République Algérienne Démocratique et Populaire

Ministère de L'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie

Houari Boumediene

FACULTÉ D'INFORMATIQUE

PROJET RNAA

Elaboré par :

- SAAL Racim 202031049366
- FERCHICHI Manel 202031036637

Section : SII

Introduction Générale :

Dans ce projet nous allons nous intéresser à une discipline de l'apprentissage automatique qui permet aux machines d'interpréter, manipuler et comprendre le langage humain : Traitement du langage naturel ou NLP.

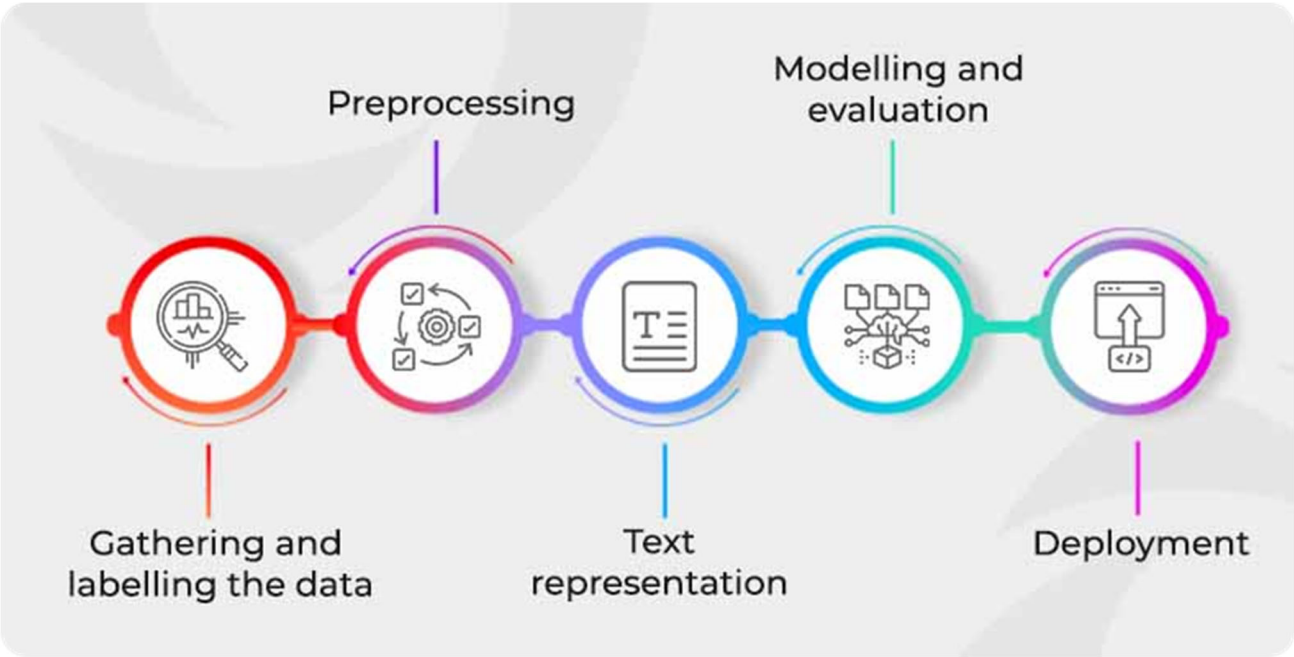
Les cas d'utilisation du NLP sont très variées, de la traduction automatique au développement des chatbots en passant par l'application dans le marketing et l'analyse des sentiments.

Dans la suite de ce travail, nous allons explorer les principes du NLP sur un exemple de l'analyse des sentiments en utilisant un jeu de données comportant 1,6 million de tweets.

Nous passerons par 2 phases :

1. **Prétraitement (préparation) des données :** dans laquelle différentes opérations seront effectuées sur nos données : nettoyage, normalisation (radicalisation des mots, construction du vocabulaire…etc.), extraction des caractéristiques.
2. **Phase d'apprentissage :** il s'agit de l'étape d'entraînement du modèle et des tests. Deux différentes approches seront exploitées, en utilisant des modèles classiques de Machine Learning et des modèles de Deep Learning.

Tout au long de ce rapport, on passera en revue toutes les tâches effectuées à chaque étape, en les expliquant. On fera une comparaison des classificateurs appliqués. Pour arriver à des conclusions présenter à la fin.



Première Étape : Préparation des données

La première phase par laquelle il faut passer est d'analyser et de préparer nos données.

Nous allons utiliser la bibliothèque *nltk* (Natural Language ToolKit) de python, qui permet de travailler avec des données de langage naturel. On fera avec le processus du nettoyage des tweets : la Radicalition de mots, la suppression des non-mots, suppression des mots vides. [1]

Organisation des données :

- Pour commencer, nous allons importer nos données dans un *DataFrame* de **pandas** :

```
df = pd.read_csv('./training.1600000.processed.noemoticon.csv', encoding='latin', header=None)
df.head()
```

Résultat :

	0	1	2	3	4	5
0	0	1467810369	Mon Apr 06 22:19:45 PDT 2009	NO_QUERY	_TheSpecialOne_	@switchfoot http://twitpic.com/2y1zl - Awww, t...
1	0	1467810672	Mon Apr 06 22:19:49 PDT 2009	NO_QUERY	scotthamilton	is upset that he can't update his Facebook by ...
2	0	1467810917	Mon Apr 06 22:19:53 PDT 2009	NO_QUERY	mattycus	@Kenichan I dived many times for the ball. Man...
3	0	1467811184	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	ElleCTF	my whole body feels itchy and like its on fire
4	0	1467811193	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	Karoli	@nationwideclass no, it's not behaving at all....

- On nomme les colonnes selon la description du dataset :

Le dataset contient 6 colonnes, comme suit : 1. sentiment (target): the polarity of the tweet (0 = negative, 4 = positive) 2. id: The id of the tweet (2087) 3. date: the date of the tweet (Sat May 16 23:58:44 UTC 2009) 4. flag: The query (lyx). If there is no query, then this value is NO_QUERY. 5. user_id: the user that tweeted (robotickilldozr) 6. text: the text of the tweet (Lyx is cool)

```
df.columns = ['sentiment', 'id', 'date', 'flag', 'user_id', 'text']
df.head()
```

Résultat :

	sentiment	id	date	flag	user_id	text
0	0	1467810369	Mon Apr 06 22:19:45 PDT 2009	NO_QUERY	_TheSpecialOne_	@switchfoot http://twitpic.com/2y1zl - Awww, t...
1	0	1467810672	Mon Apr 06 22:19:49 PDT 2009	NO_QUERY	scotthamilton	is upset that he can't update his Facebook by ...
2	0	1467810917	Mon Apr 06 22:19:53 PDT 2009	NO_QUERY	mattycus	@Kenichan I dived many times for the ball. Man...
3	0	1467811184	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	ElleCTF	my whole body feels itchy and like its on fire
4	0	1467811193	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	Karoli	@nationwideclass no, it's not behaving at all....

- Pour la suite on aura besoin que de deux colonnes pour faire notre apprentissage : *sentiment* et *text* ; on supprime les colonnes inutiles.

```
df = df.drop(['id', 'date', 'flag', 'user_id'], axis=1)
df.head()
```

Résultat :

	sentiment	text
0	0	@switchfoot http://twitpic.com/2y1zl - Awww, t...
1	0	is upset that he can't update his Facebook by ...
2	0	@Kenichan I dived many times for the ball. Man...
3	0	my whole body feels itchy and like its on fire
4	0	@nationwideclass no, it's not behaving at all....

- Désormais nous allons décoder les valeurs de la colonne sentiment, en mappant 0 à *Negative* et 1 à *Positive.*

```
lab_to_sentiment = {0: "Negative", 4: "Positive"}
def label_decoder(label):
    return lab_to_sentiment[label]
df.sentiment = df.sentiment.apply(lambda x: label_decoder(x))
df.head()
```

Résultat :


```
lab_to_sentiment = {0: "Negative", 4: "Positive"}
def label_decoder(label):
    return lab_to_sentiment[label]
df.sentiment = df.sentiment.apply(lambda x: label_decoder(x))
df.head()
```


- Comme on peut remarquer, nos données sont polluées par des ponctuations, des hyperliens, des hashtags, …etc. **Afin de pouvoir effectuer notre apprentissage nous allons devoir opérer à un prétraitement qui est le nettoyage des données en utilisant *NLTK*.**

1. **Supprimer les hyperliens, des mentions et autres caractères non alphabétiques**

```
text_cleaning_re = "@\S+|https?:\S+|http?:\S|[^A-Za-z]+"
```

2. **Radicalisation des mots**

Vu que dans un document texte les mots seront utilisés selon différentes formes pour des raisons grammaticales, il est important de faire en sorte de réduire les variations qui peuvent y résulter et ne prendre que la racine des mots. On utilisera **SnowballStemmer** de **NLTK.stem** qui donne la racine d'un mot donné en entrée.

```
nltk.download('stopwords')
from nltk.corpus import stopwords
stop_words = stopwords.words('english')
```

3. **Suppression des mots vides**

Cela revient à supprimer les mots qui sont utilisés sans avoir une signification contextuelle dans une phrase. On utilisera **la librairie Stopwords de NLTK** qui contient les mots vides en anglais.

```
from nltk.stem import SnowballStemmer
stemmer = SnowballStemmer('english')
```

4. **Nettoyage :**

La fonction nettoyage prendra en paramètre une chaîne de caractères, on commencera par appliquer l'expression régulière pour enlever les mentions et les liens et rendre tout le texte en minuscule, puis pour chaque mot de la chaîne de caractères on vérifie s'il n'existe pas dans la liste des mots vides, pour finir par prendre forme de base (sa racine) et l'ajouter dans la liste *tokens*.

```
def nettoyage(text, stem=True):
    text = re.sub(text_cleaning_re, ' ', str(text).lower()).strip()
    tokens = []
    for token in text.split():
        if token not in stop_words:
            if stem:
                tokens.append(stemmer.stem(token))
            else:
                tokens.append(token)
    return " ".join(tokens)
```

Construction du vocabulaire :

Pour faire la construction de notre vocabulaire nous allons utiliser la fonction Tokenizer de la librairie keras_preprocessing.text

```
from keras_preprocessing.text import Tokenizer

tokenizer = Tokenizer()
tokenizer.fit_on_texts(train_set.text)

word_index = tokenizer.word_index
vocab_size = len(tokenizer.word_index) + 1
print("Vocabulary Size: ", vocab_size)
```

On obtient de dictionnaire suivant :

```
for word, index in tokenizer.word_index.items():
    print(index, word)
```

```
1 go
2 get
3 day
4 good
5 work
6 like
7 love
8 quot
9 today
10 time
```

Extraction des caractéristiques :

Vecteur de comptage (Count Vectorizer) : une représentation qui permet de décrire les occurrences des mots dans un texte. À Chaque tweet correspondra un vecteur entier de taille n (n étant la taille du vocabulaire). Pour l'appliquer à notre ensemble d'entrainement nous utiliserons la méthode *CountVectorizer* de *Sklearn.feature_extraction.text*.

```
from sklearn.feature_extraction.text import CountVectorizer
# Vectorisation des données textuelles
vectorizer = CountVectorizer()
x_train_vectorized = vectorizer.fit_transform(train_set['text'])
x_test_vectorized = vectorizer.transform(test_set['text'])
```

Plongement de mots (Word embedding) : les deux méthodes d'extraction de caractéristiques présentaient (Bag-of-Words, CountVectorizer) sont faciles à implémenter mais restent mauvaises en performances pour l'entraînement du modèle et dépendent principalement sur la taille de notre vocabulaire et notre ensemble d'entraînement. Une alternative a été explorée : *plongement sémantique* (*word embedding*), qui est une méthode d'apprentissage de la représentation des mots sous forme vectorielle : elle prend en compte le contexte du mot, la similarité sémantique et syntaxique et elle réduit la dimension.

On utilisera cette technique issue du « Transfer Learning » consistant à télécharger des embeddings pré-entrainés et à les utiliser dans notre modèle, dans notre cas on utilise *GloVe Embedding from Stanford AI*. On va utiliser cela par la suite pour l'entraînement d'un réseau de neurones.

```
GLOVE_EMB = 'glove.6B/glove.6B.300d.txt'
EMBEDDING_DIM = 300
LR = 1e-3
BATCH_SIZE = 1024
EPOCHS = 10
MODEL_PATH = ''
```

```
embedding_index = {}

f = open(GLOVE_EMB, encoding='utf-8')
for line in f:
    values = line.split()
    word = value = values[0]
    coefs = np.asarray(values[1:], dtype="float32")
    embedding_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embedding_index))
```

```
Found 400000 word vectors.
```

```
embedding_matrix = np.zeros((vocab_size, EMBEDDING_DIM))
for word, i in word_index.items():
    embedding_vector = embedding_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector

print(embedding_matrix)
```

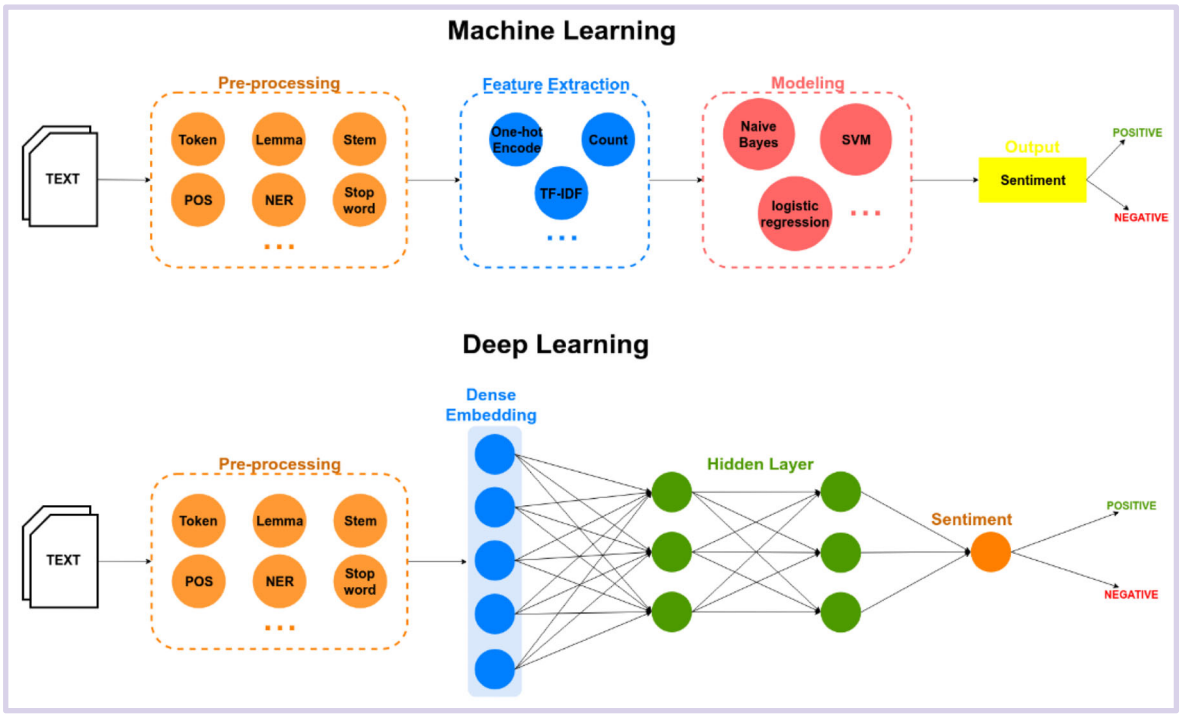
```
[[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.0073678   0.062532   -0.097432   ... -0.32431999  0.19611
   0.29361999]
 [-0.14124    -0.11836    -0.30781999 ... -0.19882999 -0.061105
   0.11568   ]
 ...
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]]
```

```
# embedding_layer = tf.keras.layers.Embedding(vocab_size) dimension and length must be specified
import tensorflow as tf
embedding_layer = tf.keras.layers.Embedding(vocab_size,
                                            EMBEDDING_DIM,
                                            weights=[embedding_matrix],
                                            input_length=MAX_SEQUENCE_LENGTH,
                                            trainable=False)
```

Deuxième Étape : Classification

En analyse des sentiments, il existe principalement deux approches pour classer les données textuelles : les méthodes de Machine Learning et celles de Deep Learning.

Concernant le Machine Learning : ses méthodes reposent sur des algorithmes traditionnels comme les SVM (Support Vector Machines), les arbres de décision ou autres. Ces techniques nécessitent généralement un prétraitement du texte (la tokenisation, le nettoyage, la vectorisation ou embedding). Une fois le texte transformé en une représentation numérique, ces derniers peuvent être appliqués pour classifier les sentiments. **Par contre, les méthodes de deep learning,** en particulier les réseaux de neurones profonds (Deep Neural Networks), ont révolutionné l'analyse de sentiments. Des architectures comme les réseaux neuronaux récurrents (RNN), les LSTM (Long Short-Term Memory) et plus, permettent de capturer des relations contextuelles complexes dans les données textuelles.



Pour l'analyse des sentiments, on a choisi d'utiliser différents classificateurs issus des approches de Machine Learning et du Deep Learning, en tenant compte de leurs avantages spécifiques pour cette tâche.

Machine Learning :

SVM (Support Vector Machines) :

Le Support Vector Machine (SVM) est un algorithme de classification supervisé qui cherche à trouver l'hyperplan optimal séparant les différentes classes dans un espace de caractéristiques. En maximisant la marge entre les points de données de chaque classe, SVM assure une meilleure généralisation sur des données non vues.

Raisons d'Utilisation de SVM

- ***Données linéairement séparables*** : SVM excelle lorsque les données peuvent être séparées de manière linéaire.
- ***Efficacité en Haute Dimension*** : Capacité à gérer des espaces de caractéristiques de haute dimension, particulièrement utile avec des représentations de texte comme TF-IDF.
- ***Généralisation*** : Bonne capacité à généraliser sur des données non vues, essentiel pour l'analyse de sentiments.
- ***Régularisation*** : Contrôle de l'overfitting via des paramètres de régularisation.
- *Interprétabilité* : Les frontières de décision sont relativement simples à interpréter par rapport à des modèles plus complexes comme les réseaux neuronaux.

Processus et Fonctionnement du SVM :

1. **Entraînement du Modèle :**
   - Nécessité d'un ensemble de données d'entraînement étiqueté.
   - Transformation des données en vecteurs de caractéristiques, souvent à l'aide de techniques comme TF-IDF ou les embeddings de mots.

2. **Séparation des Classes :**
   - Dessine un hyperplan pour séparer les deux classes.
   - Trouver l'hyperplan qui maximise la marge entre les deux catégories.
   - Les points situés sur la marge influencent la position de l'hyperplan.

3. **Prédiction :**
   - Ajustement du modèle en utilisant les données d'entraînement.
   - Utilisation du modèle entraîné pour prédire les classes des nouvelles données.

4. **Noyau :**

- Pour notre cas on a opté pour le choix de Kernel Non Linéaire ‘rbf’ et ceci après avoir fait plusieurs tests de performances.

Paramétrage de SVM :

Pour optimiser les performances de notre SVM, nous avons testé plusieurs types de noyaux :

- Noyau Linéaire : Le plus efficace pour notre cas, offrant une séparation claire avec un calcul relativement simple.

- Noyaux Non Linéaires (SVC) : Inclut les noyaux polynomiaux et RBF, nécessitant une exploration plus approfondie pour trouver les paramètres optimaux.

On a aussi fait varier le paramètre de pénalité C ça n'a pas trop affecté nos résultats

Paramètre de Pénalité (C) : Contrôle le compromis entre maximisation de la marge et minimisation de l'erreur de classification. Utilisation de la validation croisée pour ajuster ce paramètre.

Mesure des Performances : Précision et Matrice de Confusion

Pour évaluer les performances de notre modèle SVM, nous utilisons deux principales métriques :

- *Précision* : Mesure la proportion de prédictions correctes sur l'ensemble de tests. C'est un indicateur global de la performance du modèle.
- *Matrice de Confusion* : Fournit une vue détaillée des performances en montrant le nombre de vrais positifs, vrais négatifs, faux positifs et faux négatifs. Cela permet d'analyser plus finement les erreurs du modèle et de comprendre où il peut être amélioré.

Capture code :

Nous avons utilisé l'algorithme SVM de sklearn pour son optimisation et ces bonnes performances.

```
# Libraries
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
import seaborn as sns

# SVM Model , C=1.0
svm_model = SVC(kernel='rbf')

# Train
svm_model.fit(x_train_vectorized, y_train.ravel())

# Predict
y_pred = svm_model.predict(x_test_vectorized)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print("SVM Accuracy:", accuracy)

# Calculate the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
plt.show()
```

Naive Bayes :

Le classificateur **Naive Bayes** est un algorithme de classification supervisé basé sur le théorème de Bayes. Il est naïf car il suppose que les caractéristiques sont indépendantes les unes des autres, ce qui simplifie grandement les calculs. En particulier, le modèle Multinomial Naive Bayes est utilisé pour la classification de texte en comptant la fréquence des mots dans les documents.

Raisons d'Utilisation de Naive Bayes

- ***Simplicité et rapidité* :** Naive Bayes est simple à implémenter et très rapide à entraîner.
- ***Efficacité pour les Données Textuelles* :** Il fonctionne bien avec des données textuelles et des représentations de texte en haute dimension.
- ***Faible Variance* :** Tendance à ne pas sur-ajuster les données, ce qui le rend robuste pour des ensembles de données de taille limitée.
- ***Robustesse aux Caractéristiques Non-pertinentes* :** Capacité à gérer un grand nombre de caractéristiques, même si certaines ne sont pas pertinentes.

Processus et Fonctionnement de Naive Bayes :

1. **Histogrammes des Mots**
   - Création d'un histogramme des mots apparaissant dans la classe positive.
   - Création d'un histogramme des mots apparaissant dans la classe négative.

2. **Calcul des Probabilités :**
   - Calcul des probabilités d'apparition des mots dans chaque classe en fonction du nombre total de mots dans chaque catégorie.
   - Utilisation de la vectorisation pour obtenir les comptages nécessaires.

3. **Prédiction :**
   - Estimation de la probabilité initiale des classes positives et négatives basée sur leurs fréquences respectives dans les données d'entraînement.
   - Multiplication de la probabilité a priori par la probabilité d'occurrence des mots dans chaque classe.

4. **Classificateur :**
- Pour chaque Tweet, calcul des probabilités pour les classes positives et négatives.
- La classe avec la probabilité la plus élevée est choisie comme prédiction.

On peut distinguer deux cas :

Le cas Simple :

1. Calculer la probabilité a priori pour la classe positive (P(positive)).
2. Multiplier cette probabilité par la probabilité d'occurrence des mots pour la classe positive.
3. Répéter le même processus pour la classe négative.
4. Comparer les probabilités résultantes et choisir la classe avec la plus grande probabilité.

Le cas Complexe :

- Pour éviter les probabilités nulles (zéro) pour certains mots, on ajoute un "smooth" (par exemple, Laplace Smoothing) en ajoutant une unité à chaque comptage de mot.

Paramétrage de Naive Bayes :

Pour optimiser les performances de notre modèle Naive Bayes, nous avons utilisé plusieurs techniques de réglage :

1. Lissage (Smoothing) :

- Ajout d'une petite constante à tous les comptages de mots pour éviter les probabilités nulles.

2. Représentation des Caractéristiques :

- Utilisation de techniques de vectorisation comme TF-IDF pour représenter les documents.

3. Choix du Type de Naive Bayes :

- Multinomial Naive Bayes : Idéal pour les tâches de classification de texte où la fréquence des mots est importante.

- Bernoulli Naive Bayes : Peut être utilisé pour des données binaires (présence ou absence de mots).

Mesure des Performances : Précision et Matrice de Confusion

```python
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.pipeline import make_pipeline

# Vectorisation des données textuelles
vectorizer = CountVectorizer()
x_train_vectorized = vectorizer.fit_transform(train_set['text'])
x_test_vectorized = vectorizer.transform(test_set['text'])

# Modèle Naive Bayes
nb_model = MultinomialNB()

# Création du pipeline
pipeline = make_pipeline(vectorizer, nb_model)

# Entraîner le modèle
pipeline.fit(train_set['text'], y_train.ravel())

# Prédire
y_pred_nb = pipeline.predict(test_set['text'])

# Précision
accuracy_nb = accuracy_score(y_test, y_pred_nb)
print("Naive Bayes Accuracy:", accuracy_nb)

# Matrice de confusion
conf_matrix_nb = confusion_matrix(y_test, y_pred_nb)
print("Matrice de Confusion:")
print(conf_matrix_nb)
```

Deep Learning :

LSTM :

Long Short-Term Memory (LSTM) est un type de réseau de neurones récurrents (RNN) utilisé dans le domaine de l'apprentissage profond. Les LSTM ont des connexions de rétroaction, leur permettant d'exploiter les dépendances temporelles au sein de séquences de données. LSTM est conçu pour gérer les problèmes des gradients qui disparaissent ou explosent, souvent rencontrés lors de l'entraînement des RNN traditionnels sur des séquences de données. Cela les rend bien adaptés aux tâches impliquant des données séquentielles, telles que le traitement du langage naturel (NLP), la reconnaissance vocale et la prévision de séries temporelles.

Les réseaux LSTM introduisent des cellules de mémoire, capables de retenir l'information sur de longues séquences. Chaque cellule de mémoire comporte trois composants principaux : une porte d'entrée, une porte d'oubli et une porte de sortie. Ces portes aident à réguler le flux d'informations entrant et sortant de la cellule de mémoire.

Raisons d'Utilisation de LSTM :

1. **Gestion des Dépendances à Long Terme :**
   - Les LSTM sont particulièrement efficaces pour apprendre des dépendances à long terme dans les données séquentielles. Dans l'analyse de sentiment, comprendre le contexte d'une phrase sur une longue séquence de mots est crucial.

2. **Robustesse aux Séquences Longues :**
   - Les réseaux LSTM sont conçus pour résoudre les problèmes de gradients qui disparaissent, ce qui permet au modèle de se souvenir d'informations pertinentes sur de longues périodes.

3. **Capacité à Gérer des Données Séquentielles Complexes :**
   - L'analyse de sentiment nécessite de comprendre non seulement les mots individuels, mais aussi la manière dont ils sont utilisés dans des contextes complexes. Les LSTM, grâce à leurs cellules de mémoire et leurs portes de régulation, peuvent capturer et utiliser ces informations contextuelles complexes de manière plus efficace qu'un RNN classique.

4. **Applications Variées :**
   - En plus de l'analyse de sentiment, les LSTM ont prouvé leur efficacité dans diverses applications impliquant des données séquentielles, comme la traduction automatique, la reconnaissance vocale et la prévision de séries temporelles.

5. **Rapidité et Consistance :**
   - Les LSTM peuvent fournir des résultats plus robustes en moins de temps que les méthodes de machine learning traditionnelles telles que les arbres de décision (Decision Trees), les k-plus proches voisins (k-NN), et les machines à vecteurs de support (SVM). Leur capacité à capturer des dépendances complexes et à traiter des données séquentielles permet une meilleure performance globale et une consistance accrue dans les prédictions.

Processus et Fonctionnement de LSTM :

LSTM (Long Short-Term Memory) est une architecture de réseau de neurones récurrents (RNN) qui permet de résoudre le problème des gradients qui disparaissent ou explosent dans les RNN traditionnels. Les LSTM utilisent des cellules de mémoire et des portes pour conserver l'information sur de longues séquences de données, ce qui les rend particulièrement adaptés pour des tâches comme l'analyse de sentiment.

Préparation de couche d'entrée et sortie :

1- On définit la couche d'entrée pour le modèle, prenant des séquences de longueur maximale `MAX_SEQUENCE_LENGTH` et de type `int32`.

2- Appliquer une couche d'embedding pré-entraînée aux séquences d'entrée, transformant les entiers représentant des mots en vecteurs de caractéristiques.

3- Applique un dropout spatial aux séquences d'embedding pour régulariser le modèle et prévenir le surapprentissage.

4- Appliquer une couche conventionnelle avec 64 filtres et une taille de kernel de 5 pour capturer des caractéristiques locales des séquences.

5- Appliquer une couche LSTM bidirectionnelle, permettant de traiter les séquences dans les deux directions (avant et arrière) pour mieux capturer les dépendances contextuelles. Utilise des taux de drop out pour régulariser le modèle.

6- Transforme la sortie de la couche LSTM en une dimension supérieure pour extraire plus de caractéristiques, avec 512 neurones et une activation ReLU.

7- Ajouter un dropout de 50% pour régulariser le modèle et prévenir le surapprentissage. Ainsi qu’une autre transformation dense avec 512 neurones et une activation ReLU pour une extraction de caractéristiques plus approfondie.

8- Définir la couche de sortie avec une activation sigmoid pour prédire la probabilité d'appartenance à une classe binaire (sentiment positif ou négatif).

```
# LSTM process

# 1. defining the input layer
sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')

# 2. predefined hidden layer
embedding_sequences = embedding_layer(sequence_input)

# 3. this layer applies a dropout to the input
# so it regularizes the model , to prevent overfitting
x = SpatialDropout1D(0.2)(embedding_sequences)

# 4. 64 filters and a kernel size of 5 + it helps in capturing local features
x = Conv1D(64, 5, activation='relu')(x)

# 5. A Bidirectional LSTM (Long Short-Term Memory) layer processes
# the sequence in both forward and backward directions
# dropout 20% and 64 filters
x = Bidirectional(LSTM(64, dropout=0.2, recurrent_dropout=0.2))(x)

# 6. this will transform the input layer to a higher dimension so
# we can find more features that help us to have the predictions
#  512 neurons and ReLU activation
x = Dense(512, activation='relu')(x)

# 7. prevent overfitting
x = Dropout(0.5)(x)

# 8. a layer for further transformation and feature extraction
# having 512 neurons and relu function
x = Dense(512, activation='relu')(x)

# 9. define the output layer
outputs = Dense(1, activation='sigmoid')(x)
```

Compilation, Entraînement, Prédiction :

1- Compiler le modèle en spécifiant l'optimiseur Adam avec un taux d'apprentissage défini (`LR`), la fonction de perte (`binary_crossentropy`) et les métriques à surveiller (`accuracy`). Ajoute un callback `ReduceLROnPlateau` pour réduire le taux d'apprentissage si la perte de validation ne s'améliore pas.

2- Entraîner le modèle sur les données d'entraînement (`x_train`, `y_train`) pendant un nombre défini d'époques (`EPOCHS`) avec une taille de lot (`BATCH_SIZE`). Utilise les données de validation (`x_test`, `y_test`) et applique le callback `ReduceLROnPlateau` pour ajuster dynamiquement le taux d'apprentissage.

3- Prévoir les scores de sentiment sur les données de test (`x_test`) en utilisant le modèle entraîné. Utilise une fonction `decode_sentiment` pour transformer les scores de sortie en prédictions de classes (positif ou négatif).

```python
#10. call the model with both input and output layer
model = tf.keras.Model(sequence_input, outputs)

# 11. preparing the model for training by specifying the optimizer
model.compile(optimizer=Adam(learning_rate=LR), loss='binary_crossentropy',
              metrics=['accuracy'])
ReduceLROnPlateau = ReduceLROnPlateau(factor=0.1,
                                      min_lr = 0.01,
                                      monitor = 'val_loss',
                                      verbose = 1)

# 12. we can run this on cpu , but it is preferable to do so on the GPU
print("Training on GPU...") if tf.test.is_gpu_available() else print("Training on CPU...")

# 13. fit the model
history = model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS,
                    validation_data=(x_test, y_test), callbacks=[ReduceLROnPlateau])
```

```python
# 15. predict the scores
scores = model.predict(x_test, verbose=1, batch_size=10000)
y_pred_1d = [decode_sentiment(score) for score in scores]
```

Les LSTM utilisent des fonctions d'activation sigmoïde et tanh pour moduler les états de mémoire à court terme et à long terme.

1. Porte d'oubli (Forget Gate) :
Utilise une activation sigmoïde pour déterminer quelles informations de l'état de mémoire doivent être oubliées.

2. Porte d'entrée (Input Gate) :
Utilise une activation sigmoïde pour sélectionner quelles nouvelles informations doivent être ajoutées à l'état de mémoire.

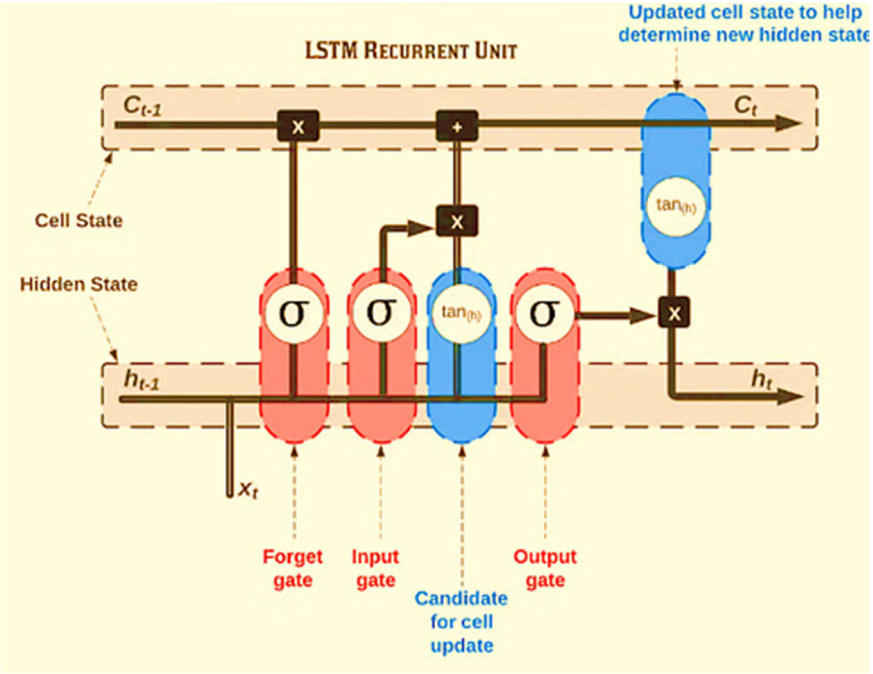
3. Création de nouvelles valeurs de mémoire (Cell State Update) :
Utilise une activation tanh pour créer de nouvelles valeurs candidates à ajouter à l'état de mémoire.

4. Mise à jour de l'état de mémoire (Cell State) :
Combine les valeurs de la porte d'oubli et de la porte d'entrée pour mettre à jour l'état de mémoire.

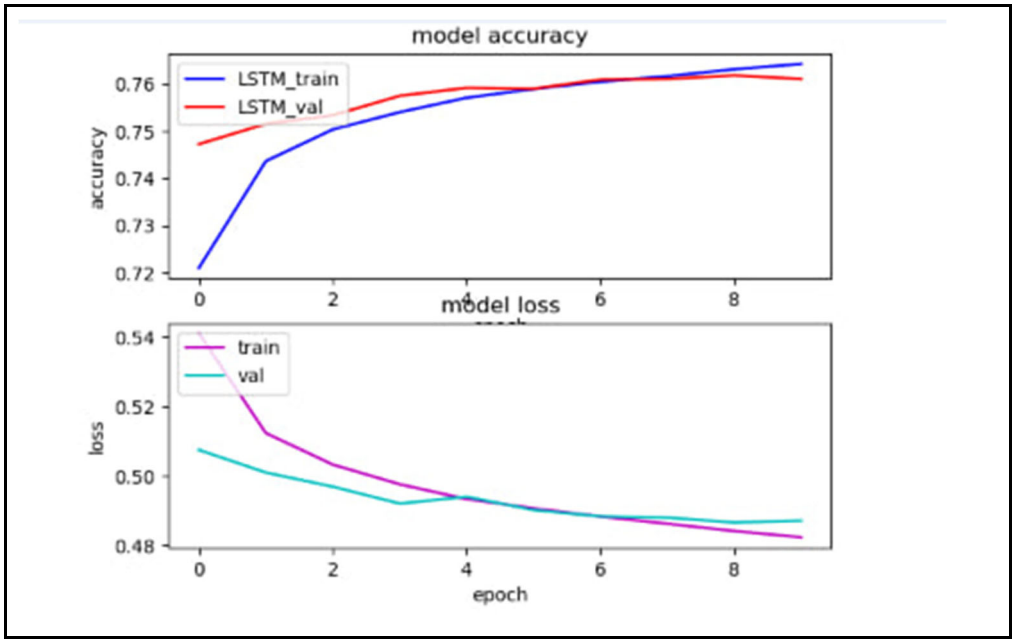
5. Porte de sortie (Output Gate) :
Utilise une activation sigmoïde pour déterminer quelles informations de l'état de mémoire doivent être sorties.

6. Calcul de l'état caché (Hidden State) :
Utilise une activation tanh sur l'état de mémoire mis à jour et la multiplie par la sortie de la porte de sortie pour produire l'état caché.

- Voici une image qui montre les calculs que LSTM passe par durant sans fonctionnement :



Mesure des Performances : Précision et Matrice de Confusion



Analyse des Résultats et Comparaison de Performances :

Temps d'exécution :

Pour L'ensemble de tous les donnes voici le temps d'exécution :

Méthode	Temps d'exécution
LSTM avec CPU	**30 - 45 min**
LSTM avec GPU	**5 min**
Naive Bayes	**5 min**
SVM	**plus que 4 h**

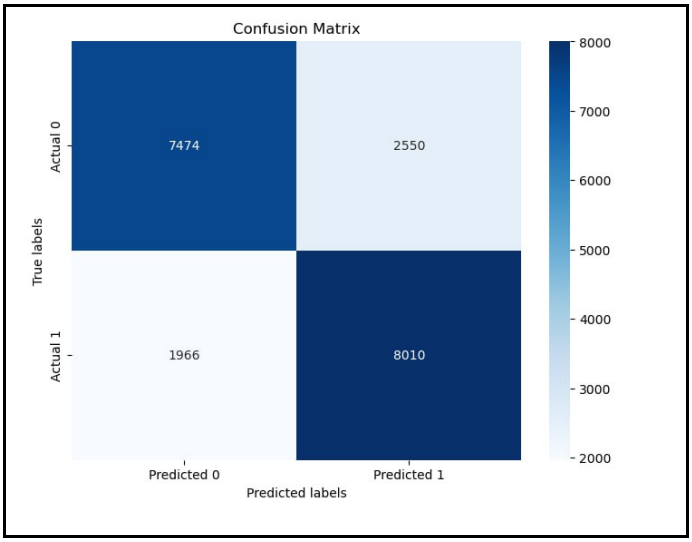
Donc pour SVM on a fait les tests sur 100000 lignes de données : tel que le nombre de tweets positifs = nombre de tweets négatifs.

Précision :

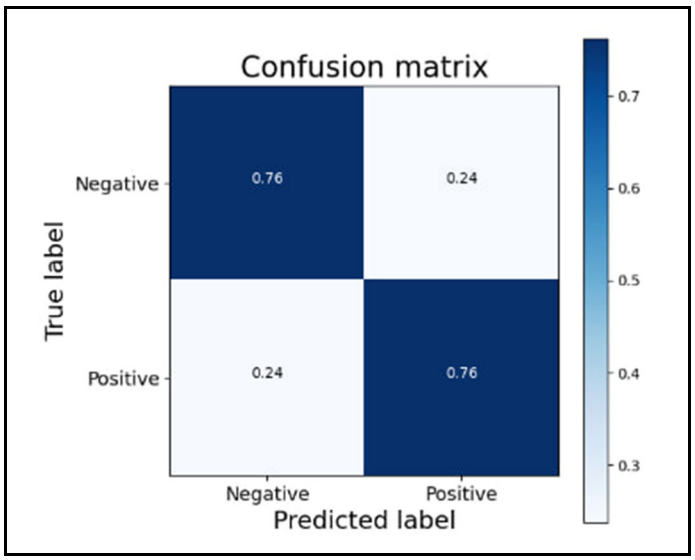
SVM :	Naive Bayes :	LSTM :
SVM Accuracy: 0.7742	Naive Bayes Accuracy: 0.76304375	LSTM Accuracy: 0.76

Matrice de confusion :

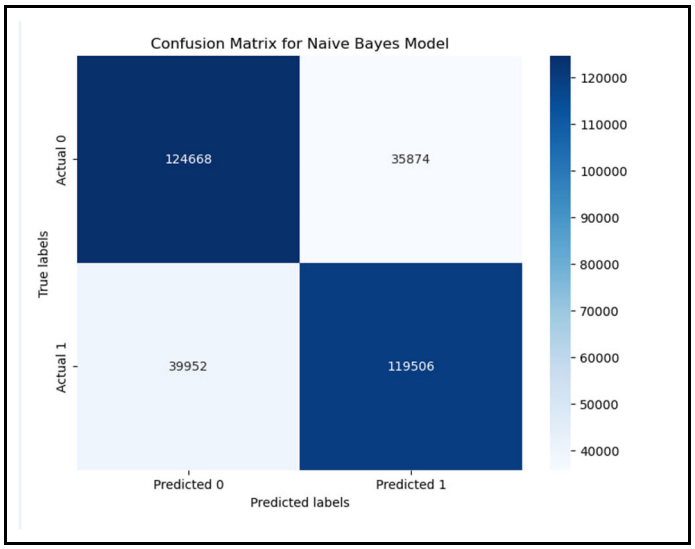
SVM :



LSTM :



Naive Bayes :



Après avoir mesurer les performances de chaques méthode on peut faire une comparaison et analyse générale :

Analyse Comparée des Méthodes :

1. **Précision :**
   - SVM a obtenu la meilleure précision avec 0.77.
   - Naive Bayes suit de près avec une précision de 0.763.
   - LSTM a une précision légèrement inférieure à 0.76.

2. **Temps de Traitement :**
   - Naive Bayes est le plus rapide, traitant 1 600 000 lignes en 5 minutes.
   - LSTM est rapide avec le GPU (5 minutes) mais beaucoup plus lent avec le CPU (35 minutes).
   - SVM est le plus lent, nécessitant 25 minutes pour seulement 100 000 lignes.

Analyse des Méthodes :

1. Naive Bayes (NB) :

Avantages :

- Rapidité de traitement, même pour des jeux de données volumineux.
- Simple à mettre en œuvre.
- Fonctionne bien avec des données textuelles et les modèles de probabilité.

Inconvénients :

- Les hypothèses d'indépendance entre les caractéristiques peuvent ne pas être toujours valides, ce qui peut limiter la précision.
- Peut avoir des performances inférieures avec des données complexes ou de grande dimensionnalité.

2. Support Vector Machine (SVM):

Avantages :

- Très bonne précision, même avec des jeux de données plus petits.
- Efficace pour les problèmes de classification avec des marges larges et claires entre les classes.

Inconvénients :

- Temps de traitement élevé, surtout pour des jeux de données volumineux.
- Complexité de la mise en œuvre et de l'optimisation des hyper paramètres.
- Sensible à l'échelle des données (nécessite souvent une normalisation).

3. Long Short-Term Memory (LSTM):

Avantages :

- Capacité à capturer les dépendances à long terme dans les données séquentielles.
- Efficace pour les données textuelles et les séquences temporelles.
- Meilleure performance avec un GPU, permettant de traiter de grands volumes de données rapidement.

Inconvénients :

- Nécessite beaucoup de ressources de calcul, surtout sans GPU.
- Temps de formation long avec CPU.
- Plus complexe à implémenter et à optimiser par rapport à NB et SVM.

Conclusion Générale :

À travers ce projet nous sommes passés par toutes les étapes qui permettent de réaliser un modèle de détection de sentiment.

En commençant par la phase de prétraitement puis la phase de choix et entraiment des différents modèles, ce qui nous a permis d'observer la différente majeure elles. En soulignant l'importance des nouvelles méthodes d'apprentissage profond, qui représente une révolution, facilitant la manipulation et l'apprentissage pour un nombre important de données qui nécessitent beaucoup de ressources.

Bibliographie

[1] E. K. a. E. L. Steven Bird, «Natural Language Processing with Python,» 15 05 2024. [En ligne]. Available: https://www.nltk.org/book_1ed/.