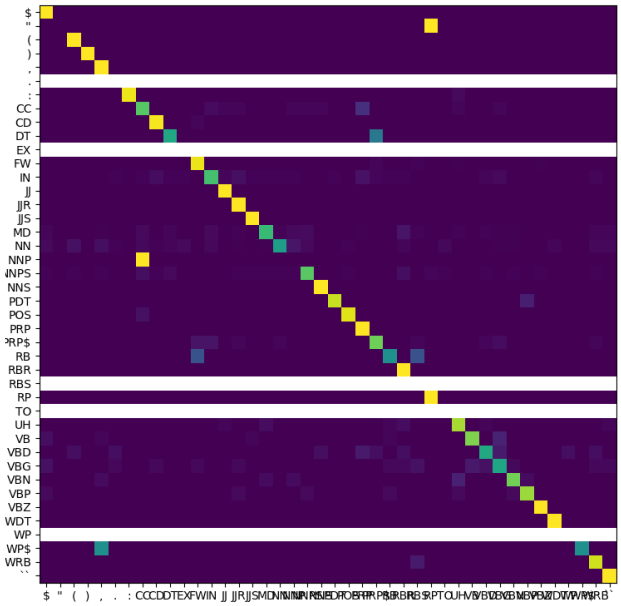
BEROUKHIM Keyvan
BUTON Nicolas

Compte rendu TME
Traitement Automatique du Langage

1) POS TAG

La tâche consiste à labelliser chacun des mots des documents.
L'approche par **dictionnaire** consiste à apprendre la clé associée à chaque mot. Sur le jeu de test, on obtient une performance de **75 %**.
Si un mot du jeu de test n'apparaît pas dans le dictionnaire appris, la meilleure heuristique consiste à renvoyer la **classe majoritaire**. Avec cette amélioration, on obtient une performance de **80 %**.
Convertir tous les caractères en minuscule ne change pas significativement les résultats.
Stemmer les mots fait perdre de l'information importante (« ly », « ing », …) les performances baissent à 70 % environ.

L'approche par **Modèle de Markov Caché** représente les classes par les états cachés et les mots par les observations. Après avoir entraîné le modèle, on détermine les labels par maximum de vraisemblance grâce à l'algorithme de Viterbi.



Matrice de confusion

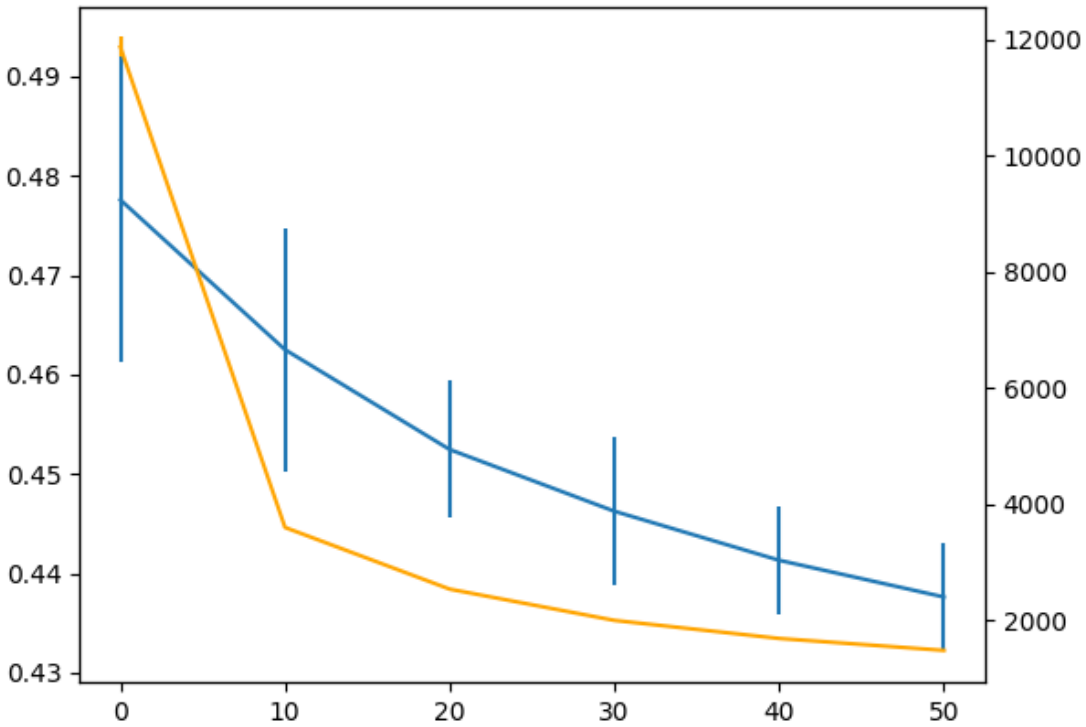
Les performances mesurées sont de **81 %** l'écart de performances avec le modèle précédent n'est pas significatif. La baseline fonctionne bien car les documents sont tous issus du même corpus.

L'approche par **Conditional Random Field** détermine les labels par maximum à posteriori. En se servant d'un modèle déjà entraîné, on obtient des meilleures de performances de l'ordre de 90 %.

2) Classification d'**auteurs** et de **sentiments** (binaires).

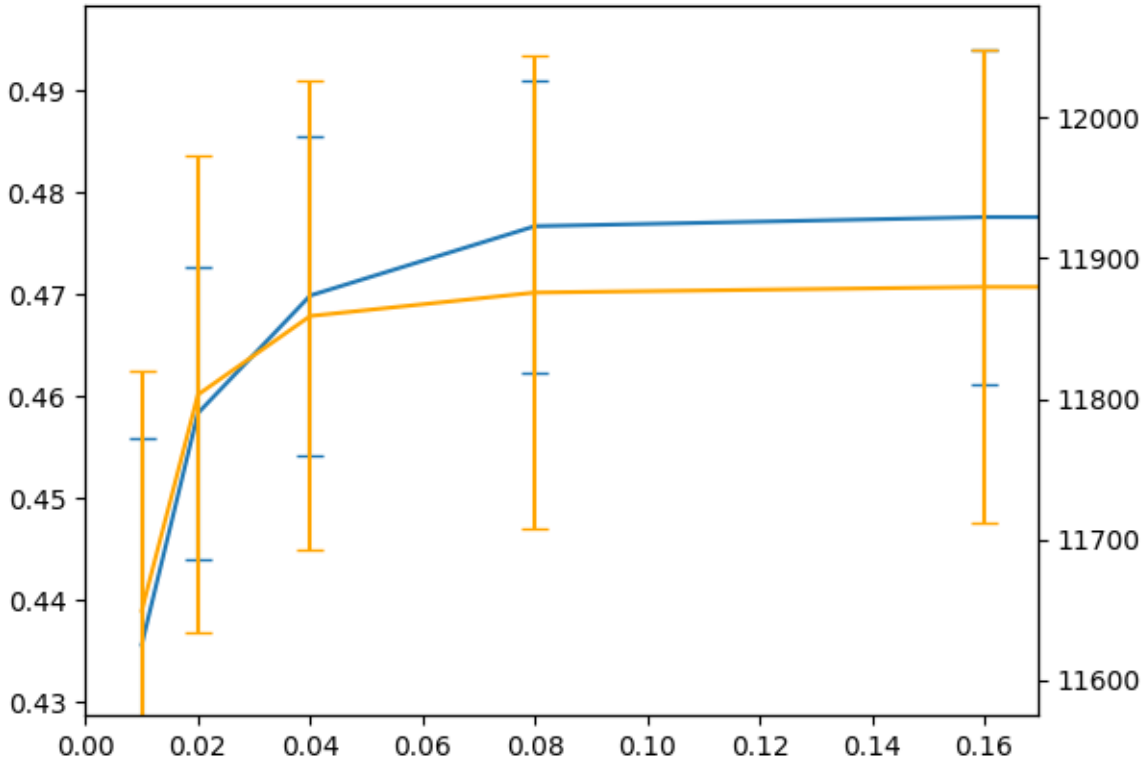
Les tâches consistent à attribuer des documents à un auteur ou à déterminer si le contenu est positif ou négatif.
Pour représenter les documents on utilise la technique du **bag of words**. Un document est représenté par un vecteur contenant à l'indice i le nombre d'occurrences dans le document du ième mot de l'ensemble des mots du vocabulaire.
Notre modèle est un **SVM linéaire**. Les performances de notre modèle sont mesurées par **cross-validation**. Les performances dépendent de la valeur prise par l'**hyper-paramètre** « C » du SVM ainsi que des différents **pre-processings** effectués sur le texte. Les paramètres n'étant pas forcément indépendants, on trouve leurs valeurs optimales par **grid search**. Les classes n'étant pas équilibrées dans le jeu d'apprentissage, on se sert de **stratified** cross validation afin que les individus de chaque classe soient **bien répartis** dans chaque fold.
Certains pré-traitements ont été appliqués systématiquement, ils consistent à enlever la **ponctuation** et les **stop-words**, et à **stemmer** les mots.
Considérer les mots un par un enlève de l'information, les **n-gram** considèrent des groupes de n mots consécutifs. Le nombre de dimension augmente très rapidement avec la taille des n-gram, ce qui explique pourquoi les performances mesurées sont presque toujours meilleures pour de simples bag of words.

Sur les graphiques suivants, la courbe en bleu représente le **score F1** mesuré par cross-validation. Si la **taille du dictionnaire** dépend de la variable, la courbe orange représente la taille moyenne du vocabulaire (et son échelle est à droite). La valeur des autres paramètres étant fixés à peu près à leur optimum. Les barres d'erreurs correspondent à deux fois l'écart type des performances mesurées sur 3 fold.



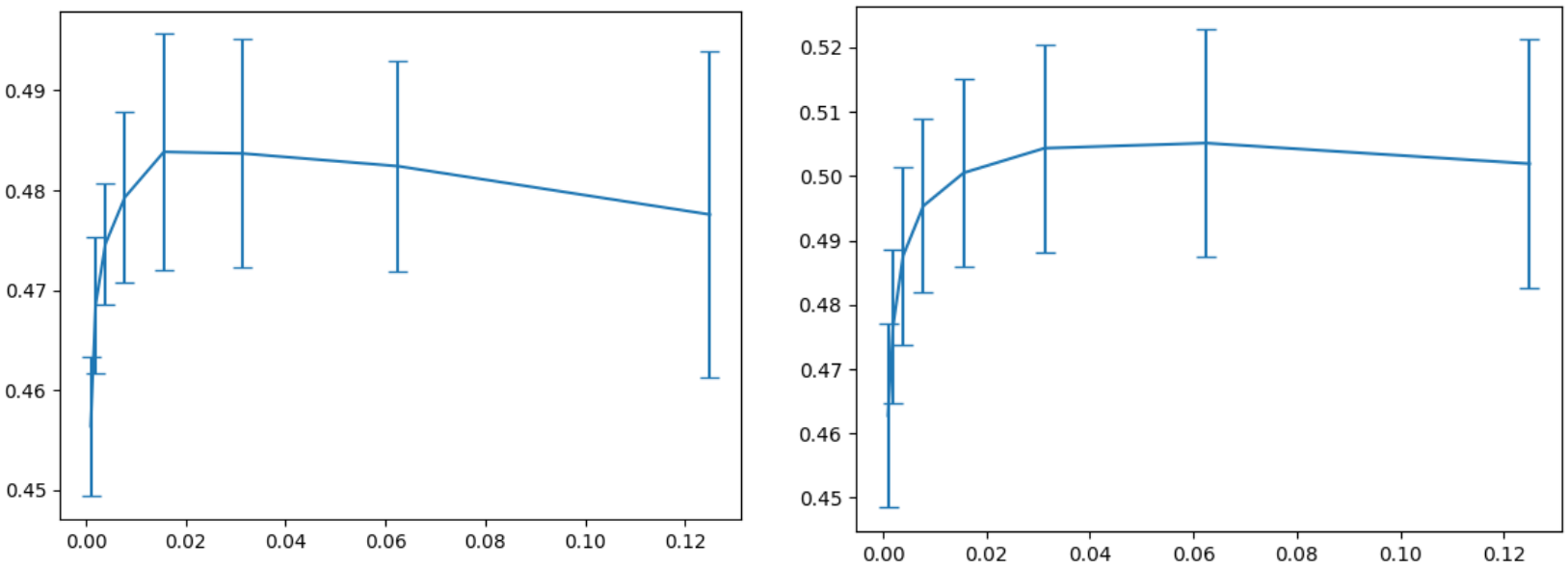
Performances en fonction du nombre d'occurrences min_df

Enlever les mots n’apparaissant dans pas assez de documents n’est pas efficace. L’optimal est de ne pas appliquer ce critère.



Performances en fonction de la fréquence d’occurrences max_df

Très peu de mots apparaissent dans plus de 8 % des documents, les enlever ne change rien. En dessous de ce seuil, les performances diminuent. A cause de la variance des performances, la grid search trouve à chaque fois un seuil arbitraire non significatif.

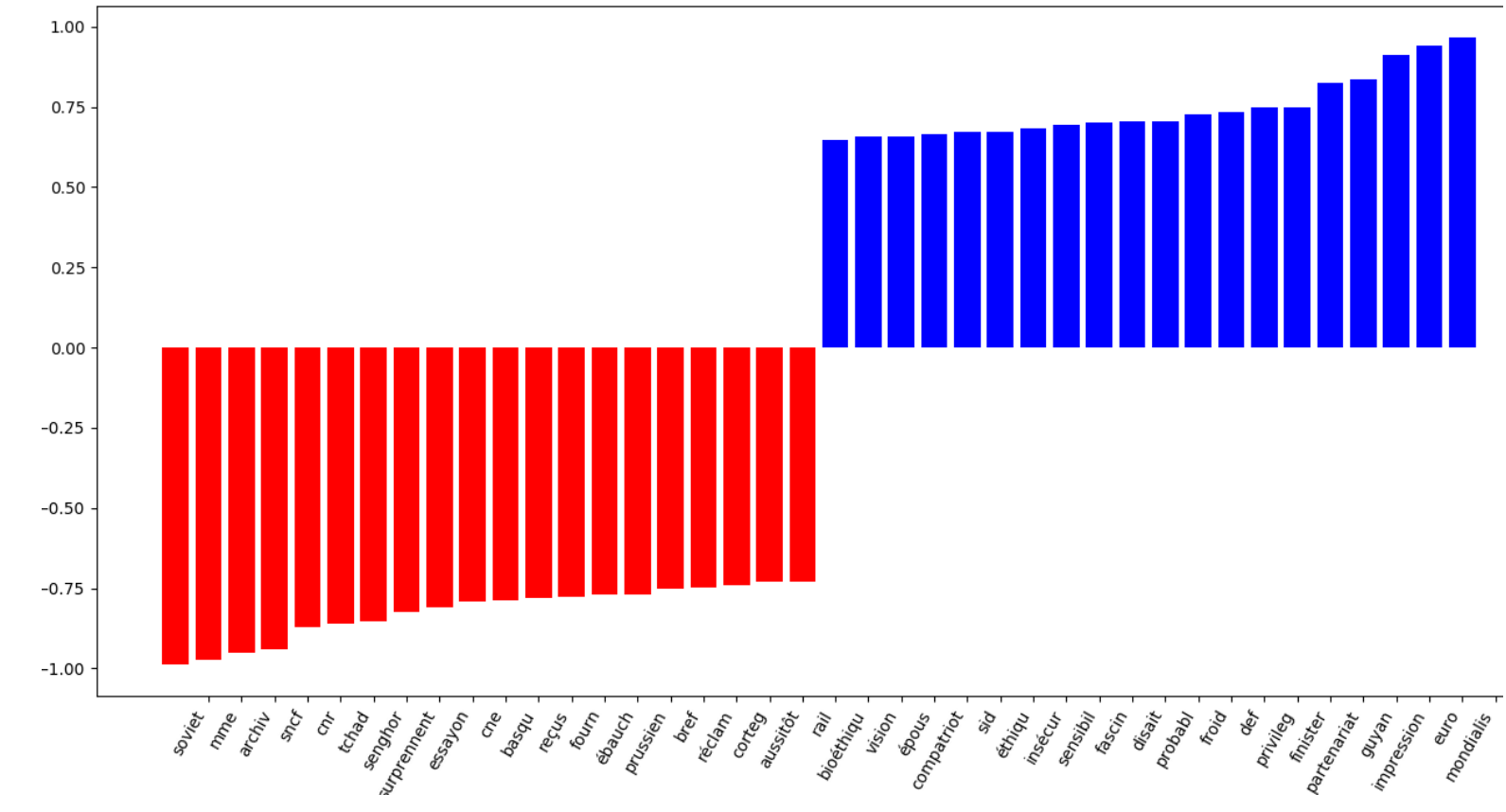


Performances en fonction de la valeur de C (mesuré deux fois en re-mélangeant les données)

L'hyper-paramètre C du modèle est la variable qui a le plus d'importance. L'optimal trouvé varie de **0.02 à 0.06**. À C=0 le modèle serait équivalent à un simple perceptron, mais bien que faible, la valeur de C est déterminante dans les performances, à C=0 les performances **chutent de 0.5 à 0.3** environ.

Un problème auquel nous faisons face est que, quelle que soit la valeur des paramètres, les différentes mesures de performances sur chaque fold ont une **forte variance**. Augmenter le nombre de fold augmente encore plus la variance.
Un autre problème était qu'en réduisant la taille de l'ensemble séparé en apprentissage/test, on a à plusieurs reprises vu le maximum de performance atteint augmenter. Cela était dû au fait que les documents sont un peu ordonnés et donc que certains sous échantillons étaient plus simples à différencier, notamment pour les 100 et 1000 premiers documents.

En analysant les poids du SVM, on peut regarder quels mots font le plus pencher la balance :



3) Code sources des fichiers auteurs.py sentiments.py et post_traitement.py:
a) auteurs.py

```
from sklearn.svm import LinearSVC
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np
from sklearn.model_selection import StratifiedKFold
import pickle
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
import random
CM = {'C': 1, 'M': -1}
CM_inv = {1: 'C', -1: 'M'}
class Parser:
    from nltk.corpus import stopwords
    from nltk.stem.snowball import SnowballStemmer
    stemmer = SnowballStemmer("french")
    stop_words = stopwords.words("french")

    @classmethod
    def load_data(cls, fileName):
        with open(fileName) as file:
            labels = []
            data = []
            all_lines = file.readlines()
            for line in all_lines:
                tab = line.split(" ")
                lab = tab[0][1:-1].split(":")[2]
                labels.append(CM[lab])
                phrase = cls.simplifie_phrase(tab[1:])
                data.append(" ".join(phrase))
            return np.array(data), np.array(labels)

    @classmethod
    def load_test(cls, fileName):
        with open(fileName) as file:
            data = []
            all_lines = file.readlines()
            for line in all_lines:
                tab = line.split(" ")
                phrase = cls.simplifie_phrase(tab[1:])
                data.append(" ".join(phrase))
            return np.array(data)


    @classmethod
    def simplifie_phrase(cls, p):
        """enlève les stopwords avant et après stemming
        sépare su apostrophe"""
        mots = " ".join([s.lower() for s in p])
        import re
        mots = re.sub("\d|\.|,|:|;|!|\?|\"|'", " ", mots)
        mots = mots.split()
        mots = [cls.stemmer.stem(s) for s in mots if s not in cls.stop_words]
        mots = [s for s in mots if s not in cls.stop_words]
        return mots
def writeTofile(tab):
    with open("result.txt", "w") as res:
        for t in tab:
```

```
            res.write(CM_inv[t] + "\n")
def load_pickled_train():
    try:
        pickle_in = open("data/x_train", "rb")
        x_train = pickle.load(pickle_in)
        pickle_in.close()

        pickle_in = open("data/y_train", "rb")
        y_train = pickle.load(pickle_in)
        pickle_in.close()

        print("lecture train effectuée")

    except FileNotFoundError:
        print("lecture train échouée")
        x_train, y_train = Parser.load_data("corpus.tache1.learn.utf8")

        pickle_out = open("data/x_train", "wb")
        pickle.dump(x_train, pickle_out)
        pickle_out.close()

        pickle_out = open("data/y_train", "wb")
        pickle.dump(y_train, pickle_out)
        pickle_out.close()
        print("train écrit")

    return x_train, y_train
def load_pickled_test():
    try:
        pickle_in = open("data/x_test", "rb")
        x_test = pickle.load(pickle_in)
        pickle_in.close()

        print("lecture test effectuée")

    except FileNotFoundError:
        print("lecture test échouée")
        x_test = Parser.load_test("data/corpus.tache1.test.utf8")

        pickle_out = open("data/x_test", "wb")
        pickle.dump(x_test, pickle_out)
        pickle_out.close()

        print("test écrit")

    return x_test
def plot_coefficients(classifier, feature_names, top_features=20):
    coef = classifier.coef_.ravel()
    ind = np.argsort(coef)
    top_positive_coefficients = ind[-top_features:]
    top_negative_coefficients = ind[:top_features]
    top_coefficients = np.hstack([top_negative_coefficients, top_positive_coefficients])
    # create plot
    plt.figure(figsize=(15, 5))
    colors = ["red" if c < 0 else "blue" for c in coef[top_coefficients]]
    plt.bar(np.arange(2 * top_features), coef[top_coefficients], color=colors)
    feature_names = np.array(feature_names)
    plt.xticks(np.arange(1, 1 + 2 * top_features), feature_names[top_coefficients],
rotation=60, ha="right")
    plt.show()
```

```
def cross_val(x, y, p_mod, p_vec):
    clf = LinearSVC(**p_mod)
    vectorizer = CountVectorizer(**p_vec)
    skf = StratifiedKFold(n_splits=3)

    score_f1 = []
    taille_dict = []
    for train_index, test_index in skf.split(x, y):
        fold_x_train, fold_x_test = x[train_index], x[test_index]
        fold_y_train, fold_y_test = y[train_index], y[test_index]

        vectorizer.fit(fold_x_train)

        # print(len(vectorizer.get_feature_names()))
        # import collections
        # buf = [set(x.split()) for x in fold_x_train]
        # mots = []
        # for b in buf:
        #     mots+=b
        # c = collections.Counter(mots)
        # print(min([c[m] for m in vectorizer.get_feature_names() if m in c]))
        # print(max([c[m] for m in vectorizer.get_feature_names() if m in c]))

        taille_dict.append(len(vectorizer.get_feature_names()))

        fold_x_train = vectorizer.transform(fold_x_train)
        fold_x_test = vectorizer.transform(fold_x_test)

        clf.fit(fold_x_train, fold_y_train)

        y_pred = clf.predict(fold_x_test)
        rapport = classification_report(fold_y_test, y_pred, output_dict=True)

        score_f1.append(rapport['-1']["f1-score"])
        # moyenne += clf.score(fold_x_test, fold_y_test)

    return (np.mean(score_f1), np.std(score_f1)), (np.mean(taille_dict),
np.std(taille_dict))
def optimize_cross_val(x_train, y_train):
    import itertools
    print("cross validation en cours")

    max_iter = [1e5]
    class_weight = ["balanced"]
    c = [2 ** (-3-i) for i in range(6)]

    # g = [2 ** (-i) for i in range(6)]
    # kernel = ["linear", "poly", "rbf"]
    # param_name = ["max_iter", "kernel", "C", "gamma"]
    params_model = [max_iter, class_weight, c]
    params_model_name = ["max_iter", "class_weight", "C"]


    ngram_range = [(1,1), (1,2), (2,2)]
    max_df = [1.0]#[.01, .02, .04, .08, 0.16, 10.0]#np.linspace(.01, .2, 5)
    min_df = range(0, 1, 5)

    params_vec = [ngram_range, max_df, min_df]
    params_vec_name = ["ngram_range", "max_df", 'min_df']
```

```
    p_max_mod, p_max_vec = None, None
    val_max, std_max = float("-inf"), None
    list_score = [[], []]
    list_taille = [[], []]
    for p_mod in itertools.product(*params_model):
        p_mod = {name: val for name, val in zip(params_model_name, p_mod)}

        for p_vec in itertools.product(*params_vec):
            p_vec = {name: val for name, val in zip(params_vec_name, p_vec)}

            (val, val_std), (taille, taille_std) = cross_val(x_train, y_train, p_mod,
p_vec)

            aff = [str(name) + ":" + str(p_mod[name]) for name in params_model_name]
            aff += [str(name) + ":" + str(p_vec[name]) for name in params_vec_name]
            list_score[0].append(val)
            list_score[1].append(val_std)
            list_taille[0].append(taille)
            list_taille[1].append(taille_std)
            print(aff, val, taille)

            if val > val_max:
                val_max = val
                std_max = val_std
                p_max_vec = p_vec
                p_max_mod = p_mod
    list_score = np.array(list_score)
    list_taille = np.array(list_taille)
    ax1 = plt.gca()
    ax1.errorbar(c, list_score[0], 2*list_score[1], capsize=5)
    #ax2 = ax1.twinx()
    #ax2.errorbar(c, list_taille[0], 2*list_taille[1], color='orange', capsize=5)
    plt.show()
    print("cross val, optimal trouvé :")
    print(p_max_mod, p_max_vec, val_max, std_max)
    return p_max_mod, p_max_vec, val_max, std_max
def main():
    x_train, y_train = load_pickled_train()

    # import collections
    # buf = [set(x.split()) for x in x_train]
    # mots = []
    # for b in buf:
    #     mots += b
    # c = collections.Counter(mots)
    #
    # print("nb doc:", len(x_train))
    # print("nb mot tot:", sum(c.values()))
    # print("nb mot diff:", len(c))
    # plt.hist(np.log2(list(c.values())),  bins=20)
    # plt.show()

    indices = list(range(len(x_train)))
    random.shuffle(indices)
    x_train = x_train[indices]
    y_train = y_train[indices]

    # n = len(x_train)
    n = 10000
```

```
    print("taille train:", n)

    p_max_model, p_max_vectorizer, _, _ = optimize_cross_val(x_train[:n], y_train[:n])
    #input("génération réponse ?")
    vectorizer = CountVectorizer(**p_max_vectorizer)
    vectorizer.fit(x_train)
    print("taille du dictionnaire:", len(vectorizer.get_feature_names()))
    print("fit en cours\n")
    p_max_vectorizer["max_iter"] = 1e3
    clf = LinearSVC(**p_max_model)
    x_train_vec = vectorizer.transform(x_train)
    clf.fit(x_train_vec, y_train)

    plot_coefficients(clf, vectorizer.get_feature_names())

    x_test = load_pickled_test()

    x_test = vectorizer.transform(x_test)

    y_pred = clf.predict(x_test)

    writeTofile(y_pred)
if __name__ == '__main__':
    main()
```

b) sentiments.py :

```
import os
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.svm import LinearSVC
from nltk.corpus import stopwords
from nltk.stem.snowball import SnowballStemmer
import numpy as np
def getListOfFiles(dirName):
    # create a list of file and sub directories
    # names in the given directory
    listOfFile = os.listdir(dirName)
    allFiles = list()
    # Iterate over all the entries
    for entry in listOfFile:
        # Create full path
        fullPath = os.path.join(dirName, entry)
        # If entry is a directory then get the list of files in this directory
        if os.path.isdir(fullPath):
            allFiles = allFiles + getListOfFiles(fullPath)
        else:
            allFiles.append(fullPath)
    return allFiles
def simplifie_phrase(p):
    stemmer = SnowballStemmer("english")
    stop_words = stopwords.words("english")
    phrase = p.lower().split(" ")
    phraseDecomp = [stemmer.stem(s) for s in phrase if s not in stop_words]
    return ' '.join(word for word in phraseDecomp)
def loadData(fileList):
    data=[]
    for fileName in fileList:
```

```
        file = open(fileName,"r")
        all_lines=file.readlines()
        all_lines = ' '.join(word for word in all_lines)
        data.append(simplifie_phrase(all_lines))
    return data
def loadData_test(fileName):
    data=[]
    file = open(fileName,"r")
    for document in file:
        data.append(simplifie_phrase(document))
    return data
def writeTofile(tab):
    with open("result_sentiment_annalysis.txt", "w") as res:
        for t in tab:
            res.write(t + "\n")
def convertOutput(pred):
    new_pred=[]
    for p in pred:
        if(p==-1):
            new_pred.append('C')
        else:
            new_pred.append('M')
    return new_pred
listFilePos = getListOfFiles("movies1000/pos/")
listFileNeg = getListOfFiles("movies1000/neg/")
dataPos = loadData(listFilePos)
print(len(dataPos))
print("je viens de load les data pos")
dataNeg = loadData(listFilePos)
print("je viens de load les data neg")
labelsPos = [1 for _ in range(len(dataPos))]
labelsNeg = [-1 for _ in range(len(dataNeg))]
############A decommenter###########
data = dataPos+dataNeg
labels = labelsPos+labelsNeg
##################################
indiceShuffle = np.random.permutation(np.array([i for i in range(len(data))]))
new_data=[]
new_labels=[]
for indice in indiceShuffle:
    new_data.append(data[indice])
    new_labels.append(labels[indice])
data = new_data
labels = new_labels
vectorizer = CountVectorizer(max_df=0.9,min_df=0.2)
vectorizer.fit(data)
data = vectorizer.transform(data)
donnees_test = loadData_test("testSentiment.txt")
donnees_test =  vectorizer.transform(donnees_test)
print(donnees_test[0])
clf = LinearSVC(verbose=1,max_iter=1e6)
print("je commence le fit")
clf.fit(data,labels)
y_pred = clf.predict(donnees_test)
y_pred = convertOutput(y_pred)
writeTofile(y_pred)
```

c) post_traitement.py

```
import numpy as np
import collections
def writeTofile(tab):
    with open("nouveau_result_sentiment_annalysis.txt", "w") as res:
        for t in tab:
            res.write(t + "\n")
file = open("result_sentiment_annalysis.txt")
lines = file.readlines()
new_line=[]
for l in lines:
    new_line.append(l.replace("\n",""))
print(collections.Counter(new_line))
moyenne=1
for k in range(moyenne,len(new_line)-moyenne):
    co = collections.Counter(new_line[k-moyenne:k+moyenne+1])
    """
    if(new_line[k]=='C' and co['M']==2):
        new_line[k]='M'
    """
    if(new_line[k]=='M' and co['C']==2):
        new_line[k]='C'
for k in range(0,len(new_line)-5):
    if(new_line[k]=='C' and new_line[k+2]=='C'):
        new_line[k+1]='C'
    if(new_line[k]=='C' and new_line[k+3]=='C'):
        new_line[k+1]='C'
        new_line[k+2]='C'
    if(new_line[k]=='C' and new_line[k+4]=='C'):
        new_line[k+1]='C'
        new_line[k+2]='C'
        new_line[k+3]='C'
    if(new_line[k]=='C' and new_line[k+5]=='C'):
        new_line[k+1]='C'
        new_line[k+2]='C'
        new_line[k+3]='C'
        new_line[k+4]='C'
print(collections.Counter(new_line))
writeTofile(new_line)
```