Cyber Abuse Detection System

Nihar Dalmia, Suchismita Padhy, Rohit Raghavan, Nia Vivekanandan

December 2016

BACKGROUND

When using social media, people intentionally or unintentionally post hurtful or insulting comments or tweets towards people which could have been construed by them as a joke or harmless text. But many people are sensitive to insulting or hurtful comments and it might impact them in ways which are unconceivable to the person posting the comment. In such cases, people might prefer to use a self-awareness mechanism which would warn them of potential hurtful or insulting content in their text before posting. Another use of this project is to use Twitter handle to identify the number of insulting tweets among the most recent 1000 tweets. This allows people to check their own as well as other tweets to be aware of the content posted by them as well as by others.

PROJECT GOALS

Original intent

The original intent of this project was to create a system to automatically detect insults by using several NLP techniques we studied in the course.

How far we got

We developed a system that would automatically detect if the entered text is insulting or not. We also included an option where a twitter handle could be entered to list the number of insulting tweets and also retrieve the top 10 insulting tweets.

What future work would be if this were to be continued

We noticed that there is a little noise in our system which is impacting the way our classifier is performing. We plan to expand our dataset to include more social media messages for a better and more complete training set. We also want to address our limitations by taking into account sarcasm, dog whistle terms and differentiating

between jokes and insults. We also wish to further classify the type of insult as being racist, sexist, or an instance of bullying or profanity.

Results / Evaluation

We tested the performance of our system based on 2 metrics, one is the accuracy of the model on the test data set and the other is how well our system is performing for new data set entered by the users.

DESCRIPTION OF DATA

Data Acquisition:

Data Set - Kaggle detecting insults dataset

Files used:

1. train.csv
2. test_with_solutions.csv

The data consists of 3 columns - Insult, Date and Comment.

DESCRIPTION OF OUR ALGORITHMS

Data Preprocessing:

Social media messages need not follow all the formal rules of written English. Thus we needed a concrete way to clean up the data and convert slang or online english to formal english as much as possible. Following steps were taken in the data preprocessing stage:

1. Tokenizer: We tested our algorithm with two tokenizers, firstly we added a custom tokenizer using regular expression to capture social media tags like mentions, hashtags, URLs, a group of symbols together and emoticons. The second tokenizer we used was the prebuilt NLTK tweet tokenizer. Currently the tweet tokenizer gives slightly better results than the custom tokenizer, and thus we decided to go ahead with that.
2. Spell Checker: Implemented a spell check mechanism where the following conditions for misspellings are checked and corrected with high accuracy:

a. Extra letters added in a word which need to be deleted. For example ‘waay’ will be corrected to ‘way’
b. Transpose of letters within words. For example, incorrect spelling ‘beleive’ will be corrected to ‘believe’
c. Replacement of certain letters for another letter altogether. For example, for our application ‘f*ck’ will be replaced as ‘fuck’
d. Inserting letters that may have been missed in the word: For example, the word ‘asume’ will be replaced by ‘assume’
e. Handling repeated letters to get the correct spelling. For example, ‘heeeeyyyyy’ will be replaced by ‘hey’

How does the spell checker work?

The spell checker was adopted from (http://norvig.com/spell-correct.html) but several changes needed to be made for it to suit our application. The code gets its dictionary of words along with the word frequency from a file called big.txt. This file is a collection of texts from Project Gutenberg, and a list of most frequent words from Wiktionary and British National Corpus. If a word is passed to the program, the program will first check if the word is present in the dictionary or not. If it is present, then the word is passed as is. If it is not present then all possible combinations (deletion, insertion, replacement and transpose) of the misspelled word are evaluated. Out of these set of possible spelling, the word with the highest frequency present in the dictionary is chosen as the correct spelling.

How we increased the efficiency of the spell checker?

Initially the results that we were getting from the spellchecker were very poor. Oftentimes, the spell checker would correct abusive words to non-abusive words (for example, haters was getting corrected to waters), or it would convert short acronyms to words which had no relation. Certain english words like ‘mom’ or

'wanna' were not present in our word dictionary and thus they ended up being wrongly modified. Thus it meant that we had to iteratively add certain words to big.txt, or change the weightage of each word, or add certain correction condition for words to get a higher level of accuracy. Thus, to the dictionary generated from big.txt, we further added words from nltk corpus and a custom list of abuses that we had prepared for our project along with a high frequency value.

What does not work ?

The correction algorithm right now heavily depends on the presence and the frequency of words in big.txt. Given that the words in big.txt are from corpora from a long time ago, and new words with higher usage frequency of usage have come in. We need to find newer and bigger corpora to feed to our spell checker logic to get better results.

3. Stopwords: A lot of the general stopwords were extremely important for our code. For example, stop words like you, your, you are were important words to consider whether the abuse was directed or not. Even "wh" words like who, what, why formed important parts of our features. Thus we decided to limit our list of stopwords to the articles a, an and the.
4. Lemmatization/ Stemming: We tried the NLTK WordNet Lemmatizer and the Porter Stemmer and found out that the former gave us slightly better results and decided to go ahead with that.

Feature Extraction:

Word2vec:

In word2vec we convert every word to a vector such that words that share common contexts in the corpus are located in close proximity to one another in an n-dimensional space. The word2vec is trained on a very large corpus of text. For our project, we initially started with a model trained on wikipedia corpus, but there were many out of

vocabulary words. Hence, we chose a word2vec model that was trained on a dataset containing 27 billion tweets.

The next task was to get a feature vector representation for the every tweet in our dataset. In order to achieve this, we built a tweet vector by adding the individual word vectors and then taking their average. The averaging operation helped to minimise the effect of variable tweet length on our final representation.

What worked?

It was interesting to observe that with only 200 dimensions we could get comparable performance with a 2500 dimensional tf-idf features.

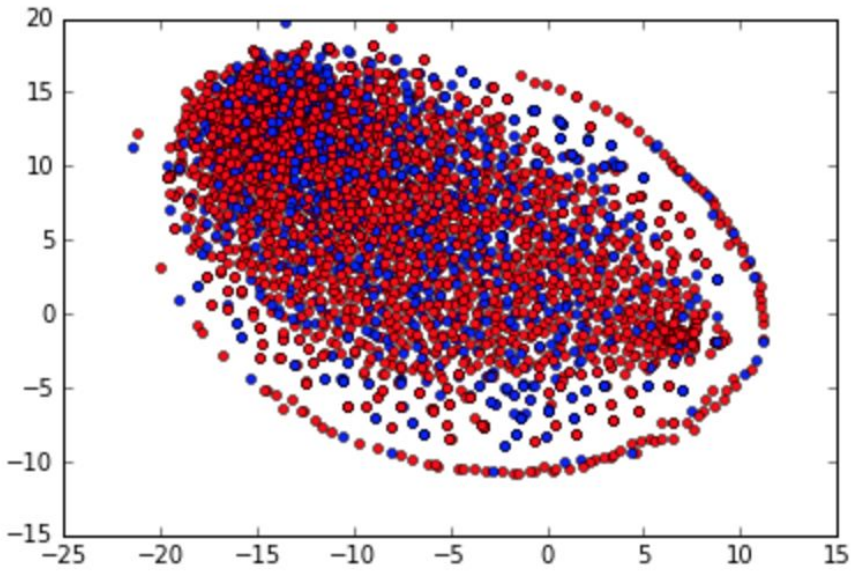


Figure: The word2vec Transformations in 2-Dimensions(Blue-Insults)

What did not work?

This method of feature extraction was not giving us the best accuracy. The main reason was, the approach adopted while building the tweet vectors was not optimal as we are not capturing the sequence information. Instead of using the bag-of-word approach that we followed, using the word embeddings along with sequence information might have been more successful for our task.

Tf-idf:

To generate tf-idf features for our dataset, we used the TfidfVectorizer module in scikit-learn's sklearn.feature_extraction.text package. We used a collection of n-grams with n ranging from 1 to 6. In order to account for misspellings, we also considered character n-grams with n ranging from 1 to 6. We removed stop words (a, an, the) since we felt that those words didn't add any additional value to our features. We also found that including all the features extracted by tf-idf had much higher accuracy than using a limited number of features.

Custom Features:

We created a class AdditionalFeatureExtractor which has methods for our custom features:

1. **Percentage of bad words:** We used a list of bad words to count the number of occurrences of bad words in a sentence in our data set. We then normalized this number with the total number of words in the sentence. We used this normalized number as our feature vector.

2. **Capturing directed communication:** Similar to #1, we used a list of bad words to identify whether a sentence in the data set contained bad words. If it did, and it contained different variations of the personal pronoun you (you, you're, youre, you are, u r, ur) and didn't contain the word not, we considered that as a directed insult. If the word didn't contain a directed insult, our feature vector would have a representation of 'False'.

3. **Sentiment analysis using vader:** We used the Vader SentimentIntensityAnalyzer() method to create a vector of different polarity scores for the sentences in our data set. We created vectors of using values of negative sentiment, positive sentiment, neutral sentiment and compound sentiment.

The above custom features were stacked with each other along with the features from tf-idf and word2vec using FeatureUnion() method to create a union of all the above features.

Training a classifier

We tried a number of classifier models such as Logistic Regression, Neural Networks, Naive Bayes linear SVM for this dataset. We also tried to build an ensemble of these classifiers using soft voting.

Tuning the hyperparameters :

To come up with the optimal set of hyperparameters, we used grid search coupled with cross-validation score. This technique allowed us to exhaustively consider all combinations of a manually specified subset of hyperparameter space. The parameters selected are those that maximize the score of the left out data.

K-fold cross-validation :

We tuned these models using stratified k-fold cross-validation. In every fold, the whole dataset is randomly split into training and validation sets. This process is repeated k times and ensures that our model does not overfit to a particular subset of the data. But we did not observe a lot of inconsistency with this approach. This could be due to the fact that there is not much imbalance between insulting and non-insulting comments.

What worked :

Linear SVM seems to work best for this dataset.

What did not work :

We also tried K-means clustering by clustering the dataset into 2 clusters, but the clustering was not successful and insulting content was present on both the clusters

indicating that clustering does not work well for this dataset.

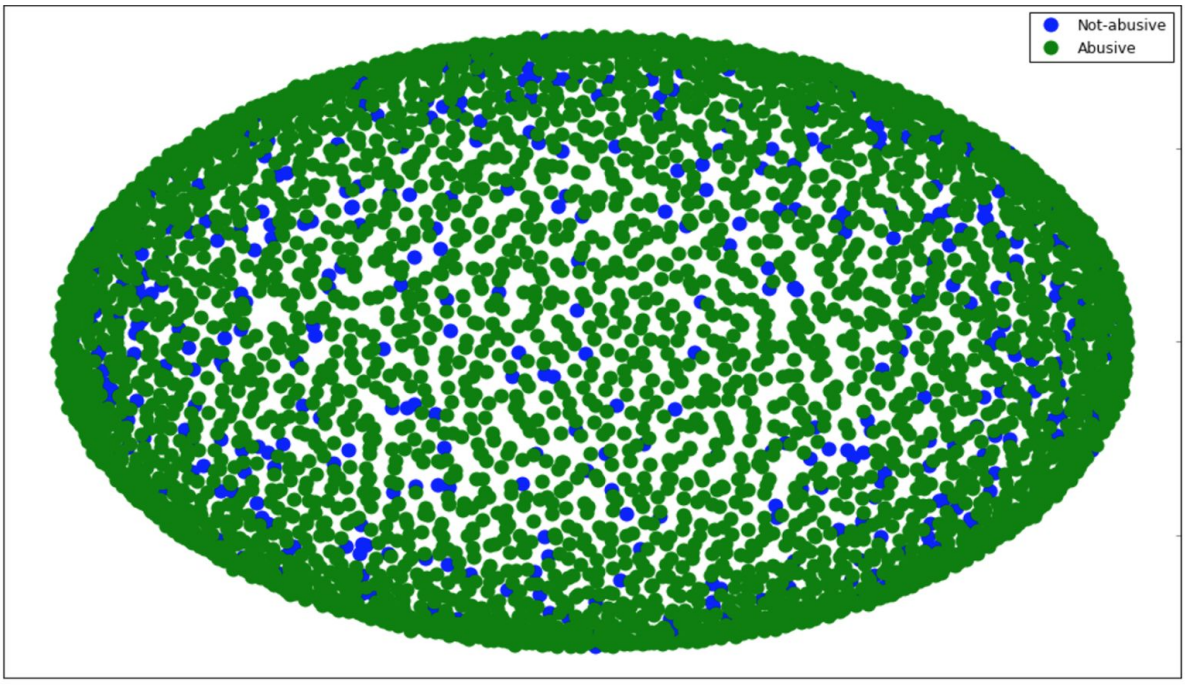
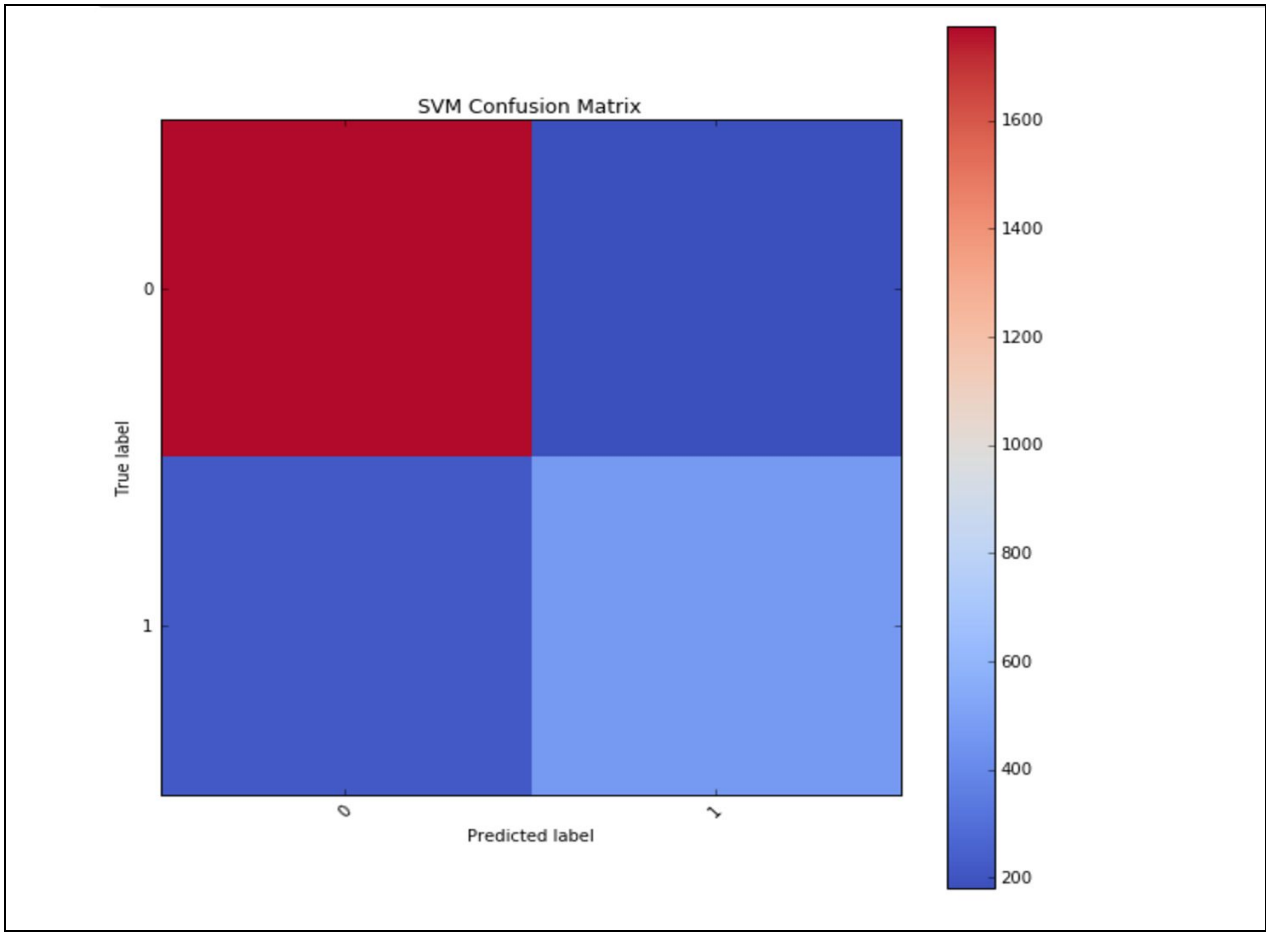


Figure: K-Means clustering (Green dots represent Abusive, Blue represent)

Evaluation Metrics:

After finalizing our training model, we tested the performance of the system using a completely isolated test set. Along with classification accuracy, we looked at the precision, recall and F-measures to evaluate the performance of our algorithm.

	precision	recall	f1-score	support
0	0.89	0.91	0.90	1954
1	0.72	0.68	0.70	693
avg / total	0.84	0.85	0.85	2647



Challenges:

- **What is Abuse**: Defining the task was the hardest problem, the notion of what constitutes abusive, is rather subjective and the degree of offensiveness varies considerably among people, rendering the labeling process in our task even harder. For example, even though our classifier works reasonably well, we found that some content are sometimes misconstrued as insulting by people when it might be interpreted as non-insulting by our classifier and some content may be construed as a joke whereas our classifier interpreted it as non-insulting. This is a highly subjective content and it is very difficult to get a perfect model attributing to this effect.
- **Sarcasm**. - Sarcasm is a difficult even for humans to get correct as it requires more than just syntactic understanding of the text. When speaking, the tone

usually gives away the sarcasm, which cannot be leveraged while processing text. Moreover, prior knowledge and the context play a huge role in the interpretation. Example- Do you still love nature, despite what it did to you?

You're proof that evolution can go in reverse. In order to understand this text, one must have background knowledge that evolution in general makes species more advanced and is associated with progress and reverse is the opposite of that.

- **Online Messages**: People from all around the world are a part of the online social media community. The way people use English and their writing styles on social media is highly diverse. Our training data set of roughly 4000 online messages does not capture the variations in the social media writing styles. We feel that we would have much better results if we have a larger data set.

CONTRIBUTIONS OF EACH TEAM MEMBER

Our work for this project can be divided into 3 different phases:

1. Phase 1: Each of us would read papers/blog posts related to abuse detection at home, and meet on a planned dates for ideation and chalking out a plan of how to proceed in the project and what all techniques to use.
2. Phase 2: Each of us then delivered the following subparts of the project

   Preprocessing - Nihar, Rohit

   Feature Extraction - Rohit, Suchi, Nia, Nihar

   Word2Vec - Suchi, Nia

   Custom feature extraction, TF-IDF - Rohit, Nihar

   Models: - Suchi, Nia

   Training - Suchi, Nia

   Hyperparameters - Suchi, Nia

   K-fold cross validation - Suchi, Nia

   K-means - Suchi, Nia

   User Interface design - Rohit, Nihar

   Documentation and Reports: Rohit, Suchi, Nia, Nihar
3. Phase 3: This phase was all about iterative development. We wanted to find the perfect combination of data preprocessing, feature collection and classification

algorithm that would give us the best results possible. We would sit together, and try out different combinations to get the highest accuracy. For example, we found that by converting all the text to lowercase, we were missing out on key information and so we preserved the original case of the sentence and captured the inherent sentiment of the message with greater accuracy. Similarly we also tuned our feature collection methods or added custom features to maximize on the accuracy of our dataset.

USER INTERFACE:

Following is our user interface to analyse texts and user handles for abuse detection:
http://cyberabusedetection.herokuapp.com/

CODE:

Following is the ipython notebook of our code:
http://nbviewer.jupyter.org/gist/niavivek/d3a56ace8291c93aa3b83fc0a467ff8e

REFERENCES

1. Detecting Offensive Language in Social Media to Protect Adolescent Online Safety: https://faculty.ist.psu.edu/xu/papers/Chen_etal_SocialCom_2012.pdf
2. Insult Detection in Social Network Comments Using Possibilistic Based Fusion Approach: http://www.springer.com/cda/content/document/cda_downloaddocument/9783319105086-c2.pdf?SGWID=0-0-45-1478815-p176900326
3. Twitter Bullying Detection: https://users.soe.ucsc.edu/~shreyask/ism245-rpt.pdf
4. Detecting flames and insults in text: https://www.cs.csustan.edu/~mmartin/LDS/Altaf_Flames.pdf
5. Detecting and Tracking Political Abuse in Social Media: http://www.aaai.org/ocs/index.php/ICWSM/ICWSM11/paper/viewFile/2850/3274
6. Abusive Language Detection in Online User Content (Yahoo) - http://www2016.net/proceedings/proceedings/p145.pdf
7. Offensive Language Detection Using Multi-level Classification - http://rd.springer.com/chapter/10.1007/978-3-642-13059-5_5
8. Detecting Offensive Tweets via Topical Feature Discovery over a Large Scale Twitter Corpus http://www.cs.cmu.edu/~lingwang/papers/sp250-xiang.pdf
9. Spell correction algorithm: http://norvig.com/spell-correct.html

10. Social media regular expression tokenizer: https://marcobonzanini.com/2015/03/09/mining-twitter-data-with-python-part-2/