Classifier du texte

March 19, 2018

1 TP - Classifier du texte

1.1 Importation des bibliothèques

In [1]:
```
# -*- coding: utf-8 -*-

        from sklearn.datasets import fetch_rcv1
        from sklearn.model_selection import train_test_split

        from sklearn.model_selection import GridSearchCV

        from sklearn.tree import DecisionTreeClassifier
        from sklearn.neighbors import KNeighborsClassifier
        from sklearn.ensemble import RandomForestClassifier

        from sklearn.multiclass import OneVsRestClassifier
        from sklearn.svm import SVC
        from sklearn import preprocessing

        import pandas as pd
        import numpy as np
```

1.2 Fonction d'importation des données Reuters

In [2]:
```
        # Fonction pour importer les données depuis la fonction packagée fecth_rcv1
        def Importer_Donnees_Reuters(Action, nombreLignes):
            # On exécute l'action qu'une seule fois pour préserver les ressources mémoire
            if(Action):
                data = fetch_rcv1(
                    data_home="E:\\Data\\RawData\\Exploit_data_texte\\Reuters",
                    subset='train')

                Index = [data.sample_id]
                Labels = [data.target_names]

                # On réduit les datasets à 300 lignes en raison de ressources mémoire limitées
                Features = pd.DataFrame(data.data.A)
                Features_Slice = Features.iloc[:nombreLignes,:]
```

```
    Classifications = pd.DataFrame(data.target.A,columns=Labels)
    Classifications_Slice = Classifications.iloc[:nombreLignes,:]

    # On libère la mémoire en supprimant les variables contenant les données sourc
    del data
    del Features
    del Classifications

    # On exporte les 2 datasets vers des fichiers CSV enregistrés en local
    Features_Slice.to_csv('Features')
    Classifications_Slice.to_csv('Classifications')
```

1.3 Création du jeu de données d'apprentissage et de test

```
In [3]: # Création dujeu de test d'apprentissage et de test
        def Construction_Jeu_Donnees():
            # On importe les données depuis des fichiers plats stockés en local pour éviter le
            Classifications_Slice = pd.read_csv("E:\Data\RawData\Exploit_data_texte\Reuters\Cl
            Features_Slice = pd.read_csv("E:\Data\RawData\Exploit_data_texte\Reuters\Features"

            # On expurge le libellé des index insérés par la fonction d'export en CSV
            Features = Features_Slice.iloc[:,1:]
            Classifications = Classifications_Slice.iloc[:,1:]

            # On construit les jeux de données de test et de training
            Features_Train, Features_Test, Labels_Train, Labels_Test = train_test_split(
                                                                    Features,
                                                                    Classifications,
                                                                    test_size=0.30)

            return Features_Train,Features_Test,Labels_Train,Labels_Test
```

1.4 Création d'un classifieur de type arbre de décision

```
In [4]: # Création d'un classifieur de type Arbre de décision
        def Classifieur_Arbre_Decision(Features_Train,Features_Test,Labels_Train,Labels_Test):
            # Classification Simple
            Classification = DecisionTreeClassifier(random_state=0)

            Classification.fit(Features_Train,Labels_Train)
            Score = Classification.score(Features_Test,Labels_Test)
            print("Le score lambda de la classification par arbre de décision est de : {} ".fo

            # Classification avec validation croisée des différents paramètres
            parameters = {"max_depth": range(3,20),
                          "random_state":[0]}
```

```
        grid_obj = GridSearchCV(
                estimator=Classification,
                param_grid=parameters)

        grid_fit =grid_obj.fit(Features_Train,Labels_Train)
        print("Les paramètres optimums sont : {} ".format(grid_fit.best_params_))

        ScoreCV = grid_fit.score(Features_Test,Labels_Test)
        print("Le score de la classification avec validation croisée par arbre de décision
```

1.5 Création d'un classifieur de type KNeighbors

```
In [5]: # Création d'un classifieur de type KNeighbors
        def Classifieur_KNeighbors(Features_Train,Features_Test,Labels_Train,Labels_Test):
            # Classification Simple
            Classification = KNeighborsClassifier(n_neighbors=3)

            Classification.fit(Features_Train,Labels_Train)
            Score = Classification.score(Features_Test,Labels_Test)
            print("Le score lambda de la classification par KNeighbors est de : {} ".format(Sc

            # Classification avec validation croisée des différents paramètres
            parameters = {"n_neighbors": range(2,10),
                          "weights":['uniform','distance'],
                          "algorithm":['ball_tree','kd_tree','brute','auto'],
                          "p":[1,2]}

            grid_obj = GridSearchCV(
                    estimator=Classification,
                    param_grid=parameters)

            grid_fit =grid_obj.fit(Features_Train,Labels_Train)
            print("Les paramètres optimums sont : {} ".format(grid_fit.best_params_))

            ScoreCV = grid_fit.score(Features_Test,Labels_Test)
            print("Le score de la classification avec validation croisée par KNeighbors est de
```

1.6 Création d'un classifieur de type Random Forest

```
In [6]: # Création d'un classifieur de type Random Forest
        def Classifieur_Random_Forest(Features_Train,Features_Test,Labels_Train,Labels_Test):
            # Classification Simple
            Classification = RandomForestClassifier(bootstrap=True,
                                                    class_weight=None,
                                                    criterion='gini',
                                                    max_depth=2,
                                                    random_state=0)
```

```
    Classification.fit(Features_Train,Labels_Train)
    Score = Classification.score(Features_Test,Labels_Test)
    print("Le score lamda de la classification par Random Forest est de : {} ".format(S

    # Classification avec validation croisée des différents paramètres
    parameters = {"n_estimators": range(5,15),
                  "criterion":['gini','entropy'],
                  "max_depth": range(3,10),
                  "bootstrap":[True,False],
                  "random_state":[0]}

    grid_obj = GridSearchCV(
            estimator=Classification,
            param_grid=parameters)

    grid_fit =grid_obj.fit(Features_Train,Labels_Train)
    print("Les paramètres optimums sont : {} ".format(grid_fit.best_params_))

    ScoreCV = grid_fit.score(Features_Test,Labels_Test)
    print("Le score de la classification avec validation croisée par Random Forest est
```

1.7 Fonction de transformation d'un vecteur multi-labels binarisé en nombre base 10

```
In [7]: # Fonction permettant de transformer une sortie multi_labels binaires en sortie encodé
        def Transformation_Base_2_Vers_Base_10(Octets):
            valeur_base_10 = []

            for octet in Octets.iterrows():
                valeur_octet = 0
                position_bit = 0

                for bit in octet[1]:
                    valeur_octet += (2**position_bit)*(int(bit))
                    position_bit += 1

                valeur_base_10.append(valeur_octet)

            return valeur_base_10
```

1.8 Fonction de création d'un classifieur One Versus Rest (estimateur non multi-labels)

```
In [8]: # Création d'un classifieur de type Random Forest
        def Classifieur_One_Versus_All(Features_Train,Features_Test,Labels_Train,Labels_Test):
            # Classification Simple
            Classification = OneVsRestClassifier(
                                    SVC(
```

```
                                        C=10,
                                        kernel="linear",
                                        degree=1,
                                        probability =True,
                                        max_iter=-1))

        Classification.fit(Features_Train,Labels_Train)
        Score = Classification.score(Features_Test,Labels_Test)
        print("Le score lamda de la classification One Versus Rest est de : {} ".format(Sc

        # Classification avec validation croisée des différents paramètres
        parameters = {"C": np.arange(1,10,1),
                      "kernel":["linear","rbf","sigmoid"],
                      "degree": range(1,5)}

        grid_obj = GridSearchCV(
                estimator=Classification,
                param_grid=parameters)

        grid_fit =grid_obj.fit(Features_Train,Labels_Train)
        print("Les paramètres optimums sont : {} ".format(grid_fit.best_params_))

        ScoreCV = grid_fit.score(Features_Test,Labels_Test)
        print("Le score de la classification avec validation croisée par One Versus Rest es
```

1.9 Programme principal

```
In [9]: # Constitution des jeux de données
        Importer_Donnees_Reuters(False,300)

        Features_Train,Features_Test,Labels_Train,Labels_Test = Construction_Jeu_Donnees()

In [10]: # On va choisir 3 types de classifieurs distints

         # A noter que le jeu de données nous contraint dans le choix des types de classifieur
         # Nous devons choisir un classifieur acceptant des sorties multi labels
         # En effet, un document peut avoir plusieurs labels d'apartenance en sortie

         print("1 - Classifieur Arbre de décision")
         Classifieur_Arbre_Decision(Features_Train,Features_Test,Labels_Train,Labels_Test)
```

```
1 - Classifieur Arbre de décision
Le score lambda de la classification par arbre de décision est de : 0.25555555555555554
Les paramètres optimums sont : {'max_depth': 15, 'random_state': 0}
Le score de la classification avec validation croisée par arbre de décision est de : 0.2333333
```

```
In [11]: print("2 - Classifieur KNeighbors")
         Classifieur_KNeighbors(Features_Train,Features_Test,Labels_Train,Labels_Test)
```

```
2 - Classifieur KNeighbors
Le score lambda de la classification par KNeighbors est de : 0.28888888888888886
Les paramètres optimums sont : {'algorithm': 'ball_tree', 'n_neighbors': 2, 'p': 2, 'weights':
Le score de la classification avec validation croisée par KNeighbors est de : 0.24444444444444
```

```
In [ ]: # Sans surprise, le classificateur des Random Forest s'avère très long... même sur un
        # En effet, le Random Forest tentant de réduire le nombre de Features à chaque itérati
        # possibles sur plus de 40 000 Features rendent l'opération très coûteuse en ressource
```

```
In [12]: print("3 - Classifieur Random Forest")
         Classifieur_Random_Forest(Features_Train,Features_Test,Labels_Train,Labels_Test)
```

```
3 - Classifieur Random Forest
Le score lamda de la classification par Random Forest est de : 0.0
Les paramètres optimums sont : {'bootstrap': False, 'criterion': 'entropy', 'max_depth': 9, 'n
Le score de la classification avec validation croisée par Random Forest est de : 0.022222222222
```

```
In [ ]: # Les scores obtenus ne sont pas pertinents dans la mesure où il serait nécessaire d'e
        # des dizaines de milliers de samples de la base de training pour avoir une classifica
        # que le nombre de features en entrée dépasse les 40 000 unités !!!

        # Les ressources de notre ordinateur s'avèrent trop faibles pour réaliser ce traitemen
        # On se contente donc de livrer le code opérationnel d'un traitement qu'il faudrait ex
        ## Amazon ou Azure par exemple.
```

1.10 Analyse complémentaire

```
In [ ]: # La sortie en prédiction du jeu de données est un vecteur à 134 dimensions
        # C'est à dire que la prédiction de la classification doit être multi labels (un sampl
        # De ce fait,cela limite le choix de classificateurs possibles...

        # Pour s'affranchir de cette limite, on va tenter de changer de base le vecteur de rés
        # Le vecteur de sortie étant un tuple de valeurs binaires (0 ou 1)
        # On peut essayer de transformer ce tuple binaire en base 10, ce qui nous permettrait

        # Cette transformation réalisée, on pourra alors choisir un classificateur ne supporta
        # Nous allons effectuer un essai avec un classificateur de type One Versus Rest...
```

```
In [13]: print("4 - Classifieur Régression logistique")

         # On initialise un encodage pour s'affranchir des avertissements sur les labels absen
         le = preprocessing.LabelEncoder()

         # Transformation problème multi-classes et multi-labels en problème multi-classes mai
         # Pour cela, on transforme le vecteur binaire en valeur entière en base 10

         Labels_Train_Binarisee = le.fit_transform(Transformation_Base_2_Vers_Base_10(Labels_Tr
         Labels_Test_Binarisee = le.fit_transform(Transformation_Base_2_Vers_Base_10(Labels_Tes
```

4 - Classifieur Régression logistique

In [15]:
```
# Cela nous permet alors d'utiliser d'autres types de classifieurs
Classifieur_One_Versus_All(Features_Train,Features_Test,Labels_Train_Binarisee,Labels_
```

Le score lamda de la classification One Versus Rest est de : 0.011111111111111112

In [ ]:
```
# Au final, le résultat ne semble pas particulièrement probant...
# De nombreuses difficultés viennent entraver cette méthode (classe trop peu peuplée,

# Cette manipulation ne permet pas de s'affranchir de l'important besoin en ressources
# un dataset aussi exigeant (nombre très important de features) !!
```