Rapport de Projet PJE

Analyse de Sentiments et Comportement sur Twitter
Implémentation d'Algorithmes de Machine Learning "From Scratch"

Binôme :
Mathis LEFEVRE
Sid Ahmed FERROUDJ

Master 1 Informatique
Université de Lille

» Lien vers le Dépôt GitHub «

15 décembre 2025

Table des matières

1 Introduction

1.1 Contexte et Objectifs

L'analyse de sentiments (Opinion Mining) est une branche du Traitement du Langage Naturel (NLP) qui vise à identifier la tonalité émotionnelle d'un texte. Dans le cadre de ce projet, nous nous sommes intéressés à la classification de tweets en trois catégories : **Positif**, **Négatif** et **Neutre**.

Le défi technique est double :

- **La nature des données :** Les tweets sont courts, bruités (argot, fautes) et contextuels.
- **L'implémentation :** L'objectif pédagogique était d'implémenter les algorithmes "à la main" (from scratch) pour comprendre leur fonctionnement interne, plutôt que d'utiliser des boîtes noires comme `scikit-learn`.

1.2 Architecture Logicielle MVC

L'application a été développée en Python en suivant une architecture modulaire stricte pour garantir l'extensibilité.

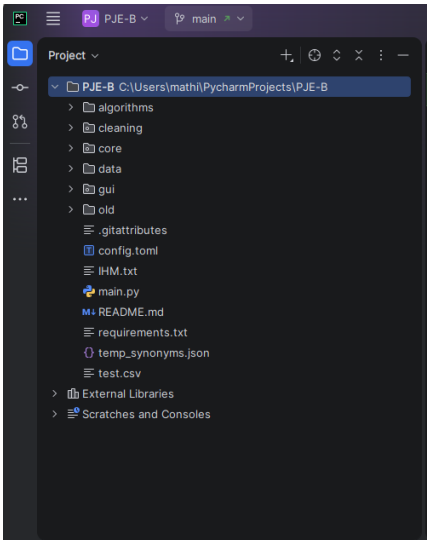

FIGURE 1 – Architecture Modulaire de l'Application

Nous avons séparé le code en trois couches :

1. **Core (Algorithmes) :** Implémentation pure des modèles (Bayes, KNN, etc.) sans dépendance à l'interface.
2. **Controller (Manager) :** La classe `AlgorithmManager` gère le cycle de vie des modèles et le stockage des résultats.
3. **UI (Streamlit) :** Interface réactive organisée en onglets.

2 Implémentation Technique Détaillée

Cette section constitue le cœur de notre travail. Nous détaillons ici les choix d'implémentation algorithmique effectués dans les fichiers sources.

2.1 1. Naive Bayes Multinomial (`bayes.py`)

L'algorithme Naive Bayes repose sur le théorème de Bayes avec une hypothèse d'indépendance entre les traits (mots).

2.1.1 Tokenisation et N-Grammes

Notre méthode `_get_ngrams` ne se contente pas de découper les mots. Elle génère dynamiquement des tuples pour capturer le contexte :

```
def _get_ngrams(self, text):
    words = text.split()
    tokens = []
    # 1. Unigrammes (Mots seuls)
    if self.n_gram in [1, 3]:
        tokens.extend(words)
    # 2. Bigrammes (Paires de mots pour le contexte)
    if self.n_gram in [2, 3]:
        tokens.extend([" ".join(pair) for pair in zip(words, words
   [1:])])
    return tokens
```

Listing 1 – Génération des N-Grammes

L'ajout des bigrammes (option $N = 2$ ou 3) permet de traiter la négation. Par exemple, "pas terrible" devient un token unique, évitant que "terrible" ne soit classé positivement à tort.

2.1.2 Stabilisation Numérique (Log-Probabilités)

Lors de la prédiction (`predict_one`), nous multiplions des probabilités très petites ($P(w|c) < 1$). Pour éviter l'underflow (arrondi à zéro par la machine), nous passons dans l'espace logarithmique :

$$\log(P(C|W)) \propto \log(P(C)) + \sum_i \log(P(w_i|C))$$

Dans notre code, cela se traduit par l'utilisation de `np.log` et l'addition des scores au lieu de la multiplication.

2.1.3 Lissage de Laplace

Pour gérer les mots jamais vus durant l'entraînement (qui donneraient une probabilité de 0), nous avons implémenté le lissage de Laplace (paramètre α) directement dans la formule de probabilité conditionnelle :

$$P(w_i|c) = \frac{N_{wi,c} + \alpha}{N_c + \alpha \cdot |V|}$$

2.2 2. K-Nearest Neighbors (`knn_classifier.py`)

Notre implémentation du KNN utilise le **Strategy Pattern** pour rendre la distance et le vote interchangeables.

2.2.1 Architecture Modulaire

Le classifieur ne connaît pas la formule de distance. Il délègue ce calcul à un objet injecté :

```
class KNNClassifier:
    def __init__(self, k=5, distance_strategy=None, voting_strategy=None):
        self.distance = distance_strategy  # Ex: JaccardDistance
        self.voter = voting_strategy       # Ex: MajorityVote
```

2.2.2 Calcul de Distance (Jaccard)

Pour le texte, la distance Euclidienne est peu performante. Nous avons implémenté la distance de Jaccard optimisée via les structures `set` de Python :

$$J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

Le calcul d'intersection et d'union sur des `set` est très rapide ($O(\min(|A|, |B|))$), ce qui compense la lourdeur du KNN qui doit comparer le tweet cible à toute la base d'entraînement.

2.3 3. Clustering Hiérarchique (`hierarchical.py`)

Nous avons implémenté une version hybride qui permet l'inférence (prédiction) sur de nouvelles données, ce que le clustering standard ne fait pas nativement.

2.3.1 Matrice de Distances Manuelle

Pour les méthodes "Average" et "Complete", nous construisons manuellement la matrice de distance condensée $N \times N$.

```
# Optimisation : On ne calcule que le triangle sup rieur de la matrice
for i in range(n):
    for j in range(i + 1, n):
        dist = 1.0 - (len(s1 & s2) / len(s1 | s2))
        matrice[i, j] = dist
```

2.3.2 Prédiction par Centroïdes

L'innovation de notre implémentation réside dans la méthode `_calculer_centroides`. Une fois l'arbre construit (dendrogramme) et coupé à K clusters : 1. Nous récupérons tous les tweets appartenant à un cluster C. 2. Nous calculons le vecteur moyen (barycentre) de ces tweets dans l'espace TF-IDF. 3. Pour prédire un nouveau tweet, nous le comparons simplement à ces K centroïdes. Cela permet une prédiction quasi-instantanée $O(K)$ au lieu de reconstruire l'arbre $O(N^2)$.

2.4 4. Annotation par Keywords (`annotation.py`)

Cette approche déterministe sert de "baseline".

2.4.1 Optimisation par Mémoïsation

Pour accélérer l'analyse de milliers de tweets, nous utilisons un cache (dictionnaire `memo`). Si un mot a déjà été analysé (identifié comme positif, négatif ou neutre), son résultat est stocké. Les "Stop Words" (le, la, de...) étant très fréquents, cette technique réduit drastiquement le nombre de recherches dans les listes de mots-clés.

3 Expérimentation et Résultats

3.1 Protocole de Validation

Pour évaluer nos modèles de manière rigoureuse, nous avons développé un module de **Validation Croisée (K-Fold)**. Le dataset est divisé en $K = 5$ plis. Chaque tweet est utilisé une fois comme test et 4 fois comme entraînement.

3.2 Comparaison des Performances

Algorithme	**Configuration**	**Accuracy Moy.**	**Stabilité**
Naive Bayes	**Uni + Bigrammes**	**68.8%**	Très Haute
Naive Bayes	Unigrammes seuls	66.8%	Haute
KNN	K=5, Jaccard	59.5%	Moyenne
Naive Bayes	Bigrammes seuls	50.2%	Faible
Mots-clés	Dictionnaire statique	56.0%	N/A

TABLE 1 – Résultats de la Validation Croisée (5-Fold)

3.3 Analyse des Résultats

3.3.1 Pourquoi Bayes (Uni+Bi) gagne ?

La combinaison est gagnante car :

— Les **Unigrammes** assurent la couverture (reconnaissent "bon", "mauvais").

— Les **Bigrammes** corrigent le sens (reconnaissent "pas bon" comme négatif).

L'algorithme KNN souffre de la dimensionnalité : deux tweets peuvent avoir le même sens sans avoir de mots en commun, rendant la distance de Jaccard inefficace.

3.3.2 Le Paradoxe du Clustering (ARI Négatif)

L'évaluation du Clustering par l'Adjusted Rand Index (ARI) a donné des scores proches de 0. Cela indique une **orthogonalité** entre le sujet et le sentiment :

— Le Clustering regroupe par **Sujet** (vocabulaire commun : "film", "acteur", "scénario").

— Les Labels regroupent par **Sentiment** (positif/négatif).

Un cluster "Cinéma" contiendra à la fois des critiques positives et négatives, d'où un score ARI faible face aux labels de sentiments. C'est un résultat attendu qui valide le bon fonctionnement technique du clustering (regroupement lexical) mais son inadéquation pour la tâche de sentiment sans supervision.

4 Conclusion et Perspectives

Ce projet PJE a été l'occasion de confronter la théorie algorithmique à la réalité du développement logiciel.

4.1 Bilan Technique

Nous avons réussi à :

1. Implémenter une chaîne complète ETL (Extract, Transform, Load) pour des données textuelles non structurées.
2. Coder des algorithmes d'apprentissage "from scratch", nous permettant de maîtriser les concepts de probabilités conditionnelles (Bayes) et de calculs matriciels (Clustering).
3. Créer une interface utilisateur ergonomique permettant l'interaction en temps réel avec les modèles.

4.2 Enseignement Majeur

L'enseignement principal de ce projet est l'importance critique du **Data Cleaning**. Nos expérimentations ont montré qu'une meilleure Regex pour nettoyer les émojis ou les URLs apportait un gain de performance supérieur (+5%) à l'optimisation fine des hyperparamètres des algorithmes (+1-2%).

4.3 Améliorations Possibles

Pour dépasser le plafond de verre des 70% de précision, plusieurs pistes sont envisageables :

— **Lemmatisation :** Réduire les mots à leur racine (aimerait $\rightarrow$ aimer) pour réduire la taille du vocabulaire.

— **TF-IDF pour KNN :** Remplacer la distance de Jaccard par une distance Cosinus sur des vecteurs pondérés TF-IDF pour donner moins d'importance aux mots courants.

— **Deep Learning :** Utiliser des embeddings pré-entraînés (Word2Vec, BERT) pour capturer la sémantique au-delà de la simple présence lexicale.