Machine Learning

N° de la lecture individuelle :	Remédiation
Semestre	4
Étudiant	Térence LAURENT, 803_1F

Support théorique

La recherche se base fondamentalement sur le document présenté ci-dessous ainsi que de la recherche. Des apports, de l'aide à la construction des exemples, et des compréhensions ont également réalisés avec ChatGPT.

Document

Cette lecture individuelle se base sur la formation OpenClassrooms intitulé « Initiez-vous au Machine Learning ». La formation est disponible gratuitement au lien suivant :
« https://openclassrooms.com/fr/courses/8063076-initiez-vous-au-machine-learning »

Les images présentes dans ce document proviennent de la formation.

Ce document est destiné à une utilisation interne au sein de la Digital Team Academy.

Table des matières

Résumé de l'introduction au Machine Learning

Le **Machine Learning** (ML), ou apprentissage automatique, est une sous-discipline essentielle de l'intelligence artificielle (IA). Son but principal est de **prédire** des résultats en se basant sur des **algorithmes** appliqués à des **jeux de données**. L'origine de l'IA remonte au **Perceptron**, une machine conçue en 1957 pour la classification d'images, marquant le début des premières avancées en IA.

Depuis, le domaine a connu plusieurs phases de développement et de ralentissement. Cependant, grâce aux progrès technologiques récents, notamment en termes de puissance de calcul et de stockage, le Machine Learning a pris une place centrale dans notre quotidien. Aujourd'hui, des outils tels que **scikit-learn** (une bibliothèque de ML) et des plateformes comme **Google Colab** facilitent son apprentissage et son usage à grande échelle.

Les objectifs pédagogiques du cours sont de :

- Comprendre les grands principes du ML.
- Manipuler les fonctions de base des modèles prédictifs.
- Savoir transformer les jeux de données.
- Optimiser les performances des modèles.

En résumé, le Machine Learning est le moteur qui permet à l'IA de réaliser des prédictions efficaces grâce à l'analyse de données.

Modélisation Statistique vs. Modélisation Prédictive en Machine Learning

Dans le domaine du Machine Learning, deux approches distinctes sont employées pour analyser les données : **la modélisation statistique** et **la modélisation prédictive**. Bien qu'elles visent à répondre à des questions relatives aux données, leurs objectifs et méthodes diffèrent.

Exemple d'application :

Imaginons que vous soyez responsable des données d'une plateforme en ligne où le succès repose sur le nombre d'abonnements souscrits. On vous pose deux questions :

1. **Quel est le profil des utilisateurs qui s'abonnent ?** (Approche statistique)
2. **Comment prédire si un nouvel utilisateur va s'abonner ?** (Approche prédictive)

Ces deux questions explorent l'acte d'abonnement, mais sous deux angles différents : **comprendre** et **prédire**. Cela mène à deux types d'approches : la **modélisation statistique** et la **modélisation prédictive**.

La Modélisation Statistique

La première question cherche à **comprendre** les caractéristiques des utilisateurs qui s'abonnent. Ici, la **modélisation statistique** est utilisée pour analyser et interpréter la relation entre les différentes variables (âge, centre d'intérêt, comportement, etc.) et l'acte d'abonnement.

L'objectif est donc de révéler des **corrélations** ou des **dynamiques** entre les données et d'évaluer leur validité à travers des **tests d'hypothèses** et des **modèles mathématiques**. On cherche à comprendre les raisons derrière l'abonnement d'un utilisateur, ce qui est essentiel pour des décisions stratégiques.

Exemple : Les utilisateurs ayant entre 25 et 35 ans, aimant le contenu éducatif et visitant le site plusieurs fois par semaine, s'abonnent plus fréquemment. Cette analyse permettrait à l'entreprise de cibler cette catégorie d'utilisateurs dans ses campagnes marketing.

La Modélisation Prédictive

La deuxième question, en revanche, se focalise sur **prédire** l'abonnement d'un nouvel utilisateur. Ici, la **modélisation prédictive**, cœur du Machine Learning, est utilisée pour créer un modèle capable de **généraliser** à partir des données d'entraînement afin de fournir des **prédictions précises**.

Cette approche met l'accent sur la **performance** du modèle, en mesurant sa qualité à l'aide d'une **métrique** (comme la précision, le rappel, ou l'AUC). Le but est de maximiser sa capacité à **extrapoler** à partir des données déjà observées, en faisant abstraction de la compréhension des processus internes, souvent trop complexes à interpréter.

Le modèle se comporte comme une **"boîte noire"** où l'essentiel est d'obtenir des prédictions fiables, même si la logique interne n'est pas entièrement visible.

Exemple : En analysant les caractéristiques d'un nouvel utilisateur, le modèle de Machine Learning pourrait prédire avec une forte précision s'il est susceptible de s'abonner ou non, sans nécessairement expliquer pourquoi.

Résumé

- **Modélisation statistique** : se concentre sur **l'analyse et l'interprétation** des données pour comprendre des relations causales ou explicatives.
- **Modélisation prédictive** : se concentre sur la **prédiction** des résultats en mettant l'accent sur la qualité et la robustesse des modèles, sans se soucier d'expliquer la logique des prédictions.

Ces deux approches sont complémentaires : la statistique aide à comprendre, tandis que la prédiction permet d'agir avec efficacité.

	Modélisation statistique	Modélisation prédictive (= Machine Learning)
Objectif	Expliquer, analyser	Prédire
Focus	Fiabilité des conclusions : tests d'hypothèses et intervalles de confiance	Performance, résilience et robustesse du modèle
Utilisation des données	Tous les échantillons	Une partie des échantillons est réservée à l'évaluation des performances du modèle
Librairies Python	statsmodel	scikit-learn, PyTorch, TensorFlow

Machine Learning Classique : Jeux de Données et Entraînement des Modèles

Le Machine Learning **classique** se base principalement sur des **jeux de données tabulaires**. Ces données sont généralement organisées sous forme de tableaux, où chaque colonne représente une variable et chaque ligne un échantillon. Ces jeux de données peuvent être stockés dans des fichiers comme **CSV**, **Excel**, **JSON**, ou provenir directement de bases de données **SQL** ou **NoSQL**.

Contrairement au **deep learning** qui est adapté aux données massives et complexes (audio, vidéo, images, IA générative), le Machine Learning classique est utilisé pour des jeux de données plus légers, généralement de moins d'un gigabit. Cela le rend particulièrement efficace pour les applications pratiques avec des données structurées.

Le Jeu de Données : Fondation du Machine Learning

Dans le Machine Learning, les **données** sont au cœur de l'apprentissage. Sans un jeu de données, il est impossible d'entraîner un modèle. L'idée est d'utiliser ces données pour **entraîner un modèle prédictif**, capable de faire des prédictions sur des données nouvelles.

Exemple : Feuille de Calcul Tabulaire

Pensez à un jeu de données comme une feuille de calcul, par exemple dans Google Spreadsheet ou Excel :

- **Colonnes** : Représentent les **variables** ou **caractéristiques** (par exemple, âge, taille, poids).

- **Lignes** : Correspondent aux **échantillons** ou **observations** (par exemple, les caractéristiques de chaque collégien dans une classe).

Variables Cibles et Prédictrices

Il est essentiel de distinguer deux types de variables :

- La **variable cible** (ou variable dépendante) : c'est ce que l'on cherche à **prédire**.
- Les **variables prédictrices** (ou indépendantes) : ce sont les variables qui vont permettre de faire cette prédiction.

Exemple : Si nous avons un jeu de données avec les informations sur l'âge, la taille et le poids de collégiens, et que nous souhaitons **prédire le poids** en fonction de la taille et de l'âge, alors :

- **Variable cible** : Le poids.
- **Variables prédictrices** : L'âge et la taille.

Dataset Exploitable

Pour que le Machine Learning soit pertinent, le jeu de données doit être **cohérent** et **exploitable**. Cela signifie que les variables doivent avoir une certaine **relation entre elles**. Le but n'est pas d'analyser des données totalement aléatoires, mais de trouver des **modèles** et des **corrélations** qui permettront d'entraîner efficacement le modèle prédictif. Par exemple, dans le cas précédent, il est raisonnable de penser que l'âge et la taille ont une influence sur le poids d'un enfant.

Résumé

Le Machine Learning classique s'appuie sur des **données structurées** et **tabulaires**, avec des jeux de données de taille raisonnable. L'objectif principal est d'entraîner un modèle capable de **prédire** une **variable cible** à partir de **variables prédictrices**, en se basant sur des données cohérentes.

Étapes pour Développer un Modèle Prédictif

Lors de la création d'un modèle prédictif en Machine Learning, il est essentiel de suivre des étapes précises pour maximiser la qualité et la robustesse du modèle. Voici les principales étapes à suivre.

1. Travail sur la Donnée (Data Cleaning)

La première étape consiste à **préparer** les données brutes pour qu'elles soient compatibles avec le modèle de Machine Learning. Ce processus est souvent appelé **data cleaning** et inclut plusieurs sous-tâches :

- **Nettoyage** : Gérer les **données manquantes** (par exemple, combler les lacunes ou supprimer les entrées incomplètes) et corriger les **données erronées** (exemple : une personne âgée de 200 ans ou un poids négatif).
- **Normalisation** : Aligner les **échelles des variables** pour éviter les déséquilibres (par exemple, normaliser des prix de maisons exprimés en centaines de milliers avec des mètres carrés en centaines).
- **Numérisation** : Convertir les données non numériques (catégorielles) en valeurs numériques que le modèle peut comprendre.
- **Fusion des données** : Parfois, les données sont dispersées sur plusieurs fichiers (exemple : 200 fichiers Excel) ou dans des formats incompatibles qu'il faut combiner en un seul ensemble cohérent.

Exemple : Si la moitié des enregistrements pour la taille d'une personne est manquante, il faudra peut-être remplir les valeurs manquantes avec une moyenne ou une médiane.

2. Validation Croisée

Pour s'assurer que le modèle généralisera bien sur des **données inconnues**, il est crucial de valider ses performances sur des données qu'il n'a jamais vues pendant l'entraînement. On divise donc le jeu de données en plusieurs parties.

- **Train** (80 % des données) : Utilisé pour entraîner le modèle.
- **Test** (20 % des données) : Utilisé pour évaluer la performance du modèle sur des données qu'il n'a pas vues.

Ensuite, on utilise une méthode de **validation croisée**, qui consiste à découper plusieurs fois et de façon aléatoire les données entre les ensembles train et test. Cela permet d'éviter que la performance du modèle ne soit biaisée par une seule répartition des données.

Séparation Avancée en Trois Parties

Pour éviter d'optimiser trop le modèle en fonction des données test, on scinde souvent le jeu de données en trois parties :

- **Train** : Pour l'entraînement.
- **Validation** : Permet d'ajuster le modèle après l'entraînement.
- **Test final** : Pour évaluer définitivement la performance du modèle une fois qu'il est optimisé.

Dans certains domaines (ex : prédiction de stocks), cette séparation stricte est essentielle, car une fois que la partie validation est utilisée, le modèle ne doit plus être modifié pour éviter toute corruption.

3. Optimisation du Modèle

Après avoir séparé et nettoyé les données, la dernière étape consiste à **optimiser les performances du modèle**. Cela implique :

- **Ajustement des paramètres** : Chaque modèle possède ses propres paramètres qu'il est possible de **tuner** (ajuster). Ces paramètres varient selon la bibliothèque et le type de modèle utilisé.
- **Évaluation** : Après chaque ajustement de paramètres, on évalue le modèle sur les différentes partitions (train/test/validation).

L'optimisation peut être répétée plusieurs fois pour améliorer les prédictions et rendre le modèle plus robuste. Au fur et à mesure, on développe un **savoir-faire** qui permet de mieux comprendre comment ajuster les modèles et quels paramètres ont le plus d'impact.

Résumé

Le développement d'un modèle prédictif en Machine Learning passe par plusieurs étapes essentielles :

- Le **nettoyage et la préparation des données** pour les rendre exploitables.
- La **validation croisée** pour garantir la robustesse du modèle sur des données non vues.
- L'**optimisation des paramètres** pour améliorer la performance du modèle.

Ces étapes forment le cœur du travail de **data scientist** et permettent d'obtenir des modèles prédictifs précis et fiables.

Différence entre Approche Supervisée et Non Supervisée en Machine Learning

Le Machine Learning propose deux grandes approches pour résoudre des problèmes à partir de données : **l'approche supervisée** et **l'approche non supervisée**. Elles diffèrent par la façon dont elles utilisent les données pour entraîner un modèle.

1. L'Approche Non Supervisée

Dans une **approche non supervisée**, les données ne sont **pas étiquetées**. Le modèle n'a pas de connaissance préalable sur les catégories ou classes des données qu'il traite. Il va donc tenter de découvrir des **structures cachées** ou des **regroupements naturels** (clusters) en fonction des similitudes entre les données.

Exemple : Clustering

Imaginons que vous ayez un grand nombre de photos de chats et de chiens, mais sans étiquette les différenciant. Un modèle non supervisé va regrouper les photos en fonction de caractéristiques partagées (formes, couleurs, textures) sans savoir qu'il s'agit de chats ou de chiens. Ce processus est appelé **clustering**.

- **Objectif** : Identifier des **groupes** ou **clusters** dans les données.
- **Utilisation** : Segmenter des clients selon leur comportement d'achat, regrouper des images ou des documents similaires.

Exemple d'algorithmes : K-means, DBSCAN, PCA (réduction de dimensionnalité).

2. L'Approche Supervisée

Dans une **approche supervisée**, les données sont **étiquetées** : chaque échantillon de données est associé à une **variable cible** ou **étiquette** que le modèle doit apprendre à prédire. Cette étiquette est généralement attribuée par un humain, ce qui peut être coûteux et long lorsque l'on travaille avec de grandes quantités de données.

Exemple : Classification et Régression

Si chaque photo de chat et de chien est étiquetée comme étant "chat" ou "chien", le modèle supervisé va apprendre à associer ces étiquettes à des caractéristiques de l'image (taille, forme, couleurs). Le modèle pourra ensuite **prédire** si une nouvelle image correspond à un chat ou à un chien.

En **supervisé**, il existe deux sous-catégories selon le type de variable cible :

- **Classification** : Lorsque la variable cible est une **catégorie** (exemple : chat ou chien, oui ou non). On peut avoir des classifications **binaires** (ex : oui/non) ou **multiclasses** (ex : bleu, vert, rouge).
- **Régression** : Lorsque la variable cible est une **variable continue** (exemple : prédire un prix, un âge, une température).
- **Objectif** : Apprendre à prédire la **valeur** ou **catégorie** de la variable cible en fonction des autres variables du jeu de données.
- **Utilisation** : Prédiction de ventes, reconnaissance d'images, prédiction de la météo.

Exemple d'algorithmes : Régression logistique, Arbres de décision, Réseaux de neurones, SVM.

Résumé des Différences

Approche	Supervisée	Non Supervisée
Données étiquetées	Oui (variable cible définie)	Non (pas de variable cible)
Objectif	Prédire une étiquette ou une valeur	Regrouper les données, découvrir des structures
Types d'algorithmes	Classification, Régression	Clustering, Réduction de dimensionnalité
Exemple d'application	Prédire si une image est un chat ou un chien	Regrouper les photos sans étiquettes par similitudes
Exemple d'algorithmes	Arbre de décision, SVM, Régression linéaire	K-means, PCA

Résumé

- L'**approche supervisée** nécessite des **données étiquetées** et vise à **prédire une valeur** ou une **catégorie** (classification ou régression).
- L'**approche non supervisée** fonctionne sans étiquettes et cherche à **découvrir des structures** ou des **groupes** dans les données (clustering).

Ces deux approches sont complémentaires selon le type de problème que l'on cherche à résoudre.

Les modèles prédictifs

Les modèles prédictifs en Machine Learning sont des outils utilisés pour réaliser des prédictions à partir de données. Dans la bibliothèque Python **scikit-learn**, on trouve une variété de modèles supervisés, adaptés à différents types de tâches (régression, classification, clustering), de nature de données (chiffres, texte, images, etc.), et à divers volumes et caractéristiques statistiques des données.

Choisir un modèle

Le choix d'un modèle dépend du **contexte du projet** :

- **Tâche à accomplir** : Régression (prédiction de variables continues), Classification (prédiction de catégories) ou Clustering (groupement non supervisé).
- **Nature des données** : Données numériques, catégories, texte, images, etc.
- **Caractéristiques des données** : Relations linéaires, complétude, cohérence.

Types principaux de modèles

1. **Modèles de régression**
   Ils sont adaptés à des jeux de données de petite taille avec des relations linéaires entre les variables. Les modèles de régression incluent la régression linéaire, mais ils peuvent également être utilisés pour la classification. Ce sont des modèles dits **GLM (Generalized Linear Models)**. Bien que leur nom suggère qu'ils soient réservés à la régression, ils peuvent être utilisés pour la **classification**.

2. **Modèles à base d'arbres**
   Les modèles comme les **arbres de décision**, les **forêts aléatoires** et **XGBoost** sont plus robustes face aux données manquantes et aux valeurs aberrantes (**outliers**). Ces modèles sont très efficaces pour des jeux de données plus volumineux et sont souvent les favoris dans les compétitions de Machine Learning comme **Kaggle**.

3. **Réseaux de neurones et Deep Learning**
   Ces modèles sont particulièrement adaptés à des données complexes et de grand volume, comme les images ou les vidéos. Le **Deep Learning** excelle dans le traitement des données massives et complexes grâce à sa capacité à modéliser des relations non linéaires à grande échelle.

Résumé

Bien que de nombreux autres types de modèles existent, les modèles de régression, à base d'arbres, et les réseaux de neurones sont aujourd'hui les plus populaires en raison de leur polyvalence et de leur robustesse dans la majorité des cas d'usage.

Définir un Modèle et un Algorithme en Machine Learning

Définir un modèle

En Machine Learning, un **modèle** représente une méthode mathématique traduite en code pour effectuer des prédictions ou des classifications. Il y a deux aspects à considérer :

1. **Type de Modèle**
   - **Exemple en Python** : Importation d'un modèle de régression linéaire depuis scikit-learn.

     ```
     from sklearn.linear_model import LinearRegression
     ```

   - Cela définit la méthode (ou le modèle) à utiliser pour entraîner et prédire les données.
2. **Instanciation du Modèle**
   - Une fois le modèle importé, il faut l'**instancier** et l'**entraîner** sur les données.
   - **Code d'exemple** :

```
from sklearn.linear_model import LinearRegression

# Données d'entraînement
X = [[0, 0], [1, 1]]  # Les données d'entrée (features)
y = [0, 1]            # Les valeurs cibles (target)

# Instanciation du modèle
mdl = LinearRegression()

# Entraînement du modèle avec les données
mdl.fit(X, y)

# Affichage des coefficients du modèle
print("Coefficients du modèle :", mdl.coef_)
```

- Ici, `mdl` est l'instance du modèle qui a été entraînée avec les données `X` et `y`. En pratique, "modèle" peut désigner à la fois la classe du modèle et son instance après entraînement.

Définir un Algorithme

Un **algorithme** est une série d'instructions ordonnées permettant de réaliser une tâche spécifique. En Machine Learning, il s'agit souvent de l'algorithme utilisé pour **entraîner** un modèle.

Exemple avec l'algorithme de Héron pour calculer la racine carrée :

1. **Initialisation**
   - Commencez avec une valeur initiale `x = 1`.
2. **Critère de Précision**
   - Définissez une précision désirée, par exemple `0,001`.
3. **Itérations**
   - Répétez jusqu'à ce que l'erreur d'estimation soit inférieure à la précision.
     - Formule :

       ```
       x = (x + 2 / x) / 2
       ```

4. **Code Python** :

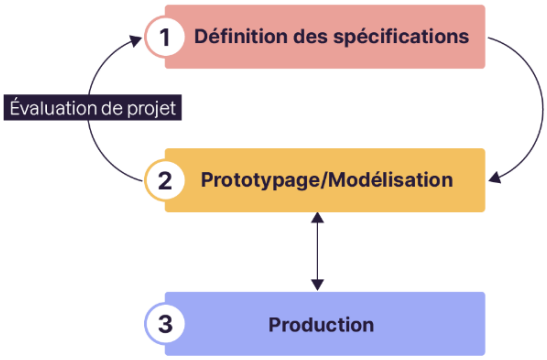
   ```python
   # Initialisation
   x = 1
   precision = 0.001

   # Boucle itérative
   while abs(x**2 - 2) > precision:
       x = (x + 2 / x) / 2

   print("Approximation de la racine carrée de 2 :", x)
   ```

   - **Erreur d'Estimation** : La différence entre la valeur estimée `x^2` et la valeur réelle `2`. Cette erreur est mise à jour à chaque itération.

Code disponible dans /Notebook/sklearn_sandbox.py

Phases d'un Projet de Data Science

1. **Définition**
2. **Prototypage**
3. **Production**



Cycle itératif d'un projet de Data Science

Phase 1 – Définir les Spécifications

Objectif : Traduire un problème business en un projet de Machine Learning.

Étapes Clés :

1. **Pertinence des Données** : Assurer que les données disponibles sont adéquates et pertinentes pour le problème à résoudre.
2. **Définition du Sujet ou Produit** : Clarifier précisément le sujet du projet ou le produit à développer.
3. **Évaluation de la Valeur Ajoutée** :
   - Déterminer si l'utilisation du Machine Learning est justifiée par rapport à des solutions plus simples basées sur des règles (rule-based solutions).
   - Réaliser une étude de benchmark pour évaluer les avantages réels du Machine Learning.
4. **Critères de Réussite** :
   - Définir les critères de succès du projet.
   - Déterminer comment mesurer la performance du modèle.
   - Choisir la métrique appropriée pour évaluer le modèle et fixer les objectifs de score nécessaires.

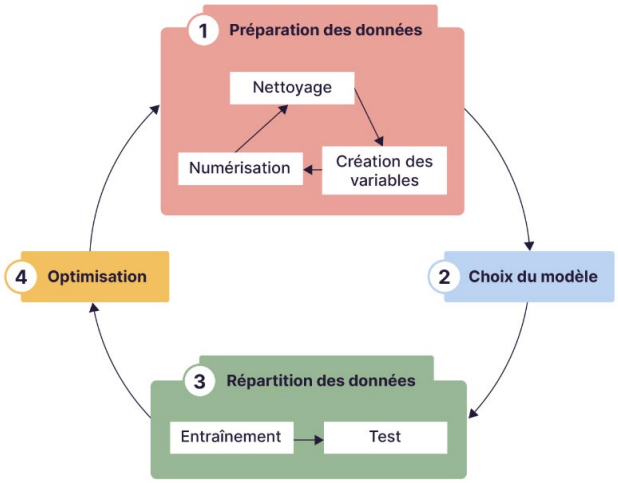
Ce processus permet de s'assurer que le projet de Machine Learning est bien cadré avant de passer aux étapes de prototypage et de production, en garantissant qu'il apportera une réelle valeur ajoutée par rapport à des méthodes plus simples.

Phase 2 – Concevoir le Prototype et Valider la Faisabilité du Projet

Objectif : Développer un modèle efficace et robuste pour résoudre la problématique business définie.

Étapes Clés :

1. **Préparation des Données** :
   - **Nettoyage** : Résoudre les problèmes de données aberrantes et manquantes.
   - **Feature Engineering** : Créer de nouvelles variables utiles à partir des données existantes.
   - **Numérisation** : Convertir les données catégoriques, textuelles ou visuelles en un format que le modèle peut traiter.
2. **Choix du Modèle** :
   - Sélectionner le type de modèle approprié (par exemple, modèles linéaires, arbres de décision, réseaux de neurones, etc.).
3. **Répartition des Données** :
   - Diviser les données en ensembles d'entraînement et de validation pour évaluer la performance du modèle.
4. **Optimisation des Paramètres** :
   - Ajuster les paramètres du modèle pour améliorer sa performance et sa robustesse



Processus Itératif : Cette phase implique des itérations fréquentes entre la collecte et la préparation des données, le choix du modèle, et son optimisation pour affiner le prototype.

Phase 3 – Mettre en Production le Projet

Objectif : Intégrer le modèle optimisé dans un environnement de production pour une utilisation réelle.

Étapes Clés :

1. **Mise en Production** :
   - Intégrer le modèle dans le produit final et déployer dans l'environnement de production.
2. **Rôle des MLOps** :
   - Les MLOps combinent les pratiques de DevOps avec le Machine Learning pour gérer le déploiement, la mise à jour, et la surveillance des modèles en production.
3. **Surveillance et Maintenance** :
   - Les modèles doivent être surveillés régulièrement pour s'assurer qu'ils restent efficaces malgré les changements dans les données ou les conditions environnementales.
   - **Drift du Modèle** : Réajuster les modèles lorsque les conditions évoluent, comme les crises économiques ou les changements climatiques.
4. **Outils de Gestion** :
   - Utilisation de plateformes de gestion à grande échelle comme Kubernetes et d'outils de surveillance comme Seldon ou MLflow pour gérer le cycle de vie des modèles en production.

Ce processus assure que les modèles non seulement répondent aux besoins définis initialement mais restent également efficaces et pertinents au fil du temps.

Évaluation de la Performance d'un Modèle Prédictif

Étapes pour Évaluer un Modèle Prédictif

1. **Charger les Données** :
   - Utilisez `pandas` pour charger les données depuis un fichier CSV :

     ```
     import pandas as pd
     df = pd.read_csv('<fichier csv des données>')
     ```

2. **Préparer les Variables** :
   - **Variables Prédictives (X)** : Sélectionnez les colonnes qui serviront de prédicteurs.

- **Variable Cible (y)** : Sélectionnez la colonne que vous souhaitez prédire.

```
X = df[['age', 'taille']]
y = df.poids
```

3. **Entraîner le Modèle** :
   - Importez et instanciez le modèle de régression linéaire :

```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
```

   - Entraînez le modèle avec la méthode `fit()` :

```
reg.fit(X, y)
```

4. **Évaluer le Modèle** :
   - **Score du Modèle** : Le score R^2 mesure la proportion de variance expliquée par le modèle. Plus il est proche de 1, mieux le modèle est performant :

```
print(reg.score(X, y))
```

   - **Coefficients du Modèle** : Les coefficients (a, b, c) indiquent l'impact de chaque prédicteur sur la variable cible :

```
print(reg.coef_)
```

5. **Faire des Prédictions** :
   - Utilisez le modèle pour prédire de nouvelles valeurs :

```
import numpy as np
poids_prédit = reg.predict(np.array([[0, 150, 153]]))
print(poids_prédit)
```

Minimisation d'une Fonction de Coût

Objectif : Trouver les meilleurs paramètres pour minimiser l'erreur d'estimation.

1. **Erreur d'Estimation** :
   - L'erreur d'estimation quantifie à quel point les prédictions du modèle diffèrent des valeurs réelles. Par exemple, pour la racine carrée de 2, l'erreur est la différence entre la valeur prédite et 2.
2. **Fonctions de Coût** :
   - **Définition** : Une fonction de coût mesure l'écart entre les valeurs prédites et les valeurs réelles. Minimiser cette fonction permet d'améliorer le modèle.
   - **Exemples de Fonctions de Coût** :

- **Erreur Quadratique Moyenne (MSE)** : La moyenne des carrés des erreurs.
- **Erreur Absolue Moyenne (MAE)** : La moyenne des valeurs absolues des erreurs.

3. **Optimisation** :
   - Les algorithmes d'optimisation ajustent les paramètres du modèle pour minimiser la fonction de coût. Par exemple, dans l'algorithme de gradient stochastique, les paramètres sont mis à jour en fonction de l'erreur d'estimation.

La **fonction de coût** est un outil clé en Machine Learning pour mesurer l'écart entre les prédictions d'un modèle et les valeurs réelles. Elle guide l'algorithme d'entraînement pour ajuster les paramètres du modèle afin de minimiser cet écart.

1. **Définition** :
   - Pour la **régression linéaire**, la fonction de coût couramment utilisée est l'**Erreur Quadratique Moyenne (MSE)** :
     MSE=1N∑Nk=1(yk−y^k)2
     - (N) est le nombre d'échantillons,
     - (y_k) sont les valeurs réelles,
     - (\hat{y}_k) sont les valeurs prédites.
2. **Interprétation Géométrique** :
   - Minimiser une fonction de coût implique d'ajuster les paramètres du modèle pour réduire l'erreur. La dérivée partielle de la fonction de coût par rapport aux paramètres aide à déterminer la direction dans laquelle l'erreur diminue le plus rapidement. Les algorithmes comme la **descente de gradient** utilisent cette information pour trouver les paramètres optimaux.

Évaluation de la Performance d'un Modèle

Après l'entraînement, il est crucial de **quantifier la performance** du modèle pour évaluer la qualité des prédictions.

1. **Scores de Performance pour la Régression** :
   - **Root Mean Square Error (RMSE)** :

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(y_k - \hat{y}_k)^2}$$

- **Mean Absolute Error (MAE)** :

$$\mathrm{MAE} = \frac{1}{N}\sum_{k=1}^{N} |y_k - \hat{y}_k|$$

- **Mean Absolute Percentage Error (MAPE)** :

$$\mathrm{MAPE} = \frac{1}{N}\sum_{k=1}^{N} \frac{|y_k - \hat{y}_k|}{|y_k|}$$

  - **RMSE** et **MAE** mesurent l'erreur en unités de la variable cible. Un score plus bas indique un meilleur modèle.
  - **MAPE** exprime l'erreur en pourcentage des valeurs réelles, facilitant la comparaison entre modèles et ensembles de données de différentes échelles.

2. **Calcul des Scores avec scikit-learn** :

   - Après avoir obtenu les prédictions du modèle :

     ```
     y_pred = reg.predict(X)
     ```

   - Calculez les scores :

     ```
     from sklearn.metrics import mean_squared_error, mean_absolute_err
     or, mean_absolute_percentage_error
     mse = mean_squared_error(y, y_pred)
     mae = mean_absolute_error(y, y_pred)
     mape = mean_absolute_percentage_error(y, y_pred)

     print(f"MSE: {mse}")
     print(f"MAE: {mae}")
     print(f"MAPE: {mape}")
     ```

Méthodologie du Processus d'Entraînement

Le processus d'entraînement d'un modèle, quel que soit le type de tâche (régression, classification, clustering), suit généralement ces étapes :

1. **Charger les Données** :

   - Importez les données nécessaires pour l'entraînement et l'évaluation.

2. **Transformer/Améliorer les Données** :

   - Nettoyez les données, gérez les valeurs manquantes, et effectuez des transformations si nécessaire.

3. **Scinder les Données** :

- Divisez les données en ensembles d'entraînement et de test (et parfois validation) pour évaluer la performance du modèle sur des données non vues.

4. **Entraîner le Modèle** :
   - Ajustez les paramètres du modèle en utilisant l'ensemble d'entraînement.
5. **Calculer les Scores** :
   - Évaluez la performance du modèle sur l'ensemble de test en utilisant les métriques appropriées.

Principe de la Régression Linéaire

Exercice : CF : Jupiter Notebook partie : Régression Linéaire: Notebook/P2C2_principe_regression_lineaire.ipynb

Introduction à la régression linéaire

La régression linéaire est une méthode statistique utilisée pour modéliser et prédire une variable continue en fonction d'une ou plusieurs variables prédictives. On peut la voir comme une ligne droite qui capture la tendance générale des données. Elle est représentée par une équation de la forme :

```
y = a * x + b
```

- **(y)** : la variable cible (la valeur que l'on veut prédire)
- **(x)** : la variable de prédiction
- **(a)** : le coefficient qui décrit l'impact de la variable (x) sur (y)
- **(b)** : l'interception (valeur de (y) lorsque (x) est nul)

Lorsqu'il y a plusieurs variables de prédiction, l'équation devient :

```
y = a_1 * x_1 + a_2 * x_2 + b
```

```
y = a_1 * x_1 + a_2 * x_2 + ... + a_N * x_N + b
```

Ici, chaque coefficient (a_i) reflète l'importance respective des prédicteurs (x_i) dans la prédiction de la variable cible (y).

Applications et avantages

La régression linéaire est largement utilisée dans divers domaines comme :

- La médecine,
- L'économie,
- Le marketing.

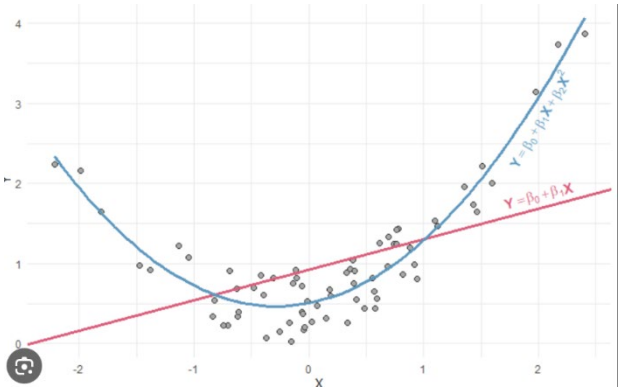
Les avantages de la régression linéaire incluent :

1. **Interprétabilité** : Les coefficients révèlent l'impact direct des prédicteurs sur la variable cible.
2. **Facilité d'implémentation** : De nombreux langages et bibliothèques (comme Python et scikit-learn) facilitent son utilisation.
3. **Rapidité de calcul** : Les algorithmes de régression linéaire sont très efficaces.
4. **Faible consommation de mémoire** : Elle convient aux applications comme l'Internet des objets (IoT).

Limites de la régression linéaire

Malgré ses avantages, la régression linéaire présente certaines limites :

1. **Relation non linéaire** :
   Si la relation entre la variable cible et les prédicteurs est non linéaire (par exemple, une courbe), une régression linéaire ne sera pas adaptée et ne pourra pas bien modéliser la tendance des données.



2. **Problème d'amplitude** :
   Les variables avec des amplitudes très différentes peuvent fausser les résultats. Par exemple, si une variable est beaucoup plus grande qu'une autre, son coefficient sera proportionnellement plus petit, même si elles ont le même pouvoir prédictif. La normalisation des données (mettre les variables sur une échelle comparable) est donc essentielle pour garantir une interprétation correcte des coefficients.

Résumé

La régression linéaire est un outil puissant pour modéliser des relations linéaires entre variables. Cependant, il faut s'assurer que les données respectent certaines conditions, comme la linéarité de la relation cible-prédicteur et une amplitude comparable des variables, pour obtenir des résultats fiables.

Classification des données avec la régression logistique

Introduction à la régression logistique

Dans un problème de classification, la variable cible représente une **catégorie** au lieu d'une **valeur continue** comme dans le cas de la régression.

Il existe plusieurs types de classifications en fonction des valeurs que peut prendre la variable cible :

1. **Classification binaire** : la variable cible n'a que deux valeurs possibles (ex. : oui/non, 0/1, etc.).
2. **Classification multiclasse** : la variable cible peut avoir plus de deux valeurs.

   Dans ce type de classification, on distingue :

   - **Classification ordinale** : les catégories ont un ordre (ex. : mauvais, moyen, bon).
   - **Classification nominale** : les catégories n'ont pas d'ordre spécifique (ex. : couleurs, animaux).
3. **Classification multilabel** : un échantillon peut appartenir à plusieurs catégories en même temps (ex. : genres d'un film).

Nous allons nous concentrer principalement sur le cas le plus simple, la classification binaire.

Liens entre régression et classification

Le principe fondamental est similaire à celui de la régression, mais avec des différences dans la manière de calculer les scores et d'interpréter les résultats.

Le terme "régression logistique" s'explique par le fait qu'on adapte la régression linéaire à un problème de classification en interprétant la prédiction comme une catégorie de la variable cible.

Prenons un exemple simple de régression linéaire avec un seul prédicteur :

```
y = a * x + b
```

Dans ce cas, y est une variable continue qui peut prendre n'importe quelle valeur réelle.

Pour adapter cette régression à la classification, on utilise une fonction qui va transformer cette prédiction y en une valeur comprise entre 0 et 1, qui peut être interprétée comme une **probabilité**. Cette fonction est la **fonction logistique** :

```
z = f(y) = 1 / (1 + e^(-y))
```

La valeur z peut être interprétée comme la probabilité que l'échantillon appartienne à une des deux catégories. Par exemple, en prenant un seuil de classification de 0.5 :

```
si f(y) < 0.5 => catégorie 0
si f(y) >= 0.5 => catégorie 1
```

Interprétation des probabilités

Une valeur entre 0 et 1 peut être interprétée comme une probabilité, où 0 signifie que l'événement est impossible et 1 qu'il est certain. Les valeurs entre les deux correspondent aux différentes chances de voir cet événement se produire.

En utilisant la régression linéaire avec la fonction logistique, nous projetons toute prédiction dans l'intervalle `[0, 1]`, puis interprétons ce résultat comme une probabilité.

La fonction logistique, qui réalise cette projection, est définie par :

```
f(y) = 1 / (1 + e^(-y))
```

La courbe de cette fonction ressemble à un "S", ce qui justifie le nom de **régression logistique**.

Étapes de la régression logistique

Voici comment fonctionne la régression logistique dans le cas d'une classification binaire :

1. **Modéliser une régression linéaire** :

   ```
   y = a * x + b
   ```

2. **Projeter la valeur y dans l'intervalle `[0, 1]`** à l'aide de la fonction logistique :

   ```
   z = f(y) = 1 / (1 + e^(-y))
   ```

1. **Interpréter `f(y)` comme une probabilité** :
   - Si `f(y) < 0.5`, alors l'échantillon est classé dans la **catégorie 0**.
   - Si `f(y) >= 0.5`, alors l'échantillon est classé dans la **catégorie 1**.

Partitionnement des Données avec K-means

Introduction au Clustering

Le **clustering** (ou partitionnement) est une technique d'apprentissage non supervisé qui regroupe des échantillons similaires en clusters sans étiquettes prédéfinies.

Objectifs du clustering :

1. **Minimiser la variance intra-groupe :** Les points au sein d'un même groupe doivent être proches les uns des autres.
2. **Maximiser la variance inter-groupes :** Les groupes doivent être bien séparés les uns des autres.

Questions Clés du Clustering

1. **Comment définir la similarité ?** La similarité est souvent mesurée par la distance euclidienne entre les points :

   ```
   distance = np.sqrt(np.sum((x1 - x2)**2))
   ```

2. **Comment choisir le nombre optimal de clusters ?** Utilisez des méthodes comme la **méthode du coude** ou la **méthode de la silhouette** pour déterminer le nombre optimal de clusters.

3. **Quels algorithmes utiliser pour le clustering ?** Nous utiliserons **k-means** pour sa simplicité.

Principe de l'Algorithme K-means

L'algorithme k-means fonctionne en plusieurs étapes :

1. **Initialisation :**
   - Choisissez le nombre de clusters (K).
   - Sélectionnez aléatoirement (K) points comme centres initiaux (centroïdes).
2. **Attribution des données :** Assignez chaque point au cluster dont le centroïde est le plus proche.

   ```
   label = np.argmin([np.linalg.norm(x - centroid) for centroid in centroi
   ds])
   ```

3. **Mise à jour des centroïdes :** Recalculez les coordonnées des centroïdes en fonction des points assignés à chaque cluster.

   ```
   new_centroid = np.mean(points_in_cluster, axis=0)
   ```

4. **Itération :** Répétez les étapes d'attribution et de mise à jour jusqu'à ce que les centroïdes ne changent plus significativement.

Application de K-means (CF : Notebook/ P2C4_partitionnez_kmean.ipynb)

Création et Clustering d'un Dataset Artificiel

Utilisez `make_blobs` pour créer un dataset artificiel :

```
from sklearn.datasets import make_blobs
centers = [[2, 2], [-2, -2], [2, -2]]
X, labels_true = make_blobs(n_samples=3000, centers=centers, cluster_std=0.7)
```

Code pour appliquer k-means :

```
from sklearn.cluster import KMeans
k_means = KMeans(n_clusters=3, random_state=808)
k_means.fit(X)
print(k_means.cluster_centers_)
```

Évaluation de la performance :

1. **Distance des points aux centroïdes :** Utilisez la méthode `score` pour évaluer la qualité du clustering.

   ```
   score = k_means.score(X)
   ```

2. **Visualisation :** Comparez graphiquement les clusters prévus avec les clusters réels.

3. **Coefficient de Silhouette :** Calculez le coefficient de silhouette pour évaluer la qualité du clustering :

   ```
   from sklearn.metrics import silhouette_score
   silhouette = silhouette_score(X, k_means.labels_)
   ```

   Le coefficient de silhouette est calculé comme suit :

   ```
   silhouette = (b - a) / max(a, b)
   ```

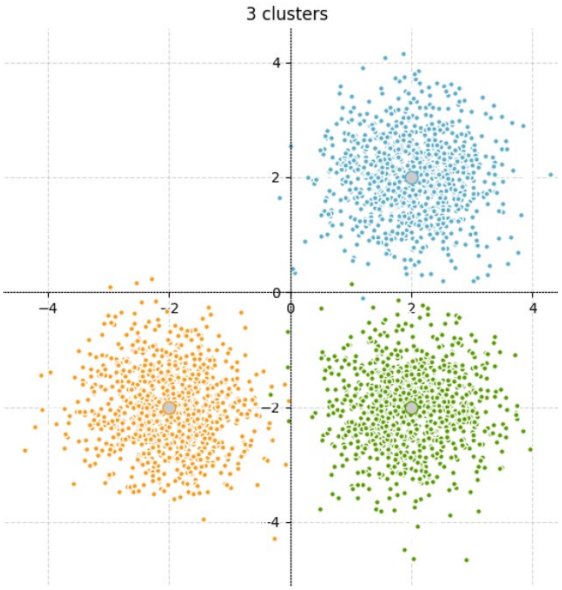
   Où :

   - `a` est la distance moyenne intra-groupe.
   - `b` est la distance moyenne au groupe le plus proche.

Application de K-means sur le Dataset Iris

Le dataset Iris contient 150 échantillons répartis en 3 espèces de fleurs. Appliquez k-means sur ce dataset :

```
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, confusion_matrix, silhouette_scor
e

iris = datasets.load_iris()
X = iris.data
y = iris.target

model = KMeans(n_clusters=3, n_init="auto")
model.fit(X)
print("score", model.score(X))
print("silhouette_score: ", silhouette_score(X, model.labels_))
```



3 nuages de points homogènes et distincts

Évaluation des résultats :

1. **Précision du modèle :** Comparez les résultats du clustering avec les étiquettes réelles.

2. **Matrice de confusion :** Calculez la matrice de confusion pour comprendre les erreurs de classification :

```
labels = model.labels_.copy()
labels[labels == 0] = 5
labels[labels == 1] = 10
labels[labels == 2] = 15
labels[labels == 5] = 0
labels[labels == 10] = 2
labels[labels == 15] = 1

from sklearn.metrics import accuracy_score, confusion_matrix
accuracy = accuracy_score(y, labels)
conf_matrix = confusion_matrix(y, labels)
print("accuracy_score", accuracy)
print("confusion_matrix\n", conf_matrix)
```

Résumé

- **k-means** est un algorithme de clustering basé sur l'itération entre attribution des points et mise à jour des centroïdes.
- **Évaluation** : Utilisez des métriques comme le coefficient de silhouette et visualisez les clusters pour évaluer la performance.

- **Application** : k-means peut également être utilisé pour la classification, mais il est principalement destiné au regroupement des données.

Comprenez le rôle central du jeu de données

Approche Data-Centric vs. Model-Centric

Approche Model-Centric :

- Concentre sur la création et l'optimisation du modèle.
- Les modèles efficaces sont souvent déjà disponibles via des librairies open source ou des services cloud.

Approche Data-Centric :

- Concentre sur l'amélioration des données d'entraînement pour booster les performances du modèle.
- Se focalise sur :
  - La qualité des données : valeurs manquantes, outliers, biais, etc.
  - La transformation des caractéristiques pour améliorer la capacité d'apprentissage du modèle.

Trouver des Jeux de Données

Sources recommandées :

- **Dataset Search** par Google.
- **Kaggle** pour des compétitions et datasets variés.
- **UCI Machine Learning Repository** pour une large gamme de datasets.
- **Portails institutionnels** de villes et institutions européennes.
- **Agences scientifiques et ONG** comme ADEME, EdF, WWF.
- **BigQuery** pour des datasets volumineux.
- **Librairies Python** : scikit-learn, statsmodels, et R Datasets.

Créez Votre Propre Dataset : Exercice complet : Notebook/P3C1_creer_dataset.ipynb

Utilisez scikit-learn pour créer des datasets adaptés :

- **`make_blobs()`** pour clustering.
- **`make_classification()`** pour classification.
- **`make_regression()`** pour régression.

Exemples de génération de datasets :

- **Circles** et **Moons** :

```
from sklearn.datasets import make_circles, make_moons

data_circles = make_circles(n_samples=1000, factor=0.5, noise=0.05)
data_moons = make_moons(n_samples=1000, noise=0.05)
```

Réduisez l'Influence des Préjugés

Problèmes de biais :

- **Échantillonnage biaisé** : Non-représentation de certaines catégories dans les données.
- **Biais dans les modèles LLM** : Reproduction de biais historiques, culturels, ou stéréotypés.

Solutions :

- Assurer la représentativité des données.
- Mettre en place des stratégies pour atténuer les biais.

En Résumé

- La performance des modèles peut être significativement améliorée en travaillant sur les données d'entraînement.
- Scikit-learn et autres outils offrent des fonctionnalités pour créer des jeux de données adaptés à divers types de tâches.
- Il est crucial de détecter et de corriger les biais dans les modèles pour éviter des prédictions injustes ou incorrectes.

Amélioration d'un Jeu de Données - Exercice complet : "Notebook/P3C2_ameliorer_dataset.ipynb"

Dans ce chapitre, nous explorons comment améliorer un jeu de données pour le rendre plus utile pour l'apprentissage automatique. Nous utilisons un jeu de données réel sur les arbres de Paris pour illustrer les étapes de nettoyage et de préparation des données.

1. Repérer les Données Manquantes et les Outliers

Les jeux de données réels contiennent souvent des valeurs manquantes et des outliers, ce qui peut fausser les résultats des modèles d'apprentissage automatique.

- **Données Manquantes** : Elles sont relativement faciles à identifier et nécessitent des méthodes de traitement comme l'imputation ou la suppression.
- **Outliers (Données Aberrantes)** : Plus difficiles à identifier, les outliers peuvent être :
  - **Vraiment absurdes** : Par exemple, un âge de 200 ans pour un arbre.

- **Extrêmes mais potentiellement valables** : Par exemple, un arbre de 80 ans dans un dataset de coureurs de marathon. Les outliers influencent les statistiques de distribution (comme la moyenne) et peuvent biaiser les modèles.

Identification des Outliers :

- **Visualisation** : Utilisez des histogrammes ou des boxplots pour détecter des valeurs aberrantes.

•

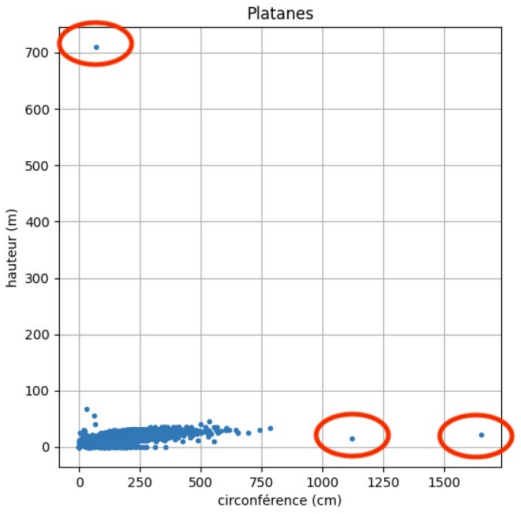
–

- **Méthode du Z-Score** : Mesure l'écart par rapport à la moyenne. Un z-score supérieur à 2 ou 3 indique généralement un outlier.

```
  ```
    ```
      ```
        ```
          ```
            ```python
            from scipy import stats
            df['z_circonference'] = stats.zscore(df.circonf
erence_cm)
            df['z_hauteur'] = stats.zscore(df.hauteur_m)
            ```
          ```

          - **Méthode de l'IQR (Interquartile Range)** : Id
entifie les valeurs au-dessus de ( Q3 + 1.5 \\times \\text{
IQR} ) ou en-dessous de ( Q1 - 1.5 \\times \\text{IQR} ).

              ```python
              import numpy as np
              iqr = np.quantile(df.hauteur_m,q=[0.25,0.75])
              limite_basse = iqr[0] - 1.5 * (iqr[1] iqr[0])
              limite_haute = iqr[1] + 1.5 * (iqr[1] iqr[0])
              ```
        ```
      ```
    ```
  ```
```



Nuage de points de la hauteur par rapport à la circonférence des arbres

2. Traitement des Outliers

- **Suppression** : Éliminez les valeurs aberrantes si elles sont clairement erronées.
- **Transformation** : Utilisez des transformations log ou des méthodes de normalisation pour réduire l'influence des outliers.
  - **Logarithme** : Appliquez le logarithme pour réduire l'amplitude des valeurs.

    ```
    df['circonference_log'] = np.log(df.circonference_cm + 1)
    df['hauteur_log'] = np.log(df.hauteur_m + 1)
    ```

  - **Discrétisation** : Utilisez `qcut` et `cut` pour diviser les données en intervalles.

    ```
    import pandas as pd
    pd.qcut(df.hauteur_m, 3, labels=["petit", "moyen", "grand"]).valu
    e_counts()
    ```

3. Normalisation des Valeurs Numériques

Certaines méthodes de Machine Learning sont sensibles à l'échelle des variables. La normalisation permet d'égaliser les amplitudes des valeurs pour améliorer les performances du modèle.

- **Min-Max Scaling** : Reduit les valeurs pour qu'elles se situent entre 0 et 1.

  ```
  from sklearn.preprocessing import MinMaxScaler
  scaler = MinMaxScaler()
  df['hauteur_normalized'] = scaler.fit_transform(df.hauteur_m.values.res
  hape(-1, 1))
  ```

- **Z-Score Standardization** : Centrage des données autour de 0 avec un écart type de 1.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df['hauteur_standard'] = scaler.fit_transform(df.hauteur_m.values.resha
pe(-1, 1))
```

- **Log Transformation** : Réduit l'amplitude des valeurs tout en conservant l'information.

```
df['hauteur_log'] = np.log(df.hauteur_m + 1)
```

4. Visualisation des Données

Utilisez des visualisations pour évaluer l'effet des transformations et assurer que les données sont bien préparées pour le modèle.

- **Histogrammes** : Comparez les distributions avant et après transformation.
- **Boxplots** : Visualisez la dispersion des valeurs et les outliers potentiels.

Résumé

Améliorer un jeu de données implique :

1. Identifier et traiter les données manquantes et les outliers.
2. Normaliser les valeurs pour garantir que toutes les variables ont une influence équitable sur le modèle.
3. Utiliser des techniques de transformation pour ajuster les distributions des données.

Cela prépare les données pour une utilisation plus efficace dans les modèles d'apprentissage automatique, augmentant ainsi leur précision et leur robustesse.

Transformation des Variables pour l'Apprentissage des Modèles (Exercice complet : "Notebook/P3C3_transformer_categories.ipynb"

1. Introduction aux Transformations des Variables

Les modèles d'apprentissage automatique nécessitent des données numériques. Les données textuelles doivent donc être converties en chiffres pour être utilisées. Les transformations dépendent du type de variable catégorique : binaire, ordonnée ou non ordonnée.

2. Cas Binaire

Pour les variables binaires (deux catégories exclusives comme "oui/non" ou "vrai/faux"), la transformation est simple. Par exemple, dans le dataset des arbres, la variable `remarquable` peut être transformée comme suit :

- **OUI** → 1
- **NON** → 0

Cette transformation peut être réalisée avec les commandes pandas suivantes :

```
df.loc[df.remarquable == 'NON', 'remarquable'] = 0
df.loc[df.remarquable == 'OUI', 'remarquable'] = 1
```

3. Cas Non Binaire avec Peu de Catégories

Pour les variables ayant plus de deux catégories mais peu de valeurs (comme les catégories ordinales), on peut les encoder en utilisant des nombres croissants tout en préservant l'ordre :

- Exemple : La variable `stade_de_developpement` avec les valeurs ordonnées telles que Jeune (arbre) < Adulte < Mature.

On peut utiliser le code suivant pour encoder ces catégories :

```
categories = ['Adulte', 'Jeune (arbre)Adulte', 'Jeune (arbre)', 'Mature']
for n, categorie in zip(range(1, len(categories) + 1), categories):
    df.loc[df.stade_de_developpement == categorie, 'stade_de_developpement']
= n
```

Pour une transformation plus structurée, on peut utiliser la librairie `category_encoders` :

```
from category_encoders.ordinal import OrdinalEncoder
mapping = [{'col': 'stade_de_developpement', 'mapping': {np.nan: 0, 'Jeune (a
rbre)': 1, 'Jeune (arbre)Adulte': 2, 'Adulte': 3, 'Mature': 4}}]
encoder = OrdinalEncoder(mapping=mapping)
stade = encoder.fit_transform(df.stade_de_developpement)
df['stade_de_developpement'] = stade.copy()
```

4. Cas Non Ordonné

Pour les variables non ordonnées, comme les couleurs ou les types, l'encodage doit éviter l'introduction d'un ordre fictif :

- **Encodage One-Hot** : Chaque catégorie est représentée par une colonne binaire indiquant sa présence ou absence. Cela génère N-1 nouvelles variables pour une variable ayant N valeurs distinctes.

Exemple avec la variable `domanialite` :

```
from sklearn import preprocessing
enc = preprocessing.OneHotEncoder()
labels = enc.fit_transform(df.domanialite.values.reshape(-1, 1)).toarray()
```

Après transformation, les nouvelles colonnes sont ajoutées au DataFrame et les colonnes originales sont supprimées.

5. Encodage Binaire

Pour éviter le problème de dimensionnalité lié à l'encodage One-Hot, l'encodage binaire est une alternative efficace :

- Chaque catégorie est transformée en une séquence binaire. Cela réduit le nombre de variables nécessaires pour représenter les catégories.

Exemple avec la variable `domanialite` :

```
from category_encoders.binary import BinaryEncoder
encoder = BinaryEncoder()
domanialite_bin = encoder.fit_transform(df.domanialite)
```

6. Feature Engineering

Le Feature Engineering consiste à créer de nouvelles variables basées sur les variables existantes pour améliorer les performances du modèle :

- **Mise en Intervalle** : Diviser des variables continues en intervalles.
- **Transformation** : Appliquer des fonctions mathématiques telles que puissance, racine carrée, ou logarithme.
- **Variables Flag** : Créer des indicateurs pour les valeurs manquantes ou des caractéristiques particulières.
- **Compilation** : Créer des variables composites à partir de plusieurs variables existantes.

Exemple Pratique : Dans un modèle de régression linéaire, ajouter le carré d'une variable (par exemple `tv`) peut améliorer la performance :

```
ventes ~ tv^2 + tv + radio + journaux
```

Résumé

Les techniques de transformation des variables sont essentielles pour préparer les données pour l'apprentissage automatique. Elles permettent de convertir les données catégoriques en formats numériques, facilitant ainsi l'apprentissage des modèles. L'encodage approprié et le Feature Engineering jouent un rôle crucial dans l'amélioration de la performance des modèles.

Optimisation du Modèle : Trouver le Bon Équilibre (Se référer au Notebook : “Notebook/P4C1_biais_overfit.ipynb”

Dans ce chapitre, nous allons explorer comment améliorer un modèle en ajustant ses hyperparamètres pour optimiser ses performances.

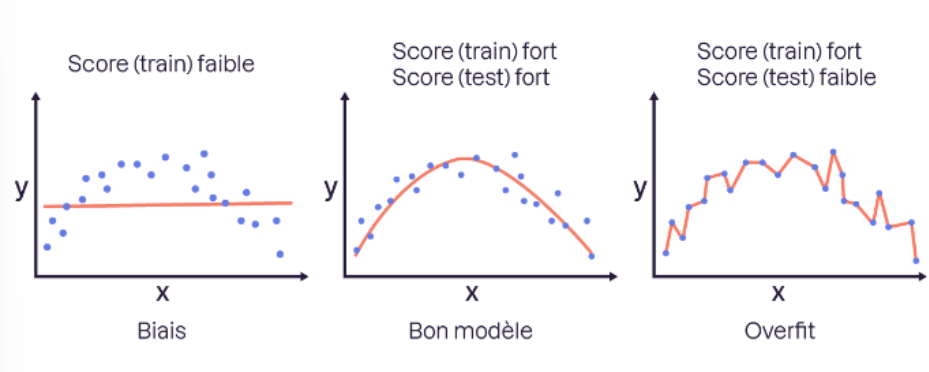
Identifier un Modèle sous-performant

Nous avons vu que pour évaluer un modèle, il est courant de diviser les données en ensembles d’entraînement et de test. La performance d’un modèle est souvent mesurée par son score sur ces deux ensembles :

- **Score d’entraînement (`score(train)`):** Performance du modèle sur les données d’entraînement.
- **Score de test (`score(test)`):** Performance du modèle sur les données de test.

L’idée est que le modèle doit apprendre les relations dans les données d’entraînement et être capable de généraliser ces relations aux données de test. Si un modèle sous-performe, il peut se manifester de trois façons :

1. **Modèle sous-performant :**
   - **Score d’entraînement faible**: Le modèle ne parvient pas à comprendre les données d’entraînement. Par conséquent, le score de test sera également faible. Ce phénomène est appelé **biais**.
2. **Modèle performant :**
   - **Scores d’entraînement et de test satisfaisants**: Le modèle est capable de généraliser et est prêt pour la production.
3. **Modèle sur-ajusté (Overfitting) :**
   - **Score d’entraînement élevé mais score de test faible**: Le modèle s’ajuste trop aux données d’entraînement et ne généralise pas bien aux nouvelles données. Cela indique un **overfitting**.

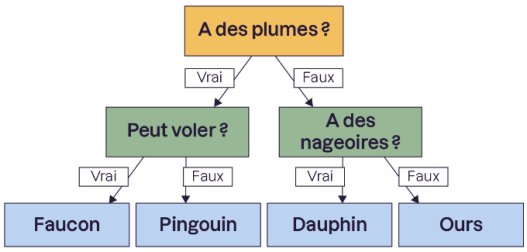


Nous illustrerons ces concepts en utilisant un **arbre de décision** comme modèle. Les arbres de décision sont idéaux pour démontrer les phénomènes de biais et d'overfitting.

Comprendre les Arbres de Décision

Un arbre de décision est une structure où chaque nœud représente une question sur les données et chaque branche représente une réponse possible. Voici un exemple simple :

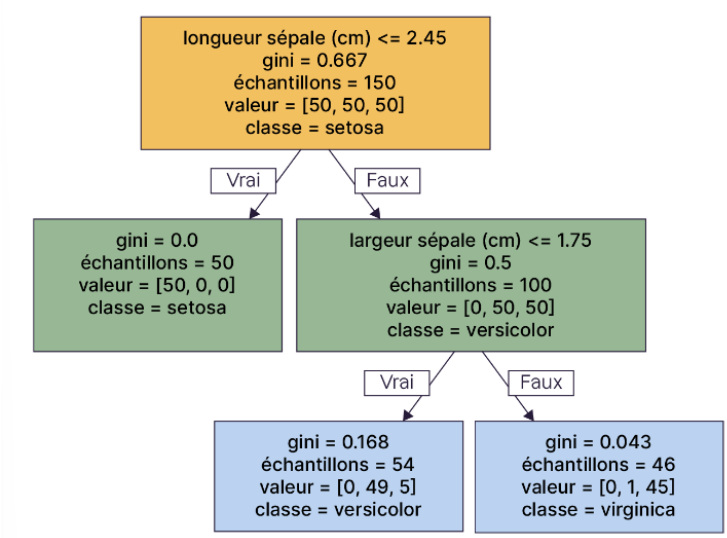
- **A-t-il des plumes ?**
  - Oui : Peut-il voler ?
    - Oui : **Faucon**
    - Non : **Pingouin**
  - Non : A-t-il des nageoires ?
    - Oui : **Dauphin**
    - Non : **Ours**



Arbre de décision simple à 2 niveaux de profondeur

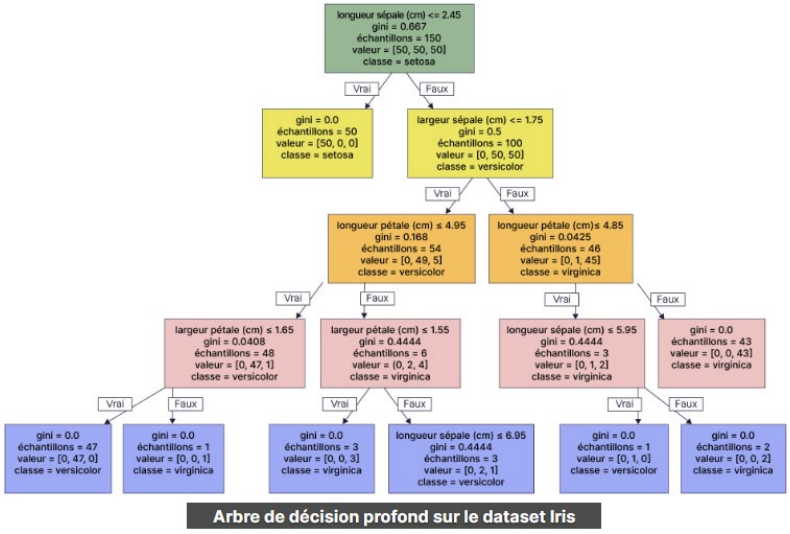
Les arbres de décision sont constitués de règles créées à partir des variables d'entrée. La profondeur de l'arbre, c'est-à-dire le nombre de niveaux de nœuds, joue un rôle crucial.

- **Arbre peu profond** : Moins complexe, avec peu de règles.



Arbre de décision de faible profondeur sur le dataset Iris

- **Arbre profond** : Plus complexe, avec beaucoup de règles.



Arbre de décision profond sur le dataset Iris

Le paramètre `max_depth` dans scikit-learn permet de limiter la profondeur de l'arbre, ce qui est essentiel pour contrôler l'overfitting.

Minimiser le Biais

Nous commençons avec un arbre de décision limité en profondeur (`max_depth = 3`), ce qui nous aide à éviter le biais mais peut également conduire à un modèle sous-performant.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score, confusion_matrix, classification_r
eport

# Chargement des données
filename = 'https://raw.githubusercontent.com/OpenClassrooms-Student-Center/8
063076-Initiez-vous-au-Machine-Learning/master/data/paris-arbres-numerical-20
23-09-10.csv'
data = pd.read_csv(filename)
X = data[['domanialite', 'arrondissement', 'libelle_francais', 'genre', 'espe
ce', 'circonference_cm', 'hauteur_m']]
y = data.stade_de_developpement.values

# Division des données
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, ran
dom_state=808)

# Entraînement du modèle avec max_depth = 3
clf = DecisionTreeClassifier(max_depth=3, random_state=808)
clf.fit(X_train, y_train)

# Évaluation
train_auc = roc_auc_score(y_train, clf.predict_proba(X_train), multi_class='o
vr')
test_auc = roc_auc_score(y_test, clf.predict_proba(X_test), multi_class='ovr'
)
print("train", train_auc)
print("test", test_auc)

# Matrice de confusion et rapport de classification
y_test_hat = clf.predict(X_test)
print(confusion_matrix(y_test, y_test_hat))
print(classification_report(y_test, y_test_hat))
```

Nous avons observé que le modèle avec `max_depth = 3` donne des scores d'AUC similaires sur les ensembles d'entraînement et de test, mais présente des faiblesses en précision pour certaines catégories. Cela suggère un biais.

Identifier et Compenser l'Overfitting

Pour tester l'overfitting, nous réentraînons le modèle sans limitation de profondeur (`max_depth = None`).

```
# Entraînement du modèle sans limite de profondeur
clf = DecisionTreeClassifier(max_depth=None, random_state=808)
clf.fit(X_train, y_train)

# Évaluation
train_auc = roc_auc_score(y_train, clf.predict_proba(X_train), multi_class='o
```

```
vr')
test_auc = roc_auc_score(y_test, clf.predict_proba(X_test), multi_class='ovr'
)
print("train", train_auc)
print("test", test_auc)
```

Nous constatons une amélioration des scores de test avec une profondeur illimitée, mais aussi un score d'entraînement nettement plus élevé, signe d'overfitting.

Trouver l'Équilibre Optimal

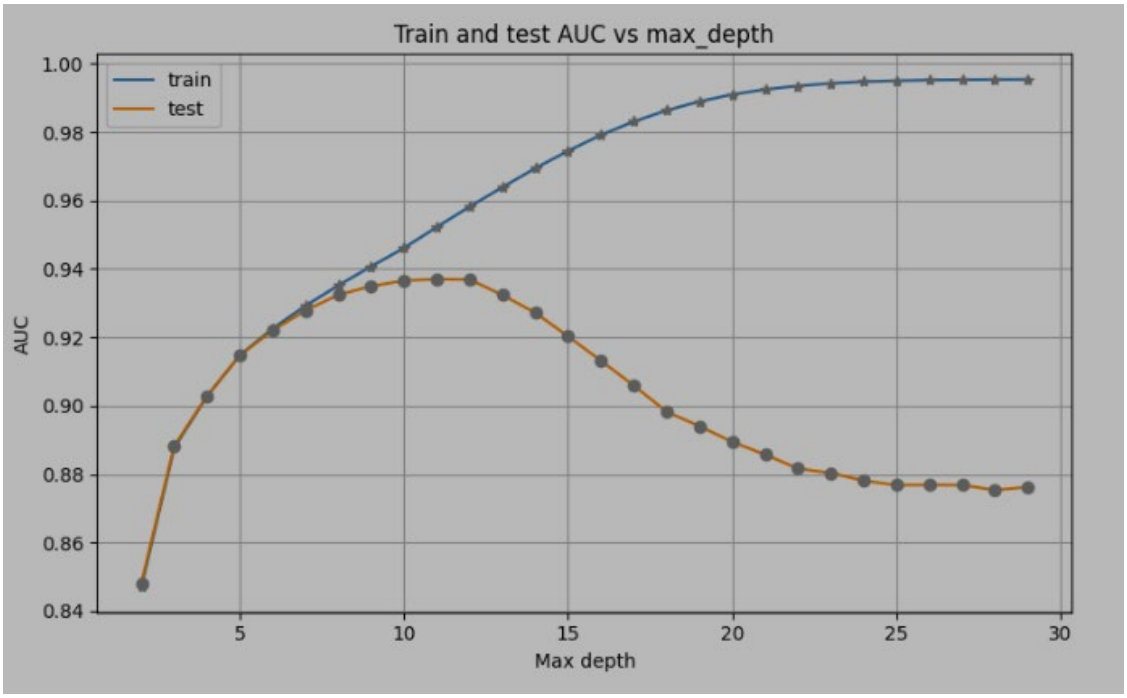
Pour équilibrer biais et overfitting, nous explorons différents niveaux de `max_depth` et enregistrons les scores :

```
import numpy as np
import pandas as pd

scores = []
for depth in np.arange(2, 30, 2):
    clf = DecisionTreeClassifier(max_depth=depth, random_state=808)
    clf.fit(X_train, y_train)
    train_auc = roc_auc_score(y_train, clf.predict_proba(X_train), multi_clas
s='ovr')
    test_auc = roc_auc_score(y_test, clf.predict_proba(X_test), multi_class='
ovr')
    scores.append({'max_depth': depth, 'train': train_auc, 'test': test_auc})

scores = pd.DataFrame(scores)
```

Les résultats montrent que les scores AUC pour l'entraînement continuent de croître, tandis que les scores AUC pour le test augmentent jusqu'à atteindre un maximum avant de diminuer. Cela illustre le compromis entre biais et overfitting.



Résumé

Pour remédier à l'overfitting, vous pouvez :

- **Augmenter la taille du dataset** : Plus de données permettent au modèle de mieux généraliser.
- **Utiliser la régularisation** : Techniques qui pénalisent la complexité du modèle pour éviter l'overfitting.

En résumé, l'optimisation des modèles implique de trouver le bon équilibre entre biais et complexité pour obtenir des performances optimales sur les données de test.

Augmentez la Robustesse de vos Modèles (Notebook complet : Notebook/P4C2_robustesse_modele.ipynb)

Compensez l'Overfit avec la Régularisation

La régularisation est une technique clé pour éviter que les modèles ne surajustent les données d'entraînement. Elle ajoute une contrainte sur le modèle afin de limiter sa capacité à coller aux données d'entraînement, ce qui améliore sa généralisation.

Prenons l'exemple de la régression linéaire. Avec la régularisation, nous utilisons le modèle **Ridge** dans scikit-learn. Ridge est une régression linéaire avec une composante de régularisation qui aide à éviter l'overfitting.

Fonction de coût du modèle Ridge :

`||y - Xw||^2_2 + α * ||w||^2_2`

- `||y - Xw||^2_2` : Erreur quadratique entre les prédictions `Xw` et les valeurs réelles `y`.
- `α * ||w||^2_2` : Terme de régularisation, où `||w||^2_2` est la norme L2 des coefficients `w`, et `α` est le paramètre de régularisation.

L'ajout du terme `α * ||w||^2_2` empêche les coefficients du modèle d'être trop grands, ce qui limite leur sensibilité aux fluctuations des données d'entraînement et réduit ainsi l'overfitting.

Étapes pour Illustrer l'Impact de la Régularisation

Étape 1 : Mise en Scène

Créons un dataset simple avec une seule variable prédictive et entraînons un modèle de régression linéaire sans régularisation.

```
from sklearn.datasets import make_regression
from sklearn.linear_model import Ridge

X, y = make_regression(n_samples=30, n_features=1, noise=40, random_state=0)
model = Ridge(alpha=0)
model.fit(X, y)
```

Étape 2 : L'Overfit

Considérons une régression polynomiale de degré 12, qui est plus complexe que la régression linéaire simple.

```
from sklearn.preprocessing import PolynomialFeatures

pol = PolynomialFeatures(degree=12, include_bias=False)
X_poly = pol.fit_transform(X)
model = Ridge(alpha=0)
model.fit(X_poly, y)
```

Le modèle polynomiale de degré 12 est sujet à un fort overfitting, car il s'ajuste de manière trop précise aux données d'entraînement.

Étape 3 : La Régularisation

Augmentons le paramètre de régularisation α pour observer comment il atténue l'overfitting.

```
alphas = [0.1, 1, 10, 100]
for alpha in alphas:
    model = Ridge(alpha=alpha)
    model.fit(X_poly, y)
    # Visualisation des résultats...
```

Vous remarquerez que l'augmentation de α rend la courbe de la régression moins ajustée aux données, ce qui réduit l'overfitting.

Autres Types de Régularisation

- **Régularisation L1 (Lasso)** : Utilise la norme L1 des coefficients, ce qui peut également réduire certains coefficients à zéro, effectuant ainsi une sélection de variables.
- **Arbres de Décision** : La régularisation peut se faire en limitant la profondeur de l'arbre ou en ajustant d'autres paramètres pour éviter l'overfitting.
- **Réseaux de Neurones** : La régularisation peut inclure des techniques comme le Dropout, mais le principe reste le même.

Implémentez la Validation Croisée

La validation croisée aide à éviter les problèmes dus à une répartition biaisée des échantillons entre les ensembles d'entraînement et de test. Elle consiste à diviser le dataset en plusieurs sous-ensembles, chaque sous-ensemble servant tour à tour de test, tandis que les autres servent à entraîner le modèle.

Validation croisée avec GridSearchCV :

```
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
import numpy as np

parameters = {'max_depth': np.arange(2, 30, 2)}
model = DecisionTreeClassifier(random_state=808)
clf = GridSearchCV(model, parameters, cv=5, scoring='roc_auc_ovr', verbose=1)
clf.fit(X, y)

# Résultats
best_params = clf.best_params_
best_score = clf.best_score_
best_model = clf.best_estimator_
```

- `best_params_` : La meilleure valeur des paramètres trouvée.
- `best_score_` : Le meilleur score obtenu lors de la validation croisée.
- `best_estimator_` : Le meilleur modèle avec les paramètres optimaux.

La validation croisée et GridSearchCV permettent de sélectionner les meilleurs paramètres et de valider la robustesse du modèle en moyennant les performances sur plusieurs sous-ensembles de données.

En Résumé

La régularisation aide à éviter que le modèle ne surajuste les données d'entraînement et perde sa capacité de généralisation. Les techniques de régularisation, comme Ridge (L2) et Lasso (L1), ainsi que les ajustements pour les arbres de décision et les réseaux de neurones,

sont cruciales pour améliorer la robustesse du modèle. La validation croisée, en utilisant des méthodes comme GridSearchCV, aide à s'assurer que le modèle est performant et généralise bien sur des données nouvelles.

Apprentissage d'ensemble (Notebook : Notebook/P4C2_robustesse_modele.ipynb)

L'apprentissage d'ensemble consiste à combiner plusieurs modèles sous-optimisés pour créer un modèle global plus performant, robuste et stable. Les deux principales techniques sont le **bagging** (forêts aléatoires) et le **boosting** (XGBoost). Cela permet de réduire le biais et la variance.

1. Bagging et Forêts Aléatoires

Le bagging (Bootstrap Aggregating) consiste à entraîner plusieurs versions d'un modèle de base, souvent un **arbre de décision peu profond**. Chaque modèle est formé sur une partie aléatoire des données d'entraînement. Le modèle final combine les prédictions des modèles faibles :

- **Régression** : moyenne des prédictions
- **Classification** : vote majoritaire

Les paramètres clés d'une forêt aléatoire incluent :

- `n_estimators` : nombre d'arbres
- `max_features` : nombre de variables utilisées par chaque arbre
- `bootstrap` : technique d'échantillonnage ("Le **bootstrap** est une technique d'échantillonnage très utile. À chaque sélection d'un échantillon, l'élément est remis dans l'ensemble source. L'ensemble échantillonné contiendra donc des doublons, mais cela offre des propriétés statistiques fortes utiles.")

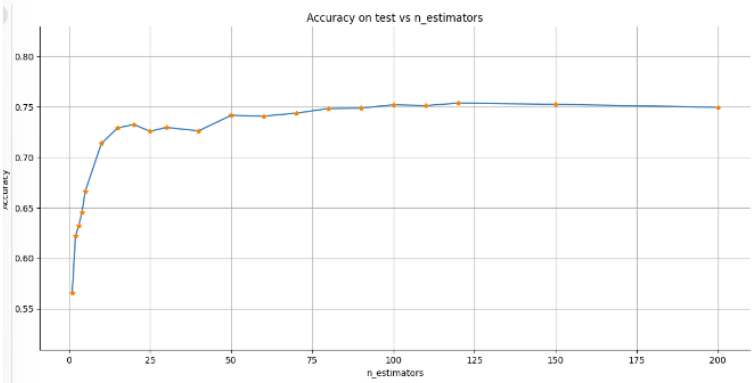
Exemple de création de forêt aléatoire en scikit-learn :

```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(n_estimators=100, max_depth=2, random_state=8)
```

2. Réduction du biais et de la variance

- **Réduction du biais** : augmenter le nombre d'arbres dans la forêt diminue le biais, comme montré par la croissance de l'accuracy jusqu'à un plateau.



Score d'accuracy en fonction du nombre d'arbres de la forêt aléatoire

- **Réduction de la variance** : le bagging réduit l'overfitting en augmentant le nombre d'arbres ou en réduisant `max_depth`.

3. Importance des variables

Les forêts aléatoires permettent d'évaluer l'importance des variables, en fonction de leur utilisation dans les arbres.

Exemple de calcul d'importance :

```
print(clf.feature_importances_)
```

4. Boosting (XGBoost et Gradient Boosting)

Le boosting, contrairement au bagging, entraîne les modèles séquentiellement. Chaque nouveau modèle se concentre sur les erreurs des prédictions précédentes. Le **Gradient Boosting** ajuste progressivement les erreurs via le paramètre **learning_rate**. Une version courante est **XGBoost**.

Exemple de Gradient Boosting avec scikit-learn :

```
from sklearn.ensemble import GradientBoostingClassifier

clf = GradientBoostingClassifier(n_estimators=500, max_depth=2, learning_rate
=0.1)
```

5. Tuning des paramètres

Pour ajuster un modèle de Gradient Boosting, les principales étapes sont :

1. Optimiser le nombre d'arbres via une validation croisée.
2. Ajuster le **learning_rate** : un taux élevé converge rapidement mais peut conduire à des erreurs, tandis qu'un taux faible est plus précis mais plus lent.

6. Benchmarking

Il est recommandé de commencer avec une régression simple (linéaire ou logistique) pour établir un benchmark de performance. Ensuite, utiliser les forêts aléatoires pour améliorer les résultats, puis passer au Gradient Boosting si nécessaire.

Résumé:

- **Forêts aléatoires** : combinaison de plusieurs arbres de décision entraînés en parallèle, robustes contre l'overfitting.
- **Gradient Boosting** : modèles séquentiels améliorant les erreurs à chaque étape, plus performants mais plus complexes à paramétrer.
- **Benchmark** : une régression simple aide à fixer des attentes avant de passer aux modèles ensemblistes.