

## Compte rendu TME Traitement Automatique du Langage

### 1) POS TAG

La tâche consiste à labelliser chacun des mots des documents.

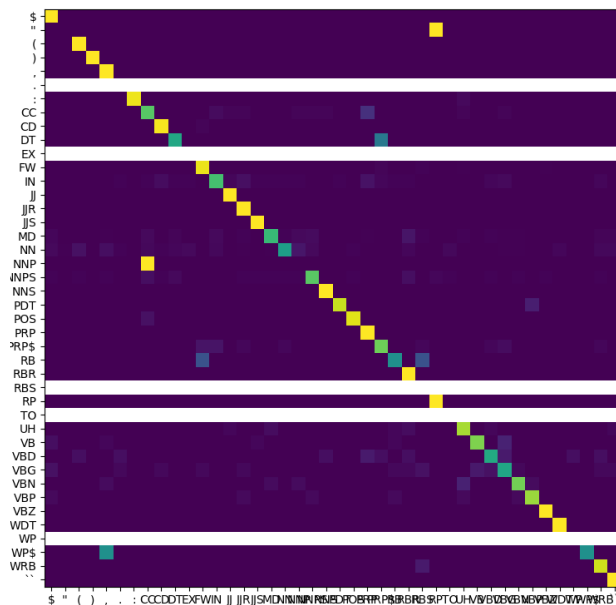
L'approche par **dictionnaire** consiste à apprendre la clé associée à chaque mot. Sur le jeu de test, on obtient une performance de **75 %**.

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 servant d'un modèle déjà entraîné, on obtient des meilleures 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.

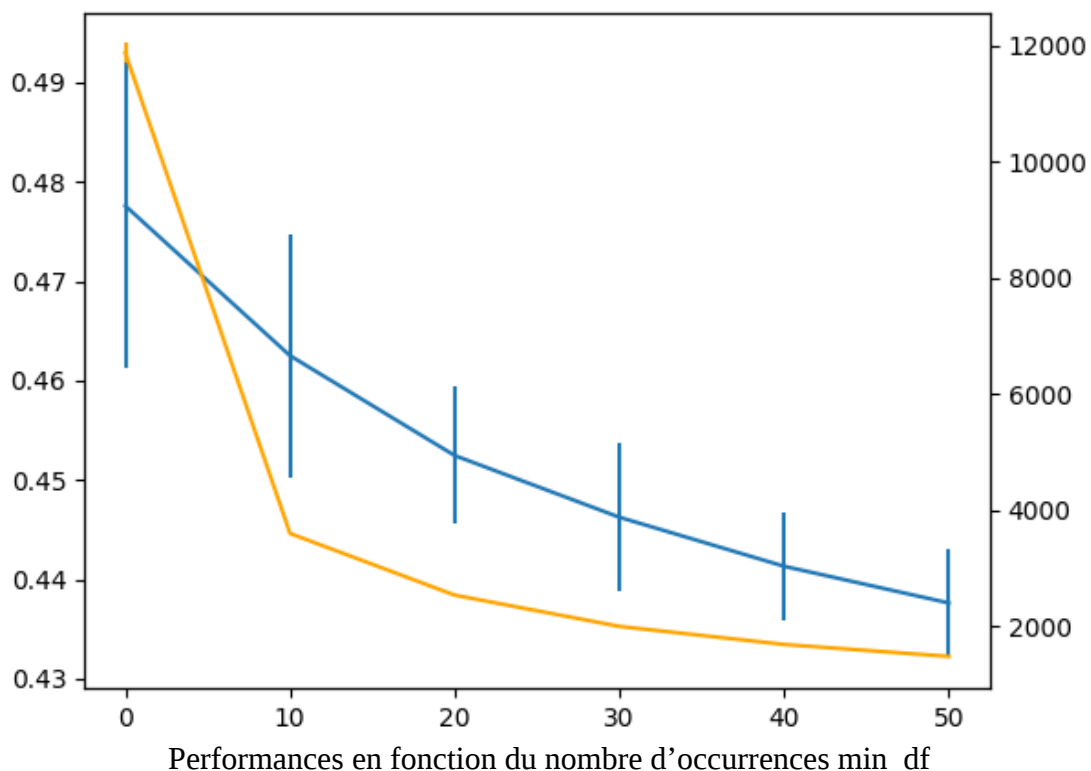
Pour représenter les documents on utilise la technique du **bag of words**. Un document est représenté par un vecteur contenant à l'indice  $i$  le nombre d'occurrences dans le document du  $i$ ème mot de l'ensemble des mots du vocabulaire.

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.

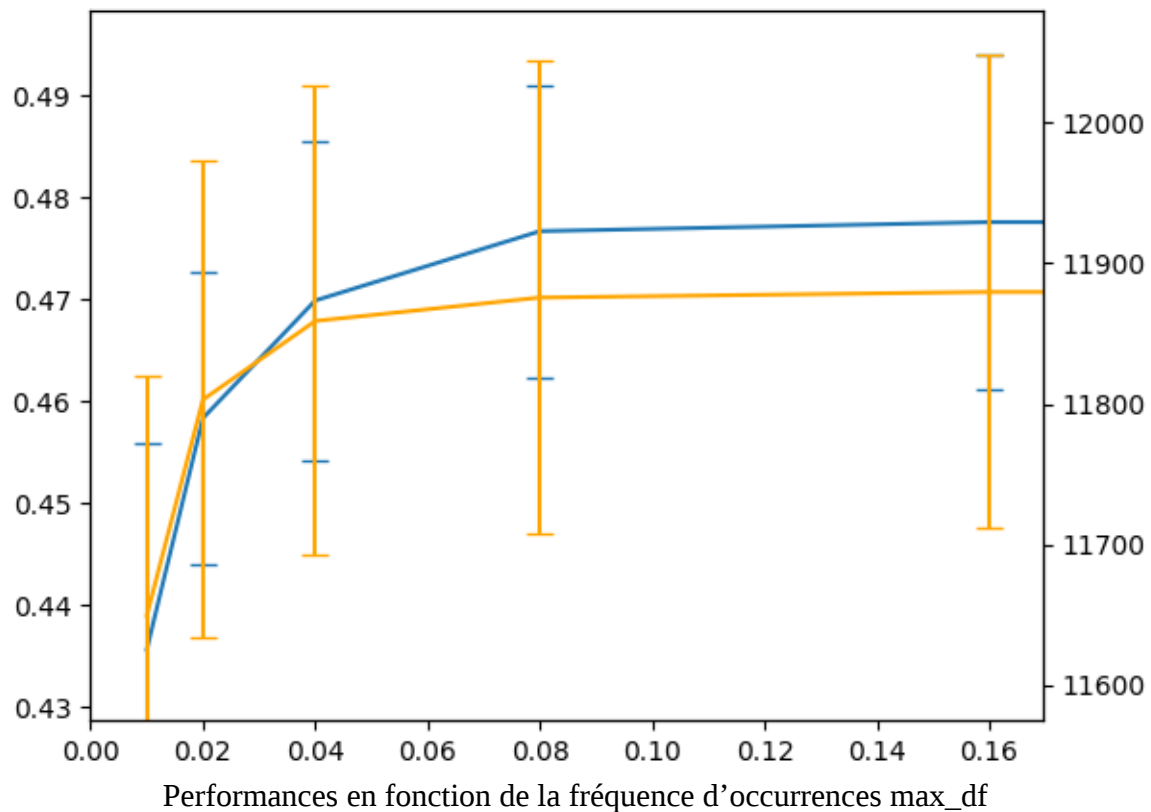
Certains pré-traitements ont été appliqués systématiquement, ils consistent à enlever la **punctuation** et les **stop-words**, et à **stemmer** les mots.

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.

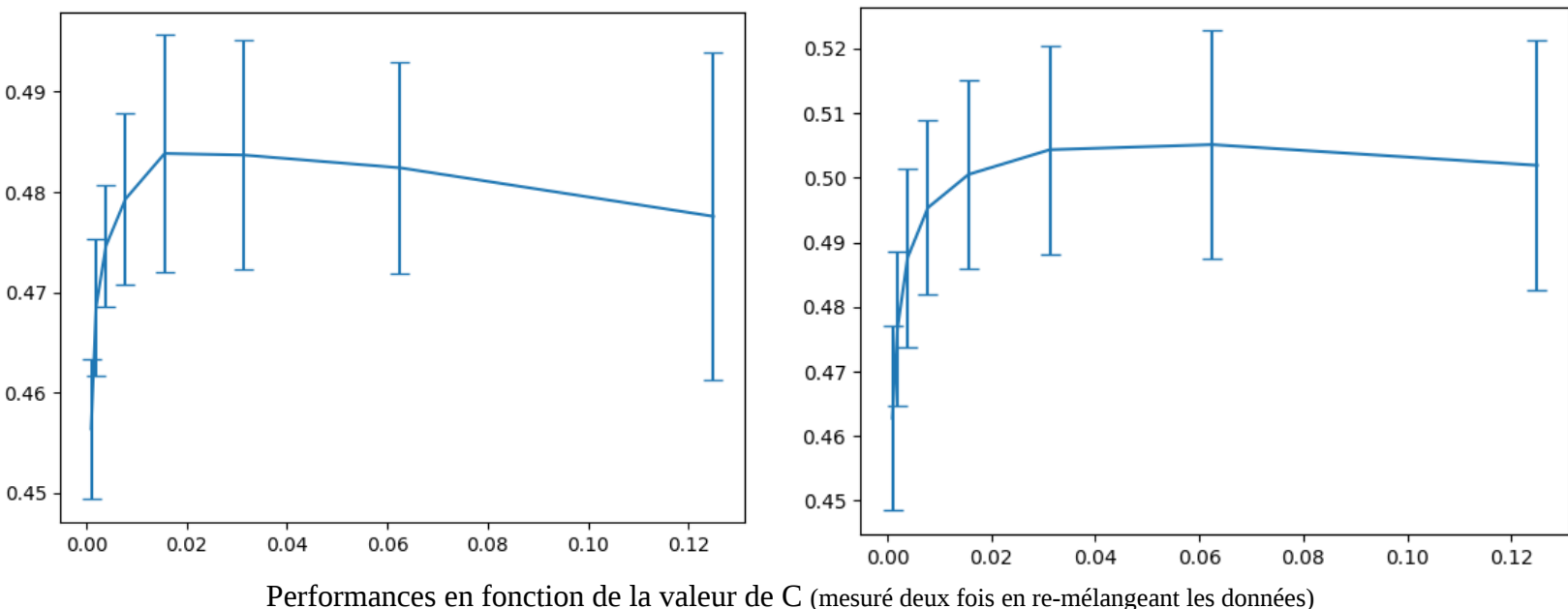
Sur les graphiques suivants, la courbe en bleu représente le **score F1** mesuré par cross-validation. Si la **taille du dictionnaire** dépend de la variable, la courbe orange représente la taille moyenne du vocabulaire (et son échelle est à droite). La valeur des autres paramètres étant fixés à peu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



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



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.

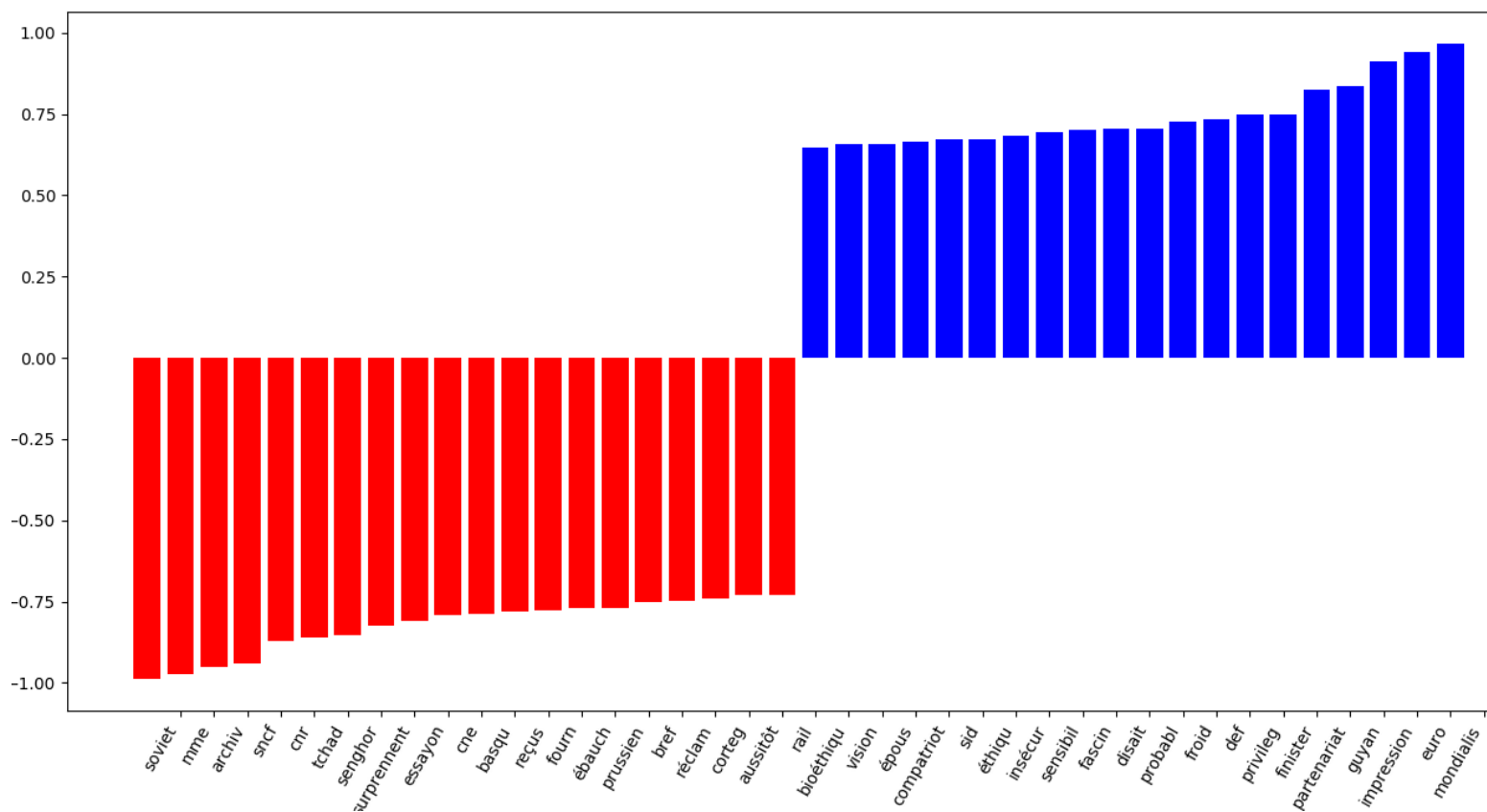


L'hyper-paramètre C du modèle est la variable qui a le plus d'importance. L'optimal trouvé varie de **0.02 à 0.06**. À C=0 le modèle serait équivalent à un simple perceptron, mais bien que faible, la valeur de C est déterminante dans les performances, à C=0 les performances **chutent de 0.5 à 0.3** environ.

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.

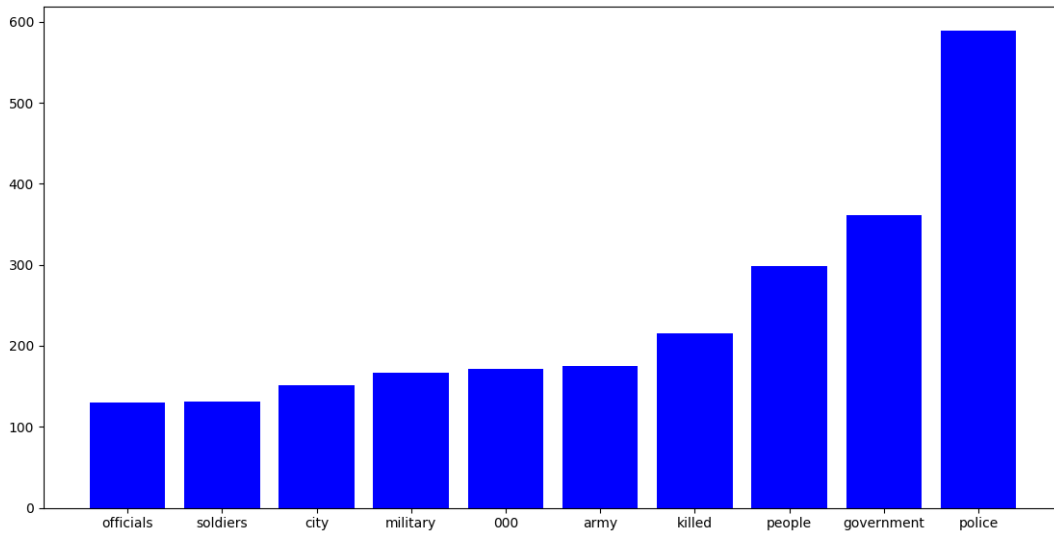
Un autre problème était qu'en réduisant la taille de l'ensemble séparé en apprentissage/test, on a à plusieurs reprises vu le maximum de performance atteint augmenter. Cela était dû au fait que les documents sont un peu ordonnés et donc que certains sous échantillons étaient plus simples à différencier, 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 :



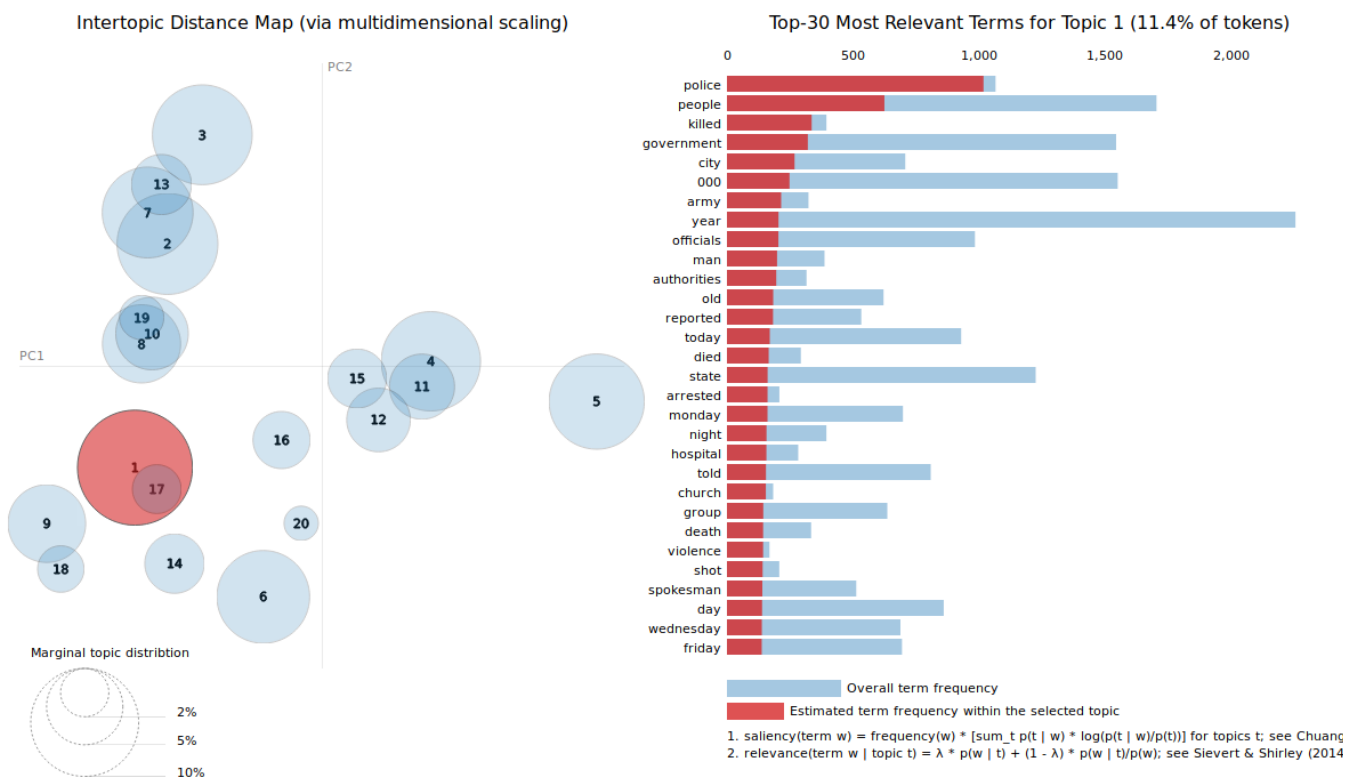
### 3) Segmentation thématique

On a utilisé le modèle **LDA** : Latent Dirichlet allocation (on aurait aussi pu utiliser PLSA ou k-means). Sans pre-processing, certains **clusters ne sont pas associés à un thème et regroupent des mots souvent utilisés**. Grâce au pre-processing, la plupart de ces clusters disparaissent. En post-processing, on peut néanmoins visualiser chacun des clusters créés et **déterminer à la main lesquels d'entre eux ne sont pas pertinents**. Cela permet d'obtenir un **classifieur** de nouveaux documents ne choisissant que parmi les clusters sélectionnés.

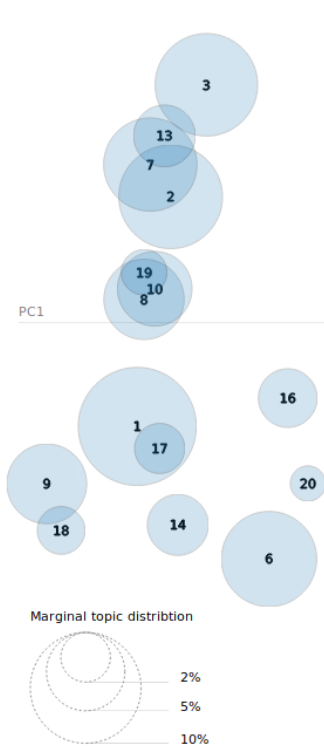


Les mots des clusters correspondent généralement bien à un même champ lexical

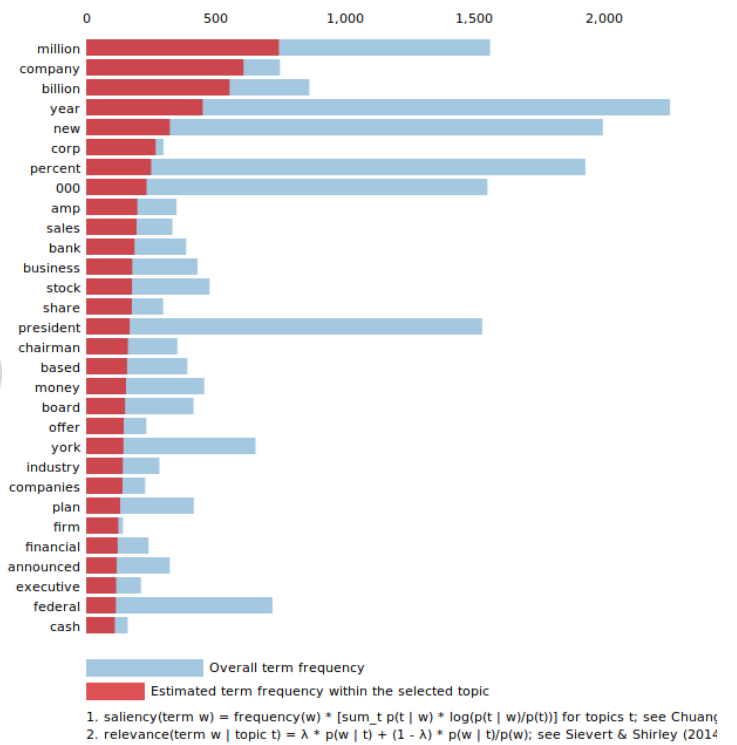
On a ensuite visualisé les clusters créés grâce à pyLDAvis. Un cluster est principalement défini par ses mots les plus **saillants** : ceux qui apparaissent beaucoup dans ce cluster mais peu ailleurs.



Intertopic Distance Map (via multidimensional scaling)



Top-30 Most Relevant Terms for Topic 4 (8.5% of tokens)



```
taper votre document :I love america
[[0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.68333333 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667]]
taper votre document :I love army
[[0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.35716206
 0.01666667 0.01666667 0.01666667 0.01666667 0.34283794 0.01666667
 0.01666667 0.01666667]]
taper votre document :space nasa
[[0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.68333333 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667]]
taper votre document :war warfare gun
[[0.0125 0.0125 0.0125 0.0125 0.0125 0.0125 0.0125
 0.0125 0.0125 0.33039669 0.0125 0.0125 0.0125
 0.44460331 0.0125 0.0125 0.0125 0.0125 0.0125
 0.0125 0.0125 ]]
taper votre document :usa junk food
[[0.0125 0.76249999 0.0125 0.0125 0.0125 0.0125 0.0125
 0.0125 0.0125 0.0125 0.0125 0.0125 0.0125
 0.0125 0.0125 0.0125 0.0125 0.0125 0.01250001 0.0125
 0.0125 0.0125 ]]
taper votre document :crocodile
[[0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05
 0.05 0.05 0.05 0.05 0.05 0.05]]
taper votre document :alligator
[[0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05
 0.05 0.05 0.05 0.05 0.05 0.05]]
taper votre document :usa
[[0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025
 0.025 0.025 0.025 0.025 0.525 0.025 0.025 0.025]]
```

Étant donné un nouveau document, le modèle retourne le score d'appartenance à chacun des clusters

Afin d'obtenir des **mesures d'évaluation**, on applique le clustering sur des **données labellisées**. Avec un jeu de données de 90 classes on cherche à regrouper les données en un même nombre de clusters. La précision d'un cluster est calculée en divisant la taille de sa classe majoritaire par sa taille totale. On obtient une **précision de 60 %** par cluster environ.

#### 4) Word2vec

Word2vec associe un **embedding** à chaque mot de telle sorte que deux mots sémantiquement proches ont une similarité en cosinus élevée entre leurs deux embeddings.

Pour entraîner le modèle word2vec nous avons deux méthodes possibles. La première 'skipgram' qui prend un mot en entrée et doit prédire les mots du contexte en sortie, et la seconde méthode 'CBOW' qui prend plusieurs mot de contexte en entrée et sort le mot qui appartient à ce contexte. Le contexte est une sliding window qui est définie au préalable par exemple 5 mots, de plus on peut définir un autre paramètre qui est la dimension de notre espace de sémantique (notre embedding), dans la majorité des cas on choisit environ 300 pour word2vec.

En entraînant le modèle sur une base de données de l'ordre de 1000 documents, on arrive à reproduire certaines expériences de la forme :

« **king** » + (« **woman** » - « **man** ») = « **queen** ».

Une fois le modèle entraîné, on entraîne un classifieur de sentiments en se servant des embeddings des mots plutôt que des bag-of-words. Afin d'obtenir la représentation d'un document, on doit **agréger** les représentations des mots. Cela peut être implémenté par une somme, une moyenne ou un max.

La base de données utilisée était petite et les résultats sont médiocres. En utilisant le modèle disponible **pré-entraîné sur Wikipedia**, on pourrait obtenir de biens meilleures performances.

## ANNEXE : Code source

### 1) POS TAG

```
import numpy as np
import matplotlib.pyplot as plt
# allx: liste de séquences d'observations
# allq: liste de séquences d'états
# N: nb états
# K: nb observation
def learnHMM(allx, allq, N, K, initTol=True):
    if initTol:
        eps = 1e-5
        A = np.ones((N, N)) * eps
        B = np.ones((N, K)) * eps
        Pi = np.ones(N) * eps
    else:
        A = np.zeros((N, N))
        B = np.zeros((N, K))
        Pi = np.zeros(N)
    for x, q in zip(allx, allq):
        Pi[int(q[0])] += 1
        for i in range(len(q) - 1):
            A[int(q[i]), int(q[i + 1])] += 1
            B[int(q[i]), int(x[i])] += 1
        B[int(q[-1]), int(x[-1])] += 1 # derniere transition
    A = A / np.maximum(A.sum(1).reshape(N, 1), 1) # normalisation
    B = B / np.maximum(B.sum(1).reshape(N, 1), 1) # normalisation
    Pi = Pi / Pi.sum()
    return Pi, A, B
def viterbi(x, Pi, A, B):
    T = len(x)
    N = len(Pi)
    logA = np.log(A)
    logB = np.log(B)
    logdelta = np.zeros((N, T))
    psi = np.zeros((N, T), dtype=int)
    S = np.zeros(T, dtype=int)
    logdelta[:, 0] = np.log(Pi) + logB[:, x[0]]
    #forward
    for t in range(1, T):
        logdelta[:, t] = (logdelta[:, t - 1].reshape(N, 1) + logA).max(0) + logB[:, x[t]]
        psi[:, t] = (logdelta[:, t - 1].reshape(N, 1) + logA).argmax(0)
    # backward
    logp = logdelta[:, -1].max()
    S[T - 1] = logdelta[:, -1].argmax()
    for i in range(2, T + 1):
        S[T - i] = psi[S[T - i + 1], T - i + 1]
    return S, logp #, delta, psi
def load(filename):
    listeDoc = list()
    with open(filename, "r") as f:
        doc = list()
        for ligne in f:
            #print "l : ",len(ligne)," ",ligne
            if len(ligne) < 2: # fin de doc
                listeDoc.append(doc)
                doc = list()
                continue
            mots = ligne.split(" ")
            doc.append((mots[0], mots[1]))
```



```

    return listeDoc
def decodeSortie(indList, ind2cle):
    new_list = []
    for i in indList:
        new_list.append(ind2cle[i])
    return new_list
def evalHmm(Pi, A, B, allxT, allqT):
    compteur = 0
    taille = 0
    confMat = np.zeros((len(A), len(A)))
    for k in range(len(allxT)):
        S, _ = viterbi(allxT[k], Pi, A, B)
        for i in range(len(S)):
            if (S[i] == allqT[k][i]):
                compteur += 1
                confMat[allqT[k][i] - 1][S[i] - 1] += 1

        taille += 1
    print("nb bonnes réponses:", compteur)
    confMat = [l / sum(l) for l in confMat]
    return compteur / taille, confMat
def affichage(A, posTag):
    filename = "test"
    plt.figure()
    plt.imshow(A, interpolation='nearest')
    localLabs = posTag # liste des POS-TAG
    plt.yticks(range(len(localLabs)), localLabs) # affichage sur l'image
    plt.xticks(range(len(localLabs)), localLabs)
    if filename is not None:
        plt.savefig(filename)
if __name__ == '__main__':
    filename = "data/wapiti/chtrain.txt" # a modifier
    filenameT = "data/wapiti/chtest.txt" # a modifier

    alldocs = load(filename)
    alldocsT = load(filenameT)

    # alldocs etant issu du chargement des données

    buf = [[m for m, c in d] for d in alldocs]
    mots = []
    [mots.extend(b) for b in buf]
    mots = np.unique(np.array(mots))
    nMots = len(mots) + 1 # mot inconnu

    mots2ind = dict(zip(mots, range(len(mots))))
    mots2ind["UUUUUUUU"] = len(mots)

    buf2 = [[c for m, c in d] for d in alldocs]
    cles = []
    [cles.extend(b) for b in buf2]
    cles = np.unique(np.array(cles))
    cles2ind = dict(zip(cles, range(len(cles))))

    ind2cle = dict(zip(range(len(cles)), cles))

    nCles = len(cles)

    print(nMots, nCles, " in the dictionary")

```

```

# mise en forme des données
allx = [[mots2ind[m] for m, c in d] for d in alldocs]
allxT = [[mots2ind.get(m, len(mots)) for m, c in d] for d in alldocsT]

allq = [[cles2ind[c] for m, c in d] for d in alldocs]
allqT = [[cles2ind.get(c, len(cles)) for m, c in d] for d in alldocsT]

Pi, A, B = learnHMM(allx, allq, nCles, nMots)

acc, confMat = evalHmm(Pi, A, B, allxT, allqT)
print("précision:", acc)

plt.figure()
plt.imshow(confMat, interpolation='nearest')
localLabs = cles # liste des POS-TAG
plt.yticks(range(len(localLabs)), localLabs) # affichage sur l'image
plt.xticks(range(len(localLabs)), localLabs)
plt.show()
X = allxT[0]
S, logP = viterbi(X, Pi, A, B)
print(decodeSortie(S, ind2cle))
affichage(A, cles)

```

## 2) Classification

### 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)
```

### 3) Segmentation thématique

```
from bs4 import BeautifulSoup
import numpy as np
from sklearn.decomposition import LatentDirichletAllocation as LDA
from sklearn.feature_extraction.text import CountVectorizer
import matplotlib.pyplot as plt
import pyLDAvis
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
import pyLDAvis.sklearn
from nltk.corpus import reuters
from nltk.corpus import stopwords
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.feature_extraction.text import TfidfVectorizer
from collections import Counter

n_classes = 90
labels = reuters.categories()
def load_data(config={}):
    """
    Load the Reuters dataset.
    Returns
    -----
    Tuple of Numpy arrays: `(x_train, y_train), (x_test, y_test)`.
    """
    stop_words = stopwords.words("english")
    vectorizer = TfidfVectorizer(stop_words=stop_words)
    mlb = MultiLabelBinarizer()
    documents = reuters.fileids()
    test = [d for d in documents if d.startswith('test/')]
    train = [d for d in documents if d.startswith('training/')]
    docs = {}
    docs['train'] = [reuters.raw(doc_id) for doc_id in train]
    docs['test'] = [reuters.raw(doc_id) for doc_id in test]
    xs = {'train': [], 'test': []}
    xs['train'] = vectorizer.fit_transform(docs['train']).toarray()
    xs['test'] = vectorizer.transform(docs['test']).toarray()
    ys = {'train': [], 'test': []}
    ys['train'] = mlb.fit_transform([reuters.categories(doc_id)
                                    for doc_id in train])
    ys['test'] = mlb.transform([reuters.categories(doc_id)
                               for doc_id in test])
    data = {'x_train': xs['train'], 'y_train': ys['train'],
            'x_test': xs['test'], 'y_test': ys['test'],
            'labels': globals()["labels"]}
    return data

d = load_data()
X=d["x_train"]
lab = d["y_train"]
#changer le max_iter pour de meilleurs performances.
clf = LDA(n_components=n_classes,max_iter=10,topic_word_prior=0.9)
clf.fit(X)
#calcul de pureté du premier cluster
pred_cluster1=[]
for k in range(len(X)):
    pred = np.argmax(clf.transform([X[k]]))
    if(pred==1):
        pred_cluster1.append(k)
lab_clust1=[]
for p in pred_cluster1:
    lab_clust1.append(np.argmax(lab[p]))
c = Counter(lab_clust1)
```

```

somme = sum(c.values())
classe_majoritaire = np.max(list(c.values()))
print("Pureté de la classe numero 1 : ")
print(classe_majoritaire/somme)

```

```

file = open("ap/ap.txt")
soup = BeautifulSoup(file, 'html.parser')
txt = soup.find_all("text")
txt = [str(t).replace("<text>", "").replace("</text>", "") for t in txt ]
vectorizer = CountVectorizer(strip_accents = 'unicode',
                             stop_words = 'english',
                             lowercase = True,
                             max_df = 0.5,
                             min_df = 10)

X = vectorizer.fit_transform(txt)
#changer le max_iter pour de meilleurs performances.
clf = LDA(n_components=20,max_iter=10)
clf.fit(X)
print(clf.components_)
print(clf.components_.shape)
parametre_classe = clf.components_
#arguments: 'doc_topic_dists', 'doc_lengths', 'vocab', and 'term_frequency'
print(len(parametre_classe))
#Visualisation
"""
tfidf_vectorizer = TfidfVectorizer(**vectorizer.get_params())
dtm_tfidf = tfidf_vectorizer.fit_transform(txt)
movies_vis_data = pyLDavis.sklearn.prepare(clf, X, vectorizer)
pyLDavis.show(movies_vis_data)
for k in range(20):
    p_c = parametre_classe[k]
    indices_mots = p_c.argsort()[-10:]
    mots=[]
    for i in indices_mots:
        tmp=[]
        tmp.append(vectorizer.get_feature_names()[i])
        tmp.append(p_c[i])
        mots.append(tmp)
    print(mots)
    plt.bar([m[0] for m in mots], [m[1] for m in mots], color="blue")
    plt.show()
"""
while(1):
    doc_a_test = input("taper votre document :")
    x_a_test= vectorizer.transform([doc_a_test])
    print(clf.transform(x_a_test))

```

#### 4) Word2vec

```
from gensim.models import Word2Vec
from nltk.stem.porter import *
import gensim
import os
import logging
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
logging.getLogger().addHandler(logging.StreamHandler())
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 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(all_lines)
    return data
def writeTofile(tab):
    with open("result_sentiment_annalysis.txt", "w") as res:
        for t in tab:
            res.write(t + "\n")
listFilePos = getListOfFiles("AFDmovies/movies1000/pos/")
listFileNeg = getListOfFiles("AFDmovies/movies1000/neg/")
dataPos = loadData(listFilePos)
print(len(dataPos))
print("je viens de load les data pos")
dataNeg = loadData(listFilePos)
print(len(dataNeg))
print("je viens de load les data neg")
labelsPos = [1 for _ in range(len(dataPos))]
labelsNeg = [-1 for _ in range(len(dataNeg))]
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
"""
```

```

sentences=[]
for d in data :
    phrases = d.split("\n")
    for p in phrases:
        sentences.append(p.split())
if(os.path.exists("word2vecModel")):
    print("Le modele etait deja entrainé")
    w2v = Word2Vec.load("word2vecModel")
else:
    print("j'entraine le modele")
    w2v = gensim.models.word2vec.Word2Vec(sentences=sentences,
                                           size=200, window=5,          ### here we train
a cbow model
                                           min_count=5,
                                           sample=0.001, workers=3,
                                           sg=1, hs=0, negative=5,      ### set sg to 1 to
train a sg model
                                           cbow_mean=1,
                                           iter=10)

    w2v.save("word2vecModel")
new_data=[]
for document in data:
    tmp = np.array([w2v.wv[mot] for mot in document.split() if mot in w2v.wv.vocab])
    rep_doc=np.sum(tmp,axis=0)
    new_data.append(rep_doc)
#print(labels)
clf = SVC(gamma='auto',kernel="linear",max_iter=1e8,verbose=True,tol=1e-10)
clf.verbose=True
clf.fit(new_data,labels)
print("coef", clf.coef_)
print("len", len(new_data), len(labels))
pred=clf.predict(new_data)
print("precision", accuracy_score(pred,labels))
print("fin entraînement model")
print(w2v.most_similar("king", topn=2))
# is great really closer to good than to bad ?
print("great and good:",w2v.wv.similarity("great","good"))
print("great and bad:",w2v.wv.similarity("great","bad"))
print(w2v.wv.most_similar(positive=["awesome","bad"],negative=["good"],topn=5))
out = w2v.wv.evaluate_word_analogies("questions-words.txt",case_insensitive=True)[0]
#original semantic syntactic dataset.
print(out)

```