# Rapport TP 2 – NLP

Pierre VALENTIN

# Sommaire

# Introduction

Ce rapport est personnel mais le travail sur le code a été fait en groupe avec Christopher de Boisvillier. La première partie, nous avons deux codes différents mais les deux dernières parties sont quasiment identiques, car nous avons travaillé ensemble.

Le code est disponible sur mon compte github à l'adresse suivante :

https://github.com/Focom/NLPWork2

# 1 - Recherche d’information

Dans un premier temps nous devons extraire l’information contenue dans les fichiers HTML. L’extraction est simple il suffit d’identifier les bonnes balises et métadonnée entourant le champ qui nous intéresse. Pour trouver les informations sur la geography, on cherche la mention Geography – Note puis on prend tout ce qui se trouve dans les balises. Voilà le code qui sen charge.

```
def findGeography(fileName) :
try :
pattern_all_file = re.compile("(Geography – note:[\w\W]*?<div.*?\">)(.*)?(<\/d)")
country_file = codecs.open(fileName, “r”, encoding="utf8 »)
test = pattern_all_file.search(country_file.read())
return str(test.group(2))
except AttributeError :
return '\"\"'
```

Une fois toutes ces fonctions écrites, on peut construire les fichiers JSON contenant les informations sur les pays. Ces fichiers seront utilisés pour alimenter ElasticSearch notre moteur de recherche utilisant Lucene. Exemple d'Albania.json

```
{
“Name”: “Albania”,
“Intro”: « Albania declared its independence from the Ottoman Empire in 1912, but was conquered by Italy in 1939, and occupied by Germany in 1943. Communist partisans took over the countr…",
“Economy”: « Albania, a formerly closed, centrally-planned state, is making the difficult transition to a more modern open-market economy. Albania managed to weather the first waves of the global financial crisis but, more recently, its negative effects…",
“Geography”: « strategic location along Strait of Otranto (links Adriatic Sea to Ionian Sea and Mediterranean Sea)"
}
```

Ensuite on « index » tous les documents dans la base d’ElasticSearch. Avec cette commande.

```
es.index(index="factbook", doc_type="country", id=i, body=content)
```

Une fois dans elasticsearch on peut lancer nos requêtes. Dans mon cas le serveur tourné en local avec une configuration par défaut. `es = Elasticsearch([{"host": "localhost", "port": 9200}])`

ElasticSearch utilise un moteur de « query » très évoluer. L'objectif est optimisé la précision et le rappelle avec cette syntaxe précise. Chaque question étant différente, il ne faut pas tomber dans le piège décrire une syntaxe de query pour chaque question. La solution que l'on a choisie est simple. Pour chaque mot de la question, le document doit faire apparaître dans un ou plusieurs de ses champs 79 % des mots de la question. La queue est définie ci-dessous :

```
question = {
"query": {
      "match": {
            "_all »: {
                  "query": words,
                  "operator": "or",
                  "minimum_should_match »: « 79 %"
                  }
            }
      }
}
```

Exemple de résultat pour la première question, « European nations with viking ancestors »

["Denmark", "Norway", 'Faroe Islands', 'European Union', "France", 'United Arab Emirates', "Singapore", "Moldova", "Guernsey", "Serbia"]

Avec cet exemple, on peut désormais calculer la « mean average precision » de la requête. Pour chaque pays donné par elasticsearch, on fait la somme des vrais positifs divisés par la position du document. Le MAP a pour avantage de prendre en compte la position du document dans le résultat. Si on devait reconstruire google, l'ordre des documents est donc important. Par exemple, avec la requête précédente, les averages précisions sont [1.0, 1.0]. On applique cette technique pour toutes les requêtes et on fait la moyenne de toutes ces valeurs et on obtient le MAP.

Ici dans mon cas le MAP est très faible de 0.3325396825396826

Une valeur si basse viens du fait que mes requêtes ont un rappelé très faible même si j'utilise un « ou » pour chaque argument de la requête. Christopher a lui utilise la même requête obtient des résultats différents peut-être cela vient de la construction de mes documents où les paramètres d'elasticsearch. Si j'avais résolus ce problème une autre manière de rendre plus efficace mon algorithme serait de créer une syntaxe de query plus efficace. Les résultats seraient sûrement meilleurs en supprimant les stops Words et sûrement en stemmant les mots mais cela donne beaucoup d'overhead au programme. Pour se faire un avis il faudrait que je continue de tester de manière empirique.

# 2 – Classifications de questions

Dans cette partie, nous devons faire une classification multiclasse. Pour cela nous utilisons la librairie SciKitLearn. Cette librairie est très puissante et nous permet en quelque ligne de prédire très précisément la classe des questions. Pour cela dans un premier temps nous créons un csv contenant les classes et les questions pour chaque elemen2ts du corpus. Ce code est géré, dans le fichier exo2.py

Une fois cette partie faite on peut s'intersser a la partie importante, la classification. Dans cette application, nous utilisons deux algorithmes naïve baise et random forest.

Avant de passer les questions dans les algorithmes,

1. On sépare en deux le corpus de questions
2. Puis nous devons créer les structures de données adéquat, pour cela nous créons des countVectorizer pour le corpus d'entraînement et de test.

```
count_vectorizer = CountVectorizer()
counts = count_vectorizer.fit_transform(vecteurQuestion[:cc].values)
```

Ici le vecteurQuestion contient uniquement les cc première question.

Ensuite on génère le classifier avec :

```
classifier.fit(counts, targetsClasse)
```

```
Enfin on peut passer le vecteur de question test et voir les predictions.
```

Pour faire une analyse de la précision et du rappel, on crée une nouvelle structure de données sur le résultat avec la fonction constructTableRP()

{'0': {'class': 'DEFINITION', 'bool': True}, '1': {'class': 'DEFINITION', 'bool': True},

On construit un JSON avec pour première clé le numéro de la question suivis de ces paramètres. Les paramètres sont la classe prédite puis un booleen définissant si la la classification est correcte.

Une fois cette structure générée on peut calculer facilement la précision et comparer les algorithmes de classifications.

Les précisions en fonctions de la classe et de l'algorithme :

| Algorithme/Classe | **Location** | **Definition** | **Temporal** |
|---|---|---|---|
| **Naive Bays** | 0.83 | 0.9 | 1 |
| **Random Forest** | 0.65 | 0.84 | 0.64 |

On remarque que naive bays est l'algorithme le plus performant. Avec ce taux de précision, on pourrait mettre ce code en production sur une vraie application.

Le calcule du rappel est défini dans le code suivant sa définition mathématique.

# 3 - Classification et analyse de sentiments

Dans cette partie nous avons aussi divisé notre logique en deux. Le premier fichier soccupe du prétraitement.

En premier on crée la structure de donne qui va nous permettre de faire le prétraitement du texte. La fonction `createDicoClasse()` s'en charge.

Une fois cette structure établie, on peut créer notre pipeline de prétraitement. Les noms de fonctions sont assez explicit mais voila les détails :

- On transforme en lowercase :
  - `lowercaseConvert()`
- On retire les stopwords :
  - `stopWords()`
- On retire la ponctuation :
  - `deletePunc()`
- On réalise le stemming :
  - `correctStringPosttag(tocorrect, listTag,ps):`

On retire les tags suivants :

["CC","CD","DT","EX","FW","IN","LS","MD","PDT","POS","PRP","RP","SYM","TO","UH","WDT","WP","WRB","PRP$","WP$"]

Une fois que l'on a réalisé notre prétraitement, on peut créer le csv qui contient le texte traite prêt a être envoyé dans scikitlearn. On utilise, ensuite la même logique que l'exercice 2. Cependant c'est encore plus simple car ici la classification est binaire.

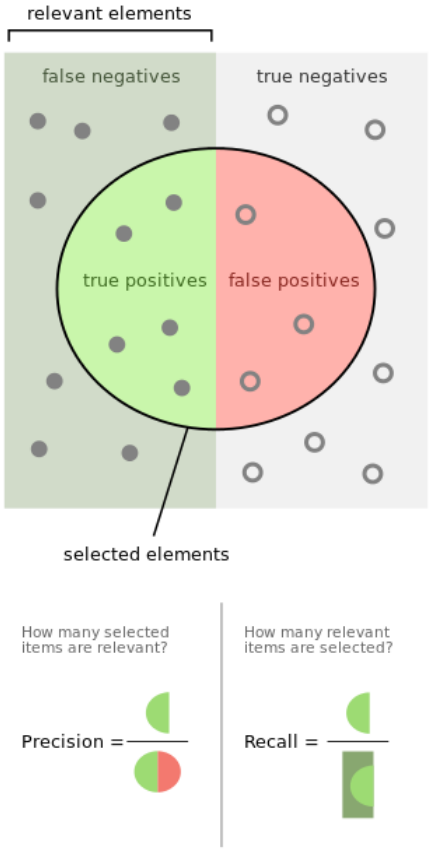
Pour créer et lire le csv, on utilise la librairie pandas, pour exécuter le code installer celle-ci avec pip.

En premier on choisit le classificateur entre naïve bayes et RandomForest.

Ensuite on crée les countvectorizer comme précédemment ou sinon on appelle la fonction suivante si on veut supprimer tous les mots qui apparaissent moins de 10 fois.

```
count_vectorizer = TfidfVectorizer(min_df=10)
```

Enfin on sépare le corpus d'entraînement et de test, puis on récupère les résultats dans la même structure de donnée. Cette structure nous permet de calculer tous les metrics comme dans la partie précédente.



Enfin on peut déterminer la précision, si vous exécuter le fichier classification.py, on obtient le résultat suivant :

| | **Avec fréquence Naive Bayes** | **Avec Fréquence Random Forest** | **Sans Fréquence Naive Bayes** | **Sans Fréquence Random Forest** |
|---|---|---|---|---|
| **Positif** | 0.81 | 0.74 | 0.80 | 0.75 |
| **Negatif** | 0.79 | 0.68 | 0.73 | 0.67 |

Encore une fois l’algorithme de classification, le plus performant est naïve bayes.