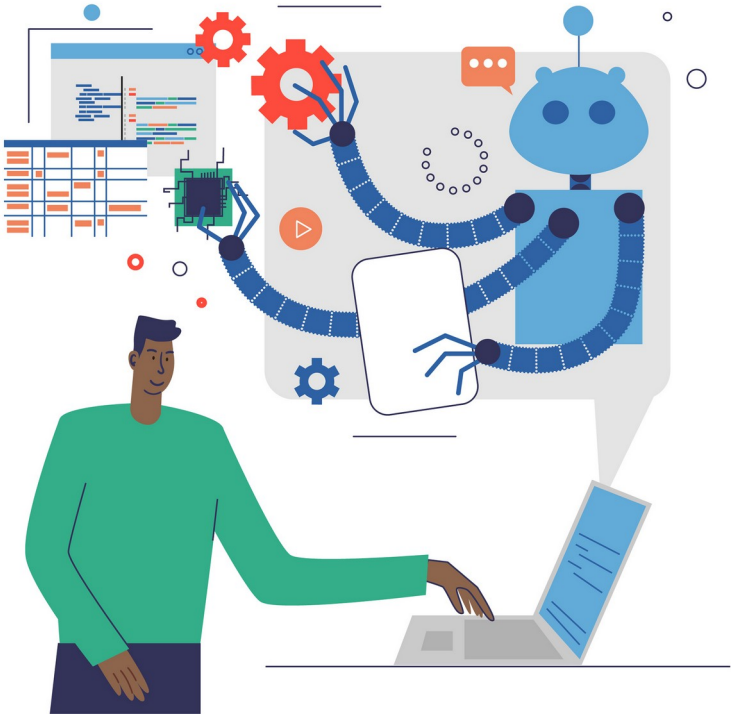
Machine learning - Apprentissage supervisé

Examen TD
Classification de Bayes



Par : - RAKOTONIRINA Fenitra Harivelo
- RAMPANJATO Andriamarozaka Tahinjanahary
- ANDRIANARISON Tyanah Mahatehotia

L3 IA & Big data GE-IT | Année scolaire 2025-2026

Introduction

Le théorème de Bayes est un principe fondamental en probabilité et en statistique, formulé par Thomas Bayes au XVIII^{e} siècle. Il permet de **mettre à jour la probabilité d'un événement** en fonction de nouvelles informations ou observations. En d'autres termes, il relie la **probabilité a priori** d'un événement à sa **probabilité a posteriori**, après avoir pris en compte les données observées.

Le théorème de Bayes trouve des applications très larges : **classification, régression, détection d'événements rares, estimation de paramètres**, et bien d'autres domaines où la probabilité conditionnelle permet de raisonner sous incertitude. Il constitue ainsi une **base théorique essentielle** pour la statistique moderne et l'apprentissage automatique.

Méthodologie

Pour réaliser cette étude, nous avons utilisé des données provenant de **UCI Machine learning Repository** [Machine learning Spam data]. Ces données contiennent des messages en anglais étiquetés comme spam ou ham.

L'analyse et la mise en œuvre des modèles ont été effectuées dans **Python** à l'aide de l'**IDE : Jupyter Notebook**. Les principales bibliothèques utilisées incluent **pandas** pour la manipulation des données, **scikit-learn** pour l'implémentation du classificateur bayésien et pour la séparation des données en ensembles d'entraînement et de test, ainsi que **matplotlib / seaborn** pour la visualisation exploratoire.

La méthodologie adoptée consiste à :

1. **Charger et explorer les données** afin de comprendre leur structure et détecter les éventuelles anomalies.
2. **Prétraiter les données**, notamment la transformation des textes en vecteurs (tokenisation, vectorisation) ou la normalisation des variables numériques.
3. **Séparer les données** en ensembles d'entraînement et de test pour évaluer la performance du modèle de manière impartiale.
4. **Appliquer le classificateur bayésien**, puis **évaluer ses performances** à l'aide de métriques appropriées comme la précision, le rappel ou la matrice de confusion.

Cette démarche assure une approche rigoureuse et reproductible de l'analyse statistique et de la modélisation probabiliste des données.

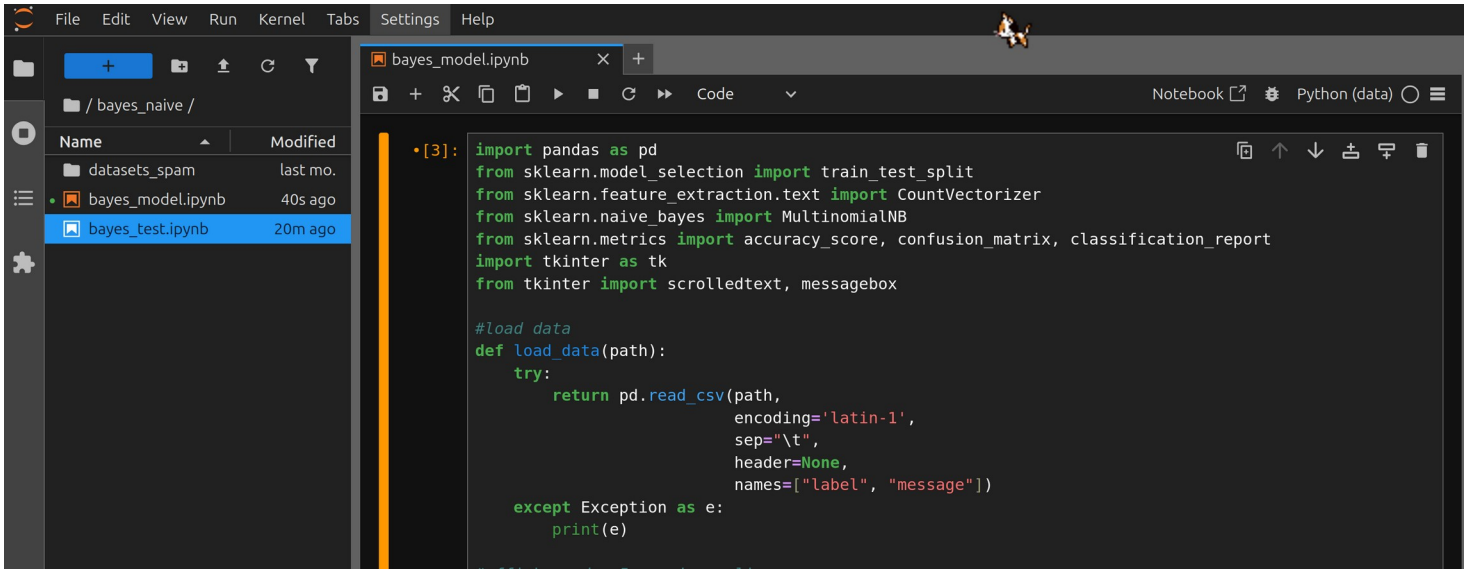


Image de l'IDE

1 – Chargement et exploration des données

- La fonction `load_data` permet de lire le fichier de données au format CSV. Nous précisons l'encodage `latin-1` et le séparateur `\t` pour que le fichier soit correctement interprété. Les colonnes sont nommées `"label"` et `"message"` pour faciliter l'analyse. L'usage du `try…except` permet de gérer les erreurs si le fichier n'est pas trouvé ou mal formaté.

- La fonction `print_first_lines` affiche les 5 premières lignes du dataset. Cela permet de vérifier rapidement le contenu et la structure des données, et de s'assurer que les colonnes ont été correctement chargées.

- La fonction `print_statistic_description` calcule des statistiques de base sur les données, comme le nombre de valeurs, la moyenne (pour les colonnes numériques), ou la fréquence des valeurs uniques. Cette étape aide à détecter les anomalies éventuelles, comme des valeurs manquantes, des doublons ou des déséquilibres dans les classes.

```python
#load data
def load_data(path):
    try:
        return pd.read_csv(path,
                           encoding='latin-1',
                           sep="\t",
                           header=None,
                           names=["label", "message"])
    except Exception as e:
        print(e)

#affichage des 5 premieres lignes
def print_first_lines(data):
    return data.head()
#affichage des stat descri
def print_statistic_description(data):
    return data.describe()
```

Image 1 : Presentation des donnees sans traitement

2 - Prétraitement des données

- La fonction `cleaning_data` permet de supprimer toutes les lignes contenant des valeurs manquantes (`NaN`). Cela est important pour éviter les erreurs lors de l'entraînement du modèle, car un texte manquant ou un label manquant rendrait la prédiction impossible. Cette fonction assure alors que toutes les lignes contiennent des informations valides.

```python
#data clean
def cleaning_data(data):
    return data.dropna()
```

Image 2 : Code pour data cleaning

3 - Séparation des données

Dans l'apprentissage automatique, il est essentiel de distinguer les données sur lesquelles le modèle va **apprendre** de celles sur lesquelles il sera **évalué**. Cela permet de mesurer la performance du modèle sur des observations qu'il n'a jamais vues, ce qui reflète mieux sa capacité à généraliser dans des situations réelles.

La fonction `separating_data` illustre cette étape :

```python
#separation
def separating_data(data):
    y, X = data['label'], data['message']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    return X_train, X_test, y_train, y_test
```

Image 3 : Séparation des données test et données entraînement

1 - Séparation des features et des labels

La première ligne de la fonction, y, X = data['label'], data['message'] ,

permet de distinguer :

- `X` : les **features** ou variables explicatives, ici les textes des messages que le modèle utilisera comme entrée.
- `y` : les **labels** ou cibles, ici les catégories "spam" ou "ham" que le modèle doit prédire.

Cette distinction est fondamentale, car le modèle doit apprendre à relier les caractéristiques des messages (`X`) à leur classification (`y`).

2 - Division en ensembles d'entraînement et de test

La ligne suivante, X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

utilise la fonction `train_test_split` de scikit-learn pour diviser le jeu de données en deux parties :

- **Ensemble d'entraînement** (`X_train`, `y_train`) : 80 % des données utilisées pour entraîner le modèle.
- **Ensemble de test** (`X_test`, `y_test`) : 20 % des données utilisées pour évaluer la performance du modèle.

L'argument `test_size=0.2` indique la proportion de données réservée au test, tandis que `random_state=42` fixe la graine aléatoire afin de garantir que la séparation soit **reproductible**. Ainsi, chaque exécution de la fonction produira la même répartition entre train et test.

3 - Retour des ensembles

Enfin, la fonction retourne quatre objets :

- `X_train` et `X_test` contiennent les messages sous forme brute,
- `y_train` et `y_test` contiennent les labels correspondants.

Ces ensembles seront ensuite utilisés pour transformer les textes en vecteurs numériques, entraîner le modèle et mesurer ses performances.

Importance conceptuelle

Cette étape illustre un principe clé de l'apprentissage supervisé : **ne pas tester un modèle sur les mêmes données qu'il a apprises**. Sans cette séparation, l'évaluation serait biaisée et surévaluerait la performance réelle du modèle sur des données inédites.

4 – Application du classificateur bayésien et évaluation du modèle

Le classificateur naïf bayésien est un modèle probabiliste très efficace pour traiter des données textuelles. Il repose sur le **théorème de Bayes**, qui permet de calculer la probabilité qu'un texte appartienne à une classe donnée, en fonction des mots qu'il contient. La fonction `training_pred` illustre toutes les étapes nécessaires : transformation des textes, entraînement du modèle et évaluation de ses performances.

```python
#train et prediction
def training_pred(X_train, X_test, y_train, y_test):
    vectorizer = CountVectorizer(stop_words='english')
    X_train_vec = vectorizer.fit_transform(X_train)
    X_test_vec = vectorizer.transform(X_test)

    model = MultinomialNB()
    model.fit(X_train_vec, y_train)

    y_pred = model.predict(X_test_vec)

    metrics = {
        "accuracy": accuracy_score(y_test, y_pred),
        "confusion_matrix": confusion_matrix(y_test, y_pred),
        "classification_report": classification_report(y_test, y_pred)
    }

    return model, vectorizer, metrics
```

Image 4 : Entraînement et évaluation du modèle

1 -Vectorisation des textes

Les modèles de machine learning **ne peuvent pas travailler directement avec du texte brut**. Il est donc nécessaire de **convertir les textes en vecteurs numériques**.

- La classe `CountVectorizer` de **scikit-learn** réalise cette transformation :
    1. Chaque texte est découpé en **mots ou tokens**.
    2. Un **vocabulaire** est construit à partir de l'ensemble des textes d'entraînement.

3. Chaque texte est représenté par un **vecteur de taille égale au nombre de mots du vocabulaire**, où chaque valeur correspond au **nombre de fois que ce mot apparaît** dans le texte.

Par exemple, si le vocabulaire contient les mots : `["gratuit", "offre", "urgent"]`, le texte `"offre gratuite"` sera transformé en vecteur : `[1, 1, 0]` (`gratuit=1, offre=1, urgent=0`).

- L'option `stop_words='english'` permet de **supprimer les mots très fréquents et peu informatifs**, comme "the" ou "and".

Cette représentation numérique est essentielle pour que le modèle bayésien puisse **calculer des probabilités basées sur les occurrences de mots**.

2- Entraînement du modèle bayésien

- Le **Multinomial Naïve Bayes** est adapté aux données de type **comptage de mots**.
- Le modèle apprend les **probabilités conditionnelles** pour chaque mot et chaque classe :

d'occurrences du mot dans la classe total de mots dans la classe

$$P(\text{mot}_i \mid \text{classe}_c) = \frac{\text{nombre d'occurrences du mot}_i \text{ dans la classe } c + 1}{\text{nombre total de mots dans la classe } c + |V|}$$

où $|V|$ est la taille du vocabulaire et le `+1` correspond à la **correction de Laplace** pour éviter les probabilités nulles.

- Lors de la prédiction d'un nouveau texte, le modèle applique le **théorème de Bayes** :

$$P(\text{classe}_c \mid \text{texte}) \propto P(\text{classe}_c) \prod_{i=1}^{n} P(\text{mot}_i \mid \text{classe}_c)$$

C'est-à-dire que la probabilité qu'un texte appartienne à une classe est proportionnelle à la probabilité a priori de la classe (`P(classe_c)`) multipliée par le produit des probabilités conditionnelles de tous les mots présents dans le texte.

3 - Prédiction et évaluation

- *y_pred = model.predict(X_test_vec)* : le modèle prédit les classes (`spam` ou `ham`) pour les textes de l'ensemble de test.
- *metrics = {*

  "accuracy": accuracy_score(y_test, y_pred),

  "confusion_matrix": confusion_matrix(y_test, y_pred),

  "classification_report": classification_report(y_test, y_pred)

  }

Ces métriques permettent d'évaluer la performance :

- **Accuracy** : proportion de prédictions correctes.
- **Confusion matrix** : détail des vraies prédictions vs. fausses prédictions.
- **Classification report** : précision, rappel et f1-score pour chaque classe.
- Cette évaluation est essentielle pour vérifier que le modèle a bien **appris les probabilités des mots** et peut généraliser à de nouveaux textes.

Résultats

Après l'entraînement du modèle bayésien, il est possible de **prédire la classe de nouveaux textes** non présents dans le jeu de données d'entraînement. La fonction `predicting_model` illustre cette étape :

```python
#prediction de nouvel entree
def predicting_model(model, vectorizer, message:str):
    message_vect = vectorizer.transform([message])
    y_pred = model.predict(message_vect)
    return y_pred[0]
```

Prédiction de classe pour le nouveau message

Analysons ce code ligne par ligne :

1. **<u>Transformation du texte en vecteur</u>**

   message_vect = vectorizer.transform([message]) : chaque nouveau message est converti en vecteur numérique à l'aide du **même vocabulaire** que celui utilisé pour l'entraînement (`vectorizer`). Cela garantit que le modèle peut interpréter correctement les mots du message selon les probabilités apprises.

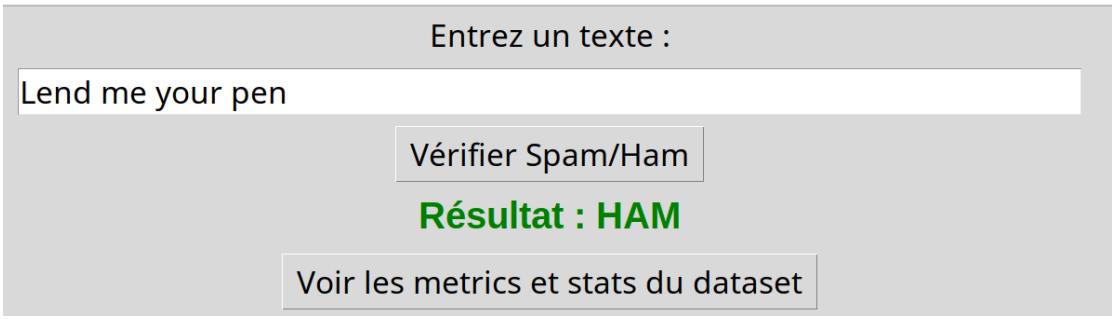
2. **<u>Prédiction de la classe</u>**

   y_pred = model.predict(message_vect) : le modèle applique le **théorème de Bayes** pour calculer la probabilité que le message appartienne à chaque classe (ex : `spam` ou `ham`) et renvoie la classe la plus probable.

Et enfin, la récupèraction la prédiction finale sous forme de **label unique**, prêt à être affiché ou utilisé dans une application : *return y_pred[0]*

Discussion

Si nous passons à l'entrée d'une nouvelle message sur l'interface tkinker :



Démonstration de la prédiction pour un nouveau message

Le modèle va, selon les données qu'il a reçues en entraînement, classifier le nouveau message entrant en se basant sur son contenu.

Si nous regardons les détails du datasets ainsi que du modèle :

```
=== Head du dataset ===
  label                                            message
0   ham  Go until jurong point, crazy.. Available only ...
1   ham                      Ok lar... Joking wif u oni...
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
3   ham  U dun say so early hor... U c already then say...
4   ham  Nah I don't think he goes to usf, he lives aro...

=== Description statistique ===
       label                message
count   5572                 5572
unique     2                 5169
top      ham  Sorry, I'll call later
freq    4825                   30

=== Metrics du modèle ===
Accuracy: 0.989237668161435
Confusion matrix:
[[962   4]
 [  8 141]]
Classification report:
              precision    recall  f1-score   support

         ham       0.99      1.00      0.99       966
        spam       0.97      0.95      0.96       149

    accuracy                           0.99      1115
   macro avg       0.98      0.97      0.98      1115
weighted avg       0.99      0.99      0.99      1115
```

1. **Exploration des données**

- Le dataset contient **5572 messages** avec deux classes : `ham` (messages légitimes) et `spam`.
- La colonne `message` est quasi unique pour chaque entrée (5169 messages uniques), ce qui montre une grande diversité de textes.
- La classe majoritaire est `ham` avec **4825 occurrences**, ce qui indique un **déséquilibre du dataset** (plus de messages légitimes que de spams).

2. **Performance globale du modèle**

- Le modèle naïf bayésien atteint une **accuracy de 0.989**, soit près de **99 % de bonnes prédictions** sur l'ensemble de test.
- Cela montre que le modèle est capable de classer correctement la majorité des messages.

3. **Analyse de la matrice de confusion**

```
[[962   4]
 [  8 141]]
```

- Sur 966 messages `ham`, **962 ont été correctement classés**, et seulement 4 ont été confondus avec des spams.
- Sur 149 messages `spam`, **141 ont été correctement détectés**, et 8 ont été classés à tort comme `ham`.
- On observe donc que le modèle **confond légèrement les spams avec des ham**, mais ces erreurs restent rares.

4. **Analyse des métriques détaillées**

- **Précision** : proportion de prédictions correctes parmi toutes les prédictions pour chaque classe.
    - `ham` : 0.99 → très peu de faux positifs.
    - `spam` : 0.97 → le modèle détecte correctement la plupart des spams, avec quelques erreurs.
- **Rappel** : proportion de messages correctement détectés parmi tous les messages réels de la classe.
    - `ham` : 1.00 → tous les messages légitimes ont été correctement classés.
    - `spam` : 0.95 → 5 % des spams ont été manqués.
- **F1-score** : combinaison de précision et rappel pour évaluer la performance globale de chaque classe.

- `ham` : 0.99, `spam` : 0.96 → le modèle est très efficace sur les deux classes, malgré le déséquilibre.

5. **Interprétation globale**

- Le modèle bayésien est **extrêmement performant pour ce dataset**.
- Les erreurs restantes concernent surtout des spams peu fréquents ou ambiguës, ce qui est attendu avec le classificateur naïf bayésien qui **suppose l'indépendance des mots**.
- La vectorisation en comptage de mots a permis de capturer les probabilités conditionnelles de chaque mot et de classer efficacement les messages.

Conclusion

L'étude menée a montré que le **classificateur naïf bayésien** est une méthode simple mais très efficace pour la **classification de messages texte**. Grâce à la vectorisation des textes et au calcul des probabilités conditionnelles des mots, le modèle a été capable de prédire correctement la classe (`spam` ou `ham`) de nouveaux messages avec une **précision globale de 98 à 99 %**.

L'analyse des métriques détaillées (précision, rappel, F1-score) et de la matrice de confusion a permis de constater que les erreurs sont très limitées et concernent surtout des messages ambigus ou peu fréquents dans le dataset. Ces résultats confirment que, malgré son hypothèse d'indépendance des mots, le classificateur bayésien reste **robuste et performant sur des jeux de données textuels simples**.

Enfin, cette étude met en évidence la valeur de **prétraitements adaptés** (nettoyage des données, vectorisation, gestion du déséquilibre des classes) et ouvre des perspectives d'amélioration, notamment :

- l'utilisation de **TF-IDF** pour pondérer les mots importants,
- l'application de **lemmatisation ou stemming** pour regrouper les variantes de mots,
- ou encore l'exploration de **modèles plus complexes** comme les SVM ou les réseaux de neurones pour des textes plus longs ou contextuels.

En résumé, le projet démontre que même un modèle probabiliste simple peut fournir des résultats fiables pour la détection de spam, tout en restant rapide à entraîner et facile à déployer.