RITAL

TME: Classification textuelle

Kevin Meetooa

Table des matières

Chapitre 1

Classification de locuteurs

1.1 Objectif

Dans cette partie, on s'intéressera à la classification de locuteurs. On dispose d'une base de données composée de phrases prononcées par Chirac et Mitterand durant différents discours qu'ils ont donné lorsqu'ils étaient présidents.

On aimerait classifier un ensemble de données non labellisé de phrases prononcées par ces mêmes présidents. Nous sommes donc confrontés à un problème de classification textuelle.

1.2 Prise en main des données

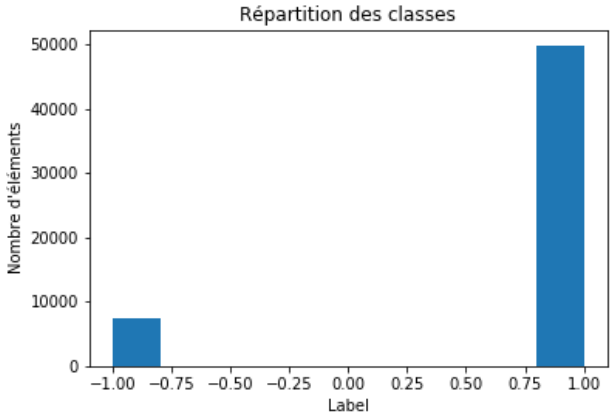

FIGURE 1.1 – Répartition des classes

La figure 1.1 montre la répartition des deux classes dans le fichier d'entraînement nous ayant été fourni. Nous pouvons observer que les classes sont fortement déséquilibrées (7 fois plus d'éléments dans la classe positive que dans la classe négative).

Il faut prendre ce déséquilibre en compte avant de s'intéresser au problème de classification. En effet, si l'on donne les données telles quelles à notre classifieur, on risque d'obtenir un classifieur entraîné à ne prédire (presque) que la classe positive.

-

	precision	recall	f1-score	support
-1.0	0.62	0.45	0.52	2218
1.0	0.92	0.96	0.94	15006
accuracy			0.89	17224
macro avg	0.77	0.71	0.73	17224
weighted avg	0.88	0.89	0.89	17224

FIGURE 1.2 – Classification report sur les données initiales

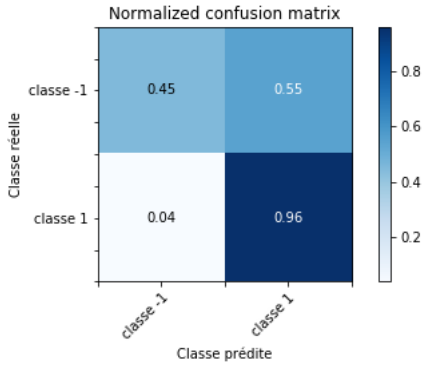

FIGURE 1.3 – Matrice de confusion sur les données initiales

Pour s'en convaincre, on peut effectuer une validation croisée sur les données telles quelles et observer les performances de notre classifieur. Au vu de la taille des données, on peut commencer par utiliser un classifieur de type Naive Bayes qui s'exécutera en temps raisonnable.

La figure 1.2 montre le classification_report obtenu sur les données telles quelles. Si l'on regarde uniquement la métrique d'accuracy, le modèle peut sembler satisfaisant. Cependant, le score de rappel montre que les performances du classifieur sur la classe -1 sont mauvaises.

Cela est d'autant plus visible sur la figure 1.3 montrant la matrice de confusion : Malgré les très bonnes performances sur la classe majoritaire (96% de bonnes prédictions), les performances sur la classe minoritaire sont mauvaises. Cette dernière est prédite correctement seulement 45% du temps, le taux de faux-positifs est bien trop important pour considérer le modèle comme satisfaisant.

D'où l'intérêt d'utiliser une ou des métriques adaptées. Pour palier à ce problème de faux-positifs, nous allons procéder à un équilibrage des données.

Pour équilibrer les données, deux choix s'offrent à nous : On peut enlever des données de la classe majoritaire (sous-échantillonage) ou rajouter des données dans la classe minoritaire, par duplication ou génération par exemple (sur-échantillonage).

Au vu du déséquilibre important entre les classes, un sur-échantillonage semble inadapté et pourrait conduire à un sur-apprentissage sur la classe négative. Nous avons donc fait le choix d'utiliser un sous-échantillonage afin d'obtenir des classes équilibrées.

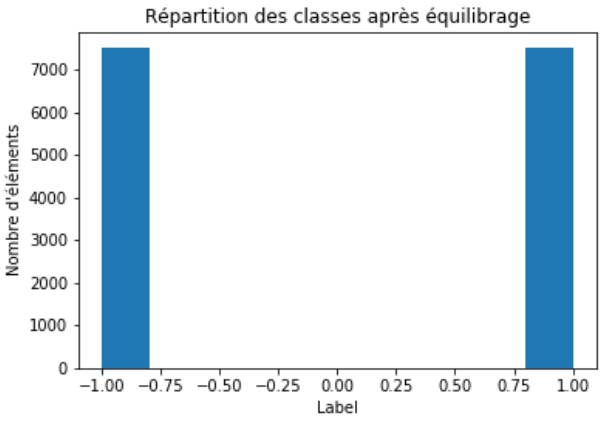

FIGURE 1.4 – Répartition des classes

La figure 1.4 montre la répartition des classes après équilibrage des données par sous-échantillonage.

En effectuant la même démarche que précédemment, on affiche le classification_report ainsi que la matrice de confusion obtenue afin de tester les performances du classifieur sur les nouvelles données.

	precision	recall	f1-score	support
-1	0.77	0.81	0.79	2262
1	0.80	0.76	0.78	2252
accuracy			0.78	4514
macro avg	0.78	0.78	0.78	4514
weighted avg	0.78	0.78	0.78	4514

FIGURE 1.5 – Classification report sur les données équilibrées

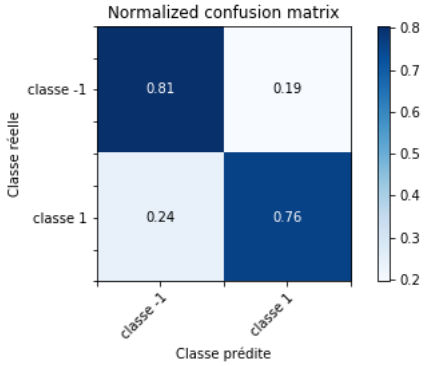

FIGURE 1.6 – Matrice de confusion sur les données équilibrées

Les figures 1.5 et 1.6 montrent que le classifieur est maintenant beaucoup plus robuste et ne présente plus de taux de faux-positifs aberrant. Ainsi, dans toute la suite de cette partie, nous testerons un classifieur entraîné sur des données équilibrées.

1.3 Pré-traitement des données

Les données sont sous la forme de texte brut. Il est possible de faire une étape de pré-traitement des données afin d'enlever les mots très fréquents, dits « stopwords » ainsi que les caractères inutiles tels que les caractères de ponctuation. De plus, il est possible de faire un « stemming » afin de raciniser les mots pour mieux identifier certains champs lexicaux.

Cependant, les résultats obtenus sont peu concluants. En mesurant la performance d'un classifieur de type Naive Bayes basique sur les données pré-traitées, on obtient de (légèrement) moins bons résultats qu'avec les données brutes. Cela peut sembler étrange mais il est possible que les noms propres soient discriminants, de même pour certaines expressions et tournures de phrases. Ainsi, dans la suite de cette partie, nous travaillerons avec les données brutes.

1.4 Représentation vectorielle des données et optimisation des paramètres

Pour représenter les données, on peut utiliser une représentation présentielle, fréquentielle ou TFIDF. Dans cette partie, nous allons comparer ces trois représentations lorsque différents paramètres de vectorisation varient. La métrique utilisée sera la métrique de $score\_f_1$ qui est bien adaptée lorsque les classes sont déséquilibrées. Toutes les courbes de score qui vont suivre ont été obtenues par **validation croisée** à 5 folds.

Le premier paramètre testé est *max_features*. Ce paramètre définit la taille maximale du dictionnaire (i.e. le nombre de mots uniques maximal) utilisé pour la vectorisation des données.

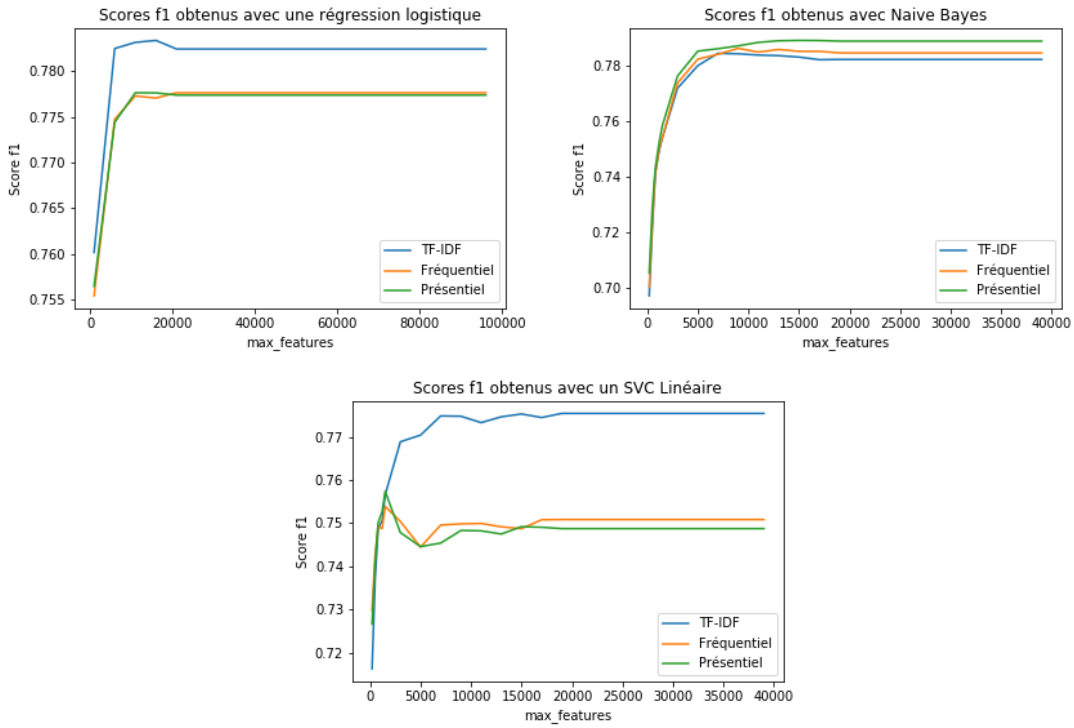

FIGURE 1.7 – Scores f1 obtenus en fonction de *max_features*

Les illustrations de la figure 1.7 montrent que pour les trois classifieurs ainsi que les trois représentations, on observe globalement le même comportement : La courbe de score f1 croît rapidement jusqu'à un certain seuil puis elle reste constante à ce seuil. On peut en déduire qu'après une certaine taille, augmenter la taille du dictionnaire n'améliore pas les performances. On remarque qu'aux alentours de $max\_features = 20000$, le score f1 est à son niveau maxi-

mal. On remarque aussi que pour de faibles valeurs de *max_features* le score f1 s'effondre, ce qui est cohérent : La perte d'information est trop importante pour obtenir une classification correcte.

Le deuxième paramètre testé est *min_df*. Ce paramètre permet de fixer une fréquence minimale d'apparition des mots. Ainsi, *min_df* =0.05 permettra de considérer uniquement les mots apparaissant dans plus de 5% du corpus. Les mots apparaissant dans moins de 5% du corpus seront alors ignorés.

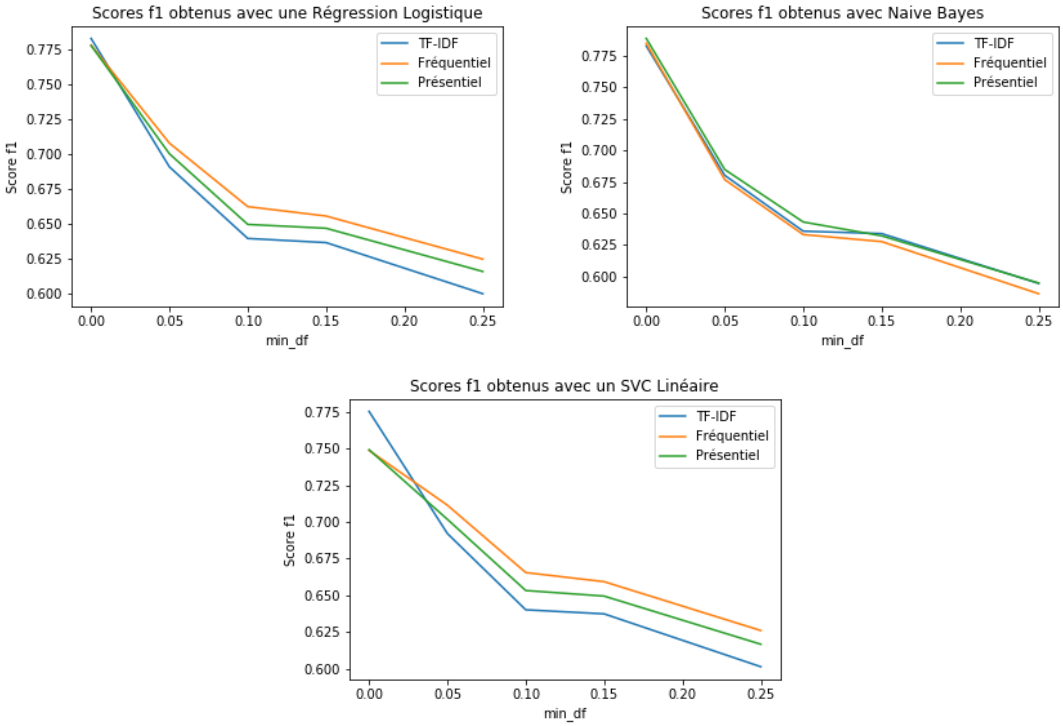

FIGURE 1.8 – Scores f1 obtenus en fonction de *min_df*

Les illustrations de la figure 1.8 montrent que pour les trois classifieurs et les trois vectoriseurs les résultats sont les mêmes : Plus la valeur de *min_df* augmente, plus le score f1 diminue. Cela est cohérent. En effet, les mots peu fréquents peuvent être très discriminants : Se passer d'eux fait perdre de l'information, ce qui explique la chute des scores lorsque *min_df* augmente.

Nous avons ensuite testé les valeurs du paramètre *max_df*. Ce paramètre permet de limiter la taille du dictionnaire en ignorant les mots les plus fréquents. Ainsi, *max_df* = 0.75 permettra d'ignorer les mots apparaissant dans plus de 75% du corpus. Seuls les mots apparaissant dans moins de 75% du corpus seront alors conservés pour la vectorisation.

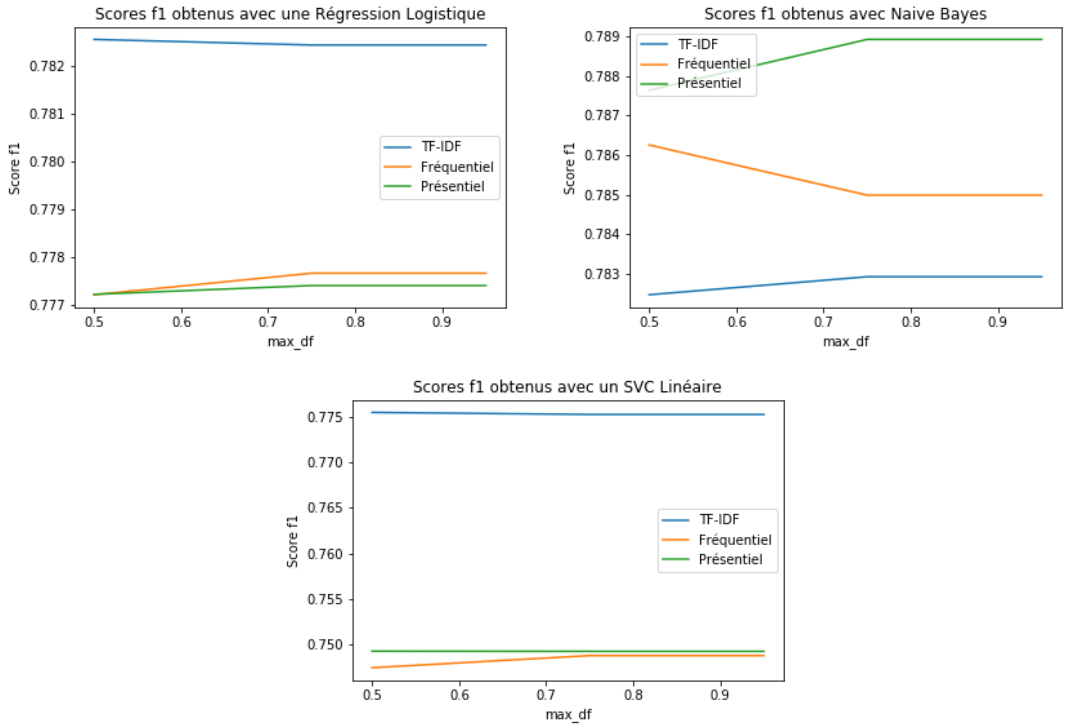

FIGURE 1.9 – Scores f1 obtenus en fonction de *max_ df*

Les résultats obtenus sont intéressants. On peut voir sur la figure 1.9 que lorsque *max_ df* varie dans l'intervalle $[0.75, 1]$, les scores restent à peu près constants. Cela est cohérent avec le fait que les mots les plus fréquents sont peu discriminants et n'aident pas dans un problème de classification. On peut donc limiter la taille du dictionnaire en fixant *max_ df* à une valeur d'environ 0.75

Enfin, le dernier paramètre de vectorisation testé est *ngram_ range* Ce paramètre permet de choisir si l'on veut considérer les mots comme des uni-grammes, des bi-grammes, etc.

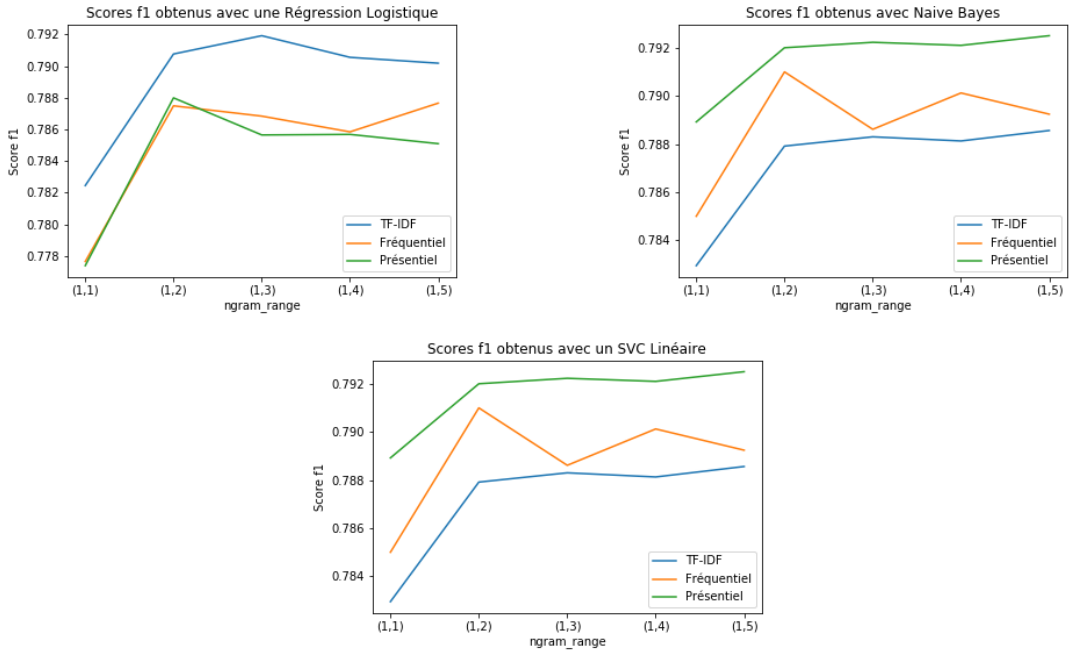

FIGURE 1.10 – Scores f1 obtenus en fonction de *ngram_ range*

On remarque des comportements globalement assez similaires sur les illustrations de la figure 1.10. Considérer les mots comme des bi-grammes donne de meilleurs scores que la représentation sous forme d'uni-grammes. Cela est cohérent. En effet, considérer le contexte d'un mot renvoie de meilleurs résultats que considérer le texte comme des mots isolés. La représentation qui semble donner les meilleurs résultats de façon générale est la représentation sous forme de bi-grammes.

Après un gridsearch sur chaque classifieur en faisant varier tous les paramètres testés plus haut en même temps, on trouve les meilleurs résultats avec une vectorisation TF-IDF et un classifieur SVC Linéaire. Le tokenizer utilisé est un tokenizer personnalisé gardant la ponctuation des textes. Les paramètres du vectoriseur TF-IDF sont les suivants :

$min\_df = 0$
$max\_df = 0.75$
$ngram\_range = (1, 2)$
$lowercase = False$

Après validation croisée sur 5 folds, le $score\_f_1$ moyen obtenu est de **80.76%**

1.5 Post-traitement des données

Afin de palier à la perte d'informations liée au sous-échantillonage, nous allons faire des prédictions en faisant la moyenne de 5 prédictions obtenues par 5 modèles entraînés sur des données sous-échantillonnées de façon aléatoire. Par exemple, si pour un texte donné, la classe positive est prédite 3 fois et la classe négative 2 fois, on prédira la classe positive. L'objectif étant d'entraîner le modèle sur le plus de données de la classe majoritaire possibles.

Nous avons ensuite "lissé" nos prédictions afin d'obtenir les mêmes "blocs de prédictions" que dans le fichier train. Par exemple, les prédictions de type "CCCCMCC" seront remplacées par "CCCCCCC"

Chapitre 2

Classification de sentiments

2.1 Objectif

Dans cette partie, on s'intéressera à la classification de sentiments. On dispose d'une base de données textuelle labellisée divisée en deux (sentiments positifs et négatifs)

On aimerait classifier un ensemble de textes non labellisés. Nous sommes donc ici aussi confrontés à un problème de classification textuelle.

2.2 Prise en main des données

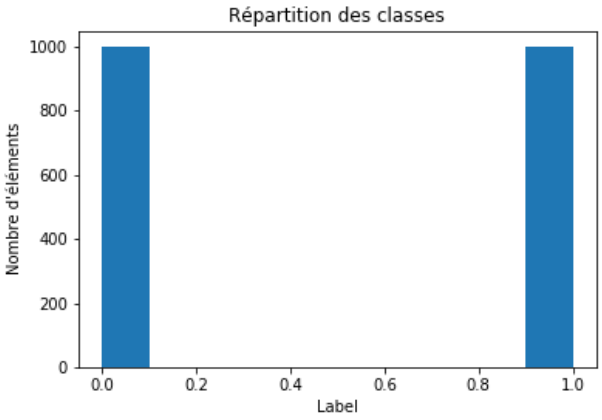

FIGURE 2.1 – Répartition des classes

La figure 2.1 montre la répartition des deux classes dans le fichier d'entraînement nous ayant été fourni.

On remarque que cette fois-ci, les données sont parfaitement équilibrées. Nous pouvons donc passer directement à l'optimisation de paramètres.

Pour optimiser les paramètres, nous pouvons cette fois utiliser la métrique d'accuracy (précision) qui semble mieux adaptée au vu de la répartition des données.

2.3 Pré-traitement des données

Comme dans la partie précédente, il est possible de faire une étape de pré-traitement des données.

Cette fois-ci, le pré-traitement des données augmente les performances du modèle. On obtient **80.65%** d'accuracy avec les données brutes et **82.96%** avec des données pré-traitées (stemming, tokenization, lettres minuscules, stopwords).

Nous travaillerons donc avec des donées pré-traitées dans la suite de cette partie.

2.4 Représentation vectorielle des données et optimisation des paramètres

Encore une fois, pour représenter les données, on peut utiliser une représentation présentielle, fréquentielle ou TFIDF. Nous allons comparer ces trois représentations en utilisant le même protocole expérimental que dans la partie précédente.

Le premier paramètre testé est *max_features*.

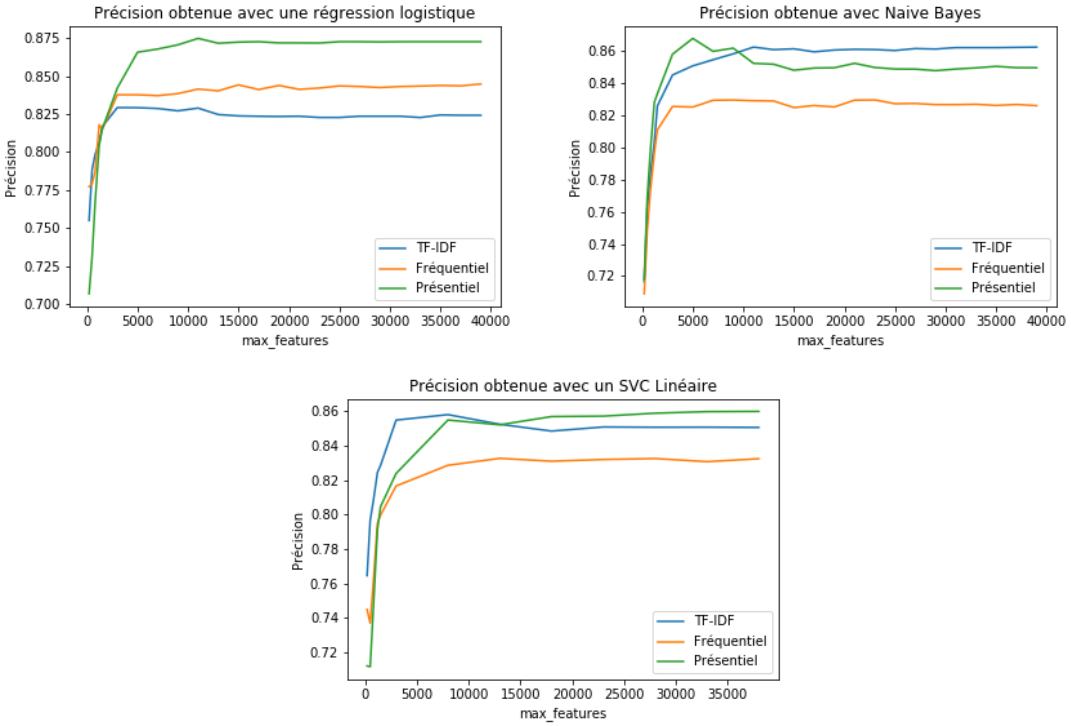

FIGURE 2.2 – Précision obtenue en fonction de *max_features*

Les illustrations de la figure 2.2 montrent que la précision est maximale dans (approximativement) l'intervalle [5000, 10000]. Le vocabulaire contient environ 25000 mots, on peut encore une fois en déduire que considérer un grand nombre de mots n'améliore pas les performances du classifieur. La classification de sentiments fonctionne bien en considérant les mots les plus

fréquents.
Le deuxième paramètre testé est *min_ df*.

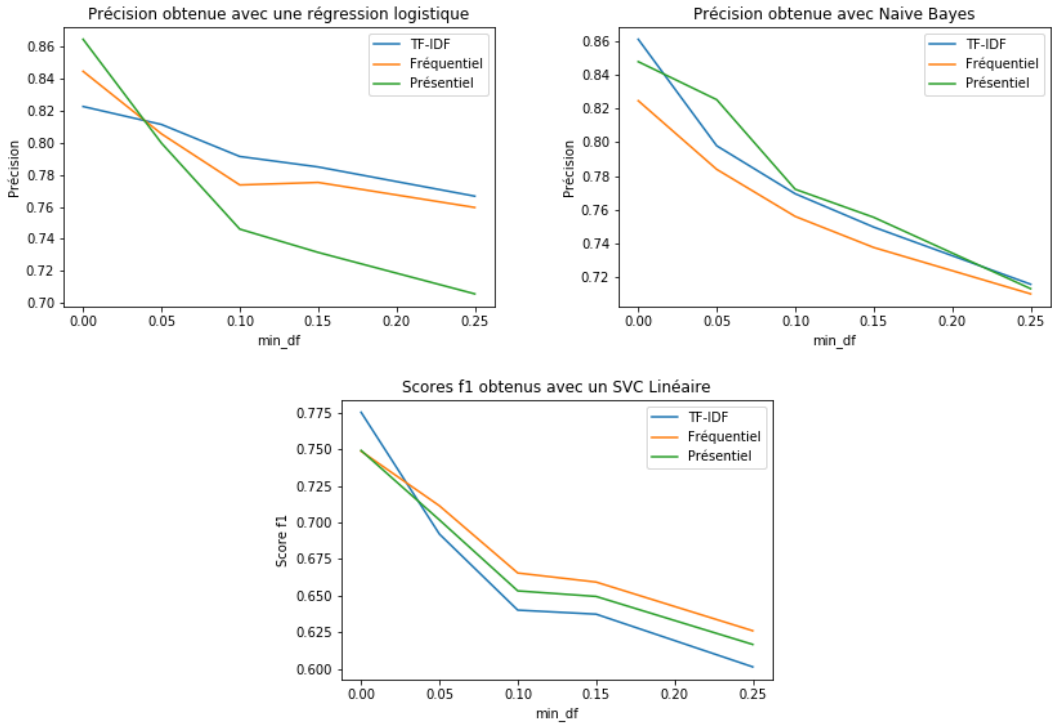

FIGURE 2.3 – Précision obtenue en fonction de *min_ df*

Les illustrations de la figure 2.3 montrent que pour les trois classifieurs et les trois vectoriseurs les résultats sont les mêmes : Plus la valeur de *min_ df* augmente, plus le score de précision diminue. Nous avons vu plus haut qu'il pouvait être intéressant de limiter la taille du dictionnaire en ne considérant que les mots les plus fréquents. Le paramètre *min_ df* pourrait augmenter le score mais pour de très faible valeurs, 0.05 étant déjà une trop grande valeur.

Nous avons ensuite testé les valeurs du paramètre *max_ df*. Ce paramètre permet de limiter la taille du dictionnaire en ignorant les mots les plus fréquents.

Encore une fois, la figure 2.4 montre que pour les vectoriseurs fréquentiels et TF-IDF, la valeur optimale de *max_ df* se situe aux alentours de 0.75

Enfin, le dernier paramètre de vectorisation testé est *ngram_ range*

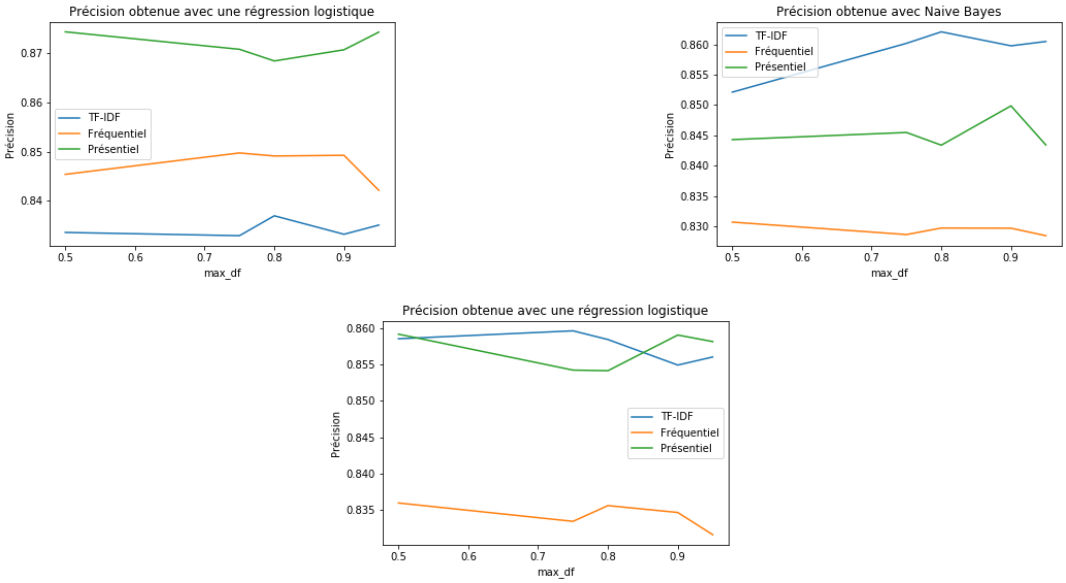

FIGURE 2.4 – Précision en fonction de *max_ df*

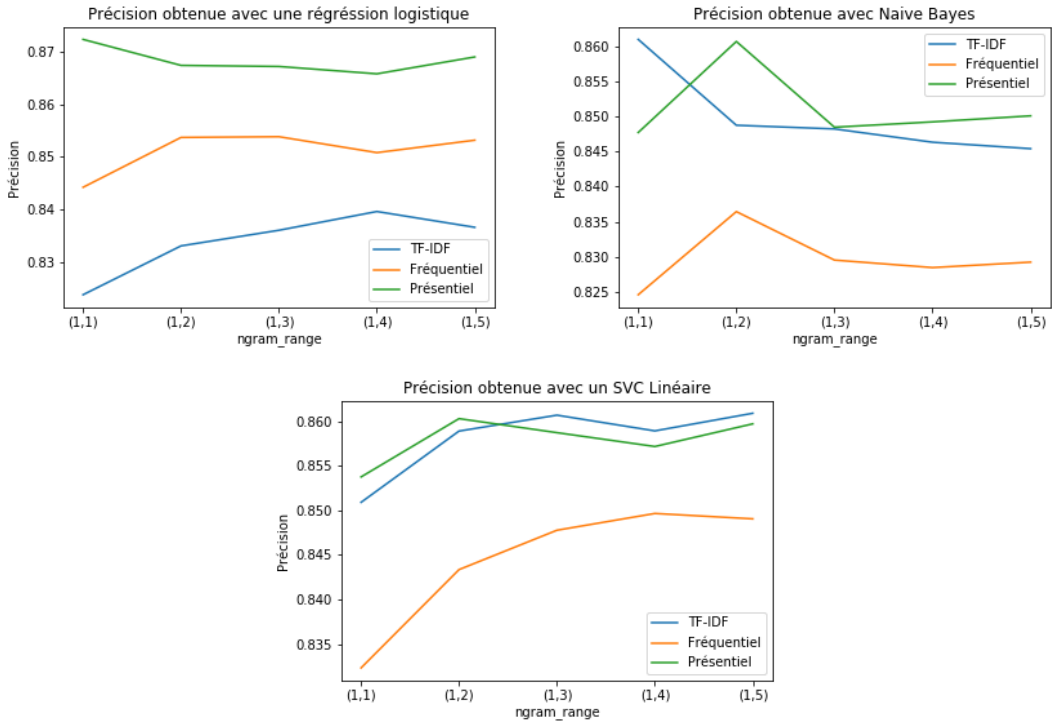

FIGURE 2.5 – Précision obtenue en fonction de *ngram_ range*

On remarque sur la figure 2.5 que considérer les mots comme des bi-grammes donne toujours de meilleurs scores que la représentation sous forme d'uni-grammes. Cela est cohérent. Le contexte d'un mot joue un rôle important dans la classification de sentiments. Les représentations qui semblent donner les meilleurs résultats sont les représentations sous forme de bi-grammes et tri-grammes.
Après un gridsearch sur chaque classifieur en faisant varier tous les paramètres testés plus haut en même temps, on trouve les meilleurs résultats avec une vectorisation *fréquentielle* et un classifieur Naive Bayes multinomial. Le tokenizer utilisé est un tokenizer personnalisé gardant la ponctuation des textes. Les paramètres du vectoriseur fréquentiel sont les suivants :

$min\_df = 0.05$
$max\_df = 0.75$
$ngram\_range = (1, 3)$
$max\_features = 5000$

On rappelle que les données ont été pré-traitées au préalable (stemming, tokenization, stop-words, passage de tous les caractères en minuscule)

Après validation croisée sur 5 folds, la précision moyenne obtenue est de **84.08%**

2.5 Post-traitement des données

Ici aussi nous avons effectué des prédictions par "moyennes de prédictions" : Nous avons entraîné 5 classifieurs sur 5 jeux de données (les données initiales) chacunes sous-échantillonées de façon aléatoire (sous-échantillonage conservant 60% des données et préservant l'équilibre des classes).

Remarque : Le sous-échantillonage n'était pas nécessaire ici car les classes sont déjà équilibrées. Cependant, on peut penser qu'une moyenne de prédictions donnera de meilleurs résultats qu'une prédiction simple... À voir.

Après prédiction du fichier test, on remarque que les classes sont quasiment équilibrées (13300 éléments prédits dans la classe négative, 11600 éléments prédits dans la classe positive).

On peut supposer que la distribution des classes dans le fichier de test suit la même distribution que dans le fichier train (à savoir une distribution parfaitement équilibrée). Ainsi, afin d'augmenter le score, nous avons classé certains éléments de la classe négative dans la classe positive afin d'avoir des classes prédites plus équilibrées.

Les éléments choisis pour le changement de classe sont les éléments pour lesquels la prédiction a été la plus "difficile" d'après notre protocole de "moyenne de prédictions".

Exemple : Un élément ayant été prédit 3 fois dans la classe négative et 2 fois dans la classe positive sera classé dans la classe négative. Cependant, on considère que la prédiction a été "difficile".

À l'inverse, un élément ayant été classé 5 fois dans la classe négative est considéré comme étant "certainement bien classifié" : On ne touche pas aux éléments de ce type.