MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
VASYL' STUS' DONETSK NATIONAL UNIVERSITY

**Elton Mzamo Dube**

*Allowed for defence:*
Head of Information Technologies Department
PhD, associate professor
_____ Tetyana Neskorodieva
«_____» June 2021

# DETECTION OF MASKED BAD WORDS IN SOCIAL NETWORK CONTENT

Specialty 122 Computer Science

**Bachelor's Thesis**

Supervisor:

Serhiy Shtovba, Professor on Information Technologies Department
DrSc, Full Professor

*Grade:*

Head of Examination Comission:
_____

Vinnytsia – 2021

# ABSTRACT

**Elton M. Dube** Detection of masked bad words in social network content. Bachelor's Thesis. Specialty 122 "Computer Science", educational program "Computer Science". Vasyl' Stus Donetsk National University, Vinnytsia, 2021.

In the qualification (bachelor's) work the problem of masked bad words detection was investigated and we looked at previous models that were used for the task of detecting bad words. A proposal was made for a technique that would be optimal to detecting masked bad word e.g., fvck, b1tch.

Keywords: toxic content detection, social network, confusion matrix, out of vocabulary words.

33 Pages, 2 Tables, _4 Figures, 20 Ref.

## Contents

# Introduction

With the increase in popularity and availability of different social media platforms, more and more people are finding it easier and easier to communicate with each other all over the world. That is just one of the highlights of social media. Now let us look at the downside of social media. While many people use it for simple communication with friends and family and keeping up with the latest trends and news, others on the other hand are using it for all the wrong reasons including bullying and circulating false information. All of this has led to the development and improvements of things such as the detection of abusive language, hate speech, cyberbullying, and trolling amongst others. Social Media Sites are being tasked to continuously improve their cybersecurity measures to protect their users from cyberbullying.

With the world ever evolving and humans becoming more and more sophisticated and intelligent, this has led to cyberbullies adapting to these restrictions put in place by social media sites and now masking bad words, profanity and hate speech, and this has made it hard for some models to detect some bad words and profanity.

With the help of machine learning we have come up with ways to detect masked bad words in social media content and in this report, we will look at some models that are being used and how we can further improve them in the future.

Some examples of masked bad words that are hard to detect automatically include the following: b1tch, 5hit, A55, D!ck, fvck, cr@p, D1ps#!t, pr1ck, idi0t, idl0t, Pus*y. All these words are out of vocabulary, but a human recognizes easily.

# Chapter 1

## A survey of techniques for toxic content detection in social networks

### 1.1 A study of toxic content on social media

Starting from the very basic question that one might ask, which is “What is a social network?”. In simple terms it can be described as a dedicated website or other application which enables users to communicate with each other by posting information, comments, messages, images, etc.

Social networking now becomes the use of Internet-based social media sites to stay connected with friends, family, colleagues, customers, or clients. Social networking can have a social purpose, a business purpose, or both, through sites like Facebook, Twitter, LinkedIn, and Instagram.

The reasons behind the huge popularity of social media these days is the fact that there are a lot of opportunities to meet new people, most of the social networking sites are user-friendly, one can find groups that share their interests, these sites are free to use, they help with job markets and help businesses to reach out to their clients and potential clients.

These are just the few of many benefits to social networks. There is also a bad side to these social networks. For the purpose of this research, we will look at specifically toxic content.

Toxic content comes in the form of profanity, use of abusive words or language in comments, threats, shameful speech etc.

Some examples of toxic content are as follows:

| | |
|---|---|
| TOXICITY | @user Fuck you, you fat piece of shit |
| INSULT | Hey @user , you are disgusting. |
| THREAT | @user Kill the traitors. |
| PROFANITY | My wrist been fucked up for nearly a month now . This time im really going to the hospital to see what the fuck is wrong with it |
| IDENTITY ATTACK | Okay everyone always talks aboht the pathetic army and all the soy boy branches and gay shit and what not [...] |
| ATTACK ON COMMENTER | @user You are all utterly delusional. If you were really pro-life" you would [...] |

Fig. 1.1 – Examples of toxic comments [3]

While many people use social networks for simple communication with friends and family and keeping up with the latest trends and news, others on the other hand are using it for all the wrong reasons including bullying and circulating false information. All of this has led to the development and improvements of things such as the detection of abusive language, hate speech, cyberbullying, and trolling amongst others. Social Media Sites are being tasked to continuously improve their cybersecurity measures to protect their users from cyberbullying.

With the world ever evolving and humans becoming more and more sophisticated and intelligent, this has led to cyberbullies adapting to these restrictions put in place by social media sites and now masking bad words, profanity and hate speech, and this has made it hard for some models to detect some bad words and profanity.

With the help of machine learning we have come up with ways to detect masked bad words in social media content and in this report, we will look at some models that are being used and how we can further improve them in the future.

Some examples of masked bad words that are hard to detect automatically include the following: b1tch, 5hit, A55, D!ck, fvck, cr@p, D1ps#!t, pr1ck, idi0t, idl0t, Pus*y. All these words are out of vocabulary, but a human recognizes easily.

## 1.2 Techniques for toxic content detections

### 1.2.1 Crowdsourcing

With more room for improvement, we have seen a few models/techniques being developed over time. In 2012 Sara Owsley Sood and other co-authors [1], developed a technique that used Crowdsourcing as its main model.

In this technique there was use of a large dataset of comments from a medium sized social news site. As with most social news sites, users of this site contribute links to news items (i.e., stories, videos and images), and other community members then comment on these items. The dataset contains all comments posted between March and May 2010. The set of 1,655,131 comments span 168,973 threads. Each thread represents a news item; that is, a story, image or video contributed by a community member.

Some methods that were taken into consideration in this technique include:

1) List-based Approaches in order to determine if a document (e.g., email, comment, tweet, blog) contains profanity, these systems simply examine each word in the document. If any of the words are present on a list of profane terms, then the document is labeled as profane.

2) Levenshtein Edit Distance, after first checking to see if a word exists on a profanity list, then a second pass is made to look for variations of the words. To identify these instances of profanity, utilization of a tool to calculate the Levenshtein edit distance between two terms is used [20]. This edit distance measures the number of letter insertions, deletions and changes to transform one word into another. To check whether a term might be profanity, a calculation of the edit distance is made from that term to each term on a list of known profane terms. If the edit distance is equal to the number of punctuation marks in the term, or if it is below some threshold (which varies by the term length), then it is flagged as profane.

3) Support Vector Machines, the above two systems utilize lists of known profane terms and tools that attempt to find variations in these terms. However, as profane language evolves over time, these lists must be updated to catch new cases. A more robust approach may be to look at the context in which the profane language occurs. As such, support vector machines were used, known for their performance in text classification [19], to learn a model of profanity. The 6500 profanity

labeled comments from the dataset described above were used. Using a bag of words approach, it was found that the optimal features were bigrams and stems using a binary presence representation and a linear kernel.

This model used useful features like Bigrams, but this model had a very low recall performance, and the system could not be optimized for a high recall performance.

### 1.2.2 Sentence embeddings

In 2019 we saw the development of a model that used Sentence embeddings from Vijayasaradhi Indurthi and other co-authors [2], and this model used word embeddings and sentence embedding as main features.

The SEMEVAL-2019 OFFENSEVAL dataset is used, that is available to participants and it contains 13240 tweets; The OFFENSEVAL task consists of three subtasks. Subtask A aims at the detection of offensive language (OFF or NOT). Subtask B aims at categorizing offensive language as targeting a specific entity (TIN) or not (UNT). Subtask C aims to identify whether the target of an offensive post is an individual (IND), a group (GRP), or unknown (OTH).

One drawback to this model is that the class distribution was highly imbalanced due to which there was a likelihood of a bias being introduced by the training algorithms.

### 1.2.3 Perspective and BERT

Another model that was proposed in 2019 was Perspective & Bert by John Pavlopoulos and other co-authors [3], and This model made use of character n-grams, word-length distribution, extra-linguistic features and geographic features.

The SEMEVAL-2019 OFFENSEVAL dataset is used, that is available to participants and it contains 13240 tweets; The OFFENSEVAL task consists of three subtasks. Subtask A aims at the detection of offensive language (OFF or NOT). Subtask B aims at categorizing offensive language as targeting a specific entity (TIN) or not (UNT). Subtask C aims to identify whether the target of an offensive post is an individual (IND), a group (GRP), or unknown (OTH).

Although good it had a few drawbacks, and one was that the geographic and word length distribution have little to no positive effect on performance and rarely improve over character-level features.

### 1.2.4 Deep machine learning

A Deep machine learning model was proposed by D. Thenmozhi and other co-authors [4]. This model made use of Word embeddings, Multinominal Naïve Bayes, SVM, Stochastic Gradient Descent, Bag of words, Bi-gram features, Skip-grams, clustering-based word representations.

The SEMEVAL-2019 OFFENSEVAL dataset is used here also.

In deep learning (DL) approach, the tweets are vectorized using word embeddings and are fed into encoding and decoding processes. Bidirectional LSTMs are used for encoding and decoding processes. A 2-layer LSTM is used for this. The output is given to softmax layer by incorporating attention wrapper to obtain the OffensEval class labels.

The drawback observed from the results of this model was that the deep learning model could not learn the features appropriately due to less domain knowledge imparted by the smaller dataset used.

### 1.2.5 Traditional Machine Learning

In 2020 a Traditional Machine Learning Model was proposed by Varsha Pathak and co-authors [5]. This model saw the use of features like Word n-gram, character n-gram, combined word, custom word embedding.

In this work supervised machine learning is used by experimenting on various Classifiers. The datasets were preprocessed for removal of noisy elements from its contents. Appropriate features are extracted to enable the machine to learn offensive term patterns. Finally, the performances of different Classifiers and feature models are compared using standard measures to choose the best performing model.

Drawback to this model that we observed was that it cannot learn offensive terms from the text contents or from speech irrespective of the language.

## 1.3 Problems with masked bad words detection

Social Media Sites are being tasked to continuously improve their cybersecurity measures to protect their users from cyberbullying.

This is in line with the fact that humans are intelligent beings and will always find a way around a problem. In the past we have seen many models being used to identify profanity, hate speech, toxicity in comments etc. but there is a slight flaw in these systems.

Humans have adapted and realized that by masking their bad words these systems are unable to detect the bad word but to the human eye it is clearly visible that it is a bad word. So, our task now is to find a way to efficiently detect masked bad words and to continuously learn the ways in which bad words are being masked so that our system can easily identify masked bad words in future with great effectiveness.

One problem that we came across comes from "Using crowdsourcing to improve profanity detection" by Sara Owsley Sood [2], In this article it was identified that using a list-based approach did not suffice. This is because in order to determine if a document (e.g., email, comment, tweet, blog) contains profanity, these systems simply examine each word in the document. If any of the words are present on a list of profane terms, then the document is labeled as profane.

This approach does not take into consideration the context in which a certain word was used in the comment and since it is only looking at a known list of profane terms and bad words it cannot identify a bad word if it is masked or intentionally misspelled.

In order to tackle these problems, we see that after first checking to see if a word exists on a profanity list, then a second pass is made to look for variations of the words. To identify these instances of profanity, utilization of a tool to calculate the Levenshtein edit distance between two terms is used (Levenshtein 1966).

To check whether a term might be profanity, a calculation of the edit distance is made from that term to each term on a list of known profane terms. If the edit distance is equal to the number of punctuation marks in the term, or if it is below some threshold then it is flagged as profane.

This approach then improves the detection of intentionally misspelled bad words found in the comments.

Another problem that was encountered in the survey is that from the article on Sentence embeddings from Vijayasaradhi Indurthi and other co-authors [3]. In this article the authors used sentence embedding and word embedding as main features in their task.

The task here was split into 3 subtasks that had to classify profanity according to whether it was offensive or not and so on. The problem that was observed was in the

training data. We observed that the class distribution was highly imbalanced due to which there was a likelihood of a bias being introduced by the training algorithms.

Although the problem was not tackled immediately during this task, it was included in future work where the authors plan to explore SMOTE [6] for further making the class labels more balanced and then train the classification which will prevent a bias towards the unbalanced classes in the training data.

In the model that was proposed in 2019 Perspective & Bert by John Pavlopoulos and other co-authors [3] we saw the use of character n-grams, word-length distribution, extra-linguistic features and geographic features.

Some of these features were quite useful and some of them not so much. One drawback that was noted was that that the geographic and word length distribution have little to no positive effect on performance and rarely improve over character-level features.

In the Deep machine learning model that was proposed by D. Thenmozhi and other co-authors [4] we saw that his model made use of Word embeddings, Multinominal Naïve Bayes, SVM, Stochastic Gradient Descent, Bag of words, Bi-gram features, Skip-grams, clustering-based word representations.

One drawback that stood out was that from the results of this model the deep learning model could not learn the features appropriately due to less domain knowledge imparted by the smaller dataset used.

In order to tackle this problem, we will need to introduce a larger dataset that will be sufficient to train our model and in turn give us more reliable results at the end of the day.

In the Traditional Machine Learning Model which was proposed by Varsha Pathak and co-authors [5] this model saw the use of features like Word n-gram, character n-gram, combined word, custom word embedding.

One notable drawback to this model was that that it cannot learn offensive terms from the text contents or from speech irrespective of the language.

This is a problem because some terms may not be offensive depending on their context in which they are used, an example being the word “bitch” (female dog), if you consider the context of the whole sentence then you will determine whether the term is offensive or not.

In order to tackle this, we will need to consider text contents and the specific language to have a clear understanding.

If we are now taking into consideration all the proposed models and techniques used in the past, we will see the need to have a new approach to bad words detection. The reason why we need a new approach is because with the world ever changing and evolving, we also need to adapt our technology to keep up with that change.

Humans are not only intentionally misspelling bad words, but they are now using some other special characters that are not alphabetical e.g., @, $, #, etc. but are just visually like alphabetic characters, so in order to detect bad words that are masked in this way we need to train our models to be able to identify all these adaptations.

## 1.4 Ideas for proposed technique for masked bad words detection

Our approach to detecting the masked bad words is as follows. The first task at hand will be to source out a vocabulary of known bad words in English that we can use for comparisons later. For this we found the dataset "Bad Bad Words" on Kaggle, and the purpose of this dataset is to support the Toxic Comment Classification Competition. It has a wide range of words, i.e., close to 2000 words.

Now in order to analyze the data we will use embedding algorithms like Word2Vec because it is a statistical method for efficiently learning a standalone word embedding from a text corpus as there is no need to analyze a full comment but rather just a single isolated word. After this we will have to do some comparisons and for this, we will use the Levenshtein distance. The Levenshtein distance is one of the methods to calculate the similarity between two strings. It is calculated by the operation how many times the character is inserted, deleted or replaced when converting one string to the other.

We also need to have a confusion matrix which is a very crucial part of the whole technique because the purpose of this confusion matrix is to detect hidden words that social media users have masked using various symbols e.g., *b1tch*. The matrix will give us a probability between 0 and 1 and we need to incorporate that probability into the Levenshtein distance. Fig. 2 shows an example of this matrix.

| Lowercase Letter–Lowercase Letter | Uppercase Letter–Uppercase Letter | Uppercase Letter–Numeral (cont'd) |
|---|---|---|
| g and q | T and I | O and 0 |
| p and n | D and O | B and 8 |
| m and n | C and G | D and 0 |
| y and z | L and I | S and 5 |
| u and v | M and N | S and 8 |
| c and e | P and B | Y and 5 |
| cursive l and cursive b | F and R | Z and 7 |
| cursive i and cursive e | U and O | T and 7 |
| cursive a and cursive o | U and V | U and 0 |
| **Lowercase Letter–Numeral** | E and F | U and 4 |
| l and 1 | V and W | **Numeral–Numeral** |
| b and 6 | X and Y | 0 and 8 |
| o and O | cursive S and cursive L | 3 and 9 |
| g and 9 | **Uppercase Letter–Numeral** | 3 and 8 |
| q and 9 | G and 6 | 4 and 9 |
| **Uppercase Letter–Lowercase Letter** | F and 7 | 5 and 8 |
| I and l | Z and 2 | 5 and 3 |
| | Q and 2 | 6 and 8 |
| | | 7 and 1 |

Fig. 1.2 – The most confused symbols [7]

We will also be looking at Support Vector Machines (SVMs) so that we can make sure that our model is able to learn bad words with time.

A more robust approach will be to look at the context in which the profane language occurs. As such, support vector machines are to be considered because of their known performance in text classification (Joachims 1998), in order to learn a model of profanity

## 1.5 Conclusions on the first Chapter

The importance of our task of bad words detection is quite simple and straightforward. As we have discussed, humans are ever evolving and adapting, and this poses a problem for systems that cannot adapt together with human adaptations. Over the years we have seen the task of bad words detection being tackled in many ways, some successful and some not as much. The importance of bad words detection comes to tackle cyberbullying and online hate speech among other things. Social media users have seen that social media platforms can easily identify know bad words and remove comments that involve these bad words or any other form of hate speech. This has led to social media users now masking their bad words by either intentionally misspelling bad words or using some other special characters that are visually like letters of the English alphabet to mask their bad words. This leads us to the task at hand, which is improve on the detection of bad words by adapting our systems to be able to recognize masked bad words.

The current approach to bad words detection is efficient in detecting known bad words from known bad words vocabularies but it fails to detect when there is a masked bad word. This is because masked bad words are easily identifiable to the human reader but are not easy to recognize by a computer.

This then leads us to our approach of masked bad words detection in which we will work with known bad words vocabularies in order to train our algorithm. In order to detect masked bad words, we will implement a confusion matrix of the English alphabet and some other characters that are visually similar to the letters of the English alphabet. Probabilities obtained from this confusion matrix will help us when incorporated with the Lavenshtein Distance to calculate the visual similarity of words found in our known bad words' vocabularies and our masked bad word.

Once we have successfully detected our masked bad words we will then use these new words to further train our algorithm so that it can easily adapt with time also therefore making the task of masked bad words detection easier for our algorithm.

# Chapter 2

# Information resources, models and algorithms for masked bad words detection

## 2.1 Bad words vocabularies

### 2.1.1 Bad Bad Words dataset.

For this task we found the dataset “Bad Bad Words” [14] on Kaggle, and the purpose of this dataset is to support the Toxic Comment Classification Competition. It has a wide range of words, i.e., close to 2000 words.

In this dataset we have a wide range of words but take note that not all of them are bad words, but they can be classified as bad words when taking into consideration the context in which they are used.

This list of bad words does not contain any intentionally misspelled bad words or any masked bad words but only contains correctly spelled words without any special characters.

### 2.1.2 YouTube dataset

In dataset ICWSM‑18‑SALMINEN [11], there are 3221 manually labeled comments posted on a YouTube channel of an online news and media company. Salminen et al. [12] note that many of the comments that are posted as reactions to the content in this channel are hateful, which makes the dataset promising for investigating online hate.

The researchers used manual coding to annotate the data into hateful and non-hateful comments (as well as subsequent themes based on the target of the hate; however, this information is not used for our classifier). They provide detailed coding guidelines as well as inter-rater agreement measurement (agreement score = 75.3%), which the researchers interpret as substantial agreement. The agreement score was calculated by dividing the number of labels where two or more coders agreed by the number of possible values. The calculation was done for each coded item, and the item-based agreements were averaged to output the overall agreement.

Overall, the dataset includes purposeful (i.e., intentionally hurtful) comments. This consideration was made because if hostility is not the purpose of the

comment, it should not be classified as hateful. For example, “Trump is a bad president” was not considered as hateful, but “Trump is an orange buffoon” was considered as hateful. Also, the annotators considered linguistic patterns when annotating, such that swearing, aggressive comments, or mentioning past political or ethnic conflicts in a nonconstructive and harmful way, were classified as hateful. When there was uncertainty about an instance, the researchers discussed it to avoid a biased label.

### 2.1.3 Twitter dataset

In the dataset DAVIDSON‑17‑ICWSM [11], this dataset is made available by Davidson et al. [13] who used crowd raters for labeling and provide a detailed description of the data collection principles. The dataset contains 25K tweets, randomly sampled from 85.4 M tweets extracted from the timeline of 33,458 Twitter users, using hate speech lexicon. The lexicon, compiled from Hatebase, contains words and phrases identified by internet users as hate speech.

This lexicon was also used by the authors as keywords to extract the 85.4 M tweets. The selected 25K tweets were manually annotated by at least 3 workers using CrowdFlower. The task was to annotate the tweet with one of three categories: hate speech, offensive but not hate speech, or neither offensive nor hate speech. An agreement of 92% was obtained between the workers regarding the class labels for the task, and the final gold label for each tweet was assigned using a majority voting approach. The tweets with no majority class were discarded, making a total of 24,802 tweets with an assigned label of which 5% was given the hateful label.

Note that the authors shared these tweets as “Tweet IDs”, i.e., references to the original tweets. Therefore, we had to utilize the Twitter API to recollect the dataset. We were able to obtain 24,783 tweets (99.9% of the original dataset), with 19 tweets either deleted or otherwise unavailable. The less of only a small number of comments is unlikely to have a significant impact on the results when comparing our performance against that of Davidson et al.

## 2.2 Confusion Matrix

The confusion matrix for the full English alphabet as seen from the article by L.R. GEYER [8] is a 26 by 26 array of stimulus-response probabilities. Each row represents an individual stimulus letter, the main-diagonal cell is the correct response probability for that letter, and incorrect responses generate off diagonal confusion probabilities. Each row must sum to 1.0; column sums may vary from unity, and the variation is sometimes taken as an estimate of response bias effects. With the help of this matrix we can better detect masked bad words that have special characters e.g., !,@,#,$ etc., in which these are visually similar to letters of the English alphabet.

Fig 2.1 below shows this matrix:

| | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | .54 | .01 | .01 | .06 | .05 | .00 | .03 | .01 | .00 | .00 | .00 | .00 | .02 | .05 | .14 | .01 | .03 | .00 | .00 | .01 | .03 | .00 | .00 | .00 | .01 | .00 |
| b | .03 | .53 | .01 | .02 | .01 | .00 | .06 | .21 | .01 | .00 | .07 | .00 | .00 | .02 | .01 | .00 | .01 | .00 | .00 | .00 | .02 | .01 | .00 | .01 | .00 | .00 |
| c | .01 | .03 | .23 | .02 | .02 | .03 | .02 | .01 | .09 | .02 | .09 | .00 | .01 | .03 | .01 | .00 | .00 | .18 | .04 | .02 | .02 | .02 | .01 | .10 | .01 | .01 |
| d | .01 | .03 | .01 | .80 | .01 | .00 | .01 | .00 | .01 | .06 | .00 | .01 | .01 | .01 | .01 | .01 | .00 | .00 | .00 | .00 | .01 | .01 | .00 | .01 | .00 | .01 |
| e | .18 | .03 | .00 | .05 | .21 | .01 | .05 | .02 | .01 | .02 | .00 | .00 | .01 | .06 | .27 | .02 | .03 | .00 | .01 | .00 | .01 | .00 | .02 | .00 | .00 | .00 |
| f | .00 | .00 | .01 | .03 | .00 | .27 | .01 | .01 | .06 | .15 | .01 | .26 | .00 | .00 | .00 | .01 | .01 | .11 | .01 | .02 | .02 | .00 | .00 | .01 | .01 | .00 |
| g | .03 | .00 | .00 | .01 | .02 | .00 | .75 | .01 | .00 | .01 | .00 | .00 | .00 | .03 | .02 | .01 | .07 | .00 | .01 | .00 | .02 | .00 | .01 | .00 | .00 | .01 |
| h | .01 | .05 | .00 | .01 | .00 | .00 | .02 | .69 | .01 | .00 | .09 | .01 | .01 | .05 | .01 | .01 | .00 | .00 | .00 | .02 | .01 | .00 | .02 | .00 | .00 | .00 |
| i | .01 | .01 | .03 | .02 | .00 | .03 | .00 | .01 | .34 | .09 | .01 | .18 | .01 | .00 | .01 | .01 | .01 | .01 | .00 | .07 | .01 | .00 | .00 | .03 | .01 | .00 |
| j | .00 | .00 | .00 | .01 | .00 | .03 | .01 | .00 | .03 | .80 | .00 | .10 | .00 | .00 | .00 | .00 | .00 | .01 | .01 | .00 | .01 | .01 | .00 | .00 | .00 | .00 |
| k | .01 | .01 | .00 | .00 | .01 | .02 | .01 | .07 | .01 | .00 | .71 | .03 | .00 | .02 | .01 | .00 | .00 | .01 | .01 | .02 | .01 | .00 | .01 | .03 | .01 | .01 |
| l | .00 | .00 | .00 | .01 | .00 | .07 | .00 | .00 | .12 | .09 | .02 | .50 | .01 | .00 | .00 | .01 | .01 | .06 | .01 | .09 | .00 | .01 | .00 | .01 | .01 | .00 |
| m | .03 | .00 | .01 | .00 | .00 | .00 | .01 | .01 | .01 | .02 | .01 | .01 | .73 | .06 | .00 | .01 | .01 | .02 | .01 | .00 | .01 | .01 | .02 | .02 | .01 | .00 |
| n | .05 | .01 | .01 | .02 | .01 | .01 | .02 | .03 | .02 | .01 | .00 | .01 | .02 | .64 | .03 | .01 | .02 | .03 | .02 | .00 | .01 | .01 | .01 | .01 | .00 | .00 |
| o | .14 | .02 | .03 | .01 | .10 | .01 | .02 | .01 | .01 | .00 | .00 | .00 | .01 | .07 | .42 | .03 | .02 | .02 | .02 | .01 | .05 | .01 | .01 | .01 | .01 | .00 |
| p | .03 | .03 | .01 | .02 | .03 | .00 | .03 | .02 | .00 | .00 | .02 | .00 | .02 | .14 | .06 | .50 | .01 | .01 | .01 | .01 | .02 | .00 | .02 | .01 | .01 | .00 |
| q | .15 | .00 | .00 | .02 | .01 | .02 | .10 | .01 | .01 | .01 | .01 | .01 | .03 | .04 | .02 | .05 | .44 | .01 | .01 | .01 | .02 | .00 | .01 | .00 | .00 | .00 |
| r | .02 | .01 | .02 | .01 | .01 | .03 | .02 | .01 | .07 | .02 | .03 | .01 | .00 | .02 | .01 | .02 | .00 | .37 | .04 | .05 | .01 | .03 | .03 | .09 | .04 | .03 |
| s | .06 | .03 | .02 | .00 | .07 | .00 | .02 | .04 | .02 | .01 | .06 | .01 | .01 | .13 | .07 | .01 | .02 | .05 | .14 | .01 | .01 | .06 | .05 | .07 | .03 | .02 |
| t | .00 | .00 | .03 | .02 | .00 | .04 | .01 | .01 | .14 | .02 | .09 | .15 | .00 | .01 | .00 | .00 | .01 | .10 | .03 | .24 | .02 | .02 | .00 | .02 | .03 | .02 |
| u | .08 | .04 | .01 | .03 | .02 | .00 | .02 | .01 | .01 | .01 | .02 | .01 | .02 | .04 | .05 | .00 | .00 | .01 | .01 | .00 | .50 | .07 | .03 | .01 | .01 | .01 |
| v | .00 | .03 | .00 | .00 | .01 | .00 | .00 | .01 | .02 | .00 | .03 | .01 | .00 | .03 | .03 | .00 | .00 | .03 | .02 | .01 | .04 | .51 | .15 | .02 | .05 | .01 |
| w | .01 | .00 | .01 | .01 | .01 | .01 | .01 | .00 | .00 | .00 | .01 | .00 | .03 | .03 | .00 | .00 | .00 | .01 | .00 | .00 | .01 | .06 | .73 | .02 | .03 | .01 |
| x | .01 | .01 | .02 | .01 | .01 | .00 | .01 | .01 | .04 | .01 | .08 | .01 | .01 | .04 | .01 | .01 | .00 | .05 | .03 | .01 | .01 | .04 | .02 | .53 | .02 | .03 |
| y | .01 | .00 | .00 | .00 | .00 | .02 | .01 | .00 | .01 | .00 | .02 | .00 | .00 | .01 | .01 | .01 | .00 | .01 | .00 | .01 | .02 | .13 | .02 | .02 | .67 | .02 |
| z | .01 | .00 | .04 | .02 | .01 | .02 | .03 | .00 | .03 | .02 | .03 | .00 | .01 | .01 | .02 | .01 | .00 | .10 | .05 | .01 | .03 | .07 | .04 | .16 | .10 | .19 |

Fig 2.1 – Confusion matrix for lowercase letters of the English alphabet [8]

Further studies Showed that J. T. TOWNSEND [9] did a similar confusion matrix using uppercase letters of the English alphabet and used a few different approaches that gave different probabilities. The best one is shown in Fig 2.2 below:

| Stimulus | Response | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| A | .83 | .01 | .00 | .01 | .00 | .01 | .00 | .01 | .01 | .01 | .01 | .02 | .00 | .00 | .02 | .00 | .00 | .00 | .02 | .01 | .00 | .03 | .00 | .00 | .00 | .01 |
| B | .02 | .36 | .01 | .05 | .01 | .04 | .03 | .07 | .02 | .03 | .01 | .03 | .01 | .02 | .10 | .05 | .00 | .01 | .02 | .04 | .01 | .03 | .01 | .01 | .01 | .01 |
| C | .00 | .00 | .79 | .01 | .00 | .01 | .03 | .02 | .01 | .01 | .00 | .01 | .01 | .00 | .05 | .01 | .00 | .01 | .02 | .00 | .00 | .01 | .00 | .01 | .01 | .00 |
| D | .01 | .01 | .01 | .73 | .01 | .01 | .00 | .03 | .00 | .01 | .00 | .00 | .00 | .01 | .11 | .02 | .03 | .01 | .01 | .00 | .01 | .01 | .00 | .01 | .00 | .01 |
| E | .03 | .01 | .03 | .00 | .43 | .09 | .01 | .01 | .05 | .01 | .03 | .09 | .01 | .00 | .06 | .01 | .00 | .01 | .03 | .03 | .00 | .01 | .00 | .02 | .03 | .01 |
| F | .05 | .01 | .02 | .03 | .05 | .42 | .01 | .05 | .04 | .01 | .01 | .05 | .01 | .00 | .05 | .03 | .01 | .00 | .01 | .07 | .01 | .03 | .00 | .01 | .04 | .01 |
| G | .00 | .00 | .11 | .01 | .01 | .01 | .57 | .04 | .00 | .01 | .01 | .00 | .00 | .01 | .12 | .02 | .01 | .01 | .00 | .00 | .03 | .03 | .00 | .00 | .00 | .01 |
| H | .05 | .01 | .00 | .03 | .03 | .05 | .03 | .19 | .09 | .01 | .01 | .05 | .03 | .03 | .10 | .03 | .01 | .02 | .02 | .05 | .05 | .07 | .00 | .01 | .01 | .03 |
| I | .05 | .01 | .01 | .02 | .01 | .03 | .01 | .07 | .38 | .03 | .00 | .12 | .00 | .01 | .07 | .01 | .00 | .01 | .01 | .05 | .01 | .05 | .00 | .00 | .02 | .01 |
| J | .03 | .00 | .00 | .02 | .01 | .03 | .01 | .03 | .08 | .49 | .01 | .03 | .00 | .00 | .06 | .01 | .01 | .00 | .02 | .05 | .01 | .06 | .01 | .00 | .01 | .02 |
| K | .03 | .00 | .00 | .00 | .00 | .03 | .00 | .05 | .01 | .01 | .61 | .05 | .01 | .00 | .04 | .01 | .00 | .01 | .00 | .02 | .01 | .01 | .00 | .06 | .03 | .02 |
| L | .03 | .01 | .03 | .02 | .02 | .02 | .01 | .03 | .04 | .00 | .02 | .59 | .02 | .00 | .07 | .01 | .00 | .00 | .01 | .03 | .01 | .00 | .01 | .00 | .02 | .00 |
| M | .09 | .01 | .01 | .01 | .01 | .01 | .04 | .06 | .04 | .03 | .01 | .03 | .41 | .05 | .08 | .02 | .00 | .01 | .00 | .00 | .00 | .05 | .01 | .01 | .01 | .03 |
| N | .04 | .00 | .01 | .02 | .01 | .02 | .01 | .05 | .05 | .01 | .01 | .05 | .01 | .39 | .09 | .01 | .01 | .01 | .01 | .07 | .01 | .05 | .02 | .00 | .03 | .00 |
| O | .01 | .00 | .01 | .02 | .00 | .01 | .07 | .01 | .01 | .00 | .00 | .01 | .00 | .00 | .72 | .02 | .06 | .00 | .02 | .01 | .00 | .01 | .00 | .01 | .00 | .01 |
| P | .00 | .02 | .01 | .02 | .01 | .01 | .00 | .02 | .03 | .01 | .01 | .02 | .00 | .00 | .05 | .67 | .01 | .02 | .03 | .01 | .01 | .01 | .00 | .01 | .02 | .01 |
| Q | .02 | .00 | .03 | .00 | .00 | .02 | .02 | .01 | .01 | .00 | .00 | .00 | .00 | .00 | .32 | .01 | .52 | .01 | .03 | .01 | .00 | .00 | .00 | .00 | .00 | .01 |
| R | .02 | .01 | .01 | .03 | .00 | .01 | .00 | .08 | .01 | .00 | .01 | .03 | .01 | .00 | .03 | .13 | .01 | .52 | .01 | .03 | .01 | .02 | .00 | .00 | .03 | .01 |
| S | .03 | .01 | .01 | .01 | .00 | .00 | .03 | .07 | .01 | .01 | .00 | .01 | .00 | .00 | .01 | .00 | .00 | .01 | .78 | .01 | .00 | .01 | .00 | .00 | .00 | .01 |
| T | .01 | .00 | .01 | .01 | .01 | .03 | .01 | .01 | .08 | .03 | .01 | .05 | .00 | .01 | .03 | .01 | .00 | .01 | .01 | .58 | .01 | .03 | .00 | .00 | .03 | .02 |
| U | .03 | .02 | .01 | .01 | .01 | .00 | .01 | .03 | .05 | .02 | .02 | .01 | .01 | .00 | .01 | .03 | .00 | .00 | .04 | .01 | .53 | .05 | .00 | .02 | .01 | .01 |
| V | .02 | .01 | .01 | .01 | .01 | .01 | .00 | .01 | .02 | .00 | .00 | .01 | .01 | .01 | .05 | .01 | .00 | .00 | .01 | .01 | .00 | .69 | .01 | .02 | .05 | .01 |
| W | .05 | .01 | .01 | .01 | .00 | .03 | .01 | .05 | .01 | .02 | .00 | .02 | .02 | .03 | .09 | .04 | .00 | .01 | .03 | .02 | .03 | .37 | .09 | .03 | .03 | .01 |
| X | .03 | .01 | .00 | .00 | .00 | .01 | .01 | .06 | .03 | .00 | .06 | .04 | .00 | .00 | .03 | .01 | .00 | .00 | .01 | .03 | .00 | .07 | .01 | .47 | .09 | .03 |
| Y | .01 | .00 | .01 | .01 | .01 | .01 | .00 | .04 | .03 | .00 | .02 | .04 | .01 | .01 | .07 | .01 | .00 | .02 | .03 | .04 | .00 | .22 | .00 | .01 | .36 | .03 |
| Z | .01 | .00 | .00 | .00 | .00 | .01 | .00 | .00 | .01 | .01 | .00 | .01 | .01 | .00 | .01 | .01 | .00 | .00 | .01 | .01 | .00 | .01 | .00 | .00 | .03 | .88 |

Fig 2.2 – Confusion matrix for uppercase letters of the English alphabet [9]

## 2.3 Metrics for words similarity

Edit distance is an important class of string metrics, which determines the distance between two strings S and T by calculating the cost of best sequence of edit operations that convert S to T. Typical edit operations are character insertion, deletion, and substitution, and transposition. There are several variants to calculate the edit distance depending on which edit operations are allowed [15].

### 2.3.1 Levenshtein Distance

Research carried out by AOURAGH SI LHOUSSAIN [10] tells us that The Metric method introduced by Levenshtein [20] measures the similarity between two words by calculating an edit distance. The edit distance is defined as the minimum number of basic editing operations needed to transform a wrong word to a dictionary word. Thus, to correct a wrong word, one retains a set of solutions requiring fewer possible editing operations.[10]

The procedure for calculating the Levenshtein distance between two strings $X = x_1x_2 ...x_m$ of length m and $Y= y_1y_2...y_m$ of length n, is to calculate step by step in a matrix of order m × n edit distance between different sub-strings of $X$ and $Y$. The corresponding values are stored in the matrix up to the box $(m,n)$ which expresses the minimum distance between $X$ and $Y$.[10]

The calculation of the cell $(i , j)$, which corresponds to the edit distance between sub-strings of $X_i^1 = x_1x_2...x_i$ and $Y_j^1 = y_1y_2...y_j$ , is given by the following recursive relationship:

$$D(i,j) = Minimum\{D(i-1,j)+1; D(i,j-1)+1; D(i-1,j-1)+\cos t\},$$

Where $\cos t = \begin{cases} 0 & if\ x_{1-i} = y_{j-1} \\ 1 & otherwise \end{cases}$ .

Thus, the following boots: $D(i,\emptyset) = i\ and\ D(\emptyset,j) = j$ , where $\emptyset$ is the empty string. [10]

The following steps show how to construct the matrix in Fig 5:

1. Construct a matrix d containing M rows and N columns.

2. Initialize the first row from 0 to M and first column from 0 to N.

3. Examine each character of S (i from 1 to M) and each character of T (j from 1 to N).

4. If S[i] equals T[j], the cost is 0. Otherwise, the cost is 1.

5. Set cell d [i, j] of the matrix equal to the minimum of:

   A. The cell immediately above plus 1: d [i-1, j] + 1.

   B. The cell immediately to the left plus 1: d [i, j-1] + 1.

   C. The cell diagonally above and to the left plus the cost: d [i-1, j-1] + cost.

6. After the iteration steps (3, 4 and 5) are complete, the distance is found in cell d [N,M].

| | | A | P | P | L | E |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| O | 1 | 1 | 2 | 3 | 4 | 5 |
| R | 2 | 2 | 2 | 3 | 4 | 5 |
| A | 3 | 2 | 3 | 3 | 4 | 5 |
| N | 4 | 3 | 3 | 4 | 4 | 5 |
| G | 5 | 4 | 4 | 4 | 5 | 5 |
| E | 6 | 5 | 5 | 5 | 5 | 5 |

Fig 2.3 - Example of Levenshtein distance [15]

### 2.3.2 Hamming distance

*Hamming distance* [16] allows only edit operation of substitution to transform *S* into *T*. Therefore, the length between *S* and *T* is the same. This algorithm is often used for error detection and correction for two strings with the same lengths [15].

### 2.3.3 Damerau–Levenshtein distance

Damerau–Levenshtein distance [17] is quite like Levenshtein distance. The only difference is that Damerau–Levenshtein distance allows one more edit operation: the transposition of two adjacent characters [15].

### 2.3.4 Q-gram

Q-gram is a consecutive substring of size that can be used as a signature of the entire string. Q-gram is typically used in approximate string matching by sliding a window of length q over the characters of a string to create several substrings. Since Q-gram can have fewer than q characters, characters "#" and "%" are used to extend the string by prefixing it with q-1 occurrences of "#" and suffixing it with q-1 occurrences of "%". When q equals to 1, the Q-gram is the same as edit distance. The foundation of the use of Q-gram is that when S and T are within a small edit distance of each other, they share many Q-gram in common. Getting the Q-gram for two query strings makes the count of identical Q-gram of these two strings and the total Q-gram available. The algorithm contains the following steps [15].

1. Extend the string by prefixing it with q-1 occurrences of "#" and suffixing it with q-1 occurrences of "%".
2. Split the S and T into two sets of Q-gram arrayS, arrayT.
3. Get the total grams number L by adding number of grams in S and T.
4. Combine two sets of Q-gram arrayS and arrayT to a set of Q-gram arrayTotal.
5. Remove the duplicate Q-gram in arrayTotal.
6. Calculate the number m1 of same Q-gram shared between arrayS and arrayTotal.

7. Calculate the number m2 of same Q-gram shared between arrayT and arrayTotal.

8. Calculate the absolute value difference by |m1-m2|.

9. The similarity of Q-gram is calculated as below:

$$similarity = \frac{L-difference}{L} .$$

2.3.5 Cosine similarity

Cosine similarity is a vector-based similarity measure. Cosine of two vectors a, b can be derived by using the Euclidean dot product formula. [15]

$$a.b = |a||b| \cos\theta$$

Where, $\theta$ represents the angle between $a$ and $b$.

The input string is transformed into vector space in order to apply the Euclidean cosine rule to determine similarity. The algorithm contains the following steps.

1. Split the S and T into two sets of 2-gram arrayS, arrayT.

2. Remove the duplicate 2-gram in arrayS, arrayT and get the number $L_1$, $L_2$ of

2-gram in arrayS and arrayT.

3. Combine two sets of 2-gram arrayS and arrayT to a new of 2-gram arrayTotal, remove the duplicate 2-gram in arrayTotal, and get the number $L$ of 2-gram in arrayTotal.

4. The variable C is calculated as follows.

$$C = (L_1 + L_2) - L$$

5. The cosine similarity is calculated as follows.

$$CosineSimilarity = \frac{C}{\sqrt{L_1} \cdot \sqrt{L_2}}$$

2.3.6 Dice coefficient

Dice coefficient [18], named after Lee Raymond Dice and known as the Dice's coefficient, is a term-based similarity measure. It is calculated as follows: [15]

$$DiceCoefficient = \frac{2 \times C}{L_1 + L_2}$$

Where *C* is the number of character bigrams found in both strings *S* and *T*, $L_1$ is the number of unique bigrams in string *S* and $L_2$ is the number of unique bigrams in string *T*. The algorithm contains the following steps:

1. Split the S and T into two sets of 2-gram arrayS, arrayT.

2. Remove the duplicate 2-gram in arrayS, arrayT and get the number L1, L2 of 2-gram in arrayS and arrayT.

3. Combine two sets of 2-gram arrayS and arrayT to a new set of 2-gram arrayTotal, remove the duplicate 2-gram in arrayTotal, and get the number L of 2-gram in arrayTotal.

4. The variable C is calculated as follows:

$$C = (L_1 + L_2) - L$$

5. The dice coefficient is calculated as follows:

$$DiceCoefficient = \frac{2 \times C}{L_1 + L_2}$$

## 2.4 Algorithms for masked bad words detection

Firstly, we need look at the confusion matrix and find out very similar symbols, with high probabilities.

Then for each symbol form a candidate list of similar symbols.

From these, we will then generate a set of new candidate words from the given word with the use of the symbol candidate list.

For example (symbol candidate list):

initial word "bear"

Candidate list for b – {d, o}

Candidate list for e – {o, c}

Candidate list for a – {o, z}

Candidate list for r – {k}

Generating new words (new candidate words):

dear, oear, boar, bcar, beor, becr, beak

generating a list of these candidate words:

doar dcar deor dezr deak

ooar odcar oeor oezr oeak

boor bozr boak

bcor bczr bcak

beok bezk

After this we can then check our set of generated candidate words against our bad word dictionary and count the number of bad words produced. The more bad words produced then the more likely that the source word is a bad word.

In order to check the similarity of these words we can use the help of the Lavenshtein Distance.

The algorithm for Lavenshtein Distance used to detect word similarity is as follows [15]:

```
int LavenshteinDistance(char S[1..M], char T[1..N]){
    declare int d[0..M, 0..N]
    for i from 0 to M
        d[i,0] := i  //the distance of any first string to an empty second string
    for j from 0 to N
        d[0,j] := j  //the distance of any second string to an empty first string
    for j from 1 to N
    {
        for i from 1 to M
        {
            if S[i] = T[j] then
                d[i,j] := d[i-1,j-1]  //no operation required
            else d[i,j] := minimum(d[i-1,j] + 1, //a deletetion
                                   d[i,j-1] + 1,  //an insertion
                                   d[i-1,j-1] + 1 )  //a substitution
        }
    }
    return d[M,N]
}
```

The above algorithm gives us the idea behind the calculation of the Lavenshtein Distance.

## 2.5 Conclusions on the second Chapter

After some research on some metrics for word similarity, our approach found it feasible to use the Lavenshtein Edit distance. This is because it will make it easier for us to be able to compare our masked bad word to our known bad word vocabularies that we use for training purposes.

We will also use the confusion matrix of the English alphabetical letters in both uppercase and lowercase so that we can identify special characters that are not English alphabetical letters and to identify those letters that have a high probability of being visually similar to each other.

For our training data and bad word vocabularies, we have identified several that have already been classified as bad words or fall under hate speech. Some datasets that we will use are not just words but are entire comments, in these comments we can find one or two words that would be labelled as bad words and we think this would be beneficial in training our algorithm to identify these bad words within comments in social media.

# Chapter 3

# Developing the software

## 3.1 Choosing Developing tools

For the task of classifying whether a word is a bad word or not we will use the help of libraries in scikit learn in python.

For this we looked in the Python Package Index (PyPI) for any existing libraries that could do this for me. The Only appropriate libraries we could find were the following:

- profanity (the ideal package name)
- better-profanity: *"Inspired from package profanity of Ben Friedland, this library is much faster than the original one."*
- profanityfilter (has 31 Github stars, which is 30 more than most of the other results have)
- profanity-filter (uses Machine Learning, enough said?!)

The profanity library contained a word list that had about 32 bad words shown below in fig 6:

33 lines (32 sloc) | 190 Bytes

```
anal
anus
ballsack
blowjob
blow job
boner
clitoris
cock
cunt
dick
dildo
dyke
fag
fuck
jizz
labia
muff
nigger
nigga
penis
piss
pussy
scrotum
sex
shit
slut
smegma
spunk
twat
vagina
wank
whore
```

Fig 3.1 - Word list in profanity library

The above libraries will be sufficient enough to train our algorithm to identify known bad words to begin with. better-profanity uses a 140-word wordlist and

profanityfilter uses a 418-word wordlist and the libraries detect bad words by simply looking for one of these words in the word list so we can easily use these with the Lavenshtein Distance for word similarities.

For the task of vectorization of the word we will use the Bag of words approach.

For this we will use the scikit-learn's CountVectorizer class, which basically turns any text string into a vector by counting how many times each given word appears.

If the only words in the English language were ‘the’, ‘cat’, ‘sat’, and ‘hat’, a possible vectorization of the sentence the cat sat in the hat might be as follows:

Table 3.1 Vectorization of words

| the | cat | sat | hat | ??? |
|---|---|---|---|---|
| 2 | 1 | 1 | 1 | 1 |

The ‘???’ represents any unknown word, which for this sentence is ‘in’. Any sentence can be represented in this way as counts of ‘the’, ‘cat’, ‘sat’, ‘hat’, and ‘???’!

A count of the words can then be represented as shown in table 3.2 below:

Table 3.2 –Bag of words representation

| Sentence | Bag of words representation |
|---|---|
| The cat sat | [1,1,1,0,0] |
| Hat on a cat | [0,1,0,1,2] |
| Cat cat cat cat cat | [0,5,0,0,0] |

In order to cater for the vast amount of words in the English language we will use the method “fit_transform()” which will do two of the following steps:

1) Fit – learns a vocabulary by looking at all words that appear in the dataset.
2) Transform: turns each text string in the dataset into its vector form.

For the task of training the algorithm we will use a Linear Support Vector Machine (SVM), which is implemented by scikit-learn's LinearSVC class.

The reason behind this decision is because the model learns which words are “bad” and how “bad” they are because those words appear more often in offensive texts. It’s as if the training process is picking out the “bad” words for us, which is much better than using a wordlist we might write ourselves!

It’s fast enough to run in real-time yet robust enough to handle many different kinds of profanity. The code for this process is found in Appendix A.

### 3.2 Future work and developments

Now this model is not good at picking up less common variants of profanities like “f4ck you” or “you b1tch” because they don’t appear often enough in the training data. Our goal will therefore be to try and take words that are detected through word similarities using the Lavenshtein Distance and the confusion matrix of the English alphabet. Words that are successfully classified as bad or profane will then be able to be incorporated into the training data so that our model can learn new variants of these masked bad words therefore making it more effective in the detection of masked bad words in the future. We will also be looking at a suitable software architecture to make our software more efficient and effective.

## References

1. Sood S. O., Antin J., Churchill E. Using crowdsourcing to improve profanity detection // Proc. Of 2012 AAAI Spring Symposium Series. – 2012.
2. Indurthi V. et al. Identifying and Categorizing Offensive Language in Social Media using Sentence Embeddings // Proc. of SemEval@NAACL-HLT 2019. – 2019.
3. Pavlopoulos J. et al. Convai at semeval-2019 task 6: Offensive language identification and categorization with perspective and bert // Proceedings of the 13th international Workshop on Semantic Evaluation. – 2019. – P. 571-576.
4. Thenmozhi D. et al. SSN_NLP at SemEval-2019 Task 6: Offensive Language Identification in Social Media using Traditional and Deep Machine Learning Approaches //Proceedings of the 13th International Workshop on Semantic Evaluation. – 2019. – P. 739-744.
5. Pathak V. et al. KBCNMUJAL@ HASOC-Dravidian-CodeMix-FIRE2020: Using Machine Learning for Detection of Hate Speech and Offensive Code-Mixed Social Media text //arXiv preprint arXiv:2102.09866. – 2021.
6. Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research, 16:321–357.
7. Shastay A. Misidentification of Alphanumeric Symbols in Both Handwritten and Computer-Generated Information //Home healthcare now. – 2015. – Vol. 33. – №6. – P. 338-339.
8. L.R. GEYER, Recognition and confusion of the lowercase alphabet Perception & Psychophysics 1977, Vol. 22 (5),487-490
9. J. T. TOWNSEND, Theoretical analysis of an alphabetic confusion matrix Perception & Psychophysics, 1971, Vol. 9 (IA)
10. A. SI LHOUSSAIN, Adapting the Levenshtein distance to contextual spelling correction.
11. Joni Salminen, Developing an online hate classifier for multiple social media platforms.
12. Salminen J, et al. Anatomy of online hate: developing a taxonomy and machine learning models for identifying and classifying hate in online news media. In: Proceedings of the international AAAI conference on web and social media (ICWSM 2018), San Francisco; 2018
13. Davidson T, et al. Automated hate speech detection and the problem of offensive language. In: Proceedings of eleventh international AAAI conference on web and social media, Montreal; 2017. P. 512–5
14. Bad bad words dataset, **https://www.kaggle.com/nicapotato/bad-bad-words**
15. Hao Chen, String Metrics and Word Similarity applied to Information Retrieval. 2012
16. W. R, Hamming, “Error detecting and error correcting codes,” Bell System Technical Journal 29, pp. 147–160, 1950.

17. F.J. Damerau, “A technique for computer detection and correction of spelling errors,” Communications of the ACM, 1964, pp. 171-176.
18. L.R. Dice, “Measures of the Amount of Ecologic Association Between Species,” Ecology 26, pp. 297–302, 1945.
19. Joachims, Thorsten. 1998. Text categorization with Support Vector Machines: Learning with many relevant features. In Machine Learning: ECML-98, ed. Claire Nédellec and Céline Rouveirol, 1398:137-142. Berlin/Heidelberg: Springer-Verlag. http://www.springerlink.com/content/drhq581108850171/.
20. Levenshtein, V I. 1966. “Binary codes capable of correcting deletions, insertions, and reversals.” Soviet Physics Doklady 10 (8): 707-710.

# Appendix A

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.calibration import CalibratedClassifierCV
from sklearn.svm import LinearSVC
from sklearn.externals import joblib

# Read in data
data = pd.read_csv('clean_data.csv')
texts = data['text'].astype(str)
y = data['is_offensive']

# Vectorize the text
vectorizer = CountVectorizer(stop_words='english', min_df=0.0001)
X = vectorizer.fit_transform(texts)

# Train the model
model = LinearSVC(class_weight="balanced", dual=False, tol=1e-2, max_iter=1e5)
cclf = CalibratedClassifierCV(base_estimator=model)
cclf.fit(X, y)

# Save the model
joblib.dump(vectorizer, 'vectorizer.joblib')
joblib.dump(cclf, 'model.joblib')
```