Techniques et applications du traitement de la langue naturelle

TP numéro 2

Auteur: de Boisvilliers Christopher
Numero: CHDEB35

Université de Laval

22 novembre 2017

Table des matières

1 Introduction

Le travail de se projet à était fait à 2, avec Pierre VALENTIN. Nous avons tous deux des parties communes. Les notions et les réflexions pour les comprendre de ce projet on était comprise par nous deux.

— Exercice 1 : Outil Choisi pour construire le sytème : *ElasticSearch*

— Exercice 2 : Choix des classifieurs *Naives Bayes* et *Random Forest*

— Exercice 3 : Idem pour les choix des classifieurs

J'ai divisé le code en "partie" noté en commentaire pour pouvoir l'expliquer pour chacun des exercices.

2 Recherche d'information – quels pays... ?

Cet exercice à était divisé en deux fichiers, *"exo1.py"* et *"exo1-2.py"*. Le premier fichier sert à construire mon ensemble d'index, en utilisant d'expression régulière afin de récupérer des informations dans le world factbook. Ce fichier comporte deux fonctions :

1. *"indexPays()"* : Me permet de créer le document indexpays si inexistant qui contiendra l'ensemble de mes index sur les pays du world factbook.
2. *"get_ wfb_ info()"* : Est la fonction maîtresse pour créée l'ensemble des mes index qui me serviras pour construire le système.

Dans exo1.py :

Partie 0 : La fonction *"indexPays()"* me permet juste de vérifier si le document indexpays est présent ou non, sinon alors il est créé.

Partie 1 : Fonction *"get_ wb_ info()"*, dans cette partie je vais mettre dans la liste listePays tous les noms de fichier des pays du world Factbook, pour ce faire j'utilise une Regex avec la fonction *"findall()"* qui me permet de trouver toutes les expressions avec le même schéma de regex.

Partie 2 : Je me déplace dans le document *"factbook/geos"* ou se trouve l'ensemble de mes fichiers html du world factbook, sachant que j'ai stocké le nom de mes fichiers dans une liste j'utilise une boucle *for* pour lire à chaque itération un nom de fichiers html . Une fois le fichier ouvert je cherche le nom du pays avec une regex que je stocke par la suite dans *"nomPays"*, après cela je créé le fichier *"****.json"* correspondant dans le dossier *"/indexpays"* par la suite j'écris directement dans ce fichier le nom de ce dernier.
À ce stade mon fichier *"json"* ressemble à cela :

```
{
  "name": "Afghanistan"

}
```

Partie 3 : J'ai remarqué qu'a certain moment des informations manquées lors de ma recherche d'information avec mes regex, j'ai donc utilisé un *"try"* et *"except AttributeError"* dans le cas ou je n'obtiens pas de valeur, j'écris tout simplement que ma clé est vide en y mettant "".
Successivement j'obtiens la construction suivante :

```
{
  "name": "Afghanistan",
   "Background":"Ahmad ... Government.",
}
```

```
{
  "name": "Afghanistan",
   "Background":"Ahmad ... Government.",
   "Geography-note":"landlocked; ... (Wakhan Corridor)",
```

```
}
```

```
{
  "name": "Afghanistan",
   "Background":"Ahmad ... Government.",
   "Geography-note":"landlocked; ... (Wakhan Corridor)",
   "Economy-overview":"Afghanistan's ... 2013."
}
```

(les "..." ne sont pas présents dans les fichiers *"****.json"* finaux, je les ai mis, car les valeurs peuvent être longues)
Le fichier final sera stocké dans le dossier *"indexpays"* avec le nom *"Afghanistan.json"*, la boucle *"for"* me permet de faire la même démarche pour tous les autres pays du world factbook et donc de construire mon ensemble d'index.

Dans *"exo1-2.py"* :

Partie 0 : Connexion au serveur ElasticSearch

Partie 1 : La fonction *"listJson()"* me permet de lister tout les fichiers json dans le répertoire courant et les stocks dans la variable *"listfile"* sous forme de liste que je retourne.

Partie 2 : La fonction *"constructIndex()"* me permet dans un premier temps de créer l'index *"indexpays"* dans la base de données, par la suite je stocke la liste que me crée la fonction *"listJson()"* dans une variable que par la suite je trie pour l'avoir dans un ordre alphabétique. Enfin une boucle *"for"* me permet d'ouvrir chaque fichier en parcourant la liste et à chaque itération je la mets sur le serveur dans l'index *"indexpays"*.

Partie 3 : La fonction *"queryElastic(query)"* est une fonction qui prend en entrée une query (par la suite la query sera la liste des questions dans le fichier *"jugements.txt"*, prise une par une.) et retourne la question que l'on posera plus tard au serveur ElasticSearch. Le choix à était de matcher la query, avec un opérateur *"or"*. C'est à dire qu'entre chaque mot de la query il y auras un *"or"*, le problème ici et que l'on ne veut pas que la phrase *"European countries with debt crisis"* renvoie tous les résultats comportant le seul mot *"with"* (trop de résultats) donc pour pallier ce problème l'utilisation du paramètre *"minimum_should_match"* était nécessaire, car il permet de dire que nous souhaitons que 79% des clauses match au minimum et donc d'avoir une plus grande précision lors de notre recherche.

Partie 4 : La fonction *"resultPrediction()"* permet dans un premier temps de chercher l'ensemble des questions dans le fichier *"liste_requetes.txt"* grâce à une regex que je stocke par la suite dans une variable qui sera une liste de String. Ensuite dans une première boucle *"for"* je pose chacune des questions au serveur l'une après l'autre. Pour chaque question je récupère les résultats que je stocke dans la même String les uns à la suite des autres. Finalement je retourne cette String qui sera dans la même forme que le fichier *"jugements.txt"*.

Partie 5 : La fonction *"writeJugementsPrediction(listeOfPrediction)"* permet tout simplement d'écrire un String dans un fichier texte.

Partie 6 : La fonction *"listOfJugements(txtjugements)"* prend un fichier texte dans le même format que le fichier *"jugements.txt"* et retourne une liste de tuples contenant le numéro de la question (Q1,Q2, etc.) et le nom du pays obtenu.

Partie 7 : *"appendTab(tab,precision)"* permet juste d'ajouter une valeur à un tableau.

Partie 8 : *"moyTab(tab)"* permet de retourner la moyenne d'un tableau.

Partie 9 : *"averagePrecision(txtjugements,txtjugementsPrediction)* est la fonction la plus importante elle permet de calculer l'average precision pour chaque question, elle prend en entrée un fichier de réponse exact tel que *"jugements.txt"* et un fichier de prédiction construit de la façons que *"jugements.txt"*.
Note : nous avons eu beaucoup de mal à comprendre l'average précision, nous avions à la base fait simplement une fonction qui faisait la moyenne des précision de l'ensemble des question et retourner cette valeur comme étant la valeur du MAP, nous avons compris par la suite que la MAP est une mesure de performance qui permet d'évaluer la performance d'un système recherche.

Je vais décrire le rôle de chaque variable :

- *"tableAverage"* : est un tableau qui contiendra toutes les valeurs des bonnes prédictions par question.
- *"resultMoyPrediction"* : est le tableau qui contiendra l'ensemble des moyennes des bonnes prédictions de chaque question.
- *"jugements" et "jugementsPrediction"* : sont deux listes qui contiennent respectivement la liste des bonnes réponses et la liste des prédictions.
- *"denominateur et numerateur"* : sont deux variables que j'incrémenterais d'un (je l'explique juste après)
- *"indexQuestions"* : est une variable de type String elle me permettra de savoir ou j'en suis dans mes questions.

Je vais maintenant décrire le principe de l'algorithme :

1. J'effectue une boucle *"for"* avec un nombre d'itérations égal au nombre de pays dans mon ensemble de prédiction.
2. Pour chaque itération je teste d'abord si ma variable *"indexQuestions"* est la même que la question étudiée, je récupère cette information avec *"jugementsPrediction[i][0]"* qui est un tuple (numéro de question , valeur)
3. Si ce n'est pas le cas je sais donc que je suis passé à la question suivante et donc que mon tableau *"tableAverage"* contient l'ensemble des valeurs des précisions des questions qui ont était bien prédites donc dans l'ordre :
   - j'ajoute la moyenne de *"tableAverage"* dans *"resultMoyPrediction"*
   - je réinitialise mon tableau *"tableAverage"*
   - je réinitialise les variables *"numerateur,denominateur"* à 0

— je mets à jour mon *"indexQuestions"*

Toujours dans la boucle je test si mon tuple prédit appartient à la liste des tuples des bonnes réponses si c'est le cas dans l'ordre :

1. J'ajoute 1 à mon numérateur et au dénominateur
2. Je calcul *"resultPrediction"*
3. J'ajoute *"resultPrediction"* à mon tableau *"tableAverage"*

Dans le cas ou mon tuple prédit n'appartient pas à la liste des tuples des bonnes réponses j'ajoute 1 à mon dénominateur.

Je retourne *"resultMoyPrediction"* qui contient l'ensemble des *"average precision"* pour toutes mes questions.

Partie 10 : La fonction *"MAP(averagePrecision"* est juste la pour calculer la moyenne d'un tableau et ici calcul le MAP.

Pour conclure, le MAP obtenu est de 0.41 qui n'est certes pas une très bonne valeur, mais nous aurions pu certainement améliorer ce résultat par exemple en classifiant les questions pour ensuite la poser à un champ souhaité (économie, géographie, etc.).

Pour l'exécution : Il faut dans un premier temps lancer le serveur ElasticSearch puis exécuter exo1.py et à la suite exo1-2.py

3 Classification de questions

L'exercice 2 à était assez simple à traité vu que les questions du fichier questions.txt avaient déjà subit un pré traitement, mais nous as permis de mieux comprendre la librairie sklearn et son utilisation. L'exercice 2 contient parties : exo2.py et exo2-naive.py. Pour la partie exo2.py elle contient deux fonctions :

1. createListe(i) : me permet de lire le fichier question.txt pour par la suite avec deux regex récupérer l'ensemble des classes dans la première et l'ensemble des questions associé aux classes dans la deuxième. Je stocke ces deux éléments dans un tableau que je retourne par la suite.
2. constructcsv() : est une fonction qui me permet de construire un fichier csv que je réutiliserais par la suite pour construire une dataFrame. Ce fichier csv contiendra l'ensemble des questions dans la colonne question et l'ensemble des classes associées dans la colonne classe.

Pour la partie exo2-naives.py :

Partie 0 : La fonction choiceClassifier(i) permet en fonction de la valeur de i, de choisir le classifier RandomForest ou alors NaiveBayes.

Partie 1 :La fonction construCsv() vérifie la présence ou l'absence de fichier csv dans mon répertoire courant si c'est le cas alors j'appelle la fonction constructCsv() pour le construire.

Partie 2 :Contient la fonction constructModel(), qui prend en paramètre deux variables cc et j, est la fonction maîtresse du programme qui va permettre la classification d'un modèle test. La variable cc permet de définir à partir de quelle ligne je veux démarrer l'ensemble test (par exemple si nous avons un ensemble test de 300 lignes et que cc vaut 200 alors les 100 lignes restantes seront notre ensemble test qui ne participera pas à la construction de notre modèle qui lui sera les 200 premières lignes), j permets tout simplement de nous faire choisir le classifier par la fonction choiceClassifier().
Le résultat de cette fonction sera la variable result qui est un tableau contenant les classes prédites, le vecteur de classe test (les vraies valeurs), examples qui contiennent notre vecteur de question de l'ensemble test et enfin la variable j. Dans cette fonction le but est de construire un modèle qui nous servira tout au long du programme, pour ce faire il faut dans un premier temps créer une dataframe, puis définir dans ce dernier les ensembles de classe et de valeur.
À ce stade il nous reste plus qu'a déterminé notre ensemble à tester puis construire le modèle et pour faire cela nous utilisons un count_vectorizer sur l'ensemble des questions d'entraînement et nous fittons les classes correspondantes à ce vecteur pour un classifier donné. Une fois le modèle d'entraînement construit nous lançons la prédiction et récupérons le résultat dans prédictions pour avoir le vecteur de classe test prédite pour les questions associées. Partie 3 : La fonction construcTableRP qui prend en entré une liste de prédictions et une liste de vraie classe associée permet de construire une structure de donnée qui nous donne :
{'2' : {'class' : 'LOCATION', 'bool' : False} si à la même itérations les deux liste ne contiennent pas la même classe 'bool' :True sinon.

Partie 4 : Contiens toutes les fonctions de mesure de performance telle que : C'est

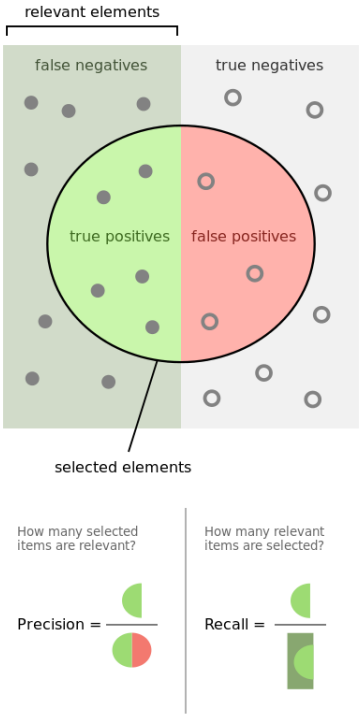

FIGURE 1 – Valeurs calculées

dans cette partie que nous serviras la tableRP qu'on utilise dans presque toutes les fonctions, du truePositive jusqu'au falseNegative leurs principes sont assez similaire, on leur donne en entré une classe, on vérifie dans la tableRP les conditions souhaitées et on incrémente une valeur que l'on return.

Partie 5 : la fonction general et result permettent simplement d'afficher les résultats à la console pour un modèle, une tableRP et une classe donnée.

Les résultats obtenues sont les suivants :

Pour Naives Bayes :
Pour la classe DEFINITION et un modèle Naives Bayes la précision est de :
0.8727272727272727
Pour un rappel de :
0.9230769230769231
Pour la classe QUANTITY et un modèle Naives Bayes la précision est de :
0.875
Pour un rappel de :

0.7
Pour la classe LOCATION et un modèle Naives Bayes la précision est de :
0.71875
Pour un rappel de :
0.8518518518518519
Pour la classe TEMPORAL et un modèle Naives Bayes la précision est de :
1.0
Pour un rappel de :
0.7894736842105263
Pour la classe PERSON et un modèle Naives Bayes la précision est de :
0.9047619047619048
Pour un rappel de :
0.9047619047619048

Pour Random Forest
Pour la classe DEFINITION et un modèle Random Forest la précision est de :
0.847457627118644
Pour un rappel de :
0.9615384615384616
Pour la classe QUANTITY et un modèle Random Forest la précision est de :
0.7894736842105263
Pour un rappel de :
0.75
Pour la classe LOCATION et un modèle Random Forest la précision est de :
0.7727272727272727
Pour un rappel de :
0.6296296296296297
Pour la classe TEMPORAL et un modèle Random Forest la précision est de :
1.0
Pour un rappel de :
0.8421052631578947
Pour la classe PERSON et un modèle Random Forest la précision est de :
0.8695652173913043
Pour un rappel de :
0.9523809523809523

Vu que nous ne pouvons pas jouer sur le prétraitement des questions (suppression de fréquence de mot) les résultats sont fixes pour la classification, mais on peut remarquer que dans l'ensemble la classification de l'ensemble TEST pour les deux classifier est bonne, avec des résultats presque tout le temps au-dessus de 70% donc notre classifier fait bien sont travaillé.
Nous pouvons jouer sur la valeur de cc afin de modifier les ensembles test et d'entraînements et on constate que les résultats sont toujours "corrects" pour une valeur de cc=15 avec Naives Bayes, mais sont un peu plus catastrophiques pour la même valeur de cc sur RandomForest.
Pour exécuter le logiciel il suffit d'exécuter le fichier exo2-naive.py.

4 Classification - Analyse de sentiments

Cette partie contient de fichier python, Exo3.py et Exo-3Classification.py, je vais dans un premier temps éxpliquer chacune des fonctions :

Partie 0 : La fonction stopWordsList() prend en entré un fichier text contenant un ensemble de stopWords qui sont séparé par un espace ou une virgule est renvoie une liste de ces stopwords.

Partie 1 : La fonction createDicoClasse() renvoie une structure de donnée de contenant l'ensemble des textes à traiter et à classer sur les reviews de livre. La structure de donnée est de la forme suivante :

```
{
                'Class': 'Negatif',
                'Text': "texte",
                'nbStdelete': 0
            }
```

Pour un texte qui est dans le dossier neg_Bk et de la forme :

```
{
                'Class': 'Positif',
                'Text': 'Text',
                'nbStdelete': 0
            }
```

Pour un texte qui est dans le dossier pos_Bk.
La clé de chacune des valeurs précédentes est un unique entier croissant.
Nous avons fait le choix de cette structure pour pouvoir par la suite accéder plus facilement à l'ensemble des textes pour les prétraitements avant la classification.
Le logiciel fonctionne comme un ensemble de pipelines par lequel le texte doit passer par être traité dans l'ordre il pas par :

1. Conversion en minuscule avec lowercaseConvert() qui parcours toutes les clés du dico et change l'ensemble des textes en minuscule.

2. Suppression de stop Word avec la fonction stopWords() qui utilise le dictionnaire en minuscule pour dans un premier temps créé une liste de mot par texte, puis les supprimes s'il appartient a la liste des stops Word et reconstruit ensuite la phrase.

3. Ensuite on supprime la ponctuation de la même façon que les stops Word avec la fonction RegexpTokenizer() qui nous permet de faire cela.

4. On passe le dictionnaire précédent a la fonction stemmingDico() qui grâce à la librairie nltk nous permet d'appliquer facilement le stemming sur notre ensemble de texte, mot par mot. À chaque itération nous tokenizons par mots pour ensuite appliquer la fonction de stemmings sur celui-ci et enfin reconstruire la phrase que l'on stocke à nouveau dans le dictionnaire.

5. La fonction de suppression de mot par la technique du Pos-Tagging est très longue lors de son exécution (une 20aine de minutes) c'est pour cela que j'ai écrit son résultat dans deux fichiers "neg.json" et "pos.json". Il y a deux étapes pour la suppression par Pos-Tagging. La première avec la fonction correctStringPosttag() permet de lire une phrase et de renvoyer la même phrase avec les tags correspondants supprimés. La deuxième avec la fonction postDico() permet de répéter l'opération précédente sur chacun des textes de notre dictionnaire issu du stemming.

Les 2 dernières fonctions permettent de créer le fichier csv "forClassification.csv" qui sera de la même forme qu'à l'exercice 2, classe et texte correspondants, cela permet de faciliter la création de la dataframe. Le dernier traitement qu'il reste à faire et celui de la suppression de mot dont la fréquence dépasse une certaine valeur.
passons au fichier **Exo-3Classification.py** :
Je ne vais pas trop m'attarder pour expliquer les algorithmes de cette partie, car ils sont quasi identiques à ceux de l'exercice 2.
Pour pouvoir retirer les fréquences de mots supérieurs à une valeur, nous avons utilisé dans la fonction constructModel() un TfidfVectorizer() qui prend en paramètre dans notre cas "max_df" qui permet d'enlever les termes avec une fréquence supérieure à la valeur qui lui est donnée.
Pour la suite des fonctions je ne vais pas les expliquer elles sont identiques à celles de l'exercice 2, le plus important maintenant et de jouer sur la modification de certains pipelines pour voir si la classification se fait mieux ou non.
Les tests que nous avons menés se font exclusivement sur la mesure de la précision lors du changement du paramètre "max_df" pour Naives Bayes et Random Forest.
voici les 2 graphes obtenus :

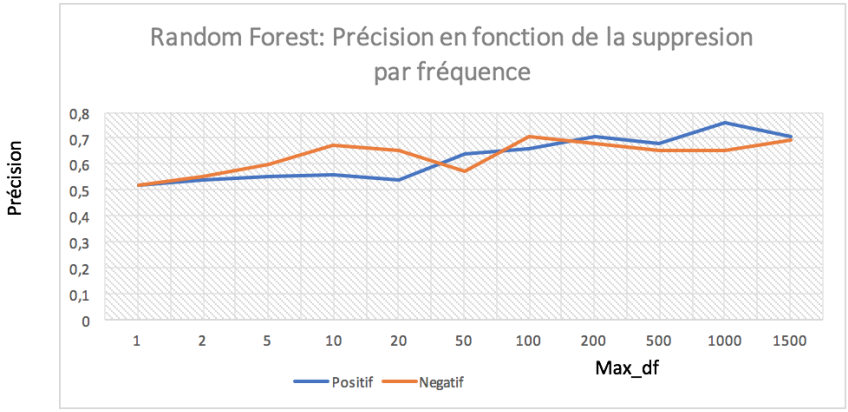

Figure 2 – Précision en fonction du paramètre max_df

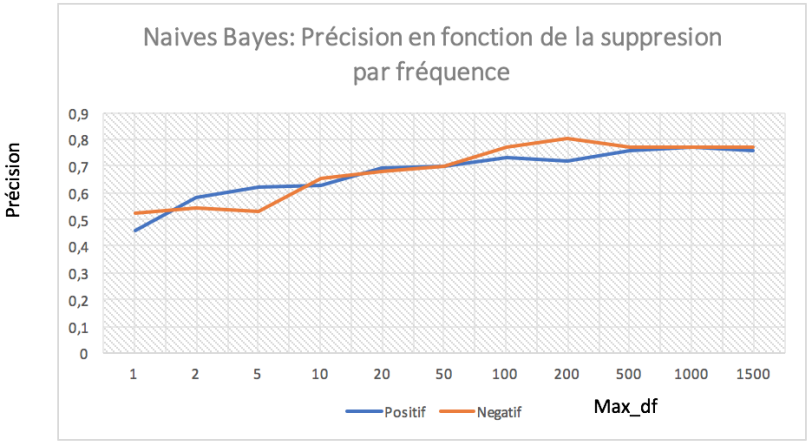

FIGURE 3 – Précision en fonction du paramètre max_df

On peut voir dans ces graphiques que :

1. Le modèle de Naives Bayes à était un peu plus efficace que random forest pour classer nos reviews.

2. On pourrait aussi penser qu'après tout les prétraitements fait avant la suppression par fréquence (Stops Word, Stemmings etc.) la méthode de suppression par fréquence max n'est pas très efficace, car l'on peut remarquer que dans les deux cas la courbe à tendance à augmenter en précision lorsque l'on augmente le max_df. Cela signifie je pense que les mots restant après l'ensemble des pipelines sont des mots importants et qui on du sens pour pouvoir déterminer la classe d'une reviews.

Pour l'exécution de notre programme il de lancer le fichier Exo-3Classifications.py, si vous voulez vérifier que chacune des fonctions font ce quelles sont censé faire il suffit d'enlever d'exécuter les commentaires en dessous de chacune des fonctions un à un.