Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Carolin Dohmen

Detecting Hate Speech in Social Media - A Machine Learning Approach

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Carolin Dohmen

Detecting Hate Speech in Social Media - A Machine Learning Approach

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 21. Februar 2019

Carolin Dohmen

Thema der Arbeit

Detecting Hate Speech in Social Media - A Machine Learning Approach

Stichworte

Computerlinguistik, Maschinelles Lernen, Soziale Medien, Hassrede

Kurzzusammenfassung

Viele soziale Medien haben mit dem Auftreten von Hassrede zu kämpfen. Seit ein paar Jahren wird vermehrt an der Erkennung von Hassrede mit Hilfe von maschinellem Lernen und Computerlinguistik geforscht. Diese Arbeit befasst sich mit der automatischen Erkennung von Hassrede im Internet. Dafür werden verschiedene Verfahren des maschinellen Lernens sowie der Computerlinguistik kombiniert und untersucht. Die Ergebnisse der durchgeführten Experimente werden im Hinblick auf ihre Praktikabilität bewertet und verglichen.

Carolin Dohmen

Title of the paper

Detecting Hate Speech in Social Media - A Machine Learning Approach

Keywords

Natural Language Processing, Machine Learning, Social Media, Hate Speech

Abstract

Many social media platforms are affected by the presence of hate speech. In the last couple of years, machine learning and natural language processing approaches have been investigated to detect harmful user content on the web. This thesis deals with the problem of automated hate speech detection. Different machine learning and natural language processing algorithms are combined and investigated. The experiment results are then compared with respect to their usefulness for this task.

Contents

1. Introduction

The growing presence of social media has brought up a multitude of possibilities. Yet it also has created problems of a whole new dimension. Especially through the large amount of users, online content has become extremely hard to control. In the last decade, the use of machine learning has become increasingly popular in order to support content moderation in online platforms. This work is dedicated to exploring machine learning and natural language processing for the automated detection of harmful content in social media.

1.1. The Problem of Hate Speech

The discrimination and public defamation of minorities or certain population groups has always been a problem society has had to face. With the rise of modern media formats, this issue is now more present than ever. Social media platforms like Facebook, Twitter or Instagram, which heavily rely on user-generated content, provide stages not only for self-expression and social networking, but also for stirring up hatred against others.

By now, many countries have introduced laws to oppose what is commonly called *hate speech*. Beyond that, most online platforms apply individual language regulation policies, built up on their own definitions. While there is no universal definition of hate speech, most regulations refer to it as the propagation of racism, sexism or religious hate.

Different approaches have been utilized to fight the dissemination of this kind of harmful content among the web. Due to the continuous generation and therefore massive amount of data, online platforms often rely on flagging [10]. This means, that users can report content directly to the platform provider, so that it can be evaluated and removed. In addition to that, bigger platforms outsource active content moderation to external companies [2].

In the last couple of years, automated hate speech detection mechanisms have become a topic of particular interest. Companies like Facebook or Google are already using artificial intelligence for this purpose. Still there is a lot of room for improvement, since existing systems do not perform well enough to replace human reporters or moderators. According to [19], only 38 percent of the posts that have been removed from Facebook because of their hateful content, have priorly been marked by the website's detection system.

Not only for machines, hate speech detection is an extraordinarily hard task. In many cases even for humans it is extremely difficult to distinguish hate speech from other content which might be toxic, but does not fall under the category of hate speech. [33] and [29] for example show that hate speech annotators more often than not disagree on their annotations, which is why human agreement can be seen as an upper bound on machine learning classification success [29].

1.2. Goals

This work focuses on exploring machine learning techniques for the purpose of identifying hateful language in social media. Therefore different preselected feature generation, resampling and classification algorithms are combined and applied on a social media dataset. The main objective is to get an insight on how different combinations algorithms perform on the dataset in order to help with the decision on which methods to use for programmatic hate speech detection.

1.3. Structure

Chapter 2 elaborates some of the principles of machine learning and natural language processing. Furthermore an overview on past and current research on hate speech detection through machine learning is given. Chapter 3 provides a deeper insight on the dataset that this work is based on. In Chapter 4, the different text processing, resampling and machine learning algorithms which are used for classification are introduced and explained. The general test setup as well as implementation details are described in Chapter 5. The test results of different algorithm combinations are then presented in Chapter 6. Finally, Chapter 7 shows an evaluation of the examined methods and also provides an assessment on their viability in practice. In addition to that, an outlook on further work regarding this topic is given.

2. Conceptual Background

In order to fully understand the nature of the problem, it is necessary to explain basic methodological concepts. Therefore this chapter introduces concepts of machine learning and natural language processing. In addition to that, related work in the context of automated hate speech detection is discussed.

2.1. Machine Learning Basics

Many tasks in decision making, pattern recognition and other areas, which used to require human intelligence, have been automated through the use of *machine learning* (further referred to as *ML*), which is a field of artificial intelligence. According to [30], ML can be divided into three categories, briefly described as follows:

- **Supervised Learning**
  In this approach, a set of input/output tuples is given. Based on this *training set*, a function is learned, which maps input to output values. This function, called *hypothesis*, is then applied on a *test set*, in order to judge its predictive power and to improve its performance on predicting new unseen instances. The learning method is called *regression*, if the output is numeric. It is called *classification*, if output values can be seen as a finite set of categories.

- **Unsupervised Learning**
  In this approach, a set of unlabeled instances is given. The main goal here is to find a model that recognizes potential patterns in the dataset, which is called *clustering*. This method is often used to identify classification categories, where no prior information on categories is available.

- **Reinforcement Learning**
  This category is based on the principle of learning by positive and/or negative feedback. It aims to train a model to find the best chain of decisions (*policy*), solving a specific problem. Decisions which are beneficial are rewarded whereas disadvantageous decisions are punished.

Besides those three categories many popular algorithms use **semi-supervised learning**, which as the name suggests is a mix of supervised and unsupervised learning. Here both labeled and unlabeled instances are used to train an ML model. This work however makes use of supervised learning algorithms, since class labels for tweets are already given.

2.2. Natural Language Processing

The computational processing of written and spoken natural language has a multitude of use cases. Applications utilizing *natural language processing* (further referred to as *NLP*) therefore are becoming increasingly popular. NLP combines different fields such as signal processing, machine learning, linguistics and psychology, just to name a few [9]. Besides speech recognition, machine translation or chat bots, NLP is also used for text analysis and classification. The automated detection of hate speech represents such a text classification problem.

Most NLP problems can be considered very difficult to solve. Unlike formal languages, which can easily be handled by machines, human language has more nuances to it, making it difficult for computers to understand and interpret. Latent factors like irony, sarcasm, humor or figures of speech can drastically change the meaning of a piece of text. Without context, it is almost impossible to fully determine the semantic meaning of text using those factors. NLP methods therefore have to find a way to overcome those obstacles.

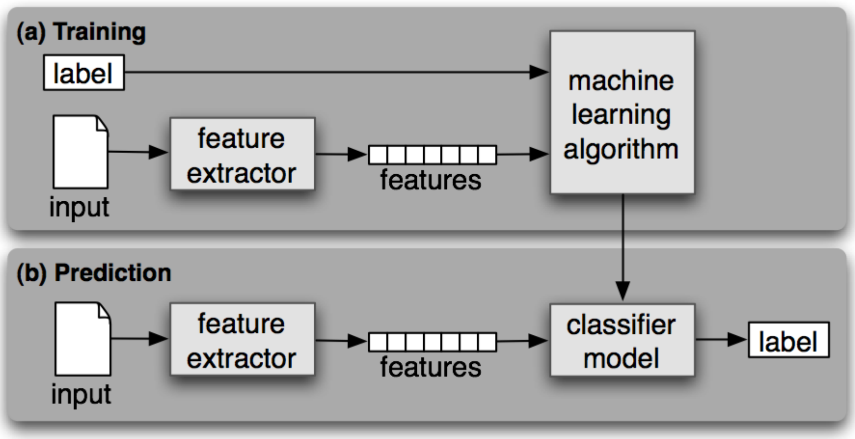


Figure 2.1.: Text classification pipeline. Reproduced from [3].

Most NLP applications follow a number of sequential steps. The first step includes transforming raw text into numerical feature vectors. To achieve this, the text has to be split up into a finite number of entities, which are called *tokens*. Accordingly, this procedure is called *tokenizing*. The most common token granularity is a small set of words or characters. Another popular preprocessing step is the removal of *stop-words*. This term refers to words like 'a', 'and' or 'the', which frequently appear in text and do not carry any semantic information. In addition to that it can be useful to normalize words by using their lowercased stems and remove punctuation. After transformation the feature vectors can be used to train and test ML classifiers, which expect numerical features as input.

Figure 2.1 shows an exemplary pipeline with raw text documents as input and class labels as output. Both in training and prediction, text samples are transformed into feature vectors, before they can serve as input for the machine learning algorithm in the training phase or the model in the test phase. The model training and class prediction step does not differ from other classification problems that do not involve text.

2.3. Related Work

Research on automated hate speech detection, especially in social media, has become increasingly relevant during the last couple of years. With the rise of social media, it has become a subject of public interest. Companies like Facebook, Google or Yahoo are actively participating in doing research in order to find solutions for this problem.

However, there are just a few publicly available datasets that are large enough to train ML models with. To achieve suitable results, datasets have to be sufficiently big. Besides that human workforce is needed in order to provide labels for the samples. In the case of hate speech it can be useful to employ trained annotators to label a dataset. The identification of hate speech requires a certain amount of skills, especially when it has to be distinguished from language that is just offensive. Most currently available datasets differ in the authors' predefined sample categories. [25] extracted two dataset (2,150,176 and 1,174,509 samples) from Yahoo! Finance and News, which were labeled 'clean' or 'abusive'. These datasets are currently the biggest ones that have been extracted so far. [32] collected a dataset of 1,000 samples, which were categorized as 'anti-semitic, antiblack, anti-asian, anti-woman, anti-muslim, antiimmigrant or other-hate' [32]. 16,914 samples were extracted from Twitter and annotated 'sexist', 'racist' or 'neither' by [34]. [20] provided a set of 24,582 Tweets labeled 'racist' or 'non-racist'. [6] also used Twitter to extract 1,901 tweets labeled 'cyber hate' or 'benign'.

Hate speech classification approaches differ mainly in feature engineering methods and in using different classifiers. One of the most popular approaches for feature engineering is the usage of word or character n-grams (see 4.1.1). This method is used by [34], [23], [6], [25], [32], [20], [38] as well as [11]. [25], [32] and [13] use grammatical characteristics to generate features. A popular example is part-of-speech tagging, where each word is assigned its part of speech in its context. Lexicon-based features are used by [6], [25], [13] and [38]. These approaches look for the presence of certain words or phrases in the text by using word lists that contain offensive, racist or sexist terms or by using lists that include words implying specific emotions. Features deduced from a text sample's sentiment score (see 4.1.4) are utilized by [13] and [38]. An increasingly popular feature generation method based on neural networks is the use of word or document embeddings (see 4.1.3). [12], [37], [1] and [38] generate features through these methods. [36] and [38] make use of latent Dirichlet allocation (see 4.1.2).

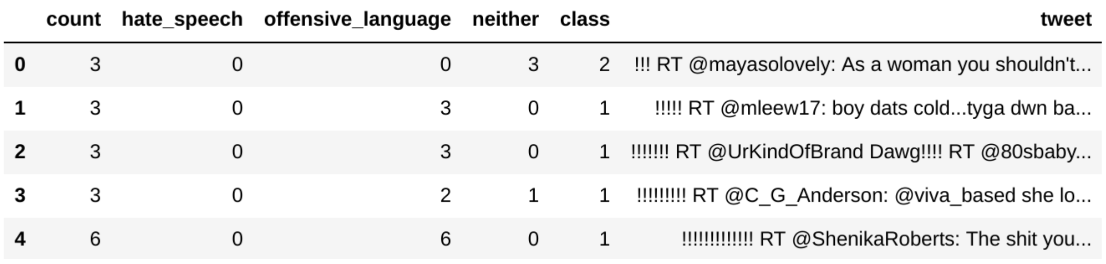
Many approaches, such as [11], [12], [6], [34] and [36], use Logistic Regression (see 4.3.1) for the purpose of classification. Another popular approach is the use of Support Vector Machines, which are utilized by [23], [6], [32] and [36]. Decision Trees, which belong to the rule based classification approaches, and Random Forest as an ensemble method of Decision Trees, are used by [6] and [36]. [20] uses the fast and simple Naive Bayes classifier (see 4.3.3). Furthermore, deep learning approaches are becoming more and more popular and are applied for example by [37] and [1].

3. Dataset

The following chapter sheds light on the dataset that is used in this thesis. First, basic features of the set are discussed. Furthermore an explanation is given, why this specific dataset has been chosen for this work. The part that follows gives deeper insights on qualitative and quantitative features of the dataset.

3.1. Description

The dataset used in this work has been created by [11]. It consists of 24,783 tweets which have been sampled from Twitter using the Twitter API. The tweets have been drawn from Twitter users who have posted content containing words from a collection of hate speech terms maintained by Hatebase.org. All samples have been labeled by human annotators as either 'hate speech', 'offensive language' or 'neither'.



	count	hate_speech	offensive_language	neither	class	tweet
0	3	0	0	3	2	!!! RT @mayasolovely: As a woman you shouldn't...
1	3	0	3	0	1	!!!!! RT @mleew17: boy dats cold...tyga dwn ba...
2	3	0	3	0	1	!!!!!!! RT @UrKindOfBrand Dawg!!!! RT @80sbaby...
3	3	0	2	1	1	!!!!!!!!! RT @C_G_Anderson: @viva_based she lo...
4	6	0	6	0	1	!!!!!!!!!!!!! RT @ShenikaRoberts: The shit you...

Figure 3.1.: First five rows of the dataset.

Figure 3.1 shows the general structure of the dataset. The column *count* represents the number of annotators that have assigned labels to the current tweet. The number of annotators varies throughout the dataset. The columns *hate_speech*, *offensive_language* and *neither* each contain the number of annotators which have assigned the respective label to the tweet. The *class* column consequently contains the majority vote of the three preceding columns. As the name says, the *tweet* column holds the extracted tweet.

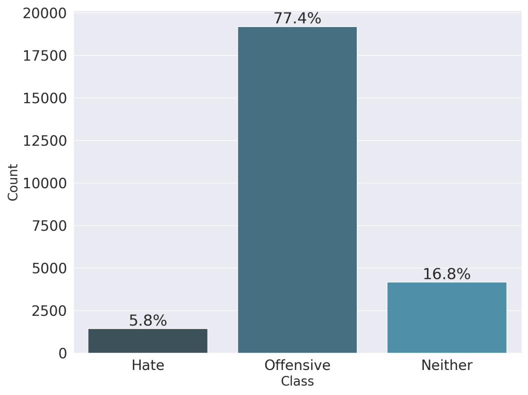
The decision to use this specific dataset for this thesis is based on several reasons. A major point is the size of the dataset. Not many large enough annotated hate speech datasets are publicly available. This set provides a sufficient amount of samples, so that the chance for reasonable classification results is bigger than with a smaller dataset. Furthermore the distinction between hate speech and insulting, but not hateful language is a problem that is far from trivial and harder to deal with than the binary case of distinguishing between offensive and non-offensive language. Hate speech detection systems in practice will have to be able to handle this distinction in order to avoid censorship and concentrate on protecting users.

3.2. Data Exploration

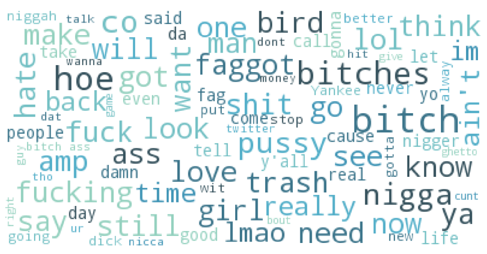
Figure 3.2a shows that the dataset is highly imbalanced. With 77.4% the majority of tweets is labeled offensive, whereas with 5.8% just a very small percentage of tweets are actually considered hate speech. Most classifiers tend to perform badly on imbalanced datasets [8], because instances of the underrepresented class are likely to be ignored. In this work oversampling is used to oppose the class imbalance problem.

Since a hate speech dictionary was used to extract the tweets, a multitude of the samples contain terms that can be considered offensive. Figure 3.2b shows a word cloud of the 90 most frequent words in the whole dataset. Besides obvious swear words there are also terms like 'bird' or 'trash', which are not inherently offensive, but are often used in a hate speech context. Figure 3.2c and Figure 3.2d show the most frequent word combinations, which also underline the toxicity in many tweets. Looking at Figure 3.2e and Figure 3.2f, it becomes visible that offensive and hate speech tweets are shorter but in turn contain more offensive words on average.

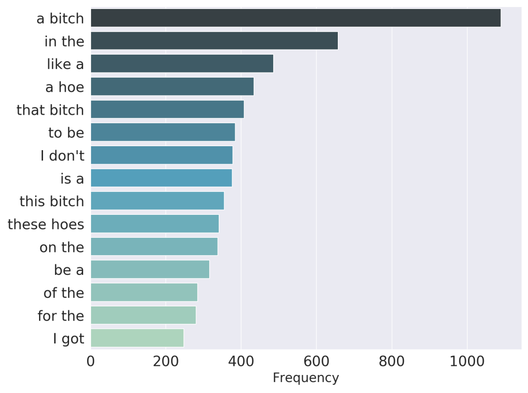
Collectively the examined qualities suggest that the biggest difficulty lies in differentiating between tweets that contain offensive content and tweets that contain hateful content. Both categories show similar qualities and cannot necessarily be distinguished by their offensive language. This underlines the inherent difficulty of the hate speech detection problem.



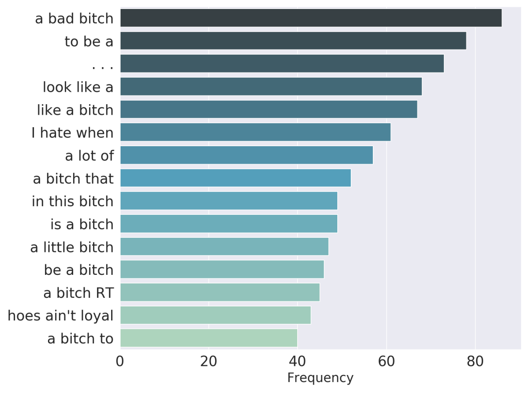
(a) Distribution of classes.



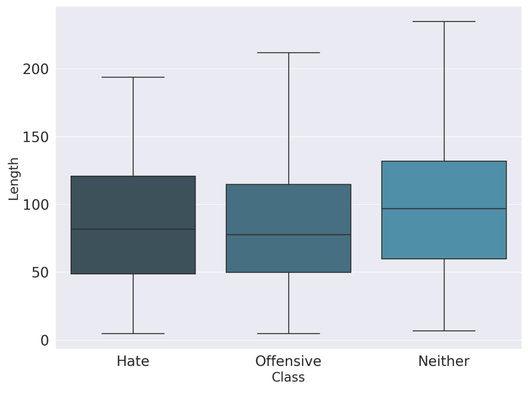
(b) Word cloud of tweets.



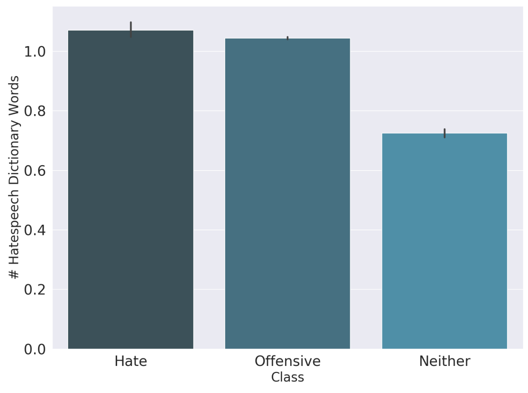
(c) The 15 most frequent word bigrams.



(d) The 15 most frequent word trigrams.



(e) Tweet length by class.



(f) Number of offensive words by class.

Figure 3.2.: Visualization of different dataset and text qualities.

4. Methods

The experiments conducted in this thesis are comprised of three main steps: Feature generation, oversampling and classification. In the feature generation step, raw text is preprocessed and transformed into feature vectors. After that, an oversampling algorithm is applied to the train data. This step is included because the dataset that is used in this thesis is highly imbalanced. The hate speech class, which is the most interesting one, is underrepresented, so that training suitable models can be difficult. Through oversampling this problem can be mitigated. The final step is the classification of the dataset, including training a classifier with a train set and testing the resulting model with a test set. The following chapter introduces the different algorithms used in this thesis and explains, why they have been chosen for the experiments.

4.1. Feature Generation

In this section different feature generation methods are introduced. A suitable feature engineering practice is a crucial part of text analysis. Other than for example in medical or economic datasets, text data always has to be preprocessed and transformed in order to be used as input for a ML classifier. Therefore classification performance relies heavily on which feature generation method is used and which qualities of the text are represented by the features.

The approaches that have been chosen to be compared are *term frequency-inverse document frequency* (*TF-IDF*), *latent Dirichlet allocation* (*LDA*), *Paragraph Vector* and *sentiment score*. The first three are used as primary feature generation methods. The sentiment score serves as a supplementary method, so that each of the other three approaches are tested both alone and in combination with sentiment features.

Out of the multitude of available methods TF-IDF has been chosen because of its popularity among text classification applications. Even though the approach is rather simple, it often leads to considerable results. The choice for LDA is based on curiosity about how a statistical model performs in comparison to a simpler method like TF-IDF. Paragraph Vector has been picked as a representative of the newer feature engineering approaches that make use of neural networks. This choice addresses the question if neural networks generally perform better than other feature generation methods. Sentiment score features have been chosen in order to investigate whether adding those values can improve the classification performance.

The methods that have been chosen for this thesis are just a selection out of many possible feature generation approaches. Lexicon-based feature engineering methods however have not further been investigated because the dataset has been collected by using a hate speech lexicon. Data exploration shows that the majority of samples contain offensive words, therefore those features might not contribute to good classification results.

4.1.1. TF-IDF

The TF-IDF algorithm is a variant of the so called *bag-of-words* models. These feature generation models make use of word and character *n-grams*, which are sequences of either words or characters of length *n*. A sentence is represented by a set ('bag') of n-grams, that are unordered and therefore independent of their actual position in the text. Consequently this representation leads to the fact that information on word or character order and context is lost. The goal of bag-of-words however is to transform the text samples into feature vectors that contain (weighted) frequencies of the n-grams present in each text sample. The sentence 'I am eating pizza.' for example would result in the bigrams 'I am', 'am eating' and 'eating pizza'.

The use of simple n-gram frequency can be problematic for text classification. Very frequent terms, as well as very rare terms are uninteresting for classification, since they do not provide information that is useful for predicting class labels. The TF-IDF algorithm offers a strategy to overcome this problem. While word counting algorithms only provide information on a term's frequency in the current document, TF-IDF gives insight on its actual importance. It weighs each term frequency by a value that represents the term's relevance in the whole corpus. The TF-IDF algorithm is comprised of the following steps:

- Compute the term frequency $tf_{t,d}$ of term t in text document d:

$$tf_{t,d} = \begin{cases} 1 + \log_{10} \text{count}\big(t, d\big), & \text{if count}\big(t, d\big) > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

- Compute the inverse document frequency idf_t:

$$idf_t = \log_{10} \left(\frac{N}{df_t} \right), \tag{4.2}$$

  where df_t is the number of documents containing term t and N is the total number of documents in the corpus.

- Multiply term frequency and inverse document frequency:

$$tf\text{-}idf_{t,d} = tf_{t,d} \times idf_t. \tag{4.3}$$

4.1.2. Latent Dirichlet Allocation

With latent Dirichlet allocation, a generative probabilistic model [5] is used to create features. LDA is based on the assumption, that each document in a document collection has an underlying distribution of *topics*, whereas each topic is modeled as a distribution over all words from the collection. According to the assumptions LDA is based on, a single document d_i with N words is generated by the following steps:

1. Draw $\theta_i \sim$ Dirichlet(α)
2. For each word w_j in d_i, $j = 1...N$:
   a) Draw a topic $z_{i,j} \sim$ Multinomial(θ_i)
   b) Draw a word $w_{i,j} \sim$ Multinomial($\beta_{z_{i,j}}$)

LDA therefore tries to infer the distribution of topics in all documents as well as the distribution of words belonging to a topic.

A graphical model of LDA can be seen in Figure 4.1. The nodes represent random variables, of which shaded ones are observed and white ones are hidden. The plates (rectangular boxes) stand for replicated variables, with the number of replications on the bottom right side. The black continuous edges show dependencies between variables.

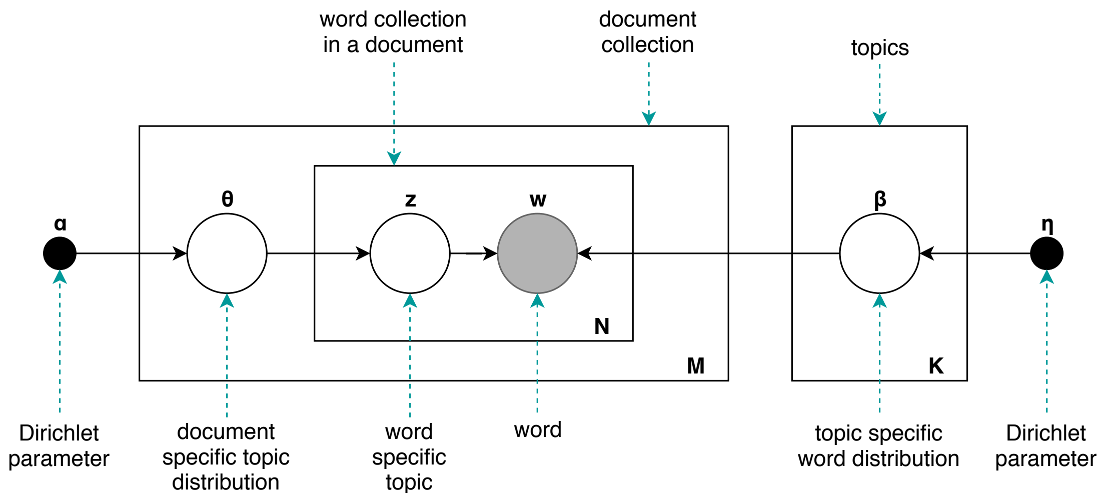

word collection in a document
document collection
topics
α
θ
z
w
β
η
N
M
K
Dirichlet parameter
document specific topic distribution
word specific topic
word
topic specific word distribution
Dirichlet parameter

Figure 4.1.: Graphical model of LDA. Adapted from [5].

To make use of LDA, the parameters α and η have to be estimated and the posterior distributions of the hidden variables have to be inferred. Since this is computationally intractable, approximation methods are used to learn an LDA model. Once a model is learned, it can be used to generate features for classification. The feature vector of each document simply consists of the underlying topic probabilities. Thus the number of features per sample corresponds to the number of topics in the document collection, which is arbitrarily set before training the model.

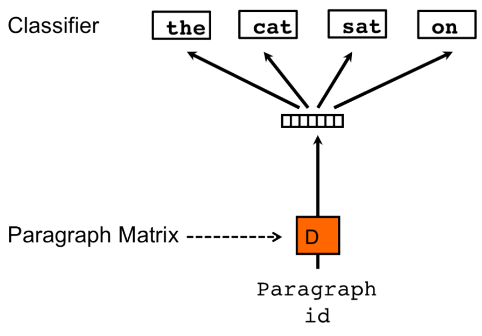
Since the number of topics is just a fraction of the number of words present in a corpus, LDA generates feature vectors that are remarkably smaller than with TF-IDF, in which feature vectors correspond to the number of unique words in the whole corpus. This is especially useful when the corpus is very big, as the training runtime of the classifier will be improved.

4.1.3. Paragraph Vector

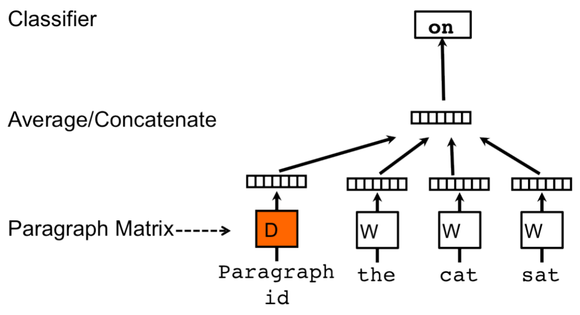
The Paragraph Vector framework [21] provides distributed representations of documents. In literature it is also referred to as *Doc2Vec*. There are two major versions of Paragraph Vector. In the Distributed Memory version (*PV-DM*), a small window of words is used to predict the next word in a context window. Each word is represented by a unique column vector in a word matrix W. In addition to that, every document is encoded in a separate unique column vector in a document matrix D. In order to predict a certain word, context word vectors as well as the document vector are averaged or concatenated. Word and document vectors are learned by using neural networks.

In contrast to bag-of-words, Paragraph Vector creates word vectors that encode similarities between words and takes their context into account, so that semantically similar words will have similar vectors. These similarities can also be shown through arithmetic operations like vector addition.

The other major variant is called Distributed Bag-of-Words version (*PV-DBOW*). In contrast to PV-DM, it only uses document vectors to predict a window of words in a context. The training is also done with a neural network.



(a) PV-DBOW version of Paragraph Vector.



(b) PV-DM version of Paragraph Vector.

Figure 4.2.: Different versions of Paragraph Vector. Reproduced from [21].

An illustration of Paragraph Vector's working principles can be seen in Figure 4.2. In this thesis the PV-DM version of Paragraph Vector is used. Comparing PV-DM and PV-DBOW exceeds the scope of this work. Therefore the default setting of the implementation from the Python library is used.

4.1.4. Sentiment Score

Sentiment analysis is a scientific approach to finding out 'what other people think' [26]. With the help of ML and NLP methods it aims to quantify human emotion in text. A popular example is the rating of online product reviews. Through text analysis a sentiment score is assigned to each review, judging the user's emotions towards the respective product. In the context of hate speech detection, sentiment analysis is used to evaluate the emotional polarity of a piece of text. It is especially helpful for distinguishing between non-offensive and offensive language, since the latter one is much more likely to have a negative sentiment score. A visualization of sentiment scores per class for this dataset can be seen in Figure 4.3. The difference between the neither class and the hate speech and offensive class becomes very visible here.

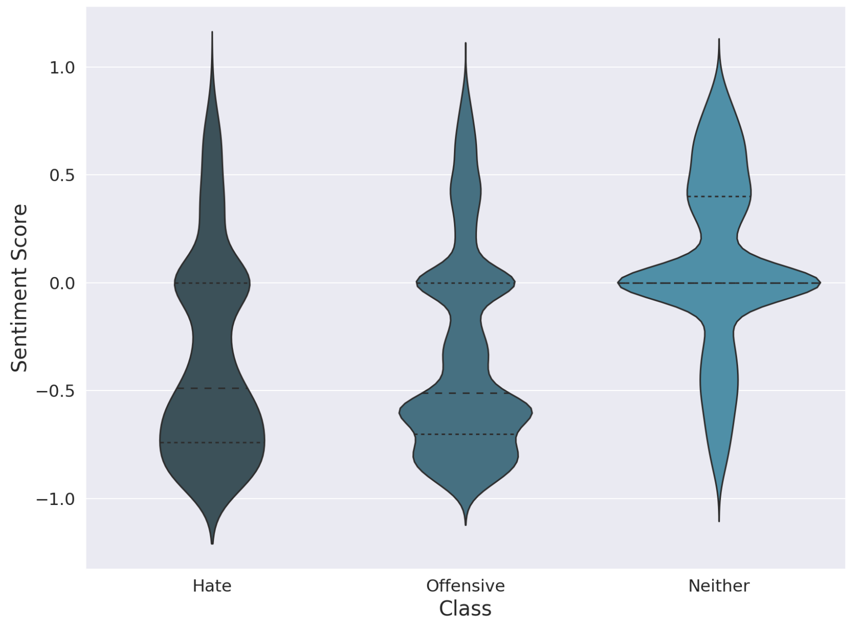


Figure 4.3.: Sentiment scores of tweets per class.

This work makes use of the pre-trained sentiment analyzer *VADER* [17] to assign sentiment scores to tweets. Every tweet is assigned three ratios for negative, neutral and positive sentiment, which add up to 1. Sentiment features are used as an additive for features generated by the other three methods that are used in the thesis. To investigate the impact of the sentiment scores, TF-IDF, LDA and Paragraph Vector are applied both with and without sentiment features, so that the differences can be compared.

4.2. Oversampling

The following section describes the two different oversampling methods that are used in this work. Imbalanced datasets can be an obstacle when it comes to training ML models. During the learning phase, classifiers develop a bias towards the class or classes that include the majority of samples. The models then tend to misclassify minority class samples. This is especially fatal if the minority class constitutes the 'interesting' class, which is the case in the present dataset.

Resampling the data often helps to gain better classification results. There are two major options for resampling. One possibility is to decrease the size of the majority class by removing data. This is called undersampling. The other option includes generating new minority class samples in order to adjust the class distribution, which is called oversampling.

This work uses oversampling instead of undersampling, because the risk of losing valuable information through removing single samples should be avoided. The two oversampling algorithms *SMOTE* (*Synthetic Minority Over-sampling Technique*) and *ADASYN* (*Adaptive Synthetic Sampling Approach*) have been chosen, because they can be considered the most popular state-of-the-art oversampling algorithms. Simple random resampling of data is not applied, since this method can lead to overfitting [8].

4.2.1. SMOTE

SMOTE constitutes an approach for resampling imbalanced datasets by generating new synthetic data samples [7]. The amount of additional samples created by SMOTE can be chosen arbitrarily. It is represented by a parameter N, so that a total of $(N/100) \times T$ new samples are generated, where T is the count of minority class samples. For every minority sample, SMOTE applies a *k-nearest-neighbors* algorithm, where k is also arbitrarily set. A new synthetic sample is calculated as follows:

$$\mathbf{s} = \mathbf{x} + (\mathbf{nn} - \mathbf{x}) \times d, \tag{4.4}$$

where $\mathbf{x}$ is an existing minority sample, $\mathbf{nn}$ is a randomly chosen nearest neighbor, d is a random weighting factor between 0 and 1 and $\mathbf{s}$ is the new synthetic feature vector.

4.2.2. ADASYN

Similar to SMOTE, *ADASYN* also generates synthetic data samples [15]. But instead of parameterizing the amount of resampled minority samples, a density distribution is used to determine how many synthetic samples per minority class sample should be generated. Through this technique, ADASYN focuses on data samples that are especially hard to classify. In the original algorithm the number of new synthetic samples g_i for a minority sample $\mathbf{x_i}$ is calculated as follows:

a) Calculate the number of synthetic data examples that need to be generated for the minority class:

$$G = (m_l - m_s) \times \beta \tag{2}$$

Where $\beta \in [0, 1]$ is a parameter used to specify the desired balance level after generation of the synthetic data. $\beta = 1$ means a fully balanced data set is created after the generalization process.

b) For each example $\mathbf{x_i} \in minorityclass$, find K nearest neighbors based on the Euclidean distance in n dimensional space, and calculate the ratio r_i defined as:

$$r_i = \Delta_i/K, i = 1, ..., m_s \tag{3}$$

where Δ_i is the number of examples in the K nearest neighbors of x_i that belong to the majority class, therefore $r_i \in [0, 1]$;

c) Normalize r_i according to $\hat{r_i} = r_i / \sum_{i=1}^{m_s} r_i$, so that $\hat{r_i}$ is a density distribution ($\sum_i \hat{r_i} = 1$)

d) Calculate the number of synthetic data examples that need to be generated for each minority example $\mathbf{x_i}$:

$$g_i = \hat{r_i} \times G \tag{3}$$

where G is the total number of synthetic data examples that need to be generated for the minority class as defined in Equation 2. [15]

The computation of the new feature values is similar to SMOTE. The quoted algorithm shows that ADASYN pays special attention to data samples that are particularly close to majority samples. The more majority samples can be found among the k nearest neighbors of a minority sample, the more synthetic data points will be generated for the respective minority sample.

4.3. Classification

This section describes the ML algorithms that are used for classifying the dataset. As mentioned before, only supervised learning methods are used in this thesis. All classifiers go through two main phases, which are training and testing. In the training phase, classifiers will be fed with train data in order to learn a model. This model will then be used in the testing phase to predict labels of test data.

The classification algorithms chosen in this thesis are *Logistic Regression*, *Support Vector Machines* (*SVM*), the *Naive Bayes classifier* and the *Multilayer Perceptron*. The first three have been chosen for this thesis because they are very popular in text classification. The main goal here is to compare between classification results and see if there are major performance differences. The last one has been chosen as a representative of the neural network classifier family. The Multilayer Perceptron is one of the simpler neural network models, unlike other deep learning approaches which are more elaborate. Since it would exceed the scope, other deep learning methods are not considered in this thesis.

4.3.1. Logistic Regression

Logistic Regression can be described as a statistical ML model. Originally, the model has been designed to solve binary classification problems. Multiclass problems however can also be solved, for example by treating each class as a binary problem. As the name suggests, Logistic Regression makes use of a *logistic function* (or *logit*) in order to distinguish between classes. Its output acts as the probability that the given sample belongs to the positive class. The definition of a logistic function can be seen in Equation 4.5. It is a special form of a sigmoid function, which is outlined in Equation 4.6.

$$p = \sigma(\theta^T(x)). \tag{4.5}$$

$$\sigma(x) = \frac{1}{1 + e^x} \tag{4.6}$$

θ is the coefficient vector that has to be learned during the training phase. Given the logistic function's output p for a specific sample, which lies in the range $(0, 1)$, the distinction between positive and negative class is done as follows:

$$y = \begin{cases} 1, & \text{if } p \geq 0.5, \\ 0, & \text{otherwise.} \end{cases} \tag{4.7}$$

The training goal of Logistic Regression is to find a θ that results in the best prediction results. This is done by minimizing the following cost function for the whole set of training instances:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^i \log(h_\theta(x^i)) + (1 - y^i) \log(1 - h_\theta(x^i)\right], \tag{4.8}$$

where x^i represents the feature vector of the training instance, y^i is the actual class label and $h_\theta(x^i)$ is the predicted class label for x^i. Many implementations of Logistic Regression use gradient descent to solve this optimization problem.

4.3.2. Support Vector Machines

Support Vector Machines are ML models that use hyperplanes for classification. A hyperplane $\vec{x}$ is defined by the following equation:

$$\vec{x} : f(x) = \vec{w} \cdot \vec{x} + b = 0, \tag{4.9}$$

where $\vec{w}$ is a normal vector and b marks the distance to the origin. SVM tries to find hyperplanes that divide the data into different classes, so that the prediction accuracy based on those planes is maximized. The following rule can be applied in order assign a class label $y \in \{1, -1\}$ to a feature vector $\vec{x}$ with normalized parameters $\vec{w}$ and b:

$$y = \begin{cases} 1, & \text{if } \vec{w} \cdot \vec{x} + b \geq 1, \\ -1, & \text{if } \vec{w} \cdot \vec{x} + b \leq -1. \end{cases} \tag{4.10}$$

The learning goal of SVM is to find the largest possible margin. This problem is geometrically shown in Figure 4.4. (b) shows a hyperplane that provides a larger margin and therefore results in a higher classification accuracy than the hyperplane in (a). The actual support vectors are the data points with the shortest distance to the hyperplane on both sides.

In general, hyperplanes provide a good solution for data that is linearly separable. In non-artificial datasets this case is rather rare. To achieve better classification accuracy, a *kernel function* is used to transform the data into a higher dimension, so that eventually it can be linearly separated.

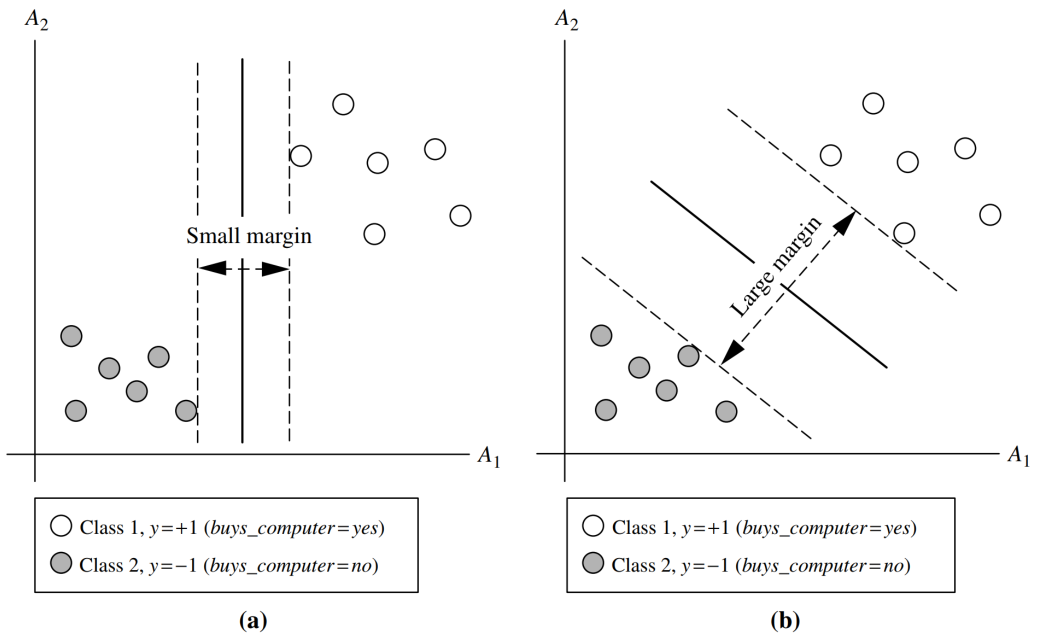

A_2
A_1
Small margin
Large margin
○ Class 1, $y=+1$ (*buys_computer=yes*)
● Class 2, $y=-1$ (*buys_computer=no*)
(a)
(b)

Figure 4.4.: Dividing data samples through hyperplanes. Reproduced from [14].

4.3.3. Naive Bayes Classifier

The Naive Bayes classifier belongs to the group of probabilistic ML models. It is a simple and fast algorithm, that often performs relatively well despite its simplicity. The Naive Bayes classifier is based on Bayes' Theorem, which is described in Equation 4.11. In connection with a classification problem, the theorem provides a possibility to estimate the probability that a sample belongs to a certain class, given the sample's set of feature values.

$$P(y|x_1, x_2, ..., x_n) = \frac{P(y)P(x_1, x_2, ..., x_n|y)}{P(x_1, x_2, ..., x_n)} \tag{4.11}$$

$P(y|x_1, x_2, ..., x_n)$ is the conditional probability of the class label given a certain set of feature values, $P(y)$ is the actual observed class label, $P(x_1, x_2, ..., x_n|y)$ is the conditional probability of the feature values given a certain class label and $P(x_1, x_2, ..., x_n)$ is the probability of the feature values.

Naive Bayes utilizes the ('naive') assumption, that all features are conditionally independent from each other. Using the chain rule of probability calculation, this can be formulated as follows:

$$P(x_i|x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n, y) = P(x_i|y) \tag{4.12}$$

The probability that a feature x_i takes on a certain value, given all other feature values of $x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n$ as well as the class label y, is the same as the probability given just the class label y. This assumption leads to the following formulation:

$$P(y|x_1, x_2, ..., x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, x_2, ..., x_n)} \tag{4.13}$$

The denominator is constant, since it represents feature value probabilities that are already given through the observed feature values. Therefore the resulting model can be described like this:

$$P(y|x_1, x_2, ..., x_n) \propto P(y)\prod_{i=1}^{n} P(x_i|y) \tag{4.14}$$

Consequently, a class prediction $\hat{y}$ can be made through the following rule:

$$\hat{y} = \underset{y}{argmax} P(y)\prod_{i=1}^{n} P(x_i|y) \tag{4.15}$$

The class label with the highest conditional probability will be chosen for a given sample.

4.3.4. Multilayer Perceptron

The Multilayer Perceptron is one of the simpler classifiers using the concept of neural networks. The model consists of an input layer, an output layer and one or more hidden layers in between. Every layer is comprised of a set of nodes, which are also referred to as *neurons* according to their biological archetype. An exemplary neural network with a single hidden layer is shown in Figure 4.5.

Every node of the input layer represents a single feature from a feature vector. The nodes of the hidden layers compute weighted sums of the first layer's output and transform it by applying a non-linear *activation function.* The most popular activation functions are the *hyperbolic tangent function*, the logistic function (see Equation 4.5) and the *Rectifier Linear Unit.* The latter one is used in this work and can be described through Equation 4.16. Finally the output layer generates an output with which the class label can be determined. Every neuron in the output layer represents one of the possible classes. Since the data flow is unidirectional from the input layer to the output layer, the multilayer perceptron is called a *feedforward* neural network.

$$f(x) = max(0, x) \tag{4.16}$$

For the learning process an error function has to be chosen in order to measure the deviation between the predicted label and the actual label of a training sample. This error function is minimized by gradually adjusting the individual weights in every node. The most common way to learn the weights is gradient descent in combination with backpropagation.

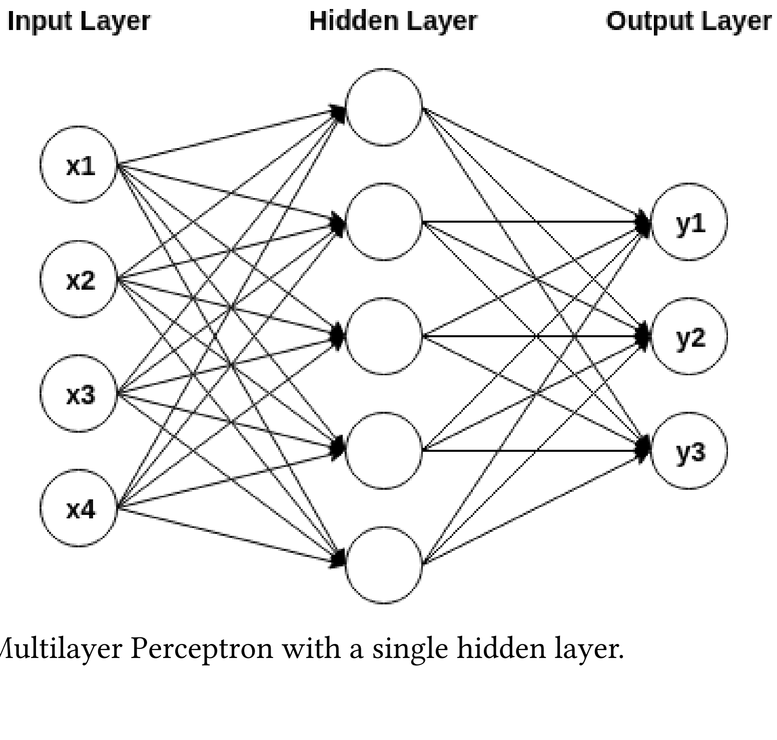


Figure 4.5.: Multilayer Perceptron with a single hidden layer.

5. Experiments

In order to investigate the problem this thesis deals with, a set of experiments are conducted. The following chapter describes the concept and setup of the conducted experiments. Furthermore it describes the technical details and concrete implementation of the tests.

5.1. Test Pipeline

All experiments aim to compare combinations of different feature generation, oversampling and classification algorithms. Every test therefore follows the same pipeline, which is shown in Figure 5.1. As a first step, the raw text data samples are preprocessed. This includes tokenizing as well as the removal of common stop-words. The actual feature generation procedure concludes with transforming every tweet into a numerical feature vector, so that the dataset that is used for the following steps is comprised of numerical feature vectors and their respective labels. The resulting

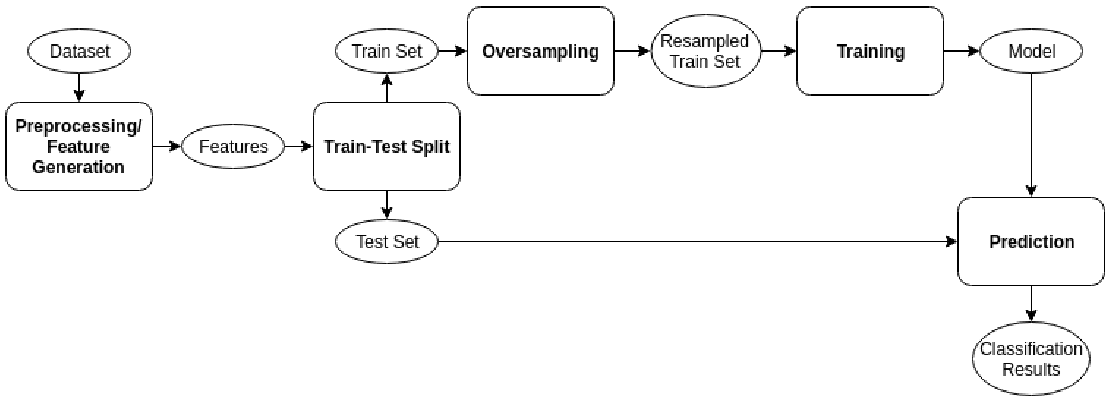


Figure 5.1.: Experiment pipeline.

dataset is randomly split into a train set and a test set by a ratio of 4:1. The test set is held out for model evaluation. This step is necessary to be able to estimate the model's actual performance. If a model is trained and tested with the same dataset, it is likely to make very accurate predictions on the known dataset, since it has been trained with it. This phenomenon is called *overfitting*. To actually evaluate the prediction quality, the model has to be tested with unseen samples. Other overfitting prevention approaches like cross-validation are not used in order to curb the overall experiment runtime. After splitting, an oversampling algorithm is applied to the train set in order to increase the amount of minority class samples. The resampled train set is then used to train the respective classifier. On completion of the learning phase, the resulting model is used to predict class labels for the held out test set. Finally, the predicted labels and the actual labels are used to compute metrics and to visualize classification results.

Many ML algorithms are sensitive to *hyperparameter* settings. These are the parameters that are not learned during training but initially have to be set. In order to tune those hyperparameters, a grid search is initially performed for every combination of algorithms. During the grid search procedures, classifiers are initialized with different combinations of hyperparameters. For every combination, a model is trained and tested using the whole dataset with 5-fold cross-validation. The hyperparameters with which the model achieves the best classification results are saved and used to train the actual model with the test set in the respective test pipeline. All values have been picked out of a range of the most common hyperparameter values for the respective classifiers. The actual values chosen for grid search can be seen in the Python code in B.3.

5.2. Technical Implementation

All experiments are implemented in Python 3. Python in general is a popular choice for machine learning purposes, besides the other fairly common language R. It offers a broad range of ML and NLP libraries. Many of those frameworks come with a high level of abstraction and allow a simple and straightforward usage. Apart from that, the decision for Python is based on personal preference and knowledge, which is why it is preferred over R.

For most algorithms in this thesis the *scikit-learn* library [27] is used. Many *scikit-learn* implementations of algorithms offer similar interfaces. This is especially useful for usage in pipelines. Furthermore, the *Natural Language Toolkit (NLTK)* [4] is used for NLP specific functionality. *NLTK* is currently one of the most popular choices for natural language processing and comes with extensive lexical resources, which is why it is used for tokenizing, stop-word removal and sentiment analysis. The oversampling algorithms are provided by the *imbalanced-learn* toolkit [22]. This is chosen because the provided interface is very similar to *scikit-learn* and therefore eases the usage. For the Paragraph Vector algorithm the *Gensim* framework [28] is utilized, since it is currently the most elaborate Python solution for this purpose. In addition to that, *scipy* [18] and *pandas* [24] are used for storing and handling the data, because both offer very useful data structures for managing big datasets. The classification results are plotted and visualized with the help of the *matplotlib* [16] and *seaborn* [35] libraries.

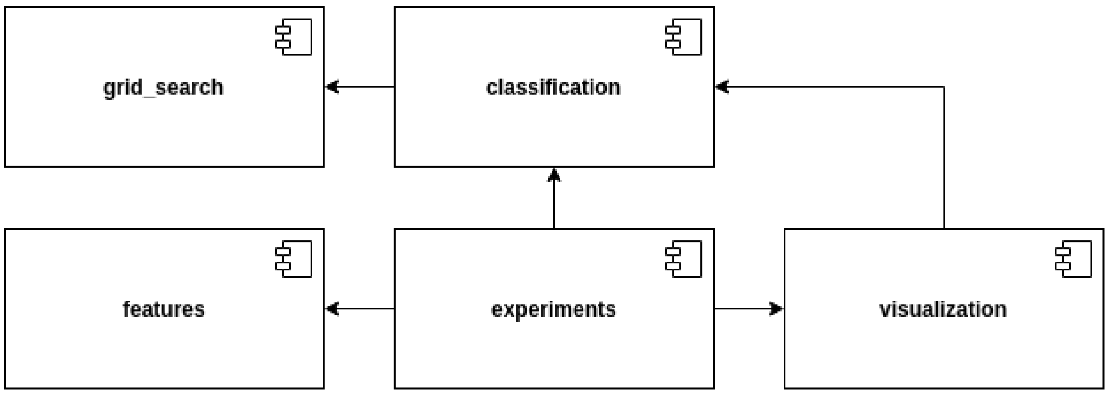


Figure 5.2.: Test system architecture.

The architecture of the test system can be seen in Figure 5.2. Every component is established through a single Python script. The *experiments* component (see B.5) constitutes the main entrypoint. In the *main()* function, the dataset is first read from a file. After that the dataset is passed to the feature generation endpoint of the *features* component (see B.2). The resulting featuresets are then passed to the train and test function of the *classification* component (see B.4). The function also receives a flag which indicates whether a grid search should be performed before the actual training and testing. The search is implemented in the *grid_search* component (see B.3). The results of the grid search (i.e. the best performing classifier hyperparameters) are stored in JSON files in a dedicated directory. For the following training process, all classifiers except for the Naive Bayes classifier are initialized with the hyperparameters from the grid search files. Naive Bayes does not require a grid search, since there are no hyperparameters to optimize.

After the training and testing, results are written to CSV and JSON files. Every combination results in one CSV file and one JSON file each. The former ones contain both predicted and true labels of the test sets. The latter ones contain certain metrics, which are further described in Section 6.1. Finally, the endpoint provided by the *visualization* component (see B.5) reads the results from the files and creates graphics and a table, which are saved to separate files as well. Apart from the aforementioned components, *utils* and *constants* are utilized by other components (see B.7 and B.6). Feature generation as well as grid search and classification are parallelized, so that single jobs are distributed among cores. The Python library *joblib* is used for this purpose.

6. Results

In this chapter the outcome of all experiments are presented. First, the statistical measures used as evaluation metrics are introduced. After that, an overview on method performances is given.

6.1. Metrics

To measure the classification success, a set of statistical metrics is used. For their computation an amount of basic measures has to be taken into account. All measures are calculated per class, so that the present three-class classification problem is treated as three binary classification problems with *positive* and *negative* samples. For every class, *true positives* (*TP*), *false positives* (*FP*), *true negatives* (*TN*) and *false negatives* (*FN*) are counted. TPs are positive samples that have been classified as such, whereas FPs are negative samples that erroneously have been classified positive. Correspondingly TNs are negative samples that have been classified correctly, and FNs are positive samples that have been labeled negative. TPs, TNs, FPs and FNs are visualized through a *confusion matrix*. All values are normalized by the respective class support.

Other metrics computed from those numbers are *precision*, *recall* and *F1-score*, which are accumulated in a table. The precision gives an insight of how exact a model can predict class membership. Therefore it measures the quality of the classification results. It is defined by the number of TPs divided by the sum of TPs and FPs. The recall however indicates how complete the model can predict class membership and therefore covers the quantitative aspect of classification success. It is defined by the number of TPs divided by the sum of TPs and FNs. The F1-score is the harmonic mean of precision and recall and can be computed by the following formula:

$$F1 = \frac{2 \times precision \times recall}{precision + recall}. \tag{6.1}$$

6.2. Method Performances

In order to compare the results, weighted averaged precision values (P), recall values (R) and F1-scores (F1) are depicted in radar charts in Figures 6.2 through 6.4. The values are taken from Table A in Appendix A. A more detailed overview on method performances can be seen there. In addition to that, confusion matrices for all experiments can as well be seen in Appendix A.

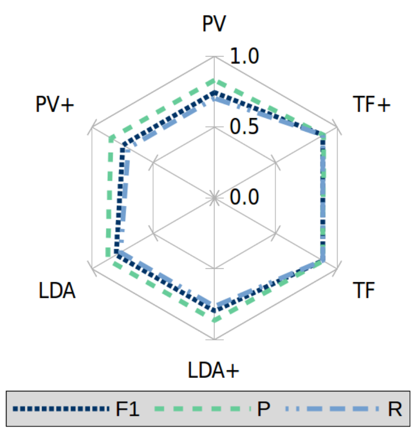
Looking at the results, there is no clear 'winner' combination. Nevertheless, some insights can be drawn from the conducted experiments. Overall, Logistic Regression and the Multilayer Perceptron are the best performing classifiers. The latter one however results in a higher recall than the first one. Especially in hate speech detection applications in social media, a high recall is preferable over a high precision. If the recall is high and the precision is in turn low, further measures can be taken to extract the actual hate speech samples from the samples classified as hate speech. This is not the case, if the recall is low and very few hate speech samples are recognized.

There is no significant performance difference between the two oversampling algorithms SMOTE and ADASYN. The main difference between SMOTE and ADASYN is the number of synthetic samples generated for every existing minority sample. This might not be as important for classification success.

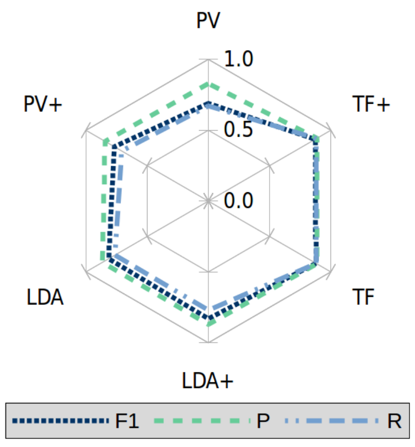
In contrast to that, the feature generation methods have a comparably high influence on the overall classification performance. Combinations that use TF-IDF achieve the best results. This goes along with TF-IDF's popularity in text classification. The worst performing feature generation method is Paragraph Vector. Against the expectations, adding sentiment features does not significantly increase the performance.

In most cases, the precision is higher than the recall. This shows, that despite oversampling it is difficult to recognize actual hate speech, so that many hate speech samples are misclassified. A lot of hate speech samples are labeled offensive and not hateful, which can be seen in most of the confusion matrices. This underlines the fact that the distinction between the hate and the offensive class is particularly difficult.

In general, the experiments show that hate speech is much harder to classify than just offensive language. The average F1-score for the hate speech class is only 0.281, whereas the average F1-scores for the offensive and the neither class attain values as high as 0.832 and 0.703.

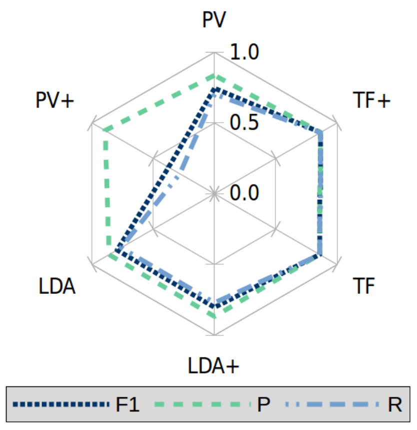


(a) SMOTE

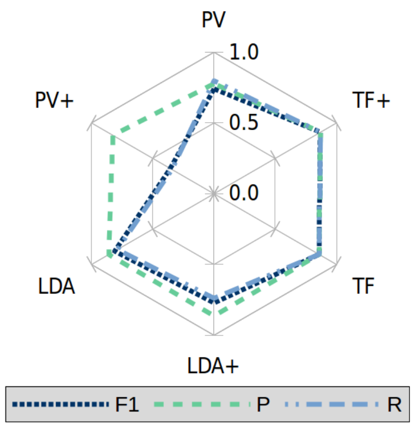


(b) ADASYN

Figure 6.1.: Logistic Regression Metrics

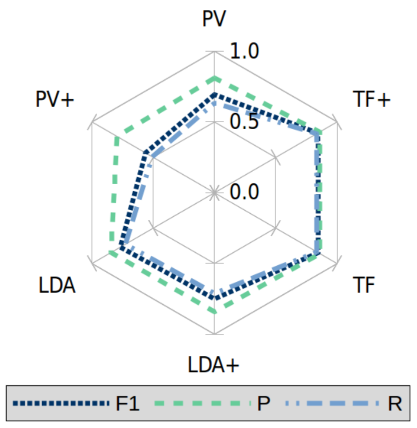


(a) SMOTE

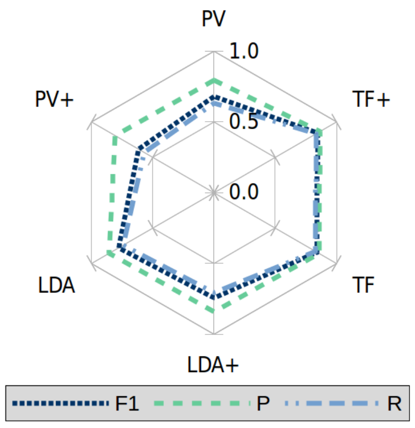


(b) ADASYN

Figure 6.2.: Support Vector Machine Metrics

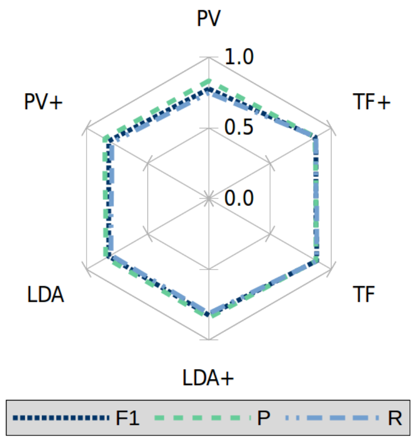


(a) SMOTE

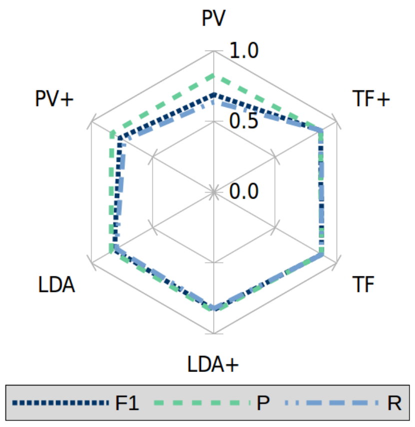


(b) ADASYN

Figure 6.3.: Naive Bayes Classifier Metrics



(a) SMOTE



(b) ADASYN

Figure 6.4.: Multilayer Perceptron Metrics

7. Conclusion

In this chapter, the results are discussed and put into context with the actual goal of this thesis. Also, an overview on the achievements of this work are given. Finally an outlook on further research regarding this topic is given.

7.1. Evaluation

In this thesis the automated detection of hate speech through the use of machine learning has been investigated. Therefore different machine learning and natural language processing methods have been combined and tested on a social media dataset. For every combination of algorithms, a test pipeline has been used in order to collect classification results.

The results of the conducted experiments have shown that detecting hate speech in social media is in general very difficult. Most experiments have provided a relatively low recall concerning the hate speech class. Furthermore it has been shown that especially the distinction between hate speech and other offensive language is extraordinarily hard.

Different algorithm combination performances have indicated that the choice of the feature generation method affects classification success the most. The choice of the classification algorithm also has a significant influence. The oversampling algorithm choice however has not had such a big impact. Also the use of sentiment score features has not been substantially beneficial for the classification results. Overall the combination of TF-IDF, SMOTE and Logistic Regression has produced the best results.

7.2. Outlook

It is important to note that the obtained results have to be interpreted with respect to the dataset that has been used. In order to prove the feasibility of the examined methods, experiments with other similar datasets should be conducted.

To increase classification performance, feature generation algorithms could be combined in order to increase the predictive quality of the featuresets. Furthermore, a feature selection algorithm could be used to decrease the featurespace dimensionality and seek out features that provide the most useful information.

Deep learning approaches have not been covered in the present thesis. Since this area of research has justifiably gained popularity not only in text classification, further investigation could be very promising.

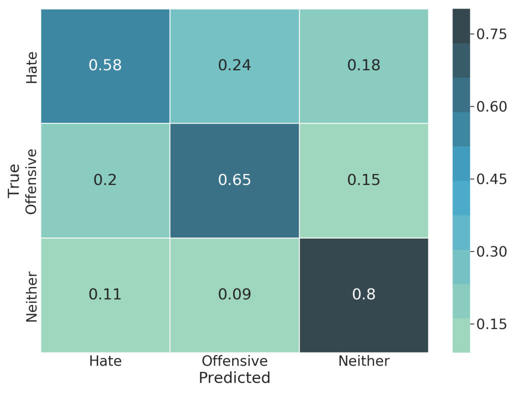
All in all, further research should focus on completeness rather than correctness of classification results, which is indicated through a higher recall than precision. The detection of hate speech with the help of machine learning could then serve as a first step with the option of further processing and filtering.

A. Results Metrics

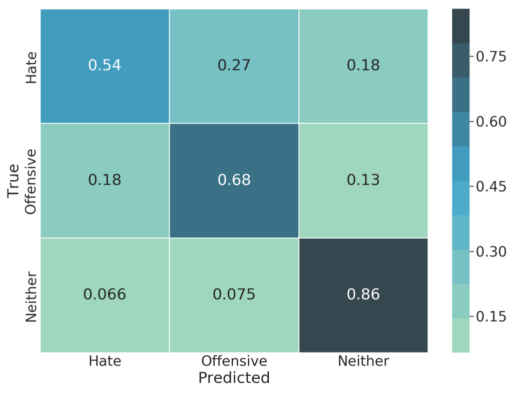
			Hate			Offensive			Neither			Weighted AVG		
			F1	P	R	F1	P	R	F1	P	R	F1	P	R
Logistic Regression	ADASYN	PV	0.256	0.164	0.585	0.771	0.946	0.650	0.627	0.515	0.802	0.717	0.829	0.671
		PV+	0.248	0.161	0.542	0.796	0.951	0.685	0.680	0.563	0.859	0.747	0.843	0.707
		LDA	0.303	0.208	0.558	0.850	0.970	0.757	0.713	0.608	0.863	0.795	0.865	0.762
		LDA+	0.314	0.213	0.594	0.850	0.970	0.757	0.741	0.634	0.892	0.801	0.871	0.770
		TF	0.372	0.356	0.390	0.930	0.941	0.920	0.849	0.823	0.876	0.883	0.885	0.881
		TF+	0.390	0.382	0.398	0.930	0.940	0.921	0.838	0.807	0.872	0.885	0.886	0.883
	SMOTE	PV	0.270	0.175	0.594	0.797	0.937	0.693	0.676	0.592	0.788	0.744	0.832	0.703
		PV+	0.248	0.159	0.565	0.797	0.951	0.686	0.694	0.594	0.834	0.748	0.845	0.704
		LDA	0.301	0.200	0.606	0.854	0.968	0.764	0.736	0.655	0.839	0.803	0.872	0.767
		LDA+	0.309	0.210	0.581	0.852	0.964	0.764	0.719	0.629	0.839	0.798	0.864	0.765
		TF	0.376	0.363	0.390	0.930	0.940	0.919	0.854	0.823	0.887	0.884	0.887	0.883
		TF+	0.392	0.392	0.392	**0.932**	0.939	0.924	**0.866**	**0.837**	0.896	**0.888**	0.888	**0.887**
Multilayer Perceptron	ADASYN	PV	0.229	0.144	0.561	0.735	0.951	0.599	0.646	0.520	0.852	0.689	0.829	0.640
		PV+	0.259	0.177	0.485	0.824	0.940	0.734	0.689	0.596	0.817	0.767	0.835	0.734
		LDA	0.240	0.190	0.328	0.880	0.927	0.837	0.702	0.642	0.775	0.814	0.838	0.798
		LDA+	0.239	0.209	0.278	0.896	0.927	0.867	0.731	0.676	0.797	0.830	0.843	0.821
		TF	0.332	0.355	0.312	0.931	0.930	0.932	0.818	0.807	0.830	0.879	0.877	0.880
		TF+	0.340	0.357	0.324	0.925	0.921	0.929	0.807	0.811	0.802	0.871	0.870	0.873
	SMOTE	PV	0.281	0.202	0.458	0.835	0.938	0.752	0.683	0.576	0.840	0.776	0.833	0.749
		PV+	0.273	0.190	0.481	0.878	0.922	0.838	0.714	0.747	0.684	0.816	0.851	0.792
		LDA	0.253	0.199	0.347	0.883	0.928	0.842	0.706	0.657	0.763	0.816	0.839	0.800
		LDA+	0.194	0.165	0.236	0.888	0.930	0.850	0.733	0.657	0.829	0.824	0.842	0.812
		TF	0.372	0.405	0.343	0.929	0.924	0.934	0.828	0.827	0.829	0.879	0.877	0.882
		TF+	0.353	0.393	0.321	0.925	0.928	0.923	0.805	0.770	0.843	0.870	0.869	0.873
Naive Bayes	ADASYN	PV	0.207	0.135	0.444	0.742	0.922	0.621	0.573	0.461	0.755	0.680	0.795	0.633
		PV+	0.205	0.133	0.443	0.667	0.954	0.512	0.523	0.376	0.863	0.616	0.807	0.570
		LDA	0.264	0.177	0.523	0.835	0.962	0.738	0.689	0.585	0.839	0.779	0.856	0.742
		LDA+	0.255	0.173	0.490	0.800	0.961	0.685	0.650	0.517	0.877	0.744	0.841	0.707
		TF	0.358	0.282	0.489	0.892	0.929	0.858	0.770	0.734	0.810	0.841	0.859	0.829
		TF+	0.362	0.292	0.477	0.896	0.940	0.856	0.782	0.719	0.857	0.845	0.865	0.834
	SMOTE	PV	0.181	0.112	0.476	0.752	0.923	0.635	0.600	0.530	0.691	0.693	0.811	0.634
		PV+	0.182	0.111	0.501	0.603	0.940	0.444	0.511	0.373	0.811	0.563	0.796	0.509
		LDA	0.221	0.145	0.470	0.829	0.949	0.737	0.674	0.599	0.769	0.769	0.845	0.727
		LDA+	0.283	0.188	0.567	0.808	0.958	0.698	0.656	0.544	0.825	0.750	0.841	0.712
		TF	0.388	0.315	0.507	0.899	0.932	0.867	0.766	0.728	0.807	0.847	0.862	0.836
		TF+	**0.423**	0.345	0.549	0.897	0.930	0.867	0.749	0.715	0.787	0.844	0.858	0.834

Support Vector Machine	ADASYN	PV	0.065	**0.419**	0.035	0.884	0.801	**0.987**	0.323	0.798	0.202	0.741	0.777	0.799
		PV+	0.155	0.087	0.714	0.329	0.974	0.198	0.549	0.399	0.880	0.357	0.826	0.343
		LDA	0.277	0.205	0.424	0.866	0.961	0.789	0.721	0.602	0.897	0.811	0.860	0.789
		LDA+	0.288	0.196	0.546	0.821	**0.976**	0.709	0.699	0.557	**0.940**	0.772	0.863	0.739
		TF	0.282	0.278	0.285	0.916	0.914	0.917	0.781	0.791	0.771	0.857	0.858	0.857
		TF+	0.344	0.340	0.349	0.923	0.919	0.927	0.800	0.819	0.781	0.870	0.870	0.870
	SMOTE	PV	0.266	0.172	0.585	0.799	0.944	0.692	0.657	0.558	0.799	0.745	0.837	0.704
		PV+	0.127	0.068	**0.919**	0.394	0.974	0.247	0.369	0.796	0.240	0.375	**0.893**	0.283
		LDA	0.275	0.187	0.519	0.850	0.962	0.761	0.718	0.619	0.853	0.796	0.861	0.763
		LDA+	0.293	0.199	0.554	0.856	0.966	0.768	0.728	0.634	0.856	0.802	0.867	0.770
		TF	0.308	0.320	0.296	0.919	0.911	0.926	0.783	0.803	0.764	0.859	0.857	0.861
		TF+	0.332	0.330	0.333	0.921	0.921	0.921	0.805	0.807	0.803	0.866	0.866	0.866

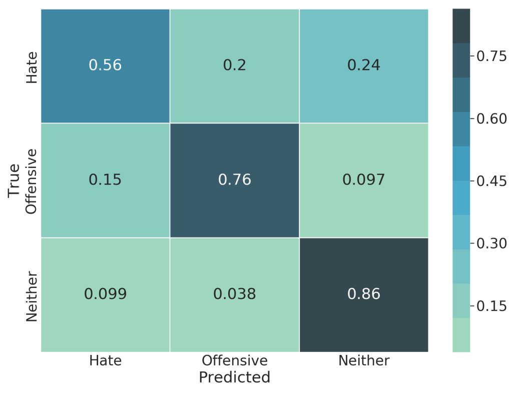
Table A.1.: Table of classification results metrics.



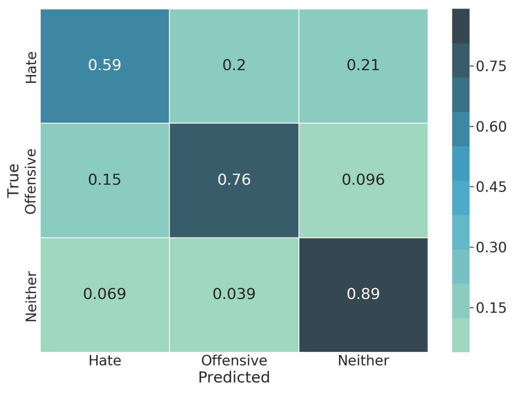
(a) Paragraph Vector



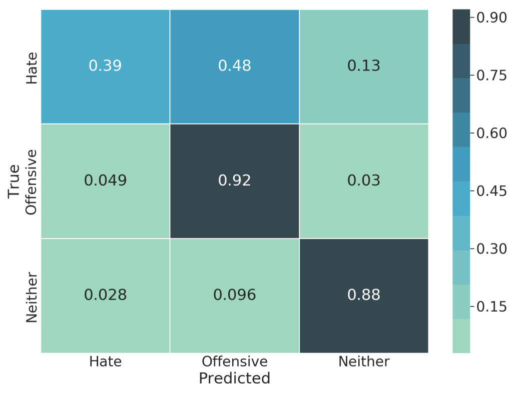
(b) Paragraph Vector with Sentiment Features



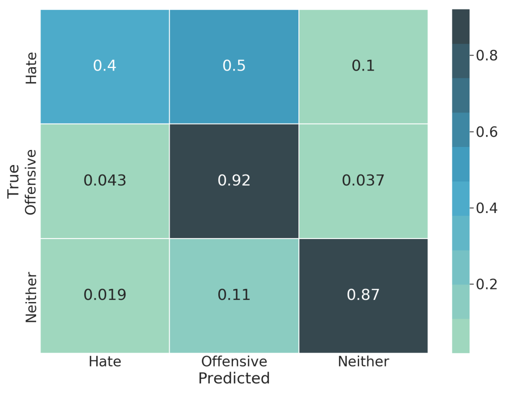
(c) LDA



(d) LDA with Sentiment Features

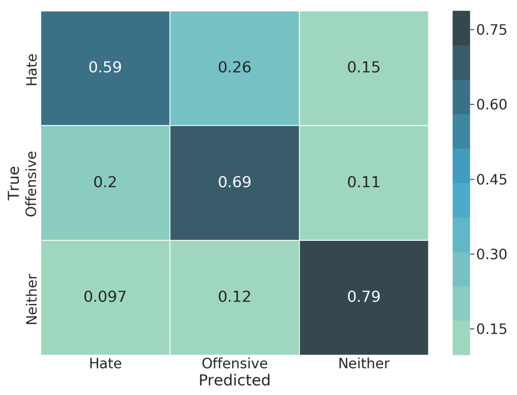


(e) TF-IDF

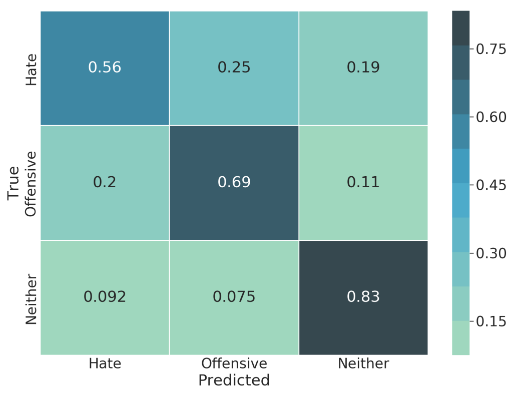


(f) TF-IDF with Sentiment Features

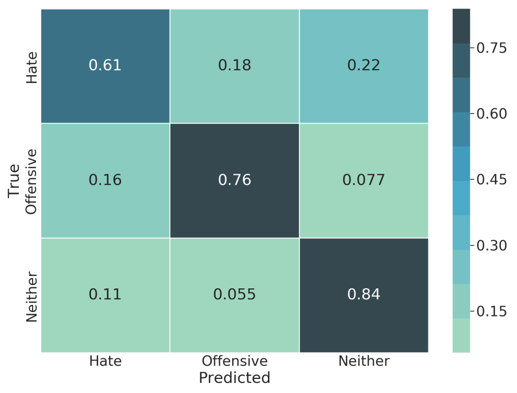
Figure A.1.: Confusion Matrices of Logistic Regression with ADASYN



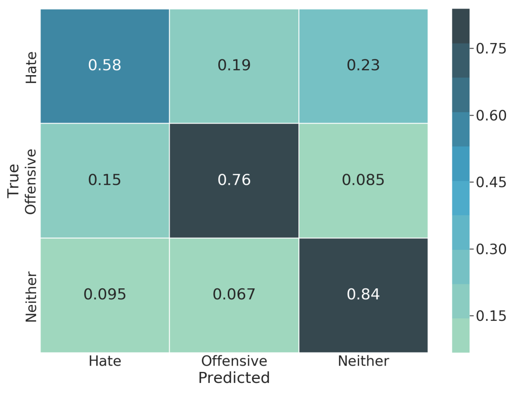
(a) Paragraph Vector



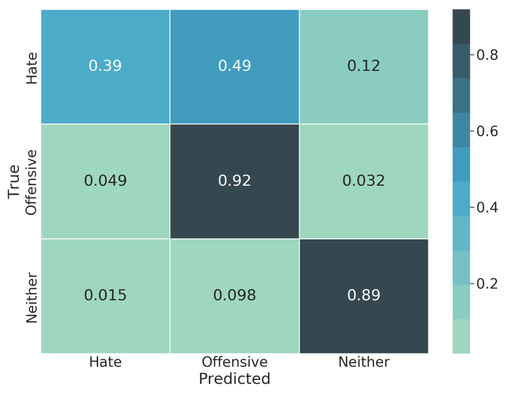
(b) Paragraph Vector with Sentiment Features



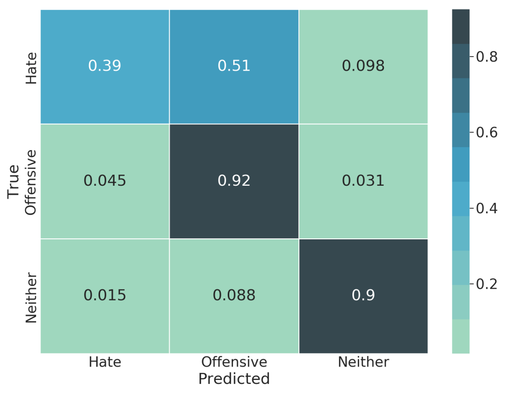
(c) LDA



(d) LDA with Sentiment Features

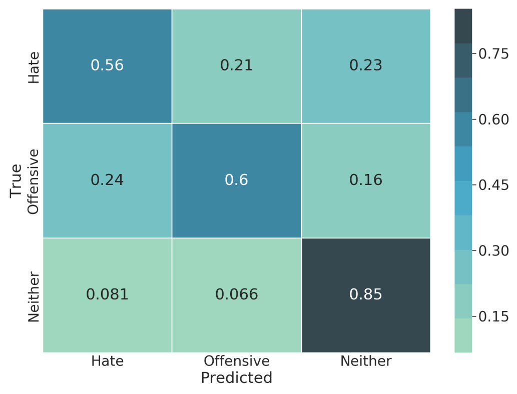


(e) TF-IDF

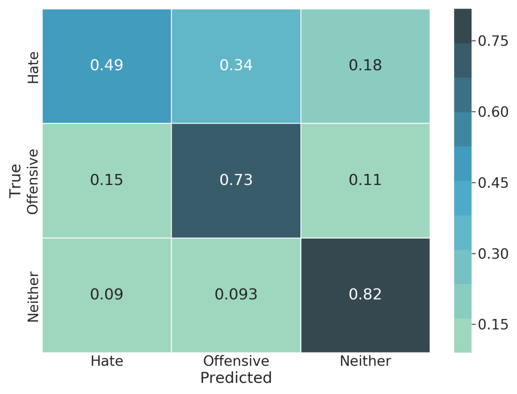


(f) TF-IDF with Sentiment Features

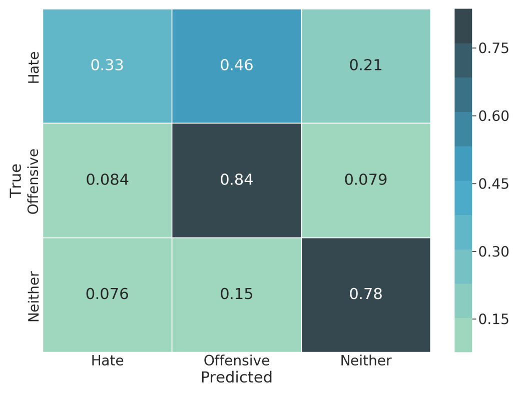
Figure A.2.: Confusion Matrices of Logistic Regression with SMOTE



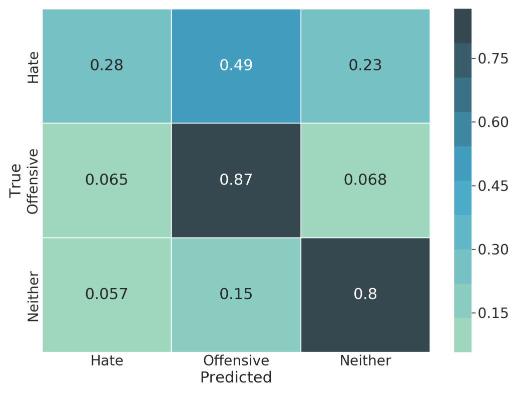
(a) Paragraph Vector



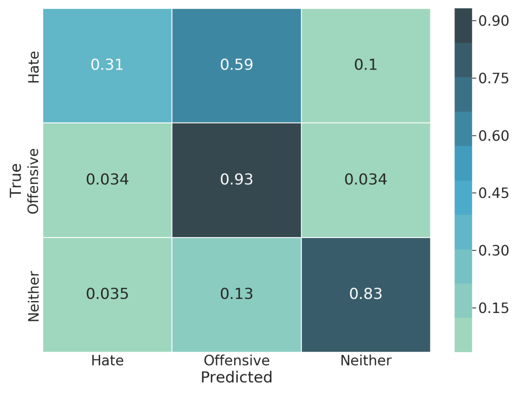
(b) Paragraph Vector with Sentiment Features



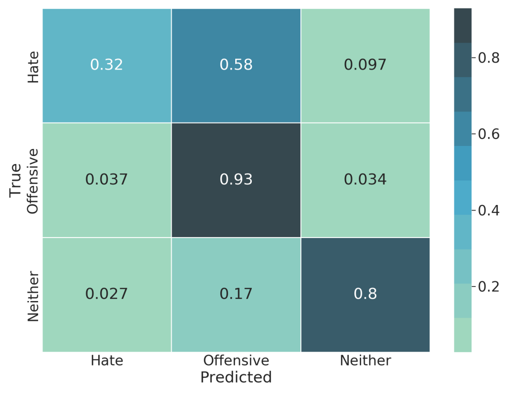
(c) LDA



(d) LDA with Sentiment Features

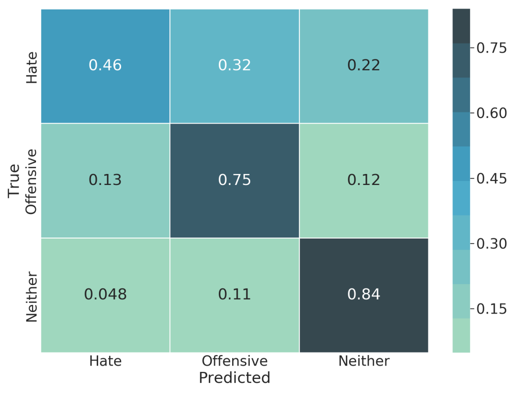


(e) TF-IDF

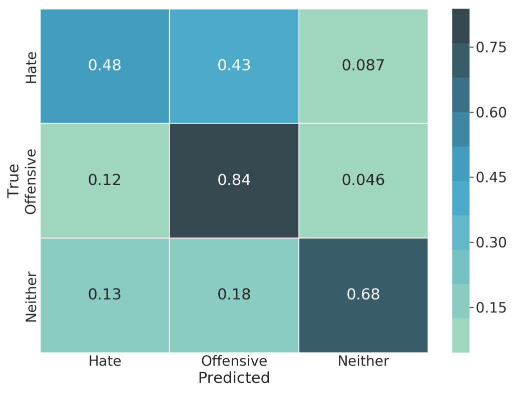


(f) TF-IDF with Sentiment Features

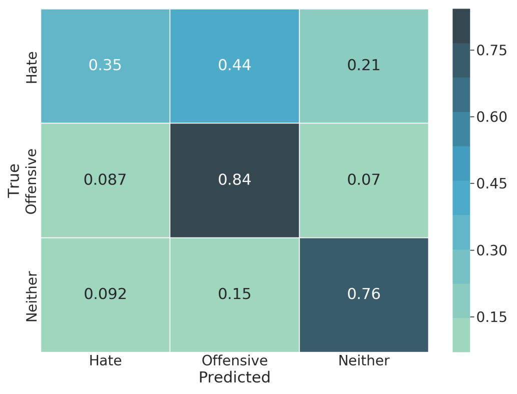
Figure A.3.: Confusion Matrices of Multilayer Perceptron with ADASYN



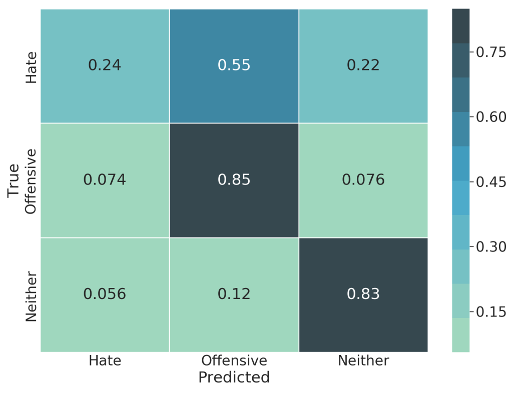
(a) Paragraph Vector



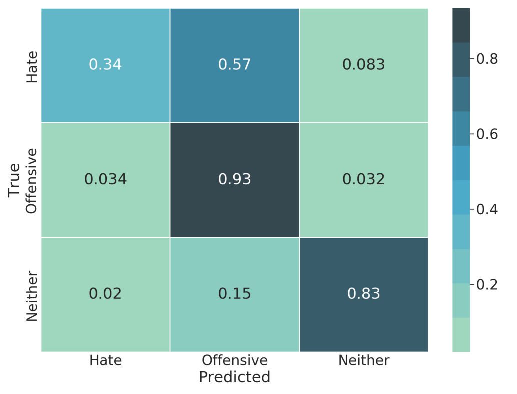
(b) Paragraph Vector with Sentiment Features



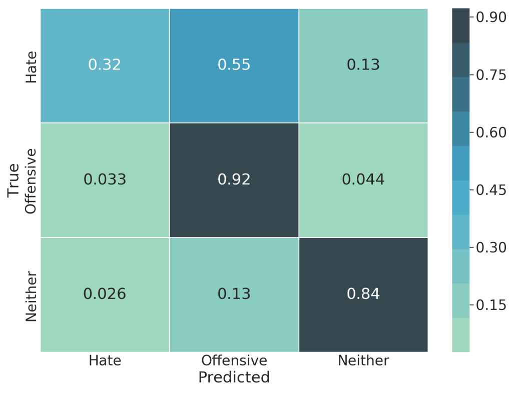
(c) LDA



(d) LDA with Sentiment Features

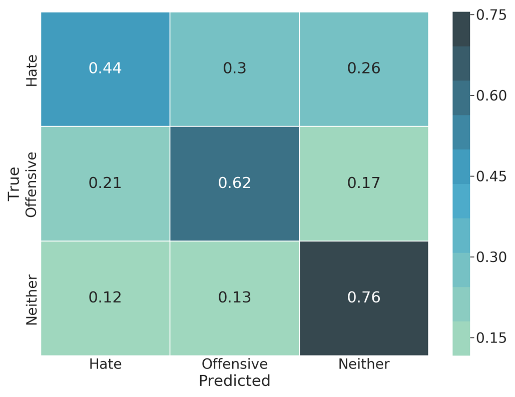


(e) TF-IDF

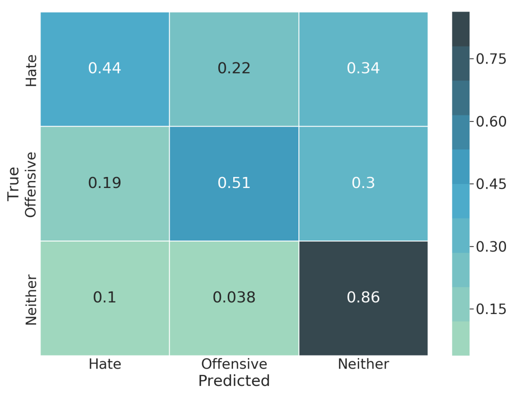


(f) TF-IDF with Sentiment Features

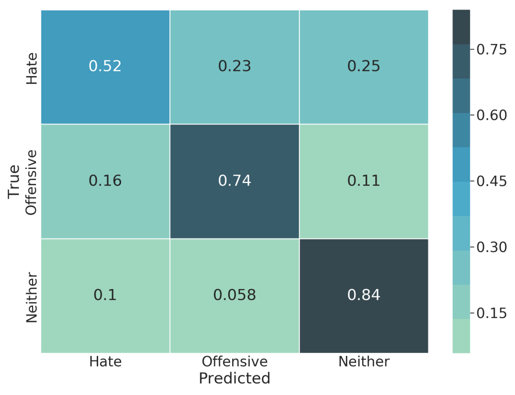
Figure A.4.: Confusion Matrices of Multilayer Perceptron with SMOTE



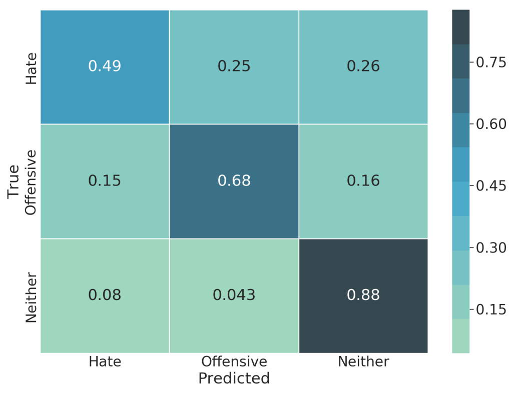
(a) Paragraph Vector



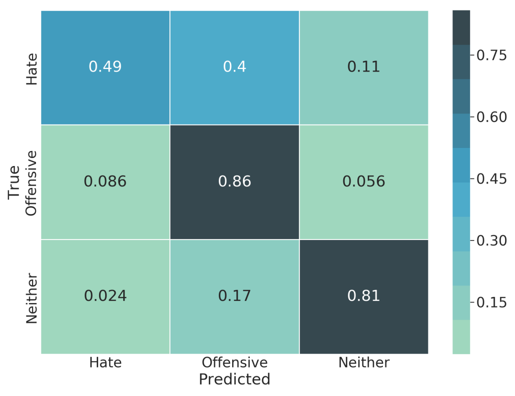
(b) Paragraph Vector with Sentiment Features



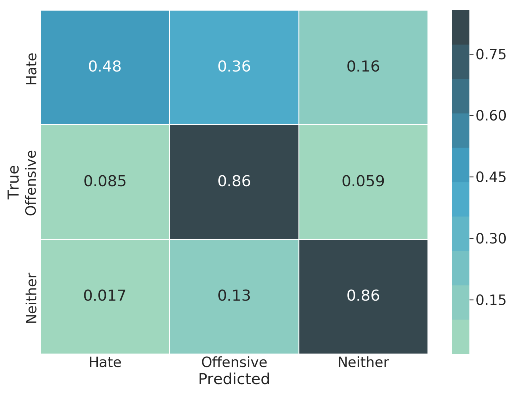
(c) LDA



(d) LDA with Sentiment Features

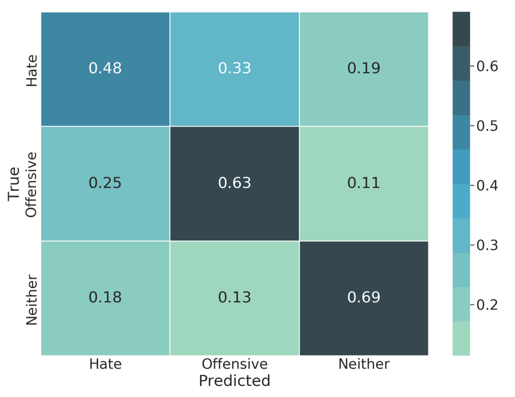


(e) TF-IDF

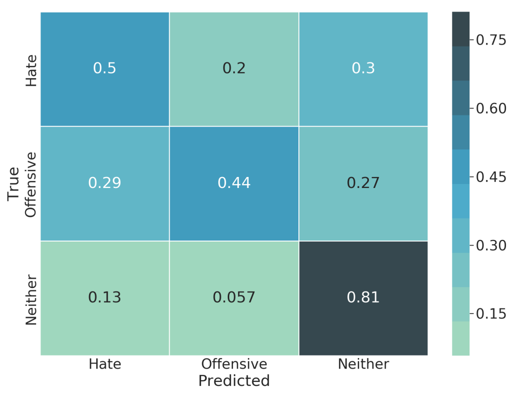


(f) TF-IDF with Sentiment Features

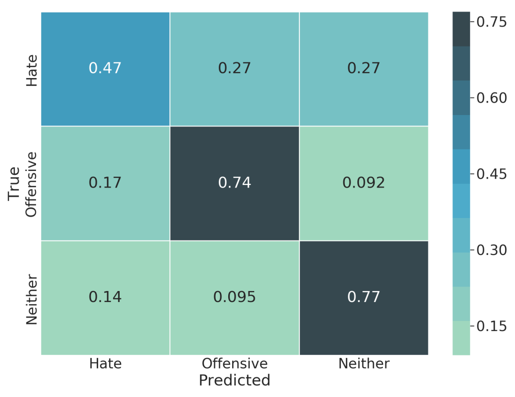
Figure A.5.: Confusion Matrices of Naive Bayes Classifier with ADASYN



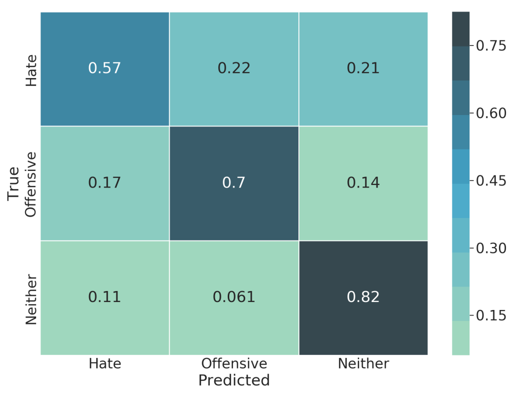
(a) Paragraph Vector



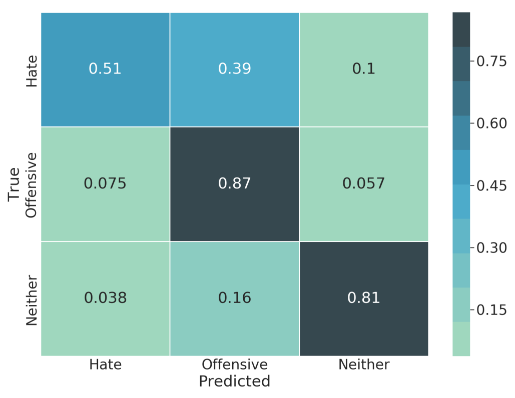
(b) Paragraph Vector with Sentiment Features



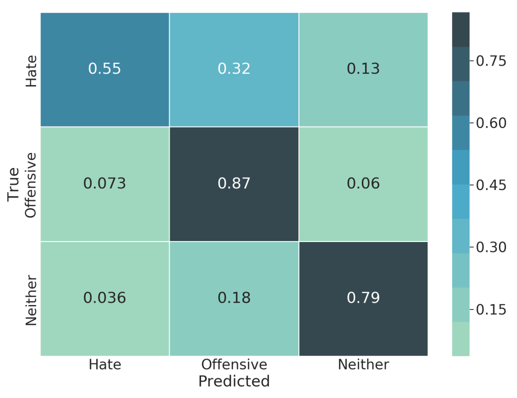
(c) LDA



(d) LDA with Sentiment Features

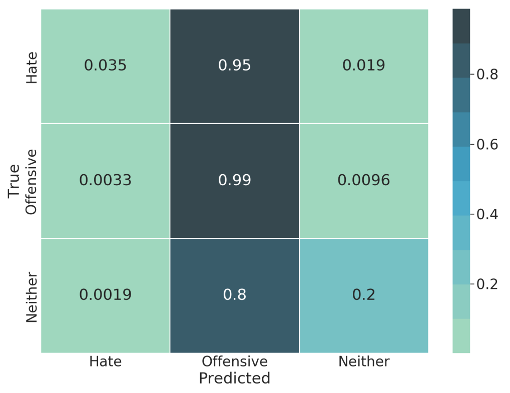


(e) TF-IDF

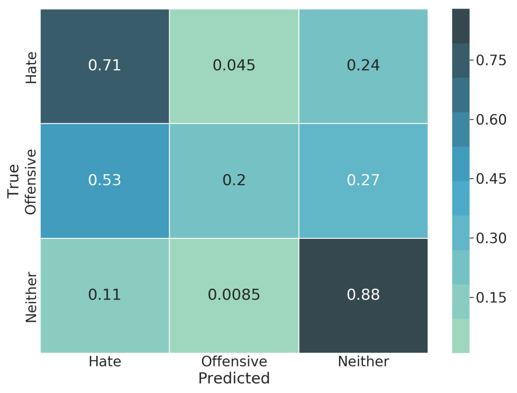


(f) TF-IDF with Sentiment Features

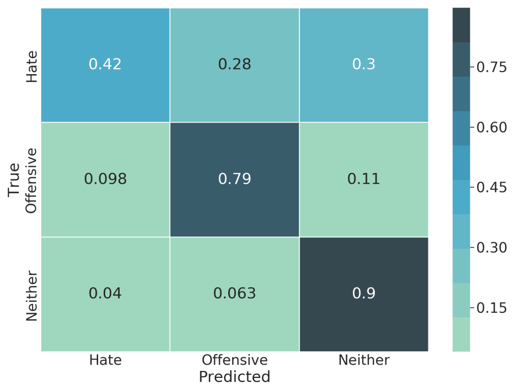
Figure A.6.: Confusion Matrices of Naive Bayes Classifier with SMOTE



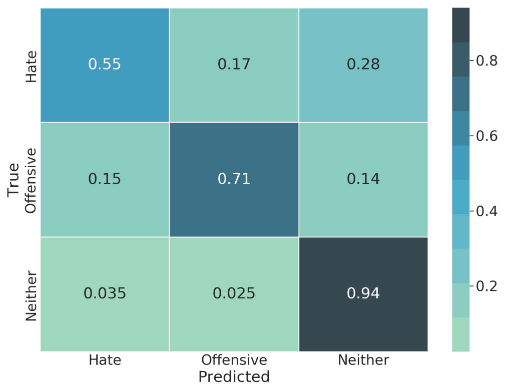
(a) Paragraph Vector



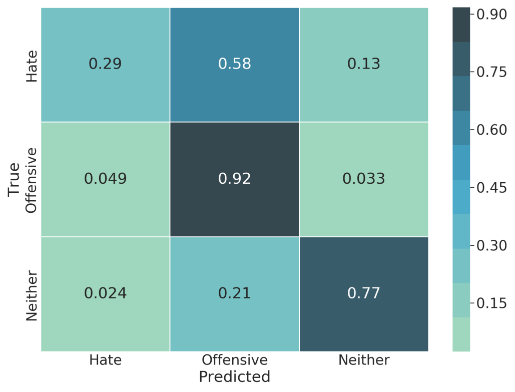
(b) Paragraph Vector with Sentiment Features



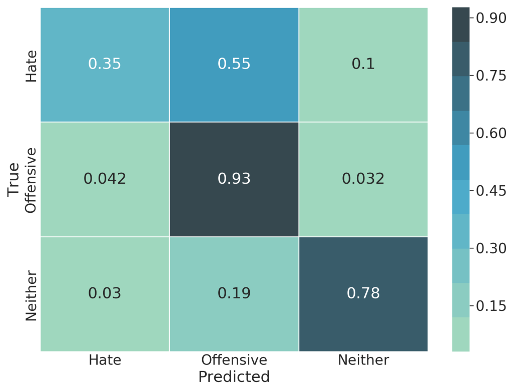
(c) LDA



(d) LDA with Sentiment Features

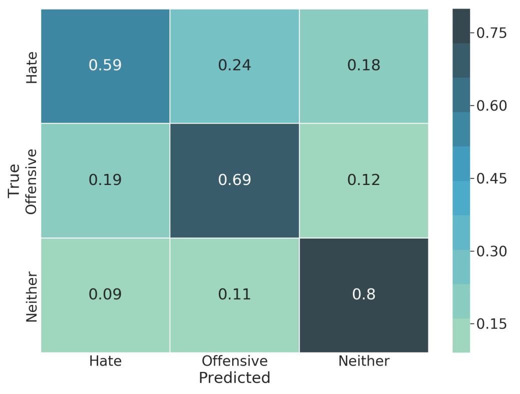


(e) TF-IDF

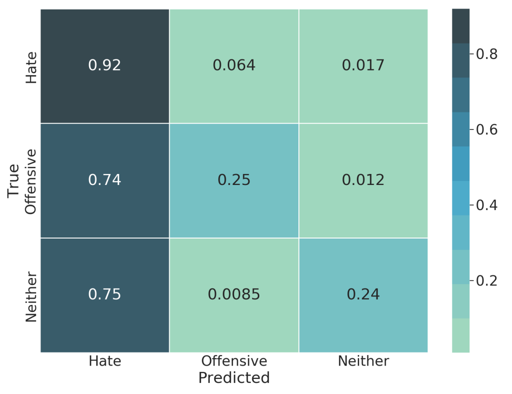


(f) TF-IDF with Sentiment Features

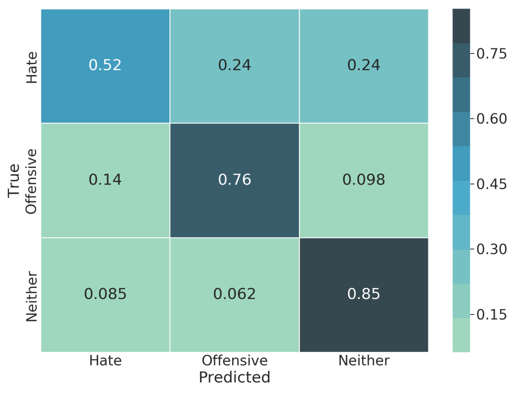
Figure A.7.: Confusion Matrices of Support Vector Machine with ADASYN



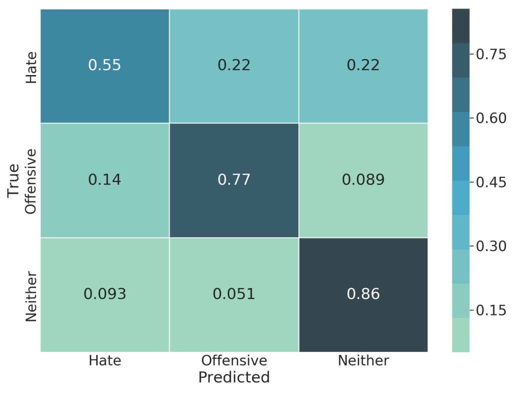
(a) Paragraph Vector



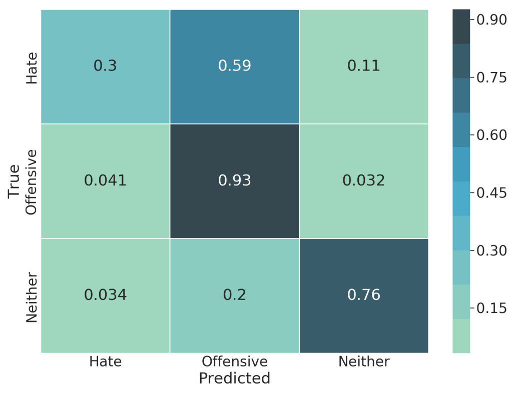
(b) Paragraph Vector with Sentiment Features



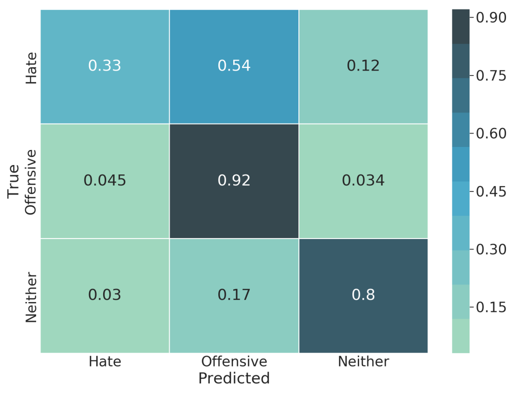
(c) LDA



(d) LDA with Sentiment Features



(e) TF-IDF



(f) TF-IDF with Sentiment Features

Figure A.8.: Confusion Matrices of Support Vector Machine with SMOTE

B. Source Code

B.1. experiments.py

```python
import argparse
import logging

import pandas as pd

import classification
from features import FeatureGenerator
import visualization

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger('experiments')


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-gs', '--gridsearch', help="Perform grid search.",
                        action='store_true')
    args = parser.parse_args()

    gs = " with grid search" if args.gridsearch else ""
    logger.info("Starting experiments" + gs)

    logger.info("Reading data from input file")
    data = pd.read_csv('input/labeled_data.csv')
    text = data['tweet']
    labels = data['class']

    featuresets = FeatureGenerator().generate_features(text)
    classification.train_test(args.gridsearch, featuresets, labels)
    visualization.plot(featuresets)


if __name__ == '__main__':
    main()
```

B.2. features.py

```python
import logging

from gensim.models.doc2vec import Doc2Vec
from gensim.models.doc2vec import TaggedDocument
from joblib import delayed
from joblib import Parallel
from nltk.corpus import stopwords
from nltk.sentiment import SentimentIntensityAnalyzer
from nltk import TweetTokenizer
from scipy.sparse import coo_matrix
from scipy.sparse import hstack
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import minmax_scale

from constants import NUM_CORES

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger('experiments')

stop_words = set(stopwords.words('english'))

tok = TweetTokenizer(
    preserve_case=False,
    reduce_len=True,
    strip_handles=True,
)
for key in logging.Logger.manager.loggerDict:
    if key != 'experiments':
        logging.getLogger(key).setLevel(logging.WARNING)


class TfIdfFeatures:
    name = 'tfidf'

    def generate(self, text):
        word_vec = TfidfVectorizer(
            decode_error='ignore',
            strip_accents='unicode',
            tokenizer=tok.tokenize,
            stop_words=stop_words,
            ngram_range=(1, 3),
            max_features=5000,
            sublinear_tf=True,
        )
        char_vec = TfidfVectorizer(
            decode_error='ignore',
            strip_accents='unicode',
            tokenizer=tok.tokenize,
            analyzer='char',
            stop_words=stop_words,
            ngram_range=(2, 6),
            max_features=10000,
            sublinear_tf=True,
        )
        word_features = word_vec.fit_transform(text)
        char_features = char_vec.fit_transform(text)
        all_features = minmax_scale(
            hstack([word_features, char_features]).toarray()
        )
        return coo_matrix(all_features)


class LDAFeatures:
    name = 'lda'

    def generate(self, text):
        word_vec = CountVectorizer(
            decode_error='ignore',
            strip_accents='unicode',
            tokenizer=tok.tokenize,
            stop_words=stop_words,
            ngram_range=(1, 3),
            max_features=5000,
        )
        char_vec = CountVectorizer(
            decode_error='ignore',
            strip_accents='unicode',
            tokenizer=tok.tokenize,
            analyzer='char',
            stop_words=stop_words,
            ngram_range=(2, 6),
            max_features=10000,
        )
        lda = LatentDirichletAllocation(
            n_components=50,
            max_iter=20,
        )
        word_counts = word_vec.fit_transform(text)
        char_counts = char_vec.fit_transform(text)
        all_counts = hstack([word_counts, char_counts])
        all_features = minmax_scale(lda.fit_transform(all_counts))
        return coo_matrix(all_features)
```

```


class Doc2VecFeatures:
    name = 'doc2vec'

    def _clean(self, tweet):
        tokens = tok.tokenize(tweet)
        return [t for t in tokens if t not in stop_words]

    def generate(self, text):
        doc2vec = Doc2Vec()
        docs = [
            TaggedDocument(self._clean(t), [i]) for i, t in enumerate(text)
        ]
        doc2vec.build_vocab(docs)
        doc2vec.train(
            documents=docs,
            epochs=20,
            total_examples=doc2vec.corpus_count
        )
        all_features = [
            doc2vec.docvecs[i] for i in range(doc2vec.corpus_count)
        ]
        return coo_matrix(minmax_scale(all_features))


class FeatureGenerator:
    featuresets = {}

    def _generate_sent_features(self, text):
        sentiment = SentimentIntensityAnalyzer()
        sent_features = minmax_scale(
            [list(sentiment.polarity_scores(tweet).values()) for tweet in text]
        )
        return sent_features

    def generate_features(self, text):
        logger.info("Starting to generate featuresets")
        sent_features = self._generate_sent_features(text)

        generators = [TfIdfFeatures(), LDAFeatures(), Doc2VecFeatures()]
        Parallel(n_jobs=NUM_CORES, require='sharedmem')(
            delayed(self._generate_features)(gen, text, sent_features)
            for gen in generators)
        logger.info("All featuresets generated")
        return self.featuresets

    def _generate_features(self, gen, text, sent_features):
        logger.info("Generating " + gen.name + " features")
        features = gen.generate(text)
        features_with_sent = hstack((features, sent_features))
        self.featuresets[gen.name] = features
        self.featuresets[gen.name + '+'] = features_with_sent
```

B.3. grid_search.py

```
import json

from sklearn.model_selection import GridSearchCV


log_grid = [
    {
        'C': [1, 10, 100, 1000],
        'solver': ['liblinear', 'saga'],
        'penalty': ['l1', 'l2']
    },
    {
        'C': [1, 10, 100, 1000],
        'solver': ['newton-cg'],
        'penalty': ['l2']
    },
]
svm_grid = [{'C': [1, 10, 100, 1000]}]
mlp_grid = [
    {
        'hidden_layer_sizes': [(50, 50), (100,)],
        'alpha': [1e-3, 1e-4, 1e-5, 1e-6]
    }
]

grids = {'log': log_grid, 'svm': svm_grid, 'mlp': mlp_grid}


def grid_search(clf, clf_name, features, labels, file_name):
    search = GridSearchCV(clf, grids[clf_name], cv=5, scoring='f1_macro')
    search.fit(features, labels)

    with open(file_name, 'w+') as f:
        f.write(json.dumps(search.best_params_))
```

B.4. classification.py

```python
import json
import logging

from imblearn.over_sampling import ADASYN
from imblearn.over_sampling import SMOTE
from joblib import delayed
from joblib import Parallel
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.neural_network import MLPClassifier
from sklearn.svm import LinearSVC

import constants
from constants import Classifiers
from constants import Oversamplers
from grid_search import grid_search
import utils


classifiers = {
    Classifiers.LOGISTIC_REGRESSION: LogisticRegression(),
    Classifiers.SVM: LinearSVC(),
    Classifiers.NAIVE_BAYES: MultinomialNB(),
    Classifiers.MULTILAYER_PERCEPTRON: MLPClassifier(),
}

oversamplers = {
    Oversamplers.ADASYN: ADASYN(),
    Oversamplers.SMOTE: SMOTE(),
}

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger('experiments')


def _train_test(clf_name, clf, perform_grid_search, featuresets, labels):
    for os_name, os in oversamplers.items():
        for gen_name, features in featuresets.items():
            combination = clf_name + os_name + gen_name
            param_file_name = utils.get_param_file_name(combination)
            result_file_name = utils.get_result_file_name(combination)
            pred_file_name = utils.get_prediction_file_name(combination)

            if perform_grid_search and clf_name != Classifiers.NAIVE_BAYES:
                logger.info("Performing grid search for " + gen_name + ", " +
                            os_name + ", " + clf_name)
                grid_search(clf, clf_name, features, labels, param_file_name)

            if clf_name != Classifiers.NAIVE_BAYES:
                with open(param_file_name, 'r') as f:
                    clf.set_params(**json.load(f))

            # split into train and test set
            X_train, X_test, y_train, y_test = train_test_split(features,
                                                                 labels)

            # oversample train set
            X_train_res, y_train_res = os.fit_sample(X_train, y_train)

            # learn model
            logger.info("Training " + clf_name + " classifier")
            clf.fit(X_train_res, y_train_res)

            # predict labels for test set
            y_pred = clf.predict(X_test)

            pred = pd.DataFrame({'y_true': y_test, 'y_pred': y_pred})
            pred.to_csv(pred_file_name)

            results = classification_report(y_true=y_test, y_pred=y_pred,
                                            output_dict=True)

            with open(result_file_name, 'w+') as f:
                f.write(json.dumps(results))


def train_test(perform_grid_search, featuresets, labels):
    logger.info("Starting to train and test classifiers")
    Parallel(n_jobs=constants.NUM_CORES, require='sharedmem')(
        delayed(_train_test)(
            name,
            clf,
            perform_grid_search,
            featuresets,
            labels
        )
        for name, clf in classifiers.items()
    )
    logger.info("Done with training and testing")
```

B.5. visualization.py

```python
import logging

from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.metrics import confusion_matrix

from classification import classifiers
from classification import oversamplers
import utils

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger('experiments')


def plot_confusion_matrix(name, y_true, y_pred):
    adjust_plot_settings()
    cmap = get_colormap()
    class_names = ['Hate', 'Offensive', 'Neither']

    c_mat = confusion_matrix(y_true, y_pred)
    c_mat = c_mat.astype('float') / c_mat.sum(axis=1)[:, np.newaxis]

    plt.figure(figsize=(12, 9))
    heatmap = sns.heatmap(c_mat, annot=True, cmap=cmap, linecolor='white',
                          linewidths=1)

    heatmap.xaxis.set_ticklabels(class_names, ha='center', va='top',
                                 rotation=0)
    heatmap.yaxis.set_ticklabels(class_names, ha='right', va='center',
                                 rotation=90)
    heatmap.tick_params(length=0)

    plt.xlabel('Predicted')
    plt.ylabel('True')

    plt.tight_layout()
    plt.savefig(utils.get_graphics_file_name(name))


def plot(featuresets):
    logger.info("Plotting results")
    results = pd.DataFrame()

    for clf_name in classifiers.keys():
        for os_name in oversamplers.keys():
            for gen_name in featuresets.keys():
                name = clf_name + os_name + gen_name

                with open(utils.get_result_file_name(name)) as f:
                    data = pd.read_json(f)
                data['classifier'] = clf_name
                data['oversampler'] = os_name
                data['feature generator'] = gen_name
                results = pd.concat([results, data], axis=0)

                with open(utils.get_prediction_file_name(name)) as f:
                    pred = pd.read_csv(f)
                plot_confusion_matrix(name, pred['y_true'], pred['y_pred'])

    results = (results
               .drop(['micro avg', 'macro avg'], axis=1)
               .drop('support', axis=0)
               .set_index(['feature generator', 'oversampler', 'classifier'],
                          append=True)
               .unstack(level=0)
               .swaplevel(0, 2)
               .sort_index()
               )

    tbl = results.to_csv()
    with open('results/table.csv', 'w+') as f:
        f.write(tbl)


def get_colormap():
    palette = sns.color_palette("GnBu_d", n_colors=10)
    palette.reverse()
    return ListedColormap(palette)


def adjust_plot_settings():
    sns.set(style='darkgrid')
    plt.rcParams['savefig.dpi'] = 150
    sns.set(font_scale=2)
```

B.6. utils.py

```python
def get_param_file_name(combination):
    return 'params/' + combination + '.json'


def get_result_file_name(combination):
    return 'results/' + combination + '.json'


def get_prediction_file_name(combination):
    return 'results/' + combination + '.csv'


def get_graphics_file_name(combination):
    return 'graphics/' + combination + '.png'
```

B.7. constants.py

```python
import multiprocessing

NUM_CORES = multiprocessing.cpu_count()


class Classifiers:
    LOGISTIC_REGRESSION = 'log'
    SVM = 'svm'
    NAIVE_BAYES = 'nvb'
    MULTILAYER_PERCEPTRON = 'mlp'


class Oversamplers:
    ADASYN = 'ada'
    SMOTE = 'smt'
```

Bibliography

[1] Pinkesh Badjatiya, Shashank Gupta, Manish Gupta, and Vasudeva Varma. Deep learning for hate speech detection in tweets. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 759–760. International World Wide Web Conferences Steering Committee, 2017.

[2] Monika Bickert. Working to keep facebook safe. `https://newsroom.fb.com/news/2018/07/working-to-keep-facebook-safe/`, Jul 2018. [Online; accessed 07-October-2018].

[3] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[4] Steven Bird and Edward Loper. Nltk: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, page 31. Association for Computational Linguistics, 2004.

[5] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[6] Pete Burnap and Matthew L Williams. Cyber hate speech on twitter: An application of machine classification and statistical modeling for policy and decision making. *Policy & Internet*, 7(2):223–242, 2015.

[7] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[8] Nitesh V Chawla, Nathalie Japkowicz, and Aleksander Kotcz. Special issue on learning from imbalanced data sets. *ACM Sigkdd Explorations Newsletter*, 6(1):1–6, 2004.

[9] Gobinda G Chowdhury. Natural language processing. *Annual review of information science and technology*, 37(1):51–89, 2003.

[10] Kate Crawford and Tarleton Gillespie. What is a flag for? social media reporting tools and the vocabulary of complaint. *New Media & Society*, 18(3):410–428, 2016.

[11] Thomas Davidson, Dana Warmsley, Michael Macy, and Ingmar Weber. Automated hate speech detection and the problem of offensive language. In *Proceedings of the 11th International AAAI Conference on Web and Social Media*, ICWSM '17, pages 512–515, 2017.

[12] Nemanja Djuric, Jing Zhou, Robin Morris, Mihajlo Grbovic, Vladan Radosavljevic, and Narayan Bhamidipati. Hate speech detection with comment embeddings. In *Proceedings of the 24th international conference on world wide web*, pages 29–30. ACM, 2015.

[13] Njagi Dennis Gitari, Zhang Zuping, Hanyurwimfura Damien, and Jun Long. A lexicon-based approach for hate speech detection. *International Journal of Multimedia and Ubiquitous Engineering*, 10(4):215–230, 2015.

[14] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[15] Haibo He, Yang Bai, Edwardo A Garcia, and Shutao Li. Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 1322–1328. IEEE, 2008.

[16] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[17] CJ Hutto and Eric Gilbert. Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Eighth International Conference on Weblogs*

and Social Media (ICWSM-14). Available at (20/04/16) http://comp. social. gatech. edu/papers/icwsm14. vader. hutto. pdf, 2014.

[18] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed <today>].

[19] Jason Koebler and Joseph Cox. Here's how facebook is trying to moderate its two billion users. `https://motherboard.vice.com/en_us/article/xwk9zd/how-facebook-content-moderation-works`, Aug 2018. [Online; accessed 07-October-2018].

[20] Irene Kwok and Yuzhou Wang. Locate the hate: Detecting tweets against blacks. In *AAAI*, 2013.

[21] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning*, pages 1188–1196, 2014.

[22] Guillaume Lemaître, Fernando Nogueira, and Christos K Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research*, 18(1):559–563, 2017.

[23] Shervin Malmasi and Marcos Zampieri. Detecting hate speech in social media. In *Proceedings of the International Conference Recent Advances in Natural Language Processing, RANLP 2017*, pages 467–472, 2017.

[24] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[25] Chikashi Nobata, Joel Tetreault, Achint Thomas, Yashar Mehdad, and Yi Chang. Abusive language detection in online user content. In *Proceedings of the 25th international conference on world wide web*, pages 145–153. International World Wide Web Conferences Steering Committee, 2016.

[26] Bo Pang, Lillian Lee, et al. Opinion mining and sentiment analysis. *Foundations and Trends® in Information Retrieval*, 2(1–2):1–135, 2008.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[28] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[29] B. Ross, M. Rist, G. Carbonell, B. Cabrera, N. Kurowsky, and M. Wojatzki. Measuring the Reliability of Hate Speech Annotations: The Case of the European Refugee Crisis. *ArXiv e-prints*, January 2017.

[30] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Pearson Education, 2010.

[31] scikit learn. One hidden layer mlp. [Online; accessed October 28, 2018].

[32] William Warner and Julia Hirschberg. Detecting hate speech on the world wide web. In *Proceedings of the Second Workshop on Language in Social Media*, pages 19–26. Association for Computational Linguistics, 2012.

[33] Zeerak Waseem. Are you a racist or am i seeing things? annotator influence on hate speech detection on twitter. In *Proceedings of the first workshop on NLP and computational social science*, pages 138–142, 2016.

[34] Zeerak Waseem and Dirk Hovy. Hateful symbols or hateful people? predictive features for hate speech detection on twitter. In *Proceedings of the NAACL student research workshop*, pages 88–93, 2016.

[35] Michael Waskom et al. mwaskom/seaborn: v0.8.1 (september 2017), September 2017.

[36] Guang Xiang, Bin Fan, Ling Wang, Jason Hong, and Carolyn Rose. Detecting offensive tweets via topical feature discovery over a large scale twitter corpus. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1980–1984. ACM, 2012.

[37] Ziqi Zhang, David Robinson, and Jonathan Tepper. Detecting hate speech on twitter using a convolution-gru based deep neural network. In *European Semantic Web Conference*, pages 745–760. Springer, 2018.

[38] Haoti Zhong, Hao Li, Anna Cinzia Squicciarini, Sarah Michele Rajtmajer, Christopher Griffin, David J Miller, and Cornelia Caragea. Content-driven detection of cyberbullying on the instagram social network. In *IJCAI*, pages 3952–3958, 2016.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 21. Februar 2019 Carolin Dohmen