Neural Network Models for Hate Speech Classification in Tweets

Acknowledgements

None of this would have been possible without the support of my advisers, mentors, and friends along the way. I would like to extend my sincere gratitude to Professor Stuart Shieber, with whom I would not be pursuing the Mind/Brain/Behavior track or writing a thesis today. His guidance over the past four years has shaped my academic career into what it is today, and his generosity with his time, empathy, and brilliance is truly remarkable. Thank you, Professor Shieber, for supporting me throughout my entire college career.

I would also like to thank Professor Yaron Singer for his support and guidance throughout the thesis writing process. His suggestions have pushed me to go beyond topics I was already comfortable with and pursue further lines of research.

I would like to thank Yoon Kim for his advice at a crucial time in my thesis in helping me overcome major hurdles in model implementation and sharing his insights from his research in natural language processing.

I would like to thank Professor Latanya Sweeney for generously offering to be a thesis reader.

I would like to thank all of my friends and family who have been so supportive throughout this process. This work would not have been complete without the daily encouragements, lessons on how to debug models and my Latex file, and food deliveries. Finally, thank you to my mom and dad for the unconditional love and support. I would not be here without you.

Contents

Abstract ii

Acknowledgements iii

1 Introduction **1**

1.1 Original Contribution 3

1.2 Related Work 3

2 Background **6**

2.1 Definition of Hate Speech 6

2.2 Online Social Networks 8

2.3 Classification Problem Definition 9

2.4 Text Classification Methods 9

2.4.1 Previous Classification Methods 10

2.4.2 Neural Networks 12

2.4.3 Convolutional Neural Networks 16

2.4.4 Recurrent Neural Networks 19

2.4.5 Word Embeddings 22

2.4.6 fastText 26

3 Methodology **28**

3.1 Problem Statement 28

3.2 Classification Methods 29

3.3 Datasets 29
3.3.1 Data Preprocessing 29
3.3.2 User Features 30
3.4 Word Embeddings 31
3.5 Experimental Settings 34

4 Feature Visualizations **35**
4.1 Visualization Methods for Neural Networks 35
4.2 Saliency Maps 35
4.2.1 Theory 36
4.2.2 Implementation 36

5 Experimental Results **41**
5.1 Results and Analysis 41

6 Discussion and Conclusion **47**
6.1 Hate Speech Word Embeddings 47
6.2 User Features 48
6.3 Future Work 49
6.4 Conclusion 50

7 Appendix **51**
7.1 Saliency Maps 51
7.2 Word Embeddings 53
7.3 Significance Tests 53

Chapter 1

Introduction

The role of social networks in influencing public opinion is a contentious issue not only among politicians but concerns the role of social media companies such as Twitter, Facebook, and Reddit in regulating and protecting free speech. In the spring of 2017, German and British parliamentary committees strongly criticized social media companies for failing to take action quickly to curb hate speech, and Germany threatened to fine the social media companies up to 50 million euros per year if they do not act on hateful postings (Thomasson 2017). On the contrary, the United States Supreme Court cites the First Amendment in unanimously agreeing to protect hate speech as free speech (Volokh 2017). The "Unite the Right" rally by self-identified white nationalists in Charlottesville, Virginia from August 11-12th, 2017, re-incited the discussion around hate speech by demonstrating how racist, anti-Semitic, and other noxious messages can lead to violence, class divides, and death. Twitter is one of the first major companies to take a stance on hate speech, launching a "hate speech crackdown" in December of 2017, which led to the suspension of several accounts associated with white nationalism (Neidig 2017).

With companies now cracking down on hate speech and governments banning hate speech from online platforms, proper identification of hate speech is a pressing

concern for all companies and organizations that allow user-generated content on their platforms. Many websites still use manual moderation, which causes potentially harmful content to be seen by the site's users or cause delays in publishing user content. This process requires labor-intensive review by the platform staff and can miss hate speech or infringe on freedom of expression. The high volume of content being generated on online platforms necessitates an accurate and automatic hate speech detection tool.

Building on Collobert et al. (2011)'s work, deep neural networks have been effective in numerous natural language processing tasks, including entity recognition, part-of-speech, and sentiment analysis. Applying these techniques to natural languages has become more common in the recent decade, replacing previous techniques such as linear regression and support vector machines (SVMs) trained on sparse features with high dimensionality. These models often perform better than linear classification models with relatively little training. The success of these techniques relies on recent developments in word embeddings and more advanced deep learning methods (Mikolov et al. 2013; Joulin et al. 2016).

Deep neural networks, specifically convolutional neural networks (CNN), can be applied to the problem of hate speech classification given their success in extracting higher-level features from sentences. Applied to hate speech, CNNs are able to recognize hateful vocabulary at a phrase-level, then classify each sentence as hate speech or not hate speech. Recently, a newer classification technique called fastText that relies solely on word embeddings to form document embeddings has been shown to performance on par with deep neural networks. We also apply fastText text classification techniques to our hate speech classification task.

1.1 Original Contribution

In this paper, we aim to develop a state-of-the-art classification method for detecting hate speech on Twitter, explore how metadata from tweets affects classification accuracy, and develop a method to capture semantics of code words used in hate speech. Specifically, this paper has the following contributions:

1. We build on work by Y. Kim (2014) to develop convolutional neural networks, long short-term memory networks, and fastText models for supervised hate speech classification. We employ additional features found in tweets such as number of likes, retweets, and user information to outperform previous approaches (Badjatiya et al. 2017; Pitsilis et al. 2018). The additional features improve the hate speech classification task by approximately 2%.

2. Borrowing the idea of hidden layer visualizations from image classification, we construct saliency maps for each sequence to qualitatively analyze high-level features that contribute strongly to hate speech classification.

3. We train our own task-specific word embeddings, due to the extensive use of code words by the alt-right community and others that engage in hate speech. These word embeddings improve the hate speech classification task by approximately an additional 2%.

1.2 Related Work

The task of identifying abusive language such as hate speech in online content has been an important research topic for the past twenty years. Seminal work by Spertus (1997) created a decision-tree based classifier named "Smokey" that used 47 hand-designed syntactic and semantic features. "Smokey" performed well on non-inflammatory messages and found 64% of abusive messages. Since then, a number of techniques have

sprung up for classifying inflammatory and abusive text, including rule-based classification methods and pattern finding (Mahmud et al. 2008; Gianfortoni et al. 2011).

Recently, a number of machine learning techniques have been employed for hate speech detection, including Naive Bayes classifiers to detect racism (Kwok and Wang 2013), support vector machines to detect anti-Semitic comments in Yahoo news (Warner and Hirschberg 2012), and topic models combined with lexical features to detect profanity-related offensive content on Twitter (Xiang et al. 2012). Bag-of-words approaches tend to have high recall, but they also lead to high rates of false positives given that the presence of offensive words can lead to misclassification of texts as hate speech. These techniques rely on bootstrapping the training algorithm or semi-supervised labeling of data with a hate speech lexicon, which results in low precision given the lack of distinction between potential hate words appearing in hate speech versus non-hate speech contexts.

Davidson et al. (2017) begins to address this issue through first differentiating hate speech from a broader category of offensive speech by clearly defining hate speech as *language that is used to expresses hatred towards a targeted group or is intended to be derogatory, to humiliate, or to insult the members of the group.* Burnap and Williams (2016) introduced the concept of *othering* language, referring to the idea of "us" versus "them" rhetoric as a feature for hate speech identification. These discussions point to a crucial idea that hate speech is not limited to the presence of words in a fixed lexicon, but rather relies on the context in which it appears. These word-based approaches not only fail to correctly identify hate-speech, but they also infringe on freedom of speech and expression.

Several researchers have focused on extracting additional features from text, including n-gram based, syntactic, and distributional semantic features at the character uni-gram and bi-gram level (Nobata et al. 2016). Nobata employs a supervised learning model, but unsupervised learning methods that exploit the lexical syntac-

tic features of sentences have become increasingly common for detecting hate speech (Warner and Hirschberg 2012).

While unsupervised methods are important to investigate in creating practical applications of text classification models to large datasets, we focus on a supervised text classification method based on neural networks. Waseem (2016) makes available a dataset of annotated tweets that classify tweets into four categories, racism, sexism, neither, or both. We employ this dataset in our work, treating "neither" as "not hate speech," and combining "racist" and "sexist" tweets into a second "hate speech" category. Badjatiya et al. (2017)'s work, which was published during the writing of this thesis, also used Waseem's dataset and achieved a higher classification performance than any prior method. They used an ensemble of an LSTM model, features extracted by character n-grams, and Gradient Boosted Decision Trees.

Our research proposes expanding on Badjatiya et al. (2017)'s work in using neural networks for hate speech classification and incorporates out-of-text features such as the posting patterns and characteristics of the users. Chen et al. (2012) took into account writing styles of users such as imperative sentences or increased use of offensive words. More recent work by Papegnies et al. (2017) also incorporated contextual features such as number of respondents to particular messages and the number of friends of these users. Most recently, work by Pitsilis et al. (2018) combined the neural network approach and contextual user features, and they claim to have have scored higher on hate speech classification than Badjatiya et al. (2017)'s performance on the same Twitter dataset. Their approach employed multiple LSTM classifiers and user behaviors such as tendency towards racism or sexism. Our line of research is similar to Pitsilis et al. (2018)'s but will not be employing ensembled models. We demonstrate in our work that simple user features and task-specific word embeddings can increase classification accuracy.

Chapter 2

Background

In this chapter, we first present a definition of hate speech that we will use throughout our analysis. We put hate speech classification in context of social networks and formally define the problem statement. Then, we define several classification techniques in Section 2.4 that will provide the framework for hate speech classification.

2.1 Definition of Hate Speech

What is considered hate speech has no formal, legal definition, but there is consensus that hate speech is speech that carries expressions of hatred toward specific groups on the basis of characteristics like race, gender, religion, sexual orientation, or disability. In the United States, hate speech is protected under the First Amendment, unlike other types of speech such as "fighting words," which are face-to-face personal insults addressed to a specific person, of the sort that are likely to start an immediate fight (Volokh 2015).

Some example of hate speech, drawn from our dataset, include:

1. "To Muslims a woman walking down the street is a giant vagina demanding to be raped if she is not covered by a tent."

2. "Yes, Muslim bigots are murdering Christians all over Africa and have been for decades."
3. "Just want to slap the stupid out of these bimbos!"
4. "Yes, you put in the wrong way. Cue dumb blonde jokes."

Davidson et al. (2017) defines hate speech as *language that is used to express hatred towards a targeted group or is intended to be derogatory, to humiliate, or to insult the members of the group.* This definition does not include all instances of offensive language because people could use highly offensive words such as *b*tch, n*gga,* and *f*g* when in non-offensive contexts, such as quoting rap lyrics or as slang on gaming platforms. The dataset that we use from Waseem and Hovy (2016) proposes a list of criteria for categorizing hate speech. Some items in this list include "uses a sexist or racial slur," "attacks a minority," and "blatantly misrepresents truth or seeks to distort views on a minority with unfounded claims."

Defining what counts and does not count as hate speech is highly subjective as perceived by people of different identities, so from hereafter, we will follow the definition in Waseem and Hovy (2016) as an objective definition of hate speech in this paper. We choose this definition because our hate speech dataset has been annotated following this definition. While perpetrators of hate speech may not consider their words to be hateful, we must analyze each text instance objectively with the criteria defined above. Hate speech is defined solely by the text and its effects on the victims, without consideration for the beliefs or objectives of the author of the speech.

Our discussion hereafter will also not pertain to philosophical or psychological arguments for how to categorize hate speech. We recognize the contentious debate surrounding whether hate speech should be regulated under John Stewart Mill's Harm Principle, as well as acknowledge the psychological harm victims suffer from hate speech. Given the subjectivity of these lines of research, we will judge text to be hate speech or not on the basis of the words alone.

2.2 Online Social Networks

The proliferation of online social networks (OSNs) has created a need for monitoring and detecting user-generated content on these platforms. OSNs are largely centered around users, who publish a profile, create content, and form relationships with other users on the platform. The social network provides a platform for finding users with particular characteristics or interests and forming relationships in the form of links between users. Sociologists have long studied the properties of social networks, such as the small-world effect (de Sola Pool and Kochen 1978), which states that neighbors of any given node are likely to be neighbors with each other. In a social network, this means that one person's friends are often friends with each other. An influential paper by Granovetter (1977) argues that a social network can be partitioned into "strong" and "weak" ties, and the "strong" ties are tightly clustered. The characteristics observed in social networks are readily observed online as well, as shown by the tendency for users to form tight-knit groups (Girvan and Newman 2002), and the decreased distance between users in the growth of large social networks (Kumar et al. 2010). By forming relationships with users who are similar to themselves, these user-groups may form polarized opinions in viewing content posted by similar users. User-groups with homogeneous characteristics have been shown to increase social polarization, which is a leading contributor towards hate speech.

Our research focuses on Twitter, which allows users to post "tweets," which are posts less than 140 characters to express themselves. A user can follow any user, but the second user does not need to follow back, creating direct edges between users. Users may tag other users in their tweets using mentions (@ symbols), tag topics using hashtags (#), and retweet another user's tweet using the phrase RT. This well-defined markup vocabulary combined with the limit of 140-character tweets results in brevity of expression. Our decision to study Twitter data stemmed from the abundance of hate speech content on the platform, the relative ease of crawling the user-graph

compared to other social networks like Facebook, and the user-graph structure of direct edges and meta-information such as likes and retweets on individual tweets.

2.3 Classification Problem Definition

We formally define the problem of text classification as follows:

Let $\boldsymbol{x}_i \in \mathbb{R}^k$ be the $k-$dimensional word vector that corresponds to the ith word in a sentence. A sentence of length n, padded where necessary (if window size is bigger than n), is represented by

$$\boldsymbol{x}_{1:n} = \boldsymbol{x}_1 \oplus \boldsymbol{x}_2 \oplus \ldots \oplus \boldsymbol{x}_{n-1} \oplus \boldsymbol{x}_n$$

where $\oplus$ is the concatenation operator for strings. $\mathcal{V}$ is the vocabulary, which consists of all of the words in all observed sentences.

For a sentence $\boldsymbol{x}^j_{1:n}$ representing the jth observation, we want to predict its label, denoted $y^j \in \{1, 2\}$ by taking the argmax over the log probabilities of each label. We test the accuracy of our model on our validation set and record the number of correctly classified sentences compared to the labeled data.

2.4 Text Classification Methods

This section introduces baseline methods of classification such as bag-of-words, naive Bayes models, and support vector machines. Then, we describe the structure of a basic neuron and discuss the relevant models such as convolutional neural networks, recurrent neural networks, and a word-embedding network called fastText.

2.4.1 Previous Classification Methods

Prior to the popularity of neural networks, researchers have performed text classification using a number of machine learning techniques, including tweet features such as punctuation, URLs, part-of-speech tagging, n-grams, bag-of-words (BoW) (Kwok and Wang 2013), and lexical features that rely on lists of offensive words (Gitari et al. 2015). They also extract user-based features such as a user's number of friends and followers, biography information, and membership duration (Chen et al. 2012). These features are used in regression models (Nobata et al. 2016), support vector machines (Magu et al. 2017), decision trees (Dumais et al. 1998), and naive Bayes classifiers (S.-B. Kim et al. 2006).

Bag of Words

The bag of words model is one of the most popular techniques for text classification. Intuitively, we create "bags of words," or corpuses, that correspond to each category, then match new text against these corpuses to identify which category the text most likely came from.

Definition 2.4.1 (Bag of Words)**.** *A text T is a sequence of individual word tokens $t_1, t_2, \ldots, t_n$. The bag of words model maps T to a vector M_T of dimension $|V|$, where V is the set of all tokens in the training data. M_T contains the term frequency of each token in V that occurs in T.*

In most languages, some words tend to appear more often than others, such as "the," "a," and "is." We can employ a method called term frequency-inverse document frequency (TF-IDF) that replaces word frequencies with a metric called relevancy. Relevancy is measured by

$$r_{i,j} = tf_{i,j} \times log(\frac{N}{df_i})$$

where $tf_{i,j}$ is the number of occurrences of token i in document j, df_i is the number of documents containing token i, and N is the total number of documents. In this paper, we treat each tweet as a document and search for words that are distinct in documents and occur frequently in its category.

While the bag-of-words technique has seen great success in text classification tasks, it suffers some shortcomings, such as ignoring the context of the words as well as word ordering. Word representations can also be sparse, resulting in limited information to match on for the validation and test sets.

Support Vector Machines

Definition 2.4.2 (Support Vector Machine)**.** *A support vector machine is a classification technique that constructs a hyperplane which separates data by a maximal margin.*

Forgoing a formal definition, the intuition behind support vector machines is that the data points in a finite dimensional space may not be linearly separable, so they are mapped into a higher dimensional space through a kernel function. We maximize the maximal margin when finding a hyperplane, which is the distance between the separating decision hyperplane and the closest data points orthogonal to the hyperplane.

A decision hyperplane is defined by an intercept b and a weight vector $\boldsymbol{w}$ which is perpendicular to the hyperplane. All points on the hyperplane satisfy $\boldsymbol{w^T x} = -b$. Given a set of training data points $\{\boldsymbol{x}, y\}$ where each vector input $\boldsymbol{x}$ has a class y, the linear classifier is

$$f(\boldsymbol{x}) = sign(\boldsymbol{w^T x} + b)$$

where a value of -1 indicates one class and 1 indicates the other class. We aim to minimize $|\boldsymbol{w}|/2$ subject to $y(\boldsymbol{w^T x} + b) \geq 1$ to find the decision hyperplane.

2.4.2 Neural Networks

In recent years, artificial neural networks have surpassed previous machine learning methods with relatively little manual tuning of parameters or feature extraction. Artificial neural networks (ANNs) were originally inspired by the biological structure of brains, in which cells called neurons form connections that strengthen or diminish based on the environmental inputs that it receives. An insight introduced by Donald Hebb in his book *The Organization of Behavior* in 1949 famously states, "neurons that fire together wire together" (Hebb 2005). This insight, called Hebbian theory, states that neurons that fire one after the other in response to environmental stimuli are strongly correlated. In ANNs, Hebb's principle can be applied to the process of adjusting the weights between the model's neurons, which will be explained in further detail below.

Basic Definitions and Notation

The basic unit of computation in an ANN is the **neuron**, also referred to as a **node** or **unit**. It receives input from other nodes or from an external source and computes an output, which is usually a number that summarizes its inputs by passing these inputs through a non-linear function. There is a weight associated with each input that represents the importance of that input to this neuron. The neuron contains non-linear function f that is applied to the weighted sum of its inputs.

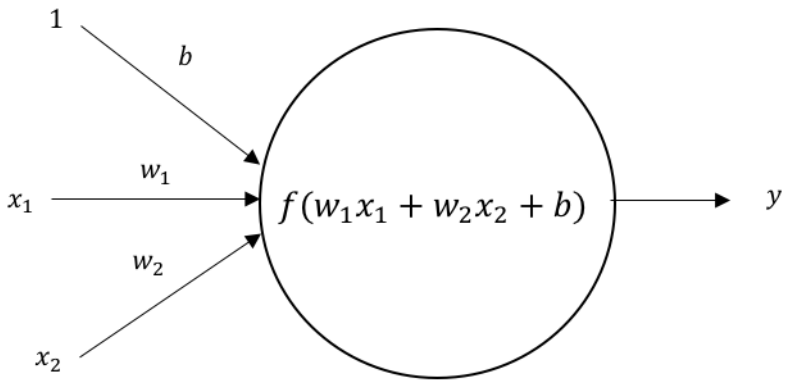


Figure 2.1: Neuron in neural model that takes inputs x_1, x_2, and a bias b, multiples times by the weights, applies an activation function f, and returns output y.

In Figure 2.1, the neuron takes inputs x_1 and x_2, associated respectively with weights w_1 and w_2. An additional input, b called bias, is also associated with the neuron. The **potential function**, ϵ, is the sum of the dot products between the weights and input neuron values. The neuron computes output y from applying function f, the **activation function**, on the potential function. If two neurons are not connected, then the weight between them is 0.

Definition 2.4.3 (Neural Network)**.** *A* ***neural network*** *is defined as a sorted tuple* (N, V, w, b) *where* N *is the set of neurons associated with this network, and* V *is a set* $\{(i,j)|i,j \in \mathbb{N}\}$ *whose elements are the* ***connections*** *between neuron* i *and neuron* j*.* $w : V \to \mathbb{R}$ *defines the* ***weights****, where* w_{ij} *is the weight between neurons* i *and* j*.*

The bias b_{ij} for each pair of connected neurons is a measure of how easy it is for the neuron to reach a state of activation. It shifts the output of the neuron by a constant for the aggregated inputs that the neuron receives. The weights can be implemented in a **weight matrix**, where the row indices are the input neurons, and the column vectors are the output neurons.

As an example, consider a neuron j with inputs $x_j = x_{1j}, x_{2j}, \ldots, x_{nj}$, weights $w_{1j}, w_{2j}, \ldots w_{nj}$, and bias b_j. The potential function and output of the neuron is

$$\epsilon_j = (\Sigma_j w_{ij} x_{ij}) + b_j$$

$$y_j = f(\epsilon_j) = f((\Sigma_j w_{ij} x_{ij}) + b_j)$$

If an ANN contains m neurons in the output layers, we obtain the output network $\mathbf{y} = y_1, y_2, \ldots, y_m$.

The purpose of activation function f is to introduce non-linearities into the neural network. Because real-world data is non-linear, our neural networks should be able to model these non-linearities. A number of common activation functions are shown in Figure 2.2.

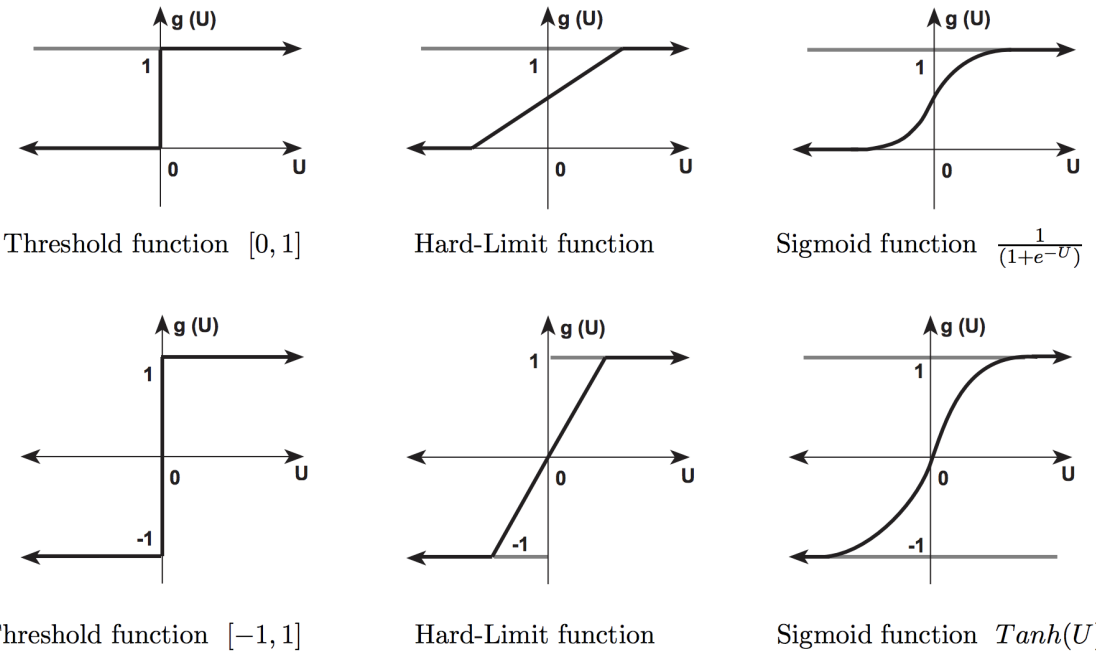


Figure 2.2: Various activation functions used in neural networks.

The activation function depends on the specific task. The rectified linear unit (ReLU), whose formula is $f(x) = \max(0, x)$, is currently popular in deep learning re-

search. The tanh function, whose formula is $\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, is popular in feedforward and recurrent neural networks. For multiclass classification with K labels, which is the the task we are interested in, we perform a softmax nonlinearity

$$\hat{y_k} = \frac{e^{x_k}}{\Sigma_{k'=1} e^{x_{k'}}} \text{ for } k = 1 \text{ to } k = K$$

where $\hat{y_k}$ is a probability between 0 and 1, and x_k is the input to the softmax function. The softmax function normalizes the terms so that the outputs of the classification probabilities sum to 1.

Feedforward Neural Networks

The feedforward neural network consists of many layers of neurons arranged in **layers**: one **input** layer, n **hidden** layers, and one **output** layer. In a feedforward network, each neuron in one layer only has directed connections to the neurons in the next adjacent layer. The input $\boldsymbol{x}$ is represented by setting values in the input layer, which then feed these values to the first hidden layer. Values for neurons in each network are successively computed, until an output $\hat{\boldsymbol{y}}$ is generated at the output layer. The neural network learns by iteratively updating the weights between the neurons by minimizing a loss function $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$.

Training Process

The most common method to train neural networks is **backpropagation** (Rumelhart et al. 1985), which uses the chain rule to calculate the derivative of the loss function $\mathcal{L}$ with respect to each parameter in the network.

The network weights are adjusted by gradient descent. Most networks are trained through stochastic gradient descent, in which weights are updated according to

$$W \leftarrow W - \eta \nabla_W F_i$$

where η is the learning rate and $\nabla_W F_i$ is the gradient of the objective function with respect to the parameters W as calculated on a single example. Many popular heuristics of gradient descent exist, including AdaGrad, AdaDelta, and Adam.

For output $\hat{\boldsymbol{y}}$ and true value $\boldsymbol{y}$, a loss function $\mathcal{L}(\hat{y_k}, y_k)$ is calculated for each output node k in the output layer. We calculate

$$\delta_k = \frac{\delta \mathcal{L}(\hat{y_k}, y_k)}{\delta \hat{y_k}} \cdot f_k'(\epsilon_k)$$

For each node in the hidden layer immediately before the output layer, we calculate

$$\delta_j = f'(\epsilon_j) \Sigma_k \delta_k \cdot w_{kj}$$

This calculation is performed successively for each hidden layer, and each δ_j value represents the derivative $\delta \mathcal{L} / \delta \epsilon_j$ of the total loss function with respect to the node's incoming activation. Given values y_j calculated from the forward pass and δ_j calculated from the backward pass, the loss $\mathcal{L}$ with respect to parameter $w_{jj'}$ is

$$\frac{\delta \mathcal{L}}{\delta w_{jj'}} = \delta_j w_{jj'}$$

2.4.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) were originally designed to recognize features in two-dimensional image data. CNNs were first inspired by the biological research of Hubel and Wiesel (1968) in neurobiological image processing in the cat cortex. They have been adapted to not only perform image recognition in machine learning, but also have been used in scene labeling, face recognition, and speech recognition. Collobert et al. (2011) has shown CNN uses in NLP tasks, such as part-of-speech tagging, chunking, named entity recognition, and semantic role labeling. Most recently, CNNs have been used in text classification (Y. Kim 2014).

Architecture of CNNs

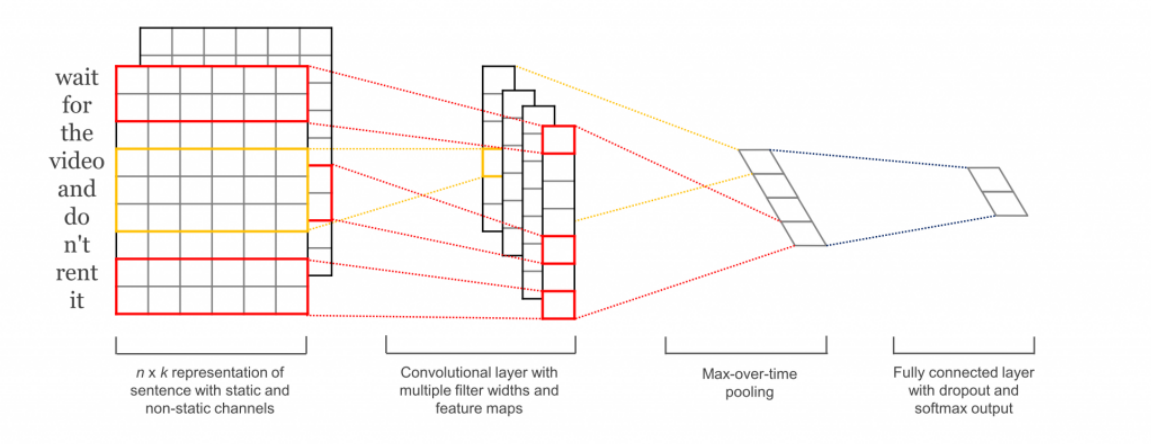


Figure 2.3: Y. Kim (2014) Convolutional Neural Networks for Sentence Classification

In sentence classification, the input is sentence s and the output is a class $y \in 0, 1$ which represent two different classes for sentence s. Four types of layers form a convolutional neural network.

Embedding Layer

Instead of image pixels, the input to an CNN for NLP tasks is a matrix-representation of sentences or documents. A CNN first processes the sentence through an embedding layer that turns words into word embeddings, also called word vectors. Each matrix row corresponds to one token, typically a word. The matrix dimension used for CNNs in this case is sentence length times word embedding dimension.

Convolutional Layer

Then, the CNN passes the embeddings through convolutional layers to extract salient n-gram features from the input sentence to create latent semantic representations of the sentence.

A convolution operation involves a filter $W \in \mathbb{R}^{hk}$, which applies to a window of h words to produce a feature. The window operations generate a new feature. A

filter c_i is generated from a window of words $\boldsymbol{x}_{i:i+h-1}$ by

$$c_i = f(W \cdot \boldsymbol{x}_{i:i+h-1} + b)$$

The filter applies to every possible window of words in the sentence $\{\boldsymbol{x}_{1:h}, \boldsymbol{x}_{2:h+1}, \ldots, \boldsymbol{x}_{n-h+1:n}\}$ to produce a feature map

$$\boldsymbol{c} = [c_1, c_2, \ldots, c_{n-h+1}]$$

Pooling Layer

Then, the pooling layer, also called a subsampling layer, reduces the spatial size of the representation by applying an operation, such as max, sum, average, or L2-norm. Two main reasons motivate pooling: to turn a variable size input into a fixed size output matrix typically necessary for classification and to reduce the output dimensionality while keeping the most salient information about the input sentence.

Given feature map $\boldsymbol{c}$, a max-over-time pooling operation takes the maximum value $\hat{c} = max\{\boldsymbol{c}\}$ as the value that corresponds to this particular feature. This captures the most important feature (the one with the highest value) for each feature map.

The pooling layer has one additional benefit when working with natural languages in that it automatically standardizes variable sentence lengths.

Fully-connected layer

In a fully-connected layer, neurons have connections to all activations in the previous layer, compared to a convolutional layer, which is connected only to a local region in the input. The output values are similarly computed by a matrix multiplication with a bias offset.

Multi-channel CNNs

Y. Kim (2014) describes a CNN model variation with multichannels, where each set of word embeddings is treated as a "channel," similar to separating RGB layers in an image. Filters are applied over all channels, while gradients are backpropagated over only one of these channels. Kim's model fine-tunes one set of vectors while keeping the other static. This multi-channel architecture can be extended to other applications, including using multiple word embeddings.

2.4.4 Recurrent Neural Networks

Another type of neural network is the recurrent neural network (RNN), which performs the same task for every element in the sequence, with the output dependent on previous computations. An RNN is more powerful than a CNN in time-dependent or sequential tasks because it maps from the entire history of previous inputs to each output. These recurrent connections allow the nodes to have "memory" of previous inputs which influences the network output.

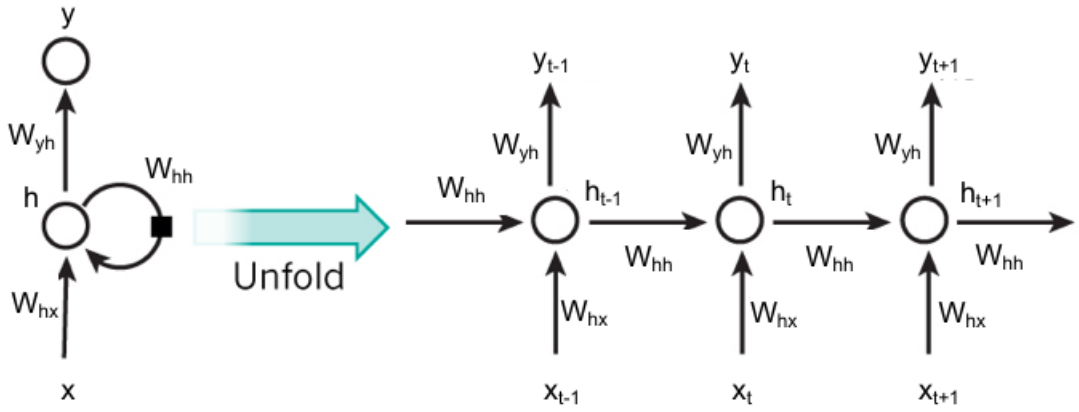


Figure 2.4: An RNN architecture. The network receives input x_t at time t, and a hidden state s_{t-1} from the previous time step. s_{t-1} is the memory of the network. At the current time step, we calculate a new $s_t = f(Ux_t + Ws_{r-1})$, where f is an activation function such as tanh or ReLU. Output o_t is derived from softmax(Vs_t)

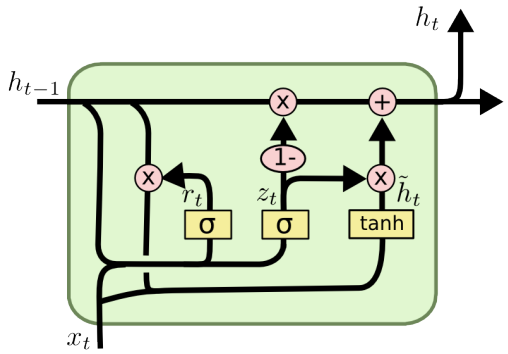
At time t, a node receives input from the current data point $\boldsymbol{x}$ and values $\boldsymbol{h_{t-1}}$ from the network's previous state. The output $\boldsymbol{\hat{y}_t}$ is calculated from the hidden node values $\boldsymbol{h_t}$. The following two equations specify the all calculations during the forward pass for a specific timestep t:

$$\boldsymbol{h_t} = f(W_{hx}\boldsymbol{x_t} + W_{hh}\boldsymbol{h_{t-1}} + \boldsymbol{b_h})$$

$$\boldsymbol{\hat{y}_t} = \text{softmax}(W_{yh}\boldsymbol{h_t} + \boldsymbol{b_y})$$

where W_{hx} is the matrix of convolutional weights between the input and hidden layer and W_{hh} is the matrix of recurrent weights between the hidden layer and itself at adjacent time steps. The vectors $\boldsymbol{b_h}$ and $\boldsymbol{b_y}$ are bias parameters that allow each node to learn an offset.

The RNN can be thought of as a deep neural network with one layer for each time step and shared weights across time steps. This process of discretizing the neural



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 2.5: The LSTM is made of four neural network layers instead of one, consisting of a forget gate f, an input gate i_t, C_t, and a sigmoid gate σ_i.

network by time steps is called unfolding. Then, it is clear that the same training and backpropagation algorithms can be applied to this neural net as described earlier.

Long Short-Term Memory Networks

The long short-term memory network (LSTM) was introduced by Hochreiter and Schmidhuber (1997) to tackle a problem found in RNNs called vanishing gradients. During backpropagation, information passes through many stages of multiplication, which causes them to grow at exponential rates or reach zero. When the gradients get large and are suppressed by a non-linear activation function, they lose information that is crucial in training the neural network.

The LSTM model resembles a standard RNN model, but each node is replaced by a *memory cell*. The memory cell is composed of an input gate, an output gate, a forget gate, and a neuron that connects back to itself. Let $c_t^l \in \mathbb{R}^n$ represent a memory cell in layer l and time step t. Let $\boldsymbol{h_t^l} \in \mathbb{R}^n$ be a hidden state at time t and layer l. $\boldsymbol{h_t^{l-1}}$ and $\boldsymbol{h_{t-1}^l}$ predicts $\boldsymbol{h_t^l}$, and x_t denotes new input at time t.

The forget gate layer decides what information does not need to be kept from the cell state, performing a sigmoid operation on an affine transform, with W_f and b_f as its weight and bias.

$$\boldsymbol{f_t} = \sigma(W_f[\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_f})$$

The next step decides what information to store in the cell state, in which we have a sigmoid layer that decides which values to update, $\boldsymbol{i_t}$, and a tanh layer that creates a vector of new candidate values, $\boldsymbol{g_t}$.

$$\boldsymbol{i_t} = \sigma(W_i[\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_i})$$

$$\boldsymbol{g_t} = \tanh\left(W_g[\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_g}\right)$$

We update the old cell $\boldsymbol{c_{t-1}}$ into a new cell $\boldsymbol{c_t}$ through

$$\boldsymbol{c_t} = \boldsymbol{f_t} \times \boldsymbol{c_{t-1}} + \boldsymbol{i_t} \times \boldsymbol{g_t}$$

Intuitively, LSTMs allow the network to decide what information to forget from the previous state in the forget gate, decide what information to store in the cell state through the input gate and and tanh layer, and finally, how to update the cell state. In the last step, we multiple the old state $\boldsymbol{c_{t-1}}$ by $\boldsymbol{f_t}$, then add the new values scaled by how much we decided to update each value $\boldsymbol{i_t} \times \boldsymbol{g_t}$.

Finally, we decide what to output, in which we run a sigmoid over the previous hidden state and current input then a tanh over the values we decide to output.

$$\boldsymbol{o_t} = \sigma(W_o[\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_o})$$

$$\boldsymbol{h_t} = \boldsymbol{o_t} \times \tanh\left(\boldsymbol{c_t}\right)$$

2.4.5 Word Embeddings

Word embeddings refer to various techniques that map words or phrases to dense vector representations that allow for computation of semantic similarities of words. Word embeddings were first popularized by Bengio et al. (2003) and have been increasingly

used in natural language processing tasks. Words can be approximately modeled in an N-dimensional space that is sufficient to encode the semantics of each particular language. Each dimension encodes some meaning in the word, such as tense (past, present, future), count (singular, plural), and gender (masculine, feminine, neutral). Recent techniques to construct word embeddings have been based on neural network language models (NNLMs) (Mikolov et al. 2013), which take into account the word's neighboring words as context for the word's meaning. Cosine similarity is the typical measure for vector similarity.

Definition 2.4.4 (Word Embedding)**.** *A word embedding* W : *words* $\to \mathbb{R}^n$ *is a parametrized function mapping words in a language to low-dimensional vectors.*

Let us start with an example: "The puppy jumped onto the sofa."

One approach is the continuous bag-of-words (CBOW) model, which treats ["The", "puppy", "onto", "the", "sofa"] as context to predict the word "jump." The context word vectors predict the current word vector, then a loss is calculated between the predicted word vector and the actual word vector and adjusted using gradient descent.

$$p(w|\mathcal{C}) = \frac{e^{h_{\mathcal{C}}^{\intercal} v_w}}{\Sigma_{k=1}^{K} e^{h_{\mathcal{C}}^{\intercal} v_k}}$$

where $h_{\mathcal{C}}$ is the feature for context $\mathcal{C}$ and v_w is the classifier for word w.

Another approach is the skip-gram model, which takes the center word, "jump," and predicts or generates the surrounding words "The," "puppy," "onto," "the," and "sofa." The training process is similar to the CBOW model, but now in reverse.

$$p(c|w) = \frac{e^{x_{w_t}^{\intercal} v_c}}{\Sigma_{k=1}^{K} e^{x_{w_t}^{\intercal} v_k}}$$

where x_w is the word vector for word w and v_c is the classifier for word c.

These word vectors are trained through minimizing log likelihood:

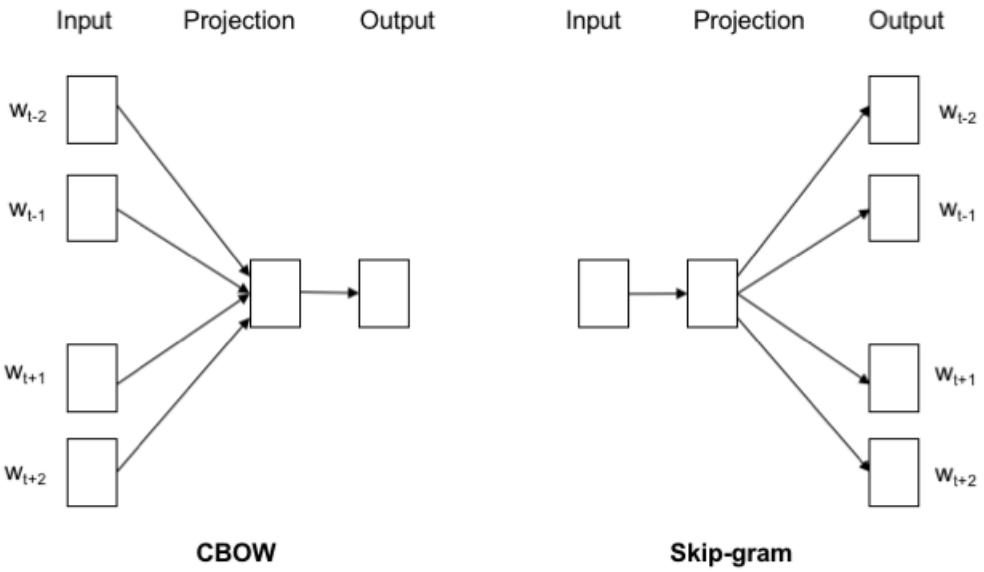


Figure 2.6: The CBOW architecture predicts the current word based on the context, and skip-gram predicts surrounding words given the current word.

$$\min_{x,v} -\Sigma_{t=1}^{T}\Sigma_{c\in\mathcal{C}_t} log \frac{e^{x_{w_t}^{\intercal} v_c}}{\Sigma_{k=1}^{K} e^{x_{w_t}^{\intercal} v_k}}$$

Word vectors are extremely powerful representations of semantic relationships between words. For example, there is a constant male-to-female difference vector:

$$W(\text{“woman”}) - W(\text{“man”}) \simeq W(\text{“queen”}) - W(\text{“king”})$$

Other relationships are encoded similarly, including cities to countries (Paris - France, Tokyo - Japan, San Francisco - California), comparative words (big - bigger - biggest, small - smaller - smallest, cold - colder - coldest), and companies to products (Microsoft - Windows, Google - Android, Apple - iPhone).

While word embeddings have become an essential part of natural language processing and are used widely in neural networks and other models, they can fall short when modeling terms that have multiple definitions. One such example is in online

extremest groups which reappropriate words such as “Google” or “Skype” to have alternate hate speech definitions. They are able to go undetected by online hate speech detection algorithms, given the difficulty in differentiating between mentions of “google” or “skype” as technology companies and as derogatory labels for black and Jewish populations. Previously, extremest groups put parentheses around Jewish surnames to target Jews for offensive slurs, similar to forced wearing of the Yellow Star during the Holocaust. Their methods are now more sophisticated and cannot be easily detected by online spam filters. Therefore, we need better methods for hate speech detection by creating word mappings that relate the semantic similarities of code words with known derogatory words.

Code Word	Meaning
Skype	Jew
Google	Black person
Butterfly	Gay man
Durden	Transgender
Car salesmen	Liberals
Reagans	Conservatives
Fishbucket	Lesbian
Skittle	Muslim
Bing	Chinese
Yahoo	Mexican

Table 2.1: Hate speech code words and their corresponding meanings.

By understanding the relationships between words and their semantic representations, natural language processing tasks such as classification can much better categorize sentences and documents.

2.4.6 fastText

fastText is a method developed by Facebook in 2016 that builds on theoretical ideas in word embeddings for text classification. Instead of modeling the probability of a target word given the context or the context given the target word, fastText models the probability of a **label** given a **paragraph**.

$$p(l|\mathcal{P}) = \frac{e^{h_{\mathcal{P}}^T v_l}}{\Sigma_{k=1}^{K} e^{h_{\mathcal{P}}^T v_k}}$$

where $h_{\mathcal{P}}$ is a feature for paragraph $\mathcal{P}$ and v_l is a classifier for label l. Each paragraph feature $h_{\mathcal{P}}$ is the sum of its word representations.

$$h_{\mathcal{P}} = \Sigma_{w \in \mathcal{P}} x_w$$

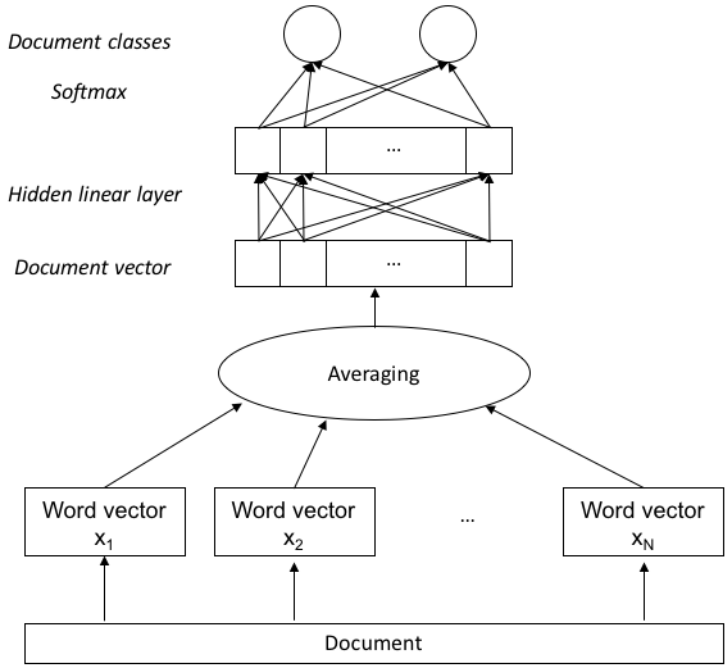


Figure 2.7: fastText architecture, which is a bag of words classifier with a hidden linear layer. Word vectors for all words in the document are averaged into one document vector representation.

A bag of n-grams can better represent phrase-level features, which are stored efficiently in memory through a hashed dictionary.

fastText can also be used to generate word embeddings. The difference between fastText and other methods such as *word2vec* and *GloVe* is that it represents words as the sum of its character n-grams. The benefits of using character n-grams is that compound nouns are easy to model, and in languages with declensions, these variations will be represented with the same root word instead of many distinct entries. This model is beneficial for identifying code words in hate speech that are misspellings or variations of other words, as well as identifying compound words that take on separate meanings than the stem words.

Chapter 3

Methodology

In this chapter, we begin by defining the hate speech classification problem. We show how our research uses the text classification methods described in Chapter 2 to classify hate speech. We also present our methods to engineer tweet-level and user-level features as well as hate-speech specific word embeddings. Finally, we describe the experimental settings used for Chapter 5.

3.1 Problem Statement

The central research question we address in this section is the following:

Given a new tweet and information about the author's Twitter account, how effectively can we classify this tweet as hate speech or not hate speech?

We use a two-pronged approach to answer this question:

1. Develop a new neural network architecture that improves state-of-the-art probabilities that a tweet is classified correctly.

2. Investigate whether incorporating additional information from the social network, such as tweet metadata or user profile information improves classification accuracy.

3.2 Classification Methods

We train a variety of CNN, LSTM, and fastText models on our dataset to classify hate speech. We use grid search to tune the hyperparameters of thees models and early stopping to prevent overfitting. We append tweet and user-level features to our dataset when running CNN and LSTM models to investigate the performance of additional features on model performance. We also try different word embeddings for each of the three models.

3.3 Datasets

Waseem and Hovy 2016 make available a dataset containing 16,914 annotated tweets, of which 5,355 tweets are labeled as either sexist or racist. These tweets belong to a group of 614 Twitter users. The dataset is constructed by performing a manual search of common slurs and terms related to hate speech, then sampling the public Twitter search API to collect tweets with the selected words. This ensured that the dataset contained potentially offensive words used in non-offensive contexts, such as *"you are right there are issues but banning Muslims from entering doesn't solve anything"*. The dataset was manually annotated by the authors and reviewed by a non-activist feminist woman studying gender studies to mitigate bias. They achieved an inter-annotator agreement of $\kappa = 0.84$ [1].

3.3.1 Data Preprocessing

We processed each tweet using the `tweet-preprocessor`[2] Python package to replace any URLs with "url," user mentions with "mention," and reserved words such as

[1] κ is calculated as Cohen's kappa, which measures inter-rater agreement for qualitative (categorical) items $\kappa = \frac{p_0 - p_e}{1 - p_e}$ where p_0 is the agreement among raters, and p_e is the hypothetical probability of chance agreement

[2] https://pypi.python.org/pypi/tweet-preprocessor/0.4.0

"RT" and "FAV" with "reserved." Doing so helped decrease vocabulary size and remove one-off words or URLs that do not exist in the English dictionary. While the URLs and user mentions contain valuable information such as links to hate speech websites or images promoting hate speech, we limit our study to only the textual content of a tweet. We lowercase our tweets and remove all punctuation.

Because we are only interested in predicting for hate speech or not hate speech, we group tweets labeled with "sexism," "racism," and "both" into the same "hate speech" category.

3.3.2 User Features

To augment our text-based data, we also construct features from the tweet's contextual information. Our features consist of the number of likes and retweets for the individual tweet, as well as the number of followers and number of friends (people the user follows) of the tweet's author. The choice of constructing features from likes and retweets stems from our hypothesis that tweets expressing stronger sentiments will incite stronger reactions from a user's followers, resulting in likes or retweets. We also hypothesize that users who are following fewer people are exposed to less diverse information than users who are following many other accounts, so their views may be more polarized than someone who receives a diversity of tweet information.

We take the square root of each user feature to ensure that the numbers were not off by an order of thousands, given that the following statistics:

retweets	favorites	user following num	user follower num
45083	830	50543	299362

Table 3.1: Maximum numbers for each of the four features. Because the numbers varied on the order of thousands, we take the square root of each of these numbers.

The four features that we are using, the number of retweets, likes, user follower

count, and user following count are appended to the hidden representation of the input immediately prior to the last fully-connected layer. We choose to append these features to the second-to-last layer to dissociate these features from being grouped into n-grams in CNNs or affecting the sequential processing in RNNs.

3.4 Word Embeddings

For word embeddings, we experimented with using pre-trained embeddings and training our own embeddings on hate speech datasets. For pre-trained embeddings, we use fastText pre-trained word embeddings (English, 300 dimensions) (Bojanowski et al. 2016). While many pre-trained embeddings exist online as published by Google (Mikolov et al. 2013), Facebook (Bojanowski et al. 2016), and Stanford (Pennington et al. 2014), training our own word embeddings better capture the semantic similarities of a particular domain of words.

We improve upon existing techniques for hate speech classification by training our word embeddings on a known hate-speech dataset instead of using pre-trained word embeddings. We use a dataset developed by Taylor et al. (2017), which consists of articles from DailyStormer[3] and tweets from Twitter accounts of identified white-supremacists that the authors created by measuring user centrality on Twitter. Tweets were selected from the users with the highest centrality. DailyStormer articles are more typically written by white supremacists.

We train fastText and word2vec models that combined DailyStormer and tweet data to create a word embedding that reflects the semantic similarities of words in domains with a high concentration of hate speech. Using words from Hatebase [4], an open-source repository of structured, multilingual, usage-based hate speech, we found the top 30 most common words for each word in Hatebase. Many words in Hatebase

[3]https://www.dailystormer.com/
[4]https://www.hatebase.org/

were not found in our vocabulary. This may be because words only need to have one sighting to be recorded on Hatebase, and Hatebase's validity is difficult to measure given that it is crowd sourced. We compare the top 30 words from the model trained on hate speech and compare it to a pre-trained embedding to identify words that are in the hate speech model that could be hate-related code words.

We implement Y. Kim (2014)'s model of multichannel convolutional networks, and instead of backpropagating through one embedding and keeping another word embedding static, we use two different word embeddings which are concatenated before the convolutional step. Yin and Schutze (2015) found multiple embeddings to improve performance because the meta-embeddings contain more information than each component embedding. We align these word embeddings so that vectors are aligned to the same axes (Hamilton et al. 2016). This allows us to compare the same word across different embeddings and simplifies the weight updates procedure. Aligning word embeddings uses orthogonal Procrustes. Defining $\boldsymbol{W}^1, \boldsymbol{W}^2 \in \mathcal{R}^{d \times |\mathcal{V}|}$ as our two word embeddings, we align word embeddings while preserving cosine similarity by optimizing $\boldsymbol{R}^2 = \arg\min_{\boldsymbol{Q}^\intercal \boldsymbol{Q} = \boldsymbol{I}} \|\boldsymbol{Q}\boldsymbol{W}^2 - \boldsymbol{W}^1\|_F$ where $\boldsymbol{R}^2 \in \mathcal{R}^{d \times d}$ and $\|\cdot\|_F$ is the Frobenius norm.

the	googles	skypes	jew	muslim
in	niqqers	cocksuckers	christ-killer	[bomb emoji] muslim
of	chars	jidf	kike	musl / mosl
which	yahoos	trogs	baby-killer	moslem
and	bants	googles	jewgle	gang-rapists
a	ferals	britcucks	ratlike	child-raping
that	dindonuffin	shitposters	jew-over	gang-sex
however	nig-nogs	wogs	jonestein	muzrats
also	ray-cyst	kikes	jewdar	tray-trays
only	jigaboos	1488ers	lolcow	muzzies
an	baboon	lap dog	kikebart	mudslimes

Table 3.2: Words from top 30 most similar words in the DailyStormer embedding that were not found in the *word2vec* top 30 embeddings, with the exception of "the" which is used as an example for comparison.

the	googles	skypes	jew	muslim
this	google.com	skyping	jews	muslims
in	google_yahoo	SKYPE	jewish	Muslim
that	wikipedia	Gtalk	rahm	islamic
ofthe	googled	MSN_messenger	mhux	Moslem
another	googling	gmail	yid	christian

Table 3.3: Words from the top 10 most similar words in *word2vec*. If a word had two different capitalizations, only one is shown on this list.

3.5 Experimental Settings

We implement all of our models in PyTorch [5], a framework for Python based around Torch [6], which is a library for deep learning. To produce results in a comparable setup with compared to Badjatiya et al. (2017), we perform 10-fold cross validation and calculate *precision, recall* and *F1-scores* for each model. We split the training and test sets into 90% training and 10% validation and evaluated performance by averaging the precision, recall, and F1-scores across the 10 folds. Our batch size is set to 128. We use categorical cross-entropy as the learning objective and ADAM as our optimizer (Kingma and Ba 2014).

Our model runs for 20 epochs for every fold in the convolutional models, 30 epochs for every fold in the recurrent models, and 40 epochs for every fold in the fastText models to avoid overfitting. We employ early stopping to maximize validation accuracy. After experimentation, the optimal epoch generally occurrs between 18-23 epochs for CNNs, 28-33 epochs for RNNs, and 39-44 for fastText.

We perform paired t-test analysis on pairs of classifiers to determine their statistical significance and set the p-value at $\alpha = 0.05$. However, given that we are performing paired t-tests on a large number of pairs, we apply Bonferroni correction[7] and divide p by the number of comparisons, m. We only treat results with $p < \frac{\alpha}{m}$ as significant.

[5] http://pytorch.org/

[6] http://torch.ch/

[7] A method to counteract the multiple comparisons problem, in which the larger the number of inferences, the more likely erroneous inferences will occur. Approximately every 1 of 20 times, a non-statistically significant inference will have value $p < 0.05$.

Chapter 4

Feature Visualizations

4.1 Visualization Methods for Neural Networks

One of the biggest criticisms of deep neural networks is the lack of interpretability of the mechanisms. Historically, neural networks have been thought of as "black boxes," but recently visualization techniques have been able to shed light on how the inputs influence the final output. Graphing the magnitude of the weights in each of the neural layers allows us to better interpret which inputs are contributing highly to the result, and which inputs are being ignored.

4.2 Saliency Maps

Originally inspired by neural network applications to vision, we apply work from Simonyan et al. (2013) and Li et al. (2015) to measure how much each input word in the tweet contributes to the final classification.

4.2.1 Theory

Formally, an input X is associated with a class label c. In our experiment, we label c_0 as not hate speech and c_1 as hate speech. Given embedding $e_i \in E$ for word $x_i \in X$, the model that we train associates each word embedding e_i with a class label $\in \{c_0, c_1\}$ to derive a score $S_c(e_i)$. In visualization, we want to determine which word embeddings contribute the most to $S_c(E)$.

In deep neural networks, calculation of $S_c(E)$ is a non-linear function. We approximate $S_c(E)$ by computing a first-order Taylor expansion

$$S_c(e) \approx w(e)^T e + b$$

where $w(e)$ is the derivative of S_c with respect to word embedding e.

$$w(e) = \frac{\delta(S_c)}{\delta e}|_e$$

The absolute value of derivative $w(e)$ indicates the final class score's sensitivity to changes in the input word. The saliency of input word w_i is given by

$$S(e_i) = |w(e_i)|$$

4.2.2 Implementation

To implement this technique on a convolutional neural network, we compute the gradients of the network's prediction with respect to the input, holding the weights between the network layers constant. This allows us to determine which inputs affect the prediction the most. We perform guided backpropagation (Springenberg et al. 2014), which is a modified version of using the gradient as a proxy for the importance of each input. In guided backpropagation, we only backpropagate the error to every

positive input, and only backpropagate positive error signals. Intuitively, we are only interested in what features the neuron detects, and not what it does not detect. Guided backpropagation has been shown to produce cleaner visualizations than other methods such as deconvnet and simple backpropagation, which is why we chose guided backpropagation to visualize the convolutional layers. Examples of our visualized convolutional layers are below:

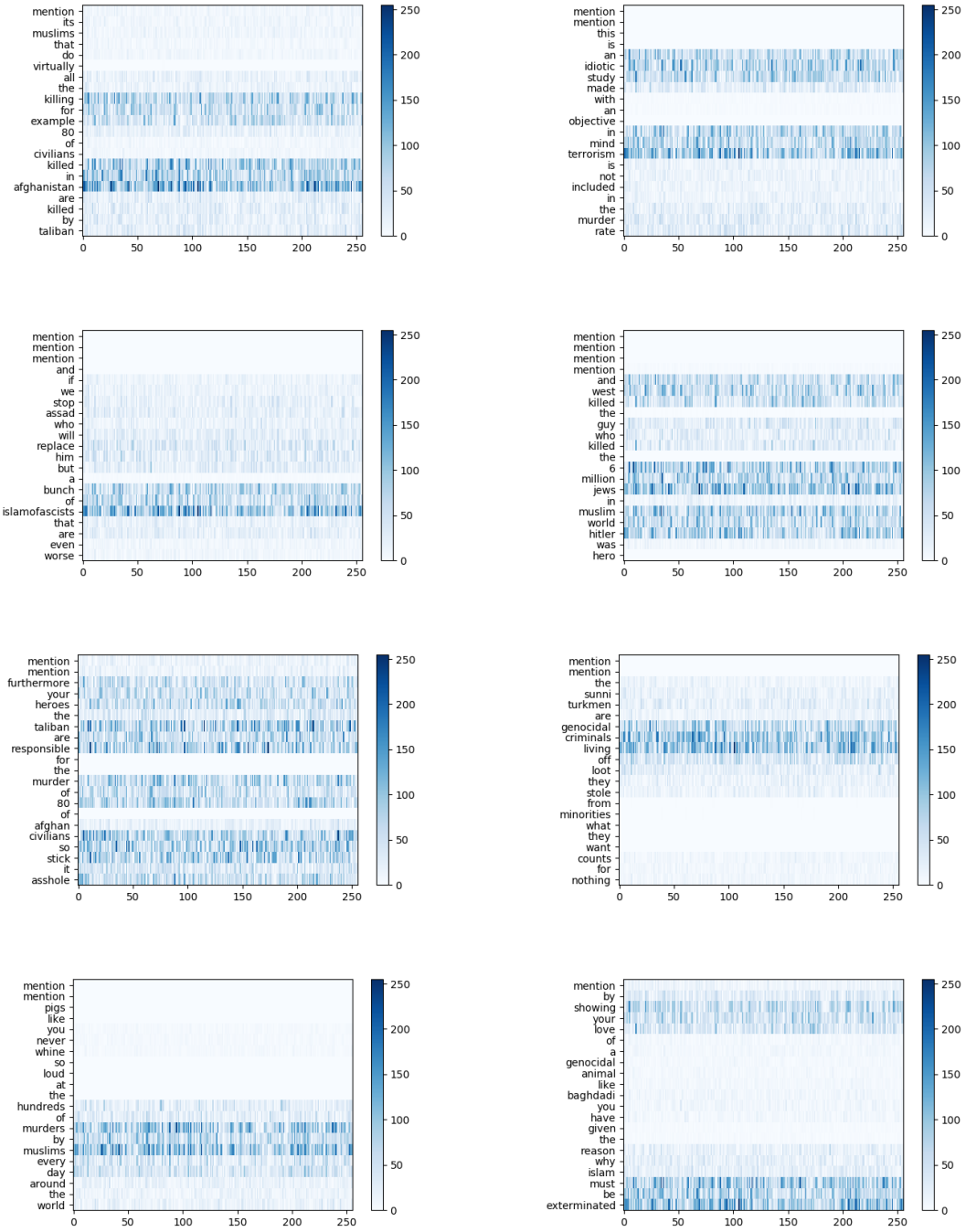


Figure 4.1: Visualizations of the guided backpropagation on the first convolutional layer in the CNN model. Each row corresponds to saliency scores for the correspondent word representation, with each grid in the row representing a unit in the hidden layer. Hidden layers have 256 units. All examples are labeled in the dataset as hate speech.

In all of the examples in Figure 4.1, the convolutional net assigns higher weights to

words that are more indicative of hate speech, including "terrorism," "islamofascists," "genocidal criminals," and "exterminated." Because the models span windows of sizes 3, 4, and 5, the models pick up on groups n-grams together, as evidenced by the higher weights that are clustered together in groups of approximately length 3. The words surrounding the highest weighted words also received higher weights.

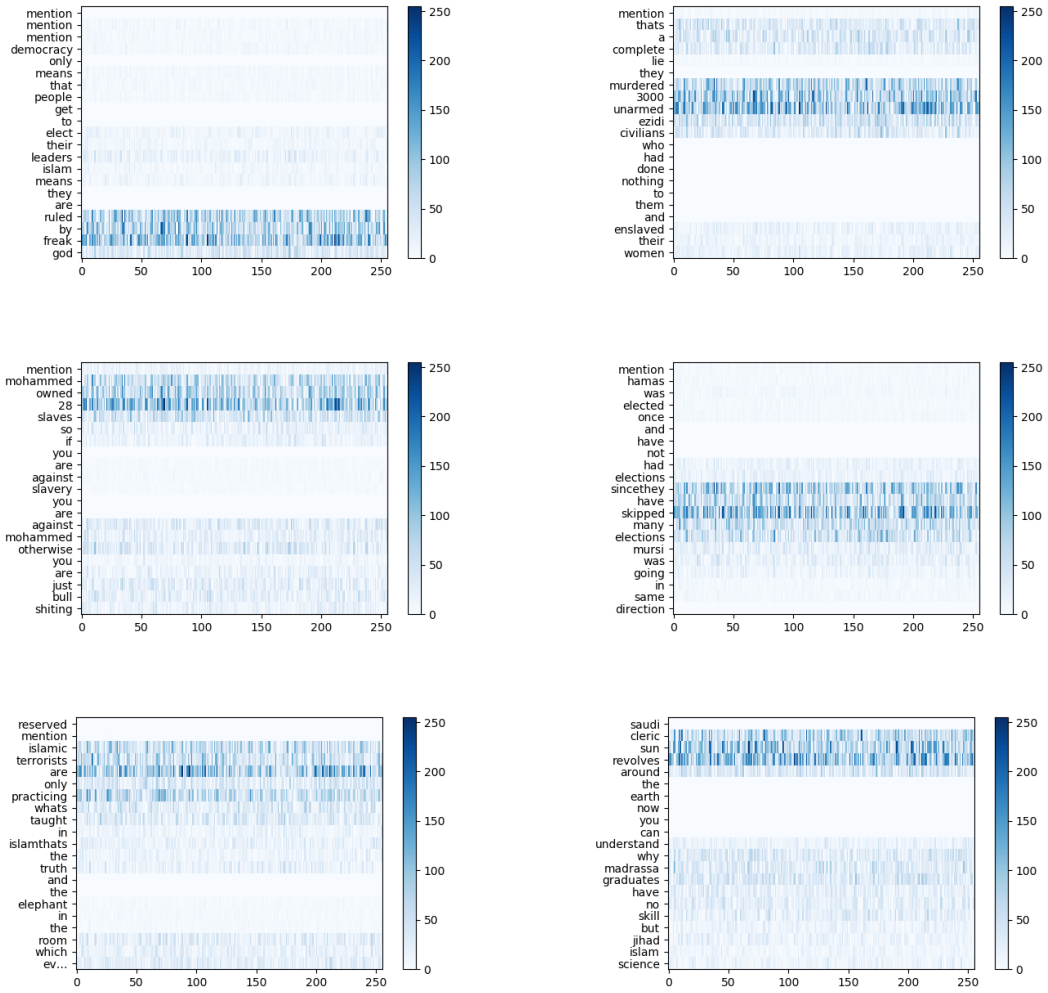


Figure 4.2: Visualizations of the guided backpropagation on the first convolutional layer in the CNN model for tweets that were incorrectly classified. Each row corresponds to saliency scores for the correspondent word representation, with each grid in the row representing a unit in the hidden layer. Hidden layers have 256 units.

Visualizing the convolutional layers also allows us to analyze sources for discrepancies between the predicted label and the actual label. Some incorrectly classified

tweets are in Figure 4.2. The model seems to place a large emphasis on numerical values, such as "28" and "3000." While we do not have clear explanations for why convolutional neural models assign larger weights to phrases such as "sincethey have skipped" or "cleric sun revolves," these visualizations are valuable tools to gain insight into hate speech features that may be missed when conducting qualitative or human-annotated classification.

Chapter 5

Experimental Results

5.1 Results and Analysis

To measure our performance, we use weighted precision, recall, and F1-score as standard measures of classification accuracy. Precision is defined as the ratio of the true positives over the total number of positive predictions, or in this case, the number of tweets correctly classified over the total number of tweets that were classified in a specific class. Recall is defined as the ratio of true positives over the total number of results that are actually in that class, or in this case, the number of tweets correctly classified over the total number of tweets in that class. The F1-score is the harmonic mean of precision and recall, expressed as $F = \frac{2 \cdot P \cdot R}{P+R}$.

We present our baseline method results in Table 5.1, experiments with CNNs and LSTMs in Table 5.2, and experiments with the fastText model in Table 5.3.

Method	**Precision**	**Recall**	**F1**
BOW + Multinomial Classifier	0.826	0.825	0.825
TF-IDF + Multinomial Classifier	0.820	0.819	0.816
TF-IDF (ngram=2) + Multinomial Classifier	0.857	0.857	0.856
BOW + Balanced SVM	0.832	0.831	0.828
TF-IDF + Balanced SVM	0.849	0.848	0.848
TF-IDF (ngram=2) + Balanced SVM	0.879	0.877	0.878

Table 5.1: Baseline methods for text classification. Multinomial parameters ($\alpha = 1 \times 10^{-3}$). SVM parameters (loss=hinge, penalty=$L2$, $\alpha = 1 \times 10^{-5}$, iterations=5)

To compare our results against baseline methods, we implement bag-of-words models with multinomial naive Bayes classifiers and SVMs. For all tasks, we see a predictable trend of increased precision, recall, and F1 from increasing the n-gram count from 1 to 2. Of the two different classification methods, balanced SVMs outperform multinomial naive Bayes classifiers in all tasks, which is unsurprising because balanced SVMs are discriminative models, while naive Bayes models are generative models, and discriminative models do better than generative models on large datasets (Ng and Jordan 2002). The TF-IDF with 2-grams and balanced SVM score the highest on all metrics. Our baseline results are better than other papers performing text classification on the same dataset. Badjatiya et al. (2017) reports the highest baseline F1-score as 0.816 with TF-IDF and Balanced SVMs. Our results may be significantly higher because we run an extensive grid search over the model parameters.

	Method	Precision	Recall	F1
Baselines	CNN	0.822	0.770	0.746
	CNN with user features	0.826	0.782	0.762
	CNN (multi, pretrained embedding)	0.819	0.768	0.744
	CNN (multi, random embedding)	0.825	0.784	**0.765**
	LSTM (random embedding)	0.813	0.779	0.761
Embeddings	CNN (word2vec+dstormer unaligned)	0.827	0.790	0.774
	CNN (word2vec+dstormer aligned)	0.825	0.794	**0.780**
	CNN (word2vec+tweets unaligned)	0.820	0.782	0.764
	CNN (word2vec+tweets aligned)	0.825	0.794	**0.780**
	LSTM (dstormer aligned)	0.811	0.771	0.750
	LSTM (tweets unaligned)	0.818	0.786	0.770
User Features +Embeddings	CNN (word2vec+dstormer unaligned)	0.834	0.803	0.790
	CNN (word2vec+dstormer aligned)	0.837	0.811	**0.800**
	CNN (word2vec+tweets unaligned)	0.837	0.811	0.799
	CNN (word2vec+tweets aligned)	0.837	0.811	0.799
	LSTM (dstormer aligned)	0.815	0.769	0.742
	LSTM (tweets unaligned)	0.822	0.793	0.778
Badjatiya (2017)	CNN (random embedding)	0.813	0.816	0.814
	CNN (GloVe embedding)	0.839	0.840	0.839
	LSTM (random embedding)	0.805	0.804	0.804
	LSTM (GloVe embedding)	0.807	0.809	0.808

Table 5.2: Comparison of various methods for hate speech classification. "multi" refers to using 2 different channels for the CNN models, "dstormer" refers to using word embeddings trained on the DailyStormer and tweets by users with high centrality on Twitter, and "tweets" refers to word embeddings that were trained on the Waseem and Hovy (2016) dataset. For number of epochs, CNN is trained with 20, LSTMs with 30, and fastText with 40. All models were trained with the Adam optimizer (learning rate = 0.001), categorical cross entropy, batch size 128. CNNs have hidden dimension 100, dropout of 0.25 after embeddings and 0.5 after the convolutions. LSTMs have hidden dimension 100, 4 layers, and are bidirectional. Pair-wise significance tests are in Appendix 7.3.

The CNN and LSTM models are below state-of-the-art results, which are most recently reported by Pitsilis et al. (2018) to have a precision of 0.931, recall of 0.933, and F1 of 0.932. However, we find problems with Pitsilis et al. (2018)'s method of

appending additional user-level features. Their features t_{Na}, t_{Ra}, t_{Sa} are constructed from taking the set of tweets by a specific user, m_a, and finding subsets of those tweets that have been labeled as *neutral*, *racist*, and *sexist*. Their features are calculated as $t_{Na} = \frac{|m_{N,a}|}{|m_a|}, t_{Ra} = \frac{|m_{R,a}|}{|m_a|}, t_{Sa} = \frac{|m_{S,a}|}{|m_a|}$. For any particular tweet, they use information about the user's other tweets that have been human-labeled, which could have been sequentially posted after the tweet being classified. In addition, real-life text classification systems are unlikely to store a history of labeled user tweets, unless tweets manually reported by other users on the platform as hate speech. Our results for CNN and LSTM models are slightly lower than results reported by Badjatiya et al. (2017) in recall, while on par in precision. Our models are more conservative in labeling tweets as negative compared to other models, which may be due to the lack of time to specifically tune hyperparameters of each of our models.

With regards to experiments involving multiple word embeddings, *dstormer* refers to a word embedding model trained on articles from DailyStormer and tweets from influential white supremacists. *tweets* refers to a word embedding model trained on our Waseem and Hovy (2016) dataset. The aligned *dstormer* embedding performs better than the unaligned embedding, even though the vocabulary size for aligned *dstormer* is smaller than the unaligned embedding. Despite the smaller vocabulary size, this is unsurprising given that by aligning these context-specific word embeddings to *word2vec*, the backpropagation models are able to update these vectors in similar ways, and model weights more accurately characterize semantic differences between words. Aligned embeddings for *dstormer* and *tweets* perform similarly, possibly due to the smaller vocabulary size of each of these compared to the unaligned *tweets*. The unaligned word embeddings trained on the Waseem and Hovy (2016) dataset also perform better than the *dstormer* dataset, which is unsurprising because we are applying these word embeddings to the Waseem and Hovy (2016) dataset.

LSTMs perform worse than CNNs, likely due to the brevity of the tweets in the

dataset. LSTM models generally perform better when trained on longer sentences or data points, given their strength in referring back to the encoded memory from earlier training data. On the other hand, CNNs perform well on feature extraction, which may better suit hate speech classification given that perpetrators often use short phrases to express hate speech. The performance of LSTMs compared to CNNs on tweet-based text classification may be over-exaggerated in some papers, and the performance of different classification models for different types of text classification tasks should be further explored.

Our CNN models also perform better after appending the four user and tweet features. CNN with user features has a 0.016 ($p < 0.001$) increase in F1-score compared to the baseline CNN model. The F1-score for each of our models improves by approximately 0.02, with approximately equal gains across precision and recall. Even with word embeddings, appending the features continued to have significant gains. The F1-score increases by 0.035 ($p < 0.001$) in CNN multichannel models with features, compared to models without, both embedded with *word2vec* and the unaligned *tweets*. F1 scores also increased for CNN models with *word2vec* and *dstormer* ($p < 0.001$). However, our LSTM model did not improve after appending features before the hidden to label layer. While we are unsure of why CNN models outperform LSTM models with additional features, we hypothesize that because all of the information is encoded in the last LSTM hidden vector prior to appending the four features, the fully-connected layer may be overestimating the importance of the features relative to the hidden encoding.

Method	Precision	Recall	F1
fastText (random embeds)	0.850	0.849	0.848
fastText (dstormer + hate tweets)	0.855	0.851	0.852
fastText (random embeds, ngrams=2)	0.889	0.888	0.888
fastText (dstormer + hate tweets, ngrams=2)	0.888	0.888	0.888
fastText (dstormer + hate tweets, ngrams=3)	0.890	0.890	**0.890**
Badjatiya et al. (2017)'s fastText (GloVe)	0.828	0.831	0.829
Badjatiya et al. (2017)'s fastText (random embeds)	0.824	0.827	0.825

Table 5.3: Variations in the fastText model with different embeddings. Training used 40 epochs, learning rate of 0.1, and hidden dimension of 100. Pair-wise significance tests are in Appendix 7.3.

We also implement the fastText method used by Badjatiya et al. (2017) for text classification, since our CNN and LSTM models did not perform as well as the experiments reported by Badjatiya et al. (2017)'s paper. fastText creates sentence representations from the word vectors by averaging word embeddings and has been shown to outperform both convolutional and recurrent neural models. In Badjatiya's paper, changing from a CNN model to a fastText classification model improved F1-scores by 0.015. The results of our experiments using fastText are in Table 5.3. We did not append user features to the fastText models, given that the architecture of fastText depends on taking an average over the word embeddings in the sentence and creating a new paragraph embedding. The fastText architecture did not provide a feasible way to append additional features. We see significant gains in precision, recall and F1 scores when using fastText over our other neural network models, similar to the results reported by Joulin et al. (2016). The F1-score increases to 0.890 when using 3-grams, improving over our baseline fastText model by 0.042 ($p < 0.001$). There is no statistical differnece between using random embeddings and *dstormer* and *tweets*. Not only does fastText perform better than other neural network models, but it is also faster to train.

Chapter 6

Discussion and Conclusion

We show improvements in hate speech classification that result from using hate-speech specific word embeddings and user features. Our work builds upon a wealth of previous research on quantitative approaches to text classification, including work with word embeddings (Mikolov et al. 2013; Bojanowski et al. 2016; Pennington et al. 2014), neural networks for text classification (Y. Kim 2014), and simpler models for classification that outperform neural models (Joulin et al. 2016). We also build on qualitative work by researchers focused on the specifics of hate-speech classification and their findings regarding using task-specific word embeddings to capture code words in hate speech (Taylor et al. 2017; Magu et al. 2017). We extend these lines of work by comparing the above methods and introducing user features as additional data for hate speech classification, as well as being the first paper to apply task-specific hate speech embeddings to text classification.

6.1 Hate Speech Word Embeddings

Our results show that using word embeddings trained on the DailyStormer articles and tweets by white supremacist Twitter accounts identified by their centrality outperform models that used pre-trained word embeddings such as fastText and GloVe.

Models trained with hate speech-specific word embeddings also outperform random embeddings, so we can conclude that the increased F1 scores for hate speech-specific word embeddings did not come from randomly seeding the embeddings for vocabulary words that did not appear in the hate speech-specific embeddings, but rather from the better semantic representations of the words.

Use of hate speech-specific embeddings play a significant role in hate speech classification compared to other classification tasks such as sentiment analysis due to the number of code words developed by the hate speech community as offensive racial slurs. Pre-trained word embeddings are unable to capture the hate community's ascribed meaning to words such as "Googles," "Skittles," or "Skypes," which refer to black people, Muslims, and Jews, respectively. By training our own word-embeddings, we can better represent code words semantically in a hate-speech context.

6.2 User Features

The addition of four user features also improves our F1 scores by 2% across all CNN models. Characteristics of the users themselves may be indicative of whether their tweet is hate speech or not. While we did not explicitly attempt to calculate a measure of how likely a user would engage in hate speech, we decide to take as features the popularity of the tweet, the user's own influence (as measured by the number of followers), and the likelihood the user is exposed to varying opinions (as measured by the number of people they follow). When text classification algorithms are applied to platforms such as Twitter, these private companies might have better metrics to measure a user's likelihood to engage in hate speech than the public data available to us.

6.3 Future Work

Near the end of the research process, we discovered a framework similar to fastText that multiplies each word embeddings by a weighted average and modifies them through PCA (Arora et al. 2016). This model is highly reminiscent of TF-IDF. According to the researchers, the model beats deep neural networks such as CNNs and RNNs. Given the success of our fastText model over deep neural models, we hypothesize that Arora's model maybe able to outperform the state-of-the-art neural networks.

We would also conduct a more thorough comparative analysis of CNN and RNN architectures. A study by Yin, Kann, et al. (2017) tests the performance of CNNs, LSTMs, and GRUs, Gated Recurrent Units, (which have a simpler architecture than LSTMs and belong to the RNN class) on the Stanford Sentiment Treebank, and found that GRUs performed better than the other two architectures on longer sentences and had comparable performance with CNNs on shorter lengths of text. They generalize this finding to state that selection of a neural network architecture depends on how often the comprehension of global or long-range semantics is required. Given our application of neural networks to tweets, we want to test the performance of GRUs against our CNN architectures.

Additionally, we would extract more tweet-level and user-level features to further test the performance of appending additional features to the models. Due to privacy restrictions on Twitter, we only extract numerical values for each tweet and user. However, additional information could be gained from the user-generated biography, the tweet cascade patterns, and image analysis for tweets with images attached.

In addition, we have collected a dataset of 50K Twitter accounts with ties to self-identified white supremacists, each with up to 2,000 of their most recent tweets. Given more time, we would label these tweets through MTurk or CrowdFlower to conduct further hate speech classification on these tweets. There may be social net-

work features such as centrality, clustering, and small-world effects that improve hate speech classification because these accounts are retrieved using breadth-first search on a graph structure.

Finally, the effectiveness of task-specific word embeddings on hate speech classification tasks could be further explored. Because hate speech culprits adapt to disguise their speech from online spam and offensive language detection algorithms, automated detection systems are necessary to capture the semantics of these newly appropriated code words. We would also want to study if using hate-speech specific word embeddings causes more false positives on a set of clean tweets and how this effect can be mitigated.

6.4 Conclusion

In this work, we investigate the classification accuracy of various deep learning models against strong baseline techniques. We find that, contrary to previous papers' results, a new baseline technique that only averages word embeddings called fastText outperforms deep neural network techniques such as CNNs and LSTMs. We also engineer new features from tweet-level and user-level metadata, significantly improving classification accuracy on our neural models. Finally, we train task-specific word embeddings on known hate speech datasets to better correlate offensive code words with targeted groups. We show that a combination of feature engineering from metadata and using hate-speech specific word embeddings can have significant improvement in hate speech classification accuracy.

Chapter 7

Appendix

7.1 Saliency Maps

Additional visualizations of the weights of the first convolutional layer in determining the output of the classification.

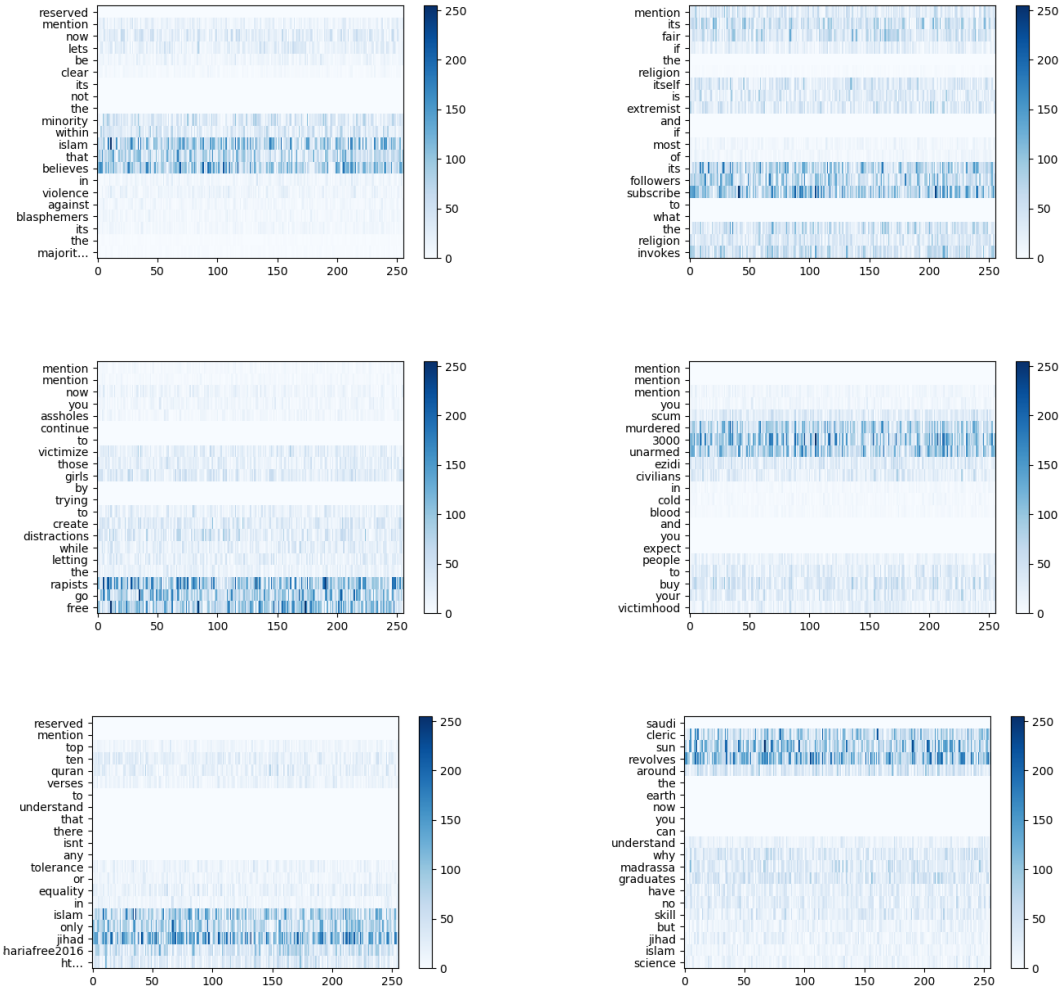


Figure 7.1: Additional Visualizations of Saliency Maps

7.2 Word Embeddings

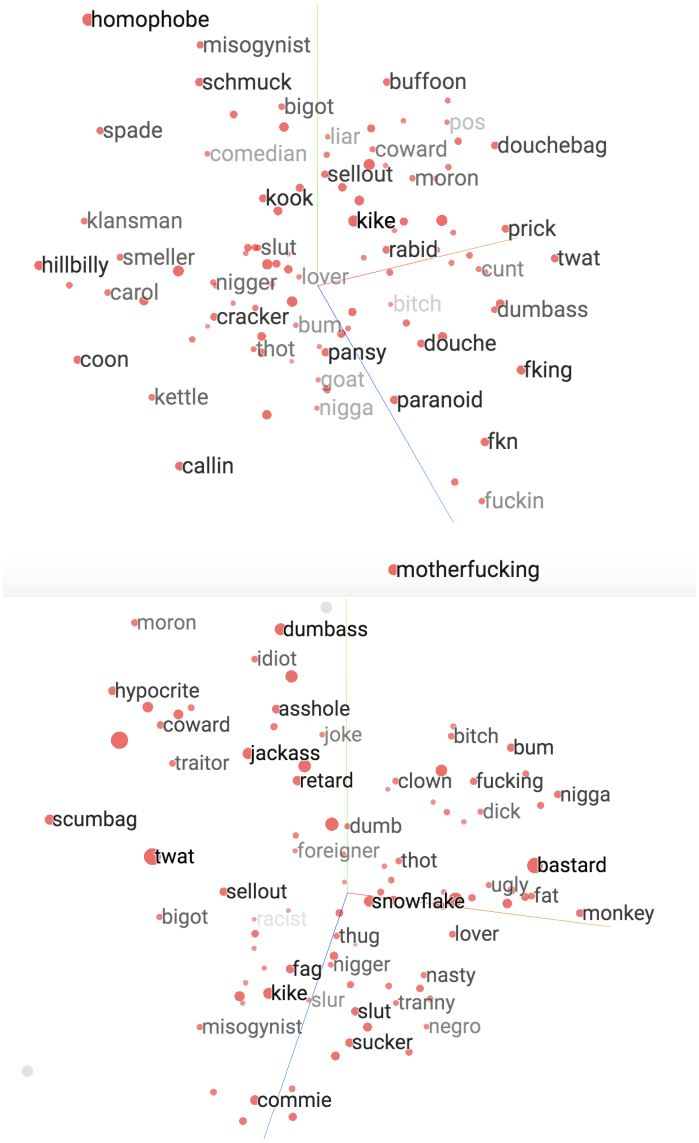


Figure 7.2: t-SNE plot of word embeddings showing the most common words to "kike" in (a) and "nigger" in (b).

7.3 Significance Tests

We conduct paired t-tests on each of the folds of the cross-validation to test whether two models were statistically significant. We choose the paired t-test over other statistical methods because we use the same train-test split in our folds every time,

and the models are trained on the same folds. The data from each of our folds across many different iterations also follows a Gaussian distribution, which allows us to apply the paired t-test.

We set a an α of 0.05, but because we are conducting mutliple pairwise comparisons on the same dataset, we divide α by m, the number of tests. For the first set of p-values, we look for a value of $p < \frac{\alpha}{m}$ where $m = 78$, so $p < 0.0006$. For the second set of comparisons, we look for a value $p < \frac{\alpha}{0.005}$.

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	XXX	**0.0001**	0.3063	0.0399	**0.0003**	**0.0000**	**0.0000**	0.0023	**0.0000**	**0.0000**	**0.0000**	**0.0000**	**0.0000**
2		XXX	**0.0002**	0.3726	0.0016	**0.0005**	**0.0000**	0.3411	**0.0000**	**0.0000**	**0.0000**	**0.0000**	**0.0000**
3			XXX	0.0362	**0.0002**	**0.0000**	**0.0000**	**0.0003**	**0.0000**	**0.0000**	**0.0000**	**0.0000**	**0.0000**
4				XXX	**0.0000**	0.2137	0.0547	0.4439	0.0698	**0.0006**	**0.0006**	0.0079	0.0048
5					XXX	0.0168	0.0511	0.0018	0.0623	0.3049	0.3049	0.2754	0.3311
6						XXX	0.0080	0.0418	0.0449	**0.0001**	**0.0001**	**0.0001**	**0.0002**
7							XXX	**0.0006**	0.4665	**0.0000**	**0.0000**	**0.0003**	0.0007
8								XXX	0.0011	**0.0000**	**0.0000**	**0.0000**	**0.0000**
9									XXX	**0.0004**	**0.0004**	0.0073	0.0034
10										XXX	XXX	0.0070	0.4236
11											XXX	0.0070	0.4236
12												XXX	0.0359
13													XXX

Table 7.1: P-values of comparing pairwise CNN models. Highlighted values are $p < 0.0006$.

Corresponding model numbers with model descriptions.

1. CNN
2. CNNFeatures
3. CNNMulti (fastText embeds)
4. CNNMulti (random embeds)
5. CNNMultiFeatures
6. CNNMulti with Embeddings (Google + DStormer)
7. CNNMulti with Embeddings (Google + DStormer Aligned)
8. CNNMulti with Embeddings (Google + My Unaligned)
9. CNNMulti with Embeddings (Google + MyAligned)

10. CNNMultiFeature with Embeddings(Google, MyAligned)
11. CNNMultiFeature with Embeddings(Google, My Unaligned)
12. CNNMultiFeature with Embeddings(Google, DStormer)
13. CNNMultiFeature with Embeddings(Google, DStormer Aligned)

	1	2	3	4	5
1	XXX	0.2159	**0.0000**	**0.0000**	**0.0000**
2		XXX	**0.0000**	**0.0000**	**0.0000**
3			XXX	0.3651	0.1899
4				XXX	0.1037
5					XXX

Table 7.2: P-values comparing fastText models pairwise. Highlighted values were $p < 0.005$.

Corresponding fastText model numbers with descriptions.

1. fastText (random embedding)
2. fastText (dstormer + hate speech users)
3. fastText random embeddings, ngram=2)
4. fastText (dstormer + hate speech users, ngrams=2)
5. fastText (dstormer + hate speech users, ngrams=3)

Bibliography

Arora, S., Liang, Y., & Ma, T. (2016). A simple but tough-to-beat baseline for sentence embeddings.

Badjatiya, P., Gupta, S., Gupta, M., & Varma, V. (2017). Deep learning for hate speech detection in tweets. In *Proceedings of the 26th international conference on world wide web companion* (pp. 759–760). International World Wide Web Conferences Steering Committee.

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, *3*(Feb), 1137–1155.

Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2016). Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*.

Burnap, P. & Williams, M. L. (2016). Us and them: Identifying cyber hate on twitter across multiple protected characteristics. *EPJ Data Science*, *5*(1), 11.

Chen, Y., Zhou, Y., Zhu, S., & Xu, H. (2012). Detecting offensive language in social media to protect adolescent online safety. In *Privacy, security, risk and trust (passat), 2012 international conference on and 2012 international confernece on social computing (socialcom)* (pp. 71–80). IEEE.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, *12*(Aug), 2493–2537.

Davidson, T., Warmsley, D., Macy, M., & Weber, I. (2017). Automated hate speech detection and the problem of offensive language. *arXiv preprint arXiv:1703.04009*.

de Sola Pool, I. & Kochen, M. (1978). Contacts and influence. *Social networks*, *1*(1), 5–51.

Dumais, S., Platt, J., Heckerman, D., & Sahami, M. (1998). Inductive learning algorithms and representations for text categorization. In *Proceedings of the seventh international conference on information and knowledge management* (pp. 148–155). ACM.

Gianfortoni, P., Adamson, D., & Rose, C. P. (2011). Modeling of stylistic variation in social media with stretchy patterns. In *Proceedings of the first workshop on algorithms and resources for modelling of dialects and language varieties* (pp. 49–59). Association for Computational Linguistics.

Girvan, M. & Newman, M. E. (2002). Community structure in social and biological networks. *Proceedings of the national academy of sciences*, *99*(12), 7821–7826.

Gitari, N. D., Zuping, Z., Damien, H., & Long, J. (2015). A lexicon-based approach for hate speech detection. *International Journal of Multimedia and Ubiquitous Engineering*, *10*(4), 215–230.

Granovetter, M. S. (1977). The strength of weak ties. In *Social networks* (pp. 347–367). Elsevier.

Hamilton, W. L., Leskovec, J., & Jurafsky, D. (2016). Diachronic word embeddings reveal statistical laws of semantic change. *arXiv preprint arXiv : 1605.09096*.

Hebb, D. O. (2005). *The organization of behavior: A neuropsychological theory*. Psychology Press.

Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Hubel, D. H. & Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, *195*(1), 215–243.

Joulin, A., Grave, E., Bojanowski, P., & Mikolov, T. (2016). Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*.

Kim, S.-B., Han, K.-S., Rim, H.-C., & Myaeng, S. H. (2006). Some effective techniques for naive bayes text classification. *IEEE transactions on knowledge and data engineering*, *18*(11), 1457–1466.

Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kumar, R., Novak, J., & Tomkins, A. (2010). Structure and evolution of online social networks. In *Link mining: Models, algorithms, and applications* (pp. 337–357). Springer.

Kwok, I. & Wang, Y. (2013). Locate the hate: Detecting tweets against blacks. In *Aaai*.

Li, J., Chen, X., Hovy, E., & Jurafsky, D. (2015). Visualizing and understanding neural models in nlp. *arXiv preprint arXiv:1506.01066*.

Magu, R., Joshi, K., & Luo, J. (2017). Detecting the hate code on social media. *arXiv preprint arXiv:1703.05443*.

Mahmud, A., Ahmed, K. Z., & Khan, M. (2008). Detecting flames and insults in text.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Neidig, H. (2017). Twitter launches hate speech crackdown. Retrieved December 18, 2017, from http://thehill.com/policy/technology/365424-twitter-to-begin-enforcing-new-hate-speech-rules

Ng, A. Y. & Jordan, M. I. (2002). On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Advances in neural information processing systems* (pp. 841–848).

Nobata, C., Tetreault, J., Thomas, A., Mehdad, Y., & Chang, Y. (2016). Abusive language detection in online user content. In *Proceedings of the 25th international conference on world wide web* (pp. 145–153). International World Wide Web Conferences Steering Committee.

Papegnies, E., Labatut, V., Dufour, R., & Linares, G. (2017). Impact of content features for automatic online abuse detection. *arXiv preprint arXiv:1704.03289*.

Pennington, J., Socher, R., & Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (emnlp)* (pp. 1532–1543).

Pitsilis, G. K., Ramampiaro, H., & Langseth, H. (2018). Detecting offensive language in tweets using deep learning. *arXiv preprint arXiv : 1801.04433*.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). *Learning internal representations by error propagation*. California Univ San Diego La Jolla Inst for Cognitive Science.

Simonyan, K., Vedaldi, A., & Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*.

Spertus, E. (1997). Smokey: Automatic recognition of hostile messages. In *Aaai / iaai* (pp. 1058–1065).

Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

Taylor, J., Peignon, M., & Chen, Y.-S. (2017). Surfacing contextual hate speech words within social media. *arXiv preprint arXiv:1711.10093*.

Thomasson, E. (2017). German cabinet agrees to fine social media over hate speech. Retrieved April 5, 2017, from https://uk.reuters.com/article/uk-germany-hatecrime-facebook/german-cabinet-agrees-to-fine-social-media-over-hate-speech-idUKKBN1771FK

Volokh, E. (2015). No, there's no "hate speech" exception to the first amendment. Retrieved May 7, 2015, from https://www.washingtonpost.com/news/volokh-conspiracy/wp/2015/05/07/no-theres-no-hate-speech-exception-to-the-first-amendment/?utm_term=.09f47e030b0c

Volokh, E. (2017). Supreme court unanimously reaffirms: There is no 'hate speech' exception to the first amendment. Retrieved June 19, 2017, from https://www.washingtonpost.com/news/volokh-conspiracy/wp/2017/06/19/supreme-court-unanimously-reaffirms-there-is-no-hate-speech-exception-to-the-first-amendment/?utm_term=.d928d5b736ba

Warner, W. & Hirschberg, J. (2012). Detecting hate speech on the world wide web. In *Proceedings of the second workshop on language in social media* (pp. 19–26). Association for Computational Linguistics.

Waseem, Z. (2016). 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* (pp. 138–142).

Waseem, Z. & Hovy, D. (2016). Hateful symbols or hateful people? predictive features for hate speech detection on twitter. In *Proceedings of the naacl student research workshop* (pp. 88–93). San Diego, California: Association for Computational Linguistics. Retrieved from http://www.aclweb.org/anthology/N16-2013

Xiang, G., Fan, B., Wang, L., Hong, J., & Rose, C. (2012). 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* (pp. 1980–1984). ACM.

Yin, W., Kann, K., Yu, M., & Schutze, H. (2017). Comparative study of cnn and rnn for natural language processing. *arXiv preprint arXiv:1702.01923*.

Yin, W. & Schutze, H. (2015). Learning meta-embeddings by using ensembles of embedding sets. *arXiv preprint arXiv:1508.04257*.