LA DATA SCIENCE DE A À Z POUR LES DEBUTANTS

Devenez un explorateur de données avec Python

Cédric Bohnert

Version 0.1

14/08/2025

PRÉFACE

Ce livre est conçu comme un parlivre d'initiation à la data science pour les lycéens, utilisant le langage de programmation Python. Il est structuré en quatre grandes parties, allant des concepts fondamentaux jusqu'à des sujets avancés et la préparation à une carrière dans le domaine.

PARTIE 1 : LES FONDATIONS DE LA DATA SCIENCE

Cette première partie pose les bases. Elle commence par définir ce qu'est la data science et son impact sur le quotidien. Ensuite, elle guide l'étudiant dans l'installation de Python et des outils essentiels comme NumPy et Pandas pour la manipulation de données. Les concepts mathématiques clés tels que l'algèbre linéaire, le calcul différentiel (pour comprendre la descente de gradient), les probabilités et les statistiques descriptives sont introduits de manière intuitive. La section se termine par l'apprentissage de la visualisation de données avec Matplotlib et Seaborn et un premier projet pratique sur le nettoyage et l'exploration du dataset du Titanic.

PARTIE 2 : LE MACHINE LEARNING

Le cœur du livre se concentre sur le machine learning. Après avoir expliqué les grandes familles d'apprentissage (supervisé, non supervisé), cette partie détaille les algorithmes les plus importants :

- Régression Linéaire et Logistique pour les tâches de prédiction et de classification.
- Arbres de Décision et Forêts Aléatoires pour des modèles plus complexes et interprétables.
- K-Means pour le clustering et les Machines à Vecteurs de Support (SVM) pour la classification avancée.

Une attention particulière est portée à l'évaluation des modèles, à la prévention du surapprentissage (overfitting) et à la création de variables pertinentes (feature engineering). Un second projet complet sur la prédiction du prix des maisons est mené de A à Z.

PARTIE 3 : LA DATA SCIENCE EN PRATIQUE

Cette section ancre les compétences théoriques dans le monde réel. Elle aborde des sujets essentiels comme :

- La communication des résultats à travers le data storytelling.
- L'extraction de données depuis des bases de données avec SQL.
- La collecte de données en temps réel via les APIs et le web scraping.
- La collaboration et le versioning de code avec Git & GitHub.
- Le déploiement d'un modèle via une API web avec Flask et Docker.
- La participation à des compétitions sur Kaggle.

PARTIE 4 : SUJETS AVANCÉS ET CULTURE DATA

La dernière partie ouvre des perspectives sur des sujets plus avancés et sur la culture du métier. Elle introduit les réseaux de neurones et le deep learning avec Keras et TensorFlow pour des applications sur les images (CNN) et le texte (RNN, Transformers). Des thèmes cruciaux comme l'éthique en IA, les systèmes de recommandation, et les algorithmes de pointe comme le Gradient Boosting sont explorés. Le livre se conclut par une vue d'ensemble des carrières en data, des conseils pour préparer les entretiens techniques et une feuille de route pour continuer à apprendre.

SOMMAIRE

INTRODUCTION

Bienvenue dans le monde fascinant de la data science ! Si vous vous êtes déjà demandé comment Netflix choisit la prochaine série à vous recommander, comment votre GPS prédit les embouteillages, ou comment les scientifiques analysent des millions de données pour faire des découvertes, alors vous êtes au bon endroit. Ce livre est votre guide pour devenir un véritable explorateur de données.

Oubliez l'image d'un domaine réservé aux génies des mathématiques. La data science est avant tout une aventure intellectuelle qui mêle curiosité, logique et un peu de programmation. Nous allons partir de zéro, en utilisant Python – un langage aussi puissant que simple à apprendre – pour transformer des tableaux de chiffres en histoires captivantes, en prédictions surprenantes et en décisions intelligentes. À la fin de ce parlivre, vous ne verrez plus le monde de la même manière. Vous apprendrez

à poser les bonnes questions, à déceler les secrets cachés dans les données et à construire vos propres modèles d'intelligence artificielle. Préparez-vous à penser comme un data scientist !

PARTIE 1 LES FONDATIONS

---

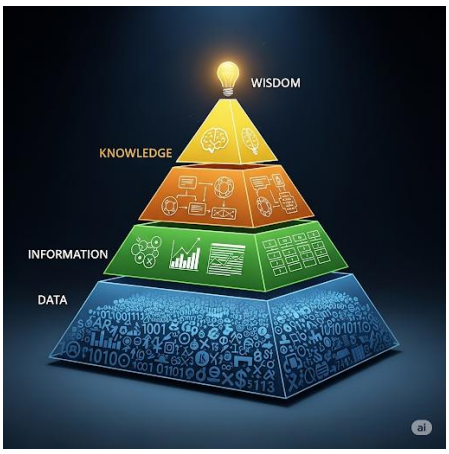
1 POURQUOI LA DATA SCIENCE VA CHANGER VOTRE FAÇON DE PENSER

Vous devez choisir un restaurant pour une occasion spéciale. Option A : vous vous fiez à votre intuition, à ce nom qui sonne bien, ou à cette devanture que vous avez aperçue en passant en voiture. Option B : vous ouvrez une application, analysez des centaines d'avis, filtrez par type de cuisine, par note, par budget, et consultez les photos des plats les plus populaires. Dans le premier cas, vous espérez faire le bon choix. Dans le second, vous prenez une décision éclairée, basée sur une multitude d'expériences.

La data science, c'est exactement ça : un passage de l'intuition à la décision éclairée, mais à une échelle bien plus grande. C'est une discipline qui transforme notre manière de comprendre le monde et d'interagir avec lui. Oubliez l'image du statisticien reclus dans sa cave ; la data science est une aventure intellectuelle qui va remodeler votre façon de penser.

QU'EST-CE QUE LA DATA SCIENCE ? LA PYRAMIDE DE LA CONNAISSANCE

Au fond, la data science est l'art de transformer les données brutes en sagesse. Pour bien comprendre, on utilise souvent la pyramide DIKW (Data, Information, Knowledge, Wisdom) :



1. Données (Data) : C'est la base. Des chiffres, des textes, des images, des clics... Pris isolément, un chiffre comme "37" ne veut rien dire. C'est du bruit.
2. Information : En ajoutant du contexte, les données deviennent de l'information. "La température extérieure est de 37°C". C'est déjà plus utile.
3. Connaissance (Knowledge) : En connectant plusieurs informations, on obtient de la connaissance. "Il fait 37°C, nous sommes en juillet à Marseille, et c'est 10°C de plus que la moyenne saisonnière." On comprend maintenant qu'il s'agit d'une canicule.
4. Sagesse (Wisdom) : C'est le sommet de la pyramide. La sagesse consiste à utiliser cette connaissance pour prendre des décisions. "Puisqu'il s'agit d'une canicule, je vais rester hydraté, éviter les efforts physiques et prendre des nouvelles de mes voisins âgés."

La data science est le processus qui nous fait grimper cette pyramide, en utilisant des méthodes scientifiques, des algorithmes et de la technologie pour extraire de la valeur des données.

SON IMPACT INVISIBLE SUR VOTRE QUOTIDIEN

Vous pratiquez la data science sans même le savoir. Quand Netflix vous recommande une série avec une précision déconcertante, ce n'est pas de la magie. C'est le résultat de l'analyse des habitudes de visionnage de millions d'utilisateurs. Quand Waze ou Google Maps vous fait quitter l'autoroute pour un itinéraire bis, il ne devine pas l'embouteillage : il l'a prédit en analysant en temps réel les données de vitesse de milliers de téléphones. De la détection de fraudes sur votre carte bancaire aux publicités que vous voyez en ligne, la data science est partout.

Les 4 compétences clés du Data Scientist

Contrairement aux idées reçues, il ne suffit pas d'être un génie des mathématiques. La data science repose sur un équilibre de quatre compétences fondamentales :

1. La Curiosité : Tout commence par une question. Pourquoi nos ventes baissent-elles dans cette région ? Quel est le facteur qui influence le plus la satisfaction de nos clients ? Sans une curiosité insatiable, les données restent muettes.

2. La Statistique : C'est la grammaire de la data. Elle permet de distinguer un signal pertinent du simple bruit, de comprendre les relations de cause à effet et de quantifier l'incertitude.

3. La Programmation : C'est l'outil qui permet de manipuler, d'analyser et de visualiser des volumes de données qu'aucun humain ne pourrait traiter manuellement. Des langages comme Python ou R sont les couteaux suisses du data scientist.

4. La Communication : C'est peut-être la compétence la plus sous-estimée. Vous pouvez avoir le meilleur modèle du monde, si vous n'êtes pas capable d'expliquer ses résultats de manière claire et convaincante à un public non-expert, votre travail n'aura aucun impact.

De la théorie à la pratique : voir les données en action

Pour cette introduction, nous n'écrivons pas de code. L'objectif est de saisir l'esprit de la discipline. Rien de mieux pour cela que de voir des données prendre vie. Le travail du regretté Hans Rosling est une référence absolue en la matière. Dans sa célèbre conférence, il utilise des bulles animées pour raconter 200 ans d'histoire du monde en quelques minutes, rendant des tendances complexes (espérance de vie, revenus) extraordinairement intuitives.

- À voir absolument : La conférence de Hans Rosling sur l'évolution du monde (vidéo TED, sous-titres français disponibles)

Regarder cette vidéo, c'est comprendre le pouvoir de la visualisation : transformer des feuilles de calcul austères en une histoire captivante.

Challenge pour vous !

Maintenant, à votre tour de chausser vos lunettes de data scientist.

Trouvez un chapitre d'actualité récent (presse, blog, rapport...) où une décision importante (économique, politique, sanitaire, sportive) a été clairement influencée par une analyse de données. Expliquez en une ou deux phrases quel a été l'impact de cette décision.

2 Python pour la Data Science : L'installation et les 10 commandes à connaître par cœur

Dans notre premier chapitre, nous avons vu que la data science est un état d'esprit. Maintenant, il est temps de se doter des outils pour passer à l'action. Et l'outil roi dans ce domaine, c'est le langage de programmation Python. Considérez-le comme le couteau suisse du data scientist : simple en apparence, mais incroyablement puissant et polyvalent.

Aujourd'hui, nous allons mettre en place votre environnement de travail et apprendre à manier les 10 commandes fondamentales qui vous ouvriront les portes de l'analyse de données.

Pourquoi choisir Python ?

Il existe des dizaines de langages de programmation, alors pourquoi Python s'est-il imposé en data science ? Pour trois raisons principales :

1. Sa lisibilité : La syntaxe de Python est proche de la langue anglaise, ce qui le rend relativement facile à apprendre et à lire, même pour un débutant. Moins de temps à se battre avec des points-virgules, plus de temps à réfléchir aux problèmes.
2. Sa communauté : Python est soutenu par une immense communauté mondiale de développeurs et de data scientists. Cela signifie que si vous rencontrez un problème, il y a de fortes chances que quelqu'un l'ait déjà résolu et partagé la solution en ligne.
3. Ses bibliothèques : C'est son atout maître. Python dispose de bibliothèques (des collections de code pré-écrit) spécialisées pour la data science comme Pandas pour la manipulation de tableaux, Matplotlib et Seaborn pour la visualisation, et Scikit-learn pour le machine learning. Ces outils vous feront gagner un temps précieux.

Anaconda ou Pip : Que choisir pour débuter ?

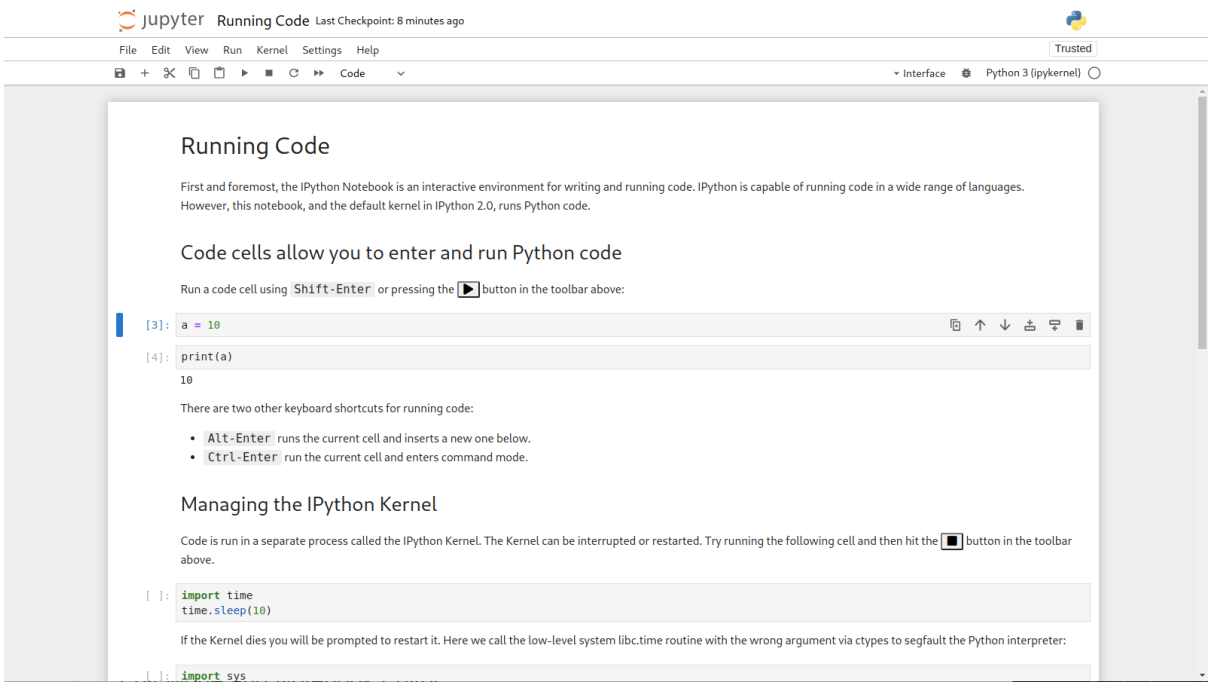
Pour installer Python et ses bibliothèques, deux voies principales existent : pip (le gestionnaire de paquets standard de Python) et Anaconda.

Pour faire simple : si vous débutez en data science, utilisez Anaconda.

Anaconda est une distribution tout-en-un qui installe non seulement Python, mais aussi des centaines de bibliothèques de data science les plus populaires et des outils très utiles, comme le Jupyter Notebook. Cela vous évite les maux de tête liés à la gestion des compatibilités entre les différentes bibliothèques. C'est le chemin le plus simple pour avoir un environnement de travail fonctionnel en quelques clics.

Qu'est-ce qu'un Jupyter Notebook ?

Imaginez un cahier de laboratoire numérique où vous pouvez écrire du texte, des équations, du code Python, et voir immédiatement le résultat de ce code (un tableau, un graphique, etc.). C'est exactement ce qu'est un Jupyter Notebook. C'est l'environnement de prédilection des data scientists pour l'exploration de données, car il permet de tester des idées rapidement et de documenter son raisonnement pas à pas. C'est l'outil que nous utiliserons principalement dans nos futurs chapitres.



GUIDE D'INSTALLATION PAS À PAS D'ANACONDA

1. Téléchargement : Rendez-vous sur la page de téléchargement officielle d'Anaconda. Le site devrait détecter automatiquement votre système d'exploitation (Windows, macOS ou Linux). Téléchargez la version la plus récente de Python (Python 3.x).

2. Lancement de l'installateur : Une fois le fichier téléchargé, ouvrez-le pour lancer l'installation.

3. Suivez les instructions : Cliquez sur "Next" ou "Continue". Pour la plupart des utilisateurs, les options par défaut sont parfaites. Il n'est pas nécessaire de modifier les paramètres avancés au début.

4. Attendez : L'installation peut prendre plusieurs minutes, car elle installe de nombreux paquets. Soyez patient !

5. Vérification : Une fois l'installation terminée, ouvrez l'application "Anaconda Navigator" qui a été ajoutée à vos programmes. Vous devriez y voir une icône pour "Jupyter Notebook". Cliquez sur "Launch" pour démarrer votre premier notebook !

LES 10 COMMANDES ESSENTIELLES POUR DÉMARRER

Ouvrez un Jupyter Notebook et tapez ces commandes dans les cellules pour vous familiariser avec les bases de Python.

1. print() : Afficher un résultat La commande la plus basique pour afficher du texte ou la valeur d'une variable.

```
print("Bonjour, Data & Découvertes !")
```

2. Assignation de variable (=) : Stocker une information On utilise le signe = pour donner un nom à une valeur.

```
nombre_de_visiteurs = 150
print(nombre_de_visiteurs)
```

3. Types de base : Les briques élémentaires Python gère différents types de données : les nombres entiers (int), les nombres à virgule (float), le texte (str), et les booléens (True/False).

```
age = 30                # int
prix = 19.99            # float
nom = "Alice"           # str
est_inscrit = True      # bool
```

4. Listes ([]) : Une collection ordonnée Pour stocker plusieurs éléments dans un ordre précis.

```
fruits = ["pomme", "banane", "cerise"]
print(fruits[0])  # Affiche 'pomme' (l'indexation commence à 0)
```

5. Dictionnaires ({}) : Une collection par clé-valeur Pour stocker des éléments associés par paires, avec un nom (la clé) pour chaque valeur.

```
utilisateur = {"nom": "Dupont", "age": 42, "ville": "Paris"}
print(utilisateur["ville"]) # Affiche 'Paris'
```

6. Boucles for : Répéter une action Pour parcourir chaque élément d'une liste (ou d'un autre objet "itérable").

```
for fruit in fruits:
    print("J'aime la", fruit)
```

7. Conditions if/else : Prendre une décision Pour exécuter du code uniquement si une certaine condition est remplie.

```
temperature = 12
if temperature > 25:
    print("Il fait chaud !")
else:
    print("Il ne fait pas si chaud.")
```

8. Définition de fonction (def) : Créer ses propres commandes Pour regrouper un bloc de code réutilisable sous un seul nom.

```
def saluer(nom):
    return f"Bonjour, {nom} !"

message = saluer("Bob")
print(message)
```

9. import : Utiliser les super-pouvoirs des bibliothèques Pour charger une bibliothèque et utiliser ses fonctionnalités.

```
import math
racine_carree = math.sqrt(16)
print(racine_carree) # Affiche 4.0
```

10. Commentaires (#) : Laisser des notes Le texte après un # est ignoré par Python. C'est crucial pour expliquer votre code.

```
# Ceci est un commentaire. Il calcule la surface d'un carré.
cote = 5
surface = cote * cote # formule : côté au carré
```

CHALLENGE POUR VOUS !

Vous avez maintenant les outils de base. Mettez-les en pratique !

Écrivez une fonction Python qui prend en entrée une liste de nombres et retourne une nouvelle liste contenant uniquement les nombres pairs.

3 NUMPY : L'ARME SECRÈTE DES DATA SCIENTISTS POUR MANIPULER LES NOMBRES

Dans l'chapitre précédent, nous avons appris les commandes de base de Python. Nous avons notamment vu les listes, qui sont très pratiques pour stocker des collections d'éléments. Mais en data science, nous ne travaillons pas avec quelques dizaines de données, mais plutôt avec des milliers, des millions, voire des milliards. À cette échelle, l'efficacité n'est plus une option, c'est une nécessité.

Pour filer la métaphore automobile : si les listes Python sont des breaks familiaux fiables et polyvalents, les tableaux NumPy sont des Formule 1, conçus pour une seule chose : la vitesse de calcul à grande échelle. Voyons pourquoi cette vitesse est cruciale et comment maîtriser ce nouvel outil.

QU'EST-CE QU'UN TABLEAU NUMPY ?

Un tableau NumPy (ou ndarray pour *n-dimensional array*) est une grille de valeurs, qui peuvent être de n'importe quelle dimension. Imaginez un vecteur (1D), une feuille de calcul (2D), ou même un cube de données (3D).

La différence fondamentale avec une liste Python est qu'un tableau NumPy est homogène. Tous ses éléments doivent être du même type (par exemple, uniquement des nombres entiers ou uniquement des nombres à virgule). Cette contrainte est la clé de ses performances exceptionnelles.

POURQUOI EST-CE SI RAPIDE ?

1. Écrit en C : Le cœur de NumPy n'est pas écrit en Python, mais en langages compilés comme le C ou le Fortran. Les opérations mathématiques ne sont pas exécutées par l'interpréteur Python, qui est relativement lent, mais par ce code optimisé et ultra-rapide. Python ne sert que de chef d'orchestre.

2. Opérations vectorielles : Au lieu de faire une boucle sur chaque élément d'une liste pour lui ajouter 5 (ce qui est lent en Python), NumPy vous permet de faire mon_tableau + 5. L'opération est appliquée à tous les éléments simultanément, au niveau du code C. C'est ce qu'on appelle la vectorisation.

3. Stockage mémoire efficace : Grâce à son type de données unique, NumPy stocke les informations de manière beaucoup plus compacte en mémoire qu'une liste Python, ce qui accélère encore les accès et les calculs.

Le "Broadcasting" : La magie de NumPy

Le broadcasting (ou "diffusion" en français) est un mécanisme puissant qui permet à NumPy d'effectuer des opérations sur des tableaux de formes différentes.

L'exemple le plus simple est celui que nous venons de voir : mon_tableau + 5. Ici, NumPy "diffuse" intelligemment le nombre 5 sur tous les éléments du tableau. Il comprend que vous voulez ajouter 5 à chaque cellule de votre grille, sans que vous ayez besoin de le lui dire explicitement avec une boucle. Cette capacité simplifie énormément le code et le rend plus intuitif.

Manipuler les tableaux NumPy

Pour utiliser NumPy, la première étape est toujours la même : l'importer. La convention est de lui donner l'alias np.

```
import numpy as np
```

1. Créer des tableaux On peut créer un tableau à partir d'une liste Python, ou en utilisant des fonctions dédiées.

```
# À partir d'une liste
liste_py = [1, 2, 3, 4]
tableau_np = np.array(liste_py)
print(tableau_np)

# Un tableau de zéros de taille 2x3 (2 lignes, 3 colonnes)
zeros = np.zeros((2, 3))
print(zeros)

# Une séquence de nombres de 0 à 9
sequence = np.arange(10)
print(sequence)
```

2. Opérations de base Les opérations arithmétiques se font élément par élément.

```
a = np.array([10, 20, 30])
b = np.array([1, 2, 3])

print(a + b)  # Résultat : [11 22 33]
print(a * 2)  # Résultat : [20 40 60]
```

3. Indexation et Slicing L'accès aux données est similaire aux listes, mais avec plus de puissance pour les dimensions multiples.

```
data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accéder à un élément (ligne 1, colonne 2)
print(data[1, 2]) # Affiche 6

# Obtenir la première ligne
print(data[0, :]) # Affiche [1 2 3]

# Obtenir la deuxième colonne
print(data[:, 1]) # Affiche [2 5 8]
```

4. Fonctions universelles (ufuncs) NumPy fournit des centaines de fonctions mathématiques qui opèrent directement sur les tableaux de manière très efficace.

```
valeurs = np.array([1, 4, 9, 16])
racines = np.sqrt(valeurs)
print(racines) # Affiche [1. 2. 3. 4.]
```

CHALLENGE POUR VOUS !

Mettez en pratique la puissance de NumPy. Le but est de résoudre ce problème sans utiliser de boucle for !

Créez un tableau NumPy de dimension 5x5 rempli d'entiers aléatoires entre 0 et 20. Ensuite :

1. Calculez la somme de tous les éléments du tableau.
2. Calculez la moyenne de chaque colonne.
3. Remplacez toutes les valeurs strictement supérieures à 10 par la valeur -1.

4 PANDAS : COMMENT TRANSFORMER DES TABLEAUX EXCEL MOCHES EN OR PUR

Jusqu'à présent, nous avons découvert l'état d'esprit de la data science et appris à manipuler des nombres à grande vitesse avec NumPy. Mais la réalité de tout projet de data science commence rarement avec des données propres et bien rangées. Elle commence plutôt avec un fichier CSV désordonné, un export Excel confus ou une base de données pleine de trous.

C'est ici qu'intervient Pandas. Si NumPy est la Formule 1 pour les calculs, Pandas est votre atelier de mécanique de précision, votre baguette magique pour nettoyer, transformer, analyser et préparer n'importe quel jeu de données. Préparez-vous à mettre de l'ordre dans le chaos.

LES DEUX STRUCTURES DE DONNÉES CLÉS : SERIES ET DATAFRAME

Pandas repose sur deux piliers que vous devez absolument connaître :

1. La Series (Série) : C'est l'équivalent d'une seule colonne dans un tableau Excel. Il s'agit d'un tableau unidimensionnel (1D) semblable à un tableau NumPy, mais avec un super-pouvoir : un index. Cet index associe une étiquette à chaque valeur, ce qui permet des sélections et des alignements de données très intuitifs.

2. Le DataFrame (Trame de données) : C'est le cœur de Pandas. Un DataFrame est un tableau à deux dimensions (2D), essentiellement une collection de Series qui partagent le même index. C'est la représentation de votre tableau Excel ou de votre table SQL dans Python. C'est l'objet que vous manipulerez 95% du temps.

L'ANATOMIE D'UN DATAFRAME

Pour bien utiliser un DataFrame, il faut comprendre de quoi il est fait. Imaginez un tableau :

- L'Index (Index) : Ce sont les étiquettes des lignes. Par défaut, ce sont des nombres (0, 1, 2...), mais cela peut être des dates, des noms de produits, ou tout autre identifiant unique. Il est situé sur le côté gauche.
- Les Colonnes (Columns) : Ce sont les étiquettes des colonnes. Elles représentent les différentes variables de votre jeu de données (ex: "Âge", "Prix", "Ville"). Elles sont situées en haut.
- Les Données (Data) : C'est l'ensemble des valeurs qui remplissent le tableau.

L'IMPORTANCE CAPITALE DES TYPES DE DONNÉES (DTYPES)

Chaque colonne d'un DataFrame a un type de données (dtype). Pandas essaie de le deviner à la lecture d'un fichier, mais il est crucial de le vérifier. Pourquoi ?

- Efficacité : Un type correct (ex: int pour des entiers) utilise moins de mémoire qu'un type générique (object).
- Exactitude : Vous ne pouvez pas calculer la moyenne d'une colonne de dates si Pandas la considère comme du simple texte. S'assurer que les nombres sont des nombres et les dates des dates est une étape de nettoyage fondamentale.

LA BOÎTE À OUTILS PANDAS

Commençons par importer Pandas. La convention universelle est de lui donner l'alias pd.

```
import pandas as pd
```

1. Lire un fichier CSV La fonction la plus utilisée pour démarrer un projet.

```
# Charge les données d'un fichier CSV dans un DataFrame
df = pd.read_csv('votre_fichier_de_donnees.csv')
```

2. Inspecter les données Une fois les données chargées, vos premiers réflexes devraient être :

```
# Affiche les 5 premières lignes pour avoir un aperçu
print(df.head())
```

```
# Fournit un résumé technique : index, colonnes, types (dtypes), valeurs non nulles
print(df.info())

# Calcule des statistiques descriptives (moyenne, écart-type, etc.) pour les colonnes numériques
print(df.describe())
```

3. Sélectionner des données C'est une compétence essentielle. Il y a plusieurs manières de le faire :

```
# Sélectionner une seule colonne (renvoie une Series)
ages = df['Age']

# Sélectionner plusieurs colonnes (renvoie un DataFrame)
infos_perso = df[['Nom', 'Ville']]

# Sélectionner des lignes par leur ÉTIQUETTE d'index avec .loc
ligne_etiquette_x = df.loc['etiquette_de_ligne']

# Sélectionner des lignes par leur POSITION numérique avec .iloc
premiere_ligne = df.iloc[0]
trois_premieres_lignes = df.iloc[0:3]
```

4. Filtrer selon des conditions C'est ici que la magie opère. On utilise une condition pour créer un masque de True/False et ne garder que les lignes qui nous intéressent.

```
# Garder uniquement les lignes où l'âge est supérieur à 30
adultes = df[df['Age'] > 30]
print(adultes.head())
```

5. Gérer les valeurs manquantes (NaN) Les données du monde réel sont rarement complètes.

```
# Compter le nombre de valeurs manquantes par colonne
print(df.isnull().sum())

# Supprimer toutes les lignes contenant au moins une valeur manquante
df_nettoye = df.dropna()

# Remplacer toutes les valeurs manquantes par une valeur spécifique (ex: la moyenne de la colonne)
moyenne_age = df['Age'].mean()
df['Age'].fillna(moyenne_age, inplace=True)
```

CHALLENGE POUR VOUS !

Il est temps de mettre les mains dans le cambouis avec un vrai jeu de données. Le jeu de données du Titanic est un classique pour débuter.

Votre mission :

1. Chargez le jeu de données du Titanic (vous pouvez le trouver facilement en ligne, par exemple ici sur Kaggle, prenez le fichier train.csv).

2. Trouvez le nombre de valeurs manquantes dans chaque colonne du DataFrame.
3. Sélectionnez un sous-ensemble de données ne contenant que les passagers ayant payé leur ticket (Fare) plus de 100.
4. Sauvegardez ce sous-ensemble de "passagers fortunés" dans un nouveau fichier CSV que vous nommerez titanic_riches.csv.

5 L'Algèbre Linéaire sans aspirine : Vecteurs et Matrices pour le Machine Learning

N'ayez pas peur du nom ! L'algèbre linéaire est simplement la grammaire qui nous permet de "parler" aux données de manière structurée. Si les données étaient des phrases, les vecteurs et les matrices en seraient les noms et les verbes. C'est le langage qu'utilisent les algorithmes de machine learning pour "penser".

Aujourd'hui, nous allons apprendre ce vocabulaire essentiel, non pas avec des équations indigestes, mais avec des visuels simples.

[Image d'un vecteur comme une flèche dans un espace 2D]

Le sens caché des nombres

Que sont réellement les vecteurs et les matrices ?

- Un Vecteur : Techniquement, c'est une liste de nombres. Mais géométriquement, on peut le voir de deux façons : soit comme un point dans un espace (par exemple, le vecteur [3, 2] est un point aux coordonnées x=3, y=2), soit comme une flèche partant de l'origine [0, 0] et pointant vers ce point. Cette flèche a une direction et une longueur. En data science, un vecteur représente souvent un objet : un utilisateur peut être un vecteur de ses notes de films, une image un vecteur de ses pixels.
- Une Matrice : C'est une grille de nombres, ou un tableau de vecteurs. Si un vecteur est un point, une matrice est une transformation de l'espace. Quand on "applique" une matrice à un vecteur, on déplace ce point ailleurs. La matrice peut étirer, compresser, faire tourner ou cisailler tout l'espace, et donc tous les points (vecteurs) qu'il contient.

Que signifie géométriquement le produit scalaire ?

Le produit scalaire (ou *dot product* en anglais) de deux vecteurs nous donne un seul nombre. Ce n'est pas juste un calcul abstrait ; ce nombre nous renseigne sur la similarité entre les deux vecteurs-flèches, en se basant sur l'angle qui les sépare.

- Si le produit scalaire est grand et positif, les vecteurs pointent dans des directions très similaires.
- S'il est proche de zéro, ils sont perpendiculaires (orthogonaux), donc très différents, sans corrélation.
- S'il est grand et négatif, ils pointent dans des directions opposées.

Qu'est-ce que la multiplication de matrices ?

Si une seule matrice est une transformation, multiplier deux matrices (par exemple, A @ B) revient à composer ou enchaîner deux transformations. Le résultat est une nouvelle matrice qui applique d'abord la transformation B, *puis* la transformation A, le tout en une seule opération. L'ordre est absolument crucial ! Faire tourner puis étirer n'est pas la même chose qu'étirer puis faire tourner.

L'Algèbre linéaire avec NumPy

Nous allons utiliser NumPy (np), notre couteau suisse pour les calculs, qui est optimisé pour ces opérations.

```
import numpy as np
```

1. Représenter des vecteurs et des matrices On utilise simplement np.array().

```
# Un vecteur (tableau 1D)
vecteur_v = np.array([2, 3])
print("Vecteur v:", vecteur_v)

# Une matrice (tableau 2D)
matrice_A = np.array([[1, 2], [3, 4]])
print("Matrice A:\n", matrice_A)
```

2. Addition et multiplication par un scalaire Ces opérations fonctionnent exactement comme on s'y attend.

```
vecteur_w = np.array([5, -1])

# Addition de vecteurs
print("v + w =", vecteur_v + vecteur_w)

# Multiplication par un nombre (un scalaire)
print("2 * v =", 2 * vecteur_v)
```

3. Produit scalaire (Dot Product) On utilise l'opérateur @.

```
# Calcule le produit scalaire entre v et w
produit_scalaire = vecteur_v @ vecteur_w
print("Produit scalaire v @ w =", produit_scalaire)
```

4. Multiplication de matrices L'opérateur @ est aussi utilisé pour la multiplication de matrices.

```
matrice_B = np.array([[5, 6], [7, 8]])

# Calcule le produit matriciel
produit_matriciel = matrice_A @ matrice_B
print("Produit A @ B:\n", produit_matriciel)
```

Challenge pour vous !

Maintenant, à vous de jouer avec les transformations.

Votre mission :

1. Créez deux matrices 2x2, A et B, avec les valeurs de votre choix.
2. Calculez C = A @ B et D = B @ A.
3. Le résultat est-il le même ? Qu'est-ce que cela implique sur la multiplication des matrices ?
4. Créez un vecteur (un point 2D), par exemple v = [1, 1].
5. Appliquez la transformation A à ce vecteur en calculant v_transforme = A @ v. Quelles sont ses nouvelles coordonnées ?

6 Le Calcul Différentiel pour les nuls : Comprendre la "descente de gradient"

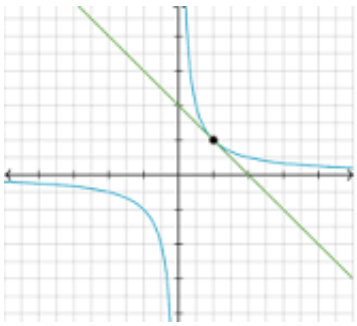
Comment une machine "apprend"-elle réellement ? Au cœur de nombreux algorithmes de machine learning se trouve un processus d'optimisation élégant appelé la descente de gradient. Pour le comprendre, oubliez les équations pour un instant et imaginez un randonneur aveugle, perdu dans une vallée brumeuse, qui cherche le point le plus bas. Il ne voit rien, mais il a un bâton. En posant son bâton à côté de son pied, il peut sentir la pente du terrain. Pour descendre, il lui suffit de faire un petit pas dans la direction où la pente est la plus forte vers le bas. Il répète ce processus encore et encore, et finit par atteindre le fond de la vallée.

Dans cette analogie, la vallée est notre "erreur", le bâton est la dérivée, et la stratégie de déplacement est la descente de gradient.

Le vocabulaire de l'optimisation

Qu'est-ce qu'une dérivée ? En termes simples, la dérivée d'une fonction en un point donné est la pente de la ligne tangente à ce point. Elle nous donne deux informations cruciales :

1. La direction : Si la pente est positive, la fonction monte. Si elle est négative, elle descend.
2. L'intensité : Une pente très forte (positive ou négative) signifie que la fonction change rapidement. Une pente proche de zéro signifie qu'on est sur un plat ou près d'un sommet/creux. C'est le "bâton" de notre randonneur qui lui indique où se trouve la descente.



Qu'est-ce qu'une fonction de coût ? Une fonction de coût (ou fonction de perte, *cost/loss function*) est une mesure de l'erreur d'un modèle. Elle prend les prédictions de notre modèle et les compare aux vraies valeurs, puis nous renvoie un seul nombre qui quantifie à quel point le modèle est "mauvais". Un score élevé signifie de grosses erreurs ; un score bas signifie que le modèle est performant. Le but de l'entraînement d'un modèle est de trouver les paramètres qui rendent ce score de coût le plus bas possible. C'est la "vallée" que notre randonneur cherche à explorer.

Qu'est-ce que le gradient ? Le gradient est simplement la généralisation de la dérivée à des fonctions avec plusieurs variables (ce qui est presque toujours le cas en machine learning). Au lieu d'être un simple nombre (une pente), le gradient est un vecteur (une flèche) qui pointe dans la direction de la plus forte pente ascendante. Pour trouver le point le plus bas de notre vallée, il nous suffit donc de faire un pas dans la direction exactement opposée à celle du gradient.

IMPLÉMENTER LA DESCENTE DE GRADIENT

Mettons en pratique cette idée avec NumPy et Matplotlib. Nous allons trouver le minimum de la fonction très simple $f(x) = x^2$.

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

# --- 1. Définir la fonction et sa dérivée (le gradient) ---
# La fonction de coût (notre "vallée")
def f(x):
    return x**2

# La dérivée de f(x), qui nous donne la pente
def derivative_f(x):
    return 2*x

# --- 2. Algorithme de la descente de gradient ---
x_initial = -8.0              # Point de départ du randonneur
learning_rate = 0.1           # Taux d'apprentissage (la taille du pas)
iterations = 25               # Le nombre de pas à effectuer

# On stocke l'historique des positions pour la visualisation
history = [x_initial]
x_current = x_initial

for i in range(iterations):
    # On calcule la pente (le gradient) à notre position actuelle
    gradient = derivative_f(x_current)
    # On fait un pas dans la direction opposée au gradient
    x_current = x_current - learning_rate * gradient
    # On enregistre notre nouvelle position
    history.append(x_current)

print(f"Point de départ : {x_initial}")
print(f"Minimum trouvé après {iterations} itérations : x =
{x_current:.4f}")

# --- 3. Visualisation ---
# Créer les données pour tracer la courbe de la fonction f(x)
x_curve = np.linspace(-10, 10, 200)
y_curve = f(x_curve)

# Convertir notre historique en tableau NumPy pour le tracé
x_history = np.array(history)
y_history = f(x_history)
```

```
# Créer le graphique
plt.figure(figsize=(12, 7))
plt.plot(x_curve, y_curve, label="Fonction de coût f(x) = x²", zorder=1)
plt.scatter(x_history, y_history, c='red', s=50, label="Étapes de la
descente", zorder=2)
plt.title("Visualisation de la Descente de Gradient", fontsize=16)
plt.xlabel("x", fontsize=12)
plt.ylabel("f(x)", fontsize=12)
plt.legend()
plt.grid(True)
plt.show()
```

Comme vous pouvez le voir sur le graphique, notre algorithme part de x=-8 et "dégringole" le long de la courbe pour s'approcher du point le plus bas, à x=0.

CHALLENGE POUR VOUS !

Le monde réel est rarement aussi simple qu'une seule vallée. Parfois, il y a plusieurs creux (des minimums locaux).

Votre mission : Prenez la fonction $f(x) = x^4 - 4x^2 + 2$. Sa dérivée est $f'(x) = 4x^3 - 8x$.

1. Modifiez le code ci-dessus pour utiliser cette nouvelle fonction et sa dérivée.
2. Lancez l'algorithme une première fois en partant de x_initial = 3.
3. Lancez-le une seconde fois en partant de x_initial = -3.

Observez-vous le même résultat final ? Qu'est-ce que cela implique sur l'importance du point de départ dans un algorithme de descente de gradient ? Partagez vos conclusions en commentaire !

7 LES PROBABILITÉS POUR LA DATA SCIENCE : PLUS QU'UN SIMPLE JEU DE DÉS !

Le Machine Learning ne prédit que très rarement des certitudes absolues. Un modèle ne vous dira pas "ce client va résilier à 100%", mais plutôt "il y a 85% de chances que ce client résilie". L'incertitude est au cœur de la data science. Comprendre le langage des probabilités est donc la première étape pour maîtriser l'IA et interpréter correctement ses résultats. C'est l'art de quantifier le doute.

LE LANGAGE DE L'INCERTITUDE

Qu'est-ce qu'une probabilité ? Une probabilité est un nombre entre 0 et 1 qui mesure la chance qu'un événement se produise. Il y a deux grandes manières de voir les choses :

- L'approche fréquentiste : C'est l'approche la plus intuitive. La probabilité d'un événement est la fréquence à laquelle il apparaît si on répète une expérience un très grand nombre de fois. Si vous lancez un dé équilibré 6 millions de fois, vous obtiendrez environ 1 million de "6". La probabilité est donc de 1/6.

- L'approche bayésienne : Ici, une probabilité est une mesure de notre degré de croyance en une proposition. Ce degré de croyance peut être mis à jour à la lumière de nouvelles preuves. C'est une vision plus flexible et fondamentale en machine learning moderne.

Probabilité conditionnelle et le Théorème de Bayes La probabilité conditionnelle, notée P(A|B), est la probabilité que l'événement A se produise sachant que l'événement B s'est déjà produit. Par exemple, la probabilité qu'il pleuve sachant qu'il y a des nuages est plus élevée que la probabilité qu'il pleuve en général.

Cette idée mène directement au Théorème de Bayes, une des formules les plus importantes en data science. Il nous permet d'inverser les probabilités conditionnelles, c'est-à-dire de calculer P(B|A) si on connaît P(A|B).

- Exemple classique du test médical :
    - Une maladie touche 1% de la population P(Malade) = 0.01.
    - Un test de dépistage est fiable à 99% :
        - Si vous êtes malade, il est positif à 99% P(Positif|Malade) = 0.99.
        - Si vous êtes sain, il est négatif à 99% (donc positif à 1%) P(Positif|Sain) = 0.01.
- Question : Si votre test est positif, quelle est la probabilité que vous soyez réellement malade P(Malade|Positif) ? L'intuition dit 99%, mais le Théorème de Bayes nous donne la vraie réponse : environ 50% ! Ce résultat contre-intuitif montre à quel point il est crucial de raisonner correctement avec les probabilités.

Variables aléatoires et distributions Une variable aléatoire est une variable dont la valeur est le résultat numérique d'un phénomène aléatoire. Le résultat d'un lancer de dé est une variable aléatoire. Ces variables suivent souvent des schémas reconnaissables, appelés distributions de probabilité.

- La Loi de Bernoulli : C'est la plus simple. Elle décrit une expérience avec seulement deux issues (succès/échec, pile/face, 1/0).
- La Loi Normale (ou Courbe de Gauss) : C'est la reine des distributions. [Image d'une courbe de Gauss en cloche]. De très nombreux phénomènes naturels (la taille des individus, les erreurs de mesure...) suivent cette fameuse courbe en cloche. C'est une hypothèse de base dans de nombreux modèles statistiques.

LES PROBABILITÉS EN CODE

1. Simuler 1000 lancers de dés avec NumPy

```
import numpy as np

# Simule 1000 lancers d'un dé à 6 faces
lancers = np.random.randint(1, 7, size=1000)

# Calcule la fréquence d'apparition du "6"
frequence_6 = np.sum(lancers == 6) / 1000
```

```
print(f"Fréquence observée du '6' sur 1000 lancers : {frequence_6:.3f}") #
Devrait être proche de 1/6 ≈ 0.167
```

2. Calculer des probabilités sur le Titanic avec Pandas (Suppose que le fichier train.csv du Titanic est chargé dans un DataFrame df)

```
import pandas as pd
# df = pd.read_csv('train.csv') # Ligne à décommenter si vous exécutez le
code

# P(Survivant)
prob_survivant = df['Survived'].mean()
print(f"P(Survivant) = {prob_survivant:.2f}")

# P(Femme)
prob_femme = (df['Sex'] == 'female').mean()
print(f"P(Femme) = {prob_femme:.2f}")

# P(Survivant | Femme)
prob_survivant_sachant_femme = df[df['Sex'] == 'female']['Survived'].mean()
print(f"P(Survivant | Femme) = {prob_survivant_sachant_femme:.2f}")

3. Visualiser une distribution normale avec Seaborn
import seaborn as sns
import matplotlib.pyplot as plt

# Génère 1000 points suivant une loi normale
donnees_normales = np.random.randn(1000)

# Crée un histogramme avec une courbe de densité
sns.histplot(donnees_normales, kde=True)
plt.title("Distribution Normale Standard")
plt.show()
```

Challenge : Le Problème de Monty Hall

C'est un célèbre casse-tête de probabilités qui a trompé même des mathématiciens. Le jeu :

1. Vous êtes face à 3 portes fermées. Derrière l'une se trouve une voiture, derrière les deux autres, des chèvres.
2. Vous choisissez une porte (disons, la porte 1).
3. L'animateur, qui sait où est la voiture, ouvre une autre porte (disons, la porte 3) derrière laquelle il y a une chèvre.
4. Il vous demande alors : "Voulez-vous changer votre choix et prendre la porte 2 ?"

Votre mission : Écrivez un script Python qui simule 10 000 parties pour prouver s'il vaut mieux garder son choix initial ou changer de porte.

```
import numpy as np
```

```python
def simulate_monty_hall(n_simulations=10000):
    victoires_en_gardant = 0
    victoires_en_changeant = 0

    for _ in range(n_simulations):
        # Les portes : 0, 1, 2. La voiture est derrière une porte au
hasard.
        porte_voiture = np.random.randint(0, 3)
        # Votre choix initial
        choix_initial = np.random.randint(0, 3)

        # L'animateur ouvre une porte qui n'est ni votre choix, ni la
voiture
        portes_restantes = [i for i in range(3) if i != choix_initial and i
!= porte_voiture]
        porte_ouverte = portes_restantes[0]

        # Stratégie 1 : Garder le choix initial
        if choix_initial == porte_voiture:
            victoires_en_gardant += 1

        # Stratégie 2 : Changer de choix
        # Le nouveau choix est la seule porte qui n'est ni le choix
initial, ni la porte ouverte
        nouveau_choix = [i for i in range(3) if i != choix_initial and i !=
porte_ouverte][0]
        if nouveau_choix == porte_voiture:
            victoires_en_changeant += 1

    return victoires_en_gardant, victoires_en_changeant

n_sims = 10000
garder, changer = simulate_monty_hall(n_sims)

print(f"Sur {n_sims} simulations :")
print(f"Taux de victoire en GARDANT son choix : {garder/n_sims:.2%}")
print(f"Taux de victoire en CHANGEANT son choix : {changer/n_sims:.2%}")
```

Alors, que dit la simulation ?

8 Les Statistiques Descriptives : L'art de faire parler vos données

Avant de construire des modèles d'intelligence artificielle complexes ou des visualisations sophistiquées, un bon data scientist fait une chose simple mais essentielle : il écoute ses données. Les statistiques descriptives sont les outils de cette première conversation. Elles nous permettent de résumer, de caractériser et de comprendre la structure d'un jeu de données sans chercher à en tirer des conclusions sur une population plus large. C'est l'art du premier contact.

Les outils de l'écoute

Mesures de tendance centrale : Où se situe le "centre" ? Ces mesures nous donnent une idée de la valeur "typique" ou "centrale" de nos données.

- La Moyenne (mean) : La somme de toutes les valeurs divisée par leur nombre. C'est la plus connue, mais elle est très sensible aux valeurs extrêmes (aberrantes).
- La Médiane (median) : La valeur du milieu d'un ensemble de données triées. 50% des données sont en dessous, 50% sont au-dessus. Elle n'est pas affectée par les valeurs extrêmes, ce qui la rend idéale pour des données comme les salaires ou les prix de l'immobilier.
- Le Mode (mode) : La valeur qui apparaît le plus fréquemment. C'est la seule mesure de tendance centrale utilisable pour des données non numériques (catégorielles), comme la couleur de voiture la plus vendue.

Mesures de dispersion : À quel point les données sont-elles étalées ? Il ne suffit pas de connaître le centre, il faut aussi savoir si les données sont regroupées ou très dispersées.

- L'Étendue (range) : La différence entre la valeur maximale et la valeur minimale. Simple, mais très sensible aux extrêmes.
- La Variance et l'Écart-type (variance, standard deviation) : L'écart-type est la mesure de dispersion la plus importante. Il représente la distance "moyenne" de chaque point de donnée par rapport à la moyenne du groupe. Une petite valeur indique que les données sont très regroupées autour de la moyenne ; une grande valeur indique qu'elles sont très étalées. La variance est simplement le carré de l'écart-type.

Le concept de distribution : Quelle est la forme de nos données ? Une distribution nous montre la fréquence d'apparition de chaque valeur dans un jeu de données. L'outil le plus courant pour la visualiser est l'histogramme. En regardant un histogramme, on peut voir si la distribution est :

- Symétrique : Comme la fameuse courbe en cloche de la loi normale, où la moyenne, la médiane et le mode sont confondus.
- Asymétrique (skewed) : La plupart des valeurs sont concentrées d'un côté. Par exemple, la distribution des revenus est souvent asymétrique à droite, avec beaucoup de salaires bas/moyens et quelques salaires très élevés qui "tirent" la moyenne vers le haut.

La conversation avec Pandas

1. Utiliser .describe() pour un résumé rapide C'est la commande la plus utile pour commencer. En une seule ligne, elle vous donne la plupart des statistiques clés pour toutes les colonnes numériques.

```
import pandas as pd
# df = pd.read_csv('train.csv') # Titanic dataset

# Obtenir un résumé statistique complet
print(df.describe())
```

2. Calculer des statistiques individuelles Vous pouvez aussi calculer chaque mesure séparément.

```
age_moyen = df['Age'].mean()
age_median = df['Age'].median()
```

```
ecart_type_tarif = df['Fare'].std()

print(f"Âge moyen : {age_moyen:.2f}")
print(f"Âge médian : {age_median:.2f}")
print(f"Écart-type du tarif : {ecart_type_tarif:.2f}")
```

3. Utiliser .value_counts() pour les catégories Pour les données non numériques, .value_counts() est parfait pour compter les occurrences de chaque catégorie.

```
# Compter le nombre de passagers par classe
print(df['Pclass'].value_counts())
```

4. Créer un histogramme avec .hist() Pandas permet de visualiser rapidement la distribution d'une colonne.

```
import matplotlib.pyplot as plt

df['Age'].hist(bins=20) # bins contrôle le nombre de barres
plt.title("Distribution de l'âge des passagers")
plt.xlabel("Âge")
plt.ylabel("Nombre de passagers")
plt.show()
```

Challenge pour vous !

Les statistiques descriptives sont encore plus puissantes lorsqu'on les utilise pour comparer des groupes.

Votre mission : En utilisant le dataset du Titanic :

1. Séparez le DataFrame en deux : un pour les passagers qui ont survécu (Survived == 1) et un pour ceux qui ne l'ont pas fait (Survived == 0).
2. Utilisez .describe() sur les colonnes 'Age' et 'Fare' pour chacun de ces deux groupes.
3. Comparez les résultats.

Quelles premières hypothèses pouvez-vous formuler en regardant les différences de moyenne d'âge ou de tarif payé entre les survivants et les non-survivants ?

9 Introduction à la Visualisation de Données avec Matplotlib

Une image vaut mille lignes de données. Après avoir appris à résumer les données avec les statistiques, il est temps de les voir. La visualisation est l'un des outils les plus puissants du data scientist pour explorer un jeu de données, identifier des tendances, repérer des anomalies et communiquer ses résultats de manière percutante. Matplotlib est la bibliothèque historique et fondamentale de l'écosystème Python pour transformer les nombres bruts en informations visuelles claires.

L'anatomie d'un graphique

Pour créer des graphiques efficaces, il faut connaître les éléments qui les composent.

- La Figure (Figure) : C'est la fenêtre ou la page globale qui contient tout. C'est le conteneur de plus haut niveau.
- Les Axes (Axes) : C'est le graphique en lui-même, la zone où les données sont tracées. Une figure peut contenir plusieurs Axes (sous-graphiques). C'est sur cet objet que l'on va dessiner nos points, lignes, barres, etc.
- Les Graduations (Ticks) : Ce sont les petites marques sur les axes X et Y qui indiquent des points spécifiques. Il y a les graduations majeures (major ticks) et mineures (minor ticks).
- Les Étiquettes (Labels) : Ce sont les noms que l'on donne aux axes (xlabel, ylabel) pour expliquer ce que les données représentent.
- Le Titre (Title) : Le titre principal du graphique, qui doit résumer son contenu.

Quand utiliser quel type de graphique ?

- Graphique linéaire (line plot) : Idéal pour visualiser l'évolution d'une variable continue dans le temps. (Ex: évolution du livre d'une action).
- Diagramme à barres (bar chart) : Parfait pour comparer des quantités entre différentes catégories. (Ex: ventes par pays).
- Nuage de points (scatter plot) : Essentiel pour visualiser la relation entre deux variables numériques. (Ex: relation entre la taille et le poids d'une personne).
- Histogramme (histogram) : Utilisé pour visualiser la distribution d'une seule variable numérique. (Ex: distribution des âges dans une population).

L'importance d'un bon étiquetage : Un graphique sans titre ni étiquettes sur les axes est inutile. Personne ne peut le comprendre. Un bon étiquetage est la forme la plus élémentaire de politesse scientifique : il permet à votre audience de comprendre instantanément ce qu'elle regarde.

LE FLUX DE TRAVAIL DE BASE

Le flux de travail avec Matplotlib (en utilisant son sous-module pyplot, que l'on importe conventionnellement sous l'alias plt) est généralement le suivant :

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

# 1. Créer une Figure et des Axes
fig, ax = plt.subplots()

# 2. Tracer les données sur les Axes
# ... code pour tracer ...

# 3. Personnaliser (titres, étiquettes...)
ax.set_title("Titre du Graphique")
ax.set_xlabel("Axe des X")
ax.set_ylabel("Axe des Y")
```

```
# 4. Afficher le graphique
plt.show()
```

Exemple 1 : Graphique linéaire

```
x = np.linspace(0, 10, 100)
y = x**2

fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title("Évolution de y = x²")
ax.set_xlabel("x")
ax.set_ylabel("y")
plt.show()
```

Exemple 2 : Diagramme à barres

```
categories = ['A', 'B', 'C']
valeurs = [10, 25, 18]

fig, ax = plt.subplots()
ax.bar(categories, valeurs)
ax.set_title("Comparaison des catégories")
ax.set_ylabel("Valeurs")
plt.show()
```

Exemple 3 : Nuage de points

```
# Données aléatoires pour l'exemple
x_scatter = np.random.rand(50) * 10
y_scatter = x_scatter + np.random.randn(50) * 2

fig, ax = plt.subplots()
ax.scatter(x_scatter, y_scatter)
ax.set_title("Relation entre deux variables")
ax.set_xlabel("Variable 1")
ax.set_ylabel("Variable 2")
plt.show()
```

CHALLENGE POUR VOUS !

La puissance de Matplotlib réside dans sa capacité à composer des visualisations complexes.

Votre mission : Créez une seule figure contenant deux sous-graphiques côte à côte.

1. À gauche : Tracez une onde sinusoïdale (de 0 à 2π).
2. À droite : Tracez une onde cosinusoïdale (de 0 à 2π).
3. Donnez à chaque sous-graphique son propre titre et étiquetez ses axes.
4. Donnez un titre principal à l'ensemble de la figure.

10 SEABORN : DES GRAPHIQUES STATISTIQUES MAGNIFIQUES EN UNE SEULE LIGNE DE CODE

Dans l'chapitre précédent, nous avons appris à construire des graphiques avec Matplotlib, le moteur de la visualisation en Python. Si Matplotlib est le moteur, puissant et flexible, alors Seaborn est le tableau de bord, beau, convivial et intelligent. C'est une bibliothèque qui s'appuie sur Matplotlib pour nous permettre de créer des graphiques statistiques époustouflants avec un minimum d'effort et souvent, en une seule ligne de code.

POURQUOI UTILISER SEABORN ?

Comment Seaborn s'appuie sur Matplotlib ? Seaborn n'est pas un remplaçant de Matplotlib, mais un complément. Il simplifie des tâches qui seraient complexes avec Matplotlib :

1. Fonctions de plus haut niveau : Seaborn propose des fonctions dédiées à des types de graphiques statistiques spécifiques (comme les boîtes à moustaches ou les cartes de chaleur), ce qui évite d'avoir à les construire manuellement.

2. Meilleure esthétique par défaut : Les graphiques Seaborn ont une apparence plus moderne et plus lisible dès le départ, avec des palettes de couleurs intelligentes.

3. Intégration parfaite avec Pandas : Seaborn est conçu pour fonctionner directement avec les DataFrames Pandas. Vous pouvez simplement lui dire "utilise telle colonne pour l'axe X et telle autre pour l'axe Y" et il s'occupe du reste.

Types de graphiques clés dans Seaborn Seaborn excelle dans la création de graphiques qui révèlent la structure de vos données.

- Boîte à moustaches (boxplot) : Idéale pour comparer la distribution d'une variable numérique à travers différentes catégories. Elle montre la médiane, les quartiles et les valeurs aberrantes.

- Graphique en violon (violinplot) : Similaire à la boîte à moustaches, mais il montre aussi la densité de la distribution de chaque côté. C'est une version plus riche et plus informative.

- Carte de chaleur (heatmap) : Parfaite pour visualiser une matrice de données, où les valeurs sont représentées par des couleurs. On l'utilise très souvent pour afficher les matrices de corrélation.

- Graphique par paires (pairplot) : C'est un outil d'exploration incroyable. Il crée une grille de graphiques montrant la relation entre chaque paire de variables dans votre jeu de données, ainsi que la distribution de chaque variable sur la diagonale.

SEABORN EN ACTION

La convention est d'importer Seaborn sous l'alias sns.

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

```
# On charge le dataset du Titanic pour nos exemples
df = pd.read_csv('train.csv')
```

1. Créer une boîte à moustaches Comparons la distribution des âges pour chaque classe de passagers.

```
plt.figure(figsize=(10, 6))
sns.boxplot(x='Pclass', y='Age', data=df)
plt.title("Distribution de l'âge par classe de passager")
plt.show()
```

2. Créer une carte de chaleur d'une matrice de corrélation La corrélation mesure la relation linéaire entre deux variables. Une carte de chaleur est le meilleur moyen de la visualiser.

```
# On ne garde que les colonnes numériques pour calculer la corrélation
numeric_df = df[['Survived', 'Pclass', 'Age', 'SibSp', 'Parch', 'Fare']]
correlation_matrix = numeric_df.corr()

plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title("Matrice de corrélation des variables du Titanic")
plt.show()
```

3. Créer un graphique par paires Explorons les relations entre plusieurs variables en une seule commande.

```
# On utilise sns.pairplot sur un sous-ensemble de colonnes
sns.pairplot(df[['Survived', 'Age', 'Fare', 'Pclass']], hue='Survived')
plt.show()
```

Le paramètre hue colore les points en fonction d'une variable catégorielle (ici, la survie).

CHALLENGE POUR VOUS !

Seaborn est livré avec quelques jeux de données classiques pour s'entraîner. Nous allons utiliser le dataset "tips", qui contient des informations sur les pourboires laissés dans un restaurant.

Votre mission :

1. Chargez le jeu de données tips avec la commande tips_df = sns.load_dataset('tips').
2. Créez un graphique en violon (violinplot) qui montre la distribution de l'addition totale (total_bill) pour chaque jour de la semaine (day).
3. Améliorez ce graphique en utilisant le paramètre hue='sex' pour séparer davantage les distributions à l'intérieur de chaque jour en fonction du sexe du client.

11 VOTRE PREMIER PROJET DE A À Z (PARTIE 1) : TROUVER ET NETTOYER LES DONNÉES

La théorie, c'est bien, mais la data science est un sport qui se pratique. Après avoir exploré les concepts et les outils fondamentaux, il est temps de mettre les mains dans le cambouis avec un véritable projet

de A à Z. Et pour cela, nous allons utiliser le défi le plus célèbre pour les débutants : prédire la survie sur le Titanic.

Cette première étape n'est pas la plus glamour, mais elle est de loin la plus importante. Avant toute analyse ou modélisation, il faut nettoyer les données.

Le workflow du Data Scientist

Un projet de data science suit généralement un cycle de vie bien défini :

1. Question : Définir l'objectif. Ici : "Peut-on prédire qui a survécu au naufrage du Titanic en se basant sur les données des passagers ?"
2. Collecte : Rassembler les données nécessaires. (Nous utiliserons un fichier CSV classique).
3. Nettoyage : Préparer et nettoyer les données. C'est notre focus aujourd'hui.
4. Exploration : Visualiser et analyser les données pour trouver des tendances (ce que nous avons fait avec les statistiques et Seaborn).
5. Modélisation : Construire un modèle de machine learning pour faire des prédictions.
6. Communication : Présenter les résultats de manière claire et actionnable.

Tâches courantes de nettoyage :

- Gérer les valeurs manquantes : Que faire quand des informations sont absentes ? On peut les supprimer, ou les "imputer" (les remplacer par une valeur logique comme la moyenne ou la médiane).
- Corriger les types de données : S'assurer que les nombres sont bien des nombres et les dates des dates.
- Gérer les outliers (valeurs aberrantes) : Identifier et décider quoi faire des points de données qui sont très différents du reste.

Pourquoi cette étape est-elle si cruciale ? Il existe un adage en informatique : "Garbage In, Garbage Out" (Déchets en entrée, déchets en sortie). Vous pouvez avoir le meilleur algorithme du monde, si vous le nourrissez avec des données de mauvaise qualité, ses prédictions seront inutiles. Le nettoyage des données représente souvent jusqu'à 80% du temps d'un projet.

Le grand nettoyage du Titanic

Commençons par charger les données et les inspecter.

```
import pandas as pd
import numpy as np

# Charger les données
df = pd.read_csv('train.csv')

# Première inspection
print(df.info())
```

1. Identifier les données manquantes La méthode .info() nous donne déjà un bon aperçu. Pour un compte précis :

```
# Afficher le nombre de valeurs manquantes par colonne
print(df.isnull().sum())
```

On voit que 'Age', 'Cabin' et 'Embarked' ont des valeurs manquantes.

2. Imputer l'âge avec la médiane L'âge est une information importante. Plutôt que de supprimer les lignes, nous allons remplacer les âges manquants par l'âge médian du dataset (la médiane est plus robuste aux âges extrêmes que la moyenne).

```
age_median = df['Age'].median()
df['Age'].fillna(age_median, inplace=True)
```

3. Supprimer la colonne 'Cabin' La colonne 'Cabin' a trop de valeurs manquantes (plus de 77%) pour être utilement remplie. La meilleure solution est de la supprimer.

```
df.drop('Cabin', axis=1, inplace=True)
```

axis=1 précise qu'on supprime une colonne, pas une ligne.

4. Remplir 'Embarked' avec le mode Il ne manque que deux valeurs pour le port d'embarquement ('Embarked'). Comme c'est une variable catégorielle, nous allons la remplir avec la valeur la plus fréquente (le mode).

```
embarked_mode = df['Embarked'].mode()[0]
df['Embarked'].fillna(embarked_mode, inplace=True)

# Vérification finale : plus aucune valeur manquante !
print(df.isnull().sum())
```

CHALLENGE POUR VOUS !

Parfois, le nettoyage se transforme en création de nouvelles informations. C'est le début du "feature engineering".

La colonne 'Name' contient des titres comme 'Mr.', 'Mrs.', 'Miss.', 'Master.', etc. Ces titres peuvent contenir des informations précieuses sur l'âge, le sexe ou le statut social.

Votre mission :

1. Créez une nouvelle colonne 'Title' dans le DataFrame.
2. Pour chaque passager, extrayez son titre de la colonne 'Name' et placez-le dans la nouvelle colonne 'Title'. (Indice : vous pouvez utiliser df['Name'].str.extract(' ([A-Za-z]+)\.')).
3. Comptez combien de titres uniques différents vous avez trouvés avec .value_counts().

12 VOTRE PREMIER PROJET DE A À Z (PARTIE 2) : EXPLORATION ET VISUALISATION

Nos données sont propres. Le travail fastidieux est terminé, et maintenant, le plaisir commence ! Dans cette deuxième partie, nous allons enfiler notre casquette de détective. En utilisant la puissance de la visualisation de données, nous allons interroger notre dataset pour qu'il nous raconte les histoires cachées du Titanic. Qui a survécu ? Pourquoi ? Les graphiques nous aideront à trouver les premières pistes.

L'ART DE L'EXPLORATION

Qu'est-ce que l'Analyse Exploratoire des Données (EDA) ? L'Analyse Exploratoire des Données (ou *Exploratory Data Analysis* en anglais) est une approche qui consiste à analyser et visualiser des jeux de données pour en résumer les principales caractéristiques, souvent à l'aide de graphiques. Le but n'est pas de prouver une hypothèse, mais d'en découvrir. C'est une phase de curiosité et d'investigation.

Analyse univariée vs bivariée

- Analyse univariée : On examine les variables une par une, indépendamment les unes des autres. L'objectif est de comprendre la distribution de chaque variable. (Ex: "Quelle était la répartition des âges sur le bateau ?").
- Analyse bivariée : On examine la relation entre deux variables simultanément. C'est ici que les histoires commencent à émerger. (Ex: "Le sexe d'une personne a-t-il eu un impact sur ses chances de survie ?").

Comment formuler des hypothèses à partir de graphiques ? Une hypothèse est une supposition éclairée. Si un graphique montre que la barre de survie des femmes est trois fois plus haute que celle des hommes, on peut formuler l'hypothèse suivante : "Le sexe a été un facteur déterminant dans la survie, les femmes ayant eu une probabilité de survie bien plus élevée". L'EDA nous fournit ces hypothèses, que la modélisation cherchera ensuite à confirmer.

FAIRE PARLER LES DONNÉES DU TITANIC

Nous utiliserons Seaborn pour sa simplicité et l'élégance de ses graphiques. Assurez-vous d'avoir votre DataFrame df nettoyé de la partie 1.

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

# Suppose que 'df' est votre DataFrame nettoyé de la partie 1
# df = pd.read_csv(...) et les étapes de nettoyage...
```

Analyse univariée : Qui était à bord ?

```
# Distribution des survivants
sns.countplot(x='Survived', data=df)
plt.title('Distribution des Survivants (0 = Non, 1 = Oui)')
plt.show()

# Distribution des classes
sns.countplot(x='Pclass', data=df)
```

```
plt.title('Distribution des Passagers par Classe')
plt.show()

# Distribution des âges
sns.histplot(df['Age'], bins=30, kde=True)
plt.title('Distribution des Âges')
plt.show()
```

Analyse bivariée : Qui a survécu ?

```
# Taux de survie par sexe
sns.barplot(x='Sex', y='Survived', data=df)
plt.title('Taux de Survie par Sexe')
plt.show()

# Taux de survie par classe
sns.barplot(x='Pclass', y='Survived', data=df)
plt.title('Taux de Survie par Classe de Passager')
plt.show()
```

Les premiers graphiques racontent déjà une histoire claire : les femmes et les passagers de première classe avaient de bien meilleures chances de survie.

CHALLENGE POUR VOUS !

Maintenant, explorons une relation un peu plus complexe.

Votre mission : Créez une visualisation qui explore la relation entre le tarif payé (Fare) et la survie.

1. Y a-t-il un lien apparent entre le prix du billet et le fait de survivre ?
2. Attention : La colonne Fare est très asymétrique (quelques billets très chers "écrasent" le graphique). Pour une meilleure visualisation, vous pouvez créer une nouvelle colonne avec une transformation logarithmique, par exemple df['LogFare'] = np.log1p(df['Fare']).
3. Utilisez un boxplot ou un violinplot pour comparer la distribution des tarifs (ou des tarifs logarithmiques) entre les survivants et les non-survivants.

PARTIE 2 LE MACHINE LEARNING

---

13 MACHINE LEARNING : MON ORDINATEUR PEUT-IL VRAIMENT APPRENDRE ?

Comment apprend-on à un très jeune enfant à reconnaître un chat ? On ne lui donne pas une liste de règles du type "s'il a des moustaches, des oreilles pointues et qu'il miaule, alors c'est un chat". C'est trop complexe. À la place, on lui montre des exemples. On lui montre une photo et on lui dit "ça, c'est un chat". On lui en montre une autre : "ça aussi, c'est un chat". On lui montre un chien : "ça, ce n'est pas

un chat". À force d'exemples, l'enfant finit par construire son propre modèle mental et par généraliser pour reconnaître des chats qu'il n'a jamais vus auparavant.

Le Machine Learning (ou Apprentissage Automatique), c'est exactement la même idée, mais appliquée à des ordinateurs.

LES 3 GRANDES FAMILLES DE L'APPRENTISSAGE

Le Machine Learning est un vaste domaine, mais la plupart des méthodes peuvent être classées dans trois grandes familles.

1. L'Apprentissage Supervisé (Supervised Learning) C'est la méthode la plus courante, et celle qui ressemble le plus à notre exemple de l'enfant et du chat. L'idée est d'entraîner un modèle sur des données étiquetées, c'est-à-dire des exemples où l'on connaît déjà la "bonne réponse". C'est comme apprendre avec un professeur qui vous fournit les questions et les réponses, et votre but est d'apprendre à trouver les réponses par vous-même pour de nouvelles questions.

- Exemple : Prédire le prix d'un appartement. On donne au modèle des milliers d'exemples d'appartements (surface, nombre de pièces, quartier) avec leur prix de vente réel (l'étiquette). Le modèle apprend la relation entre les caractéristiques et le prix.

2. L'Apprentissage Non Supervisé (Unsupervised Learning) Ici, on ne donne pas les réponses au modèle. On lui donne un ensemble de données non étiquetées et on lui demande de trouver une structure cachée, de mettre de l'ordre dans le chaos par lui-même. C'est comme donner une boîte de LEGO en vrac à quelqu'un et lui demander de regrouper les briques par couleur et par forme, sans lui dire quelles sont les couleurs ou les formes à chercher.

- Exemple : La segmentation client. On donne à un modèle les données d'achat de milliers de clients. Le modèle va créer des groupes de clients aux comportements similaires (les "acheteurs fréquents", les "chasseurs de promos", etc.) sans qu'on lui ait défini ces groupes au préalable.

3. L'Apprentissage par Renforcement (Reinforcement Learning) C'est le mode d'apprentissage le plus proche de la manière dont nous, humains, apprenons de nouvelles compétences. L'algorithme (appelé "l'agent") apprend par essais et erreurs. Il évolue dans un environnement et prend des actions. Chaque action lui rapporte une récompense (positive) ou une punition (négative). Son seul but est de maximiser la somme des récompenses sur le long terme.

- Exemple : Entraîner une IA à jouer à un jeu vidéo. L'IA essaie des actions au hasard. Si elle gagne des points (récompense), elle essaiera de refaire cette action dans une situation similaire. Si elle perd une vie (punition), elle évitera cette action à l'avenir.

DES EXEMPLES POUR CHAQUE FAMILLE

Comme promis, pas de code dans cet chapitre. L'objectif est de bien saisir les concepts. Voici un résumé visuel des exemples que nous venons de voir :

- Supervisé : Prédiction de prix
    - Entrée : Données de maisons (surface, chambres) + Prix

    - Sortie : Un modèle capable de prédire le prix d'une nouvelle maison.
- Non Supervisé : Segmentation client
    - Entrée : Données d'achats des clients
    - Sortie : Des groupes (clusters) de clients similaires.
- Par Renforcement : IA de jeu vidéo
    - Entrée : État du jeu (position du joueur, des ennemis)
    - Sortie : La meilleure action à faire (aller à gauche, sauter...).

CHALLENGE POUR VOUS !

Pour chacun des scénarios suivants, identifiez s'il s'agit d'apprentissage supervisé, non supervisé ou par renforcement.

1. Un service météo veut prédire la température de demain en se basant sur les données historiques des 50 dernières années.
2. Un site d'information veut regrouper automatiquement des milliers d'chapitres d'actualité par sujet (sport, politique, technologie) sans catégories prédéfinies.
3. On veut entraîner un robot à marcher dans un environnement inconnu. Le robot reçoit une récompense pour chaque mètre parcouru sans tomber.

14 LA RÉGRESSION LINÉAIRE : PRÉDIRE LE FUTUR AVEC UNE SIMPLE LIGNE DROITE

Imaginez un nuage de points représentant le salaire de plusieurs employés en fonction de leurs années d'expérience. On voit bien une tendance : plus on a d'expérience, plus le salaire est élevé. Et si on pouvait tracer la "meilleure" ligne droite possible à travers ces points ? On pourrait alors l'utiliser pour prédire le salaire de quelqu'un qui a 10 ans d'expérience, même si cette personne n'est pas dans nos données initiales.

C'est exactement ce que fait la régression linéaire. C'est notre premier véritable modèle de machine learning supervisé.

L'ÉQUATION D'UNE PRÉDICTION

Expliquer l'équation $y = mx + b$ Vous vous souvenez peut-être de cette équation du collège. En machine learning, elle prend tout son sens :

- y : C'est la valeur que l'on veut prédire (la variable dépendante, ex: le salaire).
- x : C'est la variable que l'on utilise pour prédire (la variable indépendante, ex: les années d'expérience).
- m : C'est la pente de la droite (le *coefficient*). Elle nous dit de combien y augmente quand x augmente d'une unité. (Ex: "Chaque année d'expérience supplémentaire ajoute en moyenne 2000€ au salaire").

- b : C'est l'ordonnée à l'origine (l'*intercept*). C'est la valeur de y quand x est égal à zéro. (Ex: "Le salaire de base à 0 année d'expérience est de 30 000€").

Comment le modèle "apprend"-il m et b ? "Entraîner" un modèle de régression linéaire signifie trouver les valeurs optimales pour m et b qui dessinent la ligne passant au plus près de tous les points de données. Mais comment définir "au plus près" ?

C'est là qu'intervient la fonction de coût. Pour chaque point, on mesure la distance verticale entre le point réel et notre droite de prédiction. Cette distance est l'erreur. Pour éviter que les erreurs positives et négatives ne s'annulent, on met chaque erreur au carré. Le but du modèle est de trouver la droite qui minimise la somme de toutes ces erreurs au carré.

Ce processus de minimisation vous rappelle quelque chose ? C'est exactement le principe de la descente de gradient que nous avons vue dans l'chapitre S6 ! L'algorithme ajuste petit à petit m et b pour "descendre la vallée" de la fonction de coût jusqu'à trouver le point le plus bas possible, c'est-à-dire la meilleure droite.

La Régression avec Scikit-Learn

Assez de théorie, passons au code avec la bibliothèque de machine learning la plus populaire : Scikit-Learn.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# 1. Créer des données d'exemple
np.random.seed(0)
X = 2 * np.random.rand(100, 1) # Années d'expérience
y = 4 + 3 * X + np.random.randn(100, 1) # Salaire

# 2. Diviser les données en ensembles d'entraînement et de test
# On entraîne le modèle sur 80% des données, on le teste sur les 20%
restants.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 3. Créer et entraîner le modèle
model = LinearRegression()
model.fit(X_train, y_train)

# 4. Faire des prédictions sur les données de test
y_pred = model.predict(X_test)

# 5. Visualiser la droite de régression
plt.figure(figsize=(10, 6))
plt.scatter(X_test, y_test, color='blue', label='Données réelles')
plt.plot(X_test, y_pred, color='red', linewidth=2, label='Droite de
régression')
plt.title('Régression Linéaire : Expérience vs Salaire')
```

```
plt.xlabel('Années d\'expérience')
plt.ylabel('Salaire')
plt.legend()
plt.show()

# Afficher les coefficients appris par le modèle
print(f"Pente (m) apprise : {model.coef_[0][0]:.2f}")
print(f"Ordonnée à l'origine (b) apprise : {model.intercept_[0]:.2f}")
```

CHALLENGE POUR VOUS !

Visualiser la droite, c'est bien. Quantifier sa performance, c'est mieux.

Votre mission :

1. Calculez l'Erreur Quadratique Moyenne (MSE - *Mean Squared Error*) et le score R² en utilisant les fonctions de Scikit-Learn sur vos données de test (y_test) et vos prédictions (y_pred).

2. Pouvez-vous interpréter ces deux chiffres avec des mots simples ?

Solution et Interprétation :

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"\nErreur Quadratique Moyenne (MSE) : {mse:.2f}")
print(f"Score R² : {r2:.2f}")
```

- MSE : "En moyenne, le carré de l'erreur entre notre prédiction de salaire et le salaire réel est de [valeur de MSE]. Plus ce chiffre est bas, mieux c'est."

- R² : "Notre modèle explique [valeur de R² en %] de la variation du salaire. Un score de 0.79 signifie que 79% des variations de salaire sont expliquées par les années d'expérience dans notre modèle. Plus on est proche de 1, mieux c'est."

15 LA RÉGRESSION LOGISTIQUE : CLASSIFIER LE MONDE EN "OUI" OU "NON"

Avec la régression linéaire, nous avons appris à prédire une valeur continue, comme un salaire. Mais que se passe-t-il si nous voulons prédire une catégorie ? "Spam" ou "Non Spam" ? "Malade" ou "Sain" ? "Survivant" ou "Non-survivant" ? Une ligne droite ne peut pas nous donner une réponse "Oui" ou "Non".

Pour cela, nous avons besoin d'un nouvel outil. Au lieu d'une ligne droite qui peut aller de -∞ à +∞, nous utilisons une courbe en forme de "S", appelée la fonction sigmoïde. Cette fonction magique prend n'importe quel score et le transforme élégamment en une probabilité, toujours comprise entre 0 et 1.

DE LA PROBABILITÉ À LA DÉCISION

La fonction sigmoïde La fonction sigmoïde est le cœur de la régression logistique. Quelle que soit la valeur que le modèle calcule en interne (un score très négatif ou très positif), la sigmoïde la "comprime"

pour qu'elle tienne dans l'intervalle [0, 1]. Le résultat peut alors être interprété comme une probabilité. Par exemple, si la sigmoïde renvoie 0.8, le modèle nous dit : "Je suis sûr à 80% que la réponse est 'Oui'".

Le seuil de décision Avoir une probabilité, c'est bien, mais nous voulons une décision finale : "Oui" ou "Non". C'est là qu'intervient le seuil de décision. C'est une règle simple que nous fixons. Par défaut, le seuil est de 0.5.

- Si la probabilité calculée est supérieure à 0.5, on prédit "Oui" (ou 1).
- Si la probabilité est inférieure ou égale à 0.5, on prédit "Non" (ou 0).

Ce seuil peut être ajusté en fonction du problème. Par exemple, pour un test de dépistage médical, on pourrait vouloir un seuil plus bas pour ne manquer aucun malade potentiel.

La Régression Logistique avec Scikit-Learn

Utilisons la régression logistique pour prédire la survie sur le Titanic. Nous aurons besoin de préparer un peu nos données pour que le modèle puisse les comprendre (par exemple, transformer le sexe en une valeur numérique).

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Supposons que 'df' est votre DataFrame du Titanic nettoyé
# df = pd.read_csv('train.csv') et étapes de nettoyage...
# Pour cet exemple, on s'assure que les données sont prêtes
# df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})
# df.fillna({'Age': df['Age'].median()}, inplace=True)

# 1. Sélectionner les variables (Features) et la cible (Target)
features = ['Pclass', 'Sex', 'Age', 'Fare']
target = 'Survived'

X = df[features]
y = df[target]

# 2. Diviser les données
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# 3. Créer et entraîner le modèle
model = LogisticRegression(max_iter=200)
model.fit(X_train, y_train)

# 4. Voir les probabilités prédites sur les données de test
# Renvoie deux colonnes : P(Non-survivant) et P(Survivant)
probabilities = model.predict_proba(X_test)
print("Probabilités (Non-survivant, Survivant) pour les 5 premiers
passagers :")
print(probabilities[:5])
```

```
# 5. Voir les classes finales prédites (avec un seuil de 0.5)
predictions = model.predict(X_test)
print("\nPrédictions finales pour les 5 premiers passagers :")
print(predictions[:5])
```

CHALLENGE POUR VOUS !

Le seuil de décision par défaut est de 0.5, mais il n'est pas toujours optimal.

Votre mission : Réfléchissez à cette question (pas de code nécessaire). Le seuil de décision par défaut est de 0.5. Que se passerait-il si, pour prédire la survie, vous le fixiez à 0.8 ? Feriez-vous plus ou moins d'erreurs de type "prédire non-survivant alors qu'il a survécu" ? Et pourquoi ?

Réponse : En fixant le seuil à 0.8, on devient beaucoup plus exigeant pour prédire qu'une personne a survécu. Il faudrait que le modèle soit sûr à plus de 80%. Par conséquent, on va prédire "survivant" beaucoup moins souvent. Cela signifie qu'on fera PLUS d'erreurs du type "prédire non-survivant alors qu'il a survécu". Ce type d'erreur est appelé un Faux Négatif. On augmente la prudence du modèle au risque de devenir trop pessimiste.

16 ÉVALUER SON MODÈLE : COMMENT SAVOIR SI VOTRE IA DIT N'IMPORTE QUOI ?

Une précision de 99% est-elle toujours excellente ? Imaginez un test pour une maladie rare qui touche 1 personne sur 100. Un modèle qui prédit systématiquement "non malade" aura une précision de 99%, mais il est complètement inutile pour détecter la maladie ! Cet exemple montre que l'accuracy seule peut être une métrique très trompeuse. Pour vraiment juger un modèle, nous avons besoin d'outils plus fins.

AU-DELÀ DE L'ACCURACY

La Matrice de Confusion C'est l'outil fondamental pour comprendre les erreurs de votre modèle de classification. C'est un tableau qui compare les valeurs réelles avec les prédictions du modèle.

- Vrai Positif (VP) : Le modèle a prédit "Oui", et la réalité était "Oui". (Il a eu raison).
- Vrai Négatif (VN) : Le modèle a prédit "Non", et la réalité était "Non". (Il a eu raison).
- Faux Positif (FP) : Le modèle a prédit "Oui", mais la réalité était "Non". (Aussi appelé erreur de type I).
- Faux Négatif (FN) : Le modèle a prédit "Non", mais la réalité était "Oui". (Aussi appelé erreur de type II).

Précision, Rappel et F1-Score À partir de cette matrice, on peut calculer des métriques bien plus informatives :

- La Précision (Precision) : Elle répond à la question : "Sur tout ce que j'ai prédit comme étant positif, combien l'étaient vraiment ?". Une haute précision signifie que le modèle ne commet que très peu de faux positifs. Précision = VP / (VP + FP)

- Le Rappel (Recall) : Il répond à la question : "Sur tout ce qui était réellement positif, combien en ai-je trouvé ?". Un haut rappel signifie que le modèle rate très peu de vrais positifs. Rappel = VP / (VP + FN)
- Le F1-Score : C'est la moyenne harmonique de la précision et du rappel. C'est une excellente métrique pour avoir une vision équilibrée de la performance du modèle, surtout quand les classes sont déséquilibrées.

Évaluer notre modèle du Titanic

Reprenons notre modèle de régression logistique de l'chapitre S15 et évaluons-le correctement avec Scikit-Learn.

```
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
# On suppose que model, X_test, y_test et predictions de l'chapitre S15
sont disponibles

# 1. Score de précision (Accuracy)
accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy Score: {accuracy:.2f}\n")

# 2. Matrice de Confusion
conf_matrix = confusion_matrix(y_test, predictions)
print("Matrice de Confusion:")
print(conf_matrix)
# [[VN, FP],
#  [FN, VP]]
print("\n")

# 3. Rapport de Classification
# Ce rapport nous donne la précision, le rappel et le f1-score pour chaque
classe.
class_report = classification_report(y_test, predictions,
target_names=['Non-survivant', 'Survivant'])
print("Rapport de Classification:")
print(class_report)
```

Le rapport de classification est extrêmement utile car il décompose la performance du modèle pour chaque classe ("Non-survivant" et "Survivant"), ce qui nous donne une vision beaucoup plus claire de ses forces et de ses faiblesses.

Challenge pour vous !

Le choix de la bonne métrique dépend toujours du contexte du problème.

Votre mission : Dans le cas d'un filtre anti-spam pour vos emails :

1. Est-il plus grave d'avoir un faux positif (un email important est classé comme spam et vous le manquez) ou un faux négatif (un spam arrive dans votre boîte de réception principale) ?

2. En conséquence, quelle métrique (précision ou rappel) voudriez-vous maximiser en priorité pour la classe "Spam" ?

Réponse : Il est bien plus grave d'avoir un faux positif. Manquer un email important (une offre d'emploi, un message de la famille) peut avoir des conséquences sérieuses, alors que voir un spam dans sa boîte de réception est juste un peu agaçant.

Pour minimiser les faux positifs, on veut que le modèle soit très sûr de lui quand il classifie un email comme "Spam". On veut donc maximiser la Précision. Une haute précision sur la classe "Spam" signifie que si le modèle dit "c'est un spam", il a presque toujours raison.

17 Overfitting : Le piège mortel du Machine Learning et comment l'éviter

Imaginez que vous apprenez pour un examen. Vous avez trois approches :

1. Le sous-apprentissage (Underfitting) : Vous survolez à peine le livre. Vous ne comprenez même pas les concepts de base. Vous raterez l'examen.

2. Le bon ajustement (Good fit) : Vous comprenez les concepts clés, ce qui vous permet de répondre correctement aux questions que vous connaissez, mais aussi à de nouvelles questions sur le même sujet. Vous réussirez l'examen.

3. Le surapprentissage (Overfitting) : Vous apprenez par cœur chaque question et chaque réponse des annales, sans comprendre la logique derrière. Vous excellerez sur les questions que vous avez déjà vues, mais vous serez incapable de répondre à une nouvelle question, même simple. Vous raterez probablement l'examen.

En machine learning, c'est la même chose. Le surapprentissage est le piège mortel qui attend tout data scientist.

Apprendre par cœur n'est pas apprendre

Qu'est-ce que l'overfitting ? L'overfitting (ou surajustement) se produit quand un modèle est trop complexe et qu'il "apprend par cœur" les données d'entraînement, y compris leur bruit et leurs particularités aléatoires. Au lieu d'apprendre la tendance générale (le "signal"), il mémorise les exemples. Par conséquent, le modèle obtient des scores spectaculaires sur les données qu'il a déjà vues (l'ensemble d'entraînement), mais il est incapable de généraliser à de nouvelles données (l'ensemble de test). Sa performance dans le monde réel est médiocre.

Comment l'éviter ? La régularisation La régularisation est une technique qui consiste à "pénaliser" la complexité d'un modèle pendant l'entraînement. On modifie la fonction de coût pour y ajouter une pénalité qui augmente avec la magnitude des coefficients du modèle. Cela force le modèle à trouver un équilibre : il doit bien ajuster les données, mais sans laisser ses coefficients devenir trop grands, ce qui le rendrait trop complexe. Les deux types de régularisation les plus courants sont :

- L1 (Lasso) : Elle a tendance à réduire certains coefficients à exactement zéro, effectuant ainsi une sorte de sélection automatique des variables les plus importantes.

- L2 (Ridge) : Elle réduit tous les coefficients de manière plus douce, sans forcément les annuler. C'est la plus utilisée.

L'OVERFITTING EN ACTION

Voyons comment un arbre de décision (DecisionTreeClassifier) peut sur-apprendre de manière flagrante.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# On suppose que 'df' est votre DataFrame du Titanic préparé
# (variables catégorielles transformées en nombres, etc.)
# df = ...

# 1. Sélectionner les variables et la cible
features = ['Pclass', 'Sex', 'Age', 'Fare']
target = 'Survived'
# Assumons que les données sont prêtes
# df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})
# df.fillna({'Age': df['Age'].median()}, inplace=True)
X = df[features]
y = df[target]

# 2. Diviser les données
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# 3. Modèle qui sur-apprend (aucune limite de profondeur)
overfit_tree = DecisionTreeClassifier(random_state=42)
overfit_tree.fit(X_train, y_train)

print("--- Modèle en surapprentissage (profondeur infinie) ---")
print(f"Score sur les données d'entraînement : {overfit_tree.score(X_train,
y_train):.2f}")
print(f"Score sur les données de test : {overfit_tree.score(X_test,
y_test):.2f}")

# 4. Modèle régularisé (profondeur limitée)
good_tree = DecisionTreeClassifier(max_depth=4, random_state=42)
good_tree.fit(X_train, y_train)

print("\n--- Modèle régularisé (profondeur max = 4) ---")
print(f"Score sur les données d'entraînement : {good_tree.score(X_train,
y_train):.2f}")
print(f"Score sur les données de test : {good_tree.score(X_test,
y_test):.2f}")
```

Analyse : Le premier modèle obtient un score quasi parfait sur l'entraînement (il a appris par cœur) mais un score bien plus faible sur le test. Le second modèle a un score d'entraînement légèrement inférieur, mais son score de test est bien meilleur. Il a mieux généralisé.

CHALLENGE POUR VOUS !

Explorons l'effet de la régularisation L2 (Ridge) sur une régression linéaire.

Votre mission :

1. Utilisez le modèle Ridge de Scikit-Learn sur un jeu de données de régression (par exemple, celui de l'chapitre S14).
2. Entraînez plusieurs modèles Ridge en faisant varier le paramètre alpha (la force de la pénalité) sur une large échelle (ex: [0.01, 0.1, 1, 10, 100]).
3. Pour chaque alpha, affichez les coefficients appris par le modèle (model.coef_).

Comment les coefficients du modèle évoluent-ils lorsque alpha augmente ? Qu'est-ce que cela signifie ?

18 Les Arbres de Décision : Le Machine Learning que vous pouvez expliquer à votre grand-mère

Vous souvenez-vous du jeu "Qui est-ce ?" ? Vous posez une série de questions "oui/non" pour deviner le personnage de votre adversaire : "Votre personnage a-t-il des lunettes ?", "Est-ce un homme ?", "Porte-t-il un chapeau ?". Chaque question réduit le champ des possibilités jusqu'à ce qu'il n'en reste plus qu'une.

Un arbre de décision fait exactement la même chose avec les données. C'est un modèle de machine learning si simple et visuel que vous pourriez littéralement l'expliquer à votre grand-mère.

L'art de poser les bonnes questions

Comment l'arbre choisit-il la meilleure question ? Au lieu de poser des questions au hasard, l'arbre cherche à chaque étape la question la plus "efficace", celle qui sépare le mieux les données. Mais qu'est-ce qu'une "bonne" séparation ?

Imaginez que vous avez un panier de pommes et de bananes mélangées (un groupe "impur"). Une bonne question serait "Le fruit est-il jaune ?". La réponse "oui" vous donnerait un groupe contenant principalement des bananes, et la réponse "non" un groupe avec principalement des pommes. Vous avez créé des groupes plus "purs" qu'au départ.

L'arbre utilise des métriques mathématiques comme l'Impureté de Gini ou le Gain d'Information pour mesurer cette "pureté". Il va tester toutes les questions possibles (sur toutes les variables) et choisir celle qui maximise la pureté des groupes résultants. Il répète ce processus à chaque nouvelle branche, créant ainsi l'arbre de haut en bas.

Visualiser les décisions

Utilisons le célèbre jeu de données "Iris" pour entraîner un arbre de décision. Ce dataset contient des mesures (longueur/largeur des pétales et sépales) pour trois espèces de fleurs d'iris. Le but est de prédire l'espèce à partir des mesures.

```
import pandas as pd
import matplotlib.pyplot as plt
```

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split

# 1. Charger le jeu de données Iris
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names

# 2. Diviser les données
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# 3. Entraîner le modèle
tree_clf = DecisionTreeClassifier(max_depth=3, random_state=42)
tree_clf.fit(X_train, y_train)

# 4. Visualiser l'arbre de décision
plt.figure(figsize=(20,10))
plot_tree(tree_clf,
          feature_names=feature_names,
          class_names=target_names,
          filled=True,
          rounded=True)
plt.title("Arbre de décision entraîné sur le dataset Iris")
plt.show()

# Évaluer le modèle
print(f"Score du modèle sur les données de test : {tree_clf.score(X_test,
y_test):.2f}")
```

Comment lire le graphique ? Chaque boîte est un "nœud".

- La première ligne est la question posée (ex: "petal width (cm) <= 0.8").
- gini mesure l'impureté du nœud. Un gini de 0.0 signifie un nœud parfaitement pur (il ne contient qu'une seule classe).
- samples est le nombre d'échantillons de données dans ce nœud.
- value montre la répartition des échantillons entre les différentes classes.
- class est la prédiction majoritaire pour ce nœud.

En suivant les flèches "True" ou "False", vous pouvez voir exactement comment le modèle prend sa décision pour n'importe quelle fleur.

CHALLENGE POUR VOUS !

Le dataset Iris a 4 "features" (variables). Que se passe-t-il si on en utilise moins ?

Votre mission :

1. Entraînez un nouvel arbre de décision en n'utilisant que deux des quatre features (par exemple, la longueur et la largeur des pétales). Visualisez l'arbre.

2. Maintenant, comparez-le à l'arbre que nous avons entraîné avec les quatre features.

3. L'arbre est-il plus ou moins complexe ? Son score de performance sur l'ensemble de test est-il meilleur ou moins bon ?

19 Les Forêts Aléatoires (Random Forests) : La puissance du collectif

Pour prendre une décision financière importante, préféreriez-vous demander l'avis d'un seul expert, aussi brillant soit-il, ou celui d'un comité de 100 experts aux opinions et aux expériences variées ? La plupart des gens choisiraient le comité, car la sagesse du groupe est souvent supérieure à celle de l'individu.

Une forêt aléatoire (*Random Forest*), c'est exactement ça : un comité d'arbres de décision. Au lieu de se fier à un seul arbre, on en entraîne des centaines, chacun avec une "perspective" légèrement différente, et on les fait voter pour prendre la décision finale. C'est l'un des modèles les plus robustes et les plus efficaces qui existent.

Le secret de la diversité

La puissance d'une forêt aléatoire vient du fait que les arbres qui la composent sont tous différents. Si tous les experts de notre comité avaient fait les mêmes études et lu les mêmes livres, leur avis collectif ne serait pas très utile. Pour garantir cette diversité, l'algorithme utilise deux sources d'aléatoire.

1. Le Bagging (Bootstrap Aggregating) : Chaque arbre de la forêt n'est pas entraîné sur la totalité des données. À la place, on lui donne un échantillon aléatoire des données, tiré "avec remise". Cela signifie que certains points de données peuvent apparaître plusieurs fois dans l'échantillon d'un arbre, et d'autres pas du tout. Chaque arbre voit donc une version légèrement différente du monde.

2. La Sélection Aléatoire de Features : C'est la deuxième source de génie. Quand un arbre doit choisir la meilleure question à poser à un nœud, on ne l'autorise pas à examiner toutes les variables disponibles. On lui en donne seulement un sous-ensemble aléatoire. Cela empêche les arbres de devenir trop dépendants d'une seule variable très prédictive et les force à explorer d'autres relations dans les données.

Ces deux mécanismes créent une forêt d'experts diversifiés, où les erreurs individuelles de chaque arbre sont compensées par la sagesse du groupe.

La Forêt en action

Utilisons le RandomForestClassifier de Scikit-Learn sur nos données du Titanic et comparons sa performance à celle de notre arbre de décision unique.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.tree import DecisionTreeClassifier

# On suppose que 'df' est votre DataFrame du Titanic préparé
# df = ...
# df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})
# df.fillna({'Age': df['Age'].median()}, inplace=True)

# 1. Sélectionner les variables et la cible
features = ['Pclass', 'Sex', 'Age', 'Fare', 'SibSp', 'Parch']
target = 'Survived'
X = df[features]
y = df[target]

# 2. Diviser les données
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# 3. Entraîner et évaluer un Arbre de Décision unique (pour comparaison)
single_tree = DecisionTreeClassifier(max_depth=5, random_state=42)
single_tree.fit(X_train, y_train)
print(f"Score de l'Arbre de Décision unique : {single_tree.score(X_test,
y_test):.4f}")

# 4. Entraîner et évaluer une Forêt Aléatoire
forest = RandomForestClassifier(n_estimators=100, random_state=42)
forest.fit(X_train, y_train)
print(f"Score de la Forêt Aléatoire : {forest.score(X_test, y_test):.4f}")

# 5. Afficher l'importance des variables
importances = forest.feature_importances_
feature_importances = pd.Series(importances,
index=features).sort_values(ascending=False)

plt.figure(figsize=(10, 6))
feature_importances.plot(kind='bar')
plt.title("Importance des variables selon la Forêt Aléatoire")
plt.ylabel("Importance")
plt.show()
```

On constate quasi-systématiquement que la forêt aléatoire est plus performante que l'arbre unique. De plus, feature_importances_ nous montre que le sexe, le tarif et l'âge étaient les facteurs les plus décisifs pour le modèle.

CHALLENGE POUR VOUS !

Le paramètre n_estimators contrôle le nombre d'arbres dans la forêt. Intuitivement, plus on a d'experts, mieux c'est. Mais est-ce toujours vrai ?

Votre mission :

1. Entraînez plusieurs modèles de RandomForestClassifier en faisant varier n_estimators (par exemple : 10, 20, 50, 100, 200, 500).

2. Pour chaque modèle, enregistrez son score sur l'ensemble de test.

3. Tracez un graphique montrant l'évolution du score en fonction du nombre d'arbres.

Le score s'améliore-t-il indéfiniment ? Y a-t-il un moment où le gain de performance devient négligeable (ou même nul) ? Partagez votre graphique et vos conclusions en commentaire !

20 Le K-Means : L'art de trouver des groupes cachés dans vos données

Regardez un nuage de points. Souvent, vos yeux voient instinctivement des groupes, des "clusters" de points qui sont proches les uns des autres. Mais comment apprendre à une machine à trouver ces mêmes groupes, sans aucune étiquette pour la guider ? C'est l'objectif du clustering, la branche la plus connue de l'apprentissage non supervisé. Et son algorithme roi est le K-Means.

L'algorithme du K-Means

Le K-Means est un algorithme étonnamment simple et intuitif. Voici comment il fonctionne, en quatre étapes :

1. Initialisation : On choisit d'abord le nombre de clusters que l'on veut trouver, un nombre K. Puis, on place K "centroïdes" (les centres des futurs clusters) à des endroits aléatoires dans les données.

2. Assignation : Chaque point de donnée est assigné au centroïde le plus proche de lui. On forme ainsi K groupes provisoires.

3. Mise à jour : On recalcule la position de chaque centroïde. Le nouveau centroïde devient le centre de gravité (la moyenne) de tous les points qui lui ont été assignés à l'étape précédente.

4. Répétition : On répète les étapes 2 et 3 jusqu'à ce que les centroïdes ne bougent plus. À ce moment-là, l'algorithme a "convergé" et nos clusters sont stables.

L'importance du choix de K : La méthode du coude La plus grande difficulté du K-Means est de choisir la bonne valeur pour K au départ. Si on choisit un K trop petit, on va regrouper des points qui ne vont pas ensemble. Si on choisit un K trop grand, on va diviser des groupes qui devraient être unis.

Pour nous aider, on utilise souvent la méthode du coude (*elbow method*). On fait tourner l'algorithme pour plusieurs valeurs de K et on mesure pour chacune "l'inertie" (la somme des distances au carré entre chaque point et le centre de son cluster). On trace ensuite un graphique de l'inertie en fonction de K. L'endroit où la courbe "casse", formant un "coude", est généralement le meilleur K. C'est le point où ajouter un nouveau cluster n'apporte plus beaucoup de gain.

Trouver des clusters avec Scikit-Learn

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# 1. Générer des données d'exemple avec 4 clusters
```

```
X, y_true = make_blobs(n_samples=300, centers=4, cluster_std=0.80,
random_state=0)
plt.scatter(X[:, 0], X[:, 1], s=50)
plt.title("Données brutes")
plt.show()

# 2. Entraîner le modèle K-Means
kmeans = KMeans(n_clusters=4, random_state=0, n_init=10)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)

# 3. Visualiser les clusters trouvés
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75,
label='Centroïdes')
plt.title("Clusters trouvés par K-Means")
plt.legend()
plt.show()

# 4. Implémenter la méthode du coude pour trouver le K optimal
inertia = []
k_range = range(1, 11)
for k in k_range:
    kmeans_elbow = KMeans(n_clusters=k, random_state=0, n_init=10).fit(X)
    inertia.append(kmeans_elbow.inertia_)

plt.plot(k_range, inertia, marker='o')
plt.title('Méthode du coude')
plt.xlabel('Nombre de clusters (K)')
plt.ylabel('Inertie')
plt.show()
```

Le graphique du coude montre clairement une "cassure" à K=4, confirmant que c'est le bon nombre de clusters pour nos données.

CHALLENGE POUR VOUS !

Le K-Means calcule des distances. Il est donc très sensible à l'échelle des variables. Si une variable a une échelle beaucoup plus grande qu'une autre, elle dominera le calcul de distance.

Votre mission :

1. Créez un jeu de données non normalisées (par exemple, avec make_blobs et en multipliant une des colonnes par 100). Appliquez K-Means et visualisez le résultat.
2. Maintenant, utilisez le StandardScaler de Scikit-Learn pour normaliser vos données (les mettre à la même échelle).
3. Ré-appliquez K-Means sur les données normalisées et visualisez à nouveau.

Les clusters trouvés sur les données normalisées sont-ils plus cohérents et plus logiques ?

21 Scikit-Learn : La boîte à outils ultime du Data Scientist

Tout au long de ce parlivre, nous avons utilisé une multitude d'outils pour nettoyer les données, les visualiser et construire des modèles. La bibliothèque qui orchestre tout cet univers, c'est Scikit-Learn. Si la data science était de la plomberie, Scikit-Learn serait votre clé à molette : l'outil polyvalent, fiable et indispensable que vous utilisez pour presque toutes les tâches. Faisons un tour du propriétaire de cette bibliothèque exceptionnellement cohérente et puissante.

Une API unifiée et des modules clairs

La beauté de Scikit-Learn réside dans son API unifiée. Que vous utilisiez un pré-processeur, un modèle de régression ou un algorithme de clustering, les commandes de base sont toujours les mêmes. C'est un "langage" que vous apprenez une seule fois.

- estimateur.fit(X, y) : C'est la commande pour entraîner l'estimateur. L'objet "apprend" à partir de vos données X (et y pour l'apprentissage supervisé).
- estimateur.predict(X) : Une fois entraîné, cette commande est utilisée pour faire des prédictions sur de nouvelles données.
- estimateur.transform(X) : Pour les outils de pré-traitement, cette commande applique la transformation apprise (par exemple, mettre à l'échelle les données).

Scikit-Learn est organisé en modules logiques :

- sklearn.preprocessing : Contient les outils de nettoyage et de mise à l'échelle (StandardScaler, SimpleImputer).
- sklearn.model_selection : Pour diviser les données et faire de la validation croisée (train_test_split, cross_val_score).
- sklearn.linear_model : Les modèles linéaires (LinearRegression, LogisticRegression, Ridge).
- sklearn.ensemble : Les modèles basés sur des ensembles (RandomForestClassifier).
- sklearn.metrics : Les outils d'évaluation (accuracy_score, confusion_matrix).

Les Pipelines : L'automatisation du workflow Un Pipeline est un objet Scikit-Learn qui permet d'enchaîner plusieurs étapes de traitement et de modélisation en un seul et unique estimateur. C'est un outil extraordinairement puissant.

Construire un Pipeline complet

Imaginez notre workflow habituel : 1. Mettre les données à l'échelle. 2. Entraîner une forêt aléatoire. Avec un pipeline, cela devient une seule étape.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
```

```
# On suppose que 'df' est votre DataFrame du Titanic préparé
# df = ...
# df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})
# df.fillna({'Age': df['Age'].median()}, inplace=True)

# 1. Sélectionner les variables et la cible
features = ['Pclass', 'Sex', 'Age', 'Fare']
target = 'Survived'
X = df[features]
y = df[target]

# 2. Diviser les données
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# 3. Créer le Pipeline
# Il enchaîne deux étapes : 'scaler' puis 'classifier'
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', RandomForestClassifier(random_state=42))
])

# 4. Entraîner le Pipeline en une seule fois !
pipe.fit(X_train, y_train)

# 5. Évaluer le Pipeline
score = pipe.score(X_test, y_test)
print(f"Score du Pipeline complet : {score:.4f}")
```

L'avantage n'est pas seulement la simplification du code. Le Pipeline garantit aussi qu'on ne fait pas de "fuite de données" (*data leakage*), car la mise à l'échelle est apprise uniquement sur les données d'entraînement (X_train) avant d'être appliquée aux données de test (X_test).

CHALLENGE POUR VOUS !

Vous avez maintenant tous les outils pour construire un workflow de machine learning robuste du début à la fin.

Votre mission :

1. Créez un pipeline qui effectue trois actions : a. Gérer les valeurs manquantes avec SimpleImputer(strategy='median'). b. Mettre les données à l'échelle avec StandardScaler(). c. Appliquer un modèle de LogisticRegression().
2. Au lieu d'utiliser train_test_split, évaluez la performance de ce pipeline en utilisant la validation croisée avec la fonction cross_val_score de Scikit-Learn. La validation croisée est une méthode encore plus robuste pour estimer la performance d'un modèle.

22 Feature Engineering : L'ingrédient secret des meilleurs modèles

Les meilleurs modèles de machine learning ne viennent que rarement des algorithmes les plus complexes, mais presque toujours des meilleures "features" (variables). Le feature engineering est l'art, souvent créatif, de transformer vos données brutes pour créer des variables plus pertinentes et plus parlantes pour votre modèle. C'est l'ingrédient secret qui permet de débloquer des performances supérieures.

L'art de créer de l'information

Le feature engineering consiste à utiliser votre connaissance du domaine et votre intuition pour extraire de la valeur de vos données existantes.

- Créer des variables d'interaction : Parfois, l'interaction entre deux variables est plus informative que les variables prises séparément. Par exemple, l'effet de l'âge sur un revenu peut être différent selon le niveau d'éducation. On pourrait donc créer une feature age * niveau_education.
- Extraire des informations d'une date : Une date brute est rarement utile. On peut en extraire le jour de la semaine, le mois, l'année, ou même si c'était un jour férié. Ces nouvelles features peuvent avoir un pouvoir prédictif énorme.
- Transformer des variables numériques : Comme nous l'avons vu, appliquer une transformation logarithmique ou carrée peut aider à normaliser une distribution ou à linéariser une relation, ce qui aide de nombreux modèles.
- Encoder les variables catégorielles : Les modèles ne comprennent pas le texte ("Rouge", "Vert", "Bleu"). Le One-Hot Encoding est une technique qui transforme une colonne catégorielle en plusieurs colonnes binaires (0/1), une pour chaque catégorie, que le modèle peut interpréter.

Enrichir le dataset du Titanic

Appliquons ces idées à notre jeu de données du Titanic.

```
import pandas as pd
import numpy as np

# On suppose que 'df' est votre DataFrame du Titanic nettoyé de la partie 1
# df = pd.read_csv('train.csv') et étapes de nettoyage...

# 1. Créer 'FamilySize'
df['FamilySize'] = df['SibSp'] + df['Parch'] + 1

# 2. Encoder 'Embarked' avec One-Hot Encoding
# pd.get_dummies crée de nouvelles colonnes pour chaque catégorie
df_embarked = pd.get_dummies(df['Embarked'], prefix='Embarked')
df = pd.concat([df, df_embarked], axis=1)
# On peut maintenant supprimer la colonne originale
df.drop('Embarked', axis=1, inplace=True)

# 3. Regrouper les âges en catégories
```

```
bins = [0, 12, 50, 100] # Définir les limites des catégories
labels = ['Enfant', 'Adulte', 'Senior'] # Donner un nom à chaque catégorie
df['AgeGroup'] = pd.cut(df['Age'], bins=bins, labels=labels, right=False)

print(df[['Name', 'FamilySize', 'Age', 'AgeGroup']].head())
```

CHALLENGE POUR VOUS !

Nous avons déjà effleuré ce challenge dans l'chapitre sur le nettoyage, mais allons maintenant jusqu'au bout pour en mesurer l'impact.

Votre mission :

1. Créez la feature 'Title' en extrayant le titre de chaque passager depuis la colonne 'Name'.
2. Simplifiez ces titres en regroupant les plus rares ('Dr', 'Rev', 'Col', etc.) sous une catégorie 'Rare'.
3. Encodez cette nouvelle colonne 'Title' pour qu'elle soit utilisable par un modèle.
4. Entraînez un RandomForestClassifier sans cette feature 'Title'. Notez son score.
5. Entraînez un second RandomForestClassifier avec cette feature 'Title'.
6. La performance s'est-elle améliorée ?

C'est la démonstration parfaite que passer du temps à créer des features intelligentes est l'un des meilleurs investissements que vous puissiez faire dans un projet de data science.

23 PROJET MACHINE LEARNING (PARTIE 1) : PRÉDIRE LE PRIX D'UNE MAISON

Nouveau projet de A à Z ! Après avoir classifié les survivants du Titanic, nous nous attaquons à un autre problème classique et fondamental de la data science : la régression. Notre objectif sera de prédire le prix de vente de maisons en se basant sur leurs caractéristiques. Dans cette première partie, nous allons suivre les étapes initiales et cruciales de tout projet : la collecte, le nettoyage et, surtout, l'exploration des données.

REVOIR LES FONDAMENTAUX

Le workflow de la data science Rappelons-nous les étapes de notre feuille de route :

1. Question : Peut-on prédire le prix de vente d'une maison à partir de ses caractéristiques (surface, nombre de chambres, etc.) ?
2. Collecte : Rassembler les données. Nous utiliserons le célèbre jeu de données "Ames Housing".
3. Nettoyage & Exploration (EDA) : Notre focus aujourd'hui.
4. Feature Engineering & Modélisation : Pour les prochains chapitres.
5. Évaluation & Communication.

L'importance de l'Analyse Exploratoire des Données (EDA) Avant même de penser à un modèle, il est impératif de comprendre vos données. L'EDA est cette phase d'enquête qui consiste à "sentir" les données, à comprendre leurs distributions, à identifier les relations entre les variables et à repérer les problèmes potentiels (valeurs manquantes, aberrantes). Une bonne exploration est la garantie d'un bon modèle.

Explorer le marché immobilier d'Ames

Nous allons utiliser le jeu de données "Ames Housing", un dataset riche contenant 79 variables décrivant (presque) tous les aspects des maisons vendues à Ames, dans l'Iowa.

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

# Charger les données (souvent disponibles sur Kaggle ou d'autres plateformes)
# Pour cet exemple, nous supposons que vous avez les fichiers train.csv et test.csv
df = pd.read_csv('train.csv')

# 1. Première inspection
print("Dimensions du dataset :", df.shape)
print("\nTypes de données :")
print(df.info())

# 2. Résumé statistique
print("\nRésumé statistique des variables numériques :")
print(df.describe())

# 3. Identifier les valeurs manquantes
missing_values = df.isnull().sum().sort_values(ascending=False)
print("\nTop 20 des variables avec le plus de valeurs manquantes :")
print(missing_values.head(20))

# 4. Visualiser la distribution de la variable cible (SalePrice)
plt.figure(figsize=(10, 6))
sns.histplot(df['SalePrice'], kde=True)
plt.title('Distribution du Prix de Vente (SalePrice)')
plt.show()
```

On remarque que la distribution du prix de vente est asymétrique. Une transformation logarithmique pourrait être utile plus tard.

Challenge pour vous !

La première étape de l'exploration est de comprendre quelles variables sont les plus liées à notre cible, SalePrice.

Votre mission :

1. Calculez la matrice de corrélation de toutes les variables numériques du DataFrame.
2. Identifiez les 10 variables qui sont les plus corrélées (positivement ou négativement) avec SalePrice.
3. Pour les 3 variables les plus importantes parmi celles-ci, créez un nuage de points (scatter plot) pour visualiser leur relation avec SalePrice.

Cela vous donnera une première intuition très forte sur les facteurs qui influencent le plus le prix d'une maison.

24 Projet Machine Learning (Partie 2) : Entraîner et évaluer 3 modèles différents

Nos données sont prêtes et explorées. Il est temps de passer à la phase la plus excitante : la modélisation ! Nous n'allons pas nous contenter d'un seul modèle. Pour trouver le meilleur, nous allons en entraîner trois, du plus simple au plus complexe : une Régression Linéaire qui nous servira de baseline, une Forêt Aléatoire, et un Gradient Boosting. Que le meilleur gagne !

L'art de choisir son champion

L'importance d'avoir un modèle de baseline Avant de sortir les modèles les plus sophistiqués, il est crucial d'établir une baseline. C'est un modèle simple et rapide à entraîner (comme la Régression Linéaire) qui nous donne un premier score de performance. Tout modèle plus complexe que nous essaierons par la suite doit obtenir un meilleur score que cette baseline pour justifier sa complexité. Si ce n'est pas le cas, c'est que quelque chose ne va pas.

Revoir les métriques de régression Pour comparer nos modèles, nous avons besoin de juges impartiaux : les métriques d'évaluation. Pour la régression, les deux plus courantes sont :

- RMSE (Root Mean Squared Error) : C'est la racine carrée de l'erreur quadratique moyenne. On peut l'interpréter comme l'erreur de prédiction "moyenne" de notre modèle, dans la même unité que notre cible (ici, en dollars). Plus le RMSE est bas, mieux c'est.
- R^2 (Coefficient de détermination) : Il représente la proportion de la variance de la variable cible qui est expliquée par notre modèle. Un R^2 de 0.75 signifie que notre modèle explique 75% de la variabilité des prix de vente. Plus on est proche de 1, mieux c'est.

Le processus de sélection de modèle Le processus est simple : on entraîne chaque modèle sur le même ensemble de données d'entraînement, puis on évalue leur performance sur un ensemble de test qu'ils n'ont jamais vu. Le modèle qui obtient les meilleures métriques sur l'ensemble de test est déclaré vainqueur, car c'est celui qui généralise le mieux à de nouvelles données.

La compétition des modèles

Nous allons utiliser des Pipelines pour nous assurer que le traitement des données est appliqué correctement et de manière cohérente pour chaque modèle.

```
import pandas as pd
import numpy as np
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor,
GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Charger les données
df = pd.read_csv('train.csv')

# 1. Préparer les données
# Séparer les features et la cible
X = df.drop('SalePrice', axis=1)
y = df['SalePrice']

# Identifier les colonnes numériques et catégorielles
numerical_features = X.select_dtypes(include=np.number).columns.tolist()
categorical_features = X.select_dtypes(exclude=np.number).columns.tolist()

# Créer les pipelines de prétraitement
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Combiner les transformers avec ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numerical_features),
        ('cat', categorical_transformer, categorical_features)
    ])

# 2. Diviser en train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 3. Définir et entraîner les modèles
models = {
    "Régression Linéaire": LinearRegression(),
    "Forêt Aléatoire": RandomForestRegressor(random_state=42),
    "Gradient Boosting": GradientBoostingRegressor(random_state=42)
}

for name, model in models.items():
    # Créer le pipeline final avec le préprocesseur et le modèle
```

```
    pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                               ('regressor', model)])

    # Entraîner le modèle
    pipeline.fit(X_train, y_train)

    # Faire des prédictions
    y_pred = pipeline.predict(X_test)

    # Évaluer le modèle
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)

    print(f"--- {name} ---")
    print(f"RMSE: {rmse:.2f}")
    print(f"R²: {r2:.4f}\n")
```

Challenge pour vous !

Le Gradient Boosting a souvent les meilleures performances "brutes", mais il est très sensible à ses hyperparamètres. Un modèle bien optimisé peut faire une énorme différence.

Votre mission :

1. Le GradientBoostingRegressor est souvent le plus performant, mais aussi le plus sensible à ses hyperparamètres.
2. Utilisez RandomizedSearchCV de Scikit-Learn pour trouver une meilleure combinaison d'hyperparamètres (n_estimators, learning_rate, max_depth) pour le GradientBoostingRegressor.
3. Le modèle optimisé bat-il le score du Random Forest par défaut ?

C'est une étape clé pour passer d'un bon data scientist à un excellent data scientist : ne pas se contenter des modèles par défaut, mais chercher à en extraire la performance maximale.

25 Les K-Plus Proches Voisins (KNN) : Simple, intuitif et étonnamment efficace

"Dis-moi qui sont tes amis, je te dirai qui tu es." Ce vieux proverbe résume parfaitement la philosophie de l'un des algorithmes les plus intuitifs du machine learning : les K-Plus Proches Voisins (K-Nearest Neighbors, ou KNN). Pour classifier un nouveau point de donnée, l'algorithme ne fait rien de plus que de regarder ses voisins les plus proches et de suivre l'avis de la majorité. C'est une approche simple, sans modèle complexe, mais étonnamment efficace.

La sagesse de la proximité

Comment fonctionne l'algorithme ? Le KNN est un algorithme basé sur la distance. Pour classifier un nouveau point, il suit ces étapes :

1. Choisir K : On décide d'abord du nombre de voisins (K) à consulter.
2. Calculer les distances : On calcule la distance (généralement la distance euclidienne) entre le nouveau point et tous les autres points du jeu de données d'entraînement.
3. Trouver les K plus proches voisins : On identifie les K points qui ont les plus petites distances par rapport au nouveau point.
4. Vote majoritaire : On regarde les étiquettes de ces K voisins. Le nouveau point se voit attribuer l'étiquette la plus fréquente parmi ses voisins. Si K=5 et que 3 voisins sont des "Chats" et 2 sont des "Chiens", le nouveau point sera classifié comme "Chat".

L'importance cruciale de la mise à l'échelle (Scaling) Puisque le KNN est entièrement basé sur la distance, il est extrêmement sensible à l'échelle des variables. Si vous avez une variable "âge" (de 20 à 70) et une variable "salaire" (de 30 000 à 150 000), le salaire va complètement dominer le calcul de la distance. L'âge n'aura quasiment aucun impact. Il est donc impératif de mettre toutes les features à la même échelle (par exemple avec StandardScaler) avant d'utiliser un KNN.

Comment choisir la bonne valeur de K ? Le choix de K est un compromis :

- Un petit K (ex: K=1) rend le modèle très sensible au bruit. La prédiction dépendra d'un seul voisin, ce qui peut mener à l'overfitting.
- Un grand K rend le modèle plus robuste au bruit, mais peut "lisser" les frontières de décision au point de rater des subtilités. Un K trop grand peut mener à l'underfitting. En général, on choisit une valeur impaire pour éviter les égalités lors du vote.

LE KNN EN ACTION SUR LE DATASET IRIS

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# 1. Charger et préparer les données
iris = load_iris()
X = iris.data
y = iris.target

# Diviser les données
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Mettre les données à l'échelle
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 2. Montrer comment la performance change avec K
```

```
k_range = range(1, 26)
scores = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_scaled, y_train)
    y_pred = knn.predict(X_test_scaled)
    scores.append(accuracy_score(y_test, y_pred))

plt.figure(figsize=(10, 6))
plt.plot(k_range, scores, marker='o')
plt.title('Performance du KNN en fonction de K')
plt.xlabel('Valeur de K')
plt.ylabel('Accuracy')
plt.xticks(k_range)
plt.grid(True)
plt.show()
# Le meilleur score est souvent obtenu pour K entre 3 et 9 dans ce cas.
print(f"Meilleur score obtenu : {max(scores):.2f}")
```

CHALLENGE POUR VOUS !

Le KNN a un fonctionnement très différent des modèles que nous avons vus jusqu'à présent, comme les forêts aléatoires.

Votre mission :

1. En utilisant le code ci-dessus, comparez le temps d'entraînement (.fit()) et le temps de prédiction (.predict()) d'un KNeighborsClassifier et d'un RandomForestClassifier.
2. Que remarquez-vous ?
3. Pourquoi dit-on que le KNN est un "lazy learner" (apprenant paresseux) ?

Réponse : Vous remarquerez que le temps de .fit() pour le KNN est quasi instantané, tandis que celui du Random Forest prend un peu plus de temps. Inversement, le temps de .predict() est beaucoup plus long pour le KNN.

C'est parce que le KNN est un "apprenant paresseux". La phase de "training" (fit) est une arnaque : il ne fait rien, à part mémoriser la position de tous les points de données. Tout le travail (calculer toutes les distances, trouver les voisins, organiser le vote) est effectué au moment de la prédiction. C'est l'opposé d'un modèle "avide" (*eager learner*) comme une forêt aléatoire, qui fait tout le travail de construction des arbres pendant l'entraînement pour que la prédiction soit ensuite très rapide.

26 LES MACHINES À VECTEURS DE SUPPORT (SVM) : TROUVER LA FRONTIÈRE PARFAITE

Pour séparer deux groupes de points, il existe une infinité de lignes possibles. Mais sont-elles toutes aussi bonnes ? Intuitivement, non. Les Machines à Vecteurs de Support (SVM) ne cherchent pas n'importe quelle ligne de séparation ; elles cherchent LA meilleure : celle qui est la plus éloignée

possible des points de chaque groupe. La SVM cherche la ligne qui maximise la "marge", la "rue" la plus large possible entre les deux groupes.

La géométrie de la séparation

L'idée de marge maximale L'objectif principal d'une SVM est de trouver un hyperplan (une ligne en 2D, un plan en 3D, etc.) qui sépare les données en laissant un maximum d'espace de chaque côté. Cette "rue" est appelée la marge. Un modèle avec une grande marge est considéré comme plus robuste, car il est moins susceptible de mal classifier de nouveaux points qui seraient proches de la frontière.

Les vecteurs de support

Les points de données qui se trouvent sur les bords de cette marge, ceux qui "soutiennent" l'hyperplan, sont appelés les vecteurs de support. Ce sont les points les plus critiques du jeu de données. En fait, tous les autres points pourraient être supprimés sans que cela ne change la position de la frontière de décision ! La SVM ne se concentre que sur les points les plus difficiles à classer.

L'astuce du noyau (Kernel Trick)

C'est bien pour les données que l'on peut séparer avec une ligne droite, mais que faire si les données sont entremêlées ? C'est là qu'intervient l'astuce du noyau. C'est une technique mathématique brillante qui projette les données dans une dimension supérieure où elles deviennent, comme par magie, linéairement séparables. Imaginez des points rouges au centre d'un cercle de points bleus sur une feuille de papier (2D). Vous ne pouvez pas tracer de ligne pour les séparer. Mais si vous "poussez" la feuille par le dessous au centre, vous créez une troisième dimension (un cône) où vous pouvez maintenant facilement placer un plan horizontal pour séparer les points rouges (en haut) des points bleus (en bas). C'est ce que fait un noyau, mais de manière mathématique, sans jamais avoir à calculer explicitement les coordonnées dans cette dimension supérieure.

Les SVM avec Scikit-Learn

Utilisons le SVC (Support Vector Classifier) de Scikit-Learn.

1. Noyau linéaire pour données séparables

```
from sklearn.datasets import make_blobs
from sklearn.svm import SVC
import matplotlib.pyplot as plt
import numpy as np

# Données linéairement séparables
X, y = make_blobs(n_samples=50, centers=2, random_state=0,
cluster_std=0.60)
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)

# Visualisation
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
# (Code de visualisation de la frontière complexe, non montré ici pour la
clarté)
```

```
plt.title("SVM avec un noyau linéaire")
plt.show()
```

2. Noyau RBF pour données complexes Le noyau RBF (Radial Basis Function) est le plus populaire pour capturer des relations complexes.

```
from sklearn.datasets import make_moons

# Données non-linéaires
X, y = make_moons(n_samples=100, noise=0.1, random_state=0)
model = SVC(kernel='rbf', C=1.0, gamma='auto')
model.fit(X, y)

# Visualisation
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plt.title("SVM avec un noyau RBF sur des données complexes")
plt.show()
```

CHALLENGE POUR VOUS !

Les SVM sont extrêmement puissants, mais leur performance dépend fortement de leurs hyperparamètres, notamment :

- C : Le paramètre de régularisation. Un petit C crée une marge plus grande mais tolère plus d'erreurs. Un grand C essaie de classer chaque point correctement au risque d'avoir une marge plus petite.
- gamma : Pour le noyau RBF, il définit l'influence d'un seul exemple d'entraînement.

Votre mission :

1. Utilisez GridSearchCV de Scikit-Learn sur le dataset Iris.
2. Cherchez la meilleure combinaison de C (ex: [0.1, 1, 10, 100]) et gamma (ex: [1, 0.1, 0.01, 0.001]) pour un SVC avec un noyau rbf.
3. Le modèle optimisé est-il plus performant que les autres modèles que nous avons testés sur ce même dataset ?

PARTIE 3 LA DATA SCIENCE EN PRATIQUE

---

27 LE "DATA STORYTELLING" : COMMENT TRANSFORMER VOS ANALYSES EN HISTOIRES CAPTIVANTES

Une analyse brillante, un modèle prédictif incroyable... tout cela ne sert à rien si personne ne vous comprend. Le "data storytelling" est la compétence finale et cruciale du data scientist : c'est l'art de

transformer vos chiffres, vos graphiques et vos résultats en une histoire claire, mémorable et convaincante qui pousse votre audience à l'action.

LES RÈGLES D'UNE BONNE HISTOIRE

La structure en 3 actes Toute bonne histoire, d'un conte de fées à un blockbuster hollywoodien, suit une structure simple :

1. Acte 1 : La Situation (Le Contexte) : On présente la situation de départ. Quel est le contexte business ? Quelle est la norme ? (Ex: "Chaque mois, nous dépensons 100 000€ en publicités en ligne.")

2. Acte 2 : La Complication (Le Problème) : Un événement survient, un problème est découvert. C'est le cœur de votre analyse. (Ex: "Mon analyse montre que 40% de ce budget est dépensé sur des canaux qui n'apportent aucune conversion.")

3. Acte 3 : La Résolution (La Solution) : Que faire ? C'est votre recommandation, basée sur vos découvertes. (Ex: "Je recommande de réallouer ces 40 000€ sur nos canaux les plus performants pour augmenter notre retour sur investissement.")

Choisir le bon graphique pour le bon message Chaque type de graphique a une fonction. Ne choisissez pas un graphique parce qu'il est "joli", mais parce qu'il transmet votre message le plus efficacement possible.

- Pour montrer une évolution dans le temps -> Graphique linéaire.
- Pour comparer des catégories -> Diagramme à barres.
- Pour montrer une relation entre deux variables -> Nuage de points.
- Pour montrer une composition (part d'un tout) -> Diagramme à barres empilées (souvent meilleur qu'un diagramme circulaire).

Le principe du "rapport signal/bruit" C'est le concept le plus important. Le "signal", c'est l'information que vous voulez transmettre. Le "bruit", c'est tout le reste (grilles inutiles, couleurs criardes, étiquettes superflues, bordures...). Votre but est de maximiser le signal et de réduire le bruit au minimum. Éliminez tout ce qui ne contribue pas directement à la compréhension de votre message.

QUESTIONS PRATIQUES : AVANT / APRÈS

Un graphique brut sorti de Matplotlib est un bon début d'exploration, mais c'est un mauvais outil de communication.

Avant : Le graphique d'exploration [Image d'un graphique Matplotlib brut avec des couleurs par défaut] Ce graphique est fonctionnel, mais il est bruyant et ne raconte aucune histoire.

Après : Le graphique explicatif Pour le transformer, on applique les principes du storytelling :

1. Titre clair et actionnable : Au lieu de "Taux de survie par classe", préférez "Les passagers de 1ère classe avaient 2.5x plus de chances de survivre".

2. Couleurs stratégiques : Utilisez une couleur neutre (gris) pour l'ensemble des données et une couleur vive (bleu, orange) pour mettre en évidence uniquement le point que vous voulez que votre audience retienne.

3. Annotations : Ajoutez du texte directement sur le graphique pour expliquer un point précis ou donner un chiffre clé.

4. Nettoyage : Supprimez les grilles, les bordures et les axes inutiles.

[Image d'un graphique "Après", clair et annoté]

CHALLENGE POUR VOUS !

Vous êtes le data scientist en chef de la compagnie White Star Line et vous devez présenter les conclusions de votre analyse sur le naufrage à un comité de direction qui n'est pas technique.

Votre mission : Racontez l'histoire du Titanic en 3 graphiques clairs et simples. Votre but est d'expliquer quels facteurs ont le plus influencé la survie. Pensez à la structure en 3 actes et au principe du rapport signal/bruit.

Quels graphiques choisiriez-vous ? Quels titres leur donneriez-vous ?

28 SQL POUR LA DATA SCIENCE : POURQUOI VOUS NE POUVEZ PAS L'IGNORER

Jusqu'à présent, nous avons principalement travaillé avec des fichiers CSV. Mais dans le monde réel, les données vivent rarement dans de simples fichiers. Le plus souvent, elles sont stockées, organisées et protégées dans des bases de données. Pour interroger ces bases de données, extraire les informations dont vous avez besoin et les préparer pour vos modèles, il n'y a qu'un seul langage universel : le SQL (Structured Query Language). L'ignorer, c'est se priver d'un accès direct à la source de vérité.

LE LANGAGE DES BASES DE DONNÉES

Structure d'une base de données relationnelle Imaginez une base de données comme une armoire de classement (la base) contenant plusieurs classeurs (les tables). Chaque table est comme un tableau Excel, avec des lignes et des colonnes.

- Tables : Contiennent des données sur un sujet spécifique (ex: une table Clients, une table Commandes).

- Clé Primaire : C'est l'identifiant unique de chaque ligne dans une table (ex: ClientID dans la table Clients). Il ne peut y avoir de doublons.

- Clé Étrangère : C'est une colonne dans une table qui fait référence à la clé primaire d'une autre table. C'est le "lien" qui permet de connecter les tables entre elles (ex: ClientID dans la table Commandes pour savoir quel client a passé quelle commande).

Les commandes SQL fondamentales Une requête SQL se lit presque comme une phrase en anglais :

- SELECT : Choisit les colonnes que vous voulez voir.

- FROM : Indique la table dans laquelle vous voulez chercher.
- WHERE : Filtre les lignes pour ne garder que celles qui respectent une certaine condition.
- GROUP BY : Regroupe des lignes pour effectuer des calculs sur chaque groupe (ex: calculer le total des ventes par client).
- ORDER BY : Trie les résultats selon une ou plusieurs colonnes.

L'importance des JOIN La véritable puissance de SQL réside dans les JOIN. Une clause JOIN permet de combiner des lignes de deux ou plusieurs tables en se basant sur une colonne commune (généralement une clé primaire et une clé étrangère). C'est ce qui vous permet de répondre à des questions comme "Quel est le nom des clients qui ont acheté tel produit ?".

Questions Pratiques : SQL avec Python

Pas besoin d'un système de base de données complexe pour commencer. La bibliothèque sqlite3, incluse par défaut dans Python, nous permet de créer et d'interroger une base de données directement en mémoire.

```
import sqlite3
import pandas as pd

# Créer une connexion à une base de données en mémoire
conn = sqlite3.connect(':memory:')
cursor = conn.cursor()

# Créer deux tables
cursor.execute('''
CREATE TABLE Clients (
    ClientID INT PRIMARY KEY,
    Nom TEXT
)''')

cursor.execute('''
CREATE TABLE Commandes (
    CommandeID INT PRIMARY KEY,
    ClientID INT,
    Montant REAL,
    FOREIGN KEY (ClientID) REFERENCES Clients(ClientID)
)''')

# Insérer des données
cursor.execute("INSERT INTO Clients VALUES (1, 'Alice'), (2, 'Bob'), (3,
'Charlie')")
cursor.execute("INSERT INTO Commandes VALUES (101, 1, 50.0), (102, 2,
75.5), (103, 1, 25.0), (104, 3, 120.0), (105, 2, 30.0)")
conn.commit()

# Exemple 1 : Filtrer avec WHERE
print("--- Commandes de plus de 60€ ---")
query1 = "SELECT * FROM Commandes WHERE Montant > 60"
```

```
print(pd.read_sql_query(query1, conn))

# Exemple 2 : Agréger avec GROUP BY
print("\n--- Dépense totale par client ---")
query2 = """
SELECT ClientID, SUM(Montant) as TotalDepense
FROM Commandes
GROUP BY ClientID
"""
print(pd.read_sql_query(query2, conn))

# Exemple 3 : Joindre les tables avec JOIN
print("\n--- Nom du client pour chaque commande ---")
query3 = """
SELECT c.Nom, cmd.CommandeID, cmd.Montant
FROM Commandes cmd
JOIN Clients c ON cmd.ClientID = c.ClientID
"""
print(pd.read_sql_query(query3, conn))

# Fermer la connexion
conn.close()
```

Challenge pour vous !

Mettez vos nouvelles compétences SQL à l'épreuve.

Votre mission : À partir des deux tables que nous avons créées (Clients et Commandes), écrivez une seule requête SQL qui retourne le top 2 des clients ayant dépensé le plus, en affichant leur nom et le montant total de leurs dépenses, trié par ordre décroissant.

29 Introduction aux APIs : Collecter des données en temps réel sur le web

Toutes les données ne sont pas sagement rangées dans des fichiers CSV statiques. Sur le web, les données sont vivantes : le livre d'une action change chaque seconde, la météo est mise à jour en permanence, de nouveaux tweets sont publiés constamment. Les APIs (Interfaces de Programmation d'Application) sont des ponts qui nous permettent de nous brancher directement sur ces services web pour obtenir des données fraîches en temps réel.

Dialoguer avec le web

Qu'est-ce qu'une API REST ? Imaginez que vous êtes au restaurant. Vous ne rentrez pas dans la cuisine pour préparer votre plat ; vous consultez un menu (la documentation), puis vous passez votre commande à un serveur (l'API) qui vous rapporte le plat. Une API REST est un ensemble de règles et de conventions qui agit comme ce serveur. Elle définit comment les différents programmes informatiques peuvent communiquer entre eux sur internet de manière standardisée.

Comprendre les requêtes HTTP Cette communication se fait via des "requêtes HTTP". Les deux plus courantes sont :

- GET : C'est la requête la plus fréquente. Elle sert à demander des données à un serveur. (Ex: "Donne-moi la météo actuelle à Paris").
- POST : Elle sert à envoyer des données à un serveur pour créer ou mettre à jour une ressource. (Ex: "Publie ce message sur mon fil d'actualité").

Le format JSON Quand le serveur vous répond, il le fait le plus souvent dans un format de données appelé JSON (JavaScript Object Notation). C'est un format texte, léger et facile à lire pour les humains, qui ressemble énormément à un dictionnaire Python (des paires de clé-valeur).

L'importance de la documentation et des clés Chaque API est unique. La première étape, et la plus importante, est toujours de lire la documentation de l'API. C'est le "menu" qui vous expliquera quelles informations vous pouvez demander et comment les demander. De nombreuses APIs nécessitent aussi une clé d'authentification (API Key), une sorte de mot de passe personnel qui vous identifie et vous autorise à utiliser le service.

Interroger une API avec Python

La bibliothèque requests est le moyen le plus simple et le plus populaire pour faire des requêtes HTTP en Python.

```
import requests
import json

# 1. Définir l'URL de l'API que nous voulons interroger
# JSONPlaceholder est une fausse API parfaite pour s'entraîner
url = "https://jsonplaceholder.typicode.com/todos/1"

# 2. Faire une requête GET à cette URL
response = requests.get(url)

# 3. Vérifier si la requête a réussi
# Un code 200 signifie "OK"
if response.status_code == 200:
    # 4. Parser la réponse JSON en un dictionnaire Python
    data = response.json()

    print("Données reçues de l'API :")
    # On utilise json.dumps pour un affichage plus joli
    print(json.dumps(data, indent=4))

    print(f"\nTitre de la tâche : {data['title']}")
else:
    print(f"Erreur lors de la requête : {response.status_code}")
```

Challenge pour vous !

Les APIs permettent de faire des choses très amusantes. Utilisons "The Dog API" pour égayer notre journée.

Votre mission : Utilisez la bibliothèque requests pour interroger l'API "The Dog API" (qui est gratuite et ne nécessite pas de clé) afin de récupérer une image aléatoire d'une race de chien spécifique (par exemple, un Beagle). Ensuite, affichez cette image dans votre notebook.

Indice : Vous aurez besoin de la bibliothèque IPython.display pour afficher l'image à partir de son URL.

30 Web Scraping avec BeautifulSoup : Quand les données ne sont pas servies sur un plateau

Parfois, les données que vous convoitez sont là, visibles sur une page web, mais il n'existe aucune API pour vous permettre de les récupérer proprement. Faut-il abandonner ? Non ! Le web scraping (ou "moissonnage web") est la technique qui permet d'écrire un programme pour naviguer sur une page web et en extraire les informations automatiquement. C'est une compétence extrêmement puissante, mais qui doit être utilisée avec précaution et éthique.

Comprendre la structure d'une page web

Les bases du HTML Une page web est structurée par un langage appelé HTML (HyperText Markup Language). Il est composé de "balises" (tags) qui décrivent le contenu.

- Tags : Ils définissent le type d'un élément. Par exemple, <h1> est un titre principal, <p> est un paragraphe, <a> est un lien.
- Attributs : Ils donnent des informations supplémentaires sur un tag. Pour un lien <a>, l'attribut href contient l'URL de destination.
- Classes et IDs : Ce sont des attributs spéciaux (class et id) que les développeurs web utilisent pour nommer et styliser des éléments spécifiques. Pour nous, scrappeurs, ce sont des cibles parfaites pour extraire des données précises.

Considérations éthiques et légales Le web scraping n'est pas un acte anodin. Il faut respecter quelques règles d'or :

1. Respecter le fichier robots.txt : Chaque site web a un fichier (ex: www.example.com/robots.txt) qui indique aux robots (comme votre script) quelles parties du site ils ont le droit de visiter ou non. Il faut toujours le consulter et le respecter.
2. Ne pas surcharger les serveurs : Ne faites pas des centaines de requêtes en quelques secondes. Comportez-vous comme un humain : faites des pauses entre vos requêtes (avec time.sleep() en Python) pour ne pas ralentir ou faire tomber le site que vous visitez.
3. Vérifier les conditions d'utilisation : Lisez les conditions d'utilisation du site pour savoir si le scraping de ses données est autorisé.

Extraire des données avec BeautifulSoup

La bibliothèque BeautifulSoup est l'outil de prédilection en Python pour "parser" (analyser et naviguer) le code HTML d'une page.

```
import requests
from bs4 import BeautifulSoup

# 1. Obtenir le HTML de la page avec requests
url = 'http://quotes.toscrape.com/'
response = requests.get(url)

# 2. Créer un objet BeautifulSoup pour le parser
soup = BeautifulSoup(response.text, 'html.parser')

# 3. Extraire des informations
# Trouver le premier titre h1
premier_titre = soup.find('h1').text
print(f"Titre de la page : {premier_titre}\n")

# Trouver la première citation (qui est dans un tag <span class="text">)
premiere_citation = soup.find('span', class_='text').text
print(f"Première citation : {premiere_citation}\n")

# Trouver tous les liens de la page
print("Quelques liens de la page :")
for link in soup.find_all('a')[:5]: # On affiche les 5 premiers
    print(link.get('href'))
```

CHALLENGE POUR VOUS !

Le site quotes.toscrape.com est spécialement conçu pour s'entraîner au scraping en toute légalité.

Votre mission : Scrapez la page d'accueil de quotes.toscrape.com et récupérez les trois informations suivantes pour la toute première citation de la page :

1. Le texte complet de la citation.
2. Le nom de son auteur.
3. La liste de ses "tags" (ex: "love", "inspirational", etc.).

31 GIT & GITHUB POUR LES DATA SCIENTISTS : COLLABOREZ SANS TOUT CASSER

Imaginez que vous travaillez sur un projet important. Pour sauvegarder vos progrès, vous créez des copies de votre fichier principal : projet_v1.py, projet_v2.py, projet_v2_corrigé.py, projet_final.py, projet_final_VRAIMENT_final.py... C'est le chaos assuré. Git est le système de contrôle de version professionnel conçu pour éviter ce désordre, suivre l'historique de vos modifications et collaborer efficacement.

L'ART DE VOYAGER DANS LE TEMPS

Pourquoi le versioning est-il crucial ? Le contrôle de version vous permet de :

- Sauvegarder l'historique complet de votre projet. Chaque modification est enregistrée.
- Revenir en arrière à n'importe quelle version précédente si vous cassez quelque chose. C'est une machine à remonter le temps pour votre code.
- Collaborer avec d'autres personnes sur le même projet sans écraser le travail des autres.
- Expérimenter de nouvelles idées sur des "branches" séparées sans affecter la version principale de votre projet.

Différence entre Git et GitHub C'est une confusion fréquente.

- Git est l'outil, le logiciel que vous installez sur votre ordinateur. Il gère l'historique de votre projet localement.
- GitHub (ainsi que GitLab, Bitbucket...) est une plateforme en ligne, un service web qui héberge vos dépôts Git. C'est le "réseau social" de votre code, qui facilite le partage, la collaboration et la sauvegarde à distance.

Les concepts de base

- Dépôt (Repository ou "repo") : C'est le dossier de votre projet, avec tout son historique Git.
- Commit : C'est un "instantané", une sauvegarde d'une version de votre projet à un moment donné. Chaque commit a un message qui décrit les changements effectués.
- Branche (Branch) : C'est une ligne de développement indépendante. La branche principale s'appelle généralement main (ou master). Vous pouvez créer de nouvelles branches pour travailler sur de nouvelles fonctionnalités sans toucher à la version stable.

LES COMMANDES DE BASE

Voici les commandes que vous utiliserez 95% du temps. Vous les tapez dans un terminal (ou une interface comme Git Bash sur Windows).

1. git init : À n'utiliser qu'une seule fois au début d'un projet. Transforme un dossier normal en dépôt Git.
2. git add <nom_du_fichier> : Ajoute un fichier à la "zone de transit" (staging area). Cela prépare le fichier à être inclus dans le prochain commit. Utilisez git add . pour ajouter tous les fichiers modifiés.
3. git commit -m "Votre message descriptif" : Enregistre un instantané de tous les fichiers en transit. Le message doit expliquer clairement ce que vous avez changé.
4. git push : Envoie vos commits locaux vers votre dépôt distant sur GitHub. C'est la façon de partager et de sauvegarder votre travail.
5. git pull : Récupère les changements qui ont été faits sur le dépôt distant (par exemple, par un collaborateur) et les fusionne avec votre version locale.

Le workflow typique :

1. Créer un nouveau dépôt vide sur GitHub.
2. Dans le dossier de votre projet sur votre ordinateur, taper git init.
3. Lier votre dépôt local au dépôt distant sur GitHub.
4. Modifier votre code.
5. Faire git add .
6. Faire git commit -m "Ajout de la fonctionnalité X".
7. Faire git push.

CHALLENGE POUR VOUS !

Il n'y a qu'une seule façon d'apprendre Git : en l'utilisant.

Votre mission :

1. Prenez un de vos projets précédents (celui sur le Titanic ou la prédiction des prix de l'immobilier, par exemple).
2. Créez un nouveau dépôt sur votre compte GitHub pour ce projet.
3. Sur votre ordinateur, ouvrez un terminal dans le dossier du projet et initialisez Git (git init).
4. Faites votre premier commit en ajoutant tous les fichiers du projet.
5. Suivez les instructions de GitHub pour lier votre dépôt local et "pousser" votre premier commit.

Félicitations, votre projet est maintenant versionné et sauvegardé professionnellement !

32 DÉPLOYER SON PREMIER MODÈLE AVEC FLASK : DE VOTRE ORDINATEUR AU MONDE ENTIER

Un modèle de Machine Learning dans un notebook, c'est bien. Un modèle utilisable par n'importe qui via une simple URL, c'est beaucoup mieux. Le déploiement est l'étape qui transforme votre projet de recherche en un produit réel. Flask est un "micro-framework" Python parfait pour créer une API web simple et légère autour de votre modèle, le rendant accessible au monde entier.

DE L'ORDINATEUR AU WEB

Qu'est-ce qu'une application web ? C'est un programme qui s'exécute sur un serveur (un ordinateur connecté à internet) et qui est accessible via un navigateur web. Quand vous tapez une URL, vous demandez à un serveur d'exécuter une application et de vous renvoyer le résultat (souvent une page web ou des données).

Qu'est-ce qu'un framework ? Un framework comme Flask vous fournit une structure et des outils pour construire une application web sans avoir à réinventer la roue. Flask est qualifié de "micro" car il est minimaliste et vous laisse une grande liberté, ce qui est parfait pour créer des APIs simples.

Le concept de "route" Une route est un lien entre une URL et une fonction Python. Vous définissez une route (par exemple /predict) et vous lui associez une fonction. Quand un utilisateur envoie une requête à cette URL, Flask exécute la fonction correspondante et renvoie le résultat.

Comment "pickler" un modèle ? Un modèle entraîné est un objet Python qui existe en mémoire. Pour le sauvegarder et le réutiliser plus tard dans notre application Flask (sans avoir à le ré-entraîner à chaque fois), nous devons le sérialiser, c'est-à-dire le convertir en un flux d'octets que l'on peut enregistrer dans un fichier. La bibliothèque Python standard pour cela s'appelle pickle.

Créer une API avec Flask

Le processus se fait en deux temps : d'abord on entraîne et on sauvegarde le modèle, ensuite on crée l'application Flask.

1. Entraîner et sauvegarder le modèle (fichier train.py)

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
import pickle

# Charger les données et entraîner un modèle simple
iris = load_iris()
X = iris.data
y = iris.target
model = LogisticRegression(max_iter=200)
model.fit(X, y)

# Sauvegarder le modèle entraîné dans un fichier avec pickle
with open('model.pkl', 'wb') as f:
    pickle.dump(model, f)
```

2. Créer l'application Flask (fichier app.py)

```
from flask import Flask, request, jsonify
import pickle
import numpy as np

# Créer l'application Flask
app = Flask(__name__)

# Charger le modèle au démarrage de l'application
with open('model.pkl', 'rb') as f:
    model = pickle.load(f)

# Définir la route pour la prédiction
@app.route('/predict', methods=['POST'])
def predict():
    # Obtenir les données JSON de la requête
    data = request.get_json(force=True)

    # Convertir les données en un tableau NumPy
```

```
    features = np.array(data['features']).reshape(1, -1)

    # Faire la prédiction
    prediction = model.predict(features)

    # Retourner la prédiction en format JSON
    return jsonify({'prediction': int(prediction[0])})

# Lancer l'application
if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

Pour lancer votre API, il suffit d'exécuter python app.py dans votre terminal.

CHALLENGE POUR VOUS !

Votre API est en livre d'exécution, mais comment l'interroger ?

Votre mission : Créez un second script Python, test_api.py, qui :

1. Utilise la bibliothèque requests pour envoyer une requête POST à votre API locale (à l'URL http://127.0.0.1:5000/predict).
2. Dans cette requête, envoyez les données d'une fleur Iris à prédire au format JSON.
3. Récupérez la réponse de l'API et affichez la prédiction retournée.

C'est la manière standard de tester et d'interagir avec une API de machine learning.

33 DOCKER POUR LA DATA SCIENCE : "MAIS ÇA MARCHAIT SUR MA MACHINE !".

"Mais, ça marchait sur ma machine !" C'est la phrase la plus redoutée en informatique, une source de frustration infinie quand un code qui fonctionne parfaitement chez vous plante chez un collègue ou sur un serveur. Ce problème vient souvent de différences d'environnement : une autre version de Python, une librairie manquante... Docker est la solution ultime à ce problème. Il vous permet d'empaqueter votre code ET tout son environnement (librairies, versions, dépendances...) dans une boîte standardisée et isolée, un "conteneur", qui fonctionnera à l'identique partout.

L'ART D'EMPAQUETER SON ENVIRONNEMENT

Différence entre une Machine Virtuelle (VM) et un Conteneur

- Une Machine Virtuelle est un ordinateur complet émulé. Elle a son propre système d'exploitation invité, ce qui la rend lourde et lente à démarrer. C'est comme construire une maison entière (avec fondations, murs, toit) pour y faire tourner une seule application.
- Un Conteneur Docker est beaucoup plus léger. Il partage le système d'exploitation de la machine hôte et n'empaquète que l'application et ses dépendances directes. C'est comme louer un appartement dans un immeuble existant : vous avez votre propre espace isolé, mais vous partagez les infrastructures communes.

[Image d'une comparaison entre Machine Virtuelle et Conteneur Docker]

Qu'est-ce qu'une Image Docker et un Conteneur ? C'est la différence entre une recette de cuisine et le gâteau lui-même.

- Une Image Docker est un plan, un modèle en lecture seule. Elle contient toutes les instructions pour créer un environnement (ex: "Prends Python 3.9, installe telle et telle librairie, copie ces fichiers...").
- Un Conteneur est une instance vivante et exécutable d'une image. Vous pouvez en lancer, en arrêter et en supprimer autant que vous voulez à partir d'une seule et même image. C'est le gâteau que vous avez préparé en suivant la recette.

L'anatomie d'un Dockerfile Le Dockerfile est un simple fichier texte qui contient la "recette" pour construire votre image, étape par étape.

- FROM : Définit l'image de base sur laquelle on va construire (ex: python:3.9-slim).
- WORKDIR : Définit le répertoire de travail à l'intérieur du conteneur.
- COPY : Copie des fichiers de votre machine vers l'intérieur du conteneur.
- RUN : Exécute une commande pendant la construction de l'image (typiquement pip install).
- CMD : La commande qui sera exécutée au démarrage du conteneur.

CONSTRUIRE SON PREMIER CONTENEUR

Pour "dockeriser" notre application Flask, nous avons besoin de deux fichiers.

1. Le fichier requirements.txt Ce fichier liste toutes les librairies Python nécessaires.

```
flask
numpy
scikit-learn
```

2. Le Dockerfile

```
# Utiliser une image Python officielle comme base
FROM python:3.9-slim

# Définir le répertoire de travail
WORKDIR /app

# Copier le fichier des dépendances
COPY requirements.txt .

# Installer les dépendances
RUN pip install -r requirements.txt

# Copier le reste du code de l'application
COPY . .
```

```
# Exposer le port que Flask utilise
EXPOSE 5000

# La commande pour lancer l'application
CMD ["python", "app.py"]
```

3. Construire l'image et lancer le conteneur Ouvrez un terminal dans le dossier de votre projet et tapez :

```
# Construire l'image et lui donner un nom ('-t' pour tag)
docker build -t flask-api .

# Lancer un conteneur à partir de l'image
# '-p 5000:5000' mappe le port 5000 du conteneur au port 5000 de votre machine
docker run -p 5000:5000 flask-api
```

CHALLENGE POUR VOUS !

Il est temps de mettre en pratique la puissance de Docker.

Votre mission :

1. Reprenez le projet de l'application Flask de l'chapitre S32.
2. Créez les fichiers requirements.txt et Dockerfile comme décrits ci-dessus.
3. Construisez l'image Docker de votre application.
4. Lancez le conteneur.
5. Prouvez que votre application fonctionne en utilisant votre script test_api.py pour interroger l'API qui tourne maintenant à l'intérieur du conteneur.

Félicitations, vous venez de créer une application de machine learning portable et reproductible !

34 INTRODUCTION À KAGGLE : PARTICIPEZ À VOTRE PREMIÈRE COMPÉTITION

Vous avez appris les algorithmes, maîtrisé les outils et même déployé un modèle. Et maintenant ? Il est temps de vous mesurer au monde, d'apprendre des meilleurs et de construire votre portfolio. Kaggle est la plus grande communauté de data scientists au monde. C'est à la fois un terrain de jeu pour tester vos compétences, une bibliothèque de connaissances infinie et une vitrine pour votre travail.

LES 3 PILIERS DE KAGGLE

Kaggle repose sur trois piliers qui en font une plateforme unique.

1. Les Compétitions : C'est la partie la plus célèbre. Des entreprises et des chercheurs publient un problème et un jeu de données, et des milliers de data scientists du monde entier s'affrontent pour créer le modèle le plus performant. Les prix peuvent aller de la simple reconnaissance à des centaines de milliers de dollars.

2. Les Datasets : Kaggle héberge des dizaines de milliers de jeux de données sur tous les sujets imaginables. C'est une ressource inestimable pour s'entraîner et pour trouver des données pour vos projets personnels.

3. Les Notebooks (anciennement Kernels) : C'est un environnement de code en ligne (similaire aux Jupyter Notebooks) où vous pouvez analyser des données et construire vos modèles directement sur la plateforme, sans rien avoir à installer. Vous pouvez voir les notebooks des autres participants, apprendre de leurs techniques et partager votre propre travail.

Comment fonctionne une compétition ? Le principe est presque toujours le même :

- Les données : On vous fournit deux fichiers : train.csv (les données d'entraînement, avec les étiquettes/réponses) et test.csv (les données de test, sans les réponses).
- Votre mission : Vous devez entraîner votre modèle sur train.csv, puis l'utiliser pour faire des prédictions sur test.csv.
- La soumission : Vous générez un fichier de soumission (submission.csv) contenant vos prédictions dans un format spécifique.
- Le Leaderboard : Vous téléchargez ce fichier sur Kaggle, qui calcule automatiquement votre score et vous classe sur un "leaderboard" (tableau de classement) public.

VOS PREMIERS PAS SUR KAGGLE

1. Créer un compte Rendez-vous sur kaggle.com et créez un compte. C'est simple et gratuit.

2. Explorer une compétition pour débutants La compétition "Titanic: Machine Learning from Disaster" est le rite de passage de tout débutant. Elle est parfaite pour commencer car le problème est bien défini et de nombreux notebooks publics sont disponibles pour vous guider.

3. Forker un notebook existant "Forker" un notebook signifie en créer votre propre copie personnelle. C'est le meilleur moyen de démarrer. Trouvez un notebook populaire et clair, puis cliquez sur le bouton "Copy & Edit". Vous vous retrouverez dans l'environnement de code de Kaggle avec une copie du code et des données, prêt à être modifié.

4. Faire une modification et soumettre Une fois dans votre notebook, vous pouvez exécuter le code existant, le modifier, changer les paramètres d'un modèle, etc. Quand vous êtes satisfait, vous pouvez "commiter" votre notebook (le sauvegarder). Le processus de commit va souvent générer automatiquement le fichier submission.csv. Vous pourrez alors, depuis la page de votre notebook, soumettre ce fichier à la compétition.

CHALLENGE POUR VOUS !

Sur Kaggle, le but n'est pas de gagner du premier coup, mais de participer et d'apprendre.

Votre mission :

1. Allez sur la page de la compétition Titanic.
2. Trouvez un notebook simple et "forkez-le".

3. Ne changez qu'un seul paramètre dans le modèle existant (par exemple, le max_depth d'un arbre de décision, ou le nombre d'arbres dans une forêt aléatoire).

4. Commitez votre notebook et soumettez le fichier de prédictions généré.

5. Allez voir votre position sur le leaderboard.

Observez comment un simple changement peut influencer votre score. Bienvenue dans la communauté Kaggle !

35 L'Analyse en Composantes Principales (ACP) : Réduire le bruit pour mieux voir

Quand vous avez un jeu de données avec 3, 4, ou même 10 variables, il est encore possible de les visualiser et de les comprendre. Mais que faire quand vous en avez des centaines ? Comment "voir" les tendances dans un espace à 500 dimensions ? L'Analyse en Composantes Principales (ACP ou PCA en anglais) est une technique de réduction de dimensionnalité qui permet de "compresser" l'information contenue dans un grand nombre de variables en quelques nouvelles super-variables (les composantes principales) qui capturent l'essentiel de l'histoire.

L'art de la synthèse

L'intuition de la réduction de dimensionnalité Le but est de réduire le nombre de variables tout en perdant le moins d'information possible. L'ACP y parvient en créant de nouvelles variables, appelées composantes principales, qui sont des combinaisons linéaires des variables originales.

Que représente chaque composante principale ?

- La première composante principale (PC1) est un nouvel axe orienté dans la direction où les données varient le plus. C'est l'axe qui capture la plus grande partie de la "variance" (de la dispersion) du jeu de données.
- La deuxième composante principale (PC2) est le deuxième axe qui capture le plus de variance restante, à la condition d'être perpendiculaire (non corrélée) à la première.
- Et ainsi de suite pour les autres composantes.

En général, les premières composantes suffisent à capturer la grande majorité de l'information, nous permettant de passer de centaines de dimensions à seulement 2 ou 3, beaucoup plus faciles à visualiser.

Le concept de "variance expliquée" La "variance expliquée" nous dit quel pourcentage de l'information totale du jeu de données est capturé par chaque composante principale. Par exemple, si PC1 explique 70% de la variance et PC2 en explique 15%, alors une visualisation en 2D de ces deux composantes nous donne déjà une image qui représente 85% de l'histoire contenue dans nos données originales.

L'ACP avec Scikit-Learn

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

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.datasets import load_digits

# 1. Charger un dataset avec beaucoup de features (64 features pour des
images 8x8)
digits = load_digits()
X = digits.data
y = digits.target

# 2. Mettre les données à l'échelle (très important pour l'ACP)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 3. Appliquer l'ACP
# On demande à réduire à 2 composantes
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# 4. Visualiser le ratio de variance expliquée
print(f"Variance expliquée par PC1 :
{pca.explained_variance_ratio_[0]:.2%}")
print(f"Variance expliquée par PC2 :
{pca.explained_variance_ratio_[1]:.2%}")
print(f"Variance totale expliquée par les 2 composantes :
{np.sum(pca.explained_variance_ratio_):.2%}")

# 5. Projeter les données sur les 2 premières composantes
plt.figure(figsize=(10, 8))
sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1], hue=y, palette='viridis',
s=50)
plt.title('Projection des données "Digits" sur les 2 premières composantes
principales')
plt.xlabel('Première Composante Principale (PC1)')
plt.ylabel('Deuxième Composante Principale (PC2)')
plt.legend(title='Chiffre')
plt.show()
```

Le graphique montre que même avec seulement deux dimensions, on peut déjà voir des groupes distincts se former pour les différents chiffres. L'ACP a réussi à trouver la structure cachée dans les 64 dimensions originales.

CHALLENGE POUR VOUS !

L'ACP n'est pas seulement un outil de visualisation, elle peut aussi être une étape de pré-traitement pour améliorer la performance ou la vitesse d'un modèle.

Votre mission :

1. Appliquez l'ACP sur le dataset Iris pour réduire ses 4 dimensions à 2.

2. Entraînez un classifieur (par exemple, une LogisticRegression) sur ces 2 composantes principales. Évaluez son score.
3. Entraînez le même classifieur sur les 4 features originales du dataset. Évaluez son score.
4. Comparez les performances. Dans ce cas précis, l'ACP a-t-elle aidé ou nui à la performance du modèle ? Pourquoi, à votre avis ?

36 Travailler avec des données textuelles : Les bases du NLP

Le texte est l'une des sources de données les plus abondantes et les plus riches au monde : chapitres de presse, avis clients, e-mails, réseaux sociaux... Mais les modèles de machine learning ne comprennent que les nombres. Le Traitement du Langage Naturel (NLP ou TAL en français) est le domaine de l'IA qui nous apprend à transformer ce chaos de mots en une structure numérique ordonnée que les machines peuvent enfin comprendre.

Donner un sens aux mots

Pour qu'un ordinateur comprenne un texte, nous devons le décomposer et le quantifier.

Les étapes de nettoyage de texte Un texte brut est "bruyant". Le nettoyer est une première étape essentielle :

- Mise en minuscules : Pour que "Chat" et "chat" soient considérés comme le même mot.
- Suppression de la ponctuation : Les virgules, points, etc., n'apportent souvent pas de sens pour les modèles simples.
- Suppression des "stopwords" : On retire les mots très courants qui n'ont que peu de valeur sémantique, comme "le", "la", "de", "un", "est"...

La Tokenisation C'est le processus qui consiste à segmenter une phrase en une liste de mots individuels, appelés "tokens". La phrase "Le chat est gris" devient la liste de tokens ['le', 'chat', 'est', 'gris'].

La Vectorisation : Transformer les mots en nombres C'est l'étape cruciale. Les deux méthodes les plus classiques sont :

- Bag-of-Words (BoW) ou Sac de Mots : Cette technique représente chaque texte comme un "sac" où l'on a jeté tous les mots, sans tenir compte de l'ordre ou de la grammaire. On crée un tableau où chaque colonne correspond à un mot unique du vocabulaire total, et chaque ligne à un texte. La valeur dans une cellule est simplement le nombre de fois que le mot de cette colonne apparaît dans le texte de cette ligne.
- TF-IDF (Term Frequency-Inverse Document Frequency) : C'est une version améliorée du BoW. Elle donne plus de poids aux mots qui sont importants pour un document spécifique, mais rares dans l'ensemble des autres documents.
    - TF (Fréquence du Terme) : Comme pour le BoW, on compte la fréquence d'un mot dans un texte.

- IDF (Fréquence Inverse de Document) : On pénalise les mots qui apparaissent partout (comme "donnée" dans nos chapitres) et on valorise les mots plus rares et donc plus discriminants. Le score final TF-IDF est la multiplication des deux.

Le NLP avec Scikit-Learn

Scikit-Learn fournit des outils incroyablement efficaces pour réaliser ces étapes en une seule fois.

```
from sklearn.feature_extraction.text import CountVectorizer,
TfidfVectorizer

# Notre corpus de documents
corpus = [
    'Le ciel est bleu et le soleil brille.',
    'Le soleil de minuit est un phénomène du nord.',
    'Un ciel bleu est un beau ciel.'
]

# 1. Vectorisation avec Bag-of-Words (CountVectorizer)
bow_vectorizer = CountVectorizer()
X_bow = bow_vectorizer.fit_transform(corpus)

print("--- Bag-of-Words ---")
print("Vocabulaire appris :", bow_vectorizer.get_feature_names_out())
print("Matrice numérique (sparse) :\n", X_bow.toarray())


# 2. Vectorisation avec TF-IDF (TfidfVectorizer)
tfidf_vectorizer = TfidfVectorizer()
X_tfidf = tfidf_vectorizer.fit_transform(corpus)

print("\n--- TF-IDF ---")
print("Vocabulaire appris :", tfidf_vectorizer.get_feature_names_out())
print("Matrice numérique (sparse) :\n", X_tfidf.toarray())
```

Comme vous pouvez le voir, chaque texte est maintenant représenté par un vecteur de nombres, prêt à être utilisé par un modèle de machine learning.

Challenge pour vous !

Le NLP permet de résoudre des problèmes très concrets, comme le filtrage des spams.

Votre mission :

1. Trouvez un jeu de données de SMS (il en existe de nombreux sur Kaggle, souvent appelés "SMS Spam Collection"). Il contient des SMS et une étiquette ("ham" pour non-spam, "spam" pour spam).
2. Utilisez TfidfVectorizer pour transformer le texte des SMS en une matrice numérique.
3. Entraînez un modèle de LogisticRegression sur cette matrice pour classifier les messages.

4. Évaluez la performance de votre premier classifieur de spam !

37 Travailler avec des séries temporelles : Prédire les tendances

Ventes mensuelles, météo quotidienne, livre de la bourse à la seconde... De très nombreuses données ont une dimension temporelle fondamentale. On ne peut pas les analyser comme un simple tableau de chiffres, car l'ordre des points est crucial. L'analyse des séries temporelles est une branche spécialisée de la data science qui nous fournit les outils pour comprendre le passé et prédire le futur de ces données.

Comprendre le rythme du temps

Les composantes d'une série temporelle Une série temporelle peut être décomposée en trois parties, un peu comme on décomposerait un morceau de musique en mélodie, rythme et harmonie.

- La Tendance (Trend) : C'est la direction générale à long terme de la série. Est-ce qu'elle monte, descend, ou reste stable ?
- La Saisonnalité (Seasonality) : Ce sont les variations prévisibles et répétitives qui se produisent à des intervalles de temps fixes (par exemple, les ventes de glaces qui augmentent chaque été).
- Le Résidu (Residual) : C'est ce qui reste une fois qu'on a retiré la tendance et la saisonnalité. C'est le "bruit" aléatoire et imprévisible de la série.

[Image d'une décomposition de série temporelle]

L'importance de l'index temporel Pour que Pandas puisse utiliser ses super-pouvoirs temporels, il est essentiel que l'index de notre DataFrame ne soit pas juste une suite de nombres, mais un véritable index temporel (un DatetimeIndex). Cela nous permet de faire des sélections et des agrégations incroyablement intuitives (ex: "donne-moi toutes les données de l'année 2020", "calcule la moyenne par mois").

Le concept de stationnarité Une série est dite stationnaire si ses propriétés statistiques (comme sa moyenne et sa variance) ne changent pas au fil du temps. Intuitivement, une série stationnaire est "plate" : elle n'a ni tendance ni saisonnalité évidente. Ce concept est crucial car la plupart des modèles de prévision sophistiqués exigent que la série soit stationnaire pour fonctionner correctement.

Manipuler le temps avec Pandas

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose

# Créer un exemple de série temporelle
dates = pd.date_range(start='2021-01-01', periods=36, freq='M')
data = [i + (i//12)*5 + np.random.randn()*2 for i in range(36)]
ts = pd.Series(data, index=dates)

# 1. Resampling : Agréger par trimestre
quarterly_mean = ts.resample('Q').mean()
```

```
print("--- Moyenne par trimestre ---")
print(quarterly_mean.head())

# 2. Rolling Windows : Calculer une moyenne mobile sur 3 mois
rolling_mean = ts.rolling(window=3).mean()

# 3. Décomposer une série temporelle avec statsmodels
decomposition = seasonal_decompose(ts, model='additive')
fig = decomposition.plot()
plt.show()

# 4. Prédiction simple avec une moyenne mobile
# La prédiction pour le prochain point est la moyenne des 3 derniers
last_3_months = ts.tail(3)
prediction = last_3_months.mean()
print(f"\nPrédiction pour le mois suivant : {prediction:.2f}")
```

CHALLENGE POUR VOUS !

Le jeu de données "Air Passengers" est un classique pour l'analyse des séries temporelles. Il contient le nombre mensuel de passagers d'une compagnie aérienne de 1949 à 1960.

Votre mission :

1. Chargez ce jeu de données (disponible sur de nombreuses plateformes, y compris Kaggle).
2. Assurez-vous que l'index est bien un index temporel.
3. Visualisez la série brute. Vous devriez voir une tendance à la hausse et une forte saisonnalité.
4. Calculez et tracez une moyenne mobile sur 12 mois (rolling(window=12).mean()) sur le même graphique que la série originale.

Que remarquez-vous ? La moyenne mobile a-t-elle réussi à lisser les variations saisonnières pour ne faire ressortir que la tendance à long terme ?

38 LE TEST A/B : COMMENT PRENDRE DES DÉCISIONS BASÉES SUR DES PREUVES

Comment savoir si la nouvelle version de votre site web, avec son bouton vert, est réellement meilleure que l'ancienne, avec son bouton bleu ? On ne devine pas, on teste ! Le test A/B est la méthode scientifique rigoureuse utilisée par les entreprises du monde entier pour prendre des décisions basées sur des données et des preuves, plutôt que sur l'intuition.

LE FRAMEWORK DU TEST D'HYPOTHÈSE

Le test A/B est une application pratique du test d'hypothèse statistique. Le but est de déterminer si une différence observée entre deux groupes est "réelle" ou si elle est simplement due au hasard.

- L'hypothèse nulle (H0) : C'est l'hypothèse du statu quo. Elle postule qu'il n'y a aucune différence entre les deux versions. (Ex: "Le changement de couleur du bouton n'a aucun effet sur le taux de clics.")

- L'hypothèse alternative (H1) : C'est l'hypothèse que l'on cherche à prouver. Elle postule qu'il y a une différence significative. (Ex: "Le bouton vert a un taux de clics différent du bouton bleu.")

Qu'est-ce qu'une p-value ? Après avoir mené l'expérience, on calcule une p-value. La p-value est la probabilité d'observer les résultats que nous avons obtenus (ou des résultats encore plus extrêmes) si l'hypothèse nulle était vraie.

- Une petite p-value (généralement < 0.05) suggère que nos résultats sont très improbables sous l'hypothèse nulle. On a donc des preuves solides pour la rejeter et accepter l'hypothèse alternative. La différence observée est dite statistiquement significative.
- Une grande p-value (généralement ≥ 0.05) signifie que nos résultats sont tout à fait plausibles sous l'hypothèse nulle. On n'a pas assez de preuves pour la rejeter.

Mener un test A/B en Python

Imaginons que nous testons deux versions d'une page web.

- Version A (Contrôle) : 1000 visiteurs, 100 clics.
- Version B (Test) : 1000 visiteurs, 125 clics.

La version B semble meilleure, mais la différence est-elle statistiquement significative ?

```
import numpy as np
from statsmodels.stats.proportion import proportions_ztest

# Données de notre test A/B
visiteurs_A, clics_A = 1000, 100
visiteurs_B, clics_B = 1000, 125

# Calculer les taux de conversion
taux_conversion_A = clics_A / visiteurs_A
taux_conversion_B = clics_B / visiteurs_B

print(f"Taux de conversion A: {taux_conversion_A:.2%}")
print(f"Taux de conversion B: {taux_conversion_B:.2%}")

# Préparer les données pour le test statistique
nombre_de_clics = np.array([clics_A, clics_B])
nombre_de_visiteurs = np.array([visiteurs_A, visiteurs_B])

# Effectuer un test Z pour les proportions
stat, p_value = proportions_ztest(count=nombre_de_clics,
nobs=nombre_de_visiteurs)

print(f"\nP-value: {p_value:.4f}")

# Interpréter le résultat
alpha = 0.05  # Notre seuil de significativité
if p_value < alpha:
    print("Résultat significatif : On rejette l'hypothèse nulle (H0).")
```

```
    print("La version B est significativement différente de la version A.")
else:
    print("Résultat non significatif : On ne peut pas rejeter l'hypothèse
nulle (H0).")
    print("Nous n'avons pas assez de preuves pour conclure à une
différence.")
```

CHALLENGE POUR VOUS !

Le choix final dépend de votre interprétation de la p-value et du seuil de risque que vous êtes prêt à accepter.

Votre mission : Votre test A/B se termine et vous obtenez une p-value de 0.07. Votre seuil de significativité (alpha) est fixé à 0.05.

1. Quelle décision prenez-vous concernant l'hypothèse nulle ?
2. Pouvez-vous lancer la nouvelle version du site avec confiance en vous basant sur ce résultat ? Pourquoi ?

Réponse :

1. Puisque la p-value (0.07) est supérieure à notre seuil de significativité (0.05), nous ne pouvons pas rejeter l'hypothèse nulle.
2. Non, nous ne pouvons pas lancer la nouvelle version avec confiance. Un tel résultat signifie qu'il y a 7% de chances d'observer une telle différence (ou une différence encore plus grande) simplement par hasard, même si la nouvelle version n'a en réalité aucun effet. C'est généralement considéré comme une preuve insuffisante pour justifier un changement qui pourrait coûter du temps et de l'argent.

39 CONSTRUIRE SON PORTFOLIO DE DATA SCIENTIST : 3 PROJETS POUR CONVAINCRE UN RECRUTEUR

Votre portfolio est plus important que votre CV. Un CV liste ce que vous *dites* savoir faire ; un portfolio prouve ce que vous savez faire. C'est la preuve tangible de vos compétences, de votre curiosité et de votre capacité à mener un projet de A à Z. Apprenons à construire un portfolio qui raconte une histoire et qui impressionne les recruteurs.

LES INGRÉDIENTS D'UN PORTFOLIO RÉUSSI

Qu'est-ce qu'un bon projet de portfolio ? Un projet mémorable n'est pas forcément le plus complexe. Il doit combiner quatre éléments :

1. Une question claire : Le projet doit chercher à répondre à une question business ou scientifique intéressante.
2. Des données intéressantes : N'hésitez pas à sortir des sentiers battus. Un projet sur des données que vous avez collectées vous-même (via scraping ou API) aura toujours plus d'impact qu'un énième projet sur le Titanic.

3. Une analyse solide : Votre méthodologie doit être claire, justifiée et bien exécutée.
4. Une communication efficace : Vos conclusions doivent être présentées de manière simple, visuelle et convaincante.

Où l'héberger ? La réponse est simple : GitHub. C'est la plateforme standard pour héberger du code. Avoir un profil GitHub actif et bien organisé est en soi une compétence que les recruteurs recherchent.

Comment le présenter ?

- Un README.md clair : C'est la page d'accueil de votre projet. C'est la première chose (et parfois la seule) qu'un recruteur lira. Il doit être impeccable.
- Un notebook propre : Votre code doit être lisible, commenté et organisé. Un notebook doit raconter l'histoire de votre analyse, pas être un simple brouillon de code.

ANATOMIE DE 3 PROJETS CONVAINCANTS

Plutôt qu'un long dislivre, analysons la structure de trois types de projets qui fonctionnent bien dans un portfolio.

1. Le projet d'analyse exploratoire et de storytelling

- Exemple : Analyser les données des Airbnb d'une ville pour comprendre les facteurs qui influencent le prix.
- Compétences démontrées : Manipulation de données (Pandas), visualisation (Seaborn), storytelling, capacité à tirer des insights d'un jeu de données.
- Structure du README :
    - Contexte : Pourquoi cette analyse est-elle intéressante ?
    - Questions : Quelles sont les 3 questions clés auxquelles vous répondez (ex: "Quel quartier est le plus cher ?", "Le type de logement a-t-il un impact ?") ?
    - Découvertes clés : Présentez vos 3 graphiques les plus parlants avec une conclusion claire pour chacun.
    - Conclusion : Résumez vos découvertes en une phrase.

2. Le projet de Machine Learning de A à Z

- Exemple : Prédire le taux de désabonnement d'un service en ligne.
- Compétences démontrées : Tout le workflow de la data science (nettoyage, feature engineering, modélisation, évaluation).
- Structure du README :
    - Objectif Business : Quel problème concret cherchez-vous à résoudre ?
    - Source des données : D'où viennent les données ?

    - Méthodologie : Décrivez brièvement les étapes : nettoyage, modèle(s) testé(s), métrique d'évaluation choisie et pourquoi.
    - Résultats : Quel score final avez-vous obtenu ? Quelle est l'importance des différentes variables ?
    - Comment l'utiliser : Expliquez comment lancer votre code.

3. Le projet "produit" : API ou Application Web

- Exemple : L'application Flask que nous avons construite.
- Compétences démontrées : Déploiement, ingénierie logicielle (Docker, Flask), capacité à transformer un modèle en un produit utilisable.
- Structure du README :
    - Description : Que fait cette application ?
    - Technologies utilisées : Listez les outils (Python, Flask, Docker...).
    - Instructions d'installation et d'utilisation : Expliquez très clairement comment un utilisateur peut lancer votre application localement (avec Docker, c'est très simple !).
    - Exemple d'appel à l'API : Montrez un exemple de requête et la réponse attendue.

CHALLENGE POUR VOUS !

Il est temps de poser la première brique de votre portfolio professionnel.

Votre mission :

1. Choisissez UN des projets que vous avez réalisés au livre de cette série (le Titanic, la prédiction des prix, le classifieur de spam...). Même le plus simple fera l'affaire.
2. Créez un nouveau dépôt sur GitHub pour ce projet.
3. Rédigez un nouveau fichier README.md pour celui-ci en suivant la structure la plus appropriée parmi les trois exemples ci-dessus. Soyez clair, concis et professionnel.

PARTIE 4 SUJETS AVANCÉS ET CULTURE DATA

---

40 INTRODUCTION AUX RÉSEAUX DE NEURONES : LE CERVEAU DERRIÈRE LE DEEP LEARNING

Nous avons exploré des modèles puissants, des régressions aux forêts aléatoires. Mais comment passer à l'étape supérieure et commencer à imiter, même de très loin, le fonctionnement du cerveau

humain ? La réponse se trouve dans les réseaux de neurones, le moteur du Deep Learning. Et tout commence par l'atome de ce cerveau artificiel : le neurone unique, ou Perceptron.

De la biologie à l'algorithme

Qu'est-ce qu'un Perceptron ? Le Perceptron est une simplification mathématique d'un neurone biologique. Il reçoit plusieurs signaux en entrée, les traite, et produit un unique signal en sortie.

1. Entrées et Poids : Chaque entrée (x1, x2, ...) est associée à un poids (w1, w2, ...). Ce poids représente l'importance de cette entrée. Un poids élevé signifie que l'entrée a beaucoup d'influence sur la décision.

2. Somme pondérée : Le Perceptron calcule la somme de toutes les entrées multipliées par leurs poids respectifs.

3. Fonction d'activation : Cette somme est ensuite passée à travers une fonction d'activation. Dans le Perceptron le plus simple, c'est une fonction "marche d'escalier" : si la somme dépasse un certain seuil, le neurone "s'active" et renvoie 1, sinon il renvoie 0.

[Image d'un schéma simple de Perceptron]

Du Perceptron au Réseau de Neurones (MLP) Un seul neurone est limité. La véritable puissance vient de leur connexion. Un réseau de neurones multicouches (Multilayer Perceptron, MLP) est simplement une succession de couches de Perceptrons. La sortie d'une couche de neurones devient l'entrée de la couche suivante. C'est cette architecture en couches qui permet d'apprendre des relations extraordinairement complexes.

Comment un réseau apprend-il ? Le processus d'apprentissage se fait en deux temps :

1. Propagation avant (Forward Propagation) : On présente des données en entrée du réseau. L'information traverse toutes les couches, de la première à la dernière, jusqu'à produire une prédiction en sortie.

2. Rétropropagation de l'erreur (Backpropagation) : On compare la prédiction du réseau à la vraie réponse pour calculer une erreur. L'algorithme de rétropropagation propage alors cette erreur en sens inverse, de la dernière couche à la première, en ajustant légèrement les poids de chaque neurone pour réduire l'erreur. C'est en répétant ce cycle des milliers de fois que le réseau "apprend".

Implémenter un Perceptron avec NumPy

Implémentons un Perceptron capable d'apprendre la porte logique "ET" (le résultat est 1 uniquement si les deux entrées sont 1).

```
import numpy as np

class Perceptron:
    def __init__(self, learning_rate=0.1, n_iters=100):
        self.lr = learning_rate
        self.n_iters = n_iters
        self.activation_func = self._step_function
```

```
        self.weights = None
        self.bias = None

    def _step_function(self, x):
        return np.where(x >= 0, 1, 0)

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                linear_output = np.dot(x_i, self.weights) + self.bias
                y_predicted = self.activation_func(linear_output)

                # Mettre à jour les poids et le biais
                update = self.lr * (y[idx] - y_predicted)
                self.weights += update * x_i
                self.bias += update

    def predict(self, X):
        linear_output = np.dot(X, self.weights) + self.bias
        y_predicted = self.activation_func(linear_output)
        return y_predicted

# Données pour la porte logique ET
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])

# Entraîner le Perceptron
p = Perceptron()
p.fit(X, y)

# Faire des prédictions
predictions = p.predict(X)
print(f"Prédictions du Perceptron pour la porte ET : {predictions}")
print(f"Le modèle a-t-il appris correctement ? {np.all(predictions == y)}")
```

HALLENGE POUR VOUS !

Le Perceptron simple est puissant, mais il a une limite fondamentale.

Votre mission : Pourquoi un Perceptron simple ne peut-il pas résoudre le problème du OU Exclusif (XOR) ? (Rappel : XOR renvoie 1 si les deux entrées sont différentes, et 0 si elles sont identiques).

Dessinez les quatre points du problème XOR sur un graphique 2D et essayez de tracer une seule ligne droite pour séparer les points "1" des points "0". Expliquez pourquoi c'est impossible.

Réponse : Le problème XOR n'est pas linéairement séparable. Il est impossible de tracer une seule ligne droite pour séparer les points (0,1) et (1,0) des points (0,0) et (1,1). Comme un Perceptron simple ne peut créer qu'une frontière de décision linéaire, il est incapable de résoudre ce problème. C'est cette

limitation qui a conduit au développement des réseaux de neurones multicouches, capables de créer des frontières de décision non-linéaires complexes.

41 Keras & TensorFlow : Construire son premier réseau de neurones en 10 minutes

Implémenter des réseaux de neurones à la main avec NumPy, comme nous l'avons fait pour le Perceptron, est extrêmement instructif. Mais c'est aussi fastidieux et peu pratique pour des modèles complexes. Des frameworks comme TensorFlow et son API de haut niveau Keras nous permettent de construire, d'entraîner et de déployer des réseaux de neurones sophistiqués en quelques lignes de code seulement.

Le langage du Deep Learning

Qu'est-ce qu'un tenseur ? Vous connaissez les vecteurs (tableaux 1D) et les matrices (tableaux 2D). Un tenseur est simplement la généralisation de ce concept à un nombre quelconque de dimensions. C'est la structure de données fondamentale de TensorFlow.

- Un nombre seul (un scalaire) est un tenseur de dimension 0.
- Un vecteur est un tenseur de dimension 1.
- Une matrice est un tenseur de dimension 2.
- Un lot d'images en couleur serait un tenseur de dimension 4 (nombre d'images, hauteur, largeur, canaux de couleur).

L'API Séquentielle vs Fonctionnelle de Keras Keras offre deux manières principales de construire des modèles :

- L'API Séquentielle : C'est la plus simple. On l'utilise pour créer un modèle qui est une simple pile linéaire de couches, ce qui est le cas pour la majorité des réseaux de neurones. On définit le modèle comme une Sequential() et on y ajoute des couches une par une avec .add().
- L'API Fonctionnelle : Elle est plus flexible et plus puissante. Elle permet de construire des architectures complexes, comme des modèles avec plusieurs entrées ou plusieurs sorties, ou des couches partagées.

Le cycle de vie d'un modèle Keras Construire et entraîner un modèle avec Keras suit toujours quatre étapes claires :

1. Définir le modèle : On choisit les couches et leur agencement.
2. compile() : On configure le processus d'apprentissage. On doit spécifier :
    - Un optimiseur (optimizer) : L'algorithme qui met à jour les poids (ex: adam).
    - Une fonction de perte (loss) : La mesure de l'erreur que le modèle doit minimiser.
    - Des métriques (metrics) : Les scores à surveiller pendant l'entraînement (ex: accuracy).

3. fit() : On entraîne le modèle sur les données d'entraînement pendant un certain nombre d'époques (une époque est un passage complet sur l'ensemble des données).

4. evaluate() / predict() : Une fois entraîné, on évalue sa performance sur des données de test ou on l'utilise pour faire de nouvelles prédictions.

CONSTRUIRE UN CLASSIFIEUR D'IMAGES

Utilisons Keras pour construire un réseau de neurones capable de classifier des images de vêtements à partir du jeu de données Fashion MNIST.

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

# 1. Charger le jeu de données Fashion MNIST
fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()

# Nom des classes
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# 2. Prétraiter les données
# Mettre à l'échelle les valeurs des pixels entre 0 et 1
train_images = train_images / 255.0
test_images = test_images / 255.0

# 3. Construire le modèle séquentiel
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),     # Aplatit l'image 28x28
en un vecteur de 784
    keras.layers.Dense(128, activation='relu'),    # Couche cachée avec 128
neurones
    keras.layers.Dense(10)                          # Couche de sortie avec
10 neurones (un par classe)
])

# 4. Compiler le modèle
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

# 5. Entraîner le modèle
print("--- Entraînement du modèle ---")
model.fit(train_images, train_labels, epochs=10)

# 6. Évaluer la performance
print("\n--- Évaluation du modèle ---")
```

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
print('\nTest accuracy:', test_acc)
```

Challenge pour vous !

L'architecture d'un réseau de neurones (le nombre de couches et de neurones) est un hyperparamètre crucial.

Votre mission :

1. Reprenez le modèle ci-dessus et ajoutez une couche cachée (Dense) supplémentaire de 64 neurones entre les deux couches existantes.
2. Augmentez le nombre d'époques à 20.
3. La performance sur l'ensemble de test s'améliore-t-elle ? Que se passe-t-il si vous ajoutez trop de neurones (par exemple, 512 dans chaque couche) ? Y a-t-il un risque d'overfitting ?

42 Le Deep Learning pour les images : Reconnaître un chat d'un chien (CNN)

Comment un réseau de neurones "voit"-il une image ? Un réseau de neurones classique, en aplatissant l'image, perd toute information spatiale. Les Réseaux de Neurones Convolutifs (Convolutional Neural Networks, ou CNN) sont une architecture spéciale, inspirée du cortex visuel humain, qui utilise des "filtres" glissants pour détecter des motifs (bords, coins, textures) de plus en plus complexes, couche après couche.

[Image d'un filtre convolutif glissant sur une image]

Les briques de la vision par ordinateur

Les deux couches clés : Convolution et Pooling

- La Convolution (Conv2D) : C'est le cœur du CNN. Au lieu de regarder tous les pixels à la fois, cette couche fait glisser de petits "filtres" (ou "kernels") sur toute l'image. Chaque filtre est un expert spécialisé dans la détection d'un motif très simple (une ligne verticale, un coin, une certaine couleur...). En appliquant ce filtre, on obtient une "carte de caractéristiques" (*feature map*) qui met en évidence les zones de l'image où le motif a été détecté.
- Le Pooling (ou Sous-échantillonnage) : Après une convolution, on a souvent une couche de pooling (généralement MaxPooling2D). Son but est de réduire la taille (hauteur et largeur) des cartes de caractéristiques. Cela rend le modèle plus rapide et plus robuste, car il devient moins sensible à la position exacte du motif dans l'image.

L'architecture typique d'un CNN Un CNN est une succession de ces blocs. Les premières couches apprennent à détecter des motifs très simples. Les couches suivantes combinent ces motifs simples pour en détecter de plus complexes (des yeux, des oreilles, des nez), et ainsi de suite, jusqu'à reconnaître des objets entiers. L'architecture typique est : [Conv2D -> Pooling] x N -> Flatten -> Dense -> Dense (Sortie)

Le Transfer Learning : Sur les épaules des géants Entraîner un CNN performant à partir de zéro demande des millions d'images et des semaines de calcul. Le transfer learning (ou apprentissage par transfert) est une technique qui consiste à prendre un modèle très puissant (comme VGG16, ResNet, EfficientNet), pré-entraîné par des chercheurs de Google ou Facebook sur d'immenses datasets, à retirer sa couche de classification finale, et à brancher notre propre classifieur par-dessus. On profite ainsi de toute la connaissance que le modèle a déjà acquise sur la détection de motifs génériques.

Construire un CNN avec Keras

Utilisons Keras pour construire un CNN capable de classifier les images du jeu de données CIFAR-10 (avions, voitures, oiseaux, chats, etc.).

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt

# 1. Charger et préparer le dataset CIFAR-10
(train_images, train_labels), (test_images, test_labels) =
keras.datasets.cifar10.load_data()

# Normaliser les valeurs des pixels entre 0 et 1
train_images, test_images = train_images / 255.0, test_images / 255.0

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

# 2. Construire le modèle CNN
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32,
3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# 3. Ajouter les couches denses pour la classification
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))

model.summary()

# 4. Compiler et entraîner le modèle
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

```
# 5. Évaluer le modèle
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
print(f"\nTest accuracy: {test_acc:.2f}")
```

CHALLENGE POUR VOUS !

Les filtres d'un CNN sont souvent considérés comme une "boîte noire". Mais on peut essayer de visualiser ce qu'ils ont appris.

Votre mission :

1. Accédez aux filtres de la toute première couche de convolution de votre modèle (model.layers[0].get_weights()[0]).
2. Utilisez Matplotlib pour visualiser quelques-uns de ces filtres.
3. Pouvez-vous deviner, en regardant ces petites images, quels types de motifs simples (lignes horizontales, lignes verticales, gradients de couleurs, etc.) ils ont appris à détecter ?

43 LE DEEP LEARNING POUR LE TEXTE : COMPRENDRE LE LANGAGE HUMAIN (RNN & TRANSFORMERS)

Le texte est séquentiel. L'ordre des mots change tout le sens d'une phrase. "Le chien a mordu l'homme" est très différent de "L'homme a mordu le chien". Comment un modèle de machine learning peut-il se souvenir du début d'une phrase pour comprendre la fin ? Les Réseaux de Neurones Récurrents (RNN) ont été la première réponse, en introduisant une forme de "mémoire" du passé.

LA MÉMOIRE ET L'ATTENTION

L'idée de boucle récurrente Un RNN est un réseau de neurones qui possède une boucle. À chaque étape, quand il traite un mot, il ne prend pas seulement ce mot en entrée, mais aussi une "mémoire" (un état caché) de ce qu'il a vu aux étapes précédentes. Cette sortie est ensuite réinjectée en entrée pour le mot suivant. C'est cette boucle qui lui permet de conserver un contexte et de comprendre des séquences.

[Image d'une boucle de réseau de neurones récurrent]

Le problème de la disparition du gradient et les solutions (LSTM & GRU) Les RNN simples ont une mémoire à court terme. Sur des textes longs, l'information du début de la phrase se "dissipe" et le gradient de l'erreur devient si petit qu'il disparaît, empêchant le réseau d'apprendre des relations à longue distance. Pour résoudre ce problème, des architectures plus complexes ont été créées :

- LSTM (Long Short-Term Memory) et GRU (Gated Recurrent Unit) : Ce sont des types de RNN sophistiqués avec des "portes" (*gates*) qui contrôlent méticuleusement quelles informations doivent être oubliées et quelles informations doivent être conservées dans la mémoire à long terme.

L'architecture Transformer et le mécanisme d'attention En 2017, un chapitre de Google, "Attention Is All You Need", a révolutionné le NLP. Il a introduit l'architecture Transformer, qui se passe complètement des boucles récurrentes. À la place, elle utilise un mécanisme d'attention qui permet au

modèle, pour chaque mot, de "regarder" tous les autres mots de la phrase et de décider lesquels sont les plus importants pour comprendre le contexte. Les modèles de pointe comme GPT et BERT sont basés sur cette architecture.

Le NLP moderne avec Keras et Hugging Face

1. Construire un RNN simple pour l'analyse de sentiments avec Keras

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Données d'exemple (simplifiées)
# 0 = négatif, 1 = positif
texts = ["j'adore ce film c'est génial", "je déteste ce film c'est nul"]
labels = [1, 0]

# Vectorisation et padding (étapes non montrées pour la clarté)
# ...

# Modèle RNN simple
model = keras.Sequential([
    layers.Embedding(input_dim=1000, output_dim=16), # Couche pour
apprendre les représentations de mots
    layers.SimpleRNN(32),
    layers.Dense(1, activation='sigmoid') # Sortie pour la classification
binaire
])

model.summary()
# La suite serait de compiler et d'entraîner le modèle
```

2. Utiliser un modèle pré-entraîné avec Hugging Face transformers La bibliothèque transformers de Hugging Face rend l'utilisation de modèles de pointe comme BERT incroyablement simple.

```
# Installer la bibliothèque
# !pip install transformers

from transformers import pipeline

# Charger un pipeline pré-entraîné pour l'analyse de sentiments
sentiment_pipeline = pipeline('sentiment-analysis')

# Analyser des phrases
result = sentiment_pipeline("J'adore cette série, les acteurs sont
incroyables !")
print(result)
# Résultat attendu : [{'label': 'POSITIVE', 'score': 0.99...}]
```

CHALLENGE POUR VOUS !

Le "remplissage de masque" (*masked language modeling*) est la tâche pour laquelle BERT a été initialement entraîné. Il doit deviner le mot manquant dans une phrase.

Votre mission :

1. Utilisez le pipeline de Hugging Face pour la tâche "fill-mask".
2. Donnez-lui une phrase comme "La capitale de la France est [MASK].".
3. Regardez les 5 prédictions les plus probables que le modèle vous propose pour remplacer le masque.

44 L'ÉTHIQUE EN IA : BIAIS, ÉQUITÉ ET RESPONSABILITÉ DES ALGORITHMES

"Un grand pouvoir implique de grandes responsabilités." Cette célèbre citation n'a jamais été aussi pertinente qu'à l'ère de l'intelligence artificielle. Nos modèles peuvent prédire, classifier et recommander avec une efficacité redoutable, mais ils peuvent aussi discriminer, amplifier les stéréotypes et prendre des décisions profondément injustes. C'est notre devoir, en tant que créateurs et utilisateurs de technologie, de comprendre, de mesurer et de combattre ces biais.

LES RACINES DE L'INJUSTICE ALGORITHMIQUE

D'où viennent les biais ? Un biais algorithmique n'est pas une erreur de code. Il est le reflet de notre monde et de nos choix.

- Biais dans les données : C'est la source la plus fréquente. Si les données d'entraînement reflètent des inégalités historiques (par exemple, si les données montrent que les hommes ont été historiquement plus promus à des postes de direction), le modèle apprendra et reproduira cette inégalité.
- Biais dans le modèle : Le choix d'un modèle ou d'une fonction de perte peut introduire un biais. Un modèle qui optimise uniquement pour la précision globale peut décider d'ignorer les sous-groupes minoritaires car ils ont peu d'impact sur le score final.
- Biais humain : Les data scientists font des choix : quelles variables inclure, comment gérer les valeurs manquantes, comment définir la "performance". Chacun de ces choix peut, consciemment ou non, introduire un biais.

Définir l'équité (Fairness) en IA L'équité n'est pas un concept mathématique unique. Il existe des dizaines de définitions, qui sont parfois contradictoires. Par exemple, veut-on que :

- Le modèle ait le même taux de prédictions positives pour tous les groupes (parité démographique) ?
- Le modèle ait le même taux de vrais positifs pour tous les groupes (égalité des chances) ? Atteindre l'un de ces objectifs peut nous éloigner de l'autre. Définir ce qui est "juste" est avant tout une décision de société, pas seulement technique.

Le dilemme performance vs interprétabilité Souvent, les modèles les plus performants (comme les réseaux de neurones profonds) sont des "boîtes noires" : il est très difficile de comprendre exactement comment ils prennent leurs décisions. À l'inverse, les modèles plus simples (comme la régression logistique ou les arbres de décision) sont plus transparents et plus faciles à auditer pour les biais, mais ils peuvent être moins performants. C'est un compromis constant.

[Image des échelles de la justice avec une puce électronique]

QUAND L'ALGORITHME SE TROMPE

Étude de cas : COMPAS et la justice prédictive COMPAS est un outil d'IA utilisé aux États-Unis pour prédire le risque de récidive des accusés. En 2016, une enquête de ProPublica a révélé que l'algorithme était fortement biaisé.

- Le problème : À risque de récidive égal, l'algorithme avait presque deux fois plus de chances de classifier à tort les accusés noirs comme "à haut risque" que les accusés blancs. Inversement, il classifiait souvent à tort les accusés blancs comme "à faible risque".
- La cause : Le modèle n'utilisait pas la variable "race" directement, mais il utilisait des variables (comme le quartier de résidence, les antécédents familiaux) qui étaient fortement corrélées à la race, agissant comme des "proxys".
- Stratégies de mitigation possibles : Cela a déclenché un immense débat sur l'utilisation de tels outils. Les stratégies discutées incluent l'audit régulier des modèles, l'utilisation de définitions d'équité plus robustes, et surtout, la remise en question de l'idée même qu'un algorithme puisse prendre une décision aussi lourde de conséquences.

CHALLENGE POUR VOUS !

Vous êtes maintenant face à un dilemme éthique classique en data science.

Votre mission : Imaginez que vous développez un modèle de scoring de crédit pour une banque. Un manager vous demande de ne pas utiliser la variable 'genre' pour être équitable. Est-ce suffisant ? Quelles autres variables pourraient indirectement recréer un biais de genre ?

Réponse et réflexions : Non, ce n'est absolument pas suffisant. C'est une approche naïve appelée "fairness through unawareness" (l'équité par l'ignorance) qui est souvent inefficace.

Le danger vient des variables proxy. D'autres variables de votre jeu de données peuvent être fortement corrélées avec le genre et permettre au modèle de le "deviner" indirectement. Par exemple :

- L'historique de salaire : En raison de l'écart salarial historique entre les sexes, cette variable peut être un proxy puissant.
- La profession : Certains métiers sont historiquement plus masculins ou féminins.
- L'historique d'achat : Les habitudes de consommation peuvent différer statistiquement.
- Le nombre d'enfants ou la situation maritale : Ces variables peuvent avoir des impacts différents sur la carrière et les finances des hommes et des femmes en raison de normes sociales.

La véritable approche éthique n'est pas de simplement supprimer la variable sensible, mais de mesurer activement si les prédictions de votre modèle ont un impact disproportionné sur différents groupes et de mettre en place des stratégies pour corriger ces biais.

45 Les Systèmes de Recommandation : Comment Netflix sait ce que vous voulez regarder

"Les utilisateurs qui ont aimé cet chapitre ont aussi aimé..." ; "Parce que vous avez écouté cet artiste..." ; "Top 10 des films pour vous". Les systèmes de recommandation sont partout. Ils sont le moteur silencieux de l'économie du web, de Netflix à Amazon en passant par Spotify et YouTube. Ils nous guident dans une mer d'options et aident les entreprises à nous proposer le bon contenu au bon moment. Explorons les deux approches principales pour construire ces systèmes.

Les deux grandes philosophies de la recommandation

1. Le Filtrage Basé sur le Contenu (Content-Based Filtering) C'est l'approche la plus simple à comprendre. La logique est : "Si vous avez aimé un item, nous vous recommanderons des items qui lui ressemblent."

- Comment ça marche ? Le système analyse les caractéristiques (le "contenu") des items que vous avez aimés. Pour des films, ce serait le genre, le réalisateur, les acteurs. Pour des chansons, le tempo, le style musical, l'artiste. Il cherche ensuite dans le catalogue les items qui ont les caractéristiques les plus similaires et vous les propose.

- Analogie : C'est comme un ami qui vous dit : "Tu as adoré 'Le Seigneur des Anneaux' ? Alors tu devrais regarder 'Le Hobbit', c'est aussi de la fantasy épique avec des magiciens et des créatures fantastiques."

2. Le Filtrage Collaboratif (Collaborative Filtering) Cette approche est plus subtile et souvent plus puissante. Sa logique est : "Si des utilisateurs qui ont les mêmes goûts que vous ont aimé un item que vous ne connaissez pas encore, alors il y a de fortes chances que vous l'aimiez aussi."

- Comment ça marche ? Le système n'a pas besoin de comprendre le contenu des items. Il se base uniquement sur le comportement des utilisateurs. Il trouve des utilisateurs qui ont noté, acheté ou regardé les mêmes choses que vous (vos "voisins" ou "jumeaux de goût"), puis il regarde ce que ces utilisateurs ont aimé d'autre et vous le recommande.

- Analogie : C'est un ami qui vous dit : "Je sais que tu aimes les mêmes films que Sarah. Elle vient de voir un film coréen et elle a adoré. Tu devrais y jeter un œil, même si tu ne regardes jamais de films coréens."

Le Problème du Démarrage à Froid (Cold Start Problem) C'est le talon d'Achille des systèmes de recommandation. Que faire quand un nouvel utilisateur s'inscrit ou qu'un nouvel item est ajouté au catalogue ?

- Avec le filtrage collaboratif, c'est un gros problème : un nouvel utilisateur n'a pas d'historique, donc on ne peut pas trouver de "voisins". Un nouvel item n'a pas été noté, donc il ne sera jamais recommandé.

- Avec le filtrage basé sur le contenu, le problème est moins grave. On peut recommander des items populaires à un nouvel utilisateur. Et pour un nouvel item, on peut analyser ses caractéristiques et le recommander immédiatement aux utilisateurs qui ont aimé des items similaires.

CONSTRUIRE UNE RECOMMANDATION SIMPLE

Implémenter un filtrage basé sur le contenu avec Scikit-Learn Nous allons construire un système très simple qui recommande des films en se basant sur leur genre.

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# 1. Créer un jeu de données simple
data = {'titre': ['Film A', 'Film B', 'Film C', 'Film D', 'Film E'],
        'genres': ['Action Aventure', 'Action Thriller', 'Comédie Romance',
'Aventure Science-Fiction', 'Comédie Drame']}
df = pd.DataFrame(data)

# 2. Vectoriser les genres avec TF-IDF
tfidf = TfidfVectorizer()
tfidf_matrix = tfidf.fit_transform(df['genres'])

# 3. Calculer la similarité cosinus entre tous les films
cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)

# 4. Créer une fonction de recommandation
def recommander(titre, cosine_sim=cosine_sim, df=df):
    # Obtenir l'index du film
    idx = df[df['titre'] == titre].index[0]
    # Obtenir les scores de similarité de ce film avec tous les autres
    sim_scores = list(enumerate(cosine_sim[idx]))
    # Trier les films par similarité
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
    # Obtenir les scores des 10 films les plus similaires (en ignorant le
film lui-même)
    sim_scores = sim_scores[1:11]
    # Obtenir les index des films
    movie_indices = [i[0] for i in sim_scores]
    # Retourner les titres
    return df['titre'].iloc[movie_indices]

# Obtenir des recommandations pour le 'Film A'
print(f"Recommandations pour '{df['titre'][0]}':")
print(recommander('Film A'))
```

CHALLENGE POUR VOUS !

Réfléchissez à la nature des deux approches.

1. Le filtrage collaboratif peut-il vous recommander des films de genres très différents de ce que vous regardez habituellement ? Pourquoi ?

2. Et le filtrage basé sur le contenu ?

Réponse :

1. Oui, absolument. C'est la plus grande force du filtrage collaboratif, un phénomène appelé "sérendipité". S'il découvre que de nombreux fans de science-fiction aiment aussi un documentaire animalier en particulier, il pourra vous recommander ce documentaire même si vous n'avez jamais montré d'intérêt pour ce genre. Il se base sur les "ponts" créés par les goûts éclectiques des autres utilisateurs.

2. Très difficilement, voire jamais. Le filtrage basé sur le contenu est par nature conservateur. Si vous n'aimez que des films d'action, il vous recommandera d'autres films d'action. Il est excellent pour approfondir un intérêt existant, mais très mauvais pour en découvrir de nouveaux, car il est limité par les caractéristiques des items que vous avez déjà consommés.

46 Le Gradient Boosting (XGBoost, LightGBM) : l'algorithme qui gagne toutes les compétitions

Les Forêts Aléatoires construisent des centaines d'arbres de décision en parallèle, de manière indépendante, puis font la moyenne de leurs prédictions. Le Gradient Boosting adopte une approche radicalement différente : il construit les arbres en série. Chaque nouvel arbre est entraîné spécifiquement pour corriger les erreurs commises par l'arbre précédent. C'est un processus d'amélioration continue qui, sur les données tabulaires (comme dans un fichier Excel), est la recette secrète du succès dans la plupart des compétitions Kaggle.

L'apprentissage par correction d'erreurs

L'idée de l'apprentissage ensembliste séquentiel (Boosting) Le "boosting" est une technique qui consiste à transformer une collection de "mauvais élèves" (des modèles très simples, souvent de petits arbres de décision, appelés *weak learners*) en un "excellent élève" (*strong learner*). Au lieu de faire voter des experts indépendants, on crée une équipe où chaque nouveau membre se concentre sur les erreurs les plus difficiles que les membres précédents n'ont pas su résoudre.

Comment cela se rapporte à la descente de gradient ? Le "Gradient" dans Gradient Boosting n'est pas un hasard. L'algorithme peut être vu comme une descente de gradient, non pas dans l'espace des paramètres d'un modèle, mais dans l'espace des fonctions possibles. Chaque nouvel arbre est ajouté pour "descendre la pente" de la fonction de perte le plus efficacement possible, en ciblant les résidus (les erreurs) du modèle précédent.

XGBoost, LightGBM, CatBoost : La sainte trinité du Boosting Ce sont trois implémentations ultra-optimisées de l'algorithme de gradient boosting.

- XGBoost (Extreme Gradient Boosting) : Le pionnier et la référence. Extrêmement performant et robuste, il a dominé les compétitions pendant des années.

- LightGBM (Light Gradient Boosting Machine) : Développé par Microsoft, il est souvent beaucoup plus rapide que XGBoost, avec des performances similaires, voire meilleures. Il construit ses arbres d'une manière différente (feuille par feuille), ce qui explique sa vitesse.
- CatBoost (Categorical Boosting) : Développé par Yandex, il excelle dans la gestion native des variables catégorielles, ce qui peut simplifier grandement le travail de pré-traitement.

Le Boosting en action

Utilisons la bibliothèque xgboost sur notre projet de prédiction des prix de l'immobilier (Ames Housing) et comparons sa performance à celle d'une forêt aléatoire.

```
import pandas as pd
import numpy as np
import xgboost as xgb
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# On suppose que vous avez un DataFrame 'df' nettoyé et préparé
# et des variables X et y prêtes pour l'entraînement.
# X, y = ...
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Entraîner un Random Forest pour comparaison
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)
rf_pred = rf.predict(X_test)
rf_rmse = np.sqrt(mean_squared_error(y_test, rf_pred))
print(f"RMSE du Random Forest : {rf_rmse:.2f}")

# Entraîner un modèle XGBoost
xgbr = xgb.XGBRegressor(objective='reg:squarederror', n_estimators=100,
random_state=42)
xgbr.fit(X_train, y_train)
xgb_pred = xgbr.predict(X_test)
xgb_rmse = np.sqrt(mean_squared_error(y_test, xgb_pred))
print(f"RMSE de XGBoost : {xgb_rmse:.2f}")

# Optimisation des hyperparamètres (non montrée ici, mais se ferait avec
GridSearchCV ou RandomizedSearchCV)
```

En général, un modèle XGBoost bien réglé surpassera un Random Forest sur ce type de problème.

Challenge pour vous !

XGBoost, comme les forêts aléatoires, peut calculer l'importance des variables. Mais le fait-il de la même manière ?

Votre mission :

1. Utilisez la fonction intégrée à XGBoost pour tracer l'importance des variables (xgboost.plot_importance(xgbr)).

2. Comparez le classement des variables les plus importantes selon XGBoost avec celui que vous aviez obtenu avec le Random Forest.

3. Sont-elles exactement les mêmes ? Y a-t-il des différences notables dans le top 5 ? Qu'est-ce que cela pourrait impliquer sur la manière dont les deux modèles "voient" les données ?

47 Introduction au Cloud pour la Data Science (AWS, GCP, Azure)

Votre ordinateur portable commence à peiner. L'entraînement de votre dernier modèle prend des heures, et vous n'osez même pas imaginer travailler sur des téraoctets de données. C'est une limite que tout data scientist rencontre. Le Cloud est la solution à ce problème. Il offre une puissance de calcul et de stockage quasi illimitée, disponible à la demande, vous permettant de passer à une échelle supérieure.

La Data Science sans limites

Les 3 principaux fournisseurs (la "cour des grands") Le marché du cloud est dominé par trois géants :

- AWS (Amazon Web Services) : Le leader historique et le plus mature, offrant la plus vaste gamme de services.
- GCP (Google Cloud Platform) : Très fort dans les domaines de la donnée, de l'analyse et du machine learning, profitant de l'expertise de Google.
- Microsoft Azure : Très populaire dans les grandes entreprises déjà intégrées à l'écosystème Microsoft.

Les services clés pour un Data Scientist Chaque fournisseur propose des centaines de services, mais pour un data scientist, quelques catégories sont essentielles :

- Stockage d'objets : Pour stocker d'immenses volumes de données non structurées (images, textes, fichiers CSV...). C'est votre disque dur infini. (Ex: AWS S3, Google Cloud Storage).
- Calcul (Machines Virtuelles) : Pour louer des serveurs à la demande. Vous pouvez choisir une petite machine pour du code simple ou une machine monstrueuse avec plusieurs GPUs pour entraîner un réseau de neurones profond. (Ex: AWS EC2, Google Compute Engine).
- Notebooks Managés : Des plateformes qui vous fournissent un environnement Jupyter Notebook prêt à l'emploi, souvent avec des fonctionnalités de ML intégrées, sans que vous ayez à gérer le serveur sous-jacent. (Ex: Amazon SageMaker, Google Vertex AI).
- Bases de Données Managées : Pour déployer des bases de données SQL ou NoSQL sans avoir à se soucier de l'administration, des sauvegardes ou de la mise à l'échelle. (Ex: Amazon RDS, Google Cloud SQL).

Lancer sa première machine virtuelle

Lançons notre premier serveur sur AWS, le service le plus courant pour commencer.

1. Créer un compte AWS : Rendez-vous sur le site d'AWS et créez un compte. Une offre gratuite ("Free Tier") vous permet d'expérimenter de nombreux services sans frais pendant un an.
2. Aller sur le service EC2 : Dans la console AWS, cherchez le service "EC2" (Elastic Compute Cloud).
3. Lancer une instance : Cliquez sur "Lancer une instance".
4. Choisir une AMI : Une AMI (Amazon Machine Image) est un modèle pour votre serveur. Choisissez une AMI standard comme "Ubuntu Server".
5. Choisir un type d'instance : C'est ici que vous choisissez la puissance de votre machine (CPU, RAM). Pour commencer, une instance du "Free Tier" (comme t2.micro) est parfaite.
6. Configurer la sécurité : Créez une "paire de clés" (.pem). C'est votre mot de passe pour vous connecter à la machine. Téléchargez et gardez ce fichier en sécurité ! Configurez aussi le "groupe de sécurité" pour autoriser le trafic entrant (au minimum, le port 22 pour SSH).
7. Lancer et se connecter : Une fois l'instance lancée, vous obtiendrez une adresse IP publique. Vous pouvez alors vous y connecter depuis votre terminal avec SSH : ssh -i /chemin/vers/votre/cle.pem ubuntu@VOTRE_ADRESSE_IP.
8. Installer l'environnement : Une fois connecté, vous êtes sur un serveur Linux brut. Vous pouvez alors installer Python, pip, Jupyter, et toutes vos librairies préférées avec des commandes comme sudo apt update et sudo apt install python3-pip.

Challenge pour vous !

Le Cloud permet de louer une puissance de calcul inaccessible pour un particulier. Mais combien ça coûte ?

Votre mission :

1. Allez sur la page de tarification "à la demande" des instances EC2 d'AWS.
2. Cherchez le coût horaire d'une instance GPU puissante, par exemple une p3.2xlarge (qui contient un GPU NVIDIA V100).
3. Calculez le coût approximatif pour entraîner un modèle pendant 8 heures sur cette instance.
4. Cherchez le prix d'achat d'une carte graphique NVIDIA V100 neuve.
5. Comparez les deux. Quelle est la conclusion la plus évidente sur l'avantage économique du Cloud pour des besoins ponctuels et intensifs ?

48 Le "MLOps" : Industrialiser le Machine Learning

Mettre un modèle en production, ce n'est que le début du voyage, pas la destination finale. Le MLOps (Machine Learning Operations) est l'ensemble des pratiques, inspirées du DevOps pour le développement logiciel, qui visent à gérer de manière fiable, reproductible et efficace le cycle de vie complet des modèles de machine learning.

DE L'EXPÉRIMENTATION À LA PRODUCTION

Les étapes du cycle de vie MLOps Le MLOps structure le processus en plusieurs étapes interconnectées :

1. Gestion des données : Collecter, nettoyer, versionner et rendre les données accessibles.
2. Entraînement : Entraîner le modèle, souvent de manière automatisée.
3. Validation : Évaluer la performance du modèle sur des critères techniques et business.
4. Déploiement : Mettre le modèle à disposition des utilisateurs via une API ou une application.
5. Monitoring : Surveiller en continu la performance du modèle en production pour détecter toute dégradation.

L'importance du versioning (Données, Code, Modèles) Pour garantir la reproductibilité, il ne suffit pas de versionner le code avec Git. Il faut aussi versionner :

- Les données : Pour savoir exactement avec quelles données un modèle a été entraîné.
- Le modèle lui-même : Pour pouvoir revenir à une version précédente si nécessaire.

Le concept de CI/CD pour le ML Le CI/CD (Continuous Integration / Continuous Deployment) est une pratique qui consiste à automatiser les tests et le déploiement. Appliqué au ML, cela signifie créer des "pipelines" automatisés qui peuvent, par exemple, ré-entraîner et re-déployer un modèle automatiquement dès que de nouvelles données sont disponibles.

DES OUTILS POUR L'INDUSTRIALISATION

DVC (Data Version Control) DVC est un outil qui fonctionne avec Git pour versionner des jeux de données volumineux sans alourdir le dépôt Git. Il stocke les "méta-informations" sur les données dans Git, tandis que les données elles-mêmes sont stockées sur un service de stockage cloud (comme AWS S3).

MLflow : Suivre ses expériences MLflow est une plateforme open-source pour gérer le cycle de vie du ML. L'une de ses fonctionnalités clés est le suivi d'expériences. Il permet d'enregistrer pour chaque entraînement :

- Les paramètres utilisés.
- Le code (via un commit Git).
- Les métriques de performance obtenues.
- Le modèle entraîné lui-même.

Cela crée un journal de bord de toutes vos expériences, ce qui est inestimable pour comparer les modèles et reproduire les résultats.

```
import mlflow
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
```

```
# Démarrer une nouvelle expérience MLflow
with mlflow.start_run():
    # Enregistrer des paramètres
    mlflow.log_param("n_estimators", 150)
    mlflow.log_param("max_depth", 5)

    # Entraîner un modèle (exemple)
    rf = RandomForestRegressor(n_estimators=150, max_depth=5)
    # rf.fit(X_train, y_train)
    # predictions = rf.predict(X_test)
    # rmse = mean_squared_error(y_test, predictions, squared=False)

    # Enregistrer une métrique
    # mlflow.log_metric("rmse", rmse)

    # Enregistrer le modèle
    # mlflow.sklearn.log_model(rf, "random-forest-model")

print("Expérience enregistrée avec MLflow !")
```

CHALLENGE POUR VOUS !

Votre modèle de prédiction des prix de l'immobilier est en production depuis 6 mois et fonctionne bien. Soudain, vous remarquez que ses performances chutent drastiquement. C'est un phénomène appelé "model drift" (dérive du modèle).

Votre mission :

1. Quelles pourraient être les causes de cette dérive ?
2. Comment mettriez-vous en place un système de monitoring simple pour détecter ce problème automatiquement, avant que les utilisateurs ne s'en plaignent ?

Réponse :

1. Causes possibles :
    - Dérive des données (*Data Drift*) : Les caractéristiques des nouvelles maisons qui arrivent sur le marché ont changé (ex: un nouveau quartier se développe, les taux d'intérêt ont changé, les styles architecturaux évoluent). Les nouvelles données ne ressemblent plus aux données sur lesquelles le modèle a été entraîné.
    - Dérive du concept (*Concept Drift*) : La relation entre les caractéristiques et le prix a changé. Par exemple, après une nouvelle réglementation écologique, les maisons avec une bonne isolation (une variable qui était peut-être peu importante avant) deviennent soudainement beaucoup plus chères.
2. Mise en place du monitoring :
    - Monitoring des données d'entrée : Mettre en place des alertes qui se déclenchent si la distribution statistique des nouvelles données en entrée (ex: la surface moyenne, le

nombre de chambres) s'écarte de manière significative de la distribution des données d'entraînement.

- Monitoring des performances du modèle : Si possible, comparer régulièrement les prédictions du modèle avec les prix de vente réels (une fois qu'ils sont connus). Mettre en place une alerte si le RMSE ou une autre métrique de performance dépasse un certain seuil.

49 Comment lire un papier de recherche en IA sans avoir mal à la tête

Le domaine de l'intelligence artificielle évolue à une vitesse fulgurante. Les dernières avancées ne sont pas dans des livres ou des livre, mais publiées quasi quotidiennement dans des "papiers de recherche" sur des plateformes comme ArXiv. Apprendre à lire, comprendre et critiquer ces papiers est une compétence essentielle pour rester à la pointe et ne pas se laisser distancer.

La stratégie du lecteur efficace

La structure d'un papier de recherche Un papier scientifique suit presque toujours la même structure. La comprendre, c'est avoir une carte pour ne pas se perdre.

- Abstract (Résumé) : Un paragraphe qui résume tout le papier : le problème, la méthode, les résultats et la conclusion. C'est la bande-annonce.
- Introduction : Met en contexte le problème, explique son importance et énonce les contributions des auteurs.
- Related Work (Travaux connexes) : Situe le travail des auteurs par rapport à ce qui a déjà été fait dans le domaine.
- Method (Méthodologie) : Le cœur technique du papier. C'est ici que les auteurs décrivent en détail leur nouvelle architecture ou leur nouvel algorithme.
- Results (Résultats) : Présente les résultats des expériences menées pour prouver que leur méthode fonctionne, souvent sous forme de tableaux comparant leur score à celui d'autres méthodes.
- Conclusion : Résume les contributions, discute des limites du travail et propose des pistes pour de futures recherches.

La stratégie de lecture en 3 passes N'essayez jamais de lire un papier de manière linéaire du début à la fin. Vous vous noierez dans les détails. Adoptez plutôt une approche stratégique :

1. Première passe (5-10 minutes) : Le survol. Lisez le titre, l'abstract et la conclusion. Regardez rapidement les figures et les tableaux. Le but est de comprendre l'idée générale et de décider si ce papier vaut la peine d'être lu plus en détail.
2. Deuxième passe (environ 1 heure) : La compréhension du contenu. Lisez le papier plus attentivement, mais sans vous attarder sur les détails mathématiques complexes. Concentrez-vous sur la compréhension des graphiques et des tableaux. Prenez des notes en marge. Le but est de saisir ce que les auteurs ont fait.

3. Troisième passe (plusieurs heures) : L'analyse critique. C'est seulement à cette étape que vous essayez de recréer le travail dans votre tête. Mettez en question les hypothèses des auteurs, réfléchissez aux alternatives et imaginez comment vous pourriez améliorer leur travail.

METTRE LA MÉTHODE À L'ÉPREUVE

Appliquer la méthode sur un papier fondateur Un excellent papier pour s'entraîner est celui d'AlexNet ("ImageNet Classification with Deep Convolutional Neural Networks"). C'est le papier qui, en 2012, a lancé la révolution du Deep Learning en vision par ordinateur. Il est relativement accessible et son impact a été monumental. Essayez d'appliquer la méthode des 3 passes sur celui-ci.

Trouver des papiers pertinents Naviguer sur ArXiv peut être intimidant. Des outils comme ArXiv Sanity Preserver ou des sites comme Papers with Code peuvent vous aider à trouver les papiers les plus populaires, les plus cités et à voir leur code d'implémentation.

CHALLENGE POUR VOUS !

Il est temps de faire votre première passe sur l'un des papiers les plus importants de la dernière décennie.

Votre mission :

1. Trouvez le papier original "Attention Is All You Need" qui a introduit l'architecture Transformer.
2. Appliquez uniquement la première passe de la stratégie de lecture : lisez le titre, l'abstract, et regardez attentivement les figures (notamment le schéma de l'architecture Transformer).
3. Après ces 10 minutes, seriez-vous capable d'expliquer l'idée principale du papier à un collègue en 30 secondes ?

50 LES CARRIÈRES EN DATA : ANALYST, SCIENTIST, ENGINEER, QUELLE EST LA DIFFÉRENCE ?

Le terme "Data" est partout, mais les métiers qu'il recouvre sont très différents. Analyst, Scientist, Engineer... Il est facile de se perdre dans ce jargon. Clarifions ces rôles, leurs missions et leurs outils pour que vous puissiez trouver la carrière qui vous correspond le mieux et dans laquelle vous pourrez vous épanouir.

QUI FAIT QUOI DANS LE MONDE DE LA DONNÉE ?

Le Data Analyst : Le conteur du passé Le Data Analyst regarde les données historiques pour répondre à la question : "Que s'est-il passé ?". Il transforme des données brutes en informations compréhensibles, souvent sous la forme de tableaux de bord et de rapports. Son but est d'identifier des tendances, de suivre des indicateurs de performance (KPIs) et de communiquer ses découvertes aux équipes business pour les aider à prendre de meilleures décisions.

- Outils de prédilection : SQL, Excel, Tableau, Power BI.

Le Data Scientist : L'architecte du futur Le Data Scientist va plus loin. Il utilise les statistiques et le machine learning pour répondre aux questions : "Pourquoi cela s'est-il passé ?" et surtout "Que va-t-il se passer ?". Il construit des modèles prédictifs, mène des expérimentations (comme les tests A/B) et cherche à découvrir des relations de cause à effet complexes dans les données.

- Outils de prédilection : Python/R, Pandas, Scikit-Learn, TensorFlow, Notebooks Jupyter.

Le Data Engineer : Le constructeur des autoroutes de la donnée Rien ne serait possible sans le Data Engineer. C'est l'architecte de l'infrastructure. Il conçoit, construit et maintient les "pipelines" qui collectent, stockent et transforment des volumes massifs de données. Son objectif est de s'assurer que les données sont fiables, propres et facilement accessibles pour les analystes et les scientists.

- Outils de prédilection : SQL, Python, Spark, Kafka, Airflow, entrepôts de données (BigQuery, Snowflake), plateformes cloud (AWS, GCP, Azure).

Le Machine Learning Engineer : Le mécanicien de l'IA Le ML Engineer est le pont entre la data science et l'ingénierie logicielle. Il prend les modèles créés par les data scientists et les met en production de manière robuste, scalable et maintenable. Il s'occupe du déploiement, du monitoring et de l'automatisation du cycle de vie des modèles (MLOps).

- Outils de prédilection : Python, Docker, Kubernetes, CI/CD, Flask/FastAPI, plateformes cloud.

Les compétences et le marché du travail

Un diagramme de Venn des compétences Ces rôles ne sont pas totalement cloisonnés. Ils partagent un socle commun de compétences, notamment en programmation et en compréhension des données.

[Image d'un diagramme de Venn des compétences en data]

Analyser des offres d'emploi Pour bien comprendre la réalité du marché, le mieux est de lire des offres d'emploi. Voici les mots-clés que vous y trouverez souvent :

- Data Analyst : "Tableaux de bord", "reporting", "KPIs", "visualisation", "SQL", "Tableau/Power BI", "communication".
- Data Scientist : "Machine Learning", "modélisation prédictive", "statistiques", "Python/R", "Scikit-learn", "A/B testing", "deep learning".
- Data Engineer : "Pipeline de données", "ETL/ELT", "Spark", "SQL", "bases de données", "AWS/GCP/Azure", "Big Data".
- Machine Learning Engineer : "Déploiement", "production", "MLOps", "Docker", "API", "Flask", "CI/CD", "monitoring".

Challenge pour vous !

Maintenant, à vous de vous situer.

1. Vous aimez passer du temps à nettoyer des données brutes, à construire des flux de traitement efficaces (pipelines ETL) et à optimiser des requêtes SQL complexes pour qu'elles s'exécutent

en quelques secondes plutôt qu'en quelques heures. Vers quel rôle devriez-vous vous orienter ?

2. Et si, au contraire, vous préférez dialoguer avec les équipes marketing ou produit, comprendre leurs besoins, créer des tableaux de bord interactifs pour suivre leurs objectifs et présenter vos découvertes de manière claire et percutante ?

Réponses :

1. Vous avez l'âme d'un Data Engineer. Votre satisfaction vient de la construction d'une infrastructure de données solide et performante.

2. Vous êtes sans doute un Data Analyst dans l'âme. Votre force réside dans la traduction des données en insights actionnables et dans la communication.

51 Préparer un entretien technique en Data Science : Les questions pièges

Vous avez les compétences, vous avez construit un portfolio, maintenant il faut le prouver en direct. L'entretien technique est l'épreuve finale. S'y préparer ne consiste pas seulement à réviser, mais à apprendre à structurer sa pensée, à communiquer clairement et à anticiper les questions les plus courantes et les pièges à éviter.

Comprendre le terrain de jeu

Les différents types d'entretiens Un processus de recrutement en data science est souvent une série d'entretiens, chacun testant une compétence différente :

- L'entretien SQL : Teste votre capacité à extraire et manipuler des données. Attendez-vous à des JOIN, des GROUP BY et des fonctions de fenêtrage.

- L'entretien de statistiques et probabilités : Évalue votre intuition des données. On vous posera des questions sur les tests A/B, la p-value, les distributions, etc.

- L'entretien de modélisation : Teste votre compréhension des algorithmes. "Expliquez le Random Forest", "Quelles sont les hypothèses de la régression linéaire ?", "Comment géreriez-vous l'overfitting ?".

- Le cas pratique (Case Study) : Le plus important. On vous donne un problème business ouvert ("Comment augmenteriez-vous l'engagement sur notre app ?") et on évalue votre capacité à le structurer.

- L'entretien de "culture fit" : Évalue votre personnalité, votre motivation et votre capacité à travailler en équipe.

La structure pour résoudre un cas pratique Face à une question ouverte, ne plongez pas tête baissée dans la technique. Respirez, et suivez une structure :

1. Clarifier : Assurez-vous de bien comprendre la question et l'objectif business. Posez des questions ! "Quel est l'objectif principal ?", "Comment mesure-t-on le succès ?".

2. Faire des hypothèses : Énoncez les hypothèses que vous faites. "Je suppose que nous avons accès à telles données...", "Je suppose que l'objectif à court terme est plus important que celui à long terme...".
3. Proposer un plan : Décrivez les grandes étapes de votre analyse. "D'abord, je regarderais les données disponibles. Ensuite, je définirais une métrique clé. Puis, j'explorerais les données pour trouver des pistes. Enfin, j'envisagerais quelques modèles simples."
4. Analyser : Déroulez votre plan.
5. Conclure : Résumez vos découvertes et vos recommandations, en n'oubliant pas de mentionner les limites de votre approche et les prochaines étapes possibles.

QUELQUES QUESTIONS TYPES

SQL :

- Quelle est la différence entre JOIN et LEFT JOIN ?
- Comment trouver la deuxième plus haute valeur d'une colonne ?
- Expliquez ce que fait une clause GROUP BY avec HAVING.

Statistiques / Probabilités :

- Expliquez la p-value à quelqu'un qui n'est pas statisticien.
- Quand utiliser la moyenne vs la médiane ?
- Vous lancez un test A/B. Comment déterminez-vous la taille de l'échantillon nécessaire ?

Modélisation :

- Quelle est la différence entre la régularisation L1 et L2 ?
- Comment géreriez-vous un jeu de données très déséquilibré ?
- Expliquez la différence entre le Bagging (Random Forest) et le Boosting (XGBoost).

CHALLENGE POUR VOUS !

Vous êtes en entretien pour un poste de Data Scientist chez un grand site e-commerce. L'interviewer vous demande :

"Nous suspectons que certains vendeurs créent de faux avis positifs pour leurs produits. Comment concevriez-vous un système pour détecter ces faux avis ?"

Structurez votre réponse comme si vous étiez en entretien, en suivant les étapes (Clarifier, Hypothèses, Plan, etc.).

Exemple de structure de réponse :

- Clarification : "Excellente question. Pour être sûr de bien comprendre, l'objectif est-il de détecter les faux avis individuels, ou de repérer les vendeurs qui trichent de manière systématique ? La détection en temps réel est-elle nécessaire ?"
- Hypothèses : "Je suppose que nous avons accès à l'historique des avis (texte, note, date, ID utilisateur, ID produit) et aux informations sur les utilisateurs et les produits."
- Plan / Features possibles : "Je commencerais par du feature engineering. Les faux avis ont souvent des caractéristiques spécifiques. Je créerais des variables comme :
    - Features textuelles : Longueur de l'avis, utilisation de superlatifs, similarité avec d'autres avis pour le même produit.
    - Features comportementales : L'utilisateur a-t-il noté d'autres produits ? Est-ce son premier avis ? À quelle vitesse a-t-il laissé l'avis après l'achat ?
    - Features de graphe : Y a-t-il des groupes d'utilisateurs qui notent systématiquement les mêmes produits ?"
- Modélisation : "Avec ces features, je pourrais envisager deux approches :
    - Non supervisée : Utiliser des algorithmes de détection d'anomalies pour trouver les avis qui s'écartent le plus de la norme.
    - Supervisée : Si nous avons des exemples d'avis déjà identifiés comme faux, je pourrais entraîner un classifieur (comme un Random Forest) pour généraliser la détection."
- Conclusion et limites : "Ce système permettrait de flagger les avis suspects pour une vérification humaine. La principale difficulté sera d'éviter les faux positifs (ne pas accuser à tort un vrai client)."

52 Un parlivre de A à Z : Bilan et prochaines étapes pour continuer à apprendre

Félicitations ! Vous avez parcouru un chemin immense, depuis les bases de Python jusqu'au déploiement de modèles de deep learning. C'est le moment de regarder en arrière, de consolider vos acquis et, surtout, de tracer la route pour la suite. Car en data science, plus que dans tout autre domaine, l'apprentissage ne s'arrête jamais.

Consolider et se spécialiser

Résumé des compétences clés acquises Au livre de cette série, vous avez construit une base solide :

- Programmation et Outils : Vous maîtrisez Python, Pandas, NumPy et les outils de visualisation.
- Statistiques et Probabilités : Vous comprenez les concepts de base qui sous-tendent la data science, des distributions aux tests A/B.
- Machine Learning : Vous savez entraîner, évaluer et interpréter une large gamme de modèles, des régressions aux réseaux de neurones.

- Workflow complet : Vous avez appris à gérer un projet de A à Z, de la collecte des données (SQL, API, Scraping) à leur déploiement (Flask, Docker) en passant par les bonnes pratiques (Git, MLOps).

L'importance de la spécialisation Maintenant que vous avez une vision d'ensemble, il est temps de penser à vous spécialiser. Personne ne peut être expert en tout. Souhaitez-vous vous concentrer sur :

- Le NLP : Travailler avec le langage, les chatbots, la traduction.
- La Computer Vision : Analyser des images et des vidéos.
- Les Séries Temporelles : Faire de la prévision financière ou de la maintenance prédictive.
- Le MLOps : Vous concentrer sur l'industrialisation et la mise en production des modèles.

Le concept d'apprentissage en T C'est un excellent modèle mental pour votre carrière. Il consiste à avoir :

- Une barre horizontale large de connaissances générales sur tous les aspects de la data science (ce que cette série vous a donné).
- Une barre verticale profonde d'expertise dans un ou deux domaines de spécialisation que vous choisirez.

[Image d'une lettre T représentant les compétences en data science]

FEUILLE DE ROUTE POUR L'APPRENTISSAGE CONTINU

Comment continuer à progresser ?

1. Suivre l'actualité : Lisez des blogs et des newsletters de référence (ex: "Towards Data Science", "The Batch" d'Andrew Ng) pour rester au courant des dernières tendances.
2. Contribuer à l'Open-Source : Trouvez un projet sur GitHub qui vous intéresse (Scikit-learn, Pandas...) et commencez par corriger de petites erreurs dans la documentation. C'est une excellente façon d'apprendre et de vous faire connaître.
3. Participer à des compétitions Kaggle : Maintenant que vous maîtrisez les bases, essayez des compétitions plus avancées. Lisez les notebooks des gagnants, c'est une mine d'or d'apprentissage.
4. Obtenir des certifications Cloud : Une certification d'un grand fournisseur (AWS, GCP, Azure) peut être un vrai plus sur votre CV, car elle prouve votre maîtrise des outils de production.

CHALLENGE POUR VOUS !

Le meilleur moyen d'apprendre est de faire. Il est temps de lancer votre propre projet, celui qui vous motivera et que vous serez fier de mettre en avant.

Votre mission : Définissez votre propre projet "capstone".

1. Trouvez une question qui vous passionne réellement. (Ex: "Peut-on prédire le succès d'une chanson sur Spotify ?", "Quels sont les facteurs qui influencent les retards de vols dans mon aéroport local ?").
2. Cherchez et collectez les données (via des APIs, du web scraping ou des datasets publics).
3. Appliquez tout ce que vous avez appris pour nettoyer les données, les explorer, construire un modèle et communiquer vos résultats.
4. Publiez-le sur votre GitHub avec un README impeccable.

C'est le début de votre nouvelle aventure en tant que data scientist autonome et compétent. Le voyage ne fait que commencer !

CONCLUSION

Félicitations ! Vous êtes arrivé au bout de ce voyage initiatique au cœur de la data science. En parcourant ces chapitres, vous avez accompli bien plus que l'apprentissage de quelques lignes de code. Vous avez assemblé une boîte à outils complète qui vous permet de transformer une simple question en un projet de machine learning concret et tangible.

Vous avez appris à :

- Parler le langage des données avec Python, NumPy et Pandas.
- Révéler les histoires cachées grâce à la visualisation avec Matplotlib et Seaborn.
- Construire des modèles prédictifs capables d'apprendre, des régressions simples aux puissantes forêts aléatoires et même aux réseaux de neurones.
- Maîtriser le workflow complet, de la collecte des données (SQL, API) à leur déploiement (Flask, Docker), en passant par les bonnes pratiques de collaboration (Git, GitHub).

Plus important encore, vous avez développé un état d'esprit. Vous savez maintenant qu'une affirmation doit être soutenue par des preuves, qu'un modèle doit être évalué avec rigueur et que la technologie que nous créons a un impact réel qui nous impose une responsabilité éthique.

Ce livre n'est pas une fin, mais un point de départ. Le monde de la data science est en constante évolution, et votre curiosité sera votre meilleur guide pour la suite. Continuez à explorer, à participer à des compétitions sur Kaggle, à lire, et surtout, à construire vos propres projets. L'aventure ne fait que commencer !