

Python pour le traitement des données

Redha Moulla

Paris, 8 - 12 septembre 2025

Plan de la formation

- Qu'est-ce que l'analyse des données?
- Ecosystème Python
- Traitement des données tabulaires
- Visualisation des données
- Modélisation des données

Qu'est-ce que l'analyse des données ?

Qu'est-ce que l'analyse des données ?

- L'analyse des données consiste à examiner des ensembles de données pour en tirer des conclusions.
- Elle permet de transformer des données brutes en informations exploitables pour la prise de décision.
- Les principales étapes de l'analyse des données incluent :
 - **Collection des Données** : Collecter des données à partir de différentes sources (bases de données, API, fichiers CSV, etc.).
 - **Nettoyage des Données** : Identifier et corriger les erreurs, gérer les valeurs manquantes.
 - **Exploration des Données** : Comprendre la distribution, les tendances, et les relations dans les données.
 - **Visualisation** : Utiliser des graphiques pour représenter les données de manière visuelle.
- Objectif : Comprendre le comportement ou les tendances pour mieux orienter la stratégie.

Types d'analyses de données

- Analyse descriptive :
 - Décrit ce qui s'est passé ou ce qui se passe actuellement.
 - Utilise des statistiques simples (moyennes, pourcentages, histogrammes).
 - Exemple : Analyse du chiffre d'affaires mensuel.
- Analyse diagnostique :
 - Explore les causes sous-jacentes des événements ou des comportements observés.
 - Techniques : Corrélations, régressions.
 - Exemple : Pourquoi les ventes ont-elles diminué le mois dernier ?
- Analyse prédictive :
 - Prédit des événements futurs à partir de données historiques.
 - Utilise des algorithmes de machine learning.
 - Exemple : Prédire le taux de désabonnement des clients.
- Analyse prescriptive :
 - Recommande des actions à partir des résultats de l'analyse descriptive et prédictive.
 - Exemple : Que devrions-nous faire pour réduire le taux de désabonnement ?

Différentes approches pour analyser les données

- **Visualisation des données :**

- Utilisation de graphiques pour mieux comprendre les tendances, les relations, et les anomalies
- Outils : histogrammes, diagrammes de dispersion (scatter plots), box plots
- Objectif : communiquer rapidement des informations clés de manière visuelle

- **Statistiques descriptives :**

- Résumer les caractéristiques des données à l'aide de mesures telles que la moyenne, la médiane, l'écart-type
- Utilisée pour obtenir une première compréhension des données
- Objectif : donner une vue d'ensemble des tendances globales

- **Modélisation statistique :**

- Créer des modèles mathématiques pour représenter des relations entre les variables
- Types : régression linéaire, analyse factorielle
- Objectif : estimer ou prédire une valeur, ou comprendre l'impact des variables

- **Apprentissage automatique (machine learning) :**

- Utilisation d'algorithmes pour faire des prédictions ou classer des données sur la base d'exemples passés
- Types : apprentissage supervisé (régression, classification), apprentissage non supervisé (clustering)

Ecosystème Python

L'écosystème Python pour l'analyse de données 1/3

Python est un langage polyvalent, open source, et largement adopté pour l'analyse de données. Il offre un large éventail de bibliothèques et d'outils spécialement conçus pour la manipulation et l'analyse des données.

Avantages de Python pour la data science :

- Syntaxe simple et facile à apprendre, rendant les analyses complexes plus accessibles.
- Grande communauté et support actif, avec des mises à jour régulières et de nombreuses ressources d'apprentissage.
- Intégration facile avec des outils de visualisation, des bases de données, et des plateformes cloud.

Ecosystème Python pour l'analyse des données 2/3

- **Jupyter Notebook :**

- Environnement interactif permettant de combiner code, visualisation et texte explicatif.
- Idéal pour tester des concepts, visualiser des résultats, et documenter des processus d'analyse.

- **NumPy :**

- Fournit des structures de données efficaces pour les calculs numériques (tableaux n-dimensionnels).
- Utilisée pour les opérations mathématiques de base et avancées.

- **Pandas :**

- Bibliothèque principale pour la manipulation et l'analyse de données tabulaires.
- Utilisée pour l'importation, le nettoyage, la transformation des données, et la manipulation de DataFrames.

Ecosystème Python pour l'analyse des données 3/3

- **Matplotlib et Seaborn :**

- **Matplotlib :** Outil fondamental pour créer des graphiques de base (lignes, barres, histogrammes).
- **Seaborn :** Basé sur Matplotlib, permet de créer des visualisations plus avancées et d'explorer des relations statistiques.

- **SciPy :**

- Extension de NumPy, offrant des fonctionnalités supplémentaires pour l'analyse scientifique.
- Utilisée pour des opérations avancées, notamment l'intégration, l'optimisation, et les statistiques.

- **statsmodels :**

- Fournit des classes et fonctions pour l'estimation de modèles statistiques et la réalisation de tests.
- Utilisée pour les analyses statistiques avancées, telles que les régressions linéaires, les séries temporelles, et les tests d'hypothèse.

- **scikit-learn :**

- Bibliothèque dédiée au machine learning, offrant de nombreux algorithmes de classification, régression, et clustering.
- Utilisée pour la création, l'entraînement, et l'évaluation de modèles prédictifs.

Traitement des données tabulaires

Introduction à NumPy 1/2

- **NumPy (Numerical Python)** est une bibliothèque fondamentale pour les calculs scientifiques en Python.
- **Origines et Objectifs :**
 - Créée pour fournir un support de calcul efficace en Python.
 - Inspirée par les bibliothèques de calcul scientifique en C et Fortran, afin d'allier la simplicité de Python à la performance des langages bas-niveau.
- **Structure de NumPy : ndarray (n-dimensional array) :**
 - L'élément central de NumPy est l'objet **ndarray**, un tableau multidimensionnel permettant de stocker des données de manière contiguë.
 - Un ndarray est fixé en taille, ce qui le rend plus performant qu'une liste Python traditionnelle.
 - L'array peut avoir des dimensions différentes (1D, 2D, 3D, etc.), permettant de représenter des scalaires, vecteurs, matrices et tenseurs.
- **Différences avec les listes Python :**
 - **Homogénéité :** Les éléments d'un ndarray sont du même type (dtype), contrairement aux listes Python, ce qui facilite le traitement et réduit la surcharge mémoire.
 - **Performance :** Les opérations sur les arrays sont optimisées pour être vectorisées, supprimant ainsi la nécessité des boucles explicites et réduisant le temps d'exécution.

Introduction à NumPy 2/2

Pourquoi NumPy est-il performant ?

- **Implémentation en C :**

- Les opérations NumPy sont implémentées en C, permettant d'éviter les lenteurs de l'interprétation Python.
- Le code est compilé et les calculs sont effectués directement en langage machine, ce qui augmente la rapidité.

- **Optimisation mémoire :**

- Les données dans un `ndarray` sont stockées de manière **contiguë en mémoire**, ce qui minimise l'overhead et améliore l'accès aux données.
- **Strides** : La notion de strides dans NumPy permet un accès rapide aux éléments en précisant le nombre de pas à effectuer en mémoire pour passer d'un élément à l'autre.

- **Vectorisation des opérations :**

- NumPy permet d'effectuer des **opérations vectorisées**, c'est-à-dire des opérations sur des ensembles d'éléments simultanément sans boucles explicites.
- Utilisation de **SIMD (Single Instruction, Multiple Data)** : Les CPU modernes peuvent traiter plusieurs éléments à la fois, et NumPy exploite cette capacité.

Création d'arrays NumPy et types de données

- **Créer un array :**

- `np.array([1, 2, 3])` : Créer un array à partir d'une liste.
- `np.zeros((3, 4))` : Créer un array de zéros de dimension 3x4.
- `np.linspace(0, 1, 5)` : Générer 5 valeurs uniformément espacées entre 0 et 1.

- **Types de données (dtype) :**

- NumPy offre une variété de types : `int32`, `float64`, `bool`, etc.
- Vous pouvez définir explicitement le type de données : `np.array([1, 2, 3], dtype='float64')`

- **Conversion de type :**

- `array.astype('int')` : Convertir un array en type entier.

Propriétés des arrays et manipulation de forme

- **Propriétés :**

- `shape` : Renvoie la dimension (`array.shape`).
- `size` : Nombre total d'éléments (`array.size`).
- `ndim` : Nombre de dimensions (`array.ndim`).

- **Changement de forme (`reshape()`) :**

- Exemple : Transformer un array 1D en 2D.
- `array = np.arange(12).reshape(3, 4)` : Crée un tableau 3x4 à partir d'une séquence de 12 éléments.

- **Flattening :**

- Transforme un tableau multidimensionnel en une dimension avec `array.flatten()`.

Indexation, slicing et conditions logiques

- **Indexation :**

- Accéder aux éléments par position : `array[2, 3]` pour une matrice 2D.

- **Slicing :**

- Extraire une partie d'un array : `array[1:4]` extrait les indices 1 à 3.
- Slicing multidimensionnel : `array[:, 1:3]` pour extraire certaines colonnes.

- **Indexation conditionnelle :**

- `array[array > 5]` : Retourne les éléments de l'array supérieurs à 5.
- Utilisé pour filtrer des données.

Opérations mathématiques avec NumPy

- **Opérations élément par élément :**

- Les opérations de base comme l'addition, la soustraction, la multiplication et la division sont réalisées de manière vectorisée, ce qui signifie qu'elles s'appliquent sur chaque élément des arrays sans avoir besoin de boucles explicites.
- Exemple : Si `array1 = np.array([1, 2, 3])` et `array2 = np.array([4, 5, 6])`, alors `array1 + array2` renvoie `[5, 7, 9]`.

- **Fonctions mathématiques appliquées à des arrays :**

- NumPy propose de nombreuses fonctions mathématiques prêtes à l'emploi, permettant d'opérer sur des arrays de manière rapide et vectorisée :
 - `np.sqrt(array)` : Racine carrée de chaque élément.
 - `np.log(array)` : Logarithme naturel de chaque élément.
 - `np.exp(array)` : Exponentielle de chaque élément.
 - `np.power(array, n)` : Élève chaque élément à la puissance `n`.

- **Fonctions trigonométriques et hyperboliques :**

- `np.sin(array)`, `np.cos(array)`, `np.tan(array)` : Calcul des fonctions trigonométriques pour chaque élément.
- `np.sinh(array)`, `np.cosh(array)` : Fonctions hyperboliques.

Produits matriciels et opérations de réduction

- **Produit matriciel et opérations de l'algèbre linéaire :**

- **Produit matriciel** : Utiliser `np.dot(array1, array2)` ou l'opérateur `array1 @ array2` pour le produit de matrices.
- **Produit matriciel élément par élément** (Hadamard) : Utiliser `array1 * array2` pour obtenir le produit élément par élément.
- **Exemple** : Multiplier deux matrices de même taille pour appliquer un filtre pixel par pixel dans le cadre d'un traitement d'image.

- **Opérations de réduction (réduction des dimensions) :**

- `np.sum(array, axis=0)` : Somme des éléments le long de l'axe spécifié (exemple : somme de chaque colonne).
- `np.prod(array, axis=1)` : Produit des éléments le long de l'axe 1 (exemple : produit des éléments de chaque ligne).
- `np.cumsum(array)` : Calcul de la somme cumulée.
- `np.cumprod(array)` : Produit cumulatif des éléments.

Fonctions d'agrégation

- NumPy propose également des fonctions qui permettent de calculer des agrégats sur les objets array :
 - `np.sum(array)` : Somme de tous les éléments.
 - `np.mean(array)` : Moyenne.
 - `np.std(array)` : Écart-type.
 - `np.max(array)` et `np.min(array)` : Maximum et minimum.
- Exemple :
 - `array = np.array([1, 2, 3, 4])`
 - `np.sum(array)` renvoie 10

Manipulation des arrays : concaténation et division

- **Concaténation :**

- Combiner plusieurs arrays en un seul.
- Utiliser `np.concatenate((array1, array2), axis=0)` pour concaténer selon une dimension donnée (lignes ou colonnes).

- **Empilement :**

- `np.vstack((array1, array2))` : Empiler verticalement (lignes).
- `np.hstack((array1, array2))` : Empiler horizontalement (colonnes).

- **Division :**

- `np.split(array, indices)` : Diviser un array en sous-arrays selon des indices spécifiques.
- `np.hsplit(array, 3)` : Diviser horizontalement en trois sous-arrays égaux.

Fonctions logiques et conditions dans NumPy

- **Conditions logiques :**

- Utiliser des opérateurs logiques pour filtrer les données :
- `array[array > 10]` renvoie les éléments supérieurs à 10.

- **Fonctions logiques :**

- `np.where(condition, x, y)` : Renvoie `x` lorsque la condition est vraie, sinon `y`.
- `np.any(array > 0)` : Vérifie si au moins un élément est supérieur à zéro.
- `np.all(array > 0)` : Vérifie si tous les éléments sont supérieurs à zéro.

- **Exemple :**

- Supposons un array :
- `array = np.array([3, 7, 12, -5, 8, 0, -1, 15])`
- Utilisation de `np.where` :
- `result = np.where(array > 5, 'grand', 'petit')`
- **Résultat** : `['petit', 'grand', 'grand', 'petit', 'grand', 'petit', 'petit', 'grand']`
- **Interprétation** :
- Les éléments de l'array sont classifiés comme 'grand' s'ils sont supérieurs à 5, sinon ils sont classés comme 'petit'.

Broadcasting dans NumPy : concept et règles

- **Concept de Broadcasting** : Le broadcasting permet d'appliquer des opérations arithmétiques sur des arrays de tailles différentes sans avoir besoin de les étendre explicitement à la même forme. NumPy "diffuse" ou "étend" les dimensions plus petites pour qu'elles correspondent aux dimensions plus grandes, rendant ainsi l'opération possible.
- **Pourquoi le Broadcasting est-il utile ?** :
 - Simplifie le code en évitant les boucles explicites pour les opérations sur des arrays de tailles différentes.
 - Améliore la **performance**, car les opérations sont optimisées en interne et bénéficient de la vectorisation.
- **Règles du Broadcasting** : Pour que le broadcasting soit possible, les deux dimensions doivent respecter les conditions suivantes :
 - Les dimensions doivent être **égales** ou
 - L'une des dimensions doit être égale à **1**.

Si une dimension est égale à 1, NumPy réplique implicitement cette dimension pour qu'elle corresponde à celle de l'autre array.

- **Exemple de règle** : `A.shape = (3, 4)` et `b.shape = (4,)` : `b` sera étendu implicitement à la forme `(3, 4)`.

Broadcasting dans NumPy : exemples

● Exemple 1 : Addition d'un vecteur à une matrice :

- Supposons une matrice A de dimension (3, 4) et un vecteur b de dimension (4,).
- NumPy va étendre implicitement le vecteur b en une matrice de dimension (3, 4) pour que l'opération $A + b$ soit valide.
- Code :

```
A = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]) b = np.  
0, 1, 0]) result = A + b  
Résultat : [[ 2, 2, 4, 4], [ 6, 6, 8, 8], [10, 10, 12, 12]]
```

● Exemple 2 : Multiplication d'un array par un scalaire :

- Lorsqu'un array est multiplié par un scalaire, chaque élément de l'array est multiplié par ce scalaire.
- Le broadcasting permet de "diffuser" le scalaire à chaque élément de l'array.
- Code :

```
array = np.array([1, 2, 3, 4]) scalar = 3 result = array * scalar
```

Chaque élément de l'array est multiplié par 3 : $[1 * 3, 2 * 3, 3 * 3, 4 * 3]$ Résultat : $[3, 6, 9, 12]$

Copie d'arrays et gestion de la mémoire

- **Assignation directe** : `array2 = array1` crée une référence à l'array original. Toute modification sur `array2` affecte `array1`.
- **Copie superficielle** (`view()`) : `array2 = array1.view()` crée une nouvelle vue. Les données sont partagées, mais les métadonnées (forme) sont indépendantes.
- **Copie profonde** (`copy()`) : `array2 = array1.copy()` crée une vraie copie indépendante. Il n'y a aucune relation avec `array1`.
- **Exemple pratique** : Si vous modifiez `array2` après une assignation directe, cela modifie également `array1`.

Introduction à Pandas

- **Qu'est-ce que Pandas ?**

- Pandas est une bibliothèque open-source pour la manipulation et l'analyse de données en Python, conçue pour traiter des données tabulaires.
- Le nom "Pandas" provient de "panel data", une structure utilisée en économétrie.

- **Fonctionnement de Pandas** : Pandas introduit deux structures de données principales :

- **Series** : Un tableau unidimensionnel étiqueté qui peut contenir n'importe quel type de données (entiers, chaînes, flottants, etc.).
- **DataFrame** : Une structure bidimensionnelle, semblable à une table SQL ou une feuille de calcul Excel, qui permet de stocker des données en lignes et colonnes étiquetées.

- **Interaction avec NumPy** : Pandas est construit sur NumPy, utilisant ses tableaux pour ses structures de données. Les opérations sur des données dans Pandas bénéficient des optimisations de performance de NumPy, rendant l'analyse des données efficace.

Avantages de Pandas

- **Facilité d'utilisation :**

- Syntaxe intuitive qui simplifie l'importation, la manipulation et l'analyse des données.

- **Flexibilité :**

- Prise en charge de multiples formats de données (CSV, Excel, SQL, JSON) pour une importation et une exportation faciles.

- **Fonctionnalités puissantes :**

- Large éventail de fonctions pour le filtrage, l'agrégation, le regroupement et la gestion des valeurs manquantes.

- **Intégration :**

- Fonctionne en harmonie avec d'autres bibliothèques Python, comme NumPy pour les calculs numériques et Matplotlib pour la visualisation.

- **Performance :**

- Basé sur NumPy, il offre des performances optimales grâce à la vectorisation et aux opérations efficaces.

- **Utilisations courantes :**

- Utilisé dans l'analyse de données exploratoire (EDA), le nettoyage des données, la transformation des données, et la préparation pour le machine learning.

Structures de données de base

- **Series :**

- `pd.Series(data, index=index)` crée une série avec des données étiquetées.
- Exemple : `pd.Series([10, 20, 30], index=['a', 'b', 'c'])` crée une série avec des valeurs et des index personnalisés.

- **DataFrame :**

- `pd.DataFrame(data, index=index, columns=columns)` pour créer un DataFrame avec des données, des index et des colonnes spécifiés.
- Exemple : `pd.DataFrame('Nom': ['Alice', 'Bob'], 'Âge': [25, 30])` crée un DataFrame avec des noms et des âges.

Chargement des données

- Pandas permet de charger des données à partir de plusieurs formats, ce qui en fait un outil flexible pour l'analyse.
- Chargement à partir de fichiers CSV :
 - `df = pd.read_csv('data.csv')` lit un fichier CSV et le convertit en DataFrame.
 - Options supplémentaires comme `sep`, `header`, et `na_values` permettent de personnaliser le chargement.
- Chargement à partir de fichiers Excel :
 - `df = pd.read_excel('data.xlsx')` pour lire les fichiers Excel directement dans un DataFrame.
 - Prend en charge plusieurs feuilles avec l'option `sheet_name`.

Exploration des données

- Une fois les données chargées, il est essentiel de les explorer pour comprendre leur structure et leur contenu.
- `df.head(n)` : Affiche les n premières lignes du DataFrame (5 par défaut) pour un aperçu rapide.
- `df.tail(n)` : Affiche les n dernières lignes du DataFrame pour voir les données à la fin.
- `df.info()` : Donne des informations sur les types de données, les valeurs manquantes et la mémoire utilisée.
- `df.describe()` : Fournit des statistiques descriptives (moyenne, écart-type, quartiles) sur les colonnes numériques.

Sélection et filtrage des données

- Pour une analyse approfondie, il est crucial de savoir sélectionner et filtrer les données dans un DataFrame.
- Sélection d'une colonne :
 - `df['colonne']` pour accéder à une colonne spécifique.
- Sélection de plusieurs colonnes :
 - `df[['colonne1', 'colonne2']]` pour obtenir un sous-DataFrame avec les colonnes désirées.
- Filtrage conditionnel :
 - `df[df['colonne'] > 10]` pour obtenir les lignes où la condition est vérifiée.
- Exemples de filtrage complexe :
 - `df[(df['colonne1'] > 5) & (df['colonne2'] < 10)]` pour filtrer avec plusieurs conditions.

Manipulation des données

- La manipulation des données est essentielle pour préparer les données à l'analyse.
- Ajout d'une nouvelle colonne :
 - `df['nouvelle_colonne'] = valeurs` pour créer une nouvelle colonne basée sur des calculs.
- Modification d'une colonne existante :
 - `df['colonne'] *= 2` pour multiplier tous les éléments d'une colonne par 2.
- Suppression d'une colonne :
 - `df.drop('colonne', axis=1, inplace=True)` pour retirer une colonne du DataFrame.
- Réorganisation des colonnes :
 - `df = df[['colonne1', 'colonne2', 'colonne3']]` pour changer l'ordre des colonnes.

Gestion des valeurs manquantes

- Gérer les valeurs manquantes est crucial pour une analyse précise des données.
- Identification des valeurs manquantes :
 - `df.isnull().sum()` pour compter les valeurs manquantes par colonne.
- Suppression des lignes avec des valeurs manquantes :
 - `df.dropna()` pour retirer toutes les lignes contenant des valeurs manquantes.
- Remplacement des valeurs manquantes :
 - `df.fillna(valeur)` pour remplacer les valeurs manquantes par une valeur spécifique, comme la moyenne.
- Exemple :
 - `df['colonne'].fillna(df['colonne'].mean(), inplace=True)` pour remplacer les valeurs manquantes par la moyenne de la colonne.

Groupement et agrégation

- Le groupement et l'agrégation sont essentiels pour résumer les données.
- Groupement des données par colonne :
 - `df.groupby('colonne').mean()` pour calculer la moyenne des valeurs par groupe.
- Utilisation de plusieurs fonctions d'agrégation :
 - `df.groupby('colonne').agg(['mean', 'sum'])` pour obtenir plusieurs statistiques.
- Agrégation sur plusieurs colonnes :
 - `df.groupby(['colonne1', 'colonne2']).size()` pour compter le nombre d'occurrences par groupe.

Tri et réorganisation des données

- Le tri des données est important pour une analyse claire et organisée.
- Tri des données :
 - `df.sort_values('colonne', ascending=True)` pour trier les données par une colonne spécifique.
 - Exemple : `df.sort_values('Âge', ascending=False)` pour trier par âge de manière décroissante.
- Réinitialisation des index :
 - `df.reset_index(drop=True)` pour réinitialiser les index après un filtrage ou un tri, en évitant d'ajouter une colonne d'index.
- Réorganisation des colonnes :
 - `df = df[['colonne1', 'colonne2', 'colonne3']]` pour changer l'ordre des colonnes.

Exportation des données

- Exporter les données est essentiel pour partager les résultats de l'analyse.
- Sauvegarde des données dans un fichier CSV :
 - `df.to_csv('output.csv', index=False)` pour écrire le DataFrame dans un fichier CSV sans les index.
- Sauvegarde des données dans un fichier Excel :
 - `df.to_excel('output.xlsx', index=False)` pour exporter le DataFrame vers Excel.
- Options supplémentaires :
 - Options comme `header`, `columns` pour personnaliser les exportations.

Visualisation des données

Introduction à la visualisation des données

- **Définition** : La visualisation des données est le processus de représentation graphique des données afin de communiquer efficacement des informations quantitatives ou qualitatives.
- **Objectif principal** : Faciliter la compréhension des données complexes par le biais de représentations visuelles intuitives.
- **Historique** : Évolution depuis les graphiques manuels jusqu'aux visualisations interactives avancées.

Importance dans la visualisation des données

- **Exploration des données** : Identification des patterns, tendances et anomalies.
- **Communication des résultats** : Transmission claire des découvertes à travers des visuels percutants.
- **Prise de décision** : Soutien aux décisions stratégiques basées sur l'analyse visuelle des données.
- **Validation des modèles** : Vérification des hypothèses et ajustement des modèles analytiques.

Types de données

- **Données quantitatives**

- **Continues** : Mesurables sur une échelle infinie (e.g., température, distance).
- **Discrètes** : Valeurs entières ou comptables (e.g., nombre de clients).

- **Données qualitatives**

- **Nominales** : Catégories sans ordre (e.g., couleur des yeux).
- **Ordinales** : Catégories avec ordre (e.g., niveau de satisfaction).

- **Données temporelles** : Données indexées par le temps (e.g., séries temporelles).

- **Données spatiales** : Données géographiques ou géolocalisées.

- **Données multivariées** : Plusieurs variables mesurées simultanément.

Principes clés de la visualisation

- **Précision** : Représentation fidèle des données sans distorsion.
- **Efficacité** : Transmission rapide et claire de l'information.
- **Intégrité** : Éviter les manipulations trompeuses ou les biais.
- **Perception visuelle** : Utilisation des principes de la perception humaine pour optimiser la compréhension (e.g., lois de Gestalt).
- **Choix des encodages** : Sélection des canaux visuels appropriés (position, couleur, taille, forme).

Bonnes pratiques de visualisation

- **Simplicité** : Éviter la surcharge d'information.
- **Cohérence** : Utiliser des styles et des encodages uniformes.
- **Annotation** : Fournir des labels clairs, légendes, titres explicatifs.
- **Accessibilité** : Considérer les différentes perceptions (e.g., daltonisme).
- **Éthique** : Représenter les données de manière honnête et transparente.

Éviter les visualisations trompeuses

- **Échelles non uniformes** : Manipulation des axes pour exagérer les effets.
- **Distorsion des proportions** : Utilisation inappropriée de volumes ou d'aires.
- **Omissions de données** : Cacher des informations cruciales.
- **Biais de sélection** : Choisir uniquement les données qui soutiennent une hypothèse.

Types de variables

Quand il s'agit de données tabulaires, on distingue généralement deux types de variables : les variables quantitatives et les variables qualitatives.

- **Variables quantitatives :**

- **Continues** : prennent une infinité de valeurs, comme le **poids d'une personne** en kilogrammes (ex : 68,3 kg).
- **Discrètes** : valeurs entières dénombrables, comme le **nombre d'enfants** par famille (ex : 0, 1, 2).

- **Variables qualitatives :**

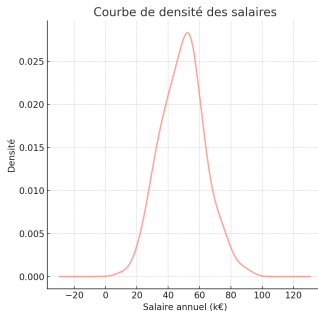
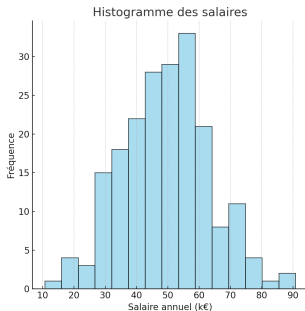
- **Nominales** : catégories sans ordre, comme la **couleur des yeux** (bleu, vert, marron).
- **Ordinales** : catégories avec un ordre, comme le **niveau d'éducation** (Bac, Bac+3, Bac+5).

Représentation des données quantitatives

Objectif : Visualiser la distribution d'une variable quantitative pour en comprendre les caractéristiques principales.

Types de représentations courantes :

- **Histogramme** : Représente la fréquence des observations dans différentes plages de valeurs.
- **Courbe de densité** : Estimation continue de la distribution des données, plus lisse qu'un histogramme.

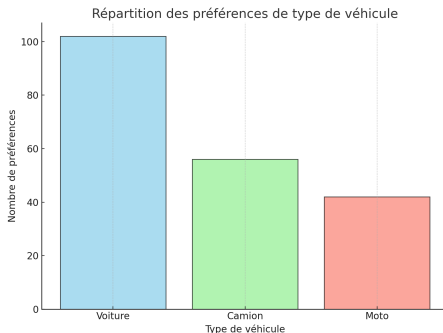


Représentation des variables qualitatives

Objectif : Visualiser la fréquence ou la distribution d'une variable qualitative pour en comprendre les proportions.

Types de représentations courantes :

- **Diagramme à barres** : Représente la fréquence des différentes catégories.
- **Diagramme circulaire** : Représente les proportions de chaque catégorie dans un ensemble. *Mais il n'est pas considéré comme une bonne pratique de visualisation des données.*



Variable continue vs variable continue

Objectif : analyser la relation entre deux variables quantitatives continues.

- **Nuage de points (scatter plot)** :

- Chaque point représente une observation.
- Permet d'identifier des corrélations, tendances ou clusters.

- **Carte de densité (heatmap)** :

- Représentation matricielle des valeurs.
- Les couleurs indiquent la densité ou la valeur des données.

- **Lignes de tendance** :

- Ajoutées aux nuages de points pour visualiser la relation générale.
- Peut inclure des modèles linéaires ou non linéaires.

Exemple : variable quantitative vs variable quantitative

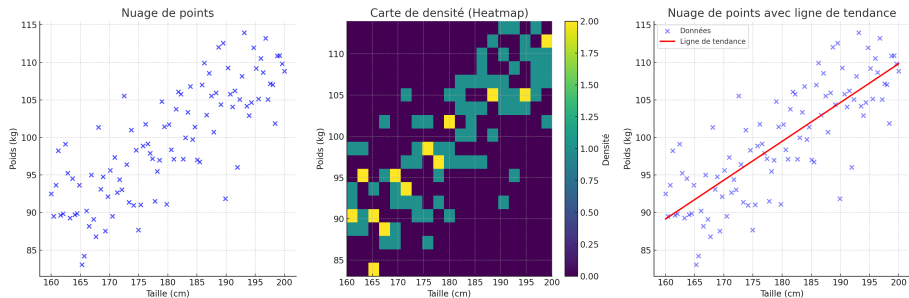


Figure: Distribution des salaires par niveau d'éducation

Variable catégorielle vs variable continue

Objectif : comparer une variable quantitative entre différentes catégories.

- **Diagramme en boîtes (box plot)** :
 - Visualise la distribution des données pour chaque catégorie.
 - Affiche les quartiles, médiane, minimum et maximum.
- **Diagramme de violon (violin plot)** :
 - Combine un box plot avec une estimation de la densité.
 - Montre la distribution complète des données.
- **Barres d'erreur (error bars)** :
 - Représentent la moyenne et la variabilité des données.
 - Utiles pour comparer les moyennes entre catégories.

Exemple : variable catégorielle vs variable continue

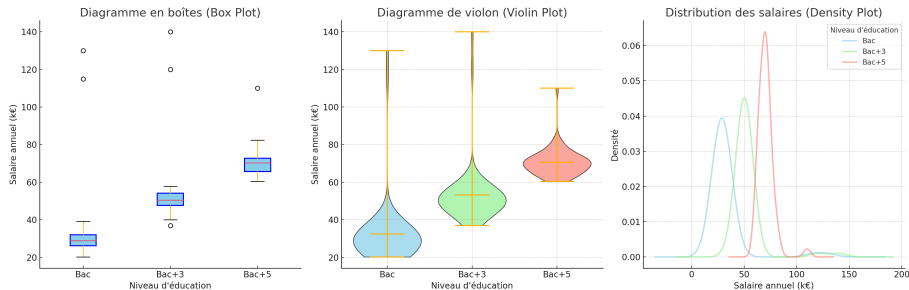


Figure: Distribution des salaires par niveau d'éducation

Variable catégorielle vs variable catégorielle

Objectif : analyser la relation entre deux variables qualitatives.

- **Tableaux de contingence** :

- Présentent les fréquences ou les pourcentages.
- Permettent de calculer des mesures d'association (chi-carré).

- **Diagrammes en mosaïque** :

- Représentation graphique des tableaux de contingence.
- La taille des rectangles est proportionnelle aux fréquences.

- **Diagrammes à barres empilées** :

- Comparaison des proportions entre catégories.
- Les barres sont divisées en segments pour chaque sous-catégorie.

Exemple : variable catégorielle vs variable catégorielle

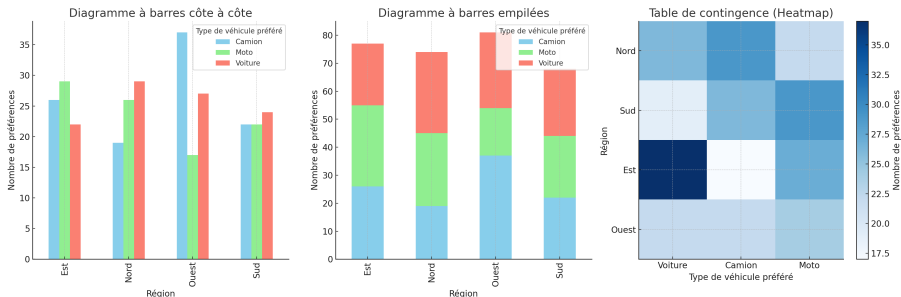


Figure: Relation entre le genre et la préférence de produit

Analyse multivariée : visualiser plus de deux variables simultanément.

- **Nuages de points en 3D** :

- Ajout d'un troisième axe pour une variable continue supplémentaire.
- Peut être difficile à interpréter sur des supports 2D.

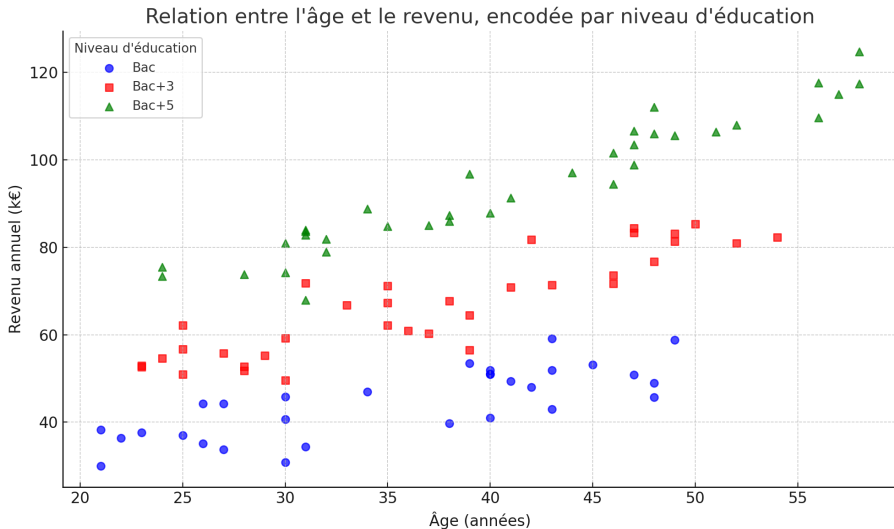
- **Utilisation de l'encodage visuel** :

- Couleur, taille, forme pour représenter des variables additionnelles.
- Exemple : couleur pour une variable catégorielle, taille pour une variable continue.

- **Coordonnées parallèles** :

- Chaque variable est représentée par un axe parallèle.
- Les observations sont des lignes traversant les axes à leurs valeurs respectives.

Exemple : nuage de points avec encodage couleur



Introduction aux données temporelles

Définition : Les données temporelles sont des observations chronologiques, où chaque point de données est associé à un instant spécifique dans le temps.

Exemples courants :

- **Séries temporelles financières** : cours des actions, indices boursiers.
- **Capteurs IoT** : mesures de température ou d'humidité collectées périodiquement.
- **Données démographiques** : évolution de la population au fil des années.

Caractéristiques des données temporelles

- **Dépendance temporelle** : Les observations successives sont corrélées. Par exemple, la température d'un jour dépend de celle du jour précédent.
- **Saisonnalité** : Répétition de schémas à intervalles réguliers (ex : ventes de glace plus élevées en été).
- **Tendance** : Évolution générale sur une longue période (ex : croissance des ventes sur plusieurs années).
- **Composante aléatoire** : Variations imprévisibles non expliquées par les autres composants.

Types de visualisations pour les données temporelles

Objectif : Visualiser l'évolution des données au fil du temps pour mieux comprendre les tendances et les variations.

- **Graphiques linéaires** : Représente une série de points connectés par des lignes. *Exemple* : évolution de la température quotidienne au cours de l'année.
- **Diagrammes de saisonnalité** : Utilisés pour visualiser les motifs saisonniers. *Exemple* : ventes de vêtements par saison.
- **Graphiques à barres empilées** : Utilisés pour décomposer les contributions de plusieurs sous-composantes dans une série temporelle. *Exemple* : décomposition des ventes mensuelles par catégorie de produit.

Exemple de graphique temporel

Exemple : Evolution des ventes mensuelles

- Ce graphique montre la tendance générale, les fluctuations mensuelles, et les effets saisonniers.
- Il aide à identifier les pics de ventes, les baisses et les éventuelles anomalies.

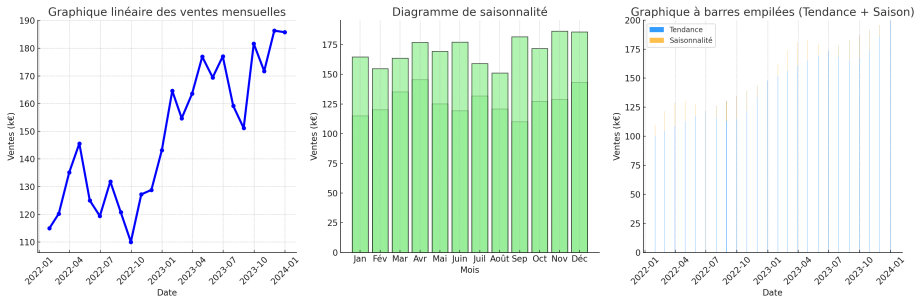


Figure: Exemple de représentation des données temporelles

Introduction à Matplotlib

Définition : Matplotlib est une bibliothèque de visualisation de données en Python, créée par John Hunter en 2003, inspirée par MATLAB.

Caractéristiques :

- **Types de graphiques** : Supporte des courbes, histogrammes, diagrammes à barres, nuages de points, box plots, pie charts, etc.
- **Flexibilité** : Personnalisation avancée (couleurs, styles de ligne, échelles logarithmiques, etc.).
- **Compatibilité** : Fonctionne avec des environnements variés (scripts Python, notebooks Jupyter, interfaces graphiques).

Comparaison : Matplotlib est souvent comparé à Seaborn et Plotly. Il se distingue par :

- Un **contrôle de bas niveau** sur chaque élément du graphique.
- Une **large compatibilité** avec des bibliothèques comme NumPy et Pandas.

Architecture de Matplotlib

Structure de Matplotlib :

- **Figure** : Conteneur principal. Peut inclure plusieurs sous-graphiques (Axes).
- **Axes** : Région de tracé qui contient les données. Une Figure peut avoir plusieurs Axes.
- **Artistes** : Tous les éléments visibles (texte, lignes, légendes, etc.) sont des artistes.

API : Matplotlib propose deux API principales :

- **pyplot** : Interface à la MATLAB, très intuitive pour des graphiques rapides.
- **API orientée objet** : Contrôle détaillé des Figures et Axes, préférable pour des visualisations complexes.

Principe : L'utilisateur crée une Figure, ajoute un ou plusieurs Axes, puis personnalise chaque élément.

Architecture de Matplotlib : illustration

- **Figure** : Conteneur principal qui peut inclure plusieurs sous-graphiques (Axes).
- **Axes** : Région de tracé qui contient les données. Une Figure peut avoir plusieurs Axes.
- **Artistes** : Les éléments visibles (texte, lignes, légendes, etc.) sont des artistes.

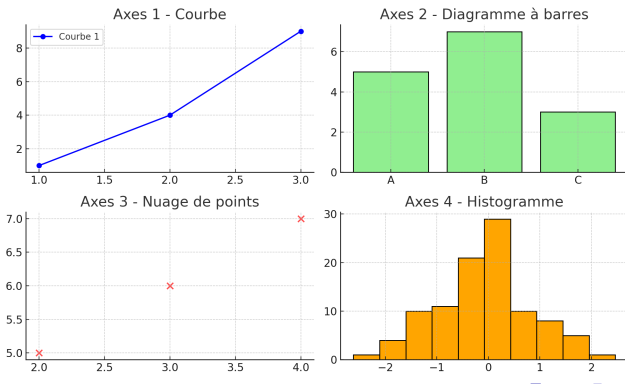


Figure et Axes : concepts clés

Figure :

- Créée avec `plt.figure()`. Contient l'ensemble des éléments graphiques.
- Peut inclure plusieurs Axes (`add_subplot()`, `subplots()`).
- Méthode `savefig()` pour exporter vers divers formats (PNG, PDF, SVG).

Axes :

- Ajoutés à une Figure via `add_axes()` ou `subplot()`.
- Chaque Axes est un graphique unique contenant des titres, légendes, et étiquettes.
- Les méthodes `set_title()`, `set_xlabel()`, `set_ylabel()` permettent de définir les attributs.

Exemple :

- `fig, ax = plt.subplots()` : Crée une Figure et des Axes.

Pyplot vs API orientée objet

Pyplot :

- Interface de type MATLAB, idéale pour des graphiques simples et rapides.
- Chaque fonction de `plt` modifie la figure courante.
- **Exemple** : `plt.plot(x, y)`, `plt.xlabel('X Axis')`, `plt.show()`.

API orientée objet :

- Donne un contrôle plus fin sur chaque élément (particulièrement utile pour des figures complexes).
- **Exemple** :
 - `fig, ax = plt.subplots()`
 - `ax.plot(x, y)`
 - `ax.set_title('Titre')`
- Recommandée pour des projets plus importants où une personnalisation est nécessaire.

Pyplot vs API orientée objet : illustration 1/2

Pyplot :

- `plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'bo-', label='Pyplot')`
- `plt.title('Pyplot (MATLAB-like)')`
- `plt.xlabel('X Axis')`
- `plt.ylabel('Y Axis')`
- `plt.legend()`
- `plt.grid(True)`

API orientée objet :

- `fig, ax = plt.subplots()`
- `ax.plot([1, 2, 3, 4], [1, 4, 9, 16], color='red', linestyle='--', marker='s', label='API OO')`
- `ax.set_title('API orientée objet')`
- `ax.set_xlabel('X Axis')`
- `ax.set_ylabel('Y Axis')`
- `ax.legend()`
- `ax.grid(True)`

Pyplot vs API orientée objet : illustration 2/2

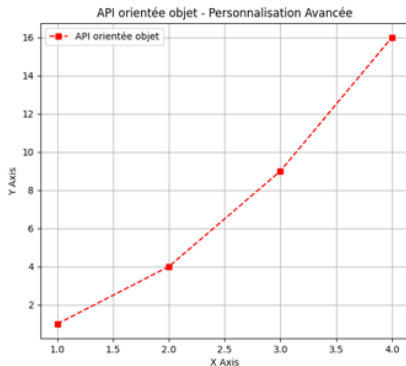
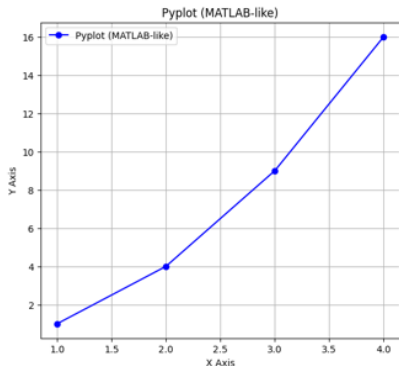


Figure: Comparaison : Pyplot (à gauche) vs API orientée objet (à droite)

Création d'une Figure et d'Axes

Création de la Figure :

- `plt.figure(figsize=(width, height))` : Définit la taille de la figure.
- `fig, ax = plt.subplots(nrows, ncols)` : Crée une grille d'Axes.

Personnalisation des Axes :

- `ax.plot(x, y)` : Ajoute une courbe aux Axes.
- `ax.set_xlabel('Label X')` et `ax.set_ylabel('Label Y')` : Ajoute des étiquettes.
- `ax.grid(True)` : Active le quadrillage.

Exemple :

- `fig, ax = plt.subplots(2, 2)` : Crée une figure avec une grille de 2x2 Axes.
- Chaque Axes est manipulé indépendamment, permettant une personnalisation distincte.

Tracé de graphiques de base

Courbes (Line plots) :

- `ax.plot(x, y, color='blue', linestyle='--', marker='o')` : Permet de tracer des courbes avec couleur, style de ligne, et marqueurs personnalisés.
- Arguments : `linewidth`, `alpha` (transparence), `label` (légende).

Diagrammes à barres (Bar plots) :

- `ax.bar(categories, values, color='green')` : Tracé d'un diagramme à barres.
- Options : `stacked=True` pour les barres empilées, `width` pour ajuster la largeur.

Nuages de points (Scatter plots) :

- `ax.scatter(x, y, c='red', s=area)` : Chaque point est tracé à la position (x, y).
- Options : `s` (taille des points), `c` (couleur), `alpha` (transparence).

Personnalisation des graphiques

Titre, étiquettes et légendes :

- `ax.set_title('Titre du graphique')` : Définit le titre du graphique.
- `ax.set_xlabel('Label de l'axe X')` : Définit l'étiquette de l'axe X.
- `ax.legend()` : Affiche la légende si des labels ont été fournis.

Couleurs et styles :

- Couleurs : Noms (`'blue'`), codes hexadécimaux (`'1f77b4'`).
- Styles de ligne : `'-'` (plein), `'--'` (pointillé), `':'`, etc.
- Marqueurs : `'o'`, `'s'`, `'.'`, *pour les points*.

Échelles et limites :

- `ax.set_xscale('log')` : Définit une échelle logarithmique.
- `ax.set_ylim([0, 100])` : Définit les limites de l'axe Y.

Sauvegarde et affichage des graphiques

Affichage :

- `plt.show()` : Affiche la figure à l'écran.
- Utilisation typique dans les scripts ou les notebooks Jupyter.

Sauvegarde :

- `fig.savefig('nom_du_fichier.png', dpi=300)` : Sauvegarde la figure avec une résolution de 300 DPI.
- Formats supportés : PNG, PDF, SVG, etc.

Conseils pratiques :

- Utilisez `bbox_inches='tight'` pour éviter les marges inutiles lors de la sauvegarde.
- Pour des publications, choisissez un DPI élevé (`dpi=300` ou plus).

Gestion des sous-graphiques (Subplots)

Création de subplots :

- `fig, axs = plt.subplots(nrows, ncols)` : Crée une grille de subplots.
- Chaque élément `axs` est un objet `Axes` manipulable indépendamment.

Personnalisation des subplots :

- `axs[i, j].plot(x, y)` : Trace un graphique sur le subplot (i, j).
- `plt.tight_layout()` : Ajuste automatiquement l'espacement entre les subplots pour éviter le chevauchement.

Exemple :

- `fig, axs = plt.subplots(2, 3)` : Crée une grille de 2x3 subplots.
- Chaque subplot peut être personnalisé (titre, légendes, étiquettes).

Gestion des sous-graphiques : illustration 1/2

Code :

```
fig, axs = plt.subplots(1, 3,  
figsize=(18, 6))  
data1 = np.random.normal(10, 2,  
100)  
data2 = np.random.normal(15, 3,  
100)
```

Sous-graphe 1 : Boxplot

- `axs[0].boxplot([data1, data2], labels=['Groupe 1', 'Groupe 2'], patch_artist=True)`
- `axs[0].set_title('Boxplot')`

Sous-graphe 2 : Violin Plot

- `axs[1].violinplot([data1, data2], showmeans=True)`
- `axs[1].set_title('Violin Plot')`

Sous-graphe 3 : Strip Plot

- `group_labels = np.concatenate([np.full(100, 'Groupe 1'), np.full(100, 'Groupe 2')])`
- `values = np.concatenate([data1, data2])`
- `axs[2].scatter(group_labels, values, alpha=0.7, color='red')`
- `axs[2].set_title('Strip Plot')`

Gestion des sous-graphiques : illustration 2/2

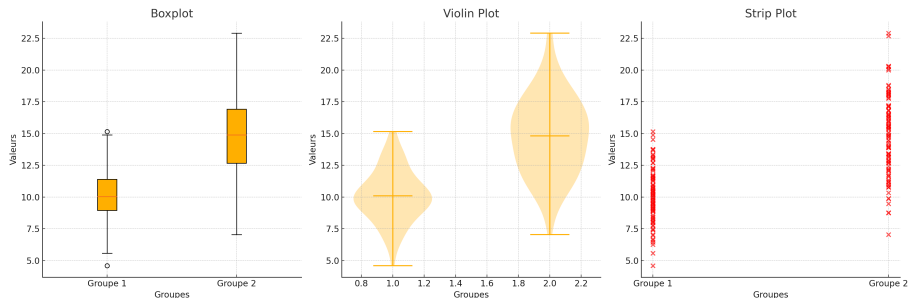


Figure: Sous-graphiques incluant Boxplot, Violin Plot et Strip Plot sur une seule ligne

Introduction à Seaborn

Définition :

- Seaborn est une bibliothèque de visualisation de données basée sur Matplotlib, conçue pour faciliter la création de graphiques statistiques en Python.
- Créée par Michael Waskom, elle est largement utilisée pour la visualisation exploratoire des données grâce à ses fonctionnalités de haut niveau.

Pourquoi utiliser Seaborn ?

- Simplifie la création de graphiques complexes et statistiquement informatifs.
- Intégration native avec Pandas pour manipuler directement les DataFrames.
- Couleurs et thèmes esthétiques par défaut, évitant des personnalisations manuelles.

Principales fonctionnalités de Seaborn

Graphiques disponibles :

- **Relations** : `scatterplot()`, `lineplot()` pour représenter les relations entre deux variables.
- **Distributions** : `histplot()`, `kdeplot()` pour examiner la distribution d'une variable.
- **Comparaisons de catégories** : `boxplot()`, `violinplot()` pour visualiser la répartition des valeurs par catégorie.
- **Matrice de corrélation** : `heatmap()` pour représenter les relations entre plusieurs variables.

Fonctionnalités avancées :

- `hue`, `size`, `style` pour l'encodage des variables supplémentaires.
- Support de graphiques multi-variés via `FacetGrid()` et `pairplot()`.

Structure et concepts de Seaborn

Seaborn vs Matplotlib :

- Seaborn est construit au-dessus de Matplotlib, offrant des abstractions de plus haut niveau.
- Utilisation simplifiée pour la visualisation statistique.
- Contrôle de bas niveau toujours possible via l'accès aux objets Matplotlib sous-jacents.

Exemple typique :

- Importation : `import seaborn as sns`
- Utilisation avec Pandas : `sns.scatterplot(data=df, x='x', y='y', hue='category')`
- Personnalisation : thèmes (`sns.set_style()`), palettes de couleurs (`sns.color_palette()`).

Thèmes et palettes de Seaborn

Thèmes prédéfinis :

- Seaborn fournit plusieurs thèmes par défaut : `white`, `dark`, `whitegrid`, `darkgrid`, `ticks`.
- `sns.set_style('whitegrid')` permet de changer le thème du graphique.

Palettes de couleurs :

- `color_palette()` permet de choisir parmi des palettes par défaut (`'deep'`, `'muted'`, etc.) ou de créer ses propres palettes.
- **Exemple** : `sns.set_palette('muted')` pour définir la palette de couleurs.

Exemple pratique :

- `sns.histplot(data=df, x='valeur', hue='groupe', palette='muted')`
- Amélioration de l'esthétique par l'utilisation de thèmes et de palettes harmonieuses.

Visualisation des relations avec Seaborn

Graphiques relationnels :

- `scatterplot()` : Représentation de la relation entre deux variables.
- `lineplot()` : Graphique de tendance, souvent utilisé pour des séries temporelles.

Encodage des informations supplémentaires :

- `hue` : Utilisé pour distinguer les groupes par couleur.
- `size` et `style` : Encodage via la taille ou la forme des marqueurs pour des informations supplémentaires.

Exemple :

- `sns.scatterplot(data=df, x='age', y='income', hue='gender', style='region')`
- Affichage des relations entre plusieurs variables avec des couleurs et des styles distincts.

Histogramme avec Seaborn : illustration 1/2

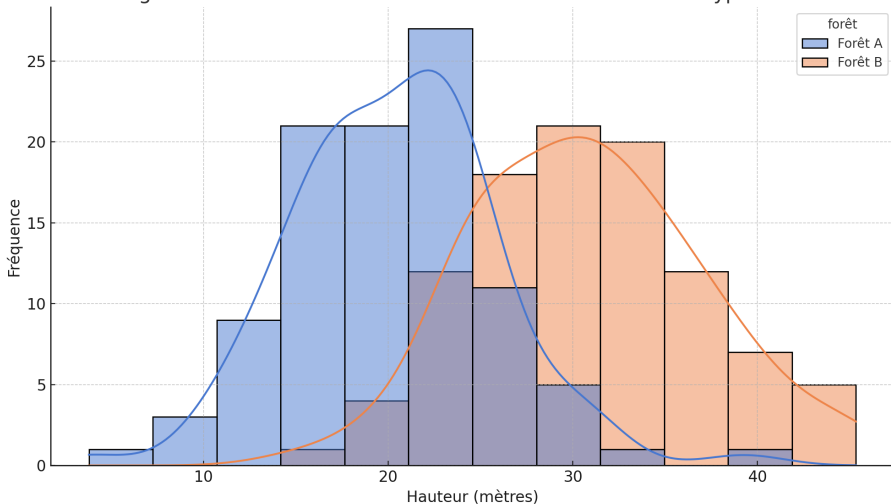
Exemple concret : Comparaison de la hauteur des arbres dans deux types de forêts (Forêt A et Forêt B).

Code pour générer l'histogramme :

```
import seaborn as sns
import pandas as pd
data = {'hauteur': [...], 'forêt': [...]}
df = pd.DataFrame(data)
sns.histplot(data=df, x='hauteur', hue='forêt', palette='muted',
kde=True)
plt.xlabel("Hauteur (mètres)")
plt.ylabel("Fréquence")
```

Histogramme avec Seaborn : illustration 2/2

Histogramme avec Seaborn - Hauteur des arbres dans deux types de forêts



Grilles de visualisation avec Seaborn

FacetGrid et Pairplot :

- `FacetGrid()` permet de créer des visualisations multi-variées en subdivisant les données selon une ou plusieurs catégories.
- `pairplot()` est utilisé pour tracer des paires de relations entre toutes les variables d'un jeu de données, utile pour l'analyse exploratoire.

Exemple de FacetGrid :

- `g = sns.FacetGrid(df, col='species')`
- `g.map(plt.hist, 'sepal_length')`

Exemple de pairplot() :

- `sns.pairplot(df, hue='species')`
- Permet de visualiser la distribution et la corrélation entre plusieurs variables simultanément.

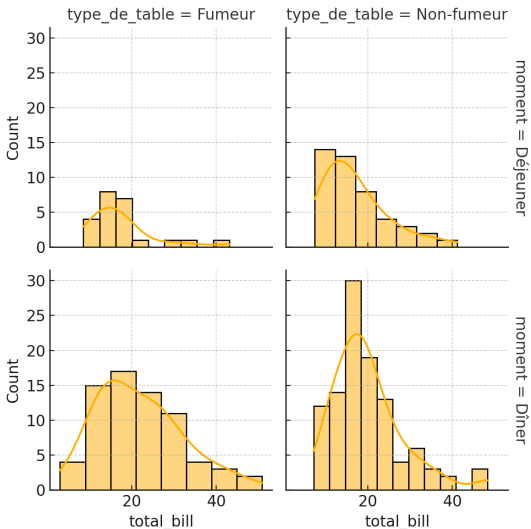
Illustration de FacetGrid avec Seaborn : 1/2

Exemple : Utilisation de FacetGrid pour explorer la distribution des factures totales (`total_bill`) par sexe et par moment de la journée (`time`).

Code :

```
import seaborn as sns
tips = sns.load_dataset("tips")
g = sns.FacetGrid(tips, col="sex", row="time",
margin_titles=True)
g.map_dataframe(sns.histplot, x="total_bill", kde=True)
```


Illustration de FacetGrid avec Seaborn : 2/2



Introduction à Plotly

Définition :

- Plotly est une bibliothèque de visualisation de données interactive pour Python, supportant une large variété de graphiques (courbes, barres, cartes, etc.).
- Permet la création de graphiques statiques, animés, et interactifs.

Pourquoi utiliser Plotly ?

- Interaction native : Permet l'exploration des graphiques en zoomant, en survolant des points pour obtenir des informations supplémentaires, etc.
- Support de graphiques complexes comme des cartes géographiques, graphiques 3D, et graphiques financiers.
- Idéal pour les dashboards grâce à l'intégration facile avec Dash.

Fonctionnalités principales de Plotly

Types de graphiques disponibles :

- **Graphiques de base** : Courbes (`line_plot`), diagrammes à barres (`bar`), et nuages de points (`scatter`).
- **Graphiques avancés** : Graphiques 3D, cartes géographiques, diagrammes en sunburst, etc.
- **Graphiques financiers** : Chandeliers (`candlestick`), graphiques OHLC, etc.

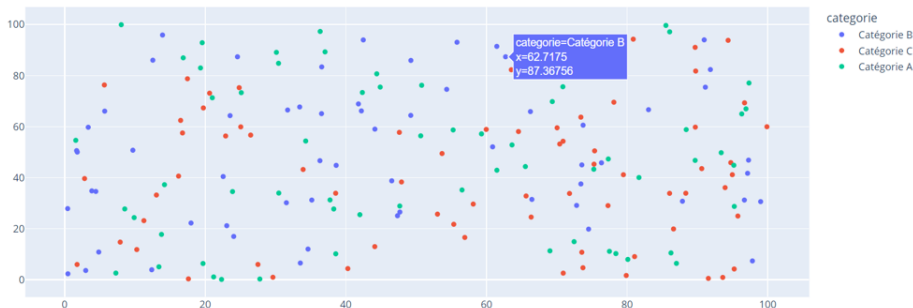
Interactivité :

- Les graphiques Plotly sont interactifs par défaut, permettant de zoomer, déplacer, et afficher des légendes dynamiques.
- Permet l'exportation en HTML pour une intégration facile dans des pages web.

Exemple de graphique avec Plotly

Code :

- `fig = px.scatter(df, x='valeur_x', y='valeur_y', color='categorie')`
- `fig.show()` pour afficher le graphique dans un navigateur ou un notebook Jupyter.



Comparaison de Plotly avec d'autres bibliothèques

Plotly vs Matplotlib :

- **Interactivité** : Plotly est interactif par défaut, tandis que Matplotlib crée des graphiques statiques.
- **Complexité** : Matplotlib offre un contrôle plus détaillé mais nécessite plus de configuration pour créer des graphiques interactifs.

Plotly vs Seaborn :

- **Visuels** : Seaborn est construit sur Matplotlib et offre des graphiques statiques stylisés.
- **Intégration** : Plotly est mieux adapté pour des dashboards interactifs, tandis que Seaborn est plus utilisé pour une exploration de données rapide et simple.

Dashboards et applications avec Plotly

Dash pour la création de dashboards :

- Plotly s'intègre parfaitement avec Dash, un framework pour créer des applications web interactives en Python.
- Les graphiques Plotly peuvent être utilisés dans Dash pour créer des interfaces utilisateurs dynamiques.

Exemple de code avec Dash :

- `import dash et import dash_html_components as html`
- `import dash_core_components as dcc`
- **Exemple** : Création d'un graphique dynamique avec des composants `dcc.Graph` pour afficher un graphique Plotly.
- `app.layout = html.Div(children=[dcc.Graph(figure=fig)])`

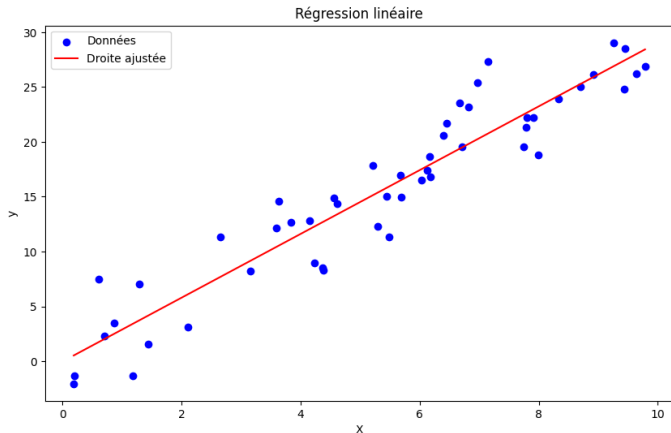
Utilité : Très utile pour la présentation de résultats, permettant une interaction avec les graphiques pour obtenir des insights approfondis.

Modélisation des données

Régression linéaire simple

Soit un ensemble de n observations x_1, x_2, \dots, x_n avec les labels correspondants y_1, y_2, \dots, y_n , on cherche le modèle linéaire qui ajuste le mieux ces données.

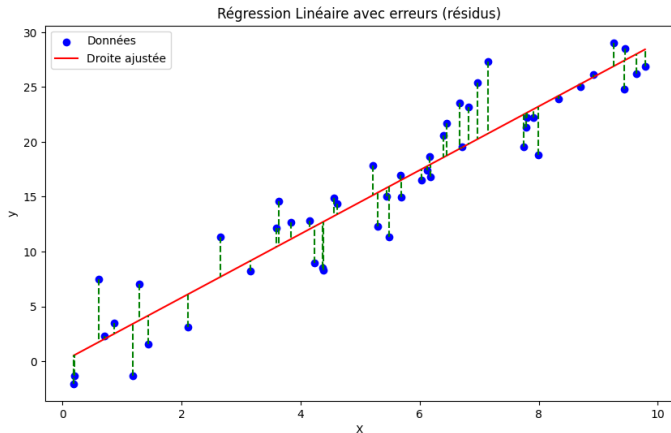
$$\hat{y} = \beta_0 + \beta_1 x$$



Résidus

Definition

Soit $\hat{y}_i = \beta_0 + \beta_1 x_i$ la i ième prédiction du modèle. Le i ième résidu, noté e_i , est alors défini comme l'erreur de prédiction sur i ième observation : $e_i = y_i - \hat{y}_i$.



Méthode des moindres carrés

On considère la somme des carrés des résidus, notée RSS :

$$RSS = e_1^2 + e_2^2 + \dots + e_n^2$$

L'approche par moindre carrés consiste à estimer les coefficients β_0 et β_1 en minimisant la RSS . Autrement dit :

$$\arg \min_{\beta_0, \beta_1} \left(\sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i)^2 \right)$$

Le problème peut être résolu d'une manière analytique. On obtient :

$$\begin{aligned} \beta_1 &= \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} \\ \beta_0 &= \bar{y} - \beta_1 \bar{x} \end{aligned} \tag{1}$$

Régression linéaire multiple

On considère n observations X^1, X^2, \dots, X^n où chaque observation X^i est désormais un vecteur ayant p composantes (p variables explicatives).

$$X^i = \begin{pmatrix} x_1^i \\ x_2^i \\ \vdots \\ x_p^i \end{pmatrix}$$

La régression linéaire s'écrit alors :

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p$$

Les coefficients $\beta_0, \beta_1, \dots, \beta_p$ sont déterminés par la méthode des moindres carrés :

$$\arg \min_{\beta_0, \beta_1, \dots, \beta_p} \sum_{i=1}^n \left(y^i - \left(\beta_0 + \sum_{j=1}^p \beta_j x_j^i \right) \right)^2$$

L'équation normale

La régression linéaire peut être écrite sous une forme vectorielle :

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$$

où X est appelée matrice de design.

$$\mathbf{X} = \begin{pmatrix} 1 & x_1^1 & \dots & x_p^1 \\ \vdots & \vdots & \dots & \vdots \\ 1 & x_1^n & \dots & x_p^n \end{pmatrix}$$

La somme des carrés des résidus est donnée par :

$$RSS = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$$

On peut montrer que si la matrice de design X est de rang plein, alors :

$$\boldsymbol{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

Régression polynomiale

La régression linéaire peut prendre en compte les dépendances non linéaires entre les variables explicatives x_1, x_2, \dots, x_p et la variable expliquée y . Lorsque cette dépendance prend la forme d'un polynôme de degré d , la régression linéaire s'écrit alors :

$$\hat{y} = \beta_{00} + \sum_{j=1}^p \beta_{ij} x_j + \sum_{j=1}^p \beta_{ij} x_j^2 + \dots + \sum_{j=1}^p \beta_{ij} x_j^d$$

Les coefficients β_{ij} peuvent être de la même manière, avec la méthode des moindres carrés.

Remarque : On peut utiliser n'importe quelle fonction non linéaire pour transformer les variables explicatives (\cos , \ln , etc.). Le modèle reste tout de même linéaire (linéarité par rapport aux coefficients).

Choix de modèle

Il y a plusieurs manières d'évaluer la pertinence d'un modèle de régression linéaire. Le coefficient de détermination R^2 mesure l'ajustement du modèle. Il est donné par :

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

Pour une régression linéaire multiple, on préférera cependant le coefficient de détermination ajusté R_a^2 .

$$R_a^2 = 1 - \frac{n-1}{n-k-1} * (1 - R^2)$$

où n est le nombre d'observations et k le nombre de variables.

Des critères comme le AIC et le BIC sont également utilisés pour sélectionner un modèle.

Métriques de performance : régression

On dispose d'un certain nombre de métriques pour évaluer les performances des modèles de machine learning. Celles-ci peuvent être divisées en deux catégories.

Régression

- L'erreur quadratique moyenne (MSE) : elle est définie comme la moyenne des carrés des écarts entre les prédictions et les valeurs observées.

$$MSE = \frac{1}{n} \sum_{i=1}^n (f(x_i) - y_i)^2$$

- La racine carrée de l'erreur quadratique moyenne (RMSE) :

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (f(x_i) - y_i)^2}$$

Régression logistique : introduction

La régression logistique est une technique d'analyse statistique utilisée pour modéliser la probabilité d'une variable dépendante binaire. C'est un cas particulier de modèle linéaire généralisé qui est utilisé pour des problèmes de classification.

Principes de la régression logistique :

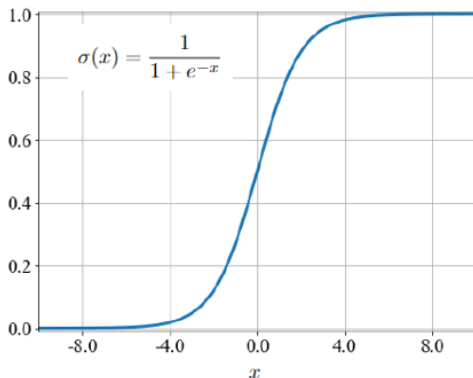
- **Variable dépendante** : On cherche la probabilité que la variable dépendante (y) appartienne à une classe (0 ou 1, vrai ou faux, succès ou échec). Autrement dit, on cherche à modéliser $P(y = 1)$ en fonction des variables dépendantes (explicatives) x .
- **Odds ratio** : Plus concrètement, on cherche à exprimer la côte anglaise (odd ratio) en fonction des variables dépendantes (x).

$$\ln \frac{p(\mathbf{x})}{1 - p(\mathbf{x})} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p$$

Régression logistique : fonction sigmoïde

Après quelques simplifications, on peut écrire la probabilité $p(x)$ (la probabilité pour que y soit un succès par exemple) :

$$p(\mathbf{x}) = \frac{1}{1 + e^{-(\beta^T \mathbf{x})}}$$



Calcul des coefficients de la régression logistique

Les coefficients de la régression logistique peuvent être calculés en minimisant le risque empirique par rapport à une fonction de coût sous forme d'entropie croisée :

$$\arg \min_{\beta_0, \beta_1, \dots, \beta_p} \left(- \sum_{i=1}^n y_i \ln(\hat{y}_i) + (1 - y_i) \ln(1 - \hat{y}_i) \right)$$

Métriques de performance : classification 1/2

Accuracy : L'accuracy est la métrique de base qui permet d'évaluer les performance d'un modèle de classification. Elle est définie comme :

$$\text{Accuracy} = \frac{\text{Nombre de prédictions correctes}}{\text{Nombre total de prédictions}}$$

Matrice de confusion : La matrice de confusion est une représentation permettant d'offrir plus de finesse par rapport à l'accuracy, notamment quand le jeu de données est déséquilibré (présence de classes majoritaires). Elle compare les prédictions du modèle avec les valeurs réelles et est structurée comme suit :

		Valeur Prédite	
		Positif	Négatif
Valeur Réelle	Positif	Vrai Positif (VP)	Faux Négatif (FN)
	Négatif	Faux Positif (FP)	Vrai Négatif (VN)

Métriques de performance : classification 2/2

A partir de la matrice de confusion, on peut dériver d'autres métriques :

- Précision : elle est définie comme la proportion des prédictions correctes parmi toutes les prédictions positives :

$$\text{Précision} = \frac{VP}{VP + FP}$$

- Rappel (recall) : il représente la proportion des vrais positifs correctement prédits par le modèle.

$$\text{Rappel} = \frac{VP}{VP + FN}$$

- Score F1 (F1-score) : Le score F1 est défini comme la moyenne harmonique de la précision et du rappel.

$$\text{Score F1} = 2 \frac{\text{Précision} \times \text{Rappel}}{\text{Précision} + \text{Rappel}}$$

Introduction au traitement des données textuelles (NLP)

Tâches classiques en NLP

On retrouve dans le NLP un certain nombre de tâches classiques, qui servent à la fois dans les applications pratiques que dans l'évaluation de l'état de l'art.

- Modélisation du langage (*language modeling*)
- Classification de texte
- Extraction d'informations
- Détection de thématiques (*topic modeling*)
- Résumé de texte
- Question/réponse
- Traduction automatique
- Agent conversationnel (domaine ouvert ou fermé)

Pourquoi le NLP est si difficile ?

Il y a plusieurs facteurs qui rendent le langage naturel difficile à traiter par la machine :

- Le langage naturel est un problème en grande dimension.

- Le langage naturel est **ambigu**.

- *"Acheter un avocat"*

- Le langage naturel repose sur le **sens commun**.

- Il fait froid *en hiver*

- Le beurre fond dans le four

- Le langage naturel n'est pas basé sur des **règles**.

- *En bleu adorable fleurit*

- Le toit de métal du clocher. Alentour*

- Plane un cri d'hirondelles, autour*

- S'étend le bleu le plus touchant. Le soleil*

- Friedrich Hölderlin*

Pré-traitement des données textuelles

Les données textuelles sont non structurées. Les principales étapes de pré-traitement impliquent :

- Segmentation des phrases et **tokenisation**.
- Suppression de **stop words**, de la ponctuation, etc.
- **Vectorisation** : Transformation du texte en vecteurs numériques.
- **Stemming, lemmatisation**, conversion des majuscules, etc.
- **Normalisation** : traitement des accents, des caractères spéciaux, etc.
- **Suppression des entités nommées** : Filtrage des noms de lieux, personnes, organisations, etc.
- Détection du langage, **POS tagging** (part-of-speech), etc.
- Correction orthographique : Pour améliorer la qualité des textes bruts.

Tokenisation

La tokenisation consiste à segmenter une phrase en unités plus petites appelées **tokens**, qui peuvent être des mots, des caractères ou des sous-parties de mots.

Types de tokenisation :

- **Tokenisation par mots** : Chaque mot est un token.
 - *Exemple* : "Il fait beau à Paris." → ["Il", "fait", "beau", "à", "Paris"]
- **Tokenisation par caractères** : Chaque caractère est un token.
 - *Exemple* : "Paris" → ["P", "a", "r", "i", "s"]
- **Tokenisation par sous-mots (subword)** : Découpe les mots en sous-unités fréquentes, utile pour gérer les langues morphologiquement complexes ou les néologismes.
 - *Exemple* : "démocratisation" → ["démocrat", "isation"]

La tokenisation est une étape essentielle du pré-traitement dans le NLP. Elle permet de transformer du texte brut en unités traitables par les algorithmes de machine learning.

Stemming and lemmatisation

- Le **stemming** est l'opération qui consiste à supprimer les suffixes et réduire les mots à leur forme de base.
 - marcher → march
 - marchait → march
 - marcha → march
 - marchassiez → march
 - marché → march
- La **lemmatisation** est l'opération qui consiste à réduire un mot à sa racine.
 - nous → nous
 - sommes → être
 - venus → venir
 - faire → faire
 - les → le
 - marchés → marché

- **One-Hot encoding:** Étant donné un vocabulaire V , chaque mot est représenté par un vecteur binaire (à 0 ou 1) de dimension V .
 - $V = \{Tom, mange, pomme, chien, ciel\}$, où $\dim(V) = 5$
 - $Tom = [1, 0, 0, 0, 0]$
 - $mange = [0, 1, 0, 0, 0]$
 - $pomme = [0, 0, 1, 0, 0]$
 - etc.
- **N-grams**
 - "machine learning est très utilisé en NLP."
 - 1-gram: {machine, learning, est, très, utilisé, en, NLP}
 - 2-gram: {machine learning, learning est, est utilisé, utilisé en, en NLP}

Reconnaissance des entités nommées (NER)

La reconnaissance d'entités nommées (**NER**) est une tâche de NLP qui consiste à identifier et classier des entités dans un texte, comme des personnes, des organisations, des lieux, des dates, etc.

Exemples d'entités nommées :

- **Personne** : "Albert Einstein"
- **Organisation** : "Université de Princeton"
- **Lieu** : "Paris"
- **Date** : "15 octobre 2024"

Catégories courantes d'entités nommées :

- **Personnes** : Noms de personnes célèbres ou non.
- **Lieux** : Pays, villes, régions.
- **Organisations** : Entreprises, institutions, ONG.
- **Dates et heures** : Dates précises, années, heures.
- **Monnaie et quantités** : Montants financiers, unités de mesure.

La NER est cruciale pour de nombreuses applications NLP, telles que l'extraction d'informations, la recherche d'informations, et la classification de documents.

Part-of-Speech (POS) Tagging

Le **Part-of-Speech (POS) Tagging** est une tâche de NLP qui consiste à assigner à chaque mot d'une phrase son rôle grammatical (catégorie grammaticale).

Exemples de catégories grammaticales (POS tags) :

- **Nom (N)** : Objet, personne, lieu, idée. *Ex. : chat, Paris*
- **Verbe (V)** : Action, état. *Ex. : mange, est*
- **Adjectif (Adj)** : Caractéristique, qualité. *Ex. : grand, heureux*
- **Adverbe (Adv)** : Modifie un verbe ou une phrase. *Ex. : rapidement, bien*
- **Déterminant (Det)** : Définit un nom. *Ex. : le, une*
- **Préposition (Prep)** : Indique une relation. *Ex. : dans, avec*

Exemple de POS Tagging :

- *"Le chat mange une souris."*
- **Résultat** : Le/Det chat/N mange/V une/Det souris/N

Le POS Tagging est une étape clé dans de nombreuses applications de NLP, notamment pour l'analyse syntaxique, la traduction automatique et l'extraction d'information.

Techniques statistiques pour le NLP

Introduction aux techniques statistiques pour le NLP

Les techniques statistiques en NLP reposent sur des modèles probabilistes pour analyser et traiter les textes. Elles permettent d'extraire des informations pertinentes en se basant sur les fréquences de mots et les relations entre eux.

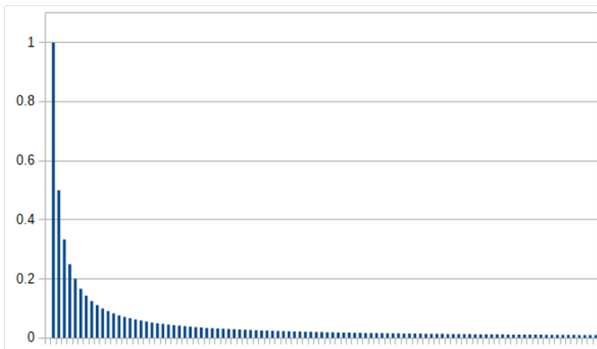
Principes de base :

- Les modèles probabilistes estiment les probabilités des séquences de mots et des événements linguistiques.
- Les algorithmes apprennent à partir des données d'entraînement, sans règles explicites définies à l'avance.

Ces techniques ont permis de remplacer les approches basées sur des règles linguistiques par des modèles plus flexibles et adaptatifs.

Principe des techniques statistiques pour le NLP

- Un document est caractérisé par la distribution des mots qui le composent.
- La distribution des mots obéit généralement à une loi de puissance (ou de Zipf). La majorité du langage que nous utilisons ordinairement est composé d'un nombre très limité de mots.



Introduction aux modèles n-grams

Les modèles **n-grams** sont une des premières techniques statistiques utilisées pour modéliser les séquences de mots dans le langage naturel.

Principe :

- Un modèle n-gram prédit la probabilité d'un mot en fonction des $n - 1$ mots précédents.
- n -gram peut être : un unigram (1 mot), bigram (2 mots), trigram (3 mots), etc.

Exemple (2-gram - bigram) :

- Phrase : " Je mange une pomme."
- Bigrammes : $\{(" Je", " mange"), (" mange", " une"), (" une", " pomme")\}$

Calcul de la probabilité avec un modèle n-grams

Les modèles n-grams estiment la **probabilité** d'une phrase en se basant sur les probabilités conditionnelles des mots.

Probabilité d'une phrase :

$$P(w_1, w_2, w_3, \dots, w_n) \approx P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_2) \cdot \dots \cdot P(w_n|w_{n-1})$$

Exemple (2-gram) :

- Phrase : " Je mange une pomme."
- Probabilité de la phrase :

$$P(\text{" Je"}) \cdot P(\text{" mange" | " Je"}) \cdot P(\text{" une" | " mange"}) \cdot P(\text{" pomme" | " une"})$$

Applications et limites des modèles n-grams

Applications des n-grams :

- **Modélisation du langage** : Utilisés pour estimer la probabilité des séquences de mots dans des applications comme la reconnaissance vocale ou la correction orthographique.
- **Traduction automatique** : Les n-grams permettent de prédire les séquences de mots dans les langues cibles.
- **Génération de texte** : Génération automatique de phrases probables à partir de contextes.

Limites des n-grams :

- **Explosion combinatoire** : La taille du modèle croît exponentiellement avec n , rendant le stockage des probabilités complexe pour de grandes valeurs de n .
- **Manque de contexte à long terme** : Les n-grams ne capturent que les relations entre les n mots précédents, ignorant les dépendances à longue distance.

Introduction à TF-IDF

TF-IDF (Term Frequency - Inverse Document Frequency) est une méthode utilisée pour évaluer l'importance d'un mot dans un document, relativement à l'ensemble d'un corpus.

Objectif :

- Mesurer l'importance d'un mot dans un document tout en réduisant l'importance des mots trop fréquents dans le corpus.

Applications :

- Extraction de caractéristiques textuelles pour des tâches de classification.
- Recherche d'information et moteurs de recherche (identifier les mots-clés).
- Résumé automatique de textes.

Term Frequency (TF)

La **fréquence d'un terme (TF)** mesure combien de fois un mot apparaît dans un document. Plus un mot apparaît souvent dans le document, plus sa fréquence est élevée.

Formule :

$$TF(t, d) = \frac{\text{Nombre d'occurrences du terme } t \text{ dans le document } d}{\text{Nombre total de mots dans le document } d}$$

Exemple :

- Document : "Le chat mange une souris. Le chat dort."
- Nombre d'occurrences du mot "chat" = 2
- Nombre total de mots dans le document = 7
- Donc :

$$TF(\text{"chat"}, d) = \frac{2}{7} = 0.285$$

Inverse Document Frequency (IDF)

L'**Inverse Document Frequency (IDF)** mesure l'importance d'un mot dans le corpus. Les mots qui apparaissent dans de nombreux documents ont une faible importance, car ils sont trop communs.

Formule :

$$IDF(t) = \log \left(\frac{\text{Nombre total de documents dans le corpus}}{\text{Nombre de documents contenant le terme } t} \right)$$

Exemple :

- Supposons que le mot "chat" apparaisse dans 100 documents d'un corpus de 1000 documents.
- Le nombre total de documents dans le corpus = 1000
- Le nombre de documents contenant le mot "chat" = 100
- Donc :

$$IDF(\text{"chat"}) = \log \left(\frac{1000}{100} \right) = \log(10) = 1$$

Calcul de TF-IDF

La valeur **TF-IDF** combine la fréquence d'un terme dans un document (TF) et l'inverse de sa fréquence dans le corpus (IDF).

Formule :

$$TF-IDF(t, d) = TF(t, d) \times IDF(t)$$

Exemple :

- Document : "Le chat mange une souris."
- $TF(\text{"chat"}, d) = 0.285$, $IDF(\text{"chat"}) = 1$
- $TF-IDF(\text{"chat"}, d) = 0.285 \times 1 = 0.285$

Exemple de matrice TF-IDF

Documents :

- Document 1 (D1) : "le chat dort."
- Document 2 (D2) : "le chien dort."
- Document 3 (D3) : "le chat mange."
- Document 4 (D4) : "le chien mange."

Matrice TF-IDF :

Matrice TF-IDF				
	D1	D2	D3	D4
chat	0.231	0.0	0.231	0.0
chien	0.0	0.231	0.0	0.231
dort	0.231	0.231	0.0	0.0
mange	0.0	0.0	0.231	0.231

Représentations distribuées

Introduction aux Embeddings

● Qu'est-ce qu'un Embedding ?

- Un embedding est une représentation vectorielle d'un mot qui capture le contexte du mot dans un document, des relations sémantiques et syntaxiques avec d'autres mots.
- Ils transforment des mots en vecteurs de nombres pour que les algorithmes de machine learning puissent les traiter efficacement.

● Pourquoi sont-ils importants ?

- Les embeddings permettent de capturer non seulement l'identité d'un mot mais aussi ses aspects sémantiques et contextuels.
- Ils facilitent des tâches telles que la classification de texte, la traduction automatique, et la détection des sentiments.

● Évolution des embeddings

- Historiquement, les mots étaient représentés comme des indices ou des vecteurs one-hot, où chaque mot est indépendant des autres.
- Les embeddings modernes, tels que Word2Vec et GloVe, représentent les mots dans des espaces vectoriels continus où les mots de sens similaires sont proches les uns des autres.

Différence entre One-hot Encoding et Embeddings

● One-hot Encoding

- Chaque mot est représenté par un vecteur avec un '1' dans la position qui lui est propre et des '0' partout ailleurs. Ce type de représentation est simple mais très inefficace en termes d'espace et ne capture pas les relations entre les mots.
- Exemple : pour un vocabulaire de dimension 100 000 :

Véhicule = $[0, 0, 1, 0, 0, 0, 0, 0, 0, \dots, 0, 0]$

Voiture = $[0, 0, 0, 0, 0, 1, 0, 0, 0, \dots, 0, 0]$

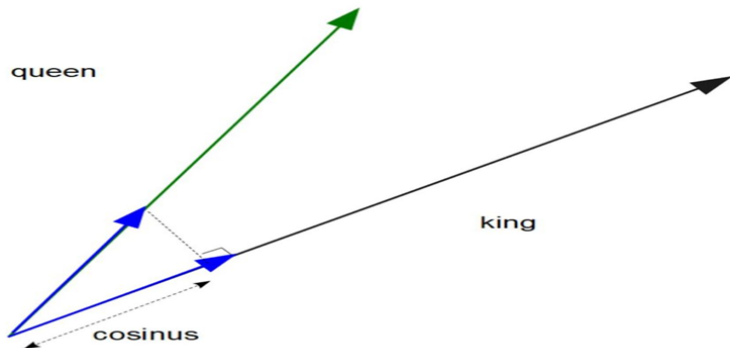
Cette représentation ne permet pas de prendre en compte la dimension sémantique

● Embeddings

- Les embeddings représentent les mots comme des vecteurs denses de nombres flottants (généralement entre 50 et 300 dimensions).
- Cette représentation est beaucoup plus riche et peut capturer des relations complexes entre les mots, comme la similarité sémantique.

Similarité sémantique

Nous sommes intéressés par des représentations de mots qui capturent la distance sémantique, sous forme de produit scalaire par exemple (ou distance cosinus).



Représentations distribuées : Principes

“You shall know a word by the company it keeps”

— J.R. Firth (1957)

Les mots sont similaires s'ils apparaissent fréquemment dans le même contexte.

- Il conduit son *véhicule* pour rentrer à la maison.
- Il conduit sa *voiture* pour rentrer chez lui.

Construction d'une représentation distribuée

Considérons le corpus suivant à titre d'exemple :

cnn in crop analysis

cnn and svm are widely used.

linear_regression performed along with svm

linear_regression for crop and farm

svm being used for farm monitoring

Do cnn, svm and linear_regression appear in the same context?

Le vocabulaire est alors :

[cnn, in, crop, analysis, and, svm, are, widely, used,
linear_regression, performed, along, with, for, farm, being,
monitoring, do, appear, the, same, context]

$$\dim(\text{vocabulaire}) = 22$$

Similarité sémantique

La matrice de co-occurrence représente la fréquence à laquelle les mots apparaissent ensemble deux à deux.

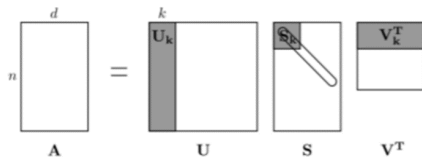
	cnn in crop ...																					
cnn	[0,	2,	1,	1,	1,	2,	1,	1,	1,	1,	0,	0,	0,	0,	0,	0,	1,	1,	1,	1,	1],
in	[2,	0,	1,	1,	0,	1,	0,	0,	0,	1,	0,	0,	0,	0,	0,	0,	1,	1,	1,	1,	1],
crop	[1,	1,	0,	1,	1,	0,	0,	0,	0,	1,	0,	0,	0,	1,	0,	0,	0,	0,	0,	0,	0],
...	[1,	1,	1,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0],
	[1,	0,	1,	0,	0,	1,	1,	1,	1,	1,	0,	0,	0,	1,	1,	0,	0,	0,	0,	0,	0],
	[2,	1,	0,	0,	1,	0,	1,	1,	2,	2,	1,	1,	1,	1,	1,	1,	1,	1,	1,	1,	1],
	[1,	0,	0,	0,	1,	1,	0,	1,	1,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0],
	[1,	0,	0,	0,	1,	1,	1,	0,	1,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0],
	[1,	0,	0,	0,	1,	2,	1,	1,	0,	0,	0,	0,	0,	1,	1,	1,	1,	0,	0,	0,	0],
	[1,	1,	1,	0,	1,	2,	0,	0,	0,	0,	1,	1,	1,	1,	0,	0,	1,	1,	1,	1,	1],
	[0,	0,	0,	0,	0,	1,	0,	0,	0,	1,	0,	1,	1,	0,	0,	0,	0,	0,	0,	0,	0],
	[0,	0,	0,	0,	0,	1,	0,	0,	0,	1,	1,	0,	1,	0,	0,	0,	0,	0,	0,	0,	0],
	[0,	0,	0,	0,	0,	1,	0,	0,	0,	1,	1,	1,	0,	0,	0,	0,	0,	0,	0,	0,	0],
	[0,	0,	1,	0,	1,	1,	0,	0,	1,	1,	0,	0,	0,	0,	2,	1,	1,	0,	0,	0,	0],
	[0,	0,	1,	0,	1,	1,	0,	0,	1,	1,	0,	0,	0,	2,	0,	1,	1,	0,	0,	0,	0],
	[0,	0,	0,	0,	0,	1,	0,	0,	1,	0,	0,	0,	0,	1,	1,	0,	1,	0,	0,	0,	0],
	[0,	0,	0,	0,	0,	1,	0,	0,	1,	0,	0,	0,	0,	1,	1,	1,	0,	0,	0,	0,	0],
	[1,	1,	0,	0,	0,	1,	0,	0,	0,	1,	0,	0,	0,	0,	0,	0,	0,	1,	1,	1,	1],
	[1,	1,	0,	0,	0,	1,	0,	0,	0,	1,	0,	0,	0,	0,	0,	0,	1,	0,	1,	1,	1],
	[1,	1,	0,	0,	0,	1,	0,	0,	0,	1,	0,	0,	0,	0,	0,	0,	1,	1,	0,	1,	1],
	[1,	1,	0,	0,	0,	1,	0,	0,	0,	1,	0,	0,	0,	0,	0,	0,	1,	1,	1,	0,	1],
	[1,	1,	0,	0,	0,	1,	0,	0,	0,	1,	0,	0,	0,	0,	0,	0,	1,	1,	1,	1,	0]

Réduction de la dimension

La décomposition en valeurs singulières est une technique permettant de réduire la dimension des données (similaire à l'ACP).

Étant donné une matrice $A \in \mathbb{R}^{n \times d}$

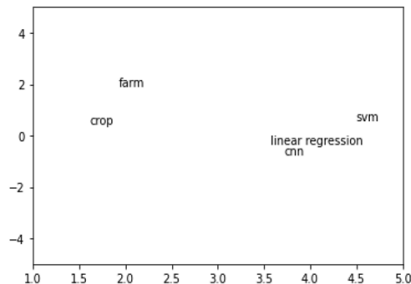
$$A = UDV^T \quad \text{où} \quad U \in \mathbb{R}^{n \times r}, D \in \mathbb{R}^{r \times r}, V \in \mathbb{R}^{d \times r}.$$



U est la matrice contenant les représentations (vecteurs de mots)

Vsualisation des représentations distribuées

```
cnn [ 3.72113518, -0.73233585],  
ln [ 2.7940387 , -1.40864784],  
crop [ 1.61178082,  0.44786021],  
... [ 0.77784994, -0.31230389],  
[ 2.12245048,  1.29571556],  
[ 4.49293777,  0.58090417],  
[ 1.33312748,  0.66758743],  
[ 1.33312748,  0.66758743],  
[ 2.25882809,  1.80738544],  
[ 3.56593605, -0.31006976],  
[ 0.95394179,  0.079159 ],  
[ 0.95394179,  0.079159 ],  
[ 0.95394179,  0.079159 ],  
[ 1.92921553,  1.94783497],  
[ 1.92921553,  1.94783497],  
[ 1.12301312,  1.42126096],  
[ 1.12301312,  1.42126096],  
[ 2.26025282, -1.31571175],  
[ 2.26025282, -1.31571175],  
[ 2.26025282, -1.31571175],  
[ 2.26025282, -1.31571175],  
[ 2.26025282, -1.31571175]
```



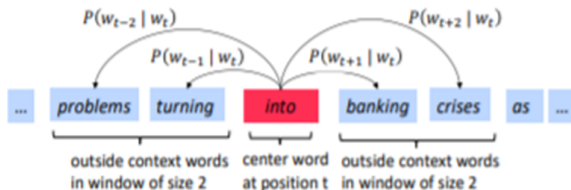
Word2Vec : Principes

- **Principes de base :**

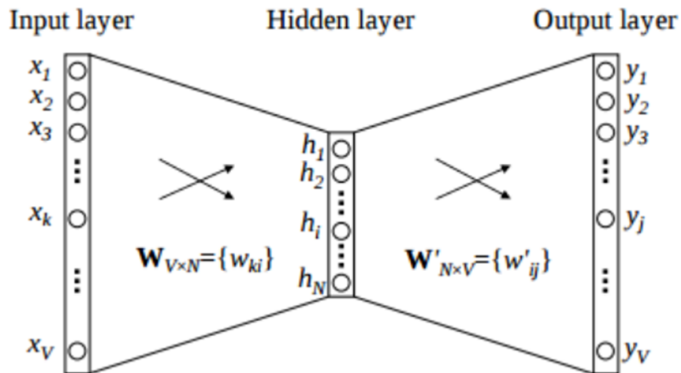
- Word2Vec est basé sur l'hypothèse distributionnelle : les mots qui apparaissent dans des contextes similaires ont des significations similaires.
- Utilise un réseau de neurones peu profond pour apprendre des embeddings de mots à partir de grands corpus.

- **Deux architectures principales :**

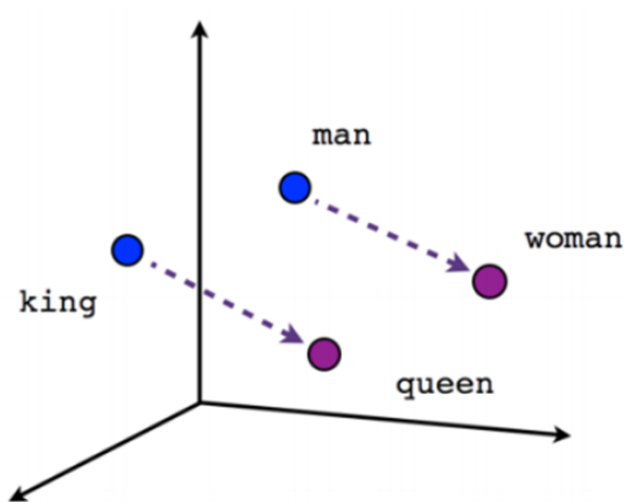
- 1 *Continuous Bag of Words (CBOW)* : Prédit le mot cible à partir du contexte.
- 2 *Skip-Gram* : Prédit le contexte à partir du mot cible.



Architecture du modèle CBOW



Représentations Word2Vec



Applications pratiques de Word2Vec

- **Analogie de mots :**

- Word2Vec est célèbre pour capturer des relations complexes, comme "homme est à femme ce que roi est à reine".
- Permet de résoudre des analogies en utilisant des opérations arithmétiques simples sur les vecteurs de mots.

- **Clustering sémantique :**

- Les embeddings peuvent être utilisés pour regrouper des mots sémantiquement similaires, facilitant l'analyse de textes volumineux.

- **Amélioration des systèmes de recommandation :**

- Les vecteurs de mots de Word2Vec peuvent être utilisés pour améliorer la précision des recommandations en comprenant mieux les préférences des utilisateurs.

Autres représentations distribuées

Il existe d'autres représentations distribuées :

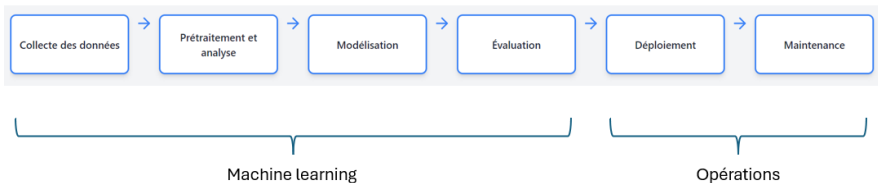
- Glove (2014) : proposé par une équipe de Stanford, il combine à la fois les techniques modernes de deep learning et les techniques statistiques (co-occurrences entre les mots). Il permet d'avoir des représentations des mots plus globales que Word2Vec.
- Fasttext (à partir de 2016) : la librairie a été créée par Facebook. Le modèle est entraîné sur des subwords (n-grams de caractères). Il est ainsi plus efficace pour traiter les mots inconnus (out of vocabulary).
- Représentations contextuelles (Transformers), etc.

Industrialisation des applications data et MLOps

Définition

Le MLOps est un ensemble de pratiques et d'outils mis en oeuvre pour automatiser et optimiser le déploiement, la gestion et la maintenance des modèles de machine learning en production.

Cycle de vie d'un projet de machine learning



Mais ce processus est en réalité très itératif.

Pourquoi le MLOps ?

Les différents problèmes que l'on peut rencontrer à l'interface entre la phase machine learning et la phase opérations.

- Problèmes de communication
- Problèmes techniques
- Problèmes de surveillance et de maintenance des modèles
- Problèmes de gestion des données
- Problèmes liés au déploiement et à l'automatisation
- Problèmes de collaboration organisationnelle
- Problèmes liés à la sécurité et à la conformité
- Problèmes de compétences et de formation

Différence de nature entre les projets Data et Dev

Projets data

- Nature exploratoire, résultats incertains
- Estimation des délais difficile
- "Produit fini" flou, amélioration continue
- Attentes parfois irréalistes
- Métriques de succès statistiques
- Cycle de vie : mises à jour fréquentes
- Confort avec l'ambiguïté, exploration

Projets dev

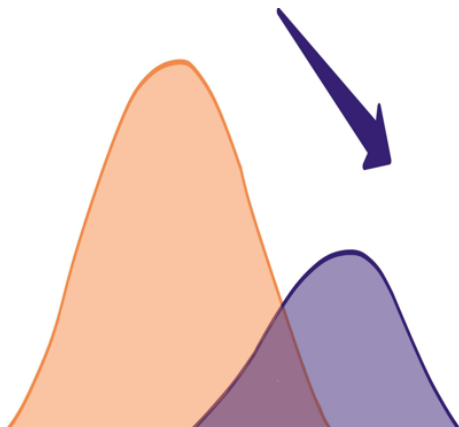
- Nature déterministe, spécifications claires
- Estimations plus précises
- "Produit fini" défini par les spécifications
- Attentes mieux alignées techniquement
- Métriques liées aux fonctionnalités
- Cycle de vie plus traditionnel
- Tendance à réduire l'incertitude

0. Problème de data drift

Le data drift est le phénomène de changement dans la distribution des données.

Il apparaît d'une manière fréquente dans les modèles mis en production. Il peut être dû à plusieurs facteurs :

- Différence entre les données de test et les données en production.
- Changement de l'environnement (avant et après le Covid par exemple).
- Changement dans le comportement des utilisateurs ou clients.
- Effets de l'algorithme lui-même sur les utilisateurs.
- Etc.



1. Problèmes de communication

- **Terminologie et langage différents :**

Les data scientists et les développeurs parlent souvent des "mêmes" concepts avec des terminologies différentes. Par exemple, un développeur peut parler d'API ou de scalabilité, tandis qu'un data scientist peut parler de modèles prédictifs et de dérive des données. Cette différence peut créer des incompréhensions.

- **Objectifs mal alignés :**

Les priorités des deux équipes peuvent différer, avec les data scientists axés sur la précision des modèles et les développeurs concentrés sur la stabilité et l'optimisation du code en production. Cette divergence d'objectifs peut entraîner des tensions si les équipes ne communiquent pas clairement leurs attentes respectives.

- **Manque de documentation :**

Les data scientists peuvent fournir des modèles ou des scripts sans documentation complète, ce qui rend difficile pour les développeurs de comprendre le contexte ou les paramètres du modèle lors du déploiement en production.

2. Problèmes techniques

- **Prototypage vs production :**

Le code écrit dans les notebooks n'est pas directement utilisable en production. Il manque souvent de structure, de gestion des erreurs et de tests nécessaires pour être fiable en environnement de production.

- **Scalabilité et performance :**

Les modèles et pipelines développés par les data scientists ne sont pas toujours adaptés pour une exécution à grande échelle en production. Les problèmes de performance (latence, mémoire) peuvent survenir lors du déploiement à grande échelle.

- **Compatibilité des outils :**

Les outils utilisés en exploration (Pandas, scikit-learn) ne sont pas toujours compatibles avec les environnements de production (Kubernetes, Docker, TensorFlow Serving), nécessitant des ajustements ou réécritures du code.

3. Problèmes de gestion des données

- **Qualité et gouvernance des données :**

Les data scientists ont besoin de données propres et de haute qualité, mais si la gouvernance des données n'est pas solide, cela peut entraîner des modèles biaisés ou des erreurs.

- **Accès et gestion des données :**

Il peut y avoir des problèmes de gestion des droits d'accès aux données entre les équipes, ce qui ralentit les projets ou complique les échanges de données entre les équipes data et dev.

- **Versioning des données :**

Les modèles peuvent être entraînés sur une version des données et mis en production avec une autre, ce qui entraîne des incohérences. La gestion des versions des données n'est pas toujours bien synchronisée entre les équipes.

4. Problèmes liés au déploiement et à l'automatisation

- **Déploiement manuel :**

Les modèles sont souvent développés dans des environnements isolés et il peut être difficile de les déployer manuellement en production sans automatisation.

- **MLOps immature :**

Si les pratiques MLOps (intégration et déploiement continus pour les modèles) ne sont pas mises en place, il peut être difficile de maintenir les modèles et de les réentraîner régulièrement en production.

- **Maintenance des modèles :**

Une fois en production, les modèles doivent être surveillés et mis à jour. Sans une automatisation du monitoring et de la reformation, les modèles peuvent devenir obsolètes et imprécis.

5. Problèmes de collaboration organisationnelle

- **Silos organisationnels :**

Les équipes data et dev travaillent souvent dans des silos, ce qui entraîne une mauvaise communication et une collaboration limitée sur les projets communs.

- **Problèmes de priorisation :**

Les équipes data peuvent prioriser des améliorations de modèles, tandis que les équipes dev se concentrent sur la stabilité du système, entraînant des désaccords sur la priorisation des projets.

- **Propriété et responsabilité :**

Il peut être difficile de définir qui est responsable de la performance continue des modèles en production. Les développeurs s'occupent souvent de l'infrastructure, mais les data scientists sont responsables de la précision des modèles.

6. Problèmes liés à la sécurité et à la conformité

- **Protection des données :**

Les data scientists et les développeurs doivent gérer des données sensibles, ce qui impose des contraintes de sécurité pour éviter les violations de la confidentialité (ex : RGPD).

- **Gestion des accès :**

Il est important de s'assurer que seuls les membres autorisés peuvent accéder aux données critiques. Une mauvaise gestion des accès peut entraîner des fuites de données ou des violations de conformité.

Les différentes organisations des équipes Data et Dev

Dans le développement et le déploiement des projets de Machine Learning, plusieurs types d'organisation sont possibles pour maximiser la collaboration et l'efficacité entre les équipes de data science, de développement et d'ingénierie.

- ➊ **Approche en silos (traditionnelle)** : Les équipes travaillent de manière indépendante, avec transfert de responsabilités des data scientists aux devs.
- ➋ **Approche collaborative (cross-fonctionnelle)** : Les équipes collaborent dès le début du projet et tout au long du cycle de vie du modèle.
- ➌ **Approche DevOps/MLOps** : Automatisation des processus d'entraînement, de déploiement et de surveillance des modèles à travers des pipelines CI/CD.
- ➍ **Équipe centralisée (Center of Excellence)** : Une équipe dédiée centralise les compétences et les processus pour tous les projets de ML de l'entreprise.
- ➎ **Squads dédiés ou équipes produits (Product Teams)** : Des équipes autonomes et multidisciplinaires sont responsables du développement de produits spécifiques intégrant des modèles ML.
- ➏ **Approche par chapitre et guildes (Spotify Model)** : Une combinaison d'équipes autonomes (squads) et de groupes d'experts (chapitres) partageant des pratiques et des standards communs.

Approche en silos (traditionnelle)

Dans cette organisation, les équipes de data science et de développement fonctionnent de manière indépendante. Les data scientists développent leurs modèles en isolation, puis passent le modèle aux développeurs pour le déploiement.

Avantages :

- Chaque équipe se concentre sur son expertise principale.
- Les processus peuvent être optimisés indépendamment pour chaque équipe.

Inconvénients :

- Manque de communication, ce qui peut entraîner des problèmes d'intégration en production (incompatibilité de l'infrastructure, erreurs dans le pipeline de données).
- Les retours d'expérience des développeurs arrivent trop tard, après la phase de conception du modèle.
- Problèmes de duplication d'efforts et de perte de temps lors des itérations.

Quand l'utiliser :

- Pour des projets simples et bien définis où les étapes sont clairement séparées.

Approche collaborative (cross-fonctionnelle)

Les équipes de data science et de développement travaillent ensemble dès le début du projet. Les développeurs participent aux étapes de conception des modèles, et les data scientists collaborent pendant le développement des systèmes de production.

Avantages :

- Meilleure communication entre les équipes.
- Les problèmes d'intégration sont détectés plus tôt.
- Les modèles sont conçus pour être déployés plus rapidement, avec moins de risques d'incompatibilités techniques.

Inconvénients :

- Plus de coordination et de gestion de projet nécessaire.
- Peut ralentir le développement initial en raison des échanges constants.

Quand l'utiliser :

- Pour des projets complexes où l'intégration du modèle dans le système de production est cruciale et nécessite une optimisation fine.
- Lorsque des itérations fréquentes entre le développement du modèle et le déploiement sont nécessaires.

Approche DevOps/MLOps

Cette organisation intègre les principes du DevOps dans la data science sous le terme de MLOps. L'idée est de créer des pipelines automatisés pour entraîner, tester, déployer, surveiller et maintenir les modèles de machine learning.

Avantages :

- Automatisation des processus, ce qui réduit les erreurs humaines.
- Déploiement plus rapide et plus fréquent des modèles en production.
- Les équipes de data science et de développement partagent les mêmes outils et processus d'intégration continue (CI/CD).
- Meilleure gestion des versions de modèles et surveillance en temps réel pour réentraîner si nécessaire.

Inconvénients :

- Nécessite une mise en place initiale complexe des outils et des processus.
- Peut nécessiter des compétences supplémentaires pour les équipes data.

Quand l'utiliser :

- Pour des projets à long terme où de nombreux modèles sont mis à jour fréquemment.
- Dans les environnements où la qualité, la performance et la réactivité du modèle en production sont critiques

Équipe centralisée (Center of Excellence)

Dans cette organisation, une équipe centralisée est responsable à la fois du développement des modèles de machine learning et de leur déploiement. Elle regroupe des compétences en data science, en développement et en MLOps.

Avantages :

- Unité dans la gestion des processus et des technologies.
- Facilite la collaboration entre les disciplines.
- Meilleure standardisation des pratiques et des outils utilisés dans toute l'entreprise.

Inconvénients :

- Peut manquer de flexibilité si les besoins spécifiques des équipes sont très variés.
- Risque d'une surcharge de travail pour l'équipe centralisée si l'organisation est trop large.

Quand l'utiliser :

- Pour des entreprises qui veulent centraliser l'expertise en machine learning et l'appliquer à plusieurs départements.

Squads dédiés ou équipes produits (Product Teams)

Les équipes de data science et de développement sont intégrées au sein de petites équipes produits, appelées "squads". Chaque squad est autonome et responsable d'un produit spécifique ou d'une fonctionnalité (ex. : un modèle de recommandation, un algorithme de détection de fraude).

Avantages :

- Les équipes sont alignées sur un objectif commun : le produit.
- Collaboration directe et quotidienne entre data scientists, développeurs, et autres membres du produit.
- Grande réactivité et agilité pour itérer rapidement sur le produit.

Inconvénients :

- Risque de duplication d'efforts si chaque squad développe des solutions similaires sans coordination.
- Peut être difficile à mettre en place dans des grandes organisations avec des équipes réparties.

Quand l'utiliser :

- Dans des organisations avec une forte culture d'agilité ou des entreprises tech où la personnalisation de chaque produit est clé.

Approche par chapitre et guilde

Inspirée du modèle d'organisation de Spotify, cette approche combine des squads (équipes produits) avec des chapitres (groupes d'experts dans un domaine, comme la data science ou le développement) et des guildes (groupes transversaux autour de pratiques communes).

Avantages :

- Maintient une expertise forte au sein des équipes tout en encourageant la collaboration inter-équipes.
- Favorise l'innovation et l'échange de bonnes pratiques.
- Flexibilité pour adapter les équipes aux besoins du projet.

Inconvénients :

- Complexité organisationnelle si mal gérée.
- Peut entraîner des conflits de priorités entre les squads et les chapitres.

Quand l'utiliser :

- Dans des entreprises de taille moyenne à grande qui veulent favoriser l'innovation tout en maintenant une cohérence entre les équipes.

Quels problèmes le MLOps résout-il ?

- **Déploiement** : Les modèles développés en laboratoire ne sont souvent pas directement utilisables en production en raison de différences d'environnement, d'outils, et de processus.
- **Répétabilité** : Les processus manuels dans le développement et le déploiement des modèles sont souvent source d'erreurs, de lenteurs, et de difficultés à reproduire les résultats.
- **Cycle de vie** : Une fois déployés, les modèles ML peuvent perdre en performance au fil du temps (en raison de la dérive des données, par exemple). Sans une gestion appropriée, cela peut entraîner des décisions erronées ou des inefficacités.
- **Collaboration** : Les équipes de data science, d'ingénierie, et d'opérations travaillent souvent en silos, ce qui complique la collaboration et entraîne des frictions lors du déploiement et de la gestion des modèles en production.
- **Scalabilité** : Le déploiement de modèles ML à grande échelle peut être complexe, surtout lorsque plusieurs modèles doivent être gérés, surveillés, et mis à jour simultanément.

Collaboration

Data scientist



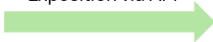
Passage du code



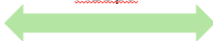
Opérations



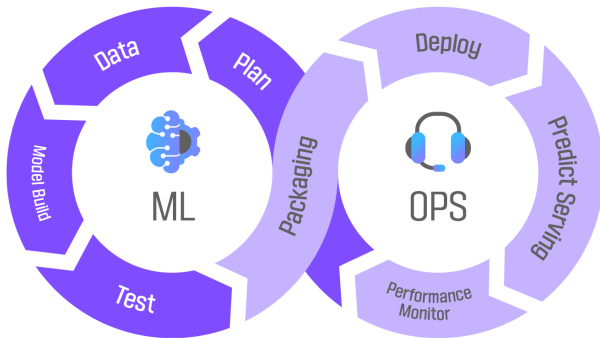
Exposition via API



MLOps



Machine learning + Opérations



Principes du MLOps

- Automatisation du cycle de vie des modèles.
- Intégration continue et déploiement continu (CI/CD).
- Collaboration entre les équipes ML (data scientists) et les équipes opérations (Data engineer, ML engineers, Développeurs).
- Monitoring des modèles et des données en production.
- Versioning et traçabilité du code et des données.
- Optimisation et gestion des ressources, notamment sur le Cloud.

Dans la pratique...

Les approches MLOps peuvent être différentes d'une entreprise à une autre et dépendent de plusieurs facteurs :

- Infrastructure de l'entreprise.
- Organisation de l'entreprise.
- Ressources disponibles en termes de compétences.
- Projets mis en oeuvre.
- Maturité de l'entreprise du point de vue du ML.
- Etc. etc.

Outils

Versioning



GitHub



GitLab



Data version Control

Virtualisation



Docker



Podman



Kubernetes

CI/CD



Jenkins



GitHub Actions



Pytest

APIs



Flask



FastAPI



Postman

Cloud



AWS SageMaker



GCP VertexAI



Azure ML

Plate-formes end-to-end



Organisation : principaux métiers de l'IA

Les métiers de l'IA sont en constante évolution et ont subi une évolution assez importantes ces dernières années. On peut distinguer cependant les métiers suivants :

- **Data scientist** : il est au cœur du projet IA. Son principal rôle consiste à traduire le besoin métier en une problématique technique pour élaborer un modèle de ML ou autre, qu'il doit tester et valider avec les métiers. Il est donc normalement amené à travailler étroitement avec les métiers.
- **Machine Learning Engineer** : son rôle consiste à adapter le code développé par le data scientist pour le déployer, souvent sur une infrastructure Cloud (AWS, GCP, Azure, etc.). Il doit ensuite veiller au bon fonctionnement de l'application.
- **Data Engineer** : il est responsable de l'infrastructure liée au projet (pipelines de données, etc.).
- **Data analyst** : son rôle est d'analyser les données pour répondre à des questions métier.
- **Data architecte** : il conçoit l'architecture de la solution qui intègre éventuellement de l'IA (bases de données, back-end, front-end, services Cloud, etc.), notamment quand celle-ci est complexe.