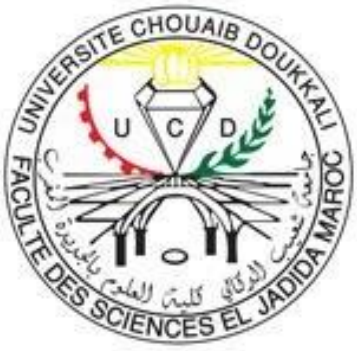


Université Chouaïb Doukkali

Faculté des Sciences, Département d'informatique

ELJADIDA

Master BIBDA

Thème

Programmation avancée : Python
Projet de Fin de module : Feature selection using TextRank

RÉALISÉ PAR : DOUAIOUI HAMZA

Année universitaire : 2020/2021

INTRODUCTION

L'objectif de ce projet est d'utiliser l'algorithme TextRank pour sélectionner les meilleurs features d'un dataset. TextRank est un algorithme basé sur PageRank, qui est souvent utilisé dans l'extraction de mots clés et la synthèse de texte.

Dans ce rapport, nous allons essayer de développer un programme d'apprentissage automatique pour identifier quand un article peut être une fausse nouvelle en utilisant l'algorithme TextRank.

Alors pour réaliser ce travail nous allons suivre les étapes suivantes :

Etape 1 : Du texte au graphe.

Etape 2 : Classification des textes.

Etape 3 : Résultats de simulation.

Le Dataset considéré

Le dataset a été récupéré du site kaggle.com. Vous pouvez le télécharger à partir de l'adresse https://www.kaggle.com/c/fake-news/data. Le dataset est composé de deux fichiers csv, l'un pour l'entraînement (train.csv) et l'autre pour le test (test.csv). La description de ces fichiers (tirée du site) est donnée ci-dessous :

train.csv: A full training dataset with the following attributes:

id: unique id for a news article

title: the title of a news article

author: author of the news article

text: the text of the article; could be incomplete

label: a label that marks the article as potentially unreliable

1: unreliable

0: reliable

test.csv: A testing training dataset with all the same attributes at train.csv without the label.

Etape 1 : Du texte au graphe

Dans cette étape nous allons essayer de suivre l'enchainement suivant :
Text —> preprocessing —> graphe —> TextRank —> {tokens avec poids}

Le choix de la colonne.

Nous allons choisir de travailler sur la colonne 'Text' et la colonne 'Tilte', donc on va concaténer leur contenu dans une nouvelle colonne, que nous appellerons 'Content'.

```
#DATA PRE-PROCESSING
#loading the dataset to a pandas DataFrame
def Data_PreProcessing():
    news_dataset = pd.read_csv('train.csv')
    # replacing the null values with empty string
    news_dataset=news_dataset.fillna('')
    #merging the text and the news title
    news_dataset['content']=news_dataset['title']+' '+news_dataset['text']
    return news_dataset
```

Le preprocessing.

Les étapes de base du prétraitement de texte. Ont pour objectif le transférer du texte du langage humain vers un format lisible par machine pour un traitement ultérieur.
Nous commençons par la normalisation du texte. La normalisation du texte comprend :
La convertion de toutes les lettres en minuscules, la suppression des nombres, la suppression les signes de ponctuation, les accents et autres signes diacritiques, la suppression des espaces blancs, La tokenisation est le processus de division du texte donné en morceaux plus petits appelés jetons. Les mots, les chiffres, les signes de ponctuation et autres peuvent être considérés comme des jetons. Supprimer les mots vides, « Stop words » sont les mots les plus courants dans une langue comme « the », « a », « on », « is », « all ». Ces mots n'ont pas de sens important et sont généralement retirés des textes. Il est possible de supprimer les mots vides à l'aide de Natural Language Toolkit (NLTK), une suite de bibliothèques et de programmes pour le traitement symbolique et statistique du langage naturel. Après on passe à la lemmatisation, « Stemming »
La lemmatisation est un processus de réduction des mots à leur racine de mot, à leur forme de base ou à leur racine (par exemple, books - book, looked - look). Pour cela nous allons utiliser l'algorithme Porter (supprime les terminaisons morphologiques et flexionnelles communes des mots).

```
port_stem=PorterStemmer()
def preProcessing(text):
    text_preprocessed=re.sub('[^a-zA-Z]',' ', text)
    text_preprocessed=text_preprocessed.lower()
    text_preprocessed=text_preprocessed.split()
    text_preprocessed=[port_stem.stem(word) for word in text_preprocessed if not word in stopwords.words('english')]
    text_preprocessed=' '.join(text_preprocessed)
    return text_preprocessed
```

Le graphe.

Dans cette étape nous allons utiliser l'algorithme text Rank, pour construire le graphe et obtenir chaque mot avec son poids,

Nous divisons un document en plusieurs phrases et nous stockons tous les mots, chaque mot est un nœud dans le PageRank.

Nous définissons la taille de la fenêtre comme k.
[w1, w2, …, w_k], [w2, w3, …, w_{k+1}], [w3, w4, …, w_{k+2}] , sont des fenêtres. Toute paire de deux mots dans une fenêtre est considérée comme ayant un bord non orienté.
Nous [time, Wandering, Earth, feels, throwback, eras, filmmaking] comme exemple, et définissons la taille de la fenêtre k=4, nous obtenons donc 4 fenêtres, [time, Wandering, Earth, feels], [Wandering, Earth, feels, throwback], [Earth, feels, throwback, eras], [feels, throwback, eras, filmmaking].
Pour la fenêtre [time, Wandering, Earth, feels] toute paire de mots a un bord non orienté. Nous obtenons donc (time, Wandering), (time, Earth), (time, feels), (Wandering, Earth), (Wandering, feels), (Earth, feels)

```
def sentence_segment(self, doc):
    sentences = []
    for sent in doc.sents:
        selected_words = []
        for token in sent:
            selected_words.append(token.text)
        sentences.append(selected_words)
    return sentences
```

```
def get_token_pairs(self, window_size, sentences):
    token_pairs = list()
    for sentence in sentences:
        for i, word in enumerate(sentence):
            for j in range(i+1, i+window_size):
                if j >= len(sentence):
                    break
                pair = (word, sentence[j])
                if pair not in token_pairs:
                    token_pairs.append(pair)
    return token_pairs
```

Après, sur la base de ces tokens_pairs, nous pouvons construire notre graphe non orienté (chaque nœud correspond à un mot), qui est représenté par une matrice.

```
def get_vocab(self, sentences):
    vocab = OrderedDict()
    i = 0
    for sentence in sentences:
        for word in sentence:
            if word not in vocab:
                vocab[word] = i
                i += 1
    return vocab
```

```
def get_matrix(self, vocab, token_pairs):
    vocab_size = len(vocab)
    g = np.zeros((vocab_size, vocab_size), dtype='float')
    for word1, word2 in token_pairs:
        i, j = vocab[word1], vocab[word2]
        g[i][j] = 1
```

Ensuite on passe à l'étape de la normalisation de la matrice.

```
# Get Symmeric matrix,we use a Symmeric matrix to represent the inbound and outbound links "we use the undirected edge"
   g = self.symmetrize(g)

   # Normalize matrix by column
   norm = np.sum(g, axis=0)
   g_norm = np.divide(g, norm, where=norm!=0) # this is ignore the 0 element in norm
```

```
def symmetrize(self, a):
    return a + a.T - np.diag(a.diagonal())
```

Ensuite, Nous pouvons calculer le poids pour chaque nœud (mot).

```
  def analyze(self, text, window_size):
      doc = nlp(text)
      sentences = self.sentence_segment(doc)
      vocab = self.get_vocab(sentences)
      token_pairs = self.get_token_pairs(window_size, sentences)
      g = self.get_matrix(vocab, token_pairs)
      pr = np.array([1] * len(vocab))
      previous_pr = 0
      while True:
        pr = (1-self.d) + self.d * np.dot(g, pr)
        if abs(previous_pr - sum(pr))  < self.min_diff:
          break
        else:
          previous_pr = sum(pr)
      node_weight = dict()
      for word, index in vocab.items():
          node_weight[word] = pr[index]
      self.node_weight = node_weight
      return node_weight
```

Les mots les plus importants peuvent être utilisés comme mots-clés.

```
    def get_keywords(self, number):
        x=[]
        node_weight = OrderedDict(sorted(self.node_weight.items(), key=lambda t: t[1], reverse=True))
        c=0
        for (key, value) in node_weight.items():
            if len(key)>1:
                x.append(key)
                print(key + ' - ' + str(value))
                c=c+1
            if c == (number-1):
                break
        return x
```

Etape 2 : Classification des textes.

Dans cette étape nous allons essayer de suivre l'enchainement suivant :
Corpus —> preprocessing —> Training —> Validation.

Corpus

On fait le preprocessing du contenu de la colonne 'Content', et on le stocke dans un corpus sous forme d'une liste phrases.

```
def MergeInList(DataSize):
    corpus=[]
    for i in range(DataSize):
        print(i)
        text= preProcessing(news_dataset['content'][i])
        corpus.append(text)
    print('fin')
    return corpus
```

Après on calcule avec TfidfVectorizer la matrice de ce corpus.

```
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
```

Ensuite, on utilise un DataFrame pour bien représenter les données, en utilisant la matrice X et les Feautures_names.

```
df = pd.DataFrame(X.toarray(), columns=vectorizer.get_feature_names())
df
```

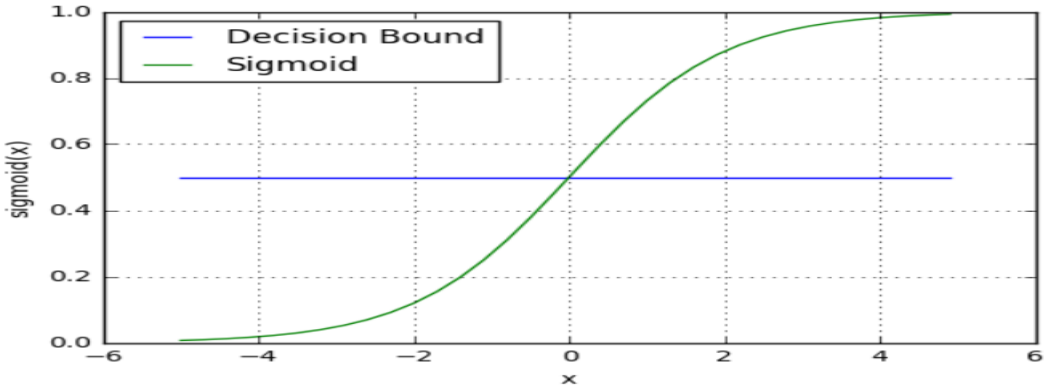
Dans cette phase, nous allons garder seulement les meilleurs features sélectionnés par l'étape précédente. Donc on va garder seulement les features qui représente les mots clés. Et cela en faisant une projection sur notre 'df'. On fixe les colonnes que nous voulons afficher dans une liste (qui représente les mots clés).

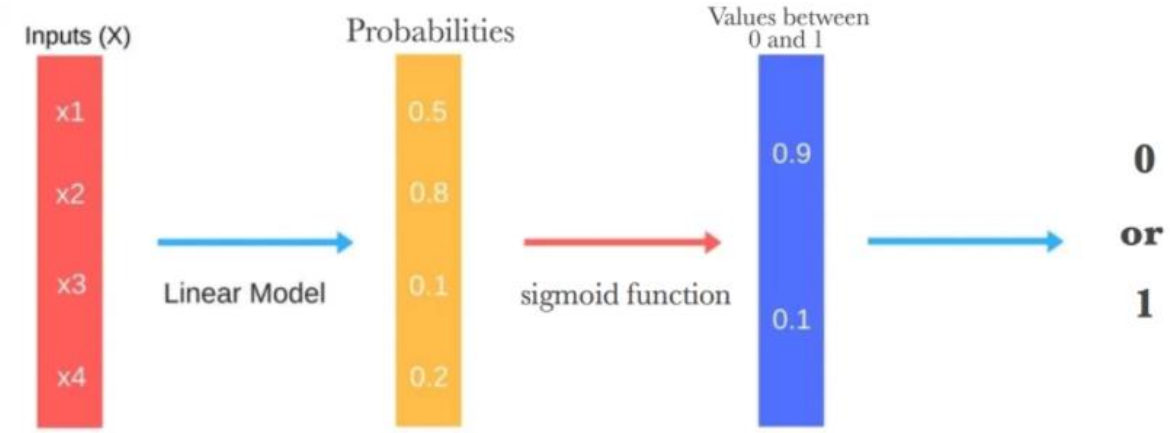
```
X = df[top_keywords]
```

Training

Dans cette étape nous allons utiliser un model nommé 'Logistic regression'.
C'est un modèle statistique utilisé pour déterminer si une variable indépendante a un effet sur une variable dépendante binaire. Cela signifie qu'il n'y a que deux résultats potentiels pour une entrée. Par exemple, il peut être utilisé pour déterminer si un e-mail est un spam ou non, en

utilisant le taux de mots mal orthographiés, un signe courant de spam. D'autres formes d'analyse de régression, comme une régression linéaire, nécessitent la définition d'un seuil pour distinguer les classes binaires (par exemple <50% mal orthographié = pas de spam, >50% mal orthographié = spam). La régression linéaire permet d'établir une probabilité, mais elle doit ensuite être appliquée à une régression logistique pour faire la classification distincte.[1]






```
model = LogisticRegression()
model.fit(X_train,Y_train)
```

accurency score on the trainig and the testing data

```
#score on the test data
X_test_prediction = model.predict(X_test)
test_data_accuracy =accuracy_score(X_test_prediction, Y_test)
print("accurancy score of the testing data: ",test_data_accuracy)
```

```
accurancy score of the testing data:  0.915
```

Validation.

A cette phase on fait un test et une comparaison entre les résultats de prédiction du notre model et les résultats déjà connus.

```
for i in range(10):
    X_new = X_test[i:i+1]
    prediction = model.predict(X_new)
    print(prediction)
    if (prediction[0]==0):
        print('The news is Real')
    else:
        print('The news is Fake')
```

```
[0]
The news is Real
[0]
The news is Real
[1]
The news is Fake
[0]
The news is Real
[0]
The news is Real
[0]
The news is Real
[0]
The news is Real
[1]
The news is Fake
[1]
The news is Fake
[1]
The news is Fake
```

Etape 3 : Results of simulation

Comparer les résultats obtenus avec ceux obtenus avec le paramètre max_features de TfidfVectorizer.
Voici les résultats obtenus avec le paramètre (max_features=100) de TfidfVectorizer.

```
# converting the textual data to numerical data
vectorizer = TfidfVectorizer(max_features=100)
vectorizer.fit(X)
X = vectorizer.transform(X)
```

```
[12] X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2, stratify=Y, random_state=2)
```

```
[13] model = LogisticRegression()
     model.fit(X_train, Y_train)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

```
# accuracy score on the test data
X_test_prediction = model.predict(X_test)
test_data_accuracy = accuracy_score(X_test_prediction, Y_test)
print('Accuracy score of the testing data : ', test_data_accuracy)
```

```
Accuracy score of the testing data :  0.92
```

Apres l'étape de l'entrainement on passe pour faire un test des résultats obtenus par le model.

```
A=[]
for i in range(10):
    X_new = X_test[i]
    prediction = model.predict(X_new)
    print(prediction)
    A.append(prediction[0])

    if (prediction[0]==0):
        print('The news is Real')
    else:
        print('The news is Fake')
```

```
[0]
The news is Real
[0]
The news is Real
[1]
The news is Fake
[0]
The news is Real
[0]
The news is Real
[0]
The news is Real
[0]
The news is Real
[1]
The news is Fake
[1]
The news is Fake
[1]
The news is Fake
```

```
R=[]
for i in range(10):
    R.append(Y_test[i])
```

```
print(A)
```

```
[0, 0, 1, 0, 0, 0, 0, 1, 1, 1]
```

```
print(R)
```

```
[0, 0, 0, 0, 0, 0, 0, 1, 1, 1]
```

CONCLUSION

Pour conclure nous allons faire une comparaison des résultats obtenus par les deux méthodes, la première se base sur la sélection des top featrures en utilisant TextRank algorithme, la deuxième s'appuie sur le paramètre max_features de TfidfVectorizer.

La première méthode donne comme résultats :

```
accurancy score of the testing data :  0.915
```

La première méthode donne comme résultats :

```
Accuracy score of the test data :  0.92
```

En comparant les résultats on peut dire que la première méthode qui se base sur la sélection des top featrures en utilisant TextRank algorithme est moins efficace que la deuxième méthode vue que le score du test est plus élevé.