Natural Language Processing – Toxic Comment Classification

Qimo Li, Chen He, Kun Qiu

Professor Marc Verhagen

December 2019

Environment

```
nltk==3.4.5
numpy==1.17.4
pandas==0.25.3
scikit-learn==0.22
```

How to run

```
$cd #your directory
$python main.py
```

1 Introduction

The advanced technology enables us to communicate with each other through different social media platforms. However, the threat of harassment can make people stop expressing themselves or listening to different opinions. Due to the lack of methods to control these toxic comments, many people choose to limit or even shut down user comments.

Our final project built a multi-headed model that can detect different types of toxicity like threats, obscenity, insults and identity-based hate. The original dataset is from Wikipedia's talk page edits. The accomplished model can be used to detect toxic comments automatically, purify online communication environment and make online discussions more respectful and productive.

The challenge was initially launched by Kaggle in 2018 at https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge, so we can get access to the results to investigate the accuracy of our model test.

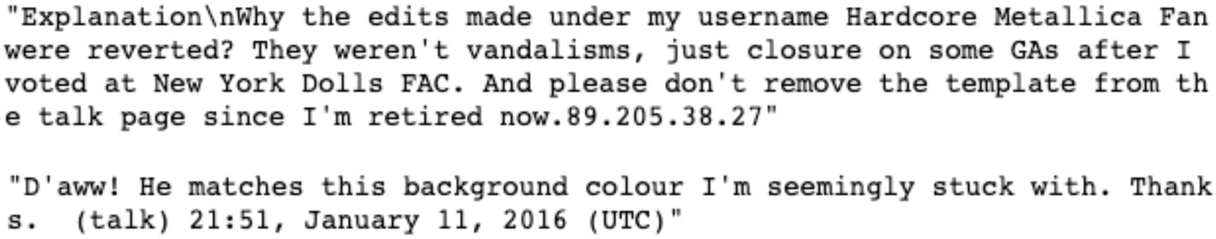
Our final project can be found on the GitHub repository at https://github.com/Kimonokimo/NLP-comment-project. Chen He successfully contributed her tasks on the data visualization and data analysis, Kun Qiu worked on the preparation process, and Qimo Li finished the feature creating and built the model.

2 Exploratory Data Analysis on Comment Data

2.1 Data Overview

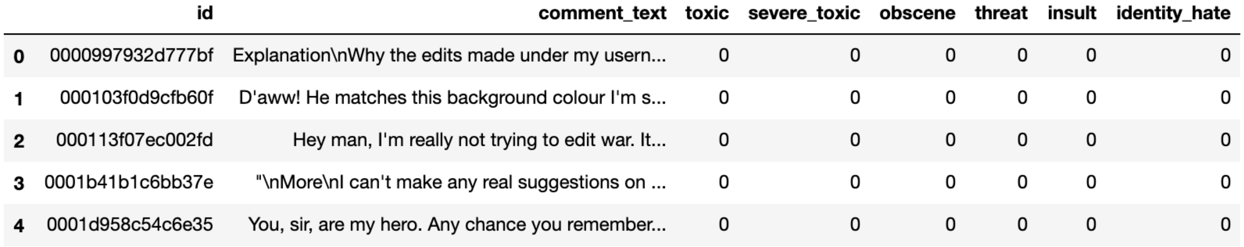
The data used to train and test the classifiers are contained in the train.csv file. The data file is exported from the “Toxic Comment Classification” Kaggle competition. The data file stores a large number of Wikipedia comments that have been labeled by human raters for toxic behavior. The csv file consists of 159571 rows and 8 columns, in which each row represents a single comment.

The text below showcase the format in which the comments are stored in the train.csv file.



```
"Explanation\nWhy the edits made under my username Hardcore Metallica Fan
were reverted? They weren't vandalisms, just closure on some GAs after I
voted at New York Dolls FAC. And please don't remove the template from th
e talk page since I'm retired now.89.205.38.27"

"D'aww! He matches this background colour I'm seemingly stuck with. Thank
s.  (talk) 21:51, January 11, 2016 (UTC)"
```

In other words, there are in total 159571 comments stored in the dataset, and each of them has a unique id and 6 class labels. The following table shows the first five rows in the train.csv dataset.

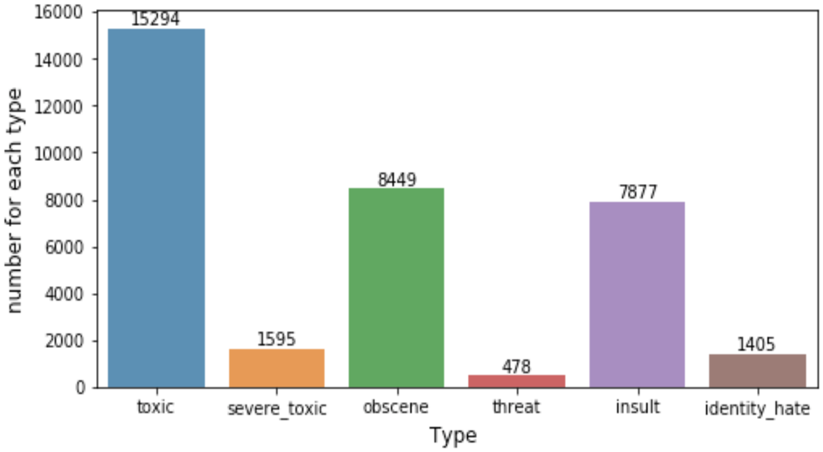


	id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
0	0000997932d777bf	Explanation\nWhy the edits made under my usern...	0	0	0	0	0	0
1	000103f0d9cfb60f	D'aww! He matches this background colour I'm s...	0	0	0	0	0	0
2	000113f07ec002fd	Hey man, I'm really not trying to edit war. It...	0	0	0	0	0	0
3	0001b41b1c6bb37e	"\nMore\nI can't make any real suggestions on ...	0	0	0	0	0	0
4	0001d958c54c6e35	You, sir, are my hero. Any chance you remember...	0	0	0	0	0	0

As the table shows, there are six predefined types of toxicity in the dataset, “toxic”, “severe toxic”, “obscene”, “threat”, “insult”, and “identity hate”. These toxicity types are encoded as Boolean values, in other words, 0 stands for “no” and 1 stands for “yes”. Notice that the comments are multi-labeled, which means one comment can fall into more than one category of toxicity behavior. For example, a comment can be both toxic and threat. Therefore, we decided to build an ensemble classification model that consists of six sub-models, one for each type of toxicity.

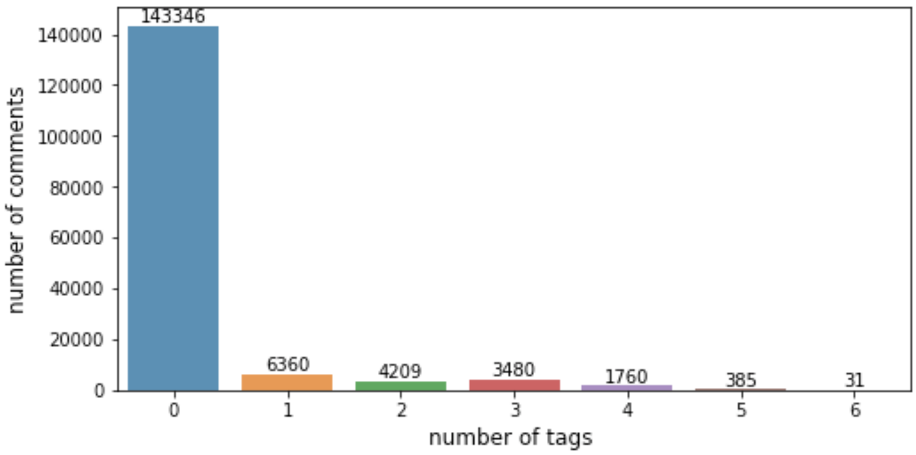
2.2 Data Visualization

In order to gain a better understanding of the data set, we visualized the data to get some insights. There are 159571 comments in the data set and each is tagged with different labels. The following chart shows the number of comments fall into each category of toxicity behavior using a bar plot.



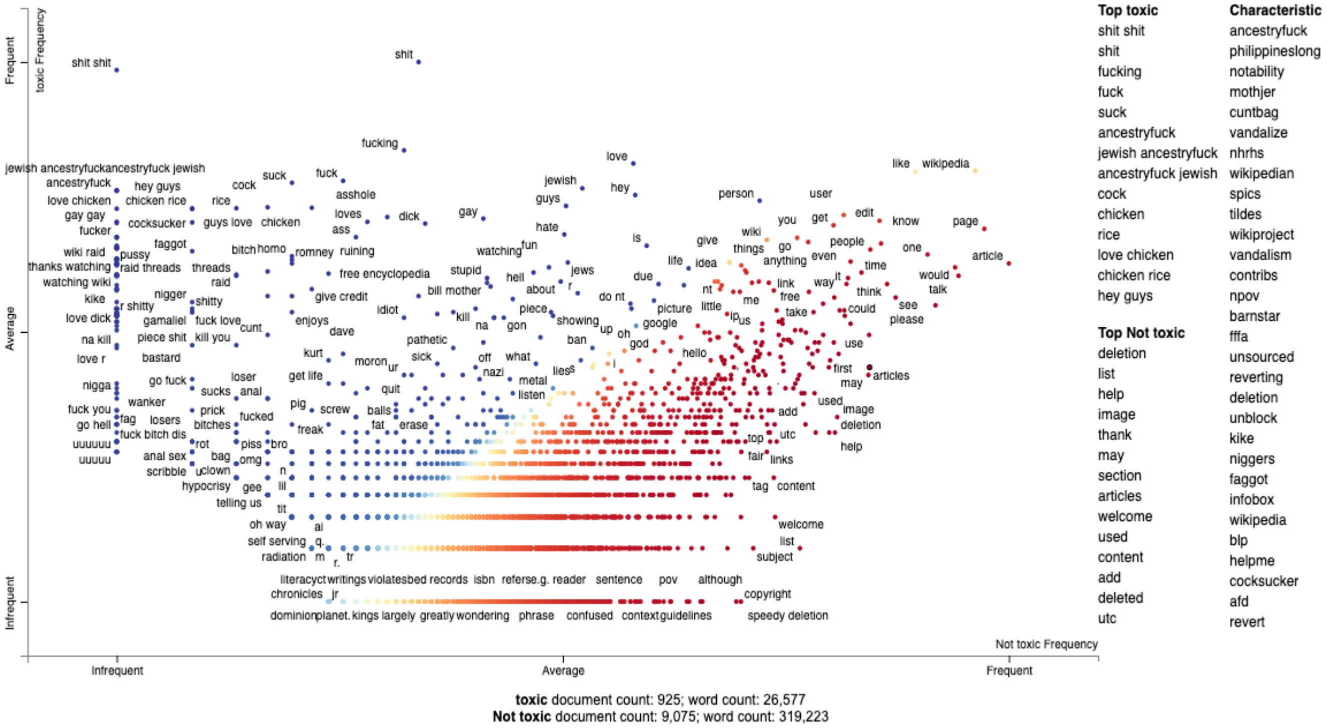
The chart shows that most of the comments are labeled “toxic”, followed by “obscene” and “insult”, which rank second and third in terms of frequency. The result is not too surprising. “Toxic” is a broad concept and includes other types of toxicity behavior. Also, “obscene” comments and “insulting” comments are more commonly observed in the online environment than “threat” and “identity hate”, the pattern of the comment data is consistent with our daily observation.

As the comments are multi-labeled, we are interested to know how many comments do not have any labels and how many of them are tagged all six labels. The following bar plot presents the information we are interested in.

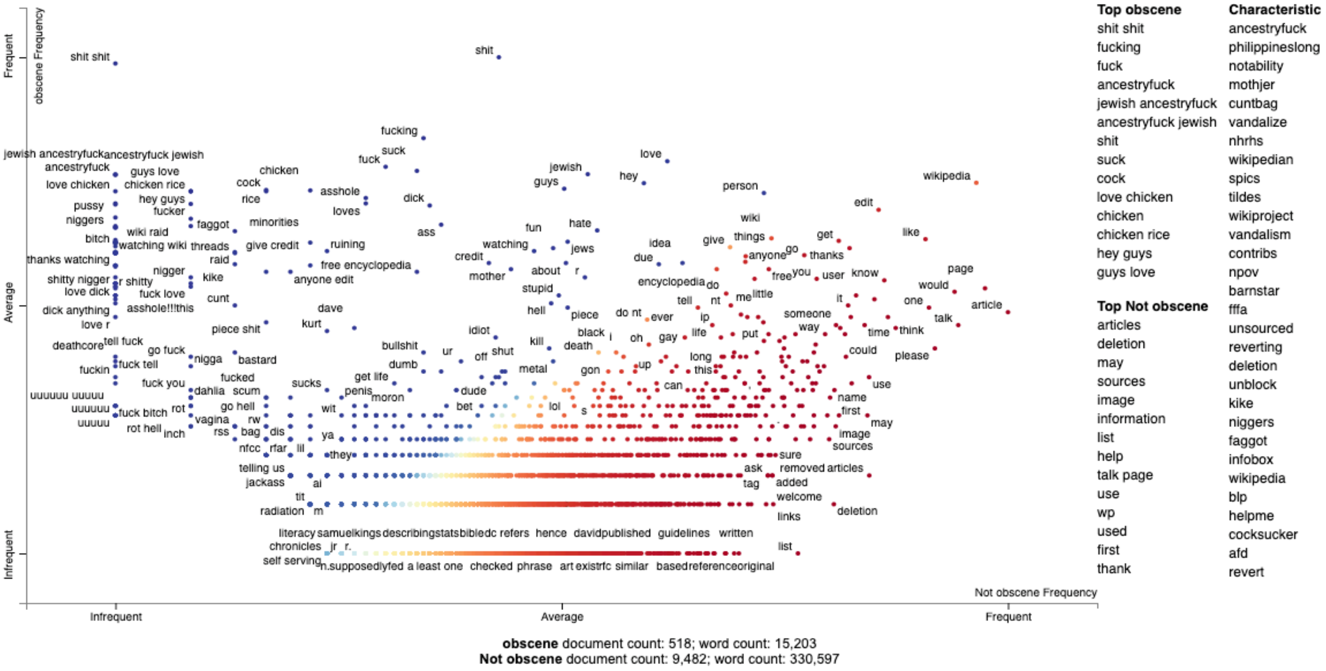


From the above chart, we can see that the data are highly imbalanced, with only 10% of the comments are toxic. The majority of the comments have 0 labels, with means they are not toxic comments. To ensure the stability of the machine learning algorithm's performance, we will need to deal with the imbalanced data.

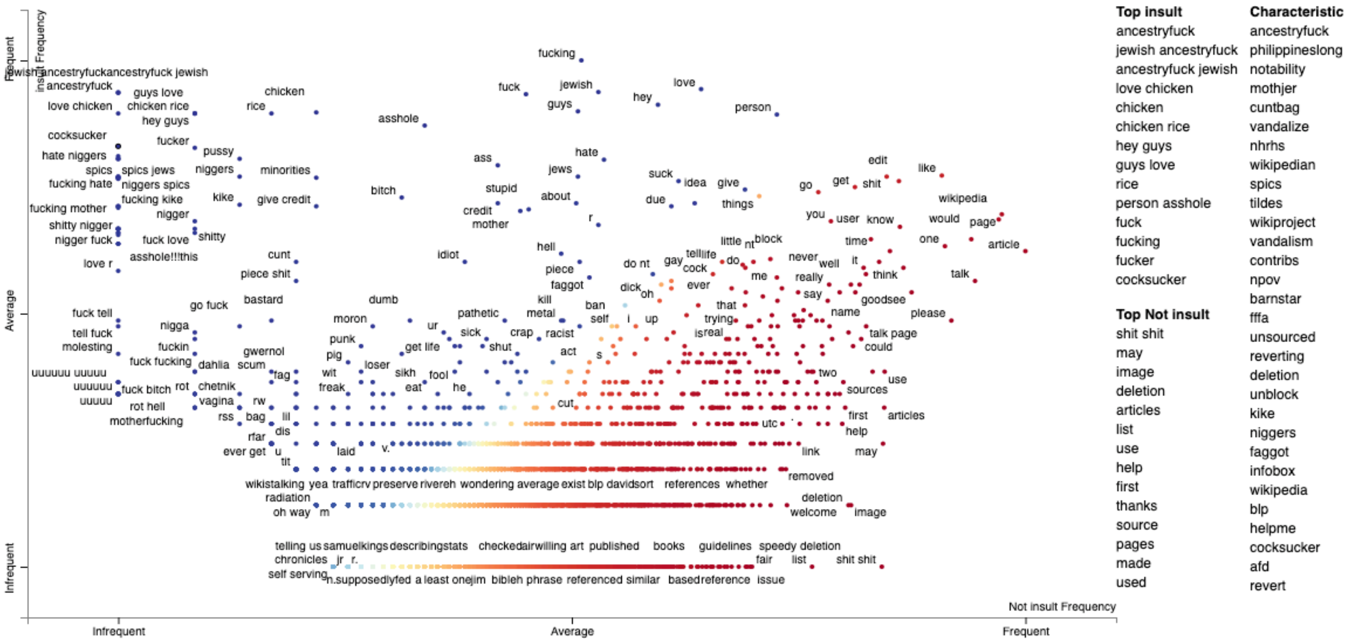
Apart from the labels, we also want to gain some insights on the text data itself. The following scatter plots show the most frequency words in different categories and their frequency.



The above plot presents the most characteristic terms used in comments that are labelled “toxic” and those that are not labelled “toxic”. The most frequently used terms in toxic comments includes words with strong negative emotion like “shit”, “fucking”, “fuck”, “suck” and etc., while those used in comments that are not labelled “toxic” includes neutral words like “deletion”, “list”, “image”, and some positive words like “help” and “thank”. The speech set used by these two groups are significantly different, which means we are safe to say that it is representative to use the vocabulary extracted from the whole text data as our feature set.

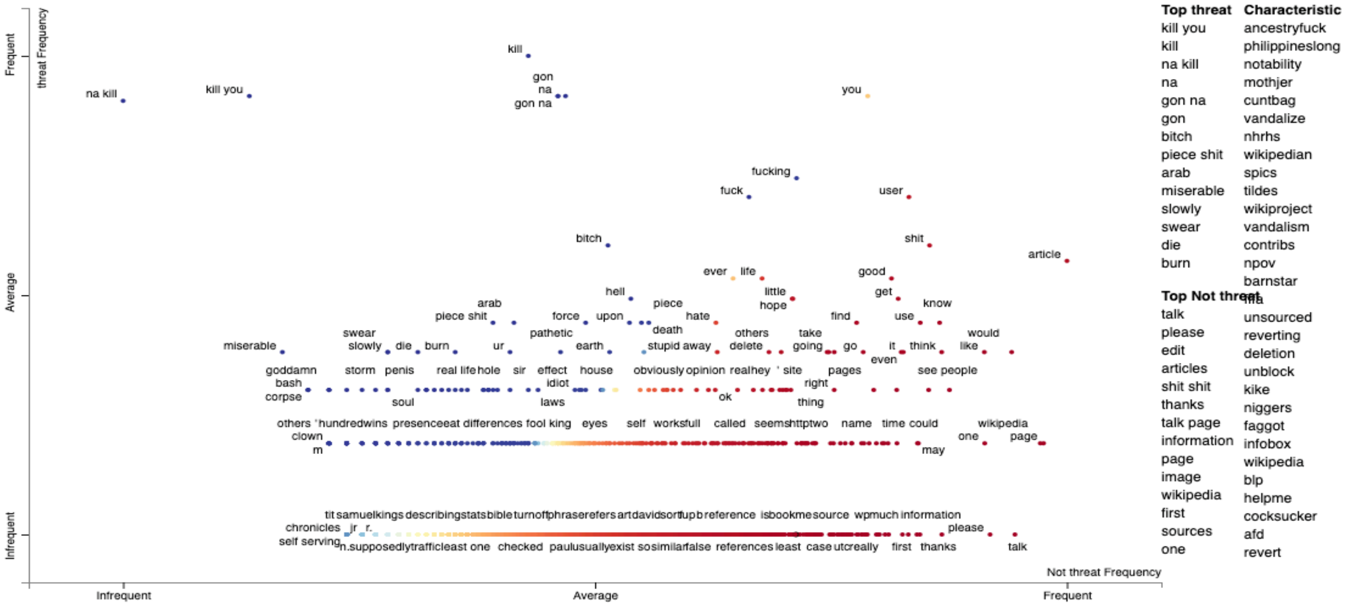


This plot shows the words that are most frequently found in “obscene” comments and its control group. Compared to the previous plot, the top obscene words and the top toxic words overlap one another. One hypothesis is that lots of toxic comments are labelled “obscene” at the same time. The overlapping will bring extra challenge for us to differentiate between these two classes.



insult document count: 496; word count: 12,224
Not insult document count: 9,504; word count: 333,576

Above is the plot that highlights the top words used in "insulting" comments. As we can see, the distribution of characteristic words in this plot is very similar with last plot ("obscene vs not_obscene"). The boundary between "insult" and other types of online harassment is hard to determine. That is probably the reason why they have similar patterns.



threat document count: 32; word count: 839
Not threat document count: 9,968; word count: 344,961

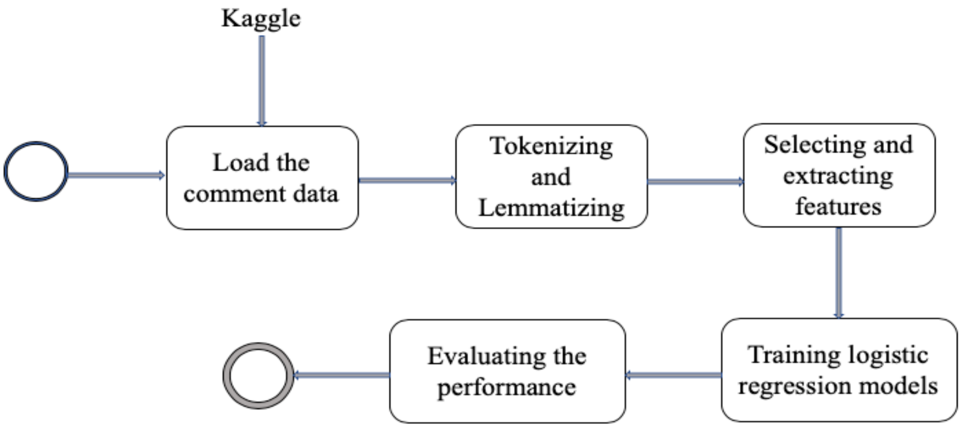
Although fewest observation fall into this category, the characteristic terms used by "threatening" comments have fairly unique pattern. The frequent terms include "kill", "kill you", "gonna kill", and etc. While the control group uses more positive terms like "please", "thanks" and neutral words like "talk", "edit" and "articles". Since this class has significantly different patterns compared to the other classes, the classifier may have better performance when predicting the labels for "threatening" comments.

There are some other visualizations we are not able to include due to the limit of length. The script and the results can be found in the visualization.ipynb file and under the visualization folder.

3 Code Explanation

The workflow of this project can be roughly divided into three phases: data preparation, which includes text preprocessing and basic data exploratory analysis; feature generation, which includes feature selection and feature extraction; model construction, which includes data partition, initial model construction, model optimization and cross-validation.

The following progress model depicts the designed workflow of this project.


Kaggle
Load the comment data
Tokenizing and Lemmatizing
Selecting and extracting features
Training logistic regression models
Evaluating the performance

In the following paragraphs, we will elaborate on how we implement each phase in detail.

In our NLP-comment-project repository, we have 7 files and 1 data repository.

(1) Preparation

Tokenizer.py

Pos_tagger.py

Lemmatizer.py

Dataclean.py

(2) Feature Creating

Vocabulary.py

Vector.py

(3) Model building

Main.py

In each python file, we built up a test function (except vector which is meaningless to do a single test). Test functions helping us to check if our classes and self-def functions work well. Otherwise, it will be very hard to debug inside the main.py file. Especially, main.py is different from others because it imports other self-def packages and uses their classes and functions to do the data preprocessing, model training and result analyzing.

To test each class just simply type:

```
$python XXXXXX.py
```

in the command line, and you may see the testing results.

3.1 Preparation

Dataclean.py

This file has a DataClean class that is used to preprocessing the raw data. The main purpose is to remove punctuation marks, numbers and abbreviation reduction. The main package we use here is 're' which is a convenient tool for string process. The function takes text as input and return the text after processing. After processing, sentences ended with punctuation marks like '.', '!' and '?' will be kept. The function will also change abbreviation reduction to its original form. For example, 'he's' will be changed into 'he is' and 'isn't' will be changed into 'is not'. Therefore, we can use those clean texts to do sent_tokenize() and tokenize().

Tokenizer.py

This file contains a Tokenizer class and test functions. The Tokenizer class has two main functions, which are tokenize() and sent_tokenize(). Both of them take 'text' type data as input. However, their returns are different. The tokenize() function will return the token list because it is simply implemented by the python in-built string.split() functions. Sent_tokenize() also takes text as input, but it will return a sentence list. The sentences are split by '.', '!' and '?',which means the boundary of a sentence. Therefore, we can implement the sent_tokenize by this rule. Unfortunately, it is the easiest implementation and online comments can not be sample like this. It means sometimes it may have a bad performance. Further optimization is needed to enable the function to have a better performance.

Pos_tagger.py

This file contains functions used to train a pos tag classification model. The dataset used to train our model here is called treebank from nltk.corpus which has 3194 tagged sentences. The model we selected is random forest. Random forest is an ensemble model based on decision tree, but it has a much better preference.

This file has three main functions. They are build_features, get_data_label and train_pos_tag. Build-features creates features for each word in sentences. It takes sentence word index and label as input and returns a python dict which contains features' names and values. The index is the position of a certain word in a sentence. Label is Boolean value. It tells whether it is a train data or test data. The get data label function uses build features go through the whole comments list and return a features list. The last and most important function is train_pos_tag. It takes nothing as input and outputs a classification model and a onehot encoder. Later we will use it to encode our comments and use the model to predict the pos tag for each word in our sentences.

Lemmatizer.py

Our lemmatizer class is very simple, implemented by wordnet's inbuilt function 'morphy'. The implementation of this function is quite similar to nltk's inbuilt lemmatizer - it takes word and pos as input, and returns the word after lemmatizing or itself if it can not lemmatize. We can get the pos tag from our pos_tagger model.

3.2 Feature Creating

Vocabulary.py

Just as its name, this class builds a vocabulary for our dataset. It has some basic functions like pos, idf and stop word remover. It creates the vocabulary with text list. Each text in text list a piece of comment. The __init__ function will build a word reference which is a python dict. Keys are words and value is its position which used to encode comments. The pos function uses word reference dict to return the position of the word.

We also have an idf function in this class. It takes nothing as input and output the idf values for each word in our vocabulary. The return of this function is a dict. Keys are the same as the word reference dict, and its values are word's idf results. This function simply goes through every word in our comment list and count their appear times. Maybe it is not a good implementation. It will take 3-4 minutes for a 10k size dataset.

Moreover, there is another class called FileReader is this python file. It takes the file path as input and reads the csv file as a pandas dataframe which is a powerful tool for data handling. This class has two sample functions, called get text and labels. They return a NumPy array of comments and a NumPy array of tags respectively.

Vector.py

We created a words vector for each comment. However, this class is almost empty. The most important function here is tf which is used to calculate the tf value for every word in comment. It returns a NumPy array. The Part-Of-Speech of words in this array is based on the word reference dict in vocabulary class.

3.3 Model Building

Main.py

Besides the main() function, we also have two data split functions here. The first one is the train_test_split(). To implement the function, we should enter 3 inputs, including our original dataset array after preprocessing, labels arrays and test size. Test size is a float value between 0 and 1, and the size of the testing set is based on this value.

The second split function is called k_fold(), which is used for cross-validation. It takes the train set from train_test_split function and number k as input. In this function, we built an index list and split the index inside based on number k. It will return a fold list at last. Based on this fold list, we can split our data into the train and validation dataset easily. We also have a tag_map function since the tag predicts from our tree model is different from the required tag in lemmatizer. This function takes a treebank tag as input and output a wordnet tag.

The main function uses our self-building classes and functions to preprocess, split, encode, clean, train, predict, and analyze our dataset. Also, It prints out some running information and will help people better understand what our program is doing. What's more, we used validation set to choose the best threshold for our model. We simply let out model go through all thresholds in our threshold list and pick the one with highest performance.

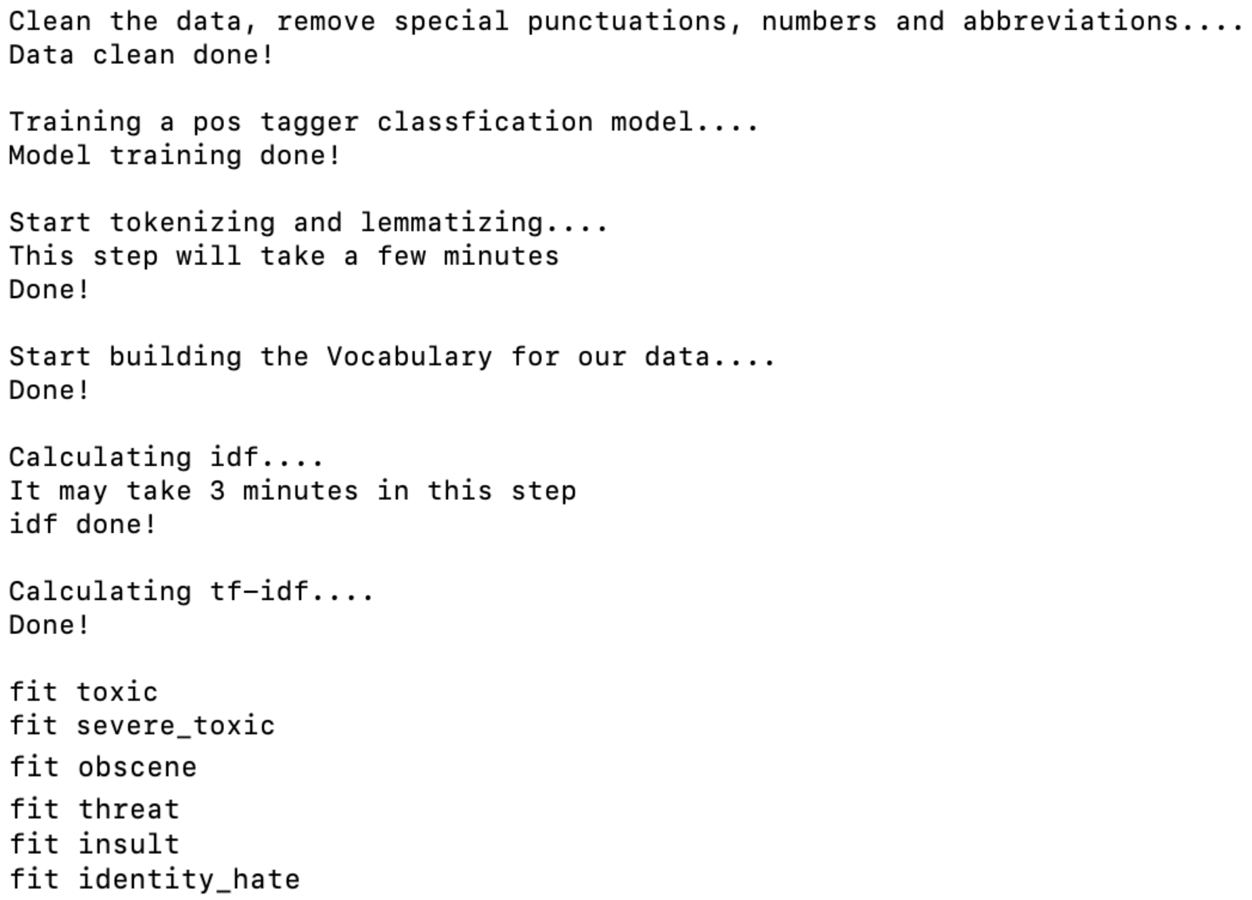
4 Conclusion

In this project, we successfully developed a model to detect potential toxic comments, the performance of our model is perfect and accurate. Most of the test set accuracies are more than 95% and for some classes it is almost 99%. However, our code may take a long time to run since we have re-created most of the nltk functions. In our environment, it took almost 10 minutes to run the whole code.

We hope to keep working on this to optimize our existing modules. As we said before, the tokenizer() may require further work to be suitable for more complex sentence types, and we may simplify our algorithm structure so that it may take less time to finish the task. After all, we gained a lot of experience from this practical natural language processing project, and we will continue optimizing it in the future.

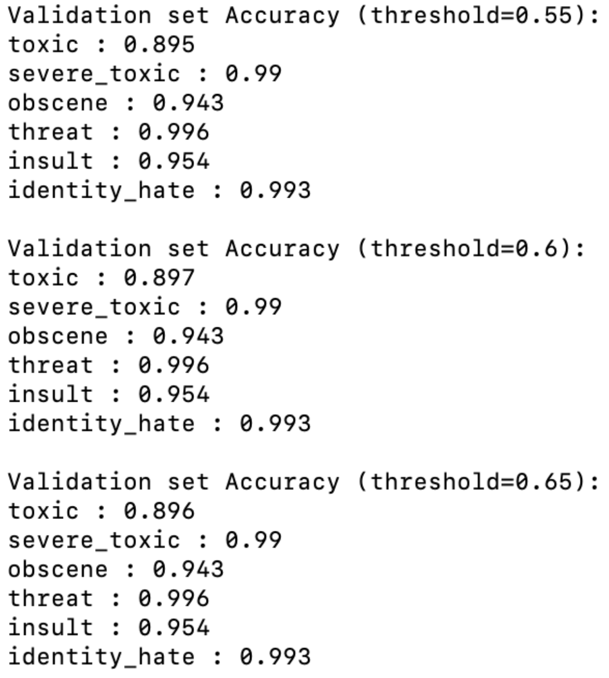
Attachment

Print out information:

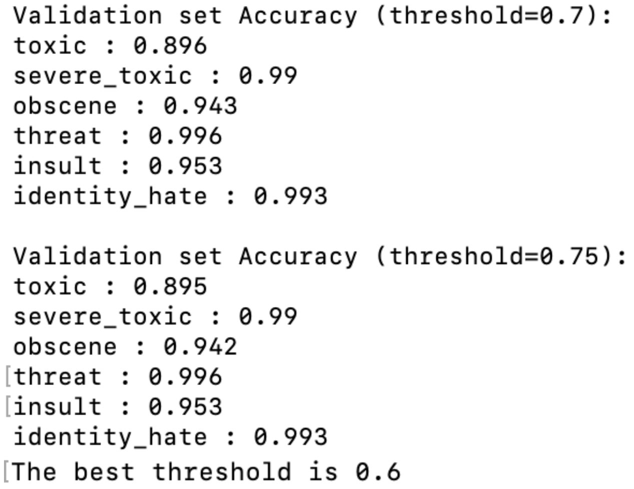


```
Clean the data, remove special punctuations, numbers and abbreviations....
Data clean done!

Training a pos tagger classfication model....
Model training done!

Start tokenizing and lemmatizing....
This step will take a few minutes
Done!

Start building the Vocabulary for our data....
Done!

Calculating idf....
It may take 3 minutes in this step
idf done!

Calculating tf-idf....
Done!

fit toxic
fit severe_toxic
fit obscene
fit threat
fit insult
fit identity_hate
```

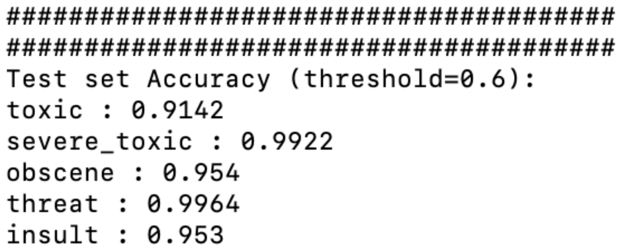
Print out the result:



```
Validation set Accuracy (threshold=0.55):
toxic : 0.895
severe_toxic : 0.99
obscene : 0.943
threat : 0.996
insult : 0.954
identity_hate : 0.993

Validation set Accuracy (threshold=0.6):
toxic : 0.897
severe_toxic : 0.99
obscene : 0.943
threat : 0.996
insult : 0.954
identity_hate : 0.993

Validation set Accuracy (threshold=0.65):
toxic : 0.896
severe_toxic : 0.99
obscene : 0.943
threat : 0.996
insult : 0.954
identity_hate : 0.993
```



```
Validation set Accuracy (threshold=0.7):
toxic : 0.896
severe_toxic : 0.99
obscene : 0.943
threat : 0.996
insult : 0.953
identity_hate : 0.993

Validation set Accuracy (threshold=0.75):
toxic : 0.895
severe_toxic : 0.99
obscene : 0.942
threat : 0.996
insult : 0.953
identity_hate : 0.993
The best threshold is 0.6
```



```
########################################
########################################
Test set Accuracy (threshold=0.6):
toxic : 0.9142
severe_toxic : 0.9922
obscene : 0.954
threat : 0.9964
insult : 0.953
```