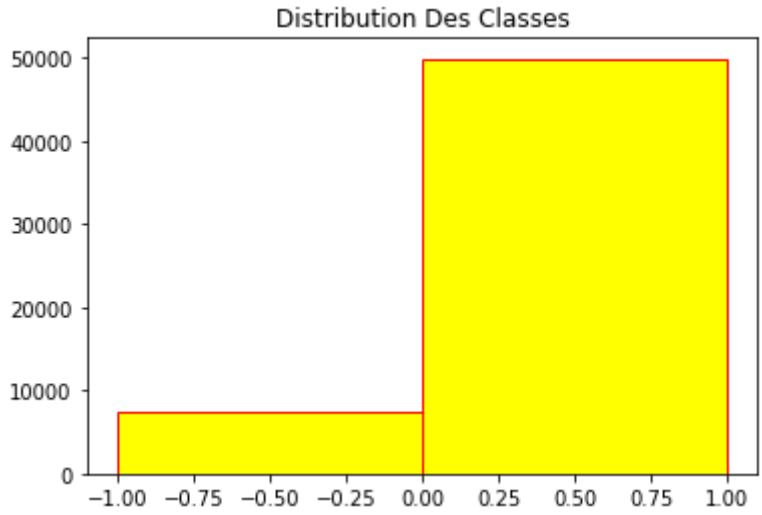
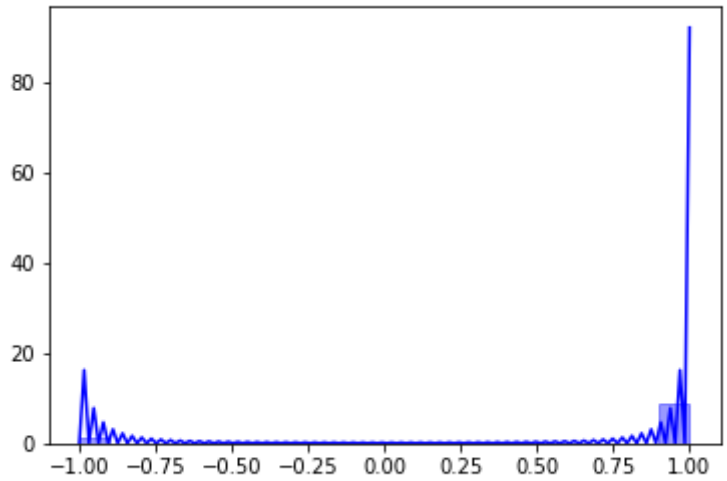
CHIRAC / MITTERAND

Rédigé par : Sahli Oussama

Professeur : Vincent Guigue

Métrique et Rééchantillonage :





Observations :

Les classes sont déséquilibrés. La classe -1 est minoritaire par rapport à la classe 1.

Les classes ne sont pas présenté de façon égale. Ce déséquilibre des classes augmente la difficulté de l'apprentissage par l'algorithme de classification. L'algorithme a peu d'exemples de la classe

minoritaire sur lesquels apprendre. L'algorithme est alors biaisé vers les données de la classe -1, et produit des prédictions moins robustes qu'en l'absence de déséquilibre.

Nous allons dans un premier temps, nous intéresser à la métrique que l'on doit utiliser dans ce genre de situation.

Dans le cas du déséquilibre des classes, l'accuracy peut être trompeuse. Car, on est dans un cas où les données de la classe 1, représente un peu moins de 90 % des données. Donc, si le classifieur prédit que chaque donnée appartient à la classe 1, l'exactitude sera de 90%. Mais ce classifieur est inutile.

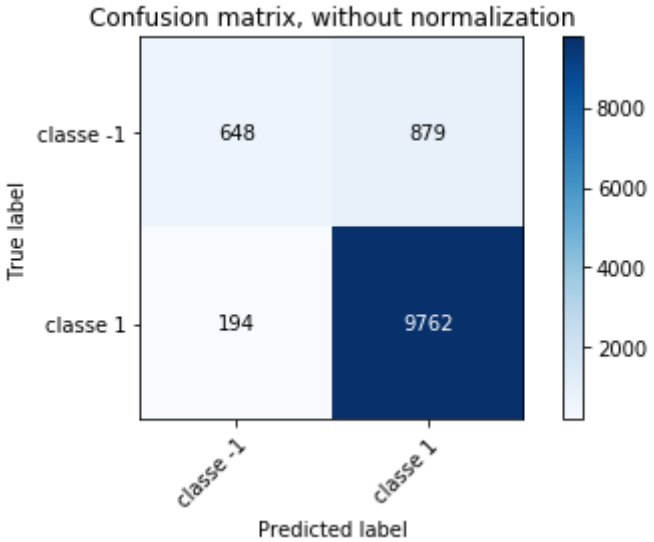
<u>Pour illustrer nos propos, nous allons calculer l'accuracy, et afficher la matrice de confusion :</u>

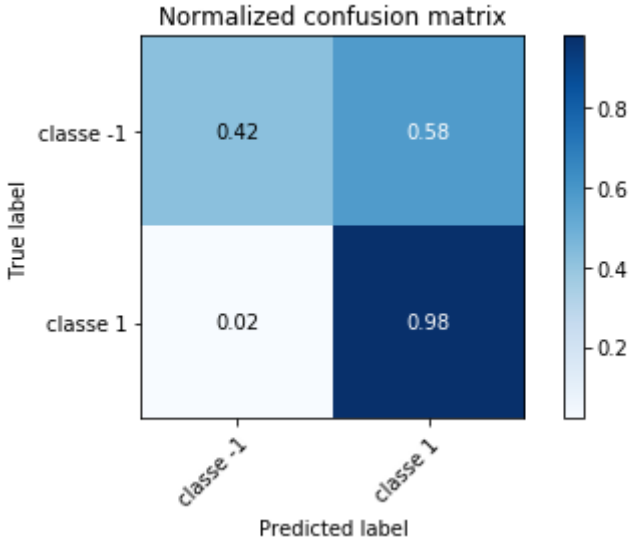
```
X_train, X_test, y_train, y_test = train_test_split(datax, datay
                                   ,test_size=0.2, random_state=1234)

cv = CountVectorizer()
X_train=cv.fit_transform(X_train)
X_test=cv.transform(X_test)

RL=LogisticRegression()
RL.fit(X_train,y_train)
y_pred = RL.predict(X_test)
print("ACCURACY: ",RL.score(X_test,y_test)*100," %")
```

```
ACCURACY:  90.65575198118958  %
```

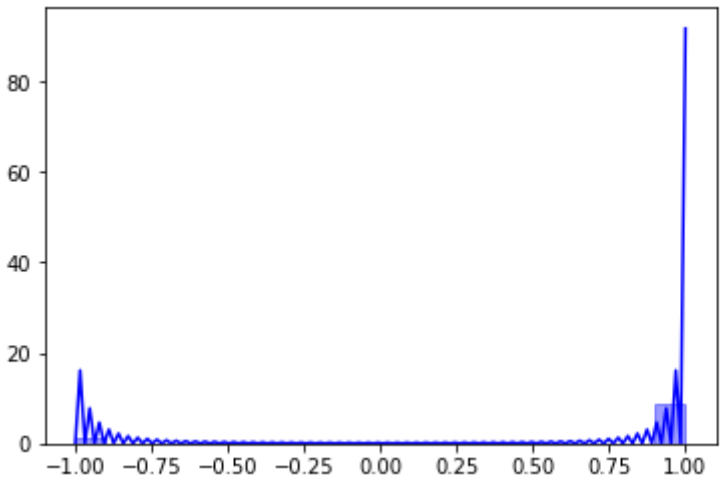




Observations :

En termes d'accuracy, le classifieur LogisticRegression réalise 90,6%. Au premier abord, le modèle semble avoir une bonne performance. Mais si on observe la matrice de confusion, on remarque que le modèle a tendance à toujours prédire la classe1. Sur la matrice de confusion normalisé, on observe que le modèle arrive très bien à prédire la classe 1, car il prédit correctement dans 98% des cas, et se trompe dans 2% des cas. En revanche, en ce qui concerne la classe minoritaire (la classe -1), le modèle prédit correctement dans 42% des cas, et se trompe dans 58% des cas. Ainsi, on constate que le modèle n'est pas efficace pour prédire la classe -1.

Distribution des données dans notre ensemble de données test (y_test) :



Si on utilise un modèle naïf, qui consiste à toujours prédire la classe majoritaire (la classe 1), avec une accuracy de 90%, car sur notre exemple ci-dessus, la classe 1 est présente à 90% dans notre ensemble de test. Alors, le modèle naïf, et le modèle de RegressionLogistique définit précédemment, sont alors presque aussi bon l'un que l'autre. Leur performance en termes d'accuracy n'est pas pertinente dans ce contexte et ne permet pas de réellement évaluer le modèle.

Dans notre cas de déséquilibre des classes, on peut utiliser d'autres métriques qui seront beaucoup plus pertinentes. Plus particulièrement, il faudrait qu'on utilise des métriques qui nous permettent de savoir si le modèle arrive à prédire correctement la classe minoritaire.

Parmi les métriques qu'on pourrait utiliser, il y a le **rappel**, la **précision**, et le score **F1**.

La mesure de Rappel, peut nous être intéressante, car elle nous permettra d'observer la capacité du modèle à prédire correctement **toutes** les données de la classe minoritaire.

La précision, elle aussi est intéressante, car elle nous permettra d'observer la capacité du modèle à ne prédire **que** les données de la classe minoritaire.

Le score F1, est lui aussi intéressant à utiliser si on cherche à avoir à la fois un bon rappel et une bonne prédiction. C'est une moyenne entre le rappel et la précision.

<u>Illustrons nos propos avec un exemple :</u>

```
X_train, X_test, y_train, y_test = train_test_split(datax, datay
                                  ,test_size=0.2, random_state=1234)

cv = CountVectorizer()

X_train=cv.fit_transform(X_train)
X_test=cv.transform(X_test)

RL = LogisticRegression(C=1,penalty='l2')
RL.fit(X_train,y_train)
p = RL.predict(X_test)
print(classification_report(y_test, p))
```

```
              precision    recall  f1-score   support

        -1.0       0.77      0.42      0.55      1527
         1.0       0.92      0.98      0.95      9956

   micro avg       0.91      0.91      0.91     11483
   macro avg       0.84      0.70      0.75     11483
weighted avg       0.90      0.91      0.89     11483
```

<u>Observations :</u>

On observe que le modèle a une faible performance au niveau du rappel et du score F1 pour la classe minoritaire (classe -1). On a un rappel de seulement 42% pour la classe minoritaire, et un score F1 de seulement 55% pour la classe minoritaire. La précision elle aussi n'est pas très bonne pour la classe minoritaire. Car si on compare la précision obtenue sur la classe minoritaire et celle obtenue sur la classe majoritaire, d'un côté on a 77% et de l'autre 92%.

Pour résoudre ce problème de déséquilibrage des classes, on peut tout d'abord rééquilibrer les classes en rééchantillonnant le jeu de données.

Rééquilibrer Les Classes :

Vu que le nombre de document (57 000) dans l'ensemble de données est assez grands, on va plutôt utiliser un sous échantillonnage.

J'ai remarqué lors de mes expériences, que la méthode utilisée pour rééquilibrer les classes, pouvait changer les résultats obtenus. Ainsi j'ai décidé de garder la méthode de rééquilibrage des données qui me donne les meilleurs résultats.

Parmi les méthodes de sous échantillonnage, j'en compte cinq :

Méthode 1 :

(on utilise la méthode Nearmiss (version1) du package imblearn.under_sampling). La version 1 de Nearmiss sélectionne des exemples de la classe majoritaire qui ont la plus petite distance moyenne aux trois exemples les plus proches de la classe minoritaire.

Méthode 2:

On utilise la version 2 de Nearmiss. La version 2 de Nearmiss, sélectionne des exemples de la classe majoritaire qui ont la plus petite distance moyenne aux trois exemples les plus éloignés de la classe minoritaire.

Méthode 3 :

On utilise la version 3 de Nearmiss. La version 3 de Nearmiss, consiste à sélectionner un nombre donné d'exemples de classe majoritaire pour chaque exemple de la classe minoritaire qui est le plus proche.

Méthode 4 :

On utilise la méthode RandomUnderSampler du package imblearn.under_sampling. Elle sous-échantillonne la classe majoritaire en prélevant au hasard des échantillons avec ou sans remplacement.

Méthode 5 :

J'ai construit ma propre méthode de rééquilibrage des classes. Elle sélectionne au hasard (sans remise) des exemples de la classe majoritaire.

Maintenant que j'ai énuméré les méthodes que je vais utiliser, je vais observer les scores obtenus pour chacune de ces méthodes.

En appliquant un sous échantillonnage avec la **méthode 1**, on obtient les résultats suivants :

```
cv = CountVectorizer()
X=cv.fit_transform(datax)

nm = NearMiss(version=1)
X_resampled, y_resampled = nm.fit_resample(X, datay)
print(sorted(Counter(y_resampled).items()))

X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled
                                ,test_size=0.2, random_state=1234)

RL = LogisticRegression()
RL.fit(X_train,y_train)

p = RL.predict(X_test)
print(classification_report(y_test, p))
```

```
[(-1.0, 7523), (1.0, 7523)]
              precision    recall  f1-score   support

        -1.0       0.94      0.85      0.89      1498
         1.0       0.87      0.95      0.90      1512

    accuracy                           0.90      3010
   macro avg       0.90      0.90      0.90      3010
weighted avg       0.90      0.90      0.90      3010
```

En appliquant un sous échantillonnage avec la **méthode 2**, on obtient les résultats suivants :

```
[(-1.0, 7523), (1.0, 7523)]
              precision    recall  f1-score   support

        -1.0       0.91      0.90      0.91      1498
         1.0       0.90      0.91      0.91      1512

    accuracy                           0.91      3010
   macro avg       0.91      0.91      0.91      3010
weighted avg       0.91      0.91      0.91      3010
```

En appliquant un sous échantillonnage avec la **méthode 3**, on obtient les résultats suivants :

```
[(-1.0, 7523), (1.0, 7175)]
              precision    recall  f1-score   support

        -1.0       0.84      0.75      0.79      1497
         1.0       0.77      0.85      0.81      1443

    accuracy                           0.80      2940
   macro avg       0.80      0.80      0.80      2940
weighted avg       0.80      0.80      0.80      2940
```

En appliquant un sous échantillonnage avec la **méthode 4**, on obtient les résultats suivants :

```
[(-1.0, 7523), (1.0, 7523)]
              precision    recall  f1-score   support

        -1.0       0.79      0.76      0.77      1498
         1.0       0.77      0.80      0.78      1512

    accuracy                           0.78      3010
   macro avg       0.78      0.78      0.78      3010
weighted avg       0.78      0.78      0.78      3010
```

En appliquant un sous échantillonnage avec la **méthode 5**, on obtient les résultats suivants :

```
[(-1.0, 7523), (1.0, 7523)]
              precision    recall  f1-score   support

        -1.0       0.78      0.76      0.77      1497
         1.0       0.77      0.79      0.78      1513

    accuracy                           0.77      3010
   macro avg       0.77      0.77      0.77      3010
weighted avg       0.77      0.77      0.77      3010
```

Observations :

En appliquant le sous échantillonnage sur l'ensemble de données, on remarque que les mesures de précision, rappel et score F1 sont meilleurs. Les classes sont équilibré, il n'y a pas plus de classe minoritaire et de classe majoritaire. Le modèle arrive à bien prédire les données de la classe -1 et celle de la classe 1.

De plus on remarque que la méthode de rééquilibrage des classes qui donne les meilleurs résultats est la méthode 2 .

Mais j'ai continué mon questionnement, en me demandant comment agirait mon modèle sur des données inconnus. Ainsi, pour être sûr que la méthode donnant les meilleurs résultats précédemment était la meilleure, j'ai décidé d'appliquer mes tests sur un ensemble de test obtenue à partir de l'ensemble initiale (je vais faire un split sur les 57 000 document de l'ensemble de base, et obtenir un ensemble de test à partir de cet ensemble de 57 000 documents).

En fait, ce que je fais, c'est que tout d'abord j'équilibre les classes, puit j'entraine mon modèle sur des données d'entrainement (obtenue en faisant un split sur les données équilibré), et au lieu de tester mon modèle sur l'ensemble de test (obtenu avec le split sur les données équilibré), je le teste sur un ensemble de test obtenue en faisant un split sur l'ensemble des 57000 documents initiales. Ainsi je ré applique les tests définit précédemment, mais en testant non plus sur l'ensemble de test des données équilibré, mais sur l'ensemble de test obtenue en faisant un split sur l'ensemble des 57000 documents initiales.

Illustrations :

```
cv = CountVectorizer()
X=cv.fit_transform(datax)

nm = NearMiss(version=1)

X_resampled, y_resampled = nm.fit_resample(X, datay)

print(sorted(Counter(y_resampled).items()))

X_train_eq, X_test_eq, y_train_eq, y_test_eq = train_test_split(X_resampled, y_resampled
                                ,test_size=0.2, random_state=1234)

RL = LogisticRegression(max_iter=3000)
RL.fit(X_train_eq,y_train_eq)

X_train_deseq, X_test_deseq, y_train_deseq, y_test_deseq = train_test_split(X, datay
                                ,test_size=0.2, random_state=1234)

p = RL.predict(X_test_deseq)
print(classification_report(y_test_deseq, p))
```

Méthode 1:

```
[(-1.0, 7523), (1.0, 7523)]
              precision    recall  f1-score   support

        -1.0       0.16      0.90      0.28      1527
         1.0       0.95      0.29      0.45      9956

    accuracy                           0.37     11483
   macro avg       0.56      0.60      0.36     11483
weighted avg       0.84      0.37      0.43     11483
```

Méthode 2:

```
[(-1.0, 7523), (1.0, 7523)]
              precision    recall  f1-score   support

        -1.0       0.18      0.97      0.30      1527
         1.0       0.99      0.31      0.47      9956

    accuracy                           0.39     11483
   macro avg       0.58      0.64      0.38     11483
weighted avg       0.88      0.39      0.44     11483
```

Méthode 3:

```
[(-1.0, 7523), (1.0, 7175)]
              precision    recall  f1-score   support

        -1.0       0.29      0.87      0.43      1527
         1.0       0.97      0.67      0.79      9956

    accuracy                           0.69     11483
   macro avg       0.63      0.77      0.61     11483
weighted avg       0.88      0.69      0.74     11483
```

Méthode 4:

```
[(-1.0, 7523), (1.0, 7523)]
              precision    recall  f1-score   support

        -1.0       0.43      0.90      0.58      1527
         1.0       0.98      0.81      0.89      9956

    accuracy                           0.82     11483
   macro avg       0.70      0.86      0.73     11483
weighted avg       0.91      0.82      0.85     11483
```

Méthode 5:

```
[(-1.0, 7523), (1.0, 7523)]
              precision    recall  f1-score   support

        -1.0       0.43      0.89      0.58      1527
         1.0       0.98      0.82      0.89      9956

    accuracy                           0.83     11483
   macro avg       0.70      0.85      0.73     11483
weighted avg       0.91      0.83      0.85     11483
```

Observations :

On peut observer que la meilleure méthode est la méthode aléatoire (méthode 4 et méthode 5). Contrairement à précédemment où la meilleure méthode était la méthode 2.

Ainsi, je préfère garder la méthode 4 pour établir mon rééquilibrage des classes.

Maintenant que l'on a compris comment résoudre le problème du déséquilibrage des classes, on va chercher à optimiser les résultats obtenus. Pour cela, nous allons déterminer le pré traitement de nos données qui nous donnera les meilleurs résultats. De plus, on déterminera les meilleures valeurs de paramètres à utiliser pour nos classifieurs.

Optimisation :

Au début, j'avais pensé à utiliser un Pipeline avec un GridSearchCV. Je voulais appliquer un GridSearchCV sur un transformateur et un classifieur à chaque fois.

Par exemple, pour le CountVectorizer et le Classifieur SVM, on aurait fait ceci :

```
import time
import nltk
start_time = time.time()

sw = set()
sw.update(tuple(nltk.corpus.stopwords.words('french')))

data_X,data_Y=equilibre(datax,datay)

pipeline = Pipeline([
    ('vect', CountVectorizer()),
    ('clf', SVC()),
])


parameters = {
    'vect__max_df': (0.5, 0.6, 0.7, 0.8, 0.9, 1),
    'vect__max_features': (None, 5000, 10000, 20000, 30000, 40000, 50000, 60000, 700000,100000),
    'vect__ngram_range': ((1, 1), (1, 2),(2,2)),
    'vect__min_df': (0.01, 0.05, 0.1, 0.15, 0.2, 0.25 ),
    'vect__lowercase': (True,False),
    'vect__analyzer' : ('word','char_wb'),
    'vect__stop_words': (None,sw),
    'vect__tokenizer': (None,my_tokenizer),

    'clf__C': (0.1, 1, 10, 100, 1000),
    'clf__gamma': (1, 0.1, 0.01, 0.001, 0.0001),
    'clf__max_iter': (1000, 2000, 3000),
    'clf__kernel': ('rbf','linear','poly'),
}

G = GridSearchCV(pipeline, parameters, n_jobs=-1, verbose=1,scoring='f1',cv=5)
X_train, X_test, y_train, y_test = train_test_split(data_X, data_Y
                                ,test_size=0.2, random_state=1234)
G.fit(X_train, y_train)
print(G.sore(X_test,y_test))
# Affichage du temps d execution
print("Temps d execution : %s secondes ---" % (time.time() - start_time))
```

Le code ci-dessus a mis énormément de temps à s'exécuter. Ainsi, j'ai compris que procéder de cette manière était une mauvaise idée. De plus, si pour mes expériences je dispose de trois transformateurs (CountVectorizer (), TFIDFVectorizer (), Countvectorizer(binary=True)), de trois classifieurs (SVM, LogisticRegression, Naïve Bayes) ou plus (SGD Classifier, Knn, Tree, Perceptron), je vais me retrouver avec beaucoup de combinaison à tester. De plus, si pour chaque combinaison j'ai en moyenne onze paramètres à faire varier, le temps de calculs serait vraiment énorme.

Ainsi, ma stratégie d'optimisation a été de d'abord trouver le meilleur transformateur pour mes données équilibrés. J'ai décidé de vérifier l'importance de chaque paramètre du transformateur. Et puis, dans un second temps, après avoir obtenue le meilleur transformateur et le classifieur qui lui

correspond ; je réalise un Grid Search sur le classifieur pour trouver les paramètres qui donnent les meilleurs résultats.

Meilleur Transformateur :

Rappel : On utilise nos données équilibrées pour faire nos expériences.

Prétraitement des données :

On transforme notre corpus de texte en un vecteur de termes**.** On peut prétraiter nos données de textes avant de générer leur représentation vectorielle. Notre objectif sera de calculer le **prétraitement de** nos données qui donne les meilleurs résultats.

Prétraiter notre texte avant de créer une matrice de termes clairsemée peut aider à réduire le bruit et à améliorer les problèmes de rareté.

Model Bag of Word “Sacs de mots”:

On transforme notre corpus de texte en en une matrice clairsemée. Chaque colonne de la matrice représente un mot unique dans le vocabulaire, tandis que chaque ligne représente le document dans notre jeu de données. Les valeurs dans chaque cellule sont le nombre de mots. Le nombre de certains mots peut être égal à 0 si le mot n'apparaît pas dans le document correspondant.

L’approche des sacs de mots, consiste à traiter chacun documents comme un simple ensemble de mots individuels. La grammaire, l’ordre des mots, la structure de la phrase sont ignoré. Chaque mot apparaissant dans un document est considéré comme un éventuel mot clé important de ce document. Chaque mot est représenté par un token. Et dans chaque document le mot est représenté par sa fréquence d’apparition dans le document.

On convertit le texte brut en une représentation vectorielle numérique des mots et des n-grammes . On va utiliser cette représentation pour réaliser nos d'apprentissage automatique.

Pour chacun des classifieurs (SVM, Naïves Bayes, LogisticRegression) , on a mené un ensemble d’expérience.

On réalise un calcule des scores en utilisant la cross validation :

Le score calculé sera le score F1 :

	SVM	Logistic Regression	Naive Bayes
données brutes	76.12500392285122	78.47386752909037	78.20617859411205
lowercase	74.88264313046886	77.54335878736637	76.78333179623354
stop_words	75.36624544380976	77.67092156528601	**78.5598070060564**
tokenizer_modifié	**76.84651992807994**	**79.22043278234334**	**79.334627355268**
stemming	76.09562754317652	78.31974665890317	78.18833170159988
Niveau des mots - bigrammes uniquement	74.58589289428231	76.14253505418726	76.43308194581834
Niveau des mots - unigrammes et bigrammes	**77.60128732551246**	**79.00237399819441**	76.63299979852759
Niveau du charactère - bigramme uniquement	74.81094735355653	74.9486041199789	72.92612938532805

Observations :

On observe que pour le classifieur SVM et le classifieur LogisticRegression, le tokenizer modifié et l'utilisation des unigrammes et bigrammes au niveau des mots, permettent d'augmenter le score. Pour le classifieur Naïve Bayes, l'utilisation de stop words et du tokenizer modifié permettent d'augmenter le score.

Maintenant nous allons faire varier la valeur de min_df et max_df avec un Grid search sur chacun des classifieurs.

Par exemple, pour le SVM, nous allons procéder comme cela :

```
X_resampled, y_resampled = equilibre(datax, datay)

#SVM

pipeline = Pipeline([
    ('vect', CountVectorizer(lowercase=False)),
    ('clf', LinearSVC(max_iter=3000)),
])

parameters = {
    'vect__max_df': (0.5, 0.6, 0.7, 0.8, 0.9, 1),
    #'vect__min_df': (0.01, 0.05, 0.1, 0.15, 0.2, 0.25 ),
    }

clf = GridSearchCV(pipeline, parameters, n_jobs=-1, verbose=1,scoring='f1',cv=5)
clf.fit(X_resampled,y_resampled)
print("Best parameter (CV score=%0.3f):" % clf.best_score_)
print(clf.best_params_)
```

Variations du paramètre max_df :

SVM :

```
Best parameter (CV score=0.749):
{'vect__max_df': 0.7}
```

LogisticRegression :

```
Best parameter (CV score=0.779):
{'vect__max_df': 0.7}
```

Naïve Bayes :

```
Best parameter (CV score=0.782):
{'vect__max_df': 0.7}
```

Variations du paramètre min_df :

SVM :

```
Best parameter (CV score=0.731):
{'vect__min_df': 0.01}
```

LogisticRegression :

```
Best parameter (CV score=0.723):
{'vect__min_df': 0.01}
```

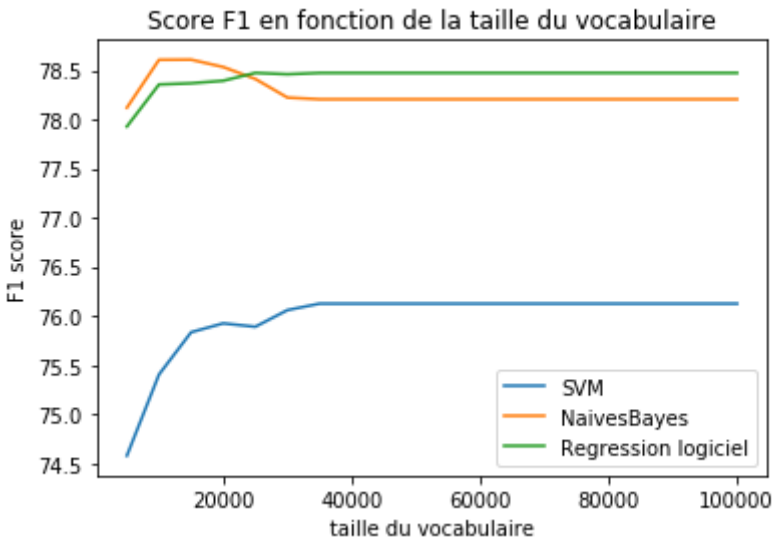
Naïve Bayes:

```
Best parameter (CV score=0.710):
{'vect__min_df': 0.01}
```

Observations :

On ne remarque pas d'améliorations.

Variations de Max_Features (taille du vocabulaires) :



Observations :

On observe que pour le classifieur Naïve bayes, les résultats sont meilleurs lorsque max_features=15000. Pour le classifieur LogisticRegression, les résultats sont meilleurs lorsque max_features=25000. Pour le classifieur SVM, les résultats sont meilleurs lorsque max_features=35000.

Maintenant, pour chaque classifieur, on va utiliser les informations obtenues précédemment, afin de maximiser le score obtenu en utilisant le transformateur bags of words.

SVM :

```
cv = CountVectorizer(lowercase=False,ngram_range=(1,2),tokenizer=my_tokenizer,max_features=35000)
```

Score :

```
76.69934262125574
```

LogisticRegression :

```
cv = CountVectorizer(lowercase=False,ngram_range=(1,2),tokenizer=my_tokenizer,max_features=25000)
```

Score :

```
79.41960807473392
```

Naïve bayes :

```
cv = CountVectorizer(lowercase=False,stop_words=sw,tokenizer=my_tokenizer,max_features=15000)
```

Score :

```
79.39477414496847
```

Observations :

Le meilleur score obtenue, en utilisant le transformateur bags of words, est 79.419 %. Ce score est obtenu en utilisant le classifieur LogisticRegression.

Maintenant, on va appliquer un Grid Search CV sur chacun des classifieurs (svm, LogisticRegression, Naïve Bayes), en utilisant le transformateur avec lequel il obtient les meilleures prédictions.

Grid Search :

SVM :

```
cv = CountVectorizer(lowercase=False,ngram_range=(1,2))
X=cv.fit_transform(datax)
nm = RandomUnderSampler(random_state=42)
X_resampled, y_resampled = nm.fit_resample(X, datay)

parameters = [{'penalty': ['l1','l2'],
               'C': np.arange(0.01,100,10)},
              ]

clf = GridSearchCV( svm, parameters,refit = True,cv=5)
clf.fit(X_resampled,y_resampled)
print("Best parameter (CV score=%0.3f):" % clf.best_score_)
print(clf.best_params_)


Best parameter (CV score=0.781):
{'C': 0.01, 'penalty': 'l2'}
```

LogisticRegression :

```
Best parameter (CV score=0.791):
{'C': 1.0, 'penalty': 'l2'}
```

Naïve Bayes :

```
Best parameter (CV score=0.793):
{'alpha': 1}
```

Observations :

Le meilleur score obtenue est 79,4% , en utilisant le classifieur LogisticRegression (avec paramètre par défaut).

Une autre approche, l'approche présentiel :

On ignore les comptes, et on utilise des valeurs binaires. On peut utiliser simplement la présence ou l'absence d'un terme au lieu des chiffres bruts. On cherche à savoir si la fréquence d'occurrence est insignifiante.

On remplace les fréquences des mots dans le document par des 0 et des 1. Dans chaque document, le mot est représenté par un 1 si le mot est dans le document, 0 sinon.

On utilise le même raisonnement que décrit précédemment pour chercher le prétraitement des données qui donnera le meilleur score.

Résultats des expériences mené en utilisant l'approche présentiel :

On réalise un calcule des scores en utilisant la cross validation :

Le score calculé sera le score F1 :

	SVM	Logistic Regression	Naive Bayes
données brutes	76.36799345413156	78.80981126951407	78.83840470353081
lowercase	75.13863252738162	77.54103978927519	77.4791779335931
stop_words	75.34474540562216	77.71360006449972	78.56935697309635
tokenizer_modifié	**76.72394721667641**	**79.49470069466736**	**79.76399443035402**
stemming	76.26346350430973	78.75184662337716	78.76999474830045
Niveau des mots - bigrammes uniquement	74.59425040438352	76.19352330646885	76.53419947817477
Niveau des mots - unigrammes et bigrammes	**77.92207203757823**	**79.27777060081901**	77.42846210747088
Niveau du charactère - bigramme uniquement	73.92390110952762	74.02773909686108	73.31506967453676

Observations :

On obtient le meilleur score, en utilisant le classifieur naïve Bayes avec le tokenizer modifié. Le score est de 79.76%.

Variations du paramètre min_df (avec un Grid search) pour chacun des classifieur :

SVM :

```
Best parameter (CV score=0.729):
{'vect__min_df': 0.01}
```

LogisticRegression :

```
Best parameter (CV score=0.724):
{'vect__min_df': 0.01}
```

Naïve Bayes :

```
Best parameter (CV score=0.712):
{'vect__min_df': 0.01}
```

Variations du paramètre max_df (avec un Grid search) pour chacun des classifieur :

SVM :

```
Best parameter (CV score=0.756):
{'vect__max_df': 0.7}
```

LogisticRegression :

```
Best parameter (CV score=0.781):
{'vect__max_df': 0.7}
```

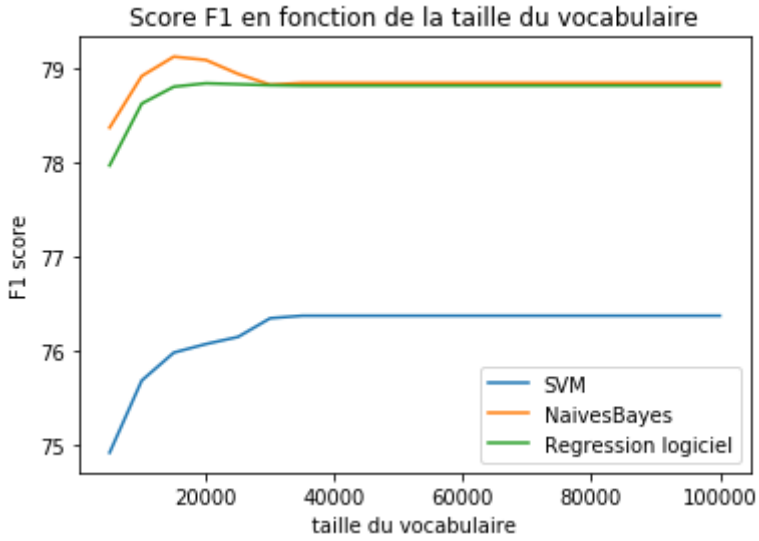
Naïve Bayes :

```
Best parameter (CV score=0.787):
{'vect__max_df': 0.5}
```

Observations :

On ne remarque pas d'améliorations.

Variations de Max_Features (taille du vocabulaires) :



Observations :

On observe que le meilleur score obtenue pour le classifieur svm est atteint lorsque max_features=35000. Pour le classifieur LogisticRegression, le meilleur score est atteint lorsque max_features=20000. Pour le classifieur Naives Bayes, le meilleur score est atteint lorsque max_features=15000.

On a obtenu les meilleurs scores avec le classifieur Naïve Bayes. Je vais utiliser les informations ci-dessus pour améliorer les performances du Naïve Bayes.

```
cv = CountVectorizer(binary=True,lowercase=False,tokenizer=my_tokenizer,max_features=15000)
```

Score :

```
79.98145525861922
```

Observations :

On observe qu'en limitant la taille de max_features à 15000, on améliore le score : 79.98%.

Maintenant je vais appliquer un Grid Search sur le Naïve Bayes pour améliorer les résultats :

```
Best parameter (CV score=0.800):
{'alpha': 1}
```

TF-IDF :

On convertit la collection de documents bruts en une matrice de fonctionnalités TF-IDF. On calcule les scores tf-idf d'un ensemble de documents.

On utilise le même raisonnement que décrit précédemment pour chercher le prétraitement des données qui donnera le meilleur score.

Résultats des expériences mené en utilisant l'approche TF-IDF :

	SVM	Logistic Regression	Naive Bayes
données brutes	78.57642348110898	78.79292037629874	77.99120828862833
lowercase	77.33325238540118	77.9557369656902	76.61231811613641
stop_words	77.48769935271767	78.2491471625686	**78.1874985197027**
tokenizer_modifié	**79.34950730665035**	**79.18824079836384**	77.99518904530636
stemming	**79.32726947098871**	**79.2292439530577**	77.97566165790656
Niveau des mots - bigrammes uniquement	76.88509250004094	75.91065417189368	75.86231286760274
Niveau des mots - unigrammes et bigrammes	**79.92384885694779**	78.46201773316176	76.84225687694908
Niveau du charactère - bigramme uniquement	74.1604913708705	73.69568661148874	72.82960055447288

Observations :

On obtient le meilleur score, en utilisant le classifieur SVM avec le tokenizer modifié, le stemming et l'utilisation d'unigrammes et de bigrammes au niveau des mots. Le score maximum est de 79.92%.

Variations du paramètre min_df (avec un Grid search) pour chacun des classifieur :

SVM :

```
Best parameter (CV score=0.720):
{'vect__min_df': 0.01}
```

LogisticRegression :

```
Best parameter (CV score=0.726):
{'vect__min_df': 0.01}
```

Naïve Bayes :

```
Best parameter (CV score=0.705):
{'vect__min_df': 0.01}
```

Variations du paramètre max_df (avec un Grid search) pour chacun des classifieur :

SVM :

```
Best parameter (CV score=0.773):
{'vect__max_df': 0.7}
```

LogisticRegression :

```
Best parameter (CV score=0.783):
{'vect__max_df': 0.5}
```

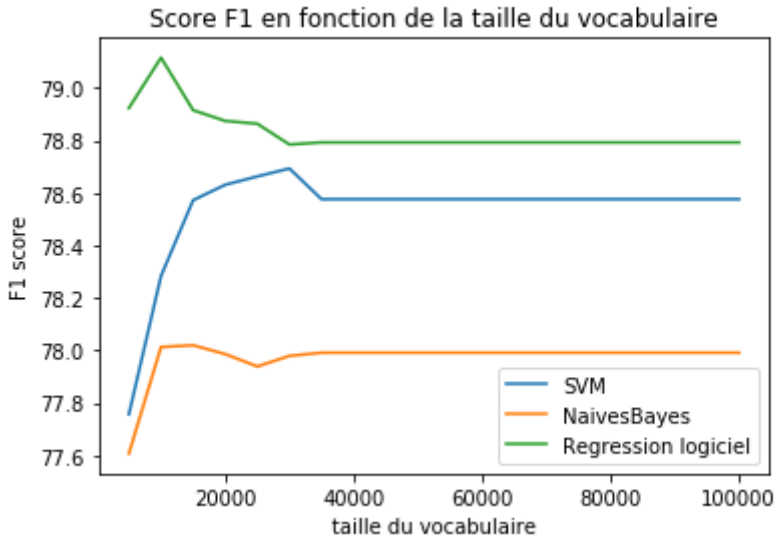
Naïve Bayes :

```
Best parameter (CV score=0.781):
{'vect__max_df': 0.5}
```

Observations :

On ne remarque pas d'améliorations.

Variations de Max_Features (taille du vocabulaires) :



Observations :

On observe que le meilleur score obtenue pour le classifieur svm est atteint lorsque max_features=30000. Pour le classifieur LogisticRegression, le meilleur score est atteint lorsque max_features=10000. Pour le classifieur Naives Bayes, le meilleur score est atteint lorsque max_features=15000.

On a obtenu les meilleurs scores avec le classifieur SVM. Je vais utiliser les informations ci-dessus pour améliorer les performances du SVM.

```
cv = vectorizer = TfidfVectorizer(lowercase=False,ngram_range=(1,2),tokenizer=my_tokenizer)
```

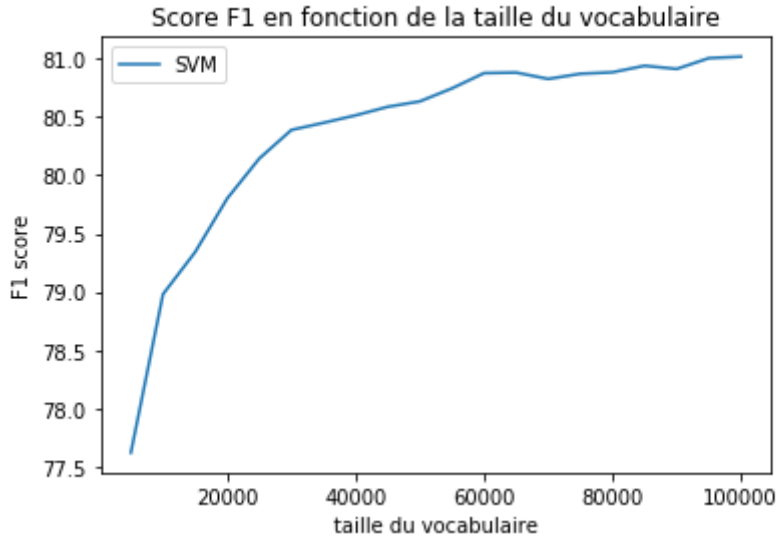
Score :

```
80.853769852148
```

Mais si je limite max_features à 30 000, j'obtiens :

```
80.07708629901039
```

On remarque que le score a diminuer, je vais donc conserver un maximum de features. Je vais observer quel est le meilleur nombre de features à garder, lorsque l'on applique une lemmatisation sur les données :



Observations :

Il est plus intéressant de garder un maximum de features.

Maintenant je vais appliquer un Grid Search sur le SVM pour améliorer les résultats :

```
Best parameter (CV score=0.802):
{'C': 1}
```

Conclusion (Partie Chirac Mitterrand):

Après tous nos tests, on en arrive à la conclusion que le meilleur transformateur et, le meilleur classifieur, qu'il faut utiliser sont :

Le
TFIDFVectorizer

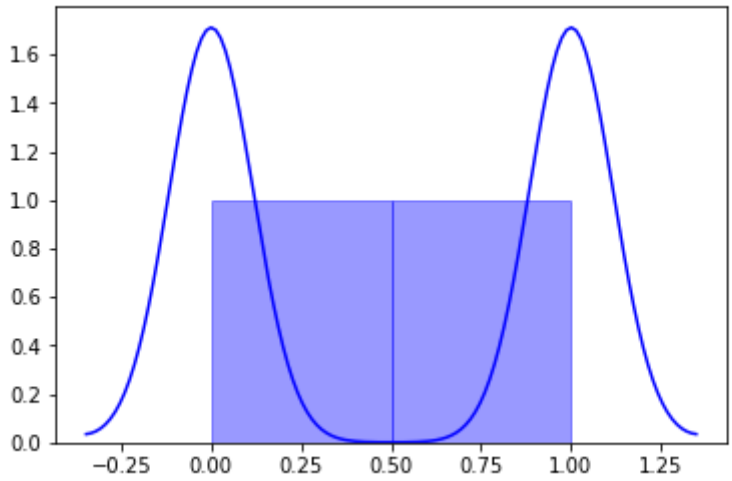
```
cv = vectorizer = TfidfVectorizer(lowercase=False,ngram_range=(1,2),tokenizer=my_tokenizer)
```

Associé avec le classifieur SVM .

Qui donne un score de : 80.85%.

Analyse de Sentiment

Distributions des classes de l'échantillon :



Les classes sont équilibrées. On peut donc utiliser le taux de bonne classification comme indicateur de performance de notre algorithme.

Model Bag of Word "Sacs de mots":

	SVM	Logistic Regression	Naive Bayes
données brutes	83.3	84.45	81.45
lowercase	83.25000000000001	84.45	81.45
stop_words	83.15	84.2	81.3
tokenizer_modifié	83.75	83.75	81.2
stemming	81.64999999999999	83.45	81.60000000000001
Niveau des mots - bigrammes uniquement	81,35	81.00000000000001	**84.3**
Niveau des mots - unigrammes et bigrammes	**85**	trop long	trop long
Niveau du charactère - bigramme uniquement	68.89999999999999	70.94999999999999	69.0

Observations :

On obtient le meilleur score, en utilisant le classifieur SVM. L'utilisation d'unigrammes et de bigrammes au niveau des mots, permet d'avoir un score de 85%.

Variations du paramètre min_df (avec un Grid search) pour chacun des classifieur :

SVM :

```
Best parameter (CV score=0.822):
{'vect__min_df': 0.01}
```

LogisticRegression :

```
Best parameter (CV score=0.832):
{'vect__min_df': 0.01}
```

Naïve Bayes :

```
Best parameter (CV score=0.821):
{'vect__min_df': 0.01}
```

Variations du paramètre max_df (avec un Grid search) pour chacun des classifieur :

SVM :

```
Best parameter (CV score=0.833):
{'vect__max_df': 0.6}
```

LogisticRegression :

```
Best parameter (CV score=0.847):
{'vect__max_df': 0.7}
```

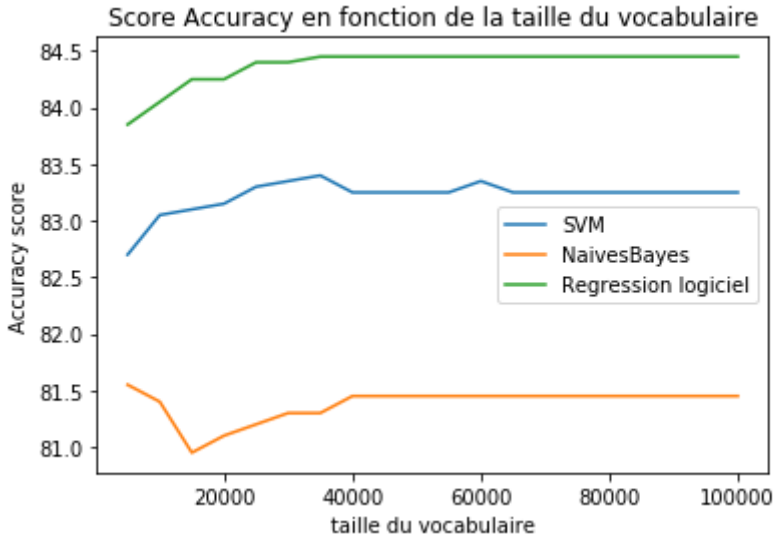
Naïve Bayes :

```
Best parameter (CV score=0.812):
{'vect__max_df': 0.9}
```

Observations :

On ne remarque pas d'améliorations.

Variations de Max_Features (taille du vocabulaires) :



Observations :

On observe que le meilleur score obtenue pour le classifieur svm est atteint lorsque max_features=35000. Pour le classifieur LogisticRegression, le meilleur score est atteint lorsque max_features=35000. Pour le classifieur Naives Bayes, le meilleur score est atteint lorsque max_features=5000.

On a obtenu les meilleurs scores avec le classifieur svm. Je vais utiliser les informations ci-dessus pour améliorer les performances du SVM.

Score :

```
SVM ACCURACY: 84.35000000000001
```

On va donc ne pas mettre de limite sur max_features.

Maintenant je vais appliquer un Grid Search sur le svm pour améliorer les résultats :

```
Best parameter (CV score=0.851):
{'C': 10}
```

Présentiel :

Observations :

	SVM	Logistic Regression	Naive Bayes
données brutes	85.30000000000001	86.7	82.95
lowercase	85.30000000000001	86.8	82.96
stop_words	85.0	86.65	83.05
tokenizer_modifié	85.39999999999999	**86.95**	83.0
stemming	83.50000000000001	85.25	82.10000000000001
Niveau des mots - bigrammes uniquement	83.15	82.85	85.15
Niveau des mots - unigrammes et bigrammes	**87.1**	**87.25**	85.45
Niveau du charactère - bigramme uniquement	64.64999999999999	67.35000000000001	68.5

On obtient le meilleur score, en utilisant le classifieur LogisticRegression. En appliquant le tokenizer modifié, ou en utilisant des unigrammes et bigrammes, le score atteint 87.25%

Variations du paramètre min_df (avec un Grid search) pour chacun des classifieur :

SVM :

```
Best parameter (CV score=0.837):
{'vect__min_df': 0.01}
```

LogisticRegression :

```
Best parameter (CV score=0.856):
{'vect__min_df': 0.01}
```

Naïve Bayes :

```
Best parameter (CV score=0.837):
{'vect__min_df': 0.01}
```

Variations du paramètre max_df (avec un Grid search) pour chacun des classifieur :

SVM :

```
Best parameter (CV score=0.853):
{'vect__max_df': 0.6}
```

LogisticRegression :

```
Best parameter (CV score=0.867):
{'vect__max_df': 0.5}
```

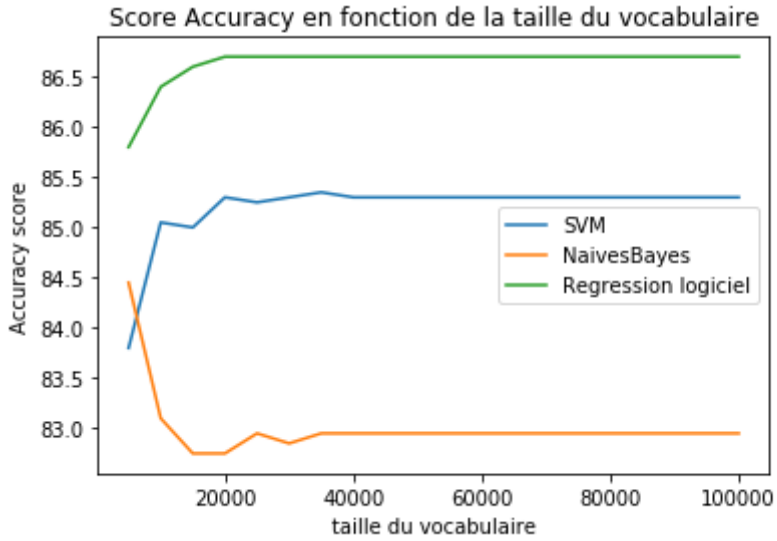
Naïve Bayes :

```
Best parameter (CV score=0.832):
{'vect__max_df': 0.5}
```

Observations :

On ne remarque aucune amélioration.

Variations de Max_Features (taille du vocabulaires) :



Observations :

On observe que le meilleur score obtenue pour le classifieur svm est atteint lorsque max_features=35000. Pour le classifieur LogisticRegression, le meilleur score est atteint lorsque max_features=20000. Pour le classifieur Naives Bayes, le meilleur score est atteint lorsque max_features=5000.

On a obtenu les meilleurs scores avec le classifieur LogisticRegression. Je vais utiliser les informations ci-dessus pour améliorer les performances du LogisticRegression.

Score :

```
RL ACCURACY: 87.25
```

Maintenant je vais appliquer un Grid Search sur le LogisticRegression pour améliorer les résultats :

```
Best parameter (CV score=0.875):
{'C': 0.001}
```

Ainsi, pour le mode présentiel, il est préférable d'utiliser le transformateur :

```
CountVectorizer(binary=True, lowercase=False,ngram_range=(1,2))
```

, et le classifieur LogisticRegression avec C=0.001.

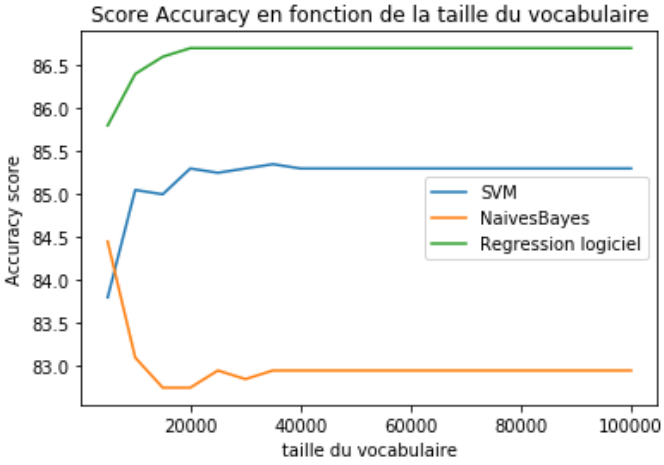
Tf-Idf :

	SVM	Logistic Regression	Naive Bayes
données brutes	85.60000000000001	82.25	81.4
lowercase	85.60000000000001	82.25	81.4
stop_words	**85.80000000000003**	82.29999999999998	81.14999999999999
tokenizer_modifié	**85.7**	80.19999999999999	80.25
stemming	85.45000000000002	83.24999999999999	80.89999999999999
Niveau des mots - bigrammes uniquement	84.44999999999997	82.99999999999999	83.39999999999999
Niveau des mots - unigrammes et bigrammes	84.65	81.0	83.50000000000001
Niveau du charactère - bigramme uniquement	72.95	69.04999999999998	66.7

Observations :

On obtient le meilleur score, en utilisant le classifieur SVM. En retirant les stop words on obtient un score de 85.8 %.

Variations de Max_Features (taille du vocabulaires) :



Observations :

On observe que le meilleur score obtenue pour le classifieur svm est atteint lorsque max_features=35000. Pour le classifieur LogisticRegression, le meilleur score est atteint lorsque max_features=20000. Pour le classifieur Naives Bayes, le meilleur score est atteint lorsque max_features=5000.

On a obtenu les meilleurs scores avec le classifieur SVM Je vais utiliser les informations ci-dessus pour améliorer les performances du SVM.

Score :

```
SVM ACCURACY: 85.80000000000003
```

(en enlevant seulement les stops words).

Maintenant je vais appliquer un Grid Search sur le SVM Pour améliorer les résultats :

```
Best parameter (CV score=0.866):
{'C': 0.001}
```

Conclusion (Partie Analyse de sentiment):

Après tous nos tests, on en arrive à la conclusion que le meilleur transformateur et, le meilleur classifieur, qu'il faut utiliser sont :

Le TFIDFVectorizer :

```
CountVectorizer(binary=True, lowercase=False,ngram_range=(1,2))
```

Et le classifieur LogisticRegression avec C=0.001. Les deux associées nous donnent un score de 87%.