Task Report:

Tweets classification

By :

Ben Ayad, Mohamed Ayoub

From **ENSIAS**

Supervised by :

M. **Rhicheek**, Patra

Table of contents:

Introduction:

I'll present in this report the different steps and approaches that I tried to solve the machine learning task.

First, I'm going to start with a description of the dataset then I will cite some of the realizations taken from the exploratory data analysis step and then I will talk about the preprocessing and how I cleaned the raw tweets and finally I'll emphasize more on the modeling approaches that I chose.

Without further ado, let's discuss every step in the process in more details.

Data Set:

- ## Data

Each observation in the data set represents a tweet, it's either labeled as **Sports** or **Politics** and the aim of the competition is to come up with a machine learning model to classify these tweets.

- ## Some statistics

Here are some statistics concerning the data set:

- Number of observations of train data : **6525**
- Number of observations in the test data : **2610**
- Mean length of tweets : **108**
- Mean length of politics tweets : **115**
- Mean length of sports tweets : **105**
- Mean of the number of hash tags in politics tweets: **0.5**
- Mean of the number of hash tags in sports tweets: **0.91**
- Mean of the number of @s in politics tweets: **0.46**
- Mean of the number of @s in sports tweets: **0.73**

I constructed a new feature for each line above for further use.

Exploratory Data Analysis

In this step I tried to have some more insight on the data and especially to know more about the type of noise in the data and the specific cleaning that I need to do later on in the preprocessing step.

Let's start with an overview of the most used words/hash tags explore in more details some specific noise in tweets

- ## Most used words:

Here is a picture that sums up the most used words in both training and testing sets, and how much they were used either in a sport politics context.

```
[29] print(tabulate(freq_most_used_words_2,headers=("Key_Words","Occurencies","Politics","Sports"))
```

Key_Words	Occurencies	Politics	Sports
football	983	0.00891266	0.991087
club	949	0.00592885	0.994071
president	407	0.994709	0.00529101
obama	316	1	0
indvaus	233	0	1
bcci	222	0	1
runs	199	0.010101	0.989899
grand	197	0.0508475	0.949153
bbl	196	0.003663	0.996337
atp	190	0	1
conference	187	0.941176	0.0588235
prix	185	0.0198675	0.980132
chief	177	0.985185	0.0148148
final	175	0.0275482	0.972452
minister	174	0.993548	0.00645161
cfc	170	0	1
secretary	168	1	0
seckerry	167	1	0
bigfinals	165	0	1
meeting	156	0.969466	0.0305344
tennis	150	0	1
nelsonmandela	148	1	0
aus	147	0.0204878	0.979512
medvedev	144	1	0
lfc	144	0	1
cabinet	130	1	0
prime	127	0.991379	0.00862069
match	126	0.00775194	0.992248
cricket	124	0.00873362	0.991266
australia	118	0.0168539	0.983146
cup	118	0.0190114	0.980989
vaus	114	0	1
mcfc	113	0	1

I only kept in this dictionary the words that are highly related to a class which means more than 94% of the occurrences are in one side, it's either mostly (more than 94%) related to **Politics** or **Sports.**

For example, the word "**obama**" was used 316 times in both train and test sets and all the occurrences in the training set where labeled as **Politics.**

- ## Types of noise:

Here are 2 random tweets taken from the data:

```
7 '1st Test. Over 39: 0 runs, 1 wkt (M Wade 0, M Clarke 24) #AUS 101/4 #IndvAus http://t.co/fVNl1KTPB1 @BCCI'

4 '@cricketfox Always a good thing. Thanks for the feedback :-)'
```

We can clearly notice the use of:

- ✓ The use of hash tags
- ✓ The use of this pattern “@\w+” as usernames.
- ✓ The use of urls.
- ✓ The use of emojis.
- ✓ The use of acronyms.

These are some examples of noise in the tweets that we need to clean in the preprocessing step.

Preprocessing:

I spent most of the time preprocessing/cleaning the data. Actually this part played a huge rule in increasing the model performance.

For testing purposes, I used a cross validation score on a **pipeline** of: **[Count Verctoriser / tf-idfVectorizer > Naive Bayes]**, the aim of this test was just to see the immediate effects of some preprocessing techniques on the model accuracy.

- ## Tokenizer:

I tried to deal with the different observations mentioned in the EDA phase and with the use of regular expressions patterns using the **re** python library and using some Glove preprocessing techniques **(1)** .

Below is an example of pre- and post-tokenization of two tweets:

```
First Example :
'Watch video highlights of the #wwc13 final between Australia and West Indies at http://t.co/lBXIIk3j'
['watch', 'video', 'highlights', '<hashtag>', 'wwc', '<number>', 'final', 'australia', 'west', 'indies', '<url>']

Second Example :
'@cricketfox Always a good thing. Thanks for the feedback :-)'
['cricketfox', 'always', 'good', 'thing', 'thanks', 'feedback', '<smile>']
```

We can see that the tokenizer in these two examples took care of:

- Stop words.
- Hash tags.
- Usernames.
- Numbers.
- URLs.
- Emojis.

In addition to this, the final version of the tokenizer transforms tokens that ended with "**fc**" to "**football club**" and "**gp**" to "**grand prix**" and can deal with uppercase tokens and with tokens that contain an elongation of characters:

For example:

- "**ESPN**" is transformed to "**espn**" + "**<allcaps>**"
- "**marieee**" is transformed to "**marie**" + "**<elong>**"

We will discuss more these additional mappings in the modeling part.

➢ Benchmarking of different data normalization techniques:

Since it's a common text cleaning method, I tried in this part to normalize my data using different stemmers and lemmatizers and to see the impact of each combination of these techniques on the model accuracy.

For the stemmers I experimented with both : **SnowBall** , **PorterStemmer**
For the lemmatizer, I only used using: **WordNet**

These are the results that I got:

	WordNet	**None**
SnowBall	0.9626	0.9634
PorterStemmer	0.9632	**0,9635**
None	0.9621	**0,9618**

As we can see in the results, using these techniques improved slightly the model accuracy.

Note: As said in the introduction of preprocessing, these scores were computed from a **cross_val_score** on the training data, the score on test data (the Kaggle held out data) was really bad and actually not using any techniques of the ones above gave much better results.

➢ Stop words effects:

In this part, I tried removing English most used words from the tweets, and it actually gave a performance boost.

Removing stop words did improve the accuracy; it went from **0.9585** without removing stop words to **0.9618** after removing stop words.

I tried removing specific twitter stop words **(2)** it didn't actually the score.

➢ Scrapping and Mapping of acronyms.

As a part of cleaning I tried to look for a dictionary of slangs in order to map the meaning of each slang, unfortunately I didn't find any available dictionaries [data structure] online, so I ended up **scrapping** a famous slang dictionary **(3)** .

More preprocessing will be discussed in the modeling step.

Modeling

- ## Modeling Introduction:

In this part I'll talk more about the approaches that I took after preprocessing to model the tweets and actually make predictions, the main approach was the classic one using a pipeline of **[Count Vectorizer / tf-idf Vectorizer > Naive Bayes] .**

- ## Approach 1: Count based features

In this approach, tokens will be transformed into feature vectors and new features will be created using the existing dataset. I tried implementing both count vectors as features and TF-IDF vectors as features.

Count Vector is a matrix notation of the dataset in which every row represents a document from the corpus, every column represents a term from the corpus, and every cell represents the frequency count of a particular term in a particular document.

TF-IDF score represents the relative importance of a term in the document and the entire corpus. TF-IDF score is composed by two terms: the first computes the normalized Term Frequency (TF), the second term is the Inverse Document Frequency (IDF), computed as the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears**. (4)**

Results:

This approach gave the best score on both training (cross_val_score) and testing data (the kaggle hidden data).

I coupled these vectors representations with multiple machine learning models: I used some ensemble methods (**rfc**), **Naive Bayes**, **svm**.

I used a grid search method to tune each one of the models above, the performance did improve for each model but overall, **Naïve Bayes** was outperforming other machine learning models.

- ## Approach 2: Pre-trained word embedding, Glove-Twitter

In this approach I tried using the Glove word embedding that was specifically trained on Twitter's data **(5) ,** I represented each tweet as vector of the sum (I tried using the mean too) of vectors of its tokens and I gave the null vector to words that aren't part of the Glove dictionary.

I tried using multiple versions of Glove, different dimensions 25, 50, 100, 200 as a base.

The model was less performing than the first approach, mean accuracy was around **0.92**

When using data augmentation/unification, the model accuracy went up to around **0.94**

- ## Approach 3: Training my own word embedding

In this approach I used the **Gensim**'s **word2vec** module, this approach didn't give any remarkable results, the model failed to recognize patterns, the accuracy on a **cross_val_score** metric was around **0.74**

When trying to explain why it failed, I found out that the word embedding approach actually needs a lot of data, so maybe the 9k tweets (I used both training and testing sets to train the embedding) wasn't enough for the model to learn to make good predictions.

- ## Problems that I faced when modeling

  - **Meaning augmentation :**

When analyzing where the model made wrong predictions, I noticed that the model makes a lot more wrong assumptions about politics than sports, which means it tends to predict more sports than politics [similar to the problem of imbalanced classes in a data set] so I thought about making a "**meaning augmentation**" of the some politics tweets so I added manually some synonymous of some politics keywords to tweets that contain them, for example I added "**president**" and "**politics**" to tweets that contain the word ""**seckerry**" or "**obama**", making this type of data augmentation has improved the performance significantly but when thinking about automating it (adding random politics tags to tweets that contain most used politics keywords) the accuracy actually decreased.

  - **Meaning unification :**

This problem concerns words with same meaning and with different forms: like **obama** / **barackobama** , the count vectorizer treat these two words as different although they represent the same meaning.

Here are some illustrations from the data set:

```
[26] find_and_group("ausgp")
```

```
Parents                Occurencies
-------------------  -------------
ausgp                           63
cantwaitforausgp                29
cantwaitfortheausgp              1
```

```
find_and_group("football")
```

```
Parents           Occurencies
--------------  -------------
football                  983
bbcfootball                39
footballleague              3
footballer                  3
detfootballer               2
```

```
[ ] find_and_group("obama")
```

```
Parents                   Occurencies
----------------------  -------------
obama                             316
barackobama                        74
obamacare                           3
obamachallengedthepriv              1
```

The illustration shows the number of parents grouped of some substrings.

The solution in processing was to add a manual data unifier for some of the most used keywords that I tried later on to automate.

- **Acronyms and signatures :**

Also I noticed that the model fail to learn the meaning of some sports keywords like “**ausgp**” “**motogp**”, it always made wrong assumption about the tweets that contains these words no matter which algorithm used in modeling .

```
[17] football_club = []
     for i in X_tokens :
         if i[len(i)-2:] == "fc" :
           football_club+=[i]

     c_fc = Counter(football_club).most_common(10)

     print(tabulate(c_fc,headers = ("Football Club Acronym","Occurencies")))
```

```
Football Club Acronym      Occurencies
-----------------------  -------------
cfc                              170
lfc                              144
mcfc                             113
efc                               18
oafc                              15
chelseafc                          9
mclfc                              7
effc                               5
crewealexfc                        5
bfc                                4
```

```
[20] gp = []
     for i in X_tokens :
         if i[len(i)-2:] == "gp" :
           gp+=[i]

     c_gp = Counter(gp).most_common(10)

     print(tabulate(c_gp,headers = ("gp$ Acronym","Occurencies")))
```

```
gp$ Acronym         Occurencies
----------------  -------------
ausgp                        63
motogp                       52
cantwaitforausgp             29
birtymotogp                   9
otogp                         8
yamahamotogp                  6
gp                            4
ondamotogp                    3
eurofractiesgp                2
australiangp                  1
```

Solution for this case was to add/reinforce “**grand prix**“ and “**football club**” to every occurrence of either “**gp**” or “**fc**” signature.

And adding this actually helped improve the accuracy.

- ## Modeling conclusion:

The first approach was outperforming other approaches so I just confined to it and tried to improve it as much as I can. Also analyzing the wrong predictions gave me a strong knowledge on where I should work more on in my preprocessing which lead to significant increase in performance.

Conclusion:

To sum up, preprocessing played a major rule, techniques like stemming and lemmatizing didn't help increase the model while other like removing stop words did increase the overall performance.

Also, although I couldn't make an automatic solution, manually augmenting the meaning and unification of some keywords helped increase the performance remarkably.

For the word embedding approach, 8k tweets wasn't enough for the model to predict with high accuracy.

And in terms of the machine learning models, the best performance was obtained with a **Naïve Bayes** applied on a count vectorizer, also to note that using variants of these statistical features like changing **n_grams** didn't improve the performance.

Other Solutions to improve my solution:

- I spent the last week thinking about a way of automating the meaning unification of keywords and meaning augmentation of tweets, but what I did didn't actually improve the accuracy, so building an algorithm that could augment the meaning of tweets would definitely help and also building a specific words cluster to unify some keywords that are written in various forms. And also mapping acronyms to their actual meaning.
- Using less brute force methods (in either preprocessing or modeling) would definitely increase the speed of the solution, especially when talking about a large scale implementation.
- I only used **Gensim** modules to train my own word embedding, so building a word embedding from scratch (using neural network) and getting more data to train would definitely be beneficial.
- Also writing a data science report was challenging for me, due to the messiness of steps and the back and forth between modeling and preprocessing, so I would love to learn more about how to make a professional report and use **Latex** for example.

References:

1. **GLoVe.** GitHub. [Online] https://gist.github.com/ppope/0ff9fa359fb850ecf74d061f3072633a.

2. **GONG Wei.** [Online] https://sites.google.com/site/iamgongwei/home/sw.

3. **webopedia.** [Online] https://www.webopedia.com/quick_ref/textmessageabbreviations.asp.

4. AnalyticsVidhya. [Online] https://www.analyticsvidhya.com/blog/2018/04/a-comprehensive-guide-to-understand-and-implement-text-classification-in-python/.

5. Glove. [Online] https://nlp.stanford.edu/projects/glove/.