Data Analytics
Coursework

Analysis Of Comments

Elena Pedrini – Boutros Azar – Fatih Özhamaratli

April 13, 2018

Contents

Introduction

Problem Statement and Hypothesis

In order to complete this coursework, we decided to take part in one of the competitions organized by *Kaggle*. Discussing things we care about can be difficult. The threat of abuse and harassment online means that many people stop expressing themselves and give up on seeking different opinions. There are already a few models for detecting toxic comments available publicly. However, they still make errors and they don't allow users to select which types of toxicity they're interested in finding [1].

This project consists of building a model capable of classifying comments posted online – blogs or discussions – according to different types of toxicity like threats, obscenity, insults, and identity-based hate. One of the purposes behind this project would be to address the ability to express personal opinions online freely and promote civil and effective conversations on the web as well as prevent people from leaving online discussions due to disrespectful comments. In other words, create environments where online discussion are more productive and respectful.

As mentioned before we will need to understand textual data, its meaning and its structure to be able to make assumptions about the content; hence the text classification problem. We will first explain in this report the process we adopted starting from the exploration of the dataset and the pre-processing steps performed. After that we will dive into how we transformed textual data to numeric vectors before finally discussing the different machine learning models built, the training and testing phases as well as the outcomes and performance of these models.

Description of the Dataset

The dataset we have for this project contains 159548 comments with their binary labels from Wikipedia's talk page edits which have been labeled by human raters for toxic behavior. The types of toxicity are :

- toxic
- severe_toxic
- obscene
- threat
- insult
- identity_hate

The model(s) must predict a probability of each type of toxicity for each comment. The code below shows an element of the dataset.

```
import pandas as pd
train = pd.read_csv('train.csv')
train.iloc[168] #169th element
```

```
id                                              00686325bcc16080
comment_text      You should be fired, you're a moronic wimp who...
toxic                                                          1
severe_toxic                                                   0
obscene                                                        0
threat                                                         0
insult                                                         1
identity_hate                                                  0
Name: 168, dtype: object
```

1 Exploring the dataset

First of all, we start by exploring the dataset to understand its content and get an idea of the different labels and the way they are partitioned. In order to do that we save the number of appearances of each target class in a dictionary data structure. We can see already that we need to add an extra class "non_offending" for the comments that do not belong to any of the target classes. In order to do that we use the loop below :

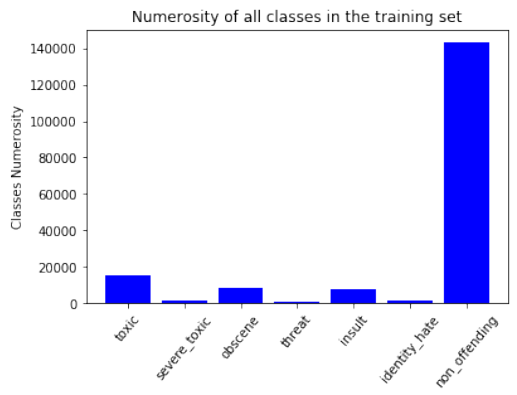
```python
i = 0
dic = {'toxic':0, 'severe_toxic':0, 'obscene':0, 'threat':0, 'insult':0, '
identity_hate':0, 'non_offending':0}
while i<len(train):
    flag = False
    for key in dic:
        if key != 'non_offending' and train.iloc[i][key]==1:
            dic[key]+=1
            flag = True
    if flag == False:
        dic['non_offending']+=1
    i+=1
```

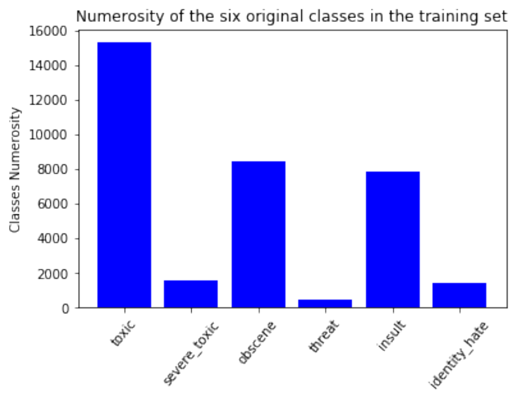
The results are the below (c.f. file `Project - Exploratory Analysis.ipynb`) :

toxic	15294
severe_toxic	1595
obscene	8449
threat	478
insult	7877
identity_hate	1405
non_offending	143323

It is important to note that classes can overlap, i.e. a comment can belong to more than one of the offending classes, meaning that the sum of the values in the defined dictionary is greater than the number of the total comments present in the training set.

Using the `matplotlib.pyplot` library, we can plot these values and we get the first plot below. The second plot is what we have without the non_offending class [2].





As we can see, the data is strongly unbalanced in favor of the `non_offending` comments. When removing this category we can still see that the `toxic` comments are much more frequent than the others while `threat` comments are very rare. This may be a challenge later on during the training phase of the models since it might cause the classifier to be more likely to assign test data to the classes having the highest frequency.

2 Pre-processing the data

Since we are dealing with textual data, we need to do some processing first. For this part we use a lot of features from the `nltk` library [3].
As we can see in the code (c.f. file `preprocess_new.ipynb` or `preprocess_new.py`), we have the function `def preProcess(text):` in which we start by tokenizing the comments into words, transforming everything to lower case, removing stop words, lemmatizing and then we use a stemmer. The function returns the processed tokens. The returned tokens can be unigrams, bigrams or trigrams. In this case we decided to stay on unigrams only for the time being.

```
    tokens = re.split(r"\s+",text) #tokenisation
    tokens = [t.lower() for t in tokens] #lower case
    tokens = [item for item in tokens if item not in stopwords] #remove stop words
    tokens=[lemmatizer.lemmatize(t) for t in tokens] #lemmatizer
    tokens = [stemmer.stem(t) for t in tokens] #stemmer

    unigrams = list(nltk.ngrams(tokens,1))
    #bigrams = list(nltk.ngrams(tokens,2))
    #trigrams =  list(nltk.ngrams(tokens,3))
    tokens = unigrams #+ bigrams #+ trigrams
```

After doing that, we noticed that we have a problem with punctuations being considered as a separate word. Therefore we added the below code to remove them before starting the preprocessing.

```
    #remove punctuation
    for el in text:
        if el in string.punctuation:
            text = text.replace(el, ' ')
```

Once the processing algorithm defined, we wrap it into the function `def prepareTrainTestSet():` that will be used in other files later on. As we can see, this function loads the training dataset and then iterates through each comment to apply the `preProcess()` function on them.

```
    def prepareTrainTestSet(trainCsvPath,testCsvPath,...):
    #Load Data
    trainDF,testDF = loadData(trainCsvPath,testCsvPath)
    ...
    #preProcess the train Dataset
    def iterateDF(df):
        numElements = len(df)
        for i in range(0,numElements):
            df.iloc[i,df.columns.get_loc('comment_text')]=preProcess(df.iloc[i,df.columns
    .get_loc('comment_text')])
        return df

        preProcessedTrainDF=iterateDF(trainDF)
    return preProcessedTrainDF
```

Looking back at the 168th comment we transform this:

```
"You should be fired, you're a moronic wimp who is too lazy to do research. It makes me
    sick that people like you exist in this world."
```

into this:

```
['fire','moron','wimp','lazi','research','make','sick','peopl','like','exist','world']
```

It is important to note that running the processing function on the whole dataset we have is quite exhaustive and it takes around 20 mins to complete on a machine with good computing power.

3 Words to vectors

We have established in the previous section (Section 2) how to preprocess the data. Now we have every comment turned into tokens that need to be transformed into numerical vectors. There are several ways to achieve this step. For our project we decided to try implementing several methods and not limit ourselves to only one of them.

3.1 Word2Vec

The first method we implemented is called `Word2Vec`. It is a two-layer neural net that processes text and it turns text into a numerical form that deep nets can understand. Its input is a text corpus and its output is a set of vectors. The advantage of `Word2Vec` is that it makes natural language computer-readable – we can start to perform powerful mathematical operations on words to detect their similarities. It allows to make analogies like "*Rome is to Italy as Beijing is to China*" [4].

Going to the implementation (c.f. file `wordstovec.py` or `WordstoVectors.ipynb`), this part was also written as a library since it will be reused later on in several files. The code is wrapped in the function `def WordsToVecFunction(preProcessedDF):` which takes the tokens as input and applies the model before returning a list containing each vector and the labels associated to the vector.

```
    def WordsToVecFunction(preProcessedDF):
        ...
        #get the tokens and the labels in lists
        for x in range(numElements):
            myList.append(preProcessedDF.iloc[x][1])
            myLabels.append([preProcessedDF.iloc[x][i] for i in range(2,8)])

        #Train the model word2vec
        model = gensim.models.Word2Vec(iter=1,min_count=0)
        model.build_vocab(myList)
        ...
        #Avg the words to have one vector per sentence
        for j in range(myListLen):
            mySum = numpy.zeros(100)
            for i in myList[j]:
                mySum += model[i]
            myAvg = mySum/len(myList[j])
            Sent2Vec.append([myAvg, myLabels[j]])
        return Sent2Vec

```

Displaying the list as a DataFrame we get something that looks like the below

```
data = pd.DataFrame(sent2vec)
data.head()
```

	vectors	labels
0	[0.0009850456144367516, 0.0010559120834201436...]	[0, 0, 0, 0, 0, 0.0]
1	[3.701340808350194e-05, 0.001403201709503524,...]	[0, 0, 0, 0, 0, 0.0]
2	[0.0005967772698828153, 0.002026788852249627,...]	[0, 0, 0, 0, 0, 0.0]
3	[0.0002812116314929041, 0.0003179908324874794...]	[0, 0, 0, 0, 0, 0.0]
4	[0.0002029971161391586, -0.000119605119107291...]	[0, 0, 0, 0, 0, 0.0]

3.2 GloVe

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. It requires a single pass through the entire corpus to collect the statistics. For large corpora, this pass can be computationally expensive, but it is a one-time up-front cost [5].

The implementation here is very short and straight forward (c.f. file GloveVect.ipynb). First a corpus is constructed using word in text8Corpus and our dataset. Then model is trained using the constructed corpus and the trained corpus model is saved for fast later usage [6].

```
    text8corpus=list(map(lambda x: list(map(lambda a: a[0],reduce(list.__add__,list(map(
    lambda y: preprocess_new.preProcess(y),x))))),list(Text8Corpus('/homes/fo301/text8'))
    ))
    sentences = list(itertools.chain(itertools.islice(text8corpus,None),preProcessedTrDF[
    'comment_text'].values))

    corpus = Corpus()
    corpus.fit(sentences, window=10)

    glove = Glove(no_components=100, learning_rate=0.05)
    glove.fit(corpus.matrix, epochs=30, no_threads=4, verbose=True)
    glove.add_dictionary(corpus.dictionary)
```

3.3 Bag of Words & TF-IDF

Bag of Words (BoW) is an algorithm that counts how many times a word appears in a document. Those word counts allow us to compare documents and gauge their similarities for applications like search, document classification and topic modeling. It's a method for preparing text for input in a deep-learning net.
Term-frequency-inverse document frequency (TF-IDF) is another way to judge the topic of an article by the words it contains. With TF-IDF, words are given weight – TF-IDF measures relevance, not frequency. That is, wordcounts are replaced with TF-IDF scores across the whole dataset. TF-IDF measures the number of times that words appear in a given document (that's term frequency). But because words such as "and" or "the" appear frequently in all documents, those are systematically discounted. That's the inverse-document frequency part. The more documents a word appears in, the less valuable that word is as a signal [7].
While Word2Vec and GloVe are great for digging into documents and identifying content and subsets of content. The vectors represent each word's context, the ngrams of which it is a part. BoW is good for classifying documents as a whole.

The implementation is very short and straight forward here as well (c.f. file BoWTfidfVectorizer.ipynb or BoWTfidfVectorizer.py).

```
    def getBoWTfidfVectors(text_ary):
        preProcessedTrDF= preprocess_new.prepareTrainTestSet('train.csv','test.csv','bow'
    ,seperateLabelInfo=1,tokenize=0)
        boWTfidfVectorizer = CustomVectorizer()
        vectors = boWTfidfVectorizer.fit_transform(preProcessedTrDF['comment_text'])
        return vectors
```

4 Machine learning models

So far, we have loaded the data, applied some preprocessing on the words by making use of the `nltk` library as well as others. Then we can use three different methods to transform the comments into vectors. Since the code for those steps was written in the form of python libraries, it can be done in a few lines as shown below :

```
import wordstovec as wtv
import preprocess_new as pre
import BoWTfidfVectorizer
```

```
#For Word2Vec Embedding
preProcessedTrainDF = pre.prepareTrainTestSet('train.csv','test.csv','word2vec',
seperateLabelInfo=1)
sent2vec = []
sent2vec = wtv.WordsToVecFunction(preProcessedTrainDF)
```

```
#For BoW with Tf-Idf
preProcessedTrDF= pre.prepareTrainTestSet('train.csv','test.csv','bow',
seperateLabelInfo=1,tokenize=0)
embed_vect = BoWTfidfVectorizer.getBoWTfidfVectors(preProcessedTrDF['comment_text'].
values)
```

```
#For using Glove Embedding and Paragraf Vectors
glove = Glove.load('glove.model')
preProcessedTrDF= pre.prepareTrainTestSet('train.csv','test.csv','glove',
seperateLabelInfo=1,tokenize=1)
embed_vect = list(map(lambda x:glove.transform_paragraph(x,ignore_missing=True),
preProcessedTrDF['comment_text'].values))
```

Now that we have the vectors, we can start applying some machine learning algorithms and see how they perform. For the competition only one model is requested but for this project we decided to implement several ones and compare their outputs. We will first use a neural network model and check its performance. Then we will see how does the linear regression perform. Afterwards we will compare the results of `LinearSVC` when using `GloVe` and `BoW`. Finally we will also compare the results of `Naive Bayes` when using `GloVe` and `BoW`.

4.1 Neural networks

The first model that we implemented is a neural network (c.f. NeuralNetwork.ipynb) [8]. We start by getting the vectors using one the three methods mentioned above. Let's say we use the `Word2Vec` to begin with.

Parse the results to get the vectors and the labels

First we parse the vectors and the labels in two different lists. Then we use the function `fillna(0)` to replace all `NaN` by 0 because we faced an error due to one `NaN` value. Also we noticed that all the labels are integers except the last one which is the reason of the cast written below.

```
# Fill the NaN values with 0
labels['identity_hate'] = labels.identity_hate.fillna(0)
# Replace the type of the last columns to int64 like the others instead of float64 (
check labels.dtypes)
labels['identity_hate'] = labels.identity_hate.astype('int64')
```

Train Test Split and Scaling

To avoid over-fitting, we will divide our dataset into training and test splits. The training data will be used to train the neural network and the test data will be used to evaluate the performance of the neural network. This helps with the problem of over-fitting because we're evaluating our neural network on data that it has not seen (i.e. been trained on) before. Here we split the dataset into train and test data using the library `sklearn.model_selection`. The above script splits 70% of the dataset into our training set and the other 30% into our testing set. It is important to note that the split is random. This means that several splits with the same ratio or a different may all lead to slightly different results. Therefore this ratio can be subject to testing to find out what could be the best value to chose for the split for more improvements.

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(vectors, labels, test_size =
0.30)
```

Before making actual predictions, we scale the features so that all of them can be uniformly evaluated. Feature scaling is performed only on the training data and not on test data.

```python
# Scaling the data (not mandatory in this case since already scaled by words2vec)

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)

X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Applying the neural network using multi-labels

At this point, we can start training our neural network. Using the `sklearn.neural_network` library, we import `MLPClassifier` [9]. This imported class is initialized with `hidden_layer_sizes` as (10, 10, 10), which is used to set the size of the hidden layers. In our script we created three layers of 10 nodes each. The second parameter specifies the number of iterations, or the epochs which is 1000 in our case. In order to train the algorithm we use the `fit()` function using the training data. Finally we make predictions on our test data. We have to mention here that the `y_train` contains all the labels for each sample. This means that here we are doing **multi-label classification**.

```python
# Fit the model on the data with the multi-labels

from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(hidden_layer_sizes=(10, 10, 10), max_iter=1000)
mlp.fit(X_train, y_train)

# Make the prediction
predictions = mlp.predict(X_test)
```

Evaluating the Algorithm

With the model trained and running, we need to do some evaluations. To evaluate an algorithm, the most commonly used metrics are confusion matrix, precision, recall, and f1-score. The confusion_matrix, classification_report, and accuracy_score methods of the `sklearn.metrics` library can help us find these scores. Since we are doing multi-label classification here, we cannot compute the confusion matrix.

```python
# The metrics

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
#print(confusion_matrix(y_test,predictions))
print(classification_report(y_test,predictions))
accuracy_score(y_test,predictions)
```

The classification report is shown in the table below. We have for each target (0 to 5):

Precision: The ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

Recall: The ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

F1-score: The F-beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and worst score at 0.

Support: The support is the number of occurrences of each class in `y_true`.

As we can see here the results are not great. We do not have a high precision and the recall is really low. Also the f1-score is below 0.51 for all the classes. We don't even have metrics for the classes that have a low support. The accuracy here says 89.9% however it is not a reliable metric in this case since it does not coincide with the rest of the outputs.

```
             precision    recall  f1-score   support

          0       0.77      0.28      0.41      4639
          1       0.49      0.04      0.07       493
          2       0.80      0.37      0.51      2570
          3       0.00      0.00      0.00       148
          4       0.68      0.33      0.45      2373
          5       0.00      0.00      0.00       452

avg / total       0.70      0.29      0.41     10675

Accuracy: 0.8991956544447927 (89.92%)
```

Applying the neural network on each label

To have a better understanding of the predictions and check thoroughly the performance of the model, we can use one label at a time. The steps are the same as before, however the we change the labels used for training and consequently the labels used for metrics. We will see the implementation for the label `toxic` and the others will be similar. This time since we have only one label at a time, we can show the confusion matrix which will be a 2×2 matrix.

```python
# Fit the model on the data on label 1: toxic
mlp = MLPClassifier(hidden_layer_sizes=(10, 10, 10), max_iter=1000)
mlp.fit(X_train, y_train.toxic)

# Make the prediction
predictions = mlp.predict(X_test)

# The metrics
print(confusion_matrix(y_test.toxic, predictions))
print(classification_report(y_test.toxic, predictions))
accuracy_score(y_test.toxic, predictions)
```

We can see already a big difference in the results when working on separate labels. We have better accuracy, better f1-scores, and improved precision and recall as well. We can still see a good amount of samples being misclassified in the confusion matrix but those numbers are smaller in the labels having the least occurrences. There might be a case of over-fitting on those labels since the support value is very low.

Label 1: toxic

```
             precision    recall  f1-score   support

          0       0.93      0.99      0.96     43226
          1       0.75      0.28      0.40      4639

avg / total       0.91      0.92      0.90     47865

Accuracy: 0.9207980779275045 (92.09%)
```

```
[[42788   438]
 [ 3353  1286]]
```

Label 2: severe_toxic

```
             precision    recall  f1-score   support

          0       0.99      0.99      0.99     47372
          1       0.36      0.40      0.38       493

avg / total       0.99      0.99      0.99     47865

Accuracy: 0.9866290609004492 (98.66%)
```

```
[[47027   345]
 [  295   198]]
```

Label 3: obscene

```
             precision    recall  f1-score   support

          0       0.97      0.99      0.98     45295
          1       0.74      0.39      0.52      2570

avg / total       0.95      0.96      0.95     47865

Accuracy: 0.9601169957171211 (96.01%)
```

```
[[44941   354]
 [ 1555  1015]]
```

Label 4: threat

```
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     47717
          1       0.18      0.11      0.14       148

avg / total       0.99      1.00      1.00     47865

Accuracy: 0.9957589052543612 (99.58%)
```

```
[[47646    71]
 [  132    16]]
```

Label 5: insult

```
             precision    recall  f1-score   support

          0       0.97      0.99      0.98     45492
          1       0.65      0.31      0.42      2373

avg / total       0.95      0.96      0.95     47865

Accuracy: 0.9577561892823566 (95.78%)
```

```
[[45099   393]
 [ 1629   744]]
```

Label 6: identity_hate

```
             precision    recall  f1-score   support

          0       0.99      1.00      1.00     47413
          1       0.42      0.10      0.16       452

avg / total       0.99      0.99      0.99     47865

Accuracy: 0.9901807165987674 (99.02%)
```

```
[[47349    64]
 [  406    46]]
```

Areas of possible improvements

When using a neural network, there are several areas of improvements and many way to tackle them.
First we have the data itself. Instead of using the `Word2Vec` method we could use one of the two other options that we have implemented.
The neural network has a lot of parameters that can be tweaked in order to achieve the best results. Those parameters include:

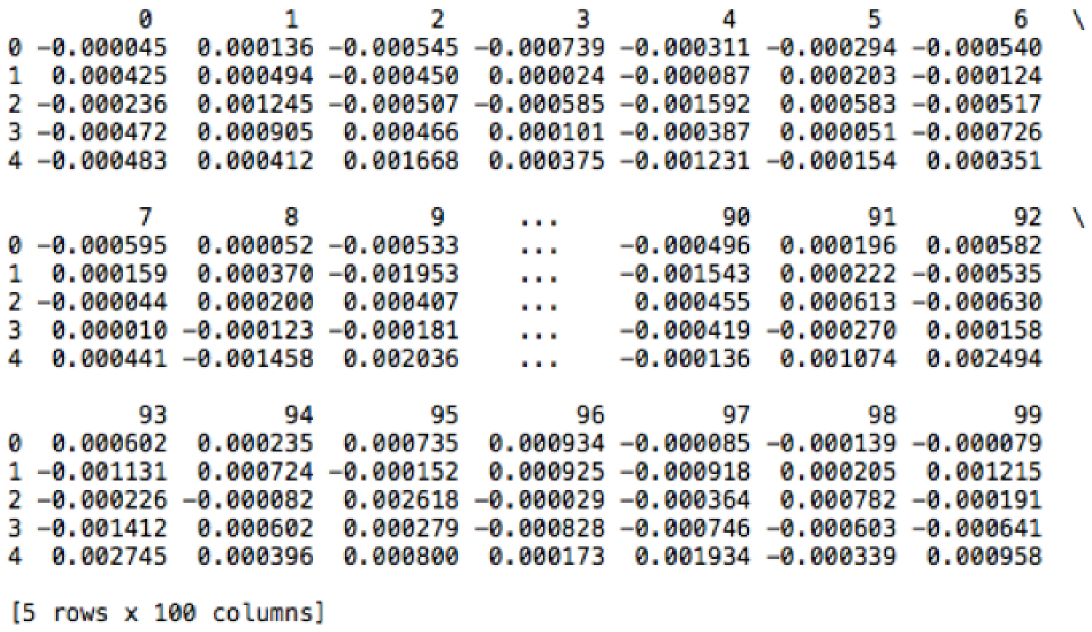
- number of hidden layers
- activation function
- solver for weight optimization.
- learning rate

etc

All of these parameters – and many others – can be changed in order to find the optimal results for our dataset.

4.2 Logistic regression

The second model we will look at is the logistic regression. For this model, we load the vectors using the `Word2Vec` method as well. Having a look at the first five rows using the `head()` function of a `DataFrame` we see the vectors as shown in figure 1.



```
         0         1         2         3         4         5         6  \
0 -0.000045  0.000136 -0.000545 -0.000739 -0.000311 -0.000294 -0.000540
1  0.000425  0.000494 -0.000450  0.000024 -0.000087  0.000203 -0.000124
2 -0.000236  0.001245 -0.000507 -0.000585 -0.001592  0.000583 -0.000517
3 -0.000472  0.000905  0.000466  0.000101 -0.000387  0.000051 -0.000726
4 -0.000483  0.000412  0.001668  0.000375 -0.001231 -0.000154  0.000351

         7         8         9    ...           90        91        92  \
0 -0.000595  0.000052 -0.000533   ...    -0.000496  0.000196  0.000582
1  0.000159  0.000370 -0.001953   ...    -0.001543  0.000222 -0.000535
2 -0.000044  0.000200  0.000407   ...     0.000455  0.000613 -0.000630
3  0.000010 -0.000123 -0.000181   ...    -0.000419 -0.000270  0.000158
4  0.000441 -0.001458  0.002036   ...    -0.000136  0.001074  0.002494

        93        94        95        96        97        98        99
0  0.000602  0.000235  0.000735  0.000934 -0.000085 -0.000139 -0.000079
1 -0.001131  0.000724 -0.000152  0.000925 -0.000918  0.000205  0.001215
2 -0.000226 -0.000082  0.002618 -0.000029 -0.000364  0.000782 -0.000191
3 -0.001412  0.000602  0.000279 -0.000828 -0.000746 -0.000603 -0.000641
4  0.002745  0.000396  0.000800  0.000173  0.001934 -0.000339  0.000958

[5 rows x 100 columns]
```

Figure 1: First 5 rows of the vectors

The **seaborn** library can be interesting to use since it allows us to see the correlation in the vectors as a heat-map (Figure 2). Except for the main diagonal, for which the color suggests that the correlation of the variables with themselves is 1, the rest of the matrix is characterized by features that present very low correlations between each other.

```python
import seaborn as sb
sb.heatmap(data.corr())
```

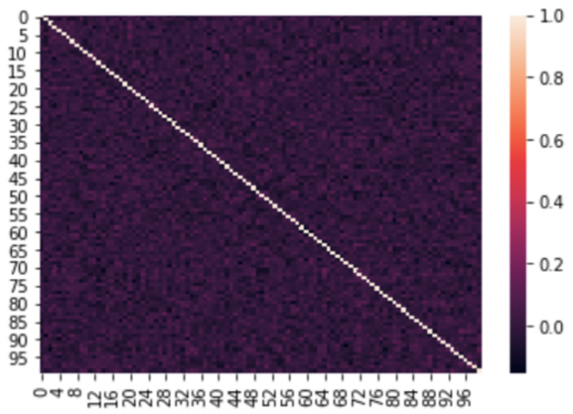


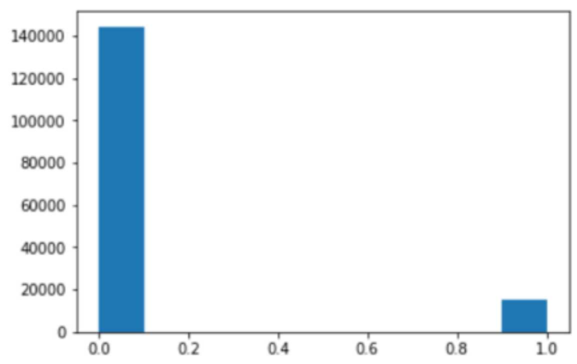
Figure 2: Heatmap showing the correlation in the vectors

Since the logistic regression in the **sklearn** library does not support multi-label classification, we analyze each label on its own. For each label, we check its percentage in the whole dataset and then we apply the same split that we did earlier for the neural network.

```python
print(plt.hist(y_1))
print("\nPercentage of toxic comments: "+str(round(sum(y_1)/len(y_1),1)*100))

X_train_1, X_test_1, y_train_1, y_test_1 = train_test_split(X, y_1, test_size = .3,
random_state=25)
```

```
Percentage of toxic comments: 10.0
```



When we have to train the model, we can leave all the values and parameters to default. This means that:

- the `class_weight` parameter set as default: all classes have weight 1
- `multi_class` parameter set as default (‘`ovr`’): binary problem set for each label

- penalty is `L2` (norm used in the regularization), also known as least squares error
- solver is `liblinear` (optimization algorithm)

We immediately notice that all `toxic` comments are predicted as `non-toxic`. The model is not working as expected but the accuracy is quite high. This is a problem that can occur when we deal with very unbalanced training sets, in which a class is rare (target = 1 in this case) while the other one is more common. This may happen because the algorithm tries to optimize the overall accuracy.
One way to address this problem can be using an optimization function that takes into consideration the `tp` (true positive rate) or the `tn` (true negative rate). In fact `tp` for class 1 (also called recall) is 0.

```
LogRegA = LogisticRegression()
LogRegA.fit(X_train_1, y_train_1)
y_predA = LogRegA.predict(X_test_1)

confusion_matrixA = confusion_matrix(y_test_1, y_predA)
print(confusion_matrixA)

print(skm.classification_report(y_test_1, y_predA))

accA = skm.accuracy_score(y_test_1, y_predA, normalize=True, sample_weight=None)
print("Accuracy: "+str(round(accA,2)*100)+'%')
```

```
                precision    recall  f1-score   support

          0          0.90      1.00      0.95     43176
          1          0.00      0.00      0.00      4689

avg / total          0.81      0.90      0.86     47865

Accuracy: 90.0%
```

```
[[43176     0]
 [ 4689     0]]
```

In this specific scenario it can be worth to try to optimize the function according to the TPR, i.e. to correctly predict the observations with target = 1 (rare class). Other methods to deal with unbalanced training sets, for example under-sampling, oversampling, set priors. In this project we let the logistic regression function automatically face this problem by setting the parameter `class_weight` to `balanced`.
In this case the accuracy is lower compared to the first model, but this works better if we consider other evaluation metrics, for example recall for class 1 (or true positive rate): `tp` for the first model is 0 (no toxic comment correctly classified), for the second model is 0.55.
Since our aim is to identify toxic comments, the second model is with no doubts better than the first one.

```
LogRegB = LogisticRegression(class_weight = 'balanced')
LogRegB.fit(X_train_1, y_train_1)

y_predB = LogRegB.predict(X_test_1)
```

Label 1: toxic

```
                precision    recall  f1-score   support

          0          0.94      0.83      0.89     43176
          1          0.26      0.55      0.36      4689

avg / total          0.88      0.81      0.83     47865

Accuracy: 81.0%
```

```
[[36042  7134]
 [ 2121  2568]]
```

We can see below the results we obtain for the rest of the labels. Compared to the neural network, we have a lower accuracy, but better f1-scores, improved precision and recall as well. We can still see this time a lower amount of samples being misclassified in the confusion matrix and those numbers are smaller in the labels having the least occurrences as well. However in this model we have fewer false negative but a high number of false positives

Label 2: severe_toxic

```
                  precision    recall  f1-score   support

          0           1.00      0.97      0.98     47374
          1           0.18      0.65      0.28       491

avg / total           0.99      0.97      0.98     47865

    Accuracy: 97.0%
```

```
        [[45911   1463]
         [  174    317]]
```

Label 3: obscene

```
                  precision    recall  f1-score   support

          0           0.97      0.92      0.95     45253
          1           0.29      0.55      0.38      2612

avg / total           0.94      0.90      0.91     47865

    Accuracy: 90.0%
```

```
        [[41654   3599]
         [ 1172   1440]]
```

Label 4: threat

```
                  precision    recall  f1-score   support

          0           1.00      0.90      0.95     47711
          1           0.02      0.68      0.04       154

avg / total           1.00      0.90      0.95     47865

    Accuracy: 90%
```

```
        [[43052   4659]
         [   49    105]]
```

Label 5: insult

```
                  precision    recall  f1-score   support

          0           0.97      0.90      0.93     45462
          1           0.22      0.55      0.31      2403

avg / total           0.94      0.88      0.90     47865

    Accuracy: 88.0%
```

```
        [[40697   4765]
         [ 1086   1317]]
```

Label 6: identity_hate

```
                  precision    recall  f1-score   support

          0           1.00      0.89      0.94     47458
          1           0.05      0.64      0.08       407

avg / total           0.99      0.88      0.93     47865

    Accuracy: 88.0%
```

```
        [[42022   5436]
         [  148    259]]
```

For possible areas of improvements here we could use one of the two other options that we have implemented.
We can also test different values of threshold (probability of belonging to the target class compared to the other class) to select when an observation should be labeled as 1 or 0 (default threshold is 0.5).

4.3 Naive Bayes

In this part, we will see the implementation of a naive Bayes model and we will compare the results obtained with both `GloVe` and `BoW`.
Once we have loaded the vectors, we need to split the data.

```
#Preparing Train and Test Dataset, splitting according to 70% train set and 30% test set
features={"toxic","severe_toxic","obscene","threat","insult","identity_hate"}
X_tr={}
X_te={}
y_tr={}
y_te={}
preProcessedTrDF=preProcessedTrDF.reset_index()
for feature in features:
    X_tr[feature], X_te[feature], y_tr[feature], y_te[feature] = train_test_split(
embed_vect, preProcessedTrDF[feature].values.astype(np.float), test_size = .3,
random_state=25)
y_tr['identity_hate'][41272]=0
```

Just as we did in the previous methods, we need to scale the data.

```
#Scaling the values between the values 0 and 1.
for feature in features:
    min_max_scaler = preprocessing.MinMaxScaler()
    X_tr[feature] = min_max_scaler.fit_transform(X_tr[feature])
    X_te[feature] = min_max_scaler.transform(X_te[feature])
```

Then we define the classifier as MultinomialNB and we train the train-set using the `fit()` function.

```
pipeline={}
for feature in features:
    pipeline[feature] =  Pipeline([('mNB', MultinomialNB())])
    pipeline[feature].fit(X_tr[feature],y_tr[feature])
```

After that we can start making the predictions and show the metrics. Here we used a loop to go over all the labels at once instead of re-writing the code for each as we did before.

```
#Prediction using test-set and report of statistics.
predicted={}
accF={}
for feature in features:
    predicted[feature] = pipeline[feature].predict(X_te[feature])
```

Below are the results when using `GloVe` first then `BoW`. As we can see when using `GloVe`, the model is highly affected by the fact that the dataset is unbalanced. This is clearly noticed in the confusion matrices since all the `tn` and all the `fn` are equal to 0. This shows that the models classifies all the test data as being `non_offending` which is the dominant class for each label. This also explains why we have a perfect recall score and really good precision for label 0.

The results of naive Bayes are slightly better with `BoW` since the model generalization is improved and we are predicting `tp` values according to label 1. It's the case for the `obscene` and `toxic` label but not that much better since the number of false negatives (for label 1) is still very high.

Metrics using GloVe

Label 1: toxic

```
                  precision    recall  f1-score   support

        0.0          0.90      1.00      0.95     43150
        1.0          0.00      0.00      0.00      4686

avg / total          0.81      0.90      0.86     47836

Accuracy: 90.0%
```

```
[[43150     0]
 [ 4686     0]]
```

Label 2: severe_toxic

```
                  precision    recall  f1-score   support

        0.0          0.99      1.00      0.99     47346
        1.0          0.00      0.00      0.00       490

avg / total          0.98      0.99      0.98     47836

Accuracy: 99.0%
```

```
[[47346     0]
 [  490     0]]
```

Label 3: obscene

```
                  precision    recall  f1-score   support

        0.0          0.95      1.00      0.97     45225
        1.0          0.00      0.00      0.00      2611

avg / total          0.89      0.95      0.92     47836

Accuracy: 95.0%
```

```
[[45225     0]
 [ 2611     0]]
```

Label 4: threat

```
                  precision    recall  f1-score   support

        0.0          1.00      1.00      1.00     47682
        1.0          0.00      0.00      0.00       154

avg / total          0.99      1.00      1.00     47836

Accuracy: 100%
```

```
[[47682     0]
 [  154     0]]
```

Label 5: insult

```
                  precision    recall  f1-score   support

        0.0          0.95      1.00      0.97     45434
        1.0          0.00      0.00      0.00      2402

avg / total          0.90      0.95      0.93     47836

Accuracy: 95.0%
```

```
[[45434     0]
 [ 2402     0]]
```

Label 6: identity_hate

```
                  precision    recall  f1-score   support

        0.0          0.99      1.00      1.00     47428
        1.0          0.00      0.00      0.00       408

avg / total          0.98      0.99      0.99     47836

Accuracy: 99.0%
```

```
[[47428     0]
 [  408     0]]
```

Metrics using BoW

Label 1: toxic

```
                precision    recall  f1-score   support

          0       0.92      1.00      0.96     43176
          1       0.98      0.15      0.26      4689

avg / total       0.92      0.92      0.89     47865

    Accuracy: 92.0%
```

```
        [[43164    12]
         [ 3995   694]]
```

Label 2: severe_toxic

```
                precision    recall  f1-score   support

          0       0.99      1.00      0.99     47374
          1       0.00      0.00      0.00       491

avg / total       0.98      0.99      0.98     47865

    Accuracy: 99.0%
```

```
        [[47371     3]
         [  491     0]]
```

Label 3: obscene

```
                precision    recall  f1-score   support

          0       0.95      1.00      0.97     45253
          1       0.95      0.09      0.16      2612

avg / total       0.95      0.95      0.93     47865

    Accuracy: 95.0%
```

```
        [[45242    11]
         [ 2384   228]]
```

Label 4: threat

```
                precision    recall  f1-score   support

          0       1.00      1.00      1.00     47711
          1       0.33      0.01      0.01       154

avg / total       0.99      1.00      1.00     47865

    Accuracy: 100%
```

```
        [[47709     2]
         [  153     1]]
```

Label 5: insult

```
                precision    recall  f1-score   support

          0       0.95      1.00      0.97     45462
          1       0.85      0.03      0.06      2403

avg / total       0.95      0.95      0.93     47865

    Accuracy: 95.0%
```

```
        [[45448    14]
         [ 2323    80]]
```

Label 6: identity_hate

```
                precision    recall  f1-score   support

        0.0       0.99      1.00      1.00     47456
        1.0       0.00      0.00      0.00       409

avg / total       0.98      0.99      0.99     47865

    Accuracy: 99.0%
```

```
        [[47453     3]
         [  409     0]]
```

4.4 LinearSVC

Similarly to the naives Bayes, we will see the implementation of the `SVM` model and we will compare the results obtained with both `GloVe` and `BoW`.
Once we have loaded the vectors, we need to split the data.

```
#Preparing Train and Test Dataset, splitting according to 70% train set and 30% test set
features={"toxic","severe_toxic","obscene","threat","insult","identity_hate"}
X_tr={}
X_te={}
y_tr={}
y_te={}
preProcessedTrDF=preProcessedTrDF.reset_index()
for feature in features:
    X_tr[feature], X_te[feature], y_tr[feature], y_te[feature] = train_test_split(embed_vect, preProcessedTrDF[feature].values.astype(np.float), test_size = .3, random_state=25)
y_tr['identity_hate'][41272]=0
```

Just as we did in the previous methods, we need to scale the data.

```
#Scaling the values between the values 0 and 1.
for feature in features:
    min_max_scaler = preprocessing.MinMaxScaler()
    X_tr[feature] = min_max_scaler.fit_transform(X_tr[feature])
    X_te[feature] = min_max_scaler.transform(X_te[feature])
```

Then we define the classifier as MultinomialNB and we train the train-set using the `fit()` function.

```
pipeline={}
for feature in features:
    pipeline[feature] =  Pipeline([('svc', LinearSVC())])
    pipeline[feature].fit(X_tr[feature],y_tr[feature])
```

After that we can start making the predictions and show the metrics. Here we used a loop to go over all the labels at once instead of re-writing the code for each as we did before.

```
#Prediction using test-set and report of statistics.
predicted={}
accF={}
for feature in features:
    predicted[feature] = pipeline[feature].predict(X_te[feature])
```

Below are the results when using `GloVe` first then `BoW`. We can start by saying that this model is better than naive Bayes. We have here better precision, recall and F1-score. Even the confusion matrix look better since we have less misclassification happening compared to the previous method.

Now let's compare the results between using `GloVe` or `BoW`. If we focus on the values of label 1 for each class, we can see that the model is able to classify more samples for each class even though the dominant class is non_offending. However using the `BoW` representation of the comments, we notice that for some classes like `toxic`, `obscene` and `insult`, the value of the `tp` is actually higher than the `fn`. For example the label `toxic` has 3188 samples classified for its class and 1501 are wrongly classified. This means that the model was able to get more than half of the samples. This is a big improvement compared to Naive Bayes.

Metrics using GloVe

Label 1: toxic

```
                 precision    recall  f1-score   support

        0.0         0.93      0.99      0.96     43150
        1.0         0.83      0.32      0.46      4686

avg / total         0.92      0.93      0.91     47836

Accuracy: 93.0%
```

```
[[42835   315]
 [ 3189  1497]]
```

Label 2: severe_toxic

```
                 precision    recall  f1-score   support

        0.0         0.99      1.00      0.99     47346
        1.0         0.59      0.08      0.15       490

avg / total         0.99      0.99      0.99     47836

Accuracy: 99.0%
```

```
[[47318    28]
 [  449    41]]
```

Label 3: obscene

```
                 precision    recall  f1-score   support

        0.0         0.97      1.00      0.98     45225
        1.0         0.85      0.38      0.53      2611

avg / total         0.96      0.96      0.96     47836

Accuracy: 96.0%
```

```
[[45054   171]
 [ 1616   995]]
```

Label 4: threat

```
                 precision    recall  f1-score   support

        0.0         1.00      1.00      1.00     47682
        1.0         0.00      0.00      0.00       154

avg / total         0.99      1.00      1.00     47836

Accuracy: 100%
```

```
[[47682     0]
 [  154     0]]
```

Label 5: insult

```
                 precision    recall  f1-score   support

        0.0         0.96      1.00      0.98     45434
        1.0         0.78      0.26      0.39      2402

avg / total         0.95      0.96      0.95     47836

Accuracy: 96.0%
```

```
[[45252   182]
 [ 1771   631]]
```

Label 6: identity_hate

```
                 precision    recall  f1-score   support

        0.0         0.99      1.00      1.00     47428
        1.0         0.86      0.01      0.03       408

avg / total         0.99      0.99      0.99     47836

Accuracy: 99.0%
```

```
[[47427     1]
 [  402     6]]
```

Metrics using BoW

Label 1: toxic

```
                precision    recall  f1-score   support

          0          0.97      0.99      0.98     43176
          1          0.87      0.68      0.76      4689

avg / total          0.96      0.96      0.96     47865

Accuracy: 96.0%
```

```
[[42701   475]
 [ 1501  3188]]
```

Label 2: severe_toxic

```
                precision    recall  f1-score   support

          0          0.99      1.00      1.00     47374
          1          0.56      0.25      0.34       491

avg / total          0.99      0.99      0.99     47865

Accuracy: 99.0%
```

```
[[47277    97]
 [  369   122]]
```

Label 3: obscene

```
                precision    recall  f1-score   support

          0          0.98      0.99      0.99     45253
          1          0.88      0.71      0.79      2612

avg / total          0.98      0.98      0.98     47865

Accuracy: 98.0%
```

```
[[45008   245]
 [  758  1854]]
```

Label 4: threat

```
                precision    recall  f1-score   support

          0          1.00      1.00      1.00     47711
          1          0.65      0.17      0.27       154

avg / total          1.00      1.00      1.00     47865

Accuracy: 100%
```

```
[[47697    14]
 [  128    26]]
```

Label 5: insult

```
                precision    recall  f1-score   support

          0          0.98      0.99      0.98     45462
          1          0.78      0.56      0.65      2403

avg / total          0.97      0.97      0.97     47865

Accuracy: 97.0%
```

```
[[45093   369]
 [ 1063  1340]]
```

Label 6: identity_hate

```
                precision    recall  f1-score   support

        0.0          0.99      1.00      1.00     47456
        1.0          0.67      0.22      0.34       409

avg / total          0.99      0.99      0.99     47865

Accuracy: 99.0%
```

```
[[47410    46]
 [  317    92]]
```

Conclusion

Throughout this project, we have successfully built several models which can classify a text into one or more labels according to different sentiments.
Running these models and looking at their results and performance, we obtained several outcomes that are worth being discussed.

The pre-processing and transformation to embedding usually resulted in improvement but for some cases the baseline model of the `BoW` performed better than for example `GloVe` which was trained using some other corpora in addition to the text of our dataset. This was a very interesting outcome since we assumed that more complex embedding should always result in better outcomes. However this was not case here and the results we have prove it.

Furthermore, if we look at each label on its own, we can see that the models performed differently but generally labelling `threat` was the most successful job with 100% accuracy on test set by most of the classification methods. `identity_hate` mainly had a classification accuracy of 99% accuracy, `severe_toxic` label can be classified with 99% accuracy. Obscene comments can be labelled using 98% accuracy and Insult Label can be classified with 97% accuracy with `LinearSVC` and `BoW`.
The least performing classification is for the label `toxic` resulting in only 96% accuracy rate using `LinearSVC` and `GloVe` embedding with paragraph vector. Again here we are surprised that the `BoW` performs better than the `GloVe` since we were expecting the opposite to be true. It is clear that `LinearSVC` with `BoW` performed best for the label classification overall and for each label followed by the neural network.

Improvements

As mentioned before the model that performed best for this dataset and accross all labels is `LinearSVC` with `BoW`. From this point, we can focus on this model and try to enhance its performance. One way of doing that would to train non-linear Kernels.
Other models can be enhanced by tweaked their parameters and other properties like neural networks.

Another area of possible improvements is at the level of preprocessing. The pre-processing part included just tokenization and as a further improvement the system can be enhanced by adding Part of Speech Tags, or by adding the dependency parse trees as features, this might help the model to perform better. Also we can use bigrams or trigrams instead of unigrams as mentioned earlier in the report.

It's worth mentioning that another way to improve our models performances is to use ensemble methods to combine those single models and then average the results, in order to take advantage from all of them and have a more generalizable view of the problem.

Tasks repartitioning

Although we are allowed to a maximum of four persons, we decided to work the three of us together for better communication, collaboration and efficient team work.

Elena: Exploring the dataset and Logistic regression with a contribution to the report

Fatih: Pre-processing, GloVe, BoW, Naive Bayes and LinearSVC with a contribution to the report

Boutros: Word2Vec and Neural Networks with a contribution to the report

Implementation

The code is uploaded with the report and on `https://github.com/QMULDataAnalytics/QMULHub`.

References

[1] Toxic comment classification challenge. https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge.

[2] Pyplot api. https://matplotlib.org/api/.

[3] Natural language toolkit. http://www.nltk.org.

[4] Word2vec: Neural word embeddings for natural language processing. https://deeplearning4j.org/word2vec.html.

[5] Glove: Global vectors for word representation. https://nlp.stanford.edu/projects/glove/.

[6] Glove implementation details. http://dsnotes.com/post/fast-parallel-async-adagrad/.

[7] Bag of words and tf-idf. https://deeplearning4j.org/bagofwords-tf-idf.

[8] Introduction to neural networks with scikit-learn. http://stackabuse.com/introduction-to-neural-networks-with-scikit-learn/.

[9] sklearn neural network mlpclassifier documentation. http://scikit-learn.org/stable/.