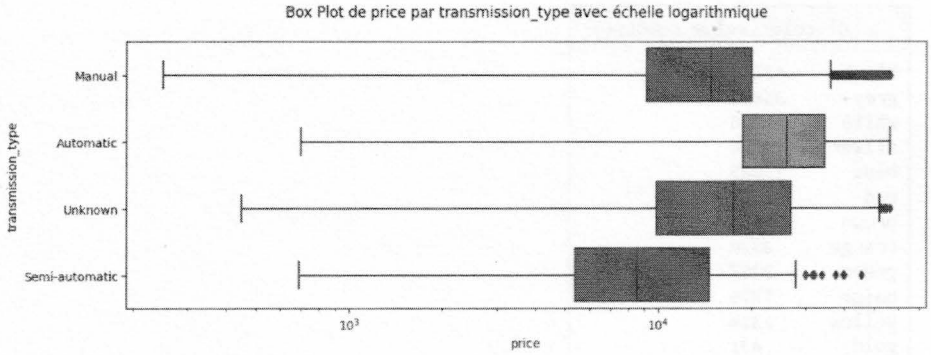
Quelques remarques sur la couleur :

- Les voitures couleur or et vert ont tendance à se vendre plus cher.
- Les prix sont les plus hétérogènes pour les couleurs noir et argent, et les plus homogènes pour les teintes orange et bronze.

Analyse bivariée du prix par rapport au type de transmission

Le type de transmission se révèle également être significatif :

```
analyze_data(df, 'price', 'transmission_type')
```




```
--- Résultats du test d'Anderson-Darling ---
Statistique d'Anderson-Darling: 1619.7220040719549
Valeurs critiques: [0.576 0.656 0.787 0.918 1.092]
Niveaux de signification: [15.  10.   5.   2.5  1. ]
Les données ne suivent pas une distribution normale. Effectuer le test de Kruskal-Wallis.
--- Résultats du test de Kruskal-Wallis ---
Statistique de Kruskal-Wallis: 49297.27284380534
P-value: 0.0
```

La mention `Unknown` est gênante car le test est très significatif et garder cette mention risque d'ajouter du bruit dans notre modélisation. Vérifions la part qu'ils représentent dans les données :

```
1  df.transmission_type.value_counts()
```

```
Automatic          95517
Manual             92880
Unknown              731
Semi-automatic       184
Name: transmission_type, dtype: int64
```

Avec une représentativité de 731/189312, soit 0.3 %, nous pouvons écarter cette mention :

```
df = df[df["transmission_type"]!="Unknown"].copy()
```

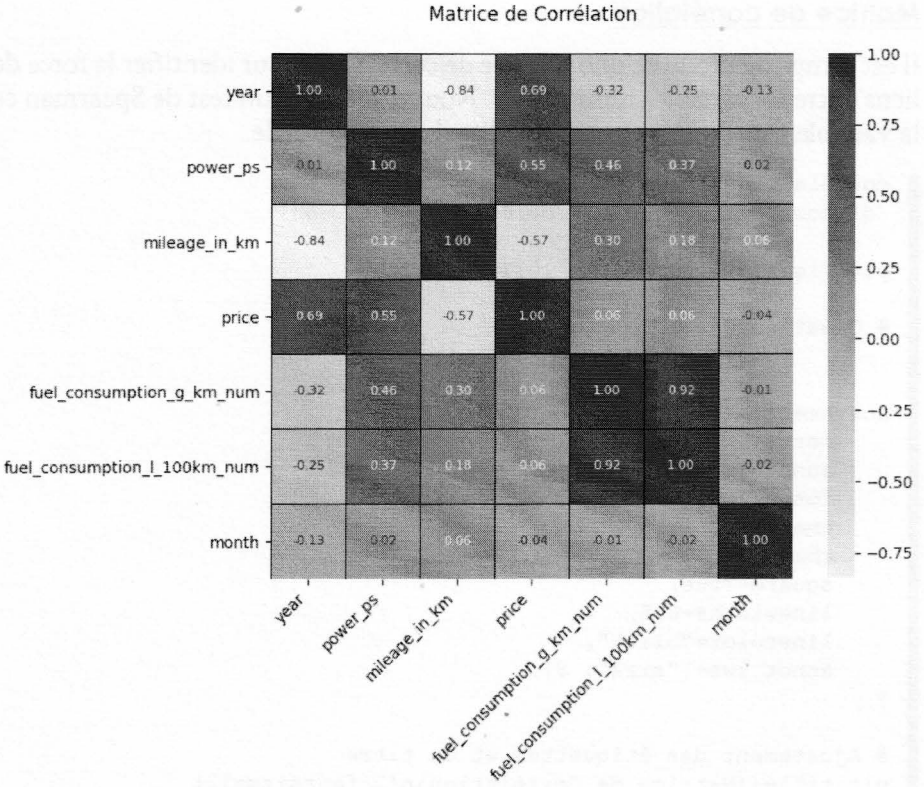
Remarque

La décision de garder ou supprimer certaines mentions est motivée par le temps et la taille de l'échantillon. Lorsque l'échantillon est faible, il faut tout faire pour tout conserver mais ces actions sont chronophages et peuvent se heurter au mur de la réalité économique.

Matrice de corrélation

Il est temps de produire une matrice de corrélation pour identifier la force des liens entre les variables numériques. Nous utiliserons un test de Spearman car la variable `price` ne suit pas une distribution normale.

```
correlation_matrix =
 df.corr(numeric_only=True,method='spearman')

plt.figure(figsize=(12, 10))

# Création d'une heatmap


sns.heatmap(
    correlation_matrix,
    annot=True,
    fmt=".2f",
    cmap="coolwarm",
    cbar=True,
    square=True,
    linewidths=0.5,
    linecolor="black",
    annot_kws={"size": 8},
)

# Ajustement des étiquettes et du titre
plt.title('Matrice de Corrélation\n', fontsize=12)
plt.xticks(rotation=45, ha='right')
plt.yticks(rotation=0)

# Affichage de la figure
plt.show()
```



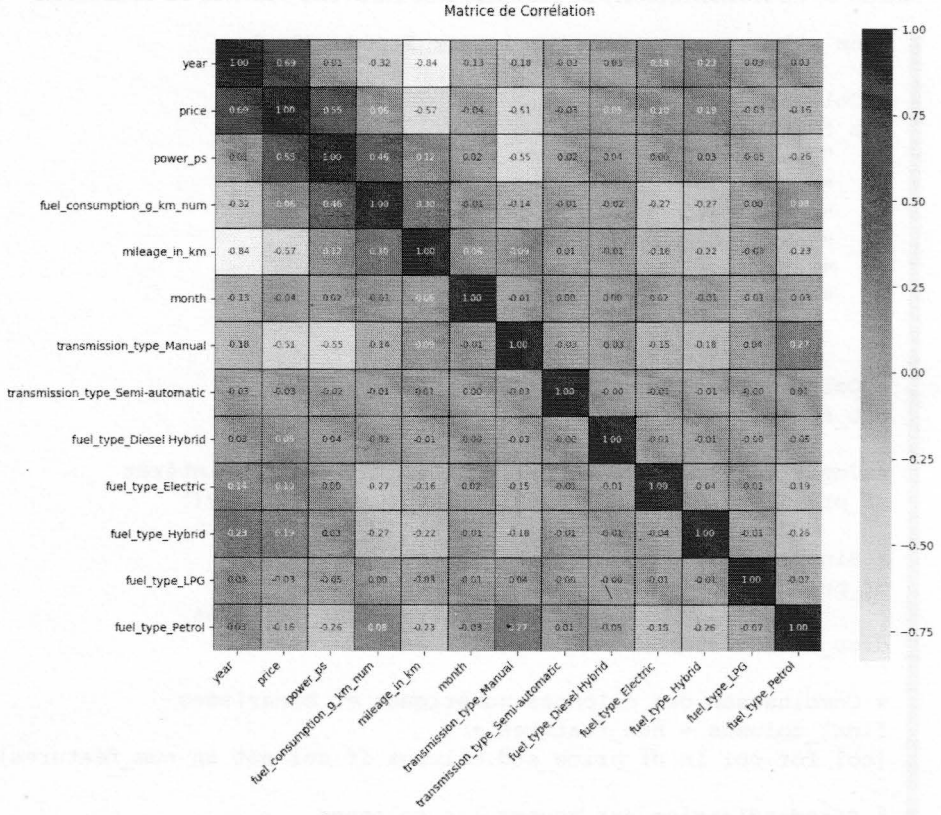
Voici un petit résumé des phénomènes constatés dans la matrice :

- Les deux variables liées à `full_comsumption` sont proches d'être jumelles. A priori, nous conserverons `fuel_consumption_g_km_num`, qui partage plus de corrélations fortes que `fuel_consumption_l_100km_num`.
- Il existe une corrélation forte entre l'année et le kilométrage, ce qui paraît logique.
- La variable `month` semble complètement décorrélée de toutes les autres. Étudier si cela peut être une opportunité ou un handicap.

Avant de passer à l'ACP qui va nous apporter une meilleure compréhension de nos données, nous allons inclure deux variables catégorielles (type de motorisation et de transmission) et produire une nouvelle matrice de confusion :

```
from sklearn.preprocessing import StandardScaler

# Colonnes numériques
num_features = [
    "year",
    "price",
    "power_ps",
    "fuel_consumption_g_km_num",
    "mileage_in_km",
    "month",
]

# Colonnes catégorielles à binariser
cat_features = ["transmission_type", "fuel_type"]

# Copie du DataFrame en gardant les variables d'intérêt
df_price_std = df[num_features+cat_features].copy()

# Binarisation des variables catégorielles
df_price_std = pd.get_dummies(df_price_std,
                              columns=cat_features,
drop_first=True)

# Combinaison des colonnes numériques et binarisées
final_columns = num_features + \
[col for col in df_price_std.columns if col not in num_features]

# Standardisation sur toutes les colonnes
scaler = StandardScaler()
df_price_std[final_columns] = scaler.fit_transform(
    df_price_std[final_columns]
)
```

En utilisant le code fourni précédemment et en changeant juste le nom de la table, nous obtenons la matrice suivante :



L'ajout des variables catégorielles apporte plus de subtilité dans l'analyse bien que certaines mentions n'émergent pas avec des corrélations trop faibles globalement. Nous allons ainsi les supprimer :

```
df_price_std.drop(["month","transmission_type_Semi-automatic",
                   "fuel_type_Diesel Hybrid",
                   "fuel_type_LPG"],axis=1,inplace=True)
```

ACP

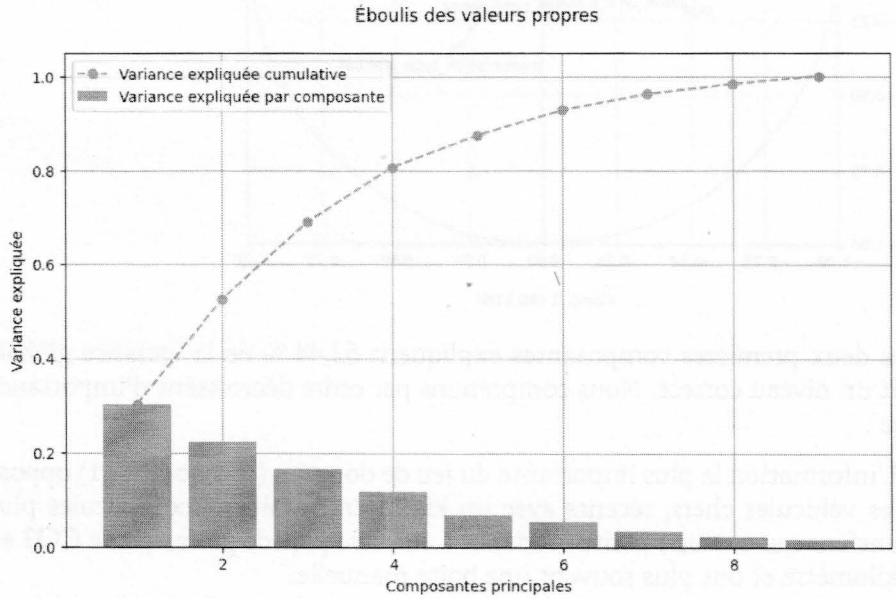
En reprenant les codes des fonctions de l'ACP étudiées au chapitre L'apprentissage non supervisé nous allons obtenir directement les graphiques nécessaires.

Il convient juste d'appeler l'ACP et de transformer les données :

```
# Initialisation de l'ACP
pca = PCA()
pca.fit(df_price_std)
```

Nous allons ensuite lancer la production de l'éboulis des valeurs propres :

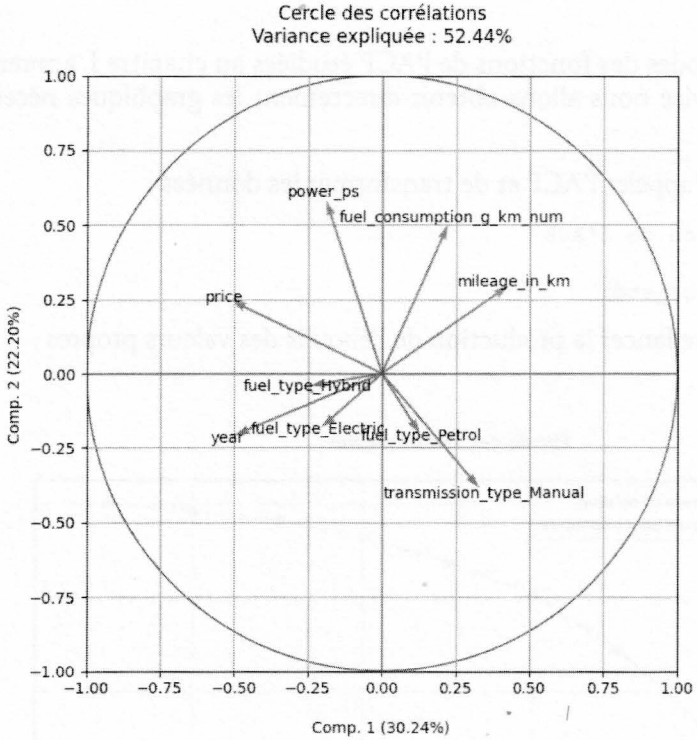
```
scree_plot(pca)
```



Avec 4 composantes, nous avons un bon résumé de nos données avec un pourcentage de variance expliqué supérieur à 80 %.

Nous allons étudier les cercles des corrélations des deux premiers plans factoriels.

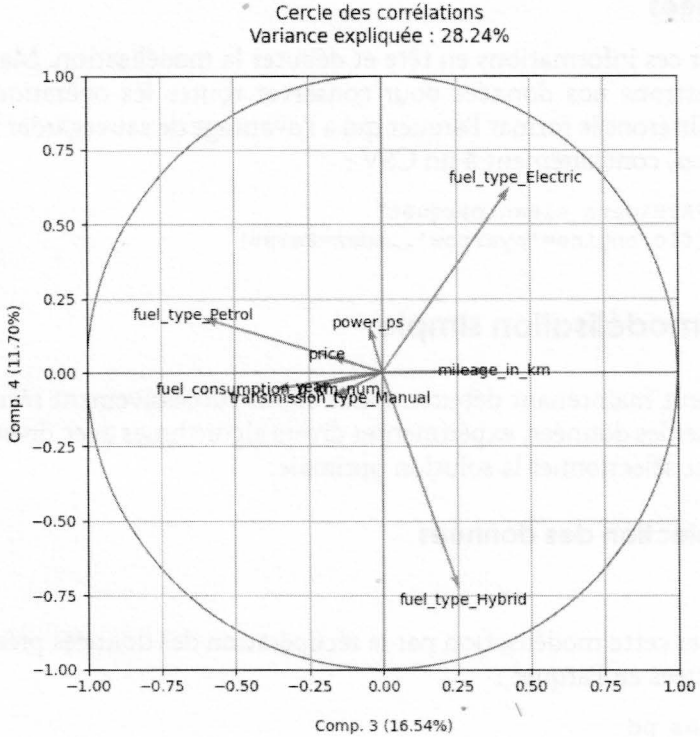
```
plot_correlation_circle(pca, [0,1], df_price_std.columns)
```



Ces deux premières composantes expliquent 52,44 % de la variance globale soit un niveau correct. Nous comprenons par ordre décroissant d'importance que :

- L'information la plus importante du jeu de données (composante 1) oppose les véhicules chers, récents avec un kilométrage faible aux véhicules plus anciens qui ont une petite tendance à émettre plus de grammes de CO2 au kilomètre et ont plus souvent une boîte manuelle.
- Sur la composante 2, nous retrouvons une opposition entre les voitures puissantes, plus émettrices de CO2 et avec, en moyenne, plus de kilomètres d'une part. Et d'autre part, les voitures peu puissantes et moins polluantes qui ont plus tendance à avoir une boîte de vitesses manuelle.

```
plot_correlation_circle(pca, [2,3], df_price_std.columns)
```



Pour ce deuxième plan factoriel, voici les informations qui ressortent :

- Sur la composante 3, nous retrouvons l'opposition entre les moteurs thermiques plus polluants à gauche et les moteurs électriques ou semi-électriques à droite qui ont tendance à arborer un kilométrage plus important.
- Enfin, la composante 4 marque la différence entre les véhicules électriques au nord et les véhicules hybrides (essence) au sud. Cette opposition est vraiment centrée sur le type de motorisation et ne peut être rapprochée d'autres variables.

En résumé, les informations de la base de données reposent par ordre d'importance décroissante sur le prix, l'ancienneté, le kilométrage, la puissance, les émissions, l'opposition entre thermique et non thermique et, enfin, l'opposition entre électrique et hybride.

Export des données

Nous allons garder ces informations en tête et débuter la modélisation. Mais avant cela, enregistrons nos données pour conserver toutes les opérations réalisées. Nous utiliserons le format Parquet qui a l'avantage de sauvegarder le formatage des types, contrairement à un CSV :

```
fic = r"YOUR_PATH\data_clean.parquet"
df.to_parquet(fic,engine='pyarrow',index=False)
```

3.2 Notebook 2 : modélisation simple

La modélisation peut maintenant débuter. Nous allons successivement récupérer et sélectionner les données, expérimenter divers algorithmes avec divers hyperparamètres et sélectionner la solution optimale.

3.2.1 Acquisition et sélection des données

Acquisition

Nous allons débuter cette modélisation par la récupération des données préalablement enregistrées en Parquet :

```
import pandas as pd
import numpy as np
pd.set_option("display.max_columns", 999)

fic = r"YOUR_PATH\data_clean.parquet"

pd.set_option("display.max_columns", 999)
df = pd.read_parquet(fic engine='pyarrow')
```

Point sur les variables catégorielles

Pour cette première modélisation, nous allons directement intégrer des variables catégorielles qui contribueront à renforcer la précision des prédictions. Le tableau suivant va nous indiquer nos possibilités de sélection concernant ce type de variables :

```
df.describe(include=['object','category'])
```

	brand	model	color	transmission_type	fuel_type	version
count	188581	188581	188581	188581	188581	188581
unique	44	1037	14	3	6	152496
top	volkswagen	Volkswagen Golf	black	Automatic	Petrol	Titanium
freq	24928	5840	43006	95517	107705	166

Premier constat : les variables `model` et `version` présentent beaucoup trop de modalités donc leur usage n'est pas envisageable en utilisant une binarisation des mentions.

Les autres variables sont de potentielles candidates mais nous allons tout de même contrôler le nombre d'occurrences pour `brand` car certaines marques pourraient se révéler trop confidentielles :

```
# Affichage des 15 mentions les plus rares
df.brand.value_counts().tail(15)
```

```
maserati        179
lada            145
isuzu           138
daihatsu        122
cadillac         78
saab             77
chevrolet        75
infiniti         74
dodge            64
lancia           59
bentley          16
aston-martin     10
daewoo            9
rover             3
chrysler          1
```

Les effectifs des cinq dernières marques sont bien trop faibles et pour maximiser nos chances de réussir la modélisation, nous les écarterons :

```
# Comptage des occurrences de chaque marque
brand_counts = df['brand'].value_counts()

# Filtrage des marques avec plus de 50 occurrences
brands_above_50 = brand_counts[brand_counts > 50].index.tolist()
```

```
# Filtrage des données
df = df[df['brand'].isin(brands_above_50)].copy()
```

Les effectifs par mention des catégorielles restantes ont tous été testés dans le notebook correspondant et tous présentent au moins une centaine de cas par mention donc nous les conserverons tous pour la modélisation.

3.2.2 Modélisation

Nous pouvons maintenant passer à la modélisation en commençant par importer les modules nécessaires :

```
# Option pour afficher tout le contenu
pd.set_option('display.max_colwidth', None)

import re

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.compose import ColumnTransformer
from sklearn.metrics import mean_absolute_error, mean_squared_error
```

Définissons maintenant la variable cible, le prix, ainsi que les variables explicatives :

```
# Variable cible
y = df['price']

# Variables explicatives
num_features = ["year", "power_ps",
                "fuel_consumption_g_km_num", "mileage_in_km"]

categorical_features = ["brand", "transmission_type",
                        "fuel_type","color"]

X = df[num_features + categorical_features]
```

Nous allons ensuite séparer les données d'entraînement des données de test :

```
X_train, X_test, y_train, y_test = \
train_test_split(X, y, test_size=0.3, random_state=42)
```

Puis il faut définir le préprocesseur traitant les variables selon leur type :

- Les variables catégorielles sont binarisées.
- Pour les variables numériques, le `StandardScaler` a été choisi après comparaison avec le `RobustScaler`, car il offrait une exécution plus rapide avec des scores similaires.

```
# Définition du préprocesseur
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(),
         num_features),
        ('cat', OneHotEncoder(drop='first', sparse=False),
         categorical_features)
    ]

)
```

Trois algorithmes sont comparés ici avec différents hyperparamètres. Nous pourrions en tester davantage, mais cela augmenterait considérablement le temps d'exécution. À ce propos, l'`HistGradientBoostingRegressor` a remplacé le `RandomForestRegressor`, car le temps d'entraînement de ce dernier était démesurément long pour des performances très proches.

```
param_grid = [
    {
        'regressor': [LinearRegression()],
        'regressor__fit_intercept': [True, False]
    },
    {
        'regressor': [KNeighborsRegressor()],
        'regressor__n_neighbors': [1, 2, 5]
    },
    {
        'regressor':
[HistGradientBoostingRegressor(random_state=42)],
        'regressor__max_iter': [500, 700],
        'regressor__max_depth': [None, 10],
        'regressor__learning_rate': [0.1, 0.07],
    }
]
```

Remarque

La liste des algorithmes est limitée à trois pour l'exemple mais en conditions réelles, il faut en tester le plus possible pour maximiser les chances de trouver l'optimum.

Le pipeline peut maintenant être défini et lancé à travers une `GridSearchCV` :

```
# Création d'un pipeline incluant les transformations
pipeline = Pipeline([
    ('preprocessor', preprocessor),  # Prétraitement
    ('regressor', LinearRegression())  # Modèle par défaut
])

# Création de la GridSearchCV
grid_search = GridSearchCV(pipeline,
                           param_grid, cv=3,
                           return_train_score=True,
                           n_jobs=8)

# Entraînement
grid_search.fit(X_train, y_train)
```

Remarque

Le paramètre `n_jobs` définit le nombre de cœurs CPU à utiliser pour les calculs. Par défaut à 1, il peut être modifié selon le nombre de cœurs disponibles. L'option -1 utilise la totalité des cœurs.

Le temps d'entraînement, de quelques minutes, est relativement rapide dans notre cas. Nous pouvons maintenant étudier les performances.

3.2.3 Résultats

Résultats par simulation

À l'aide du code suivant, nous allons pouvoir établir le classement des algorithmes testés :

```
# Récupération des résultats dans un DataFrame
results = pd.DataFrame(grid_search.cv_results_)

# Extraction du nom de l'algorithme
results['regressor'] = results['param_regressor'].apply(
```

```
    lambda x: re.sub(r'\s*\(.*?\)', '', str(type(x).__name__))
)

# Nettoyage de params_clean
results['params_clean'] = results['params'].apply(
    lambda d: {k.replace('regressor__', ''): v
               for k, v in d.items()
               if 'regressor__' in k}
)

# Colonnes à conserver
cols_to_keep = ['regressor', 'params_clean',
                'mean_test_score', 'std_test_score',
                'rank_test_score']

# Réorganisation des colonnes
results_clean =
results[cols_to_keep].sort_values('rank_test_score')

display(results_clean)
```

	regressor	params_clean	mean_test_score	std_test_score	rank_test_score
6	HistGradientBoostingRegressor	{'learning_rate': 0.1, 'max_depth': None, 'max_iter': 700}	0.886135	0.001600	1
5	HistGradientBoostingRegressor	{'learning_rate': 0.1, 'max_depth': None, 'max_iter': 500}	0.885288	0.001703	2
8	HistGradientBoostingRegressor	{'learning_rate': 0.1, 'max_depth': 10, 'max_iter': 700}	0.885216	0.002488	3
12	HistGradientBoostingRegressor	{'learning_rate': 0.07, 'max_depth': 10, 'max_iter': 700}	0.885026	0.001615	4
10	HistGradientBoostingRegressor	{'learning_rate': 0.07, 'max_depth': None, 'max_iter': 700}	0.884908	0.001608	5
7	HistGradientBoostingRegressor	{'learning_rate': 0.1, 'max_depth': 10, 'max_iter': 500}	0.884661	0.002211	6
9	HistGradientBoostingRegressor	{'learning_rate': 0.07, 'max_depth': None, 'max_iter': 500}	0.883504	0.001702	7
11	HistGradientBoostingRegressor	{'learning_rate': 0.07, 'max_depth': 10, 'max_iter': 500}	0.883097	0.001650	8
4	KNeighborsRegressor	{'n_neighbors': 5}	0.853607	0.001999	9
3	KNeighborsRegressor	{'n_neighbors': 2}	0.830147	0.001535	10
0	LinearRegression	{'fit_intercept': True}	0.798591	0.003123	11
2	KNeighborsRegressor	{'n_neighbors': 1}	0.783100	0.002860	12
1	LinearRegression	{'fit_intercept': False}	0.778917	0.002852	13

L'algorithme `HistGradientBoostingRegressor` s'impose sur les deux autres avec un score d'environ 0,886.

RMSE et MAE

Pour parfaire notre connaissance de cet algorithme, nous allons d'abord afficher la RMSE et la MAE :

```
# Calcul de la MAE et RMSE pour le meilleur modèle
best_model = grid_search.best_estimator_
y_pred_best = best_model.predict(X_test)

mae_best = mean_absolute_error(y_test, y_pred_best)
rmse_best = np.sqrt(mean_squared_error(y_test, y_pred_best))

# Affichage des résultats
bm = best_model.named_steps['regressor'].__class__.__name__

print(f"\nModèle : {bm}")
print(f"Score : {grid_search.best_score_:.3f}")
print(f"RMSE : {rmse_best:.2f}")
print(f"MAE : {mae_best:.2f}")
```

```
Modèle : HistGradientBoostingRegressor
Score : 0.886
RMSE : 3871.09
MAE : 2482.95
```

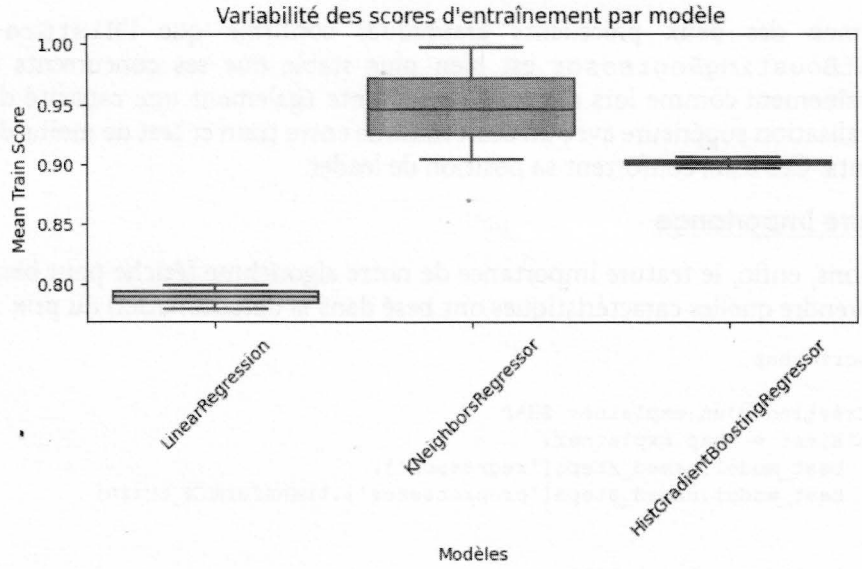
Ainsi, avec notre système de variables, nous trouvons le prix d'un véhicule à environ 2483 euros près. La RMSE d'environ 3871 euros indique la présence d'écarts importants, ce qui est en accord avec la présence de modèles plus chers qui y contribuent.

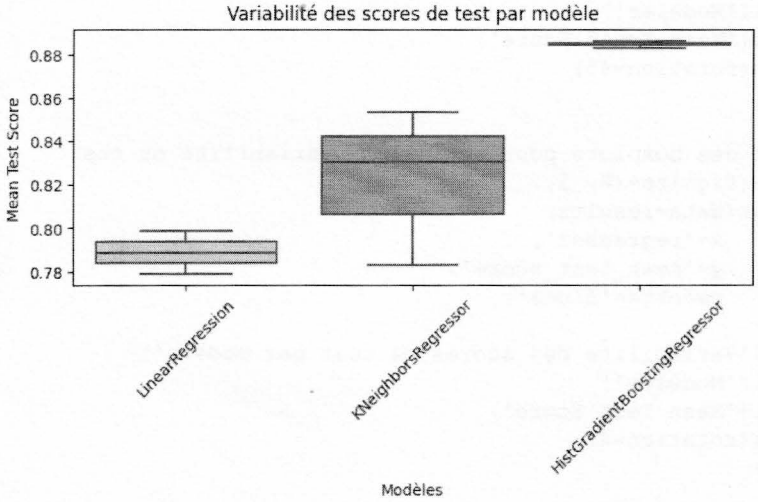
Stabilité des algorithmes

Ensuite, nous allons étudier la stabilité des algorithmes testés en visualisant la variabilité des scores d'entraînement et de test issus de la `GridSearchCV` :

```
# Création des boxplots pour évaluer la variabilité du train
plt.figure(figsize=(8, 3))
sns.boxplot(data=results,
            x='regressor',
            y='mean_train_score',
            palette='Blues')

plt.title('Variabilité des scores d\'entraînement par modèle')
```

```
plt.xlabel('Modèles')
plt.ylabel('Mean Train Score')
plt.xticks(rotation=45)
plt.show()

# Création des boxplots pour évaluer la variabilité du test
plt.figure(figsize=(8, 3))
sns.boxplot(data=results,
            x='regressor',
            y='mean_test_score',
            palette='Blues')

plt.title('Variabilité des scores de test par modèle')
plt.xlabel('Modèles')
plt.ylabel('Mean Test Score')
plt.xticks(rotation=45)
plt.show()
```





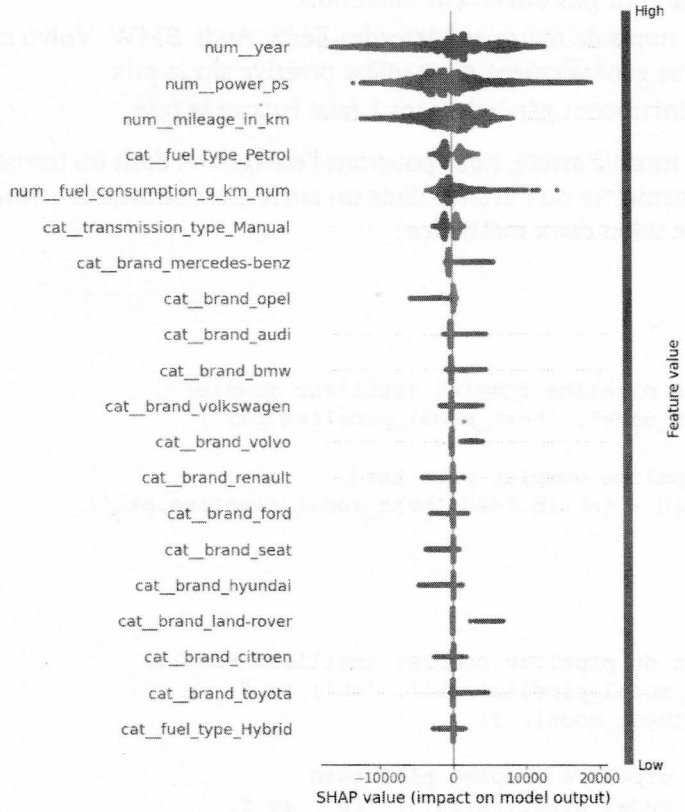
L'examen des deux précédents graphiques confirme que l'`HistGradientBoostingRegressor` est bien plus stable que ses concurrents à l'entraînement comme lors des tests. Il présente également une capacité de généralisation supérieure avec un écart minime entre train et test de moins de 2 points. Ces tests confortent sa position de leader.

Feature importance

Étudions, enfin, le feature importance de notre algorithme fétiche pour bien comprendre quelles caractéristiques ont pesé dans la détermination du prix :

```
import shap

# Création d'un explainer SHAP
explainer = shap.Explainer(
    best_model.named_steps['regressor'],
    best_model.named_steps['preprocessor'].transform(X_train)
)

# Calcul des valeurs SHAP
shap_values = explainer(
    best_model.named_steps['preprocessor'].transform(X_test),
    check_additivity=False
)
```

```
# Extraction des noms des caractéristiques
preprocessor = best_model.named_steps['preprocessor']
cols_names = preprocessor.get_feature_names_out()

# Visualisation de l'importance des caractéristiques
shap.summary_plot(
    shap_values,
    best_model.named_steps['preprocessor'].transform(X_test),
    feature_names=cols_names
)
```



L'affichage précédent mixe le feature importance global et local en présentant les variables par ordre décroissant d'importance. Chaque observation est représentée par un petit cercle et indique dans quelle mesure elle a pesé à la hausse ou la baisse dans la prédiction du prix.

Nous pouvons ainsi mieux comprendre notre système et son fonctionnement :

- Quelles sont les variables les plus importantes ?
- Leur effet contribue-t-il à la hausse ? À la baisse ? Dans les deux sens ?

De cette visualisation, nous comprenons que :

- C'est l'année, suivie de la puissance et du nombre de kilomètres qui contribuent le plus à l'établissement du prix.
- Que leurs effets ne sont pas forcément bilatéraux :
  - La présence des noms de marques Mercedes-Benz, Audi, BMW, Volvo ou Toyota contribue généralement de manière positive sur le prix.
  - Opel ou Seat contribuent généralement à faire baisser le prix.

Une fois le choix du modèle arrêté, nous pourrons l'enregistrer dans un format adéquat pour le transmettre ou l'utiliser dans un autre environnement. Voici la procédure à suivre selon deux méthodes :

- Avec Joblib :

```
import joblib

# Enregistrer le pipeline complet (meilleur modèle)
joblib.dump(best_model, 'best_model_pipeline.pkl')

# Charger le pipeline complet plus tard
best_model_loaded = joblib.load('best_model_pipeline.pkl')
```

- Avec Pickle :

```
import pickle

# Enregistrement du pipeline complet (meilleur modèle)
with open('best_model_pipeline.pkl', 'wb') as f:
    pickle.dump(best_model, f)

# Chargement du pipeline complet plus tard
with open('best_model_pipeline.pkl', 'rb') as f:
    best_model_loaded = pickle.load(f)
```

3.3 Notebook 3 : modélisation mixte

Les résultats de la modélisation précédente sont satisfaisants, mais une approche mixte pourrait être plus efficace. En utilisant le traitement du langage naturel (NLP) pour analyser les données textuelles et en les combinant avec les variables numériques, il serait possible de créer un système décisionnel encore plus performant.

C'est ce que nous allons expérimenter dans ce dernier notebook.

3.3.1 Acquisition et sélection des données

Pour cette nouvelle modélisation, nous allons repartir du jeu de données `df` tel que défini juste avant la sous-section Modélisation.

Commençons par importer les librairies nécessaires :

```
import pandas as pd
import numpy as np

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error,
mean_absolute_error, r2_score
from sklearn.preprocessing import StandardScaler
```

Nous allons ensuite créer la variable `vehicule` qui sera la concaténation de toutes les variables textuelles :

```
# Concaténation des colonnes textuelles en 'vehicule'
df['vehicule'] = df['model'] + ' ' + df['version'] + \
' ' + df['color'] + ' ' + df['transmission_type']+ ' ' +
df['fuel_type']

# Comblement des vides
df['vehicule'] = df['vehicule'].fillna('')

# Suppression des anciennes variables
df.drop(["brand","model","color","transmission_type",
"fuel_type","version"],axis=1,inplace=True)
```

Une fois mise en place, un TF-IDF peut être initialisé :

```
# Utilisation d'un TFIDF
vectorizer = TfidfVectorizer(stop_words=None,
                             max_features=500,
                             min_df=0.005,
                             max_df=0.9)

# Transformation de la variable textuelle en vecteur
vehicule_matrix = vectorizer.fit_transform(df['vehicule'])
```

Remarque

Aucun `stop_words` n'est demandé car toutes les informations sont potentiellement importantes dans ce type de cas.

```
print(vehicule_matrix.shape)

# Output: (188542, 300)
```

Avec les contraintes définies, 300 mots sont ainsi retenus. Les différentes commandes disponibles dans le notebook permettent par exemple d'afficher les 20 mots considérés comme les plus importants :

Mots les plus importants (TF-IDF) :

	word	tfidf_score
207	petrol	0.083202
182	manual	0.080143
54	automatic	0.074981
104	diesel	0.065704
192	navi	0.054668
62	black	0.053783
135	grey	0.046352
291	white	0.043324
173	led	0.040934
63	blue	0.036458

Sans surprise, nous retrouvons des caractéristiques générales liées, par exemple, au carburant, au type de boîte de vitesses ou à la couleur.

Nous allons à présent étudier comment modéliser les données en combinant les variables textuelles du TF-IDF aux autres variables numériques indispensables à l'établissement du prix.

3.3.2 Modélisation

La modélisation mixte demande peu d'étapes.

Il faut juste importer les modules nécessaires. Nous utiliserons ici un `ExtraTreesRegressor`, qui, après expérimentations, affiche les meilleurs résultats :

```
from scipy.sparse import hstack
from sklearn.ensemble import ExtraTreesRegressor
```

Vient ensuite l'assemblage des données :

```
# Sélection des variables numériques
numerical_feature_names = ['year',
                           'power_ps',
                           'fuel_consumption_g_km_num',
                           'mileage_in_km']

# Concaténation des deux tables
X_combined = hstack([vehicule_matrix,
                     df[numerical_feature_names]])

# Récupération des mots du TfidfVectorizer
words = vectorizer.get_feature_names_out()

# Combinaison des noms des features textuels et numériques
combined_feature_names = list(words) + numerical_feature_names

# Variable cible
y = df['price']

# Séparation des données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(
    X_combined.toarray(),
    y,
    test_size=0.3,
    random_state=42
    )
```

Et pour finir, place à l'expérimentation :

```
# Définition et entraînement du modèle
model = ExtraTreesRegressor(n_estimators=100,
                            max_depth=None,
                            random_state=42)

model.fit(X_train, y_train)

# Prédire les valeurs sur l'ensemble de test
y_pred = model.predict(X_test)

# Calculer les performances du modèle
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Afficher les résultats
print(f"ExtraTreesRegressor - RMSE : {rmse:.2f}")
print(f"ExtraTreesRegressor - MAE : {mae:.2f}")
print(f"ExtraTreesRegressor - R²: {r2:.3f}")
```

```
ExtraTreesRegressor - RMSE : 3781.83
ExtraTreesRegressor - MAE : 2242.35
ExtraTreesRegressor - R² : 0.894
```

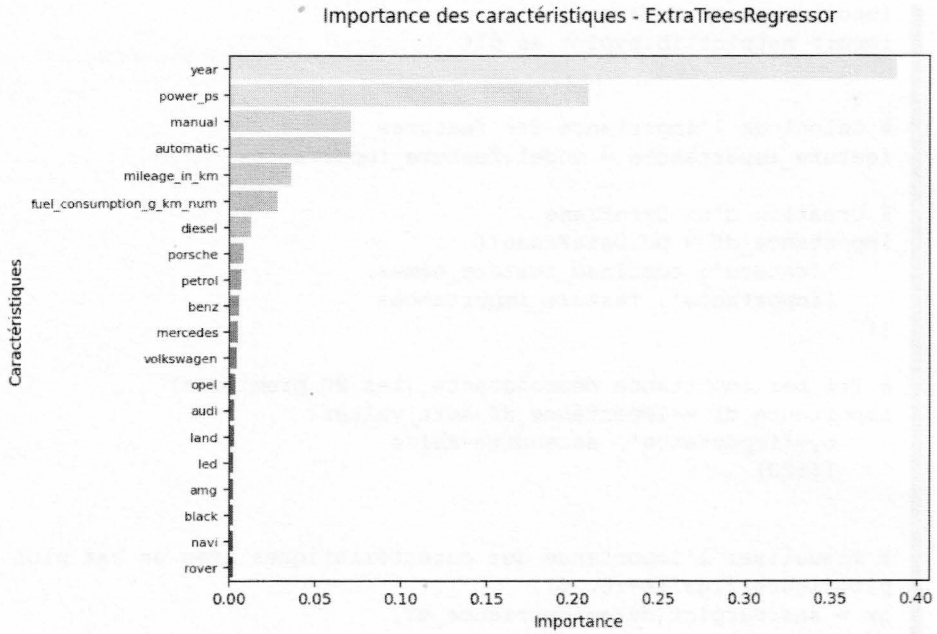
3.3.3 Résultats

La méthode mixte améliore nettement les résultats. Le R^2 augmente de 0,886 à 0,894 (+0,8 point), et la MAE diminue de 2482 à 2242 euros, soit une réduction notable de 240 euros.

Cependant, tout n'est pas parfait : SHAP est inutilisable, le fichier de l'algorithme est hyper volumineux (1,5 Go), et les temps d'entraînement sont sensiblement plus longs. Cela complique l'utilisation d'un pipeline, devenu trop lent pour une mise en production efficace.

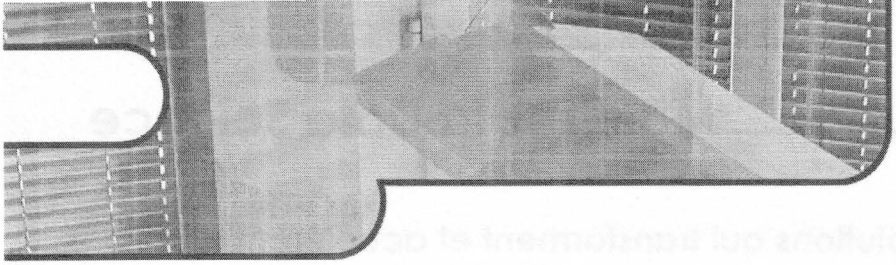
Heureusement, l'algorithme intègre un feature importance qui permet d'afficher les variables d'importance, même si la visualisation restera moins instructive qu'avec SHAP.

```
import seaborn as sns
import matplotlib.pyplot as plt


# Calcul de l'importance des features
feature_importances = model.feature_importances_

# Création d'un DataFrame
importance_df = pd.DataFrame({
    'feature': combined_feature_names,
    'importance': feature_importances
})

# Tri par importance décroissante (les 20 premières)
importance_df = importance_df.sort_values(
    by='importance', ascending=False
    )[:20]


# Visualiser l'importance des caractéristiques avec un bar plot
plt.figure(figsize=(8, 6))
ax = sns.barplot(data=importance_df,
                 x='importance',
                 y='feature',
                 palette='Blues')
plt.title('Importance des caractéristiques -
ExtraTreesRegressor\n')
plt.xlabel('Importance')
plt.ylabel('Caractéristiques')

# Modifier la taille de la police de l'axe y
ax.set_yticklabels(ax.get_yticklabels(), fontsize=8)

plt.show()
```



4. Conclusion

Globalement, comme dans la précédente simulation, ce sont les mêmes variables qui ressortent. Bien que les performances se soient améliorées, ces expérimentations montrent que chaque tentative d'optimisation nécessite de composer avec des contraintes, notamment de puissance de calcul. Ici, il serait impossible d'envisager une `GridSearchCV`, car nos machines ne sont tout simplement pas assez puissantes pour mener à bien ces recherches exhaustives dans des délais raisonnables. L'équilibre entre performance et faisabilité reste, la plupart du temps, le principal enjeu.



Conclusion

1. Le rôle central des données et de leur compréhension

Tout au long de cet ouvrage, les données sont apparues comme le facteur déterminant de tout processus de data science en soulignant l'identification des caractéristiques pertinentes comme le préalable indispensable à la réussite d'une modélisation. Lorsque les résultats ne sont pas au rendez-vous, il apparaît clairement que toutes les variables explicatives n'ont pas été identifiées. Pour certains phénomènes, c'est d'ailleurs tout simplement impossible car ils sont soumis à trop de facteurs.

Prenons l'exemple de la prédiction du temps passé dans un musée. Identifier précisément cette durée pour chaque personne nécessiterait une quantité énorme d'informations pour prédire au mieux la réalité. En pratique, c'est tout bonnement impossible.

Ici, nous touchons du doigt les problèmes de systèmes chaotiques où une modification mineure sur l'un des paramètres peut aboutir à des résultats complètement différents et inattendus.

Une bonne connaissance de la théorie sera donc une précieuse conseillère et nous évitera de perdre notre temps à élaborer des modélisations qui ne pourraient aboutir. Pour autant, l'évolution rapide des techniques ne doit pas être ignorée.

2. Des évolutions qui transforment et accélèrent tout

Nous sommes actuellement dans une phase d'accélération de l'intelligence artificielle où même les professionnels du secteur les plus aguerris sont surpris de constater l'émergence de techniques qu'ils n'auraient pas imaginées il y a encore quelques années, remettant en question certaines certitudes et modifiant en profondeur le paysage de la data science.

Ces évolutions peuvent être regroupées selon les trois thématiques suivantes :

2.1 L'évolution du matériel technologique

Le développement du matériel technologique est au cœur des évolutions et de leur accélération. En proposant toujours plus de puissance, il rend possible l'émergence de nouveaux modèles parfois révolutionnaires.

Les capacités de puissance de calcul reposent majoritairement sur deux piliers : la GPU (*Graphic Processing Unit*) et la TPU (*Tensor Processing Unit*).

- Conçus à l'origine pour le rendu graphique, les GPU se sont avérés polyvalents pour des applications variées, notamment dans le domaine de l'apprentissage automatique, grâce à leur capacité à effectuer des calculs parallèles de manière efficace.
- Les TPU ont, en revanche, été spécifiquement créés par Google pour accélérer les calculs liés aux réseaux de neurones, facilitant ainsi l'entraînement et le déploiement des modèles d'IA à grande échelle.

En parallèle de ces progrès sur la puissance de calcul émergent de nouvelles technologies prometteuses, comme l'ordinateur quantique. Ce nouveau type d'ordinateur propose une capacité de traitement sans précédent et pourrait résoudre des problèmes complexes inaccessibles aux architectures classiques, ce qui pourrait accélérer encore davantage le développement de l'IA.

2.2 L'amélioration des modèles

Les modèles algorithmiques sont les grands bénéficiaires de ces avancées. L'augmentation de la puissance de calcul permet notamment :

- Le développement des modèles multimodaux combinant différents types de données (texte, images, son) permettant l'enrichissement du contenu créé, ce qui renforce les capacités créatives.
- L'émergence des modèles génératifs de type GAN (*Generative Adversarial Networks*) qui vont pouvoir proposer des contenus créatifs de plus en plus sophistiqués ou la génération de données synthétiques favorisant l'entraînement d'autres modèles.
- L'automatisation des processus comme l'apprentissage autosupervisé ou l'apprentissage par renforcement qui vont faciliter encore davantage l'entraînement en rendant les systèmes quasiment autonomes.
- Les LLM (*Large Language Model*), comme ChatGPT, qui sont la manifestation la plus visible pour le grand public de toutes les évolutions précédemment citées. Elles permettent désormais de comprendre et générer du texte de manière quasiment humaine.

2.3 La diffusion dans le grand public et la prise en compte progressive des enjeux

Avec sa diffusion dans le grand public, l'IA a connu une nouvelle phase de son évolution. En l'adoptant, les individus ont à la fois pris conscience des nouvelles opportunités offertes par cette technologie mais aussi des interrogations, voire des craintes, liées à sa généralisation et son amplification dans tous les rouages de nos sociétés. De nouvelles professions verront certainement le jour bientôt comme autant de réponses à l'organisation, l'intégration et l'encadrement de cette technologie de premier ordre.

3. Importance de la théorie et invitation à l'exploration

Au-delà des considérations sociétales et philosophiques sur lesquelles nous n'avons que peu d'influence, il est important de comprendre le fonctionnement de cette nouvelle technologie pour mieux l'appréhender et la contrôler plutôt que la subir. En envisageant nos connaissances théoriques comme une boussole dans ce monde numérique en perpétuel changement, nous l'aborderons de manière positive comme une invitation à l'exploration qui renforcera notre savoir et notre connaissance du monde.

En compagnie de Python, embarquons pour ce voyage d'exploration et de découverte. Grâce à lui, nous disposons non seulement d'une solution complète pour extraire, nettoyer, visualiser et analyser nos données, et ainsi en comprendre les enjeux, mais aussi pour concevoir des modèles prédictifs et automatiser des tâches complexes, nous ouvrant ainsi la voie à des prises de décisions plus éclairées et pertinentes.

A

B

C

D

E

F

G

K

L

M

N

R

S

T

V

X

Maîtrisez la Data Science avec Python

Maîtriser les techniques de modélisation et **comprendre les données**, véritable carburant de l'**intelligence artificielle**, sont devenues des compétences clés dans une société transformée par la révolution numérique.

Que vous soyez débutant ou en quête de nouvelles compétences, ce livre vous guide dans **l'univers de la data science**, une discipline qui transcende les frontières de la programmation pour extraire des informations pertinentes et concevoir des systèmes capables d'offrir des solutions concrètes dans tous les domaines.

Conçu comme un véritable mode d'emploi, ce livre vous accompagne à travers toutes les étapes du **traitement et de l'analyse des données** : collecte, préparation, exploration, modélisation prédictive et mise en application. Grâce à **Python et ses bibliothèques incontournables**, vous découvrirez une méthode claire et des exemples concrets pour **transformer vos données en savoir et en valeur**, de manière à vous permettre d'appliquer immédiatement les concepts abordés.

Vous apprendrez à :

- comprendre les mécanismes fondamentaux de la data science ;
- importer, manipuler et visualiser des données complexes avec des outils comme **Pandas**, **Seaborn** et **Matplotlib** ;
- analyser les variables avec **Scipy** et **Statmodels** ;
- appliquer des **algorithmes de Machine Learning** pour résoudre des problématiques réelles sur des données classiques, des images ou du texte ;
- automatiser et structurer vos analyses dans un environnement riche et accessible.

Eric DEMANGEL est data scientist freelance depuis 2020 après avoir accumulé 11 années d'expérience en tant que data analyste dans une société d'études de marché. Passionné par l'enseignement, il a également endossé le rôle de mentor en data, avec lequel il guide ses étudiants avec une curiosité insatiable et un désir constant de transformer les données en savoir. À travers ses enseignements, il s'efforce de rendre ce domaine complexe accessible à tous, en simplifiant ses concepts et en partageant son expertise de manière claire et pédagogique.

Sur **www.editions-eni.fr** :
➜ Notebooks des exemples du livre.

Pour plus d'informations :

39 €

ISBN : 978-2-409-04872-2

eni
www.editions-eni.fr