Projet TAL

AYED Hatem, GUERIN Titouan

1er mars 2024

1 Introduction

Ce projet avait pour objectif de développer des modèles de classification pour reconnaître les sentiments exprimés dans les critiques de films, ainsi que pour identifier le locuteur lors d'une discussion entre deux individus (Chirac et Mitterrand).

Différents corpus nous étaient fournis pour travailler, sur lesquels nous avons appliqué des techniques de prétraitement et nous avons entraîné plusieurs modèles de classification. Nous avons exploré différentes techniques de traitement du langage naturel, telles que la vectorisation de texte et la régression logistique, pour classifier les critiques de films en sentiments positifs ou négatifs notamment.

Nos résultats ont démontré l'efficacité ainsi que les problèmes de certains modèles de classification. Ce projet met en évidence l'importance croissante des techniques d'analyse de texte et de traitement du langage naturel dans la compréhension des sentiments et des interactions humaines à partir de données textuelles.

2 Objectif

Dans le cadre de ce projet, plusieurs objectifs majeurs ont été définis afin d'aborder de manière systématique et approfondie la tâche de reconnaissance de sentiments dans les critiques de films, ainsi que l'identification du locuteur dans les dialogues entre deux individus. Les principaux objectifs sont les suivants :

2.1 Transformation Paramétrique du Texte (Prétraitement)

Le premier objectif du projet était de mettre en place une pipeline de prétraitement du texte afin de le préparer pour la classification. Cela inclut des étapes telles que la suppression des caractères spéciaux, la mise en minuscule, la lemmatisation ou encore la suppression des mots vides. L'objectif était d'obtenir une représentation normalisée et homogène du texte en entrée pour les modèles de machine learning.

2.2 Extraction du Vocabulaire Bag-of-Words (BoW)

Un autre objectif était d'extraire le vocabulaire Bag-of-Words (BoW) à partir du corpus de données. Le BoW est une représentation vectorielle du texte où chaque mot unique dans le corpus est représenté par un index dans le vecteur. Cette extraction permet de créer des vecteurs de caractéristiques pour chaque document, qui seront utilisés comme entrée pour les modèles de machine learning.

2.3 Modèles de Machine Learning

Le cœur du projet reposait sur l'implémentation et l'évaluation de différents modèles de machine learning pour la classification des sentiments et l'identification du locuteur. Cela comprenait l'utilisation de divers algorithmes tels que la régression logistique, les machines à vecteurs de support (SVM) et Naive Bayes.

2.4 Métriques d'Évaluation

Un aspect crucial était l'évaluation des performances des modèles. Pour cela, différentes métriques d'évaluation ont été utilisées, telles que l'accuracy, la précision, le rappel (recall), le score F1, etc. Ces métriques permettent de mesurer la qualité des prédictions des modèles. Pour l'analyse des locuteurs, le f1 était la métrique la plus importante.

2.5 Variantes sur les Stratégies d'Entraînement

Dans le but d'améliorer les performances des modèles, différentes stratégies d'entraînement ont été explorées. Cela inclut l'utilisation de techniques d'échantillonnage comme l'over-sampling et l'under-sampling pour équilibrer les classes, ainsi que d'autres techniques de régularisation et d'optimisation des hyperparamètres.

2.6 Post-Processing

Une étape cruciale du processus de classification des sentiments et de l'identification du locuteur est le post-traitement des prédictions des modèles. Le post-traitement vise à améliorer la qualité des prédictions en appliquant des techniques supplémentaires après que les modèles ont produit leurs sorties.

La technique que nous avons utilise est le lissage gaussien. Le lissage gaussien est une méthode qui permet de lisser les prédictions des modèles en appliquant un filtre gaussien aux probabilités de classe prédites. Le lissage gaussien est utilisé pour lisser les distributions de probabilité des classes positives et négatives afin d'obtenir des prédictions plus fiables et robustes.

3 Analyse Sentimentale

Dans cette section, nous explorerons en détail l'approche adoptée pour réaliser cette tâche d'analyse de sentiments sur les critiques de films. Nous examinerons les différentes étapes du processus, du prétraitement des données à l'évaluation des performances des modèles de classification. Nous mettrons en évidence les techniques utilisées pour nettoyer et préparer les données, ainsi que les méthodes de représentation du texte et de modélisation utilisées pour extraire les sentiments des critiques.

3.1 explication du code

Le script commence par charger les données de critiques de films à partir du répertoire spécifié. La fonction load_movies parcourt chaque sous-répertoire pour extraire les critiques de films et attribuer des étiquettes de classe correspondantes.

```
npath = "./datasets/movies/movies1000/"
alltxts_mov, alllabs_mov = load_movies(npath)
```

Une fois les données chargées, nous utilisons la vectorisation TF-IDF pour convertir les critiques de films en vecteurs de caractéristiques numériques. La fonction TfidfVectorizer de scikit-learn est utilisée à cet effet. TfidfVectorizer permet de convertir une collection de documents textuels en une représentation numérique à l'aide de la technique de vectorisation TF-IDF (Term Frequency-Inverse Document Frequency)

```
tfidf = TfidfVectorizer(use_idf=True, norm='l2', smooth_idf=True, lowercase=False)
X = tfidf.fit_transform(alltxts_mov)
```

3.2 Entraînement et Évaluation des Modèles

Nous divisons ensuite les données en ensembles d'entraînement et de test à l'aide de la fonction train_test_split de scikit-learn. Ensuite, nous utilisons une régression logistique avec validation croisée pour entraîner un modèle de classification. Une GridSearchCV est effectuée pour sélectionner les meilleurs hyperparamètres pour le modèle.

```
X_train, X_test, y_train, y_test = train_test_split(X, alllabs_mov,
test_size=0.20,
random_state=12)
param_grid = {'Cs': [1, 10, 100], 'solver': ['lbfgs', 'liblinear']}
nlr = LogisticRegressionCV(cv=5, scoring='accuracy', random_state=0,
n_jobs=-1, verbose=3, max_iter=1000)
grid_search = GridSearchCV(nlr, param_grid, cv=5, scoring='accuracy',
verbose=3, n_jobs=-1)
grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 16 concurrent workers.
[Parallel(n_jobs=-1)]: Done 2 out of 5 | elapsed: 2.5s remaining: 3.8s
[Parallel(n_jobs=-1)]: Done 5 out of 5 | elapsed: 2.6s finished
Done!
```

Cet output est généré lors de l'exécution de GridSearchCV. La ligne "Fitting 5 folds for each of 6 candidates, totalling 30 fits" indique que la recherche par grille est effectuée en utilisant une validation croisée à 5 plis avec 6 combinaisons de paramètres différentes à tester, pour un total de 30 modèles entraînés. Le processus est parallélisé avec 16 workers (CPU) pour accélérer l'exécution. Les lignes suivantes montrent la progression de l'exécution de la recherche par grille, indiquant le nombre de plis de validation croisée complétés et le temps écoulé. Enfin, la ligne "Done!" indique que la recherche par grille est terminée et que les meilleurs hyperparamètres ont été trouvés, rendant le modèle prêt à être utilisé pour la prédiction ou l'évaluation.

Une fois le modèle entraîné, nous évaluons ses performances sur l'ensemble de test à l'aide de différentes métriques telles que l'exactitude et le score F1. Voici un extrait du code pour cette étape :

```
y_pred = grid_search.best_estimator_.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average=None)
```

Output :

```
Best Score: 0.844375
Best Estimator: LogisticRegressionCV(cv=5, max_iter=1000, n_jobs=-1, random_state=0,
                     scoring='accuracy', verbose=3)
Accuracy: 0.8375
Confusion Matrix:
[[170  37]
 [ 28 165]]
F1 Score: [0.83950617 0.83544304]
```

- **Best Score** : Le meilleur score obtenu lors du GridSearch pour trouver les hyperparamètres optimaux. Dans cet exemple, le meilleur score est de 0.844375, ce qui représente la précision du modèle.

- **Best Estimator** : Il s'agit d'une instance de LogisticRegressionCV avec les hyperparamètres optimaux trouvés.

- **Accuracy** : Ici l'accuracy est de 0.8375, ce qui signifie que le modèle prédit correctement 83.75% des échantillons de l'ensemble de test.

- **Confusion Matrix** : La matrice de confusion qui montre le nombre de vrais positifs, de faux positifs, de vrais négatifs et de faux négatifs. Ici, la matrice de confusion est affichée avec 170 vrais négatifs, 37 faux positifs, 28 faux négatifs et 165 vrais positifs.

- **F1 Score** : Le score F1, qui est une mesure de la précision et du rappel du modèle. Il est calculé comme la moyenne harmonique de la précision et du rappel. Dans notre cas, le score F1 pour la classe 0 est de 0.83950617 et pour la classe 1 est de 0.83544304.

3.3 Conclusion

En conclusion, les résultats de l'évaluation du modèle de régression logistique pour la classification des sentiments des critiques de films sont globalement satisfaisants. Avec un meilleur score de 0.844375 obtenu lors de la recherche par grille pour trouver les hyperparamètres optimaux, le modèle a démontré une capacité à classifier correctement la plupart des échantillons de l'ensemble de test. De plus, l'accuracy du modèle sur l'ensemble de test s'élève à 0.8375, ce qui signifie qu'il prédit correctement environ 83.75% des échantillons de l'ensemble de test. Les scores F1 élevés pour les deux classes (0 et 1) indiquent un bon équilibre entre précision et rappel pour chaque classe. En somme, ces résultats suggèrent que le modèle de régression logistique est capable de fournir des prédictions précises et fiables pour la classification des sentiments des critiques de films.

4 Analyse de locuteurs

Dans cette section, nous nous concentrons sur l'analyse des locuteurs dans une discussion entre deux personnalités politiques : Chirac et Mitterrand. L'objectif de cette analyse est d'attribuer chaque intervention dans la conversation à son locuteur correspondant. Nous explorons différentes approches de modélisation de machine learning pour résoudre cette tâches.

subsectionDataset Le dataset est constitué d'extraits de discours, d'interviews et de débats impliquant les deux locuteurs. Chaque intervention dans la conversation est représentée par un identifiant unique, suivi d'un numéro de séquence, d'un code indiquant le locuteur (C pour Chirac, M pour Mitterrand), et enfin du texte de l'intervention. Echantillon du dataset :

```
<100:1:C> Quand je dis chers amis, il ne s'agit pas là d'une formule diplomatique,
mais de l'expression de ce que je ressens.
<100:2:C> D'abord merci de cet exceptionnel accueil que les Congolais,
les Brazavillois, nous ont réservé cet après-midi.
...
<100:23:M> Monsieur le maire, mesdames et messieurs, l'Université technologique a donné
le signal, un grand signal.
```

4.1 Transformation du Texte à la Main

Pour préparer les données textuelles, nous avons appliqué plusieurs étapes de transformation manuelle pour tester. Nous avons normalisé le texte en le convertissant en minuscules et en supprimant la ponctuation et les caractères

non ASCII. Ensuite, nous avons utilisé la tokenization pour diviser le texte en mots et appliqué la racinisation pour réduire les mots à leur forme de base. Ces étapes de transformation manuelle visent à simplifier et normaliser le texte pour faciliter la classification. Un exemple du texte transformé est présenté ci-dessous.

```
class: 1 , texte: quand je dis cher amis il ne s agit pas
    la d une formul diplomat mais de l express de ce que
    je ressen
class: -1 , texte: je ne sais ni pourquoi ni comment on s
    est oppos il y a quelqu douz anne douz ou treiz an a
    la creation de l universit technolog
```

4.2 Utilisation de TF-IDF et de la sur-échantillonnage SMOTE

```
tfidf = TfidfVectorizer(use_idf=True, norm='l2',
    smooth_idf=True, lowercase=False)
X = tfidf.fit_transform(alltxts)

X_train, X_test, y_train, y_test = train_test_split(X,
    alllabs, test_size=0.20, random_state=12)
```

Nous utilisons ici le vecteur de caractéristiques TF-IDF. Le paramètre 'use_idf=True' indique que nous utilisons la fréquence inverse du document pour pondérer les termes. Le paramètre 'norm='l2'' spécifie la normalisation des vecteurs de caractéristiques en utilisant la norme L2. Le paramètre 'smooth_idf=True' ajoute une unité à la fréquence de document inverse pour éviter les divisions par zéro.

Nous divisons ensuite nos données en ensembles d'entraînement et de test à l'aide de la fonction 'train_test_split'.

```
oversampler = SMOTE(sampling_strategy='minority',
    random_state=0, k_neighbors=5)
X_train_resampled, y_train_resampled = oversampler.
    fit_resample(X_train, y_train)
```

Nous utilisons également la méthode de sur-échantillonnage SMOTE (Synthetic Minority Over-sampling Technique) pour équilibrer les classes dans notre ensemble d'entraînement. La stratégie de sur-échantillonnage est définie sur 'minority', ce qui signifie que la classe minoritaire sera sur-échantillonnée pour équilibrer les classes.

4.3 Analyse BoW

Le code complet est disponible ici 6 ou dans le fichier 1_1-BoW-Project.ipynb.

Dans cette section, nous procédons à l'extraction du vocabulaire en utilisant l'approche Bag of Words (BoW). Nous commençons par une exploration préliminaire des jeux de données afin de comprendre la composition du vocabulaire

initial. Tout d'abord, nous examinons la taille initiale du vocabulaire pour avoir une idée du nombre de mots uniques dans nos données. Ensuite, nous générons un nuage de mots pour visualiser les 100 mots les plus fréquents et les 100 mots dont la fréquence documentaire est la plus grande. De plus, nous identifions les 100 mots les plus discriminants au sens de l'odds ratio. Par la suite, nous examinons la distribution d'apparition des mots pour vérifier si elle suit la loi de Zipf, et nous identifions les 100 bigrammes/trigrammes les plus fréquents dans nos données.

Ensuite, nous explorons différentes variantes de BoW, notamment TF-IDF, la réduction de la taille du vocabulaire en utilisant les paramètres min_df, max_df et max_features, BoW binaire, et l'utilisation de bi-grammes et de tri-grammes. Nous discutons des performances attendues et des avantages et inconvénients de chaque variante en termes de représentation des données textuelles.

L'objectif principal de cette section est de comprendre la structure et la distribution du vocabulaire dans nos données textuelles, ainsi que d'explorer les différentes variantes de l'approche BoW pour obtenir une représentation optimale des données en vue d'une analyse ultérieure.

FIGURE 1 – 100 mots les plus frequents sans stop words

FIGURE 2 – 100 bigrammes et trigrammes les plus frequents sans stop words

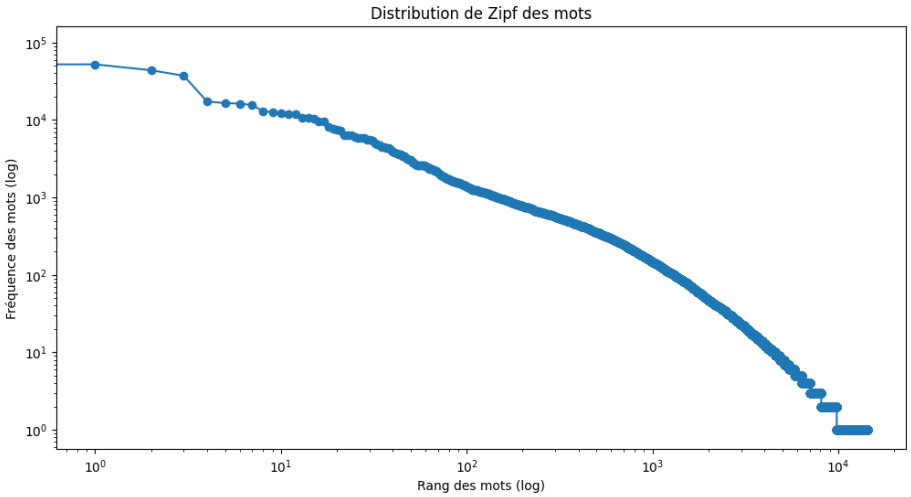

FIGURE 3 – Distribution de Zipf des mots

La loi de Zipf décrit comment les mots sont répartis dans un texte. En gros, elle dit que quelques mots sont très fréquents (comme "le" ou "et"), tandis que la plupart des autres mots apparaissent beaucoup moins souvent. C'est un peu comme une pyramide inversée : les mots les plus courants sont en haut, tandis que les mots moins courants sont en bas.

4.4 Premiers tests

Le code pour cette partie est disponible dans l'annexe 7. Dans cette première série de tests, nous évaluons trois modèles de classification : le Naive Bayes, la régression logistique et la SVM (Support Vector Machine).

Pour chacun de ces modèles, nous commençons par les entraîner sur l'ensemble d'entraînement (X_train, y_train), puis nous effectuons des prédictions sur l'ensemble de test (X_test). Ces prédictions nous permettent d'évaluer les performances de chaque modèle en utilisant différentes mesures telles que la précision, le rappel et le F1-score, qui combinent précision et rappel.

En plus de ces mesures, nous utilisons l'AUC (Area Under the Curve), qui représente l'aire sous la courbe ROC (Receiver Operating Characteristic), comme indicateur global de la performance du modèle.

Enfin, nous visualisons les performances des modèles à l'aide des courbes ROC, qui illustrent le trade-off entre le taux de faux positifs et le taux de vrais positifs.

NB accuracy :

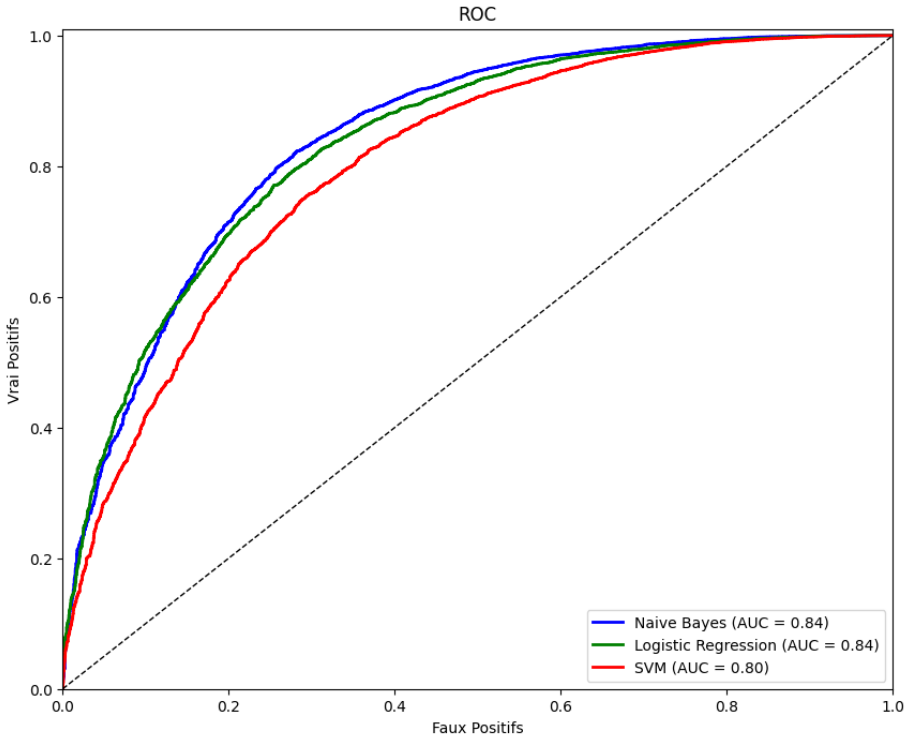

FIGURE 4 – ROC

	precision	recall	f1−score	support
−1	0.57	0.40	0.47	2998
1	0.91	0.95	0.93	19968
accuracy			0.88	22966
macro avg	0.74	0.68	0.70	22966
weighted avg	0.87	0.88	0.87	22966

LR accuracy :

	precision	recall	f1−score	support
−1	0.69	0.34	0.45	2998
1	0.91	0.98	0.94	19968
accuracy			0.89	22966
macro avg	0.80	0.66	0.70	22966

```
weighted avg       0.88      0.89      0.88     22966

SVM accuracy:
                precision    recall  f1-score   support

          -1       0.58      0.40      0.48      2998
           1       0.91      0.96      0.93     19968
...
    accuracy                           0.88     22966
   macro avg       0.75      0.68      0.70     22966
weighted avg       0.87      0.88      0.87     22966
```

Pour le reste du projet, on decide donc d'utiliser la regression logistique pour des raisons de simplicité.

4.5 Analyse code

Dans l'Annexe 8, vous trouverez le code utilisé pour traiter les données présidentielles.

Les trois pipelines de modèles présentent des différences dans les techniques d'échantillonnage et dans les hyperparamètres des modèles de régression logistique utilisés.

Le premier pipeline (first_classif_v1) utilise la méthode d'échantillonnage SMOTE pour gérer le déséquilibre des classes dans les données d'entraînement. Il utilise également la régression logistique avec validation croisée (LogisticRegressionCV) en spécifiant un scoring F1 pour évaluer les performances du modèle. De plus, il applique une pondération équilibrée des classes (class_weight='balanced') pour compenser le déséquilibre des classes.

Le deuxième pipeline (pipeline_v2) utilise la méthode d'échantillonnage RandomOverSampler pour suréchantillonner la classe minoritaire. Il effectue également une régression logistique avec validation croisée, mais cette fois-ci en spécifiant une grille d'hyperparamètres (param_grid_v2) pour effectuer une recherche sur la meilleure combinaison d'hyperparamètres, notamment les valeurs de régularisation (Cs), les pénalités (penalty), les solveurs (solver), les intercepteurs (fit_intercept), le nombre maximum d'itérations (max_iter) et les poids de classe (class_weight).

Le troisième pipeline (pipeline_v3) est similaire au deuxième, mais utilise un conteneur de pipeline différent (Pipeline) et une technique d'échantillonnage différente (SMOTE). Il utilise également une grille d'hyperparamètres (param_grid_v3) pour effectuer une recherche sur la meilleure combinaison d'hyperparamètres de régression logistique.

Les différences entre les pipelines résident principalement dans les techniques

d'échantillonnage utilisées et dans les hyperparamètres spécifiés pour les modèles de régression logistique. Ces différences peuvent avoir un impact sur les performances et la robustesse des modèles finaux.

4.6 Gaussian smoothing

```
def gaussian_pred_smoothing(model, X, sigma):
        # Definition du noyau de filtre gaussien
        pred =  model.predict_proba(X)
        smoothed_pred = gaussian_filter(pred, sigma)
        return smoothed_pred
```

La fonction réalise un lissage gaussien des probabilités de prédiction générées par un modèle de classification. La fonction commence par utiliser le modèle pour prédire les probabilités de chaque classe pour les données d'entrée (model.predict_proba(X)). Ensuite, elle applique un filtre gaussien à ces probabilités prédites en utilisant la fonction gaussian_filter avec le paramètre sigma. Cette opération de lissage gaussien permet de réduire les fluctuations brusques dans les probabilités de prédiction et d'améliorer la stabilité des probabilités de prédiction..

4.7 Resultats first_classif_v1

Le premier pipeline, first_classif_v1, montre des résultats relativement bons malgré une optimisation minimale par rapport aux deux autres pipelines. Cela peut être attribué à l'utilisation de la méthode de sur-échantillonnage SMOTE et à la spécification de la pondération des classes dans le modèle LogisticRegressionCV. La sur-échantillonnage SMOTE permet de mieux équilibrer les classes en créant des données synthétiques pour la classe minoritaire, ce qui améliore la capacité du modèle à généraliser. De plus, la spécification de la pondération des classes dans le modèle de régression logistique permet de tenir compte du déséquilibre des classes lors de l'apprentissage, ce qui peut conduire à de meilleurs résultats pour la classe minoritaire.

Output :

```
F1 Score for Chirac (label 1): 0.9303381401634935
F1 Score for Mitterand (label -1): 0.6607450955332609
Accuracy: 0.8844110603091662

Test Set:
F1 Score for Chirac (label 1): 0.9048092521848343
F1 Score for Mitterand (label -1): 0.5283899403681618
Accuracy: 0.8415919184881999
- SMOOTHED RESULTS -
Training Set:
F1 Score for Chirac (label 1): 0.9428600633752428
F1 Score for Mitterand (label -1): 0.6710797293321565
```

```
Accuracy: 0.9026344437187024

Test Set:
F1 Score for Chirac (label 1): 0.9223795456860548
F1 Score for Mitterand (label -1): 0.5426170468187275
Accuracy: 0.8672820691456936
```

4.8 Resulats pipeline_v3

Les résultats obtenus pour la pipeline_v3 montrent des performances globalement solides, avec un F1 score élevé pour la classe Chirac (label 1) mais plus faible pour la classe Mitterand (label -1). Nous choisissons de ne pas considérer la version 2 du modèle, car elle est moins performante que la version 3.

Sur l'ensemble d'entraînement, le F1 score pour Chirac est d'environ 0.87 tandis que pour Mitterand, il est d'environ 0.42. Ces résultats indiquent que le modèle a une capacité élevée à prédire la classe Chirac mais rencontre des difficultés à bien classer la classe Mitterand.

Sur l'ensemble de test, les résultats sont similaires, avec un F1 score légèrement plus élevé pour Chirac (environ 0.87) et une amélioration légère du F1 score pour Mitterand (environ 0.43). Cela suggère que le modèle généralise bien aux données non vues, mais continue de montrer une performance relativement faible pour la classe Mitterand.

```
F1 Score for Chirac (label 1): 0.8694816633747932
F1 Score for Mitterand (label -1): 0.42225897920604916
Accuracy: 0.7870672762900065

Test Set:
F1 Score for Chirac (label 1): 0.8703555199144614
F1 Score for Mitterand (label -1): 0.43088476883360716
Accuracy: 0.7888182530697553
-- SMOOTHED RESULTS --
Training Set:
F1 Score for Chirac (label 1): 0.8699966655551851
F1 Score for Mitterand (label -1): 0.4227420787681374
Accuracy: 0.7877857609405617

Test Set:
F1 Score for Chirac (label 1): 0.8707919204873357
F1 Score for Mitterand (label -1): 0.43132643461900283
Accuracy: 0.7894278498650179
```

En observant les résultats après lissage des prédictions, on constate des valeurs similaires, ce qui indique que le lissage gaussien n'a pas eu un impact significatif sur les performances du modèle. A priori cela est du au fait que les probabilités

initiales de prédiction étaient déjà relativement lisses et régulières. Donc le lissage gaussien pourrait ne pas avoir un effet notable sur les prédictions finales. Si les probabilités étaient fortement concentrées autour de certaines valeurs, le lissage pourrait aider à adoucir les prédictions, mais dans ce cas, les résultats après lissage auraient été différents.

Le problème semble résider dans le prétraitement des données et la difficulté du modèle à distinguer efficacement les deux classes. Les prédictions semblent souvent être proches d'une répartition égale entre les deux classes, ce qui suggère que le modèle ne parvient pas à discerner de manière fiable les caractéristiques distinctives de chaque classe.

5 Conclusion

Dans l'ensemble, ce projet a été une exploration enrichissante de l'analyse de discours politique à l'aide de techniques de traitement automatique du langage naturel et de modélisation de données.

En ce qui concerne la modélisation, nous avons testé plusieurs algorithmes de classification, notamment Naive Bayes, la régression logistique et SVM. Dans l'ensemble, nous avons obtenu des performances prometteuses, avec des scores de précision et de rappel relativement élevés en fonction du probleme.

Cependant, il est important de noter que notre projet a été confronté à plusieurs défis, notamment des contraintes de temps et des limitations matérielles. Avec plus de temps et de ressources, nous aurions pu explorer davantage de techniques de prétraitement et de modélisation, ce qui aurait pu améliorer encore les performances des modèles. De plus, nous avons rencontré des problèmes avec nos machines lors de l'entraînement des modèles, notamment des crashs et des temps d'entraînement trop élevés. Ces obstacles ont limité notre capacité à expérimenter pleinement et à optimiser nos modèles.

Tout le code utilisé dans ce projet, y compris les scripts de prétraitement, de modélisation, et d'évaluation des modèles, est disponible dans le fichier zip soumis avec ce rapport. Ce fichier zip contient l'intégralité du code source organisé en différents répertoires pour une meilleure lisibilité et accessibilité. De plus, le dossier contient également des fichiers pkl qui stockent les modèles entraînés ainsi que les prédictions faites pour le serveur d'évaluation.

6 Annexe Code BoW

```
from sklearn.feature_extraction.text import
    CountVectorizer, TfidfVectorizer
from wordcloud import WordCloud
from collections import Counter
from nltk.corpus import stopwords
from sklearn.feature_selection import mutual_info_classif
import pandas as pd

''' Exploration pr liminaire des jeux de donn es '''
nltk.download('stopwords')

all_words = ' '.join(alltxts).split()

# frequence de chaque mot avec counter
word_freq = Counter(all_words)

# Quelle est la taille d'origine du vocabulaire?
original_vocab_size = len(word_freq)
print(f"Taille d'origine du vocabulaire: {
    original_vocab_size}")

# - Que reste-t-il si on ne garde que les 100 mots les
    plus fr quents? [word cloud]
#removed stopwords
vectorizer = CountVectorizer(stop_words=stopwords.words("
    french"))
X = vectorizer.fit_transform(alltxts)
frequent_words = pd.Series(
    np.array(X.sum(axis=0))[0], index=sorted(vectorizer.
        vocabulary_)
)

wordcloud = WordCloud(background_color="white", max_words
    =100, width=2000, height=1000)
wordcloud.generate_from_frequencies(frequent_words)
plt.imshow(wordcloud)
plt.axis("off")
plt.show()

print("20 most frequent words")
print(frequent_words.sort_values(ascending=False)[:20])
print(len(frequent_words))
print(sum(frequent_words))
```

```
#kept stopwords
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(alltxts)
frequent_words = pd.Series(
    np.array(X.sum(axis=0))[0], index=sorted(vectorizer.
        vocabulary_)
)

wordcloud = WordCloud(background_color="white", max_words
    =100, width=2000, height=1000)
wordcloud.generate_from_frequencies(frequent_words)
plt.imshow(wordcloud)
plt.axis("off")
plt.show()

print("20 most frequent words")
print(frequent_words.sort_values(ascending=False)[:20])
print(len(frequent_words))
print(sum(frequent_words))

#Quelle est la distribution d'apparition des mots (Zipf)
word_counts = np.array(X.sum(axis=0))[0]
sorted_indices = np.argsort(word_counts)[::-1]

sorted_word_counts = word_counts[sorted_indices]
plt.figure(figsize=(12, 6))
plt.plot(sorted_word_counts, marker='o')
plt.xscale('log')
plt.yscale('log')
plt.title('Distribution de Zipf des mots')
plt.xlabel('Rang des mots (log)')
plt.ylabel('Fr quence des mots (log)')
plt.show()

#- Quels sont les 100 mots dont la fr quence
    documentaire est la plus grande? [word cloud]
#removed stopwords
vectorizer = TfidfVectorizer(stop_words=stopwords.words("
    french"))
X = vectorizer.fit_transform(alltxts)
frequent_words = pd.Series(
    np.array(X.sum(axis=0))[0], index=sorted(vectorizer.
        vocabulary_)
)
```

```
wordcloud = WordCloud(background_color="white", max_words
    =100, width=2000, height=1000)
wordcloud.generate_from_frequencies(frequent_words)
plt.imshow(wordcloud)
plt.axis("off")
plt.show()

print("20 most frequent words")
print(frequent_words.sort_values(ascending=False)[:20])
print(len(frequent_words))
print(sum(frequent_words))

#kept stopwords
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(alltxts)
frequent_words = pd.Series(
    np.array(X.sum(axis=0))[0], index=sorted(vectorizer.
        vocabulary_)
)

wordcloud = WordCloud(background_color="white", max_words
    =100, width=2000, height=1000)
wordcloud.generate_from_frequencies(frequent_words)
plt.imshow(wordcloud)
plt.axis("off")
plt.show()

print("20 most frequent words")
print(frequent_words.sort_values(ascending=False)[:20])
print(len(frequent_words))
print(sum(frequent_words))

#Quels sont les 100 bigrammes/trigrammes les plus
    fr quents?
#removed stopwords
vectorizer = CountVectorizer(stop_words=stopwords.words("
    french"), ngram_range=(2,3))
X = vectorizer.fit_transform(alltxts)
frequent_words = pd.Series(
    np.array(X.sum(axis=0))[0], index=sorted(vectorizer.
        vocabulary_)
)

wordcloud = WordCloud(background_color="white", max_words
    =100, width=2000, height=1000)
wordcloud.generate_from_frequencies(frequent_words)
```

```
plt.imshow(wordcloud)
plt.axis("off")
plt.show()

print("20 most frequent bigrams and trigrams")
print(frequent_words.sort_values(ascending=False)[:20])
print(len(frequent_words))
print(sum(frequent_words))

#kept stopwords
vectorizer = CountVectorizer(ngram_range=(2,3))
X = vectorizer.fit_transform(alltxts)
frequent_words = pd.Series(
    np.array(X.sum(axis=0))[0], index=sorted(vectorizer.
        vocabulary_)
)

wordcloud = WordCloud(background_color="white", max_words
    =100, width=2000, height=1000)
wordcloud.generate_from_frequencies(frequent_words)
plt.imshow(wordcloud)
plt.axis("off")
plt.show()

print("20 most frequent bigrams and trigrams")
print(frequent_words.sort_values(ascending=False)[:20])
print(len(frequent_words))
print(sum(frequent_words))
```

7 Annexe Premiers Tests

```
def tests_models(X_train, X_test, y_train, y_test,
    class_weights=None, smoothing=None):
    naive_bayes = MultinomialNB()
    naive_bayes.fit(X_train, y_train)

    logistic_reg = LogisticRegression(class_weight=
        class_weights, n_jobs=-1) #n_jobs = -1 pour
        utiliser tous les processeurs
    logistic_reg.fit(X_train, y_train)

    svm = LinearSVC(class_weight=class_weights)
    svm.fit(X_train, y_train)
```

```
if smoothing is None:
    pred_nb = naive_bayes.predict(X_test)
    pred_lr = logistic_reg.predict(X_test)
    pred_svm = svm.predict(X_test)
#gaussian smoothing, on passe la fonction dans l'
    attribut smoothing de tests_models
else:
    try:
        smoothing_size = 5 #a modifier pour voir l'
            impact
        pred_nb = smoothing(naive_bayes.predict_proba
            (X_test)[:, 1], smoothing_size)
        pred_lr = smoothing(logistic_reg.
            predict_proba(X_test)[:, 1],
            smoothing_size)
        pred_svm = smoothing(svm.decision_function(
            X_test), smoothing_size)

        # threshold sinon on a un mix de valeurs
            binaires et continues
        pred_nb = (pred_nb > 0.5).astype(int)
        pred_lr = (pred_lr > 0.5).astype(int)
        pred_svm = (pred_svm > 0.5).astype(int)
    except Exception as e:
        print("Error with the gaussian smoothing:", e
            )


#fonction de scikit pour print les metriques
    interresantes
print(f"NB accuracy: \n{classification_report(y_test
    , pred_nb, zero_division=1)}")
print(f"LR accuracy: \n{classification_report(y_test
    , pred_lr, zero_division=1)}")
print(f"SVM accuracy: \n{classification_report(
    y_test, pred_svm, zero_division=1)}")

#on recupere les faux positifs et vrai positifs pour
    chaque modele
#pour ca on utilise la proba d'apartenir a une classe
     plutot que la classe predite
fp_naive_bayes, vp_naive_bayes, _ = roc_curve(y_test,
     naive_bayes.predict_proba(X_test)[:, 1]) #on
    recupere la proba de la classe "positive"
fp_logistic_reg, vp_logistic_reg, _ = roc_curve(
    y_test, logistic_reg.predict_proba(X_test)[:, 1])
```

```
    fp_svm, vp_svm, _ = roc_curve(y_test, svm.
        decision_function(X_test))
    #le _ ici permet d'ignorer la veleur du threshold
        comme on l'utilise pas ici

    roc_auc_nb = auc(fp_naive_bayes, vp_naive_bayes)
    roc_auc_lr = auc(fp_logistic_reg, vp_logistic_reg)
    roc_auc_svm = auc(fp_svm, vp_svm)

    plt.figure(figsize=(10, 8))
    plt.plot(fp_naive_bayes, vp_naive_bayes, color='blue'
        , lw=2, label=f'Naive Bayes (AUC = {roc_auc_nb:.2f
        })') #le :.2f permet d'avoir la valeur au centieme
         pres
    plt.plot(fp_logistic_reg, vp_logistic_reg, color='
        green', lw=2, label=f'Logistic Regression (AUC = {
        roc_auc_lr:.2f})')
    plt.plot(fp_svm, vp_svm, color='red', lw=2, label=f'
       SVM (AUC = {roc_auc_svm:.2f})')
    plt.plot([0, 1], [0, 1], color='black', lw=1,
        linestyle='--') #random classifier
    plt.xlim([0.0, 1.0]) #limite des valeurs pour les
        axes
    plt.ylim([0.0, 1.01]) #je mets un peu plus que 1 ici
        sinon la visibilit  est pas parfaite
    plt.xlabel('Faux Positifs')
    plt.ylabel('Vrai Positifs')
    plt.title('ROC')
    plt.legend(loc="lower right") #legende en bas a
        droite
    plt.show()

tests_models(X_train, X_test, y_train, y_test)
```

8 Annexe Code President

```
import numpy as np
import matplotlib.pyplot as plt
import os
import codecs
import re
import time
import joblib
from sklearn.feature_extraction.text import
```

```
    TfidfVectorizer
from sklearn.linear_model import LogisticRegressionCV
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.model_selection import GridSearchCV
from imblearn.over_sampling import RandomOverSampler,
    SMOTE
from scipy.ndimage import gaussian_filter
import string
import unicodedata
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem.snowball import FrenchStemmer
nltk.download('punkt')

# Chargement des donnes:
def load_pres(fname):
    alltxts = []
    alllabs = []
    s=codecs.open(fname, 'r','utf-8') # pour rgler le
        codage
    while True:
        txt = s.readline()
        if(len(txt))<5:
            break
        #
        lab = re.sub(r"<[0-9]*:[0-9]*:(.)>.*","\\1",txt)
        txt = re.sub(r"<[0-9]*:[0-9]*:.>(.*)","\\1",txt)
        if lab.count('M') >0:
            alllabs.append(-1)
        else:
            alllabs.append(1)
        alltxts.append(txt)
    return alltxts,alllabs

fname = "../datasets/AFDpresidentutf8/corpus.tache1.learn
    .utf8"
alltxts,alllabs = load_pres(fname)

#
    ----------------------------------------------------------------------


print('OVERVIEW DATASET POUR PRESIDENTS \n')
print(f'{len(alltxts)} phrases')
print('Chirac == label 1 et Mitterand == label -1 \n')
```

```
print(f'{alltxts[0]} -> classe: {alllabs[0]}')
print(f'{alltxts[11]} -> classe: {alllabs[11]}')
print(f'Chirac: {np.sum(np.array(alllabs) == 1)} phrases
    - Mitterand: {np.sum(np.array(alllabs) == -1)} phrases
    ')
print(f'on remarque que Chirac a parl {np.round
    (49890/7523)} fois plus que Mitterand\n')

pretraitement = True
if pretraitement:
    print("Doing pre treatement...")
    punc = string.punctuation
    punc += '\n\r\t'
    stemmer = FrenchStemmer()

    for i in range(0,len(alltxts)):
        #alltxts[i] = re.sub('[0-9]+', '', alltxts[i]) #a
            garder ou enlever
        alltxts[i] = alltxts[i].lower()
        alltxts[i] = alltxts[i].translate(str.maketrans(
            punc, ' ' * len(punc))) #ponctuation
        alltxts[i] = unicodedata.normalize('NFD', alltxts
            [i]).encode('ascii', 'ignore').decode("utf-8")
            #normalize en unicode, enleve non ascii et
            reconvertir en utf 8
        words = word_tokenize(alltxts[i], language='
            french')
        stemmed = [stemmer.stem(word) for word in words]
        alltxts[i] = ' '.join(stemmed)

#
```

```
#fonction d'utilite
def gaussian_pred_smoothing(model, X, sigma):
        # D finition du noyau de filtre gaussien
        pred =  model.predict_proba(X)
        smoothed_pred = gaussian_filter(pred, sigma)
        return smoothed_pred

#
```

```
tfidf = TfidfVectorizer(use_idf=True, norm='l2',
```

```
    smooth_idf=True, lowercase=False)
X = tfidf.fit_transform(alltxts)

X_train, X_test, y_train, y_test = train_test_split(X,
    alllabs, test_size=0.20, random_state=12)

oversampler = SMOTE(sampling_strategy='minority',
    random_state=0, k_neighbors=5) #je mets ici
    arbitraiement grace aux tests
X_train_resampled, y_train_resampled = oversampler.
    fit_resample(X_train, y_train)

#
```

```
num_cores = os.cpu_count()

print("Number of CPU cores available:", num_cores)
cores = 5 #hard coded temporairement pour l'entrainement

#
```

```
from imblearn.pipeline import make_pipeline, Pipeline

first_classif_v1 = make_pipeline(
        oversampler,
        LogisticRegressionCV(
            cv=5,
            scoring='f1',
            random_state=0,
            n_jobs=-1,
            verbose=3,
            max_iter=1000,
            class_weight='balanced'
        )
    )

first_classif_v1.fit(X_train_resampled, y_train_resampled
    )

#
```

```
#second pipeline test
pipeline_v2 = make_pipeline(
    RandomOverSampler(sampling_strategy='minority',
        random_state=0),
    LogisticRegressionCV(cv=5, scoring='f1', n_jobs=cores
        , verbose=2, max_iter=1000)
)

param_grid_v2 = {
    'logisticregressioncv__Cs': [0.001, 0.01, 0.1, 1],
    'logisticregressioncv__penalty': ['l1', 'l2'],
    'logisticregressioncv__solver': ['liblinear', 'saga'
        ],
    'logisticregressioncv__fit_intercept': [True],
    'logisticregressioncv__max_iter': [1000, 2000],
    'logisticregressioncv__class_weight': ['balanced',
        {1: 1, -1: 10}]
}

#
```

```
pipeline_v3 = Pipeline([
    ('sampling', 'passthrough'),
    ('classifier', LogisticRegressionCV(cv=5, scoring='f1
        ', n_jobs=cores, verbose=1))
])

    # Define the parameter grid for grid search
param_grid_v3 = {
        'sampling': [
                    SMOTE(sampling_strategy='minority',
                        random_state=0, k_neighbors=5)],
                        # RandomOverSampler(
                        sampling_strategy='minority',
                        random_state=0),
        'classifier__Cs': [0.1, 1, 10],
        'classifier__penalty': ['l1', 'l2'],
        'classifier__solver': ['liblinear', 'saga'],
        'classifier__fit_intercept': [True],
        'classifier__max_iter': [2000],
        'classifier__class_weight': ['balanced', {1: 1,
            -1: 50}]
    }
```

```
#
```

```
# Create GridSearchCV object with scoring='f1'
model_filename = "best_logistic_regression_model.pkl"
train = True
if train:
    start_time = time.time()
    grid_search = GridSearchCV(estimator=pipeline_v3,
        param_grid=param_grid_v3, cv=5, scoring='f1',
        n_jobs=cores, verbose=1)

    # Fit the GridSearchCV object to the training data
    grid_search.fit(X_train, y_train)

    elapsed_time = time.time() - start_time
    print("Grid search fait en {:.2f} seconds.".format(
        elapsed_time))

    log_reg_classifier = grid_search.best_estimator_

    joblib.dump(log_reg_classifier, 'best_model.pkl')

    print("Model saved")
```

```
#
```

```
#loading model
log_reg_classifier = joblib.load('best_model.pkl')
print("Best parameters found:")
print(log_reg_classifier.named_steps['classifier'])
print("Best sampler:")
print(log_reg_classifier.named_steps['sampling'])

test = False
if test:
    log_reg_classifier = first_classif_v1
```

```
#
```

```
def convert_to_labels(probas):
```

```
    labels = np.where(probas[:, 1] > 0.5, 1, -1)
    return labels

#
```

```
y_pred_train = log_reg_classifier.predict(X_train)
y_pred_test = log_reg_classifier.predict(X_test)

#
```

```
#On cherche la meilleure valeur de sigma pour le gaussian
    smoothing

sigma_values = [round(x, 1) for x in np.arange(0.1, 1.1,
   0.1)]
sigma_values.extend([2,4,5,8,10,20,30])
best_sigma = None
best_f1_score = 0.0

for sigma in sigma_values:
    # Apply Gaussian smoothing
    smoothed_pred_train = gaussian_pred_smoothing(
        log_reg_classifier, X_train, sigma)
    smoothed_pred_test = gaussian_pred_smoothing(
        log_reg_classifier, X_test, sigma)

    # Convert smoothed probabilities to labels
    smoothed_pred_train_labels = convert_to_labels(
        smoothed_pred_train)
    smoothed_pred_test_labels = convert_to_labels(
        smoothed_pred_test)

    # Calculate F1 score for class -1
    f1_score_class_minus_one = f1_score(y_test,
        smoothed_pred_test_labels, pos_label=-1)

    # Update best sigma if current F1 score is higher
    if f1_score_class_minus_one > best_f1_score:
        best_f1_score = f1_score_class_minus_one
        best_sigma = sigma

print(f'Best sigma value found: {best_sigma}')
```

```
smoothed_pred_train = gaussian_pred_smoothing(
    log_reg_classifier, X_train, best_sigma)
smoothed_pred_test = gaussian_pred_smoothing(
    log_reg_classifier, X_test, best_sigma)

smoothed_pred_train_labels = convert_to_labels(
    smoothed_pred_train)
smoothed_pred_test_labels = convert_to_labels(
    smoothed_pred_test)

# ----------------------------------------------------------------------


print('Before smoothing:\n')
print(log_reg_classifier.predict_proba(X_train))

print('After smoothing:')
print(smoothed_pred_train)
print(smoothed_pred_train_labels)

# ----------------------------------------------------------------------


#les f1 scores
f1_train_chirac = f1_score(y_train, y_pred_train,
    pos_label=1)
f1_train_mitterand = f1_score(y_train, y_pred_train,
    pos_label=-1)

f1_test_chirac = f1_score(y_test, y_pred_test, pos_label
    =1)
f1_test_mitterand = f1_score(y_test, y_pred_test,
    pos_label=-1)

# ----------------------------------------------------------------------


#accuracy
accuracy_train = accuracy_score(y_train, y_pred_train)
accuracy_test = accuracy_score(y_test, y_pred_test)

#display resultats
print("F1 Score for Chirac (label 1):", f1_train_chirac)
print("F1 Score for Mitterand (label -1):",
    f1_train_mitterand)
print("Accuracy:", accuracy_train)
```

```
print("\nTest Set:")
print("F1 Score for Chirac (label 1):", f1_test_chirac)
print("F1 Score for Mitterand (label -1):",
    f1_test_mitterand)
print("Accuracy:", accuracy_test)

#
-------------------------------------------------------------------------------


print('- SMOOTHED RESULTS -')
print("Training Set:")
print("F1 Score for Chirac (label 1):", f1_score(y_train,
     smoothed_pred_train_labels, pos_label=1))
print("F1 Score for Mitterand (label -1):", f1_score(
    y_train, smoothed_pred_train_labels, pos_label=-1))
print("Accuracy:", accuracy_score(y_train,
    smoothed_pred_train_labels))

print("\nTest Set:")
print("F1 Score for Chirac (label 1):", f1_score(y_test,
    smoothed_pred_test_labels, pos_label=1))
print("F1 Score for Mitterand (label -1):", f1_score(
    y_test, smoothed_pred_test_labels, pos_label=-1))
print("Accuracy:", accuracy_score(y_test,
    smoothed_pred_test_labels))

#
-------------------------------------------------------------------------------


#prediction sur le fichier test pour serveur d'eval
```