Text Embedding Methods on Different Levels for Tweets Authorship Attribution

A PROJECT SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Zhenduo Wang

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Professor Yang Li, Professor Ted Pedersen

May 2018

© Zhenduo Wang 2018

Acknowledgements

Many people supported or helped me on this project, and they all deserve my gratitude. First, I appreciate my advisors Professor Yang Li and Ted Pedersen for their guidance on my graduate study. It was my pleasure to work with such knowledgeable, passionate and patient advisors like them. Then I would like to thank my parents. I could not be who I am or achieve what I have today without their unfailing support in my life. I should also thank Professor Richard Green for serving on my committee and his patience on my questions. Finally, thank all the faculty and staff members in UMD Mathematical Science department; you have given me a welcoming atmosphere during my two-year-study here.

Abstract

Text classification tasks on Twitter corpora are challenging because the language people use on Twitter is very unstable and unpredictable. These tasks are also meaningful since Twitter has become a platform where massive information exchanging and interaction is happening every moment. Our work aims to solve one such kind of problem, Tweets Authorship Attribution. Traditional methods for text classification use classifiers based on manually selected features while our system uses text embedding as low-level features. According to the level of text embedding, our system can be categorized as character, sub-word, word, or document based. In order to find more indicative features, we use Convolutional Neural Networks to extract higher-level features from the basic text representations. With the higher-level features, we use neural network classifier to predict the authorship of tweets. We conduct experiments on the dataset from the work by Schwartz et. al using text embedding methods on different levels and compare our experiment results with state-of-art methods. Our results we get show that convolutional neural network systems based on text embeddings are accurate.

Contents

Contents iii

List of Tables v

List of Figures vi

1 Introduction 1

2 Background 3

2.1 Natural Language Processing and Text Classification 3

2.2 Terminology 5

2.2.1 N-grams 5

2.2.2 Word Co-occurrence Matrix 5

2.3 Explicit Feature Engineering 6

2.4 Implicit Feature Engineering 8

2.4.1 Word Embedding 8

2.4.2 Feature Extraction, Convolutional Neural Network 12

2.5 Embedding on other levels 15

2.5.1 Sub-word Embedding 15

2.5.2 Character Embedding 16

2.5.3 Document Embedding 18
2.6 Training Strategies for Embedding 19
2.6.1 Pre-trained Embedding 19
2.6.2 Joint Training 20
2.7 Data set 20

3 System Description 21

3.1 Text Representation 22
3.1.1 Word Embedding, word2vec 22
3.1.2 Document Embedding, doc2vec 28
3.1.3 Character Embedding 32
3.1.4 Sub-word Segmentation and Embedding 33
3.2 Feature Extraction 35
3.3 Classification 38
3.3.1 Other Classifiers 38
3.3.2 Embedding as Dimension Reduction Method 39

4 Experimental Results 42

4.1 Experiment Settings 42
4.2 Hyperparameter Details 42
4.3 Results and Evaluation 43

5 Conclusions and Future Work 48

5.1 Conclusions 48
5.2 Future Work 48

References 52

List of Tables

3.1 Accuracy for keeping different % of importance of feature set for 50 authors with 1,000 tweets each 41

4.1 Network parameter settings 43

4.2 Accuracy for different # of authors with 200 tweets each 45

4.3 Most frequent 50 sub-words in 200,000 tweets from 1,000 authors . . 46

4.4 Accuracy for 50 authors with 1,000 tweets each 47

List of Figures

2.1 Neural Network 13

3.1 System Overview 21

3.2 Structure of Continuous Bag of Words 24

3.3 Structure of Skip-Gram 27

3.4 Structure of Distributed Memory 30

3.5 Multi-channel Convolution Layer 36

3.6 1-D Convolution 37

3.7 Max-Pooling 37

3.8 % of importance of features after PCA 40

4.1 Accuracy versus training epochs for sub-word system 44

4.2 Accuracy versus training epochs for character system 44

5.1 Frequency and Cumulative distribution function of word and sub-word on Sentiment Analysis dataset (2) 49

5.2 Frequency and Cumulative distribution function of word and sub-word on Tweet Authorship Attribution dataset (50) 50

1 Introduction

Being offered the opportunity to own work is in most cases a pleasure, but this is not always true. Occasionally, there are cases where people would rather hide due to the danger of being identified as the author of their ideas, discoveries or, perhaps, protests. Usually it is fine and we may feel better not to know the authorship, if that is the author's will. But somehow not knowing the true authorship could also cause a problem. Think about the rise of Twitterbots (which are Artificial Intelligent machine programs that can automatically generate tweets) and the potential problems that could arise if we cannot distinguish a Twitterbot from a human. One example of such Twitterbots is *DeepDrumpf*, which is created to mimic the twittering style of Mr. Donald Trump.

The work done by Mosteller and Wallace (1964) about the authorship of disputed Federalist Papers marked the start of the study of authorship attribution. [1] Years have passed, and these articles have been studied thoroughly enough so the complete list of authorship can now be found on Wikipedia. Many such authorship attribution work has been done, especially for tasks with long documents like identifying anonymous articles. If we take a look at these articles, we will find that most of them are texts with thousands of words. Compared to these long texts, twitter texts are relatively short and irregular. If we want to solve the authorship attribution task on Twitter, we have to consider it as a different task.

In this work, we build a network system based on text embeddings for Tweets authorship attribution problem. We use a text embeddings and feature extraction

strategy instead of feature engineering. We conduct experiments for our system on different levels of text embeddings and compare their performances with state-of-the-art methods. Among our systems built on different embeddings, the character based system achieve the best result. Under the same condition, it improves on other systems significantly. We also study the document embedding system thoroughly and give analyses of its results.

2 Background

In this project, we design text classification system using convolutional neural network based on text embeddings. In this chapter, we will give explanation for the background and introduce some related works to our project. We will talk about the definition of text classification task, the idea of text embedding and the concept of implicit feature engineering. We will also give details about dataset and training strategies.

2.1 Natural Language Processing and Text Classification

Our work is in the field of Natural Language Processing (NLP), which is a subfield of Artificial Intelligence. NLP focuses on the study of the interaction between machine and human natural language. NLP tasks include word sense disambiguation, machine translation, language modeling, syntactic parsing, etc. NLP is closely related to statistics since most NLP systems study the statistics of corpora and rely on the most significant ones. Statistical methodologies are also commonly seen as the essence of NLP systems.

Text classification has long been a problem in Natural Language Processing (NLP). Text classification tasks are usually to label a text with a class label, according to its attributes such as sentiment positivity, authorship or whether it is humor. For

example, authorship attribution (AA) is one instance of the problems that need to be solved. Authorship attribution can be seen as a general classification problem. Given an unlabeled set of texts and a set of names of candidate authors, assign each text with an author name. Depending on whether the set of author is determinant, authorship attribution can be either closed (the set of author is determinant) or open (indeterminate).

Many previous works have been done in authorship attribution, especially for tasks with long documents such as identifying anonymous articles [1]. This work by Mosteller and Wallace about the authorship of the disputed Federalist Papers marked the start of the study of authorship attribution. The difference between their work and ours is that the former used inference methods and was done on long text while ours uses learning methods and is done on tweets which are relatively short.

Authorship attribution on Twitter is a meaningful and challenging task. It is meaningful since twitter has become a platform where massive information exchanging and interaction is happening every moment. Authorship attribution on tweets is challenging because Twitter language is so different from formal written languages. Twitter language is very unstable and unpredictable. Various expressions are invented and abandoned regularly there. Hence we can hardly generalize consistent and convincing knowledge about Twitter corpora. This limits the performance of most traditional methods since they are heavily dependent on the knowledge base of language. Therefore authorship attribution on Twitter is still a problem to be solved.

Other text classification problems include sentiment analysis, emoji prediction and figurative language (humor, irony, sarcasm, etc.) detection. They are very similar to authorship attribution problem, but they are usually binary classification problems. Given an unlabeled set of text, assign each text with a label of 1 or 0, indicating whether the text tends to be positive/ironic or negative/not ironic, etc.

2.2 Terminology

2.2.1 N-grams

N-gram refers to a sequence of n consecutive objects, depending on what n-gram is being used. 1-gram is usually called unigram, 2-gram bigram, 3-gram trigram, then four-gram, five-gram and so on. For example, *"I play"* is a **word bigram** in the sentence *"I play a star in today's play."* So is *"play a"*. We can also say *"I p"* is a **character trigram** in the same sentence. Note that we count the space as a character in this case, but we do not have to. The statistics of N-grams are widely used in NLP problems.

2.2.2 Word Co-occurrence Matrix

If two words appear close enough so that they are inside a context window of a certain size, then we call this relationship word co-occurrence. Note that this relationship is symmetric. A word co-occurrence matrix is a logical data structure to store word co-occurrence information. Here is an example. Suppose our corpus consists of only one sentence, *"I play a star in today's play."* Let the context window size be 4, which means 4 consecutive words are considered to be in one context.

Word	I	play	a	star	in	today's
I	0	1	1	1	0	0
play	1	0	1	2	2	1
a	1	1	0	1	1	1
star	1	2	1	0	1	1
in	0	2	1	1	0	1
today's	0	1	1	1	1	0

As shown above, the matrix, called C, is an n by n matrix where n is the vocabulary size of the corpus ($n = 6$ in this case). The matrix element C_{ij} denotes the frequency of word co-occurrence of word pair w_i and w_j. By this definition, we can assert that the word co-occurrence matrix is a symmetric matrix because of the symmetric property of co-occurrence. For convenience, the diagonal elements are often set as 0, since we care more about the co-occurrence relationships between different words.

2.3 Explicit Feature Engineering

Almost all classification systems end up with a classifier. An important step before in creating a classifier is to find variables (a.k.a. features) and to transform the text into a numerical representation of these features. The power of a classification system is largely determined by the feature selection step. Features that are indicative and easy to measure are necessary for the classification system to be accurate. After we represent the text with features, the original text is then discarded and will never be used in later steps. This means we must put all our faith in the features. For example, the number of positive/negative words used is usually considered as a good feature for sentiment analysis while the length of text may not be equally powerful. As an example, consider the sentiment analysis task consisting of two texts - *"I like this movie! Its plot is interesting!"* - *"This book is too abstruse for a student."*. If we can use only one feature for representation - the number of positive/negative words, then we represent the first text with a integer '2' because of the *"like"* and *"interesting"* used in it, and similarly '-1' for the second because of appearance of *"abstruse"*. With this feature, we can simply build a linear classifier which sets '0' as the threshold to decide the positivity of the sentiment. But if the only one feature we can choose is

the length of text, then we will represent the two texts above by integers '8' and '8'. In this case, they cannot be classified differently with any classifier. Note that we discuss the power of a feature in general because it is indeed possible to find cases for almost any feature where it is not good.

Usually the selected features are "meaningful" so that each feature is well defined and its meaning is clearly known by users of the system. A very convenient and popular way to select features is the Bag-of-Words (BOW) model. The simplest BOW model counts the frequency of every word in the vocabulary, from which it generates a vector representation. To make this feature set more efficient, the vector is usually truncated to a shorter size by setting a threshold on the minimum of word counts, and removing function words like article "the" and preposition "to" from the set. Many works combine BOW features with different classifiers such as SVM or logistic regression for the classification system. [2][3]

Apart from BOW features, some tasks require more precise features depending on the complexity of the task, and sometimes features are manually defined. Manually defined features can be the count of n-grams or part-of-speech n-grams. Generally these complex features are also based on statistics of the corpus. The advantage of this feature strategy is that we know exactly the features we choose and why we choose them. This enables us to understand the task and the strength and weakness of our system. However it does have disadvantages. Selecting a good feature set usually requires expert linguistic knowledge and a thorough study of the task. Also a good feature set for one task may not generalize to other tasks.

2.4 Implicit Feature Engineering

Just like we have many different classifiers, explicit feature selection is not the only feature engineering strategy.

2.4.1 Word Embedding

The idea of implicit feature engineering is based on word embedding. Word embedding is one kind of Vector Space Model, which uses vectors as bag of features to represent a document or a query. The features are usually frequencies of terms in the documents or other functions with frequency as a factor. The best part of this model is that the problem to be solved now has a view from the embedded vector space, and we can use measures from it. For example, we can use the cosine of the angle between two vectors to measure the similarity between two documents. Bag of Words feature is one example of Vector Space Model. But it is an embedding for the whole text, rather than a single word.

A simple instance of embedding is one-hot embedding, where each word is mapped to a sparse vector with length exactly equal to the vocabulary size. Each dimension of the vector space can be seen as the binary feature of some word, denoting if it is that word. A word vector has only one nonzero element in "its own" dimension and zeros in all other dimensions. This is very intuitive and sparsity is usually a good property. However, one-hot embedding yields a dimension disaster as the vocabulary size grows. Additionally, we cannot compute word similarity or do many other semantic analyses using this embedding method, because any pair of two words will be perpendicular to each other and thus totally independent under this representation. Because of all these problems, it is rarely useful in real problems.

Nowadays, a popular family of word embedding method is called continuous word

representation. In continuous word representation methods, words are mapped to non-zeros dense vectors in a smaller vector space with fewer dimensions. Compared with one-hot embedding, continuous representation methods can be seen as dimensionality reduction methods. Useful information is gleaned together and redundant dimensions are removed. Continuous word representation methods are so powerful and widely accepted that the concept of word embedding is almost the same as continuous word representation methods. We will use word embedding to refer to continuous word representation methods for the rest of this dissertation.

Normally, we need to give definition to all the dimensions in a Vector Space Model so that we are able to decide the exact vector representation for each word. However, word embedding is different. We do not define the dimensions, instead we simply assume the number of dimensions. That means we only choose how many dimensions we want the vector space to have. Then there come two questions. How can we determine the value on each dimension if we do not know its meaning? How do we determine the length?

The answer to the first question is that we cannot decide the exact value for each dimension, since we cannot generate rules or scoring functions for them if we do not know what they are. But we do not have to set these values by ourselves. The strategy is to initialize them and optimize them numerically in a specific task. Since these features are all abstract and obtained by optimization, it should be safe to say that the number of dimensions and the power of the model have positive correlation. Hence determining the number of dimensions becomes balancing the trade-off between model capacity and computational cost.

Here is a brief example of word embedding. The embedding dimension is set as 5.

Word	Embedding vector
I	[0.3, 0.4, 0.1, 0.4, 0.9]
play	[0.7, 0.6, 0.5, 0.1, 0.2]
a	[0.1, 0.3, 0.2, 0.7, 0.9]
star	[0.8, 0.8, 0.9, 0.2, 0.1]
...	...

This brief example shows what the embedding vectors will be like. As shown in the table, all the words are mapped to vectors with equal length. We are not able to tell the exact meaning of each dimension, but we can assume that they all have implicit meanings and their values are already optimized.

Word2vec

One popular word embedding model is the word2vec. In 2013, Mikolov showed the power of word2vec and it soon became one of the mainstream word embedding approaches in NLP. [4] One year later, the same group expanded word2vec into distributed document representation (doc2vec), which could also give a whole document a vector representation. Both models are now widely used as text representation methods in NLP tasks.

The word2vec model shows its capacity in solving the word analogy problem. A word analogy problem is to find the target word d from word c based on a parallel relationship from word a to word b. A word analogy problem could be semantic or syntactic. A typical semantic word analogy problem could be "Man is to King as Woman is to...". A typical syntactic word analogy problem could be "dance is to dancing as fly is to...". As mentioned earlier, word embedding as a Vector Space Model can map word to vectors. Hence the word analogy problems can be solved by finding the nearest word vector to the vector of $b - a + c$. In the word2vec paper, there

is a famous formula $v(Woman) - v(Man) + v(King) = v(Queen)$, which explains how the word analogy problem is solved. First, we use the embedding vectors of man, woman and king to calculate $v(Woman) - v(Man) + v(King)$. Then we compute the cosine value between the resulting vector and the embedding vector for each word. Finally we rank all the cosine values and find the closest (largest cosine value) word "queen" as the answer to the word analogy problem. The success of word2vec model on this task shows that the embedding vectors indeed inherit semantic information from words.

The composition property is believed to be the key factor for the success of word embedding in word analogy. One previous work tries to explain this property in a mathematical way. In this paper, Gittens et al. studies the Skip-Gram model in two aspects [5]. First, they make some assumptions about the distribution of word in corpus. Based on these assumptions, they use mathematical and statistical proof to provide a theoretical understanding of word composition property. Then, they explain the relationship between Skip-gram model and the Sufficient Dimensionality Reduction (SDR), which is a technique for reducing dimension of matrices. A shorter feature list that keeps the most valuable information will greatly reduce the classification cost and increase accuracy. The idea to extract needed word embedding information from word co-occurrence matrix in SDR and Skip-gram are very similar. But their training objectives are slightly different. SDR fits the entire probability mass function of word co-occurrence $P(x_1, x_2)$, while Skip-Gram fits conditional distributions $P(x_1|x_2)$. Fitting the entire distribution is more precise, strictly speaking. But fitting conditional distributions will greatly. reduce the cost and give similar result. From this point of view, Skip-gram is a heuristic method for SDR.

2.4.2 Feature Extraction, Convolutional Neural Network

In NLP tasks, n-grams are often chosen as default features for the representation of word collocation in texts. But this could be difficult if we have no sense of what kind(s) of n-grams could be useful for a certain problem. Recently, a trend [6] [7] for solving text classification tasks is to break text into the smallest units and build a representation for them. Then the scheme uses complex neural networks such as convolutional neural network (CNN) or recurrent neural network (RNN) to find abstract features upon the representations. This system configuration takes advantage of the feature extraction ability of complex neural networks. The work [8] tries to explain the mechanism of character-entry-CNN by showing that the features extracted by CNN are specific n-grams. We believe this scheme could be very useful for finding features for the classification when we have little knowledge of the corpus. We consider a Twitter corpus as such a case.

Neural networks (a.k.a. artificial neural networks) are a family of configurations used in machine learning. Its most interesting ability is that it can approximate any function. Because of this, it can be used to fit any classification or regression function. Imagine a classifier as a function which takes the features as input and outputs a number as a label which indicates the class it belongs to. What if we do not know what the desired classifier looks like but we can approximate it with a very simple model? Yes, this is exactly what a neural network is!

Neural network gets its name because it is a hierachical network of neurons, which are actually functions in particular forms. A neural network is actually a big function F that consists of smaller functions $f_1, f_2 ... f_n$ as neurons, e.g., $F = f_n(f_1, f_2, ...)$.

A neuron function $f = \sigma(g(\vec{x}))$ is usually divided into two parts, the linear function $g(\vec{x})$ and the activation function $\sigma(y)$. The linear function is always defined as $g(\vec{x}) =$

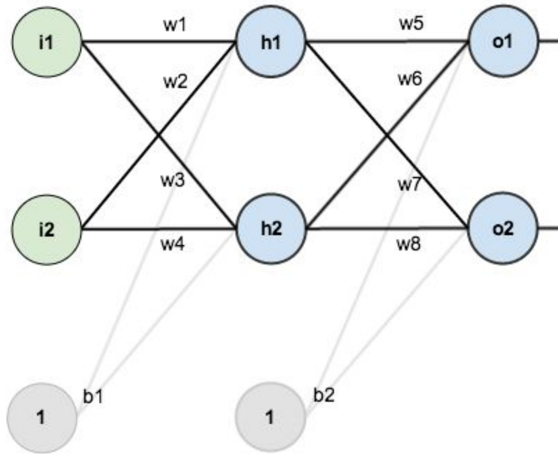

Figure 2.1: Neural Network

$\omega^T\vec{x} + b$, which is the general representation of a linear map from $\mathcal{R}^n \rightarrow \mathcal{R}$, given $\vec{x} \in \mathcal{R}^n$. There are several commonly used activation functions such as tanh or the sigmoid function $\sigma(y) = (1 + e^{-y})^{-1}$. If we do not use activation functions, then the neuron is simply a linear function and the whole network will also be linear because it is composed of only linear functions, which means it can only approximate linear functions. So, having an activation function in a neuron is important for a neural network. From the definition, we know that each neuron outputs a single value. Generally, if we put the outputs of many neurons together, we will get a vector, and this vector can be again used as input for subsequent neurons. This makes the hierachical structure of a neural network. A layer is the set of neurons which work together on the same input vector and contribute to the same vector which will be used by the next layer.

Now thousands or even millions of neurons work together, approximating the desired function. Since neural network has already given us the structure, in order to approximate the desired function with neural network, we only need to optimize the parameters $\vec{\theta} = (W, B)$ (see Fig 2.1) in the network. This is an optimization

problem with the form $F(\vec{\theta}, \vec{x})$. We use the data with pairs of $(\vec{x}, y)$ to optimize the parameters $\vec{\theta}$. The optimization strategy is usually to define a loss function to use gradient strategies, such as stochastic gradient descent [9]. Commonly chosen loss functions include squared error between the current network output and the data y value, $L = (y - y_0)^2$. With squared error loss function, we can compute the gradients of the parameters of the last layer. Because the whole network is a composition of functions, we can apply the chain rule to all other parameters and compute their gradients layer by layer, from the last to the beginning. Since the gradients are computed in reverse order, this process is often referred to back propagation. The optimization of network parameters is usually called the training of the network. And after we finish the optimization, we finish the approximation of the desired function. Then our network is ready to go! This is just the simplest structure of a neural network and is called fully connected network. Nowadays there are many kinds of neural networks with more complex structures.

A complete system with implicit feature engineering strategy which uses word embedding as text representation and convolutionl neural network as feature extractor solves the text classification task in this way. For example, suppose we want to know if the tweet *"I play a star in today's play"* (7 words) is ironic. First we use a large corpus such as Wikipedia to train an embedding model. This model can map any word to a vector with a fixed length. Then we use this model as a look-up table and transform all the words in the tweet to word vectors w_1 through w_7. After that, we use convolutional neural networks to generate features from the word embedding vector sequence. Then we use a classification layer afterwards. In conclusion, this system takes as input pure text, then after word embedding and higher-level feature extraction, feeds the features into classifier and produces the classification result. We will explain all these in details in Chapter 3.

2.5 Embedding on other levels

The idea of mapping words to word vectors is great, but it still can be improved by using embedding at a different level. Besides word embedding methods, people also extend the study to lower or higher level embeddings. On decomposing a word and analyzing its morphology information, there are works on sub-word or character embeddings (e.g. [10] [7]). For building a general model for one entire sentence or document, there are works on document embedding [11].

2.5.1 Sub-word Embedding

Some previous works [12] [13] [14] point out that including morphological information may help word embedding. In these papers, the authors argue that there is one disadvantage of the word2vec model. Because it assigns completely different vectors to different word types. This fails to include word morphological information in word embedding. This means words with similar morphemes, for example $worker$ and $working$, should share some embedding parameters but they don't. These works build morphological word-morpheme mapping based on existing dictionaries and improve the original word embedding in different tasks to different degrees.

Among all the sub-word embedding methods, fasttext [10] designed by Facebook Research is the widely accepted since it is the most time efficient embedding method for text classification tasks. In the paper, Mikolov et al. introduce a new way to represent a word not only by itself but also with its n-grams as sub-words. The model assigns each character n-gram a vector and maps the sub-word embedding to classes numbered of 1 to N with a hash function, where N is the number of hash classes. A word is represented by its index in the vocabulary together with its sub-word vector classes. The authors train the word prediction model with sub-word

based word vectors and it learns a better word representation. The hash function is designed to reduce the time cost, and it does so significantly. The authors show that this model has better performances in several tasks including word analogy and word similarity.

Again, consider the irony detection task for the tweet *"The workers are working in the workshop."*. Recall that in word2vec model, we represent the tweet with a sequence of word vectors w_1 through w_7. If we use a sub-word embedding model, instead of a word representation, we use sub-word representation. We segment the words in this tweet into sub-words *"The work ers are work ing in the work shop"* (10 sub-words). Then we represent the tweet with a sequence of sub-word vectors s_1 through s_{10}. By doing this, we share parameters among the representations of *worker*, *working* and *workshop*. As the vocabulary grows larger, the advantage of sharing parameters will become more and more significant. To solve text classification tasks with sub-word embeddings, the other parts of higher-level feature extraction and classification are very similar to that in word embeddings.

In another work [15], Mikolov et al. introduce a new text classification method using fasttext. They use an embedding and classifier model. To represent a document, the model uses as input the average of fasttext word embedding of words contained in the document. Since the fasttex embedding is time efficient, the whole model outperforms all other state-of-art method in terms of time cost.

2.5.2 Character Embedding

We also replace a word embeddings with character embeddings. Consider the following example: *"The workers are working in the workshop."* In character embedding, we segment all the words in this tweet into characters. This tweet will become

a sequence of 33 characters, not including spaces and the period. Then we represent the tweet with a sequence of character vectors c_1 through c_{33}. The advantage of using character embeddings is that the character vocabulary is extremely small compared to word or sub-word and can largely reduce the cost of embedding. But there is no free lunch. One character does not contain much information, just like one pixel in a graph. Hence we need more complicated feature extraction models for character embedding to get equally good features for the whole text. This is where the cost is finally paid.

There are several works discussing the design of character representations [6] [8]. In previous work [7], shrestha et al. use character-level embedding as input and both convolutional neural network (CNN) and recurrent neural network (RNN) to extract local and long-term dependencies based on the embeddings. This model is shown to be reliable in many different text classification tasks.

The combination of two neural networks in this paper is worth noting. Both CNN and RNN are used in previous work for feature extraction. CNN is indeed able of capturing dependencies. But there is a problem with CNN. In these works, the convolution kernels are chosen to be small. We will talk about the meaning of convolution kernel later, but one can assume that kernel is an abstract context window. If a kernel is small, the context window for possible dependencies will be small. In order to include long-term dependencies, researchers most build a very deep network which increase the computational cost. In particular, when we embed on character level, the length of dependency can be very long. But with RNN, dependencies of any length can be captured in only one RNN layer. So using CNN and RNN together can capture both local and long-term dependencies. Such a complex and deep network may work well on long texts, but not neccessarily be good on tweets.

2.5.3 Document Embedding

Document embedding is different from the embedding methods we discussed previously. These methods assign each different entity at their levels a vector, and then represent the whole document as a sequence of vectors. Document vectors do not do this. Document embedding builds a direct document to vector mapping rather than a combination of embeddings on lower levels. For example, for the tweet *I play a star in today's play* (7 words), document embedding represent this tweet with one single vector d. The advantage of this method is that we can use the document vector d as the features of the document and feed that to a classifier directly. In [15], Mikolov et al. give an interesting interpretation of document embedding. That is, document embedding is a method which saves the intermediate document vector for reuse. It uses non-trivial constructions to make the document vector unique, and does more than simply averaging word vectors.

A representative document embedding method is doc2vec in [11]. The idea of doc2vec model is based on word2vec model. The doc2vec model initializes random vectors to represent words or documents and then optimize the representation by fitting the document-word co-occurrence information. This is very similar to the idea of word2vec.

Since the doc2vec paper was published, there have been many discussions about its validity. There are two frequently asked questions. One is that whether doc2vec really outperforms other baselines on other data sets. The other one is that out of the two models in the doc2vec paper, which one is better? However, many researchers only reported their experiment result as being worse than that of the doc2vec paper, without explanation. Rigorous academic discussion about these issues are missing. The work by Lau and Baldwin [16], first reported that doc2vec shows its advantage

in text-level tasks with an empirical experimented evaluation. They conduct experiments including a forum task duplication test and sentiment analysis. They compare doc2vec embedding with component-wise mean of word2vec as document embedding and ngram maximum likelihood estimation baseline. In both tasks, doc2vec scores higher accuracies overall. This work also compares the two models in the doc2vec paper and reports that the Distributed Bag of Words (DBOW) model shows higher accuracy in both tasks. Although it surprisingly (or not?) contradicts the conclusion of the doc2vec paper, it is the first to provide rigorous experimental results and analyses.

Doc2vec model provides a new approach to treat a document as one entity, which can be seen as a combination of text representation and feature extraction. It is shown that many text classification problems can be solved by using doc2vec and choosing classifier carefully. That is the reason we try it for tweet authorship attribution.

2.6 Training Strategies for Embedding

Two strategies for training the character, sub-word, word or document embeddings are seen in previous works.

2.6.1 Pre-trained Embedding

The first strategy, used in word2vec, doc2vec, GloVe [17] and fasttext, is to pre-train the embedding individually and independently from the task where it will be used. Usually the pre-training task is to fit word co-occurrence statistics of some large and regular corpus like Wikipedia. The advantage of this strategy is that the embedding is meaningful since it roots from the co-occurrence information, and it can be reused in different tasks once trained. The pre-trained strategy generally shows

better performance than joint training strategy. One possible explanation is that it can be seen as multitask learning [18].

2.6.2 Joint Training

The second strategy is to add an embedding layer in the network, and then jointly train the embedding parameters and network parameters to optimize the classification target function. The advantage of this strategy is that this embedding is trained specifically for the task and thus can be more efficient. But on the other hand, this embedding may not be meaningful without the task.

We use the joint training strategy in our system. As said earlier, this is because that Twitter language is very unstable and unpredictable, thus having a reliable reusable embedding for tweets is difficult.

2.7 Data set

The dataset we use is a subset from previous works [19] [6]. The data set consists of about 9,000 authors with up to 1,000 tweets each. Each tweet is a single line of text without other information about the author. The dataset has been preprocessed with several substitution rules. User names are substituted by an initial 'N', URLs by an initial 'U' and numbers by an initial 'R'. We believe the data provider does this in order to keep the usage of these symbols while minimizing the noise.

Due to the information privacy-preserving principle of Twitter, this dataset is not downloadable directly. We obtain this dataset from one of the previous researchers directly. We have used and will only use this data for this research.

3 System Description

We build several different models for Tweets Authorship Attribution. As shown in Figure 3.1, their structures are similar to each other and can be roughly divided into three parts: the text representation, feature extraction, and the classifier. We will explain our system by parts.

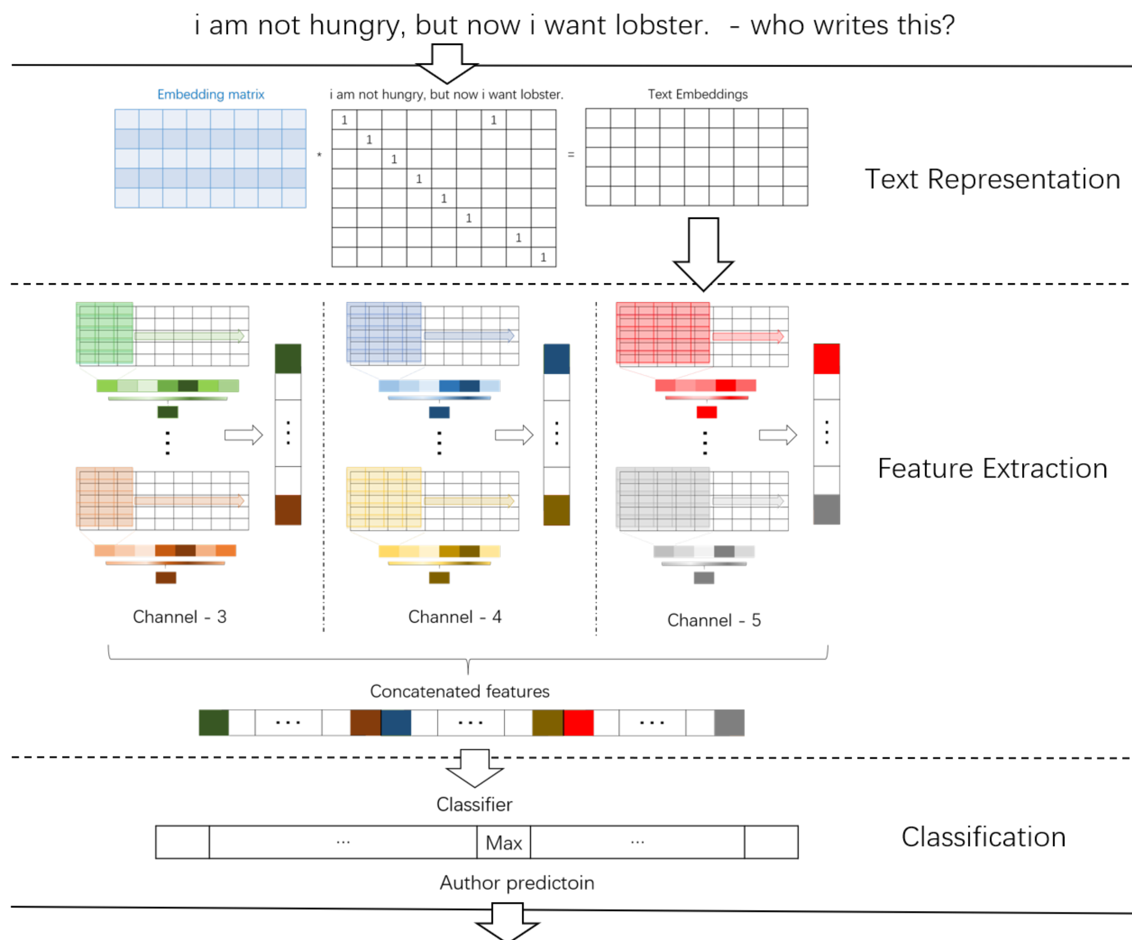

Figure 3.1: System Overview

3.1 Text Representation

The first part of our system is text representation. The text representation models we use in different systems are slightly different. But they do the same thing, which is to convert pure text into numerical representations.

3.1.1 Word Embedding, word2vec

In word2vec, two different models are chosen to train the language model, namely Continuous Bag of Words (CBOW) and Skip-gram (SG). The CBOW model (Figure 3.2) predicts word at specific position t from its context. The input of CBOW model is all the context words. It sums the vector representation (word embeddings) of context words to get a context vector v_c. After that, it computes the dot product of the context vector v_c and each candidate word w_t for position t and then normalizes with the soft-max function to generate an output as distribution. The soft-max function is commonly used in neural network classifiers for converting the final output into a categorical distribution. It is defined as:

$$\sigma(\vec{x})_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Explaining a neural network usually involves two main phases, forward and back propagation. Forward propagation is how the neural network works from the input to the output. The back propagation is how the gradients of the parameters are computed. It gets its name because the gradient computed by the chain rule is propagated from the output to the input. We will go through forward and back propagation in detail to explain the model.

Continuous Bag of Words Model

Forward Propagation

We now explain the forward propagation of CBOW, given a word sequence $w_1, w_2, ..., w_T$.

1. We start with initializing two word embedding matrices V and U of size $N \times D$, where N is the vocalulary size, and D is the embedding size that we want. Each word w_t corresponds to two different representations v_t and u_t in this model. v_t is used in the input side, while u_t is used in the output side. In subsequent analyses, V is often referred as *input vector*, and U as the *output vector*.

2. The input for the model is the sequence of one-hot embedding vectors for words in the sequence $o_1, o_2, ..., o_T$. We use $V^\top o_i$ as the word embedding to represent word w_i. This matrix works like a dictionary because each row is the D dimensional word embedding for the associated word. It works just like a look up table.

$$v_i = V^\top o_i$$

3. Then we compute the average of all the embedding vectors inside the context window with length k to represent the context vector:

$$v_c = \frac{1}{2k}(v_{t-k} + ... + v_{t+k})$$

4. Next, we compute the dot product between the context vector and each word candidate w_i using the output vector u_i. That is to compute

$$y_j = u_j^\top v_c$$

5. Finally we use all the y_j in the soft-max function to calculate the final categorical distribution over the word candidates.

$$\log p(w_t|w_{t-k}, ..., w_{t+k})_m = \frac{\exp(y_m)}{\sum_j \exp(y_j)}$$

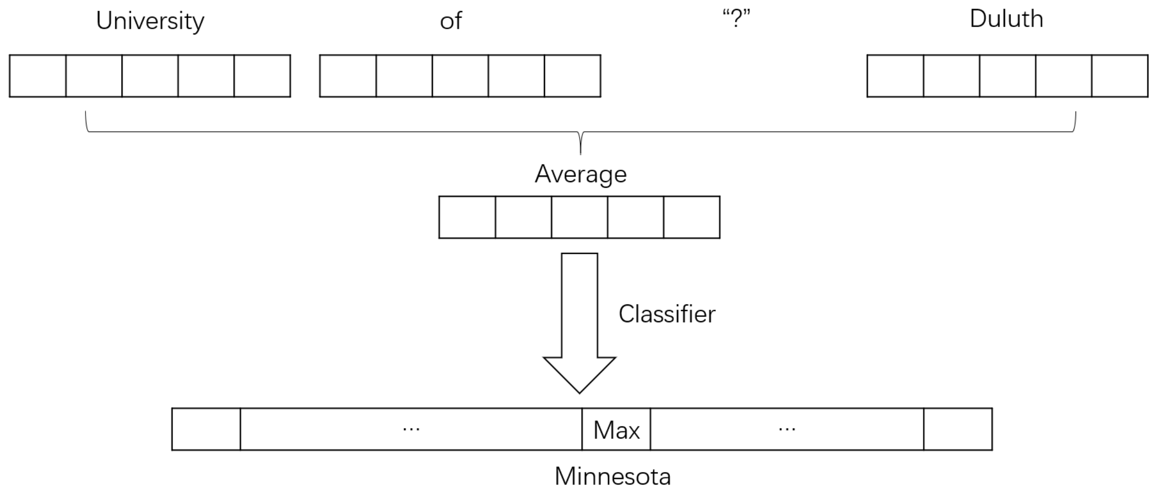

Figure 3.2: Structure of Continuous Bag of Words

Back Propagation

In back propagation, we will update the parameters to maximize the conditional probability of predicting the correct word at position t given the context:

$$\log p(w_{Out}|w_{In}) = \log p(w_t|w_{t-k}, ..., w_{t+k})]$$

where k is the window size. In order to do this, we define the loss function as

$$
\begin{aligned}
E_{j'} &= -\log p(w_t | w_{t-k}, ..., w_{t+k}) = -\log \frac{\exp(y_{j'})}{\sum_j \exp(y_j)} \\
&= -y_{j'} + \log \sum_j \exp(y_j)
\end{aligned}
$$

where $w_{j'}$ is any word candidate at position t. Our goal is to minimize this loss function. Because minimizing this loss function will maximize the conditional probability of the seen word given its context words. We will derive the update equation of the parameters.

Let us first take the derivative of the loss function with respect to $y_{j'}$. Note that this means we are now updating the j'th row of U according to the input and output pair $(w_{t-k}, ..., w_{t+k}, w_{j'})$.

$$
\begin{aligned}
\frac{\partial E}{\partial y_{j'}} &= \frac{\partial[-y_m + \log \sum_j \exp(y_j)]}{\partial y_{j'}} \\
&= -I_{j'} + \frac{\exp(y_{j'})}{\sum_j \exp(y_j)}
\end{aligned}
$$

where $I_{j'} = 1$ if $w_{j'}$ is the actual word, otherwise $I_{j'} = 0$. Remember that

$$
y_{j'} = u_{j'}^{\top} v_c
$$

Hence the gradient of $u_{ij'}$ is calculated according to the chain rule as follows:

$$
\frac{\partial E}{\partial u_{ij'}} = \frac{\partial E}{\partial y_{j'}} \cdot \frac{\partial y_{j'}}{\partial u_{ij'}} = \left[-I_{j'} + \frac{\exp(y_{j'})}{\sum_j \exp(y_j)} \right] \cdot v_{ci}
$$

The update equation for U is:

$$u_{ij'} = u_{ij'} - \alpha \cdot \left[-I_{j'} + \frac{\exp(y_{j'})}{\sum_j \exp(y_j)} \right] \cdot \frac{1}{2k} (v_{t-k} + ... + v_{t+k})_i$$

where α is called the learning rate or step size.

Note that this update equation can be interpreted in the following way. Recall that v_c is the context vector. If the word $w_{j'}$ is exactly the word seen at the position t. $I_{j'}$ will be 1 and the subtracted term becomes negative. Then we actually add a portion of v_c to $u_{j'}$, thus making $u_{j'}$ closer to v_c and all the context words. If the word $w_{j'}$ is not the actual word, then we subtract a portion of v_c from $u_{j'}$, thus making $u_{j'}$ further from v_c and all the context words.

The update function for V is similar, according to the chain rule:

$$\frac{\partial E}{\partial v_{fi}} = \frac{\partial E}{\partial y_{j'}} \cdot \frac{\partial y_{j'}}{\partial v_{ci}} \cdot \frac{\partial v_{ci}}{\partial v_{fi}} = \sum_{j'} \{ \left[-I_{j'} + \frac{\exp(y_{j'})}{\sum_j \exp(y_j)} \right] \cdot u_{ij'} \cdot \frac{1}{2k} \}$$

where f is the indices of words that are in the context window w_t. Hence,

$$v_{fi} = v_{fi} - \alpha \cdot \sum_{j'} \{ \left[-I_{j'} + \frac{\exp(y_{j'})}{\sum_j \exp(y_j)} \right] \cdot u_{ij'} \cdot \frac{1}{2k} \}$$

It can also be interpreted in a similar way. We add a positive portion of the actual word u_m to each context word, making each of them closer to the actuall word. On the other hand we subtract a portion of word that are not seen at position t from each context word, making each of them away from those words.

This is how we update the word embedding matrices V and U in CBOW model.

Skip-gram Model

Another model for word2vec word embedding pre-traininig is the Skip-gram (SG) model. SG does the opposite of CBOW model. Given a word w_t, it predicts context word at all context position i. The input of SG model is the word vector w_t at position t. Then for each context position i, it computes the dot product of the word vector and each candidate word w_i for position i and then normalizes with soft-max to generate an output as distribution. The forward propagation and back propagation of SG model are very similar to those of CBOW model. Hence we do not go through them in details. In both models, we train two matrices of word embeddings, namely

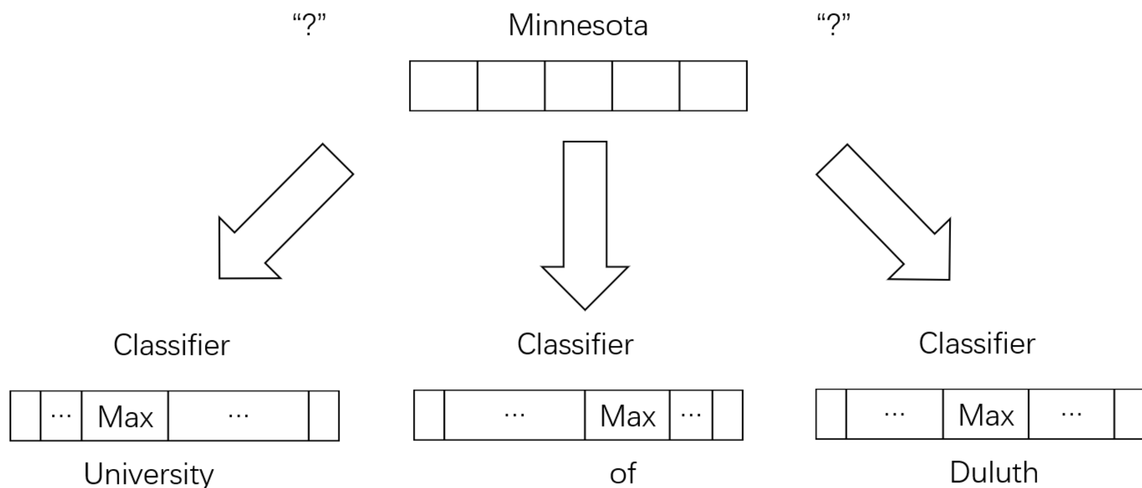

Figure 3.3: Structure of Skip-Gram

V and U. After they are trained, word occurrence information is contained within the embedding layer. V and U can be seen as two different and independent set of representations for one vocabulary. The best thing is that they work as dictionaries. So we can save the embedding matrices for reuse; when we need word embedding for a word, we can simply look up the matrix and find the vector. Usually we only need one of them and we usually choose the input vector V for reuse.

There is also a joint training strategy for word embedding. We do not train a reusable word embedding matrix. When we need word embeddings, we add a word embedding step containing embedding matrix V into the system. Then we jointly train the other parameters and this task specific word embedding matrix together. The advantage of this strategy is that task specific word embedding can usually be tuned to fit the task well.

In our system, we do not directly use word2vec, but the word2vec model is the foundation of the document embedding model we use, which is the doc2vec. Our system with word embedding configuration actually uses joint training strategy. The joint training strategy for word, sub-word and character embedding models are almost exactly the same.

3.1.2 Document Embedding, doc2vec

The doc2vec model is based on the word2vec model. It adds an embedding vector for the whole document to the prediction task and optimize both the word embeddings and the document embedding at the same time. Doc2vec extends the Continuous Bag of Words (CBOW) and Skip-gram (SG) models and creates two similar models named Distributed Memory (DM) and Distributed Bag of Words (DBOW), respectively.

Distributed Memory Model

DM model is an extended version of CBOW model. It adds a document vector v_d to the input of CBOW (see Fig 3.4). Before taking the dot product, the model adds the document vector v_d into the average context vector v_c. Then it takes dot product of the resulting vector v_c and each candidate word w_t for position t and then and then normalizes.

Forward Propagation

The forward propagation of DM model is almost the same as the CBOW model, except that we now have a document vector as input.

1. Initialize word embedding matrices V and U with size $N \times D$, **and a document embedding vector with length** D.

2. The input for the model is the sequence of one-hot embedding vectors for words in the sequence $o_1, o_2, ..., o_T$, **and the document embedding vector** v_d**. Note that the length of** v_d **is equal to the word embedding vector** $V^\top o_i$.

3. Then we compute the average of all the embedding vectors inside the context window with length k **and the document vector** to represent the **context-topic** vector:
$$v_{c+d} = \frac{1}{2k+1}(v_{t-k} + ... + v_{t+k} + v_d)$$

4. Next, we compute the dot product between the **context-topic** vector and each word candidate w_i using the output vector u_i. That is to compute
$$y_j = u_j^\top v_{c+d}$$

5. Finally we use all y_j in the soft-max function to calculate the final categorical distribution over the word candidates, which is the same as word2vec.
$$\log p(w_t | w_{t-k}, ..., w_{t+k})_m = \frac{\exp(y_m)}{\sum_j \exp(y_j)}$$

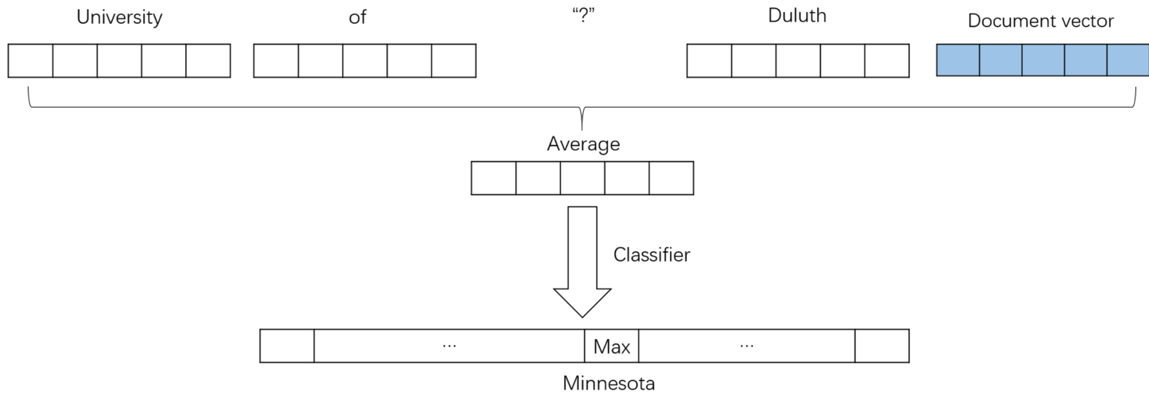

Figure 3.4: Structure of Distributed Memory

Back Propagation

In back propagation, we will update the parameters to maximize the conditional probability of getting the correct word at position t given the context **and topic**:

$$\log p(w_{Out}|w_{In}) = \log p(w_t|w_{t-k}, ..., w_{t+k}, \mathbf{doc})]$$

where k is the window size.

Here, if we consider the document just as *topic word* and consider it as part of the context, then the optimization problem of DM is exactly the same as CBOW. We can use the same loss function.

$$\begin{aligned} E_{j'} &= -\log p(w_t|w_{t-k}, ..., w_{t+k}) = -\log \frac{\exp(y_{j'})}{\sum_j \exp(y_j)} \\ &= -y_{j'} + \log \sum_j \exp(y_j) \end{aligned}$$

where $w_{j'}$ is any word candidate at position t. The update equations for V and U are almost the same only except for some trivial differences. Let's derive the update equation for v_d. Since we see the topic word as a word which is equivalent to context

words, they should have similar update equations. Recall that the update equation for word embedding matrix V:

$$v_{fi} = v_{fi} - \alpha \cdot \sum_{j'} \{ \left[-I_{j'} + \frac{\exp(y_{j'})}{\sum_j \exp(y_j)} \right] \cdot u_{ij'} \cdot \frac{1}{2k} \}$$

where f is the index of word that is in the context window w_t. For the document embedding vector v_d, we only need to change the constant $\frac{1}{2k}$ to $\frac{1}{2k+1}$.

$$v_{di} = v_{di} - \alpha \cdot \sum_{j'} \{ \left[-I_{j'} + \frac{\exp(y_{j'})}{\sum_j \exp(y_j)} \right] \cdot u_{ij'} \cdot \frac{1}{2k+1} \}$$

We can also try to understand what happens when we update the document embedding vector. Since we train the model for all positions, all words in the sentence will finally be actual words. Hence, we add a proportion of each word embedding vectors in the sentence to the document embedding, making the document topic closer to each of them. This is how we update the document embedding vector v_d in DM model.

Distributed Bag of Words Model

Similarly, Distributed Bag of Words model (DBOW) is an extended version of Skip-gram (SG) model. It replaces the input word vector w_t with the document vector w_d and predicts words for each surrounding position i just like in the SG model. Since the document embedding vector has exactly the same length as word embedding vectors, both the forward and back propagation of DBOW are exactly the same as SG. We do not go through them in details.

In both models, we train two word embeddings matrices and a document embedding vector, namely V, U and v_d. This is the case when we have no pre-trained word embeddings. The structure of doc2vec also allows us to use pre-trained word embed-

ding vectors to train a document embedding. If we now have pre-trained V and U, we can use them directly in the forward propagation and "freeze" them in the back propagation. The only thing we need to train is the document vector v_d. Therefore with pre-trained word embedding, doc2vec will be very efficient. Using trained word embedding to get document vector is referred as "infer" by doc2vec users.

In word2vec, we have the property that similar words tend to have similar embeddings. It is similar in doc2vec as well. Actually it does have. The document vector is also called topic vector in doc2vec paper [11], because documents with similar topic tend to have similar vectors. This is what we want for text classification tasks.

We implement the whole word embedding step with the help of gensim [20] NLP package on Python. This package offers several operable input parameters and training approaches. We set DM model as the training approach and other entries with default parameters given by the package. See https://github.com/JeromWang/Authorship-Attribution-with-Doc2vec

3.1.3 Character Embedding

As mentioned earlier, embedding models can have two strategies. We use the pre-training only strategy for document embedding and joint training for other embeddings including character embedding.

Suppose that the text sequence is $w_1, w_2, ..., w_T$. We split all the words into characters and represent the text as $c_1, c_2, ..., c_L$, which is a sequence of characters of length L. First, we represent each character with one-hot embedding. Now text is represented by $o_1, o_2, ..., o_L$. This step can be seen as a Vector Space Model. [21] The dimension of vector space for ASCII characters is 128, and it will be larger if the twitter corpus contains some non-ASCII characters. We do not want to use sparse

vectors with 100-200 dimensions to represent 100-200 items. Hence we project it onto a lower-dimensional space. This is done by multiplying the one-hot vectors with a character embedding matrix C with size $N \times D$, where N is the vocabulary size of characters and D is the character embedding size we want. Then each character is represented as a dense vector $c_i = C^\top o_i$. The text is represented by a sequence of character embedding $(c_1, c_2, ..., c_L)$. The character embedding matrix C is trained jointly with other parameters. We implement the one-hot and character embedding matrix with Keras library [22] in Python.

3.1.4 Sub-word Segmentation and Embedding

The work, Mikolov et al. tries to enrich word representations with sub-word information and it improves the original word embedding methods [15]. They show that sub-word is an efficient level for text representation. However, the method used to generate sub-words in their work [15] divides words into fix length n-grams, which fails to use word morphology knowledge. We believe that *Language is never, ever, ever, random* [23]. Sub-words as character n-grams could be detected by our knowledge, which is conveyed by their frequencies.

We employ the byte-pair-encoding (BPE) algorithm [24] which is a frequency based text compression algorithm to detect the sub-words. The original algorithm first splits the text into bytes (characters). Then it ranks all the byte pairs which are simply bi-grams according to their frequencies. The most frequent byte pairs are then joined together and encoded by a single byte. By repeating this simple process, the original text will be compressed. In our system, we do not use BPE as a compression algorithm. Instead, we use this algorithm to find sub-words as n-grams with high frequencies for word segmentation. This can be achieved if we only join characters

together but do not replace them with new symbols. This is not used for the first time in NLP, it was also shown to be successful in the work [25].

For example, suppose the text to be segmented is

$$S_0 = \textit{workers work in workshop.}$$

First, we split S_0 into character sequence

$$S_1 = w\ o\ r\ k\ e\ r\ s\ w\ o\ r\ k\ i\ n\ w\ o\ r\ k\ s\ h\ o\ p\ .$$

Then in the first iteration, we choose the most frequent bi-gram *wo* and join them together

$$S_2 = \underline{wo}\ r\ k\ e\ r\ s\ \underline{wo}\ r\ k\ i\ n\ \underline{wo}\ r\ k\ s\ h\ o\ p\ .$$

In the second iteration, we choose the most frequent bi-gram $\underline{wo}r$ and join them together

$$S_3 = \underline{wor}\ k\ e\ r\ s\ \underline{wor}\ k\ i\ n\ \underline{wor}\ k\ s\ h\ o\ p\ .$$

Similarly, the next bi-gram should be $\underline{wor}k$

$$S_4 = \underline{work}\ e\ r\ s\ \underline{work}\ i\ n\ \underline{work}\ s\ h\ o\ p\ .$$

We expect that the text are represented by a sub-word sequence after the algorithm finishes in N iterations, and this is the result.

$$S_N = \underline{work}\ \underline{er}\ s\ \underline{work}\ \underline{in}\ \underline{work}\ \underline{shop}\ .$$

After word segmentation, the text is now sub-word sequences. In order to use sub-word embedding to represent the text, we first represent each sub-word type with a one-hot vector. The one-hot vector for the ith sub-word in vocabulary is a sparse binary vector o_i which has 1 as the ith element and all 0 for others. We project this embedding hyperspace onto a smaller hyperspace by multiplying the one-hot embedding with a sub-word embedding matrix S with size $N \times D$, where N is the sub-word vocabulary size and D is the dimension for the target embedding

hyperspace. Now each sub-word is represented by a dense vector $s_i = S^\top o_i$, and the text with length of T is represented by a sequence of sub-word embedding vectors $(s_1, s_2, ..., s_T)$. The sub-word embedding matrix S is trained jointly. We implement the one-hot and sub-word embedding matrix with Keras library in Python.

3.2 Feature Extraction

The second part of our system extracts features based on the text embeddings. Given the text embeddings, we will use a convolutional neural network to extract the features. Again, the features extracted by CNN are hard to interpret. But as we optimize the loss function, we believe that everything in our system works as they are expected. Our convolutional neural network can be used on any level of text embeddings.

A Convolutional Neural Network is a neural network with specific structures. Our convolutional neural network is a one-layer multi-channel convolutional neural network that works as follows:

The input of the convolution layer is the text embedding sequence. Let us use the word embedding $v_1, v_2, ..., v_L$ as an example, where L is the sequence length. If we set the embedding dimension to be D, then our text will be represented by a matrix with size $D \times T$. The convolution operation is done with a set of convolution *kernels*. Each kernel is also a matrix with size $D \times k$, where the k is called the kernel size. Our convolution layer consists of two steps.

1. Calculate the convolutions of the resulting vector sequence from the sub-word embedding vector sequence $T = (v_1, v_2, ..., v_L)$ and convolution kernels. The kernel size r represents the context window of feature extraction and the kernel

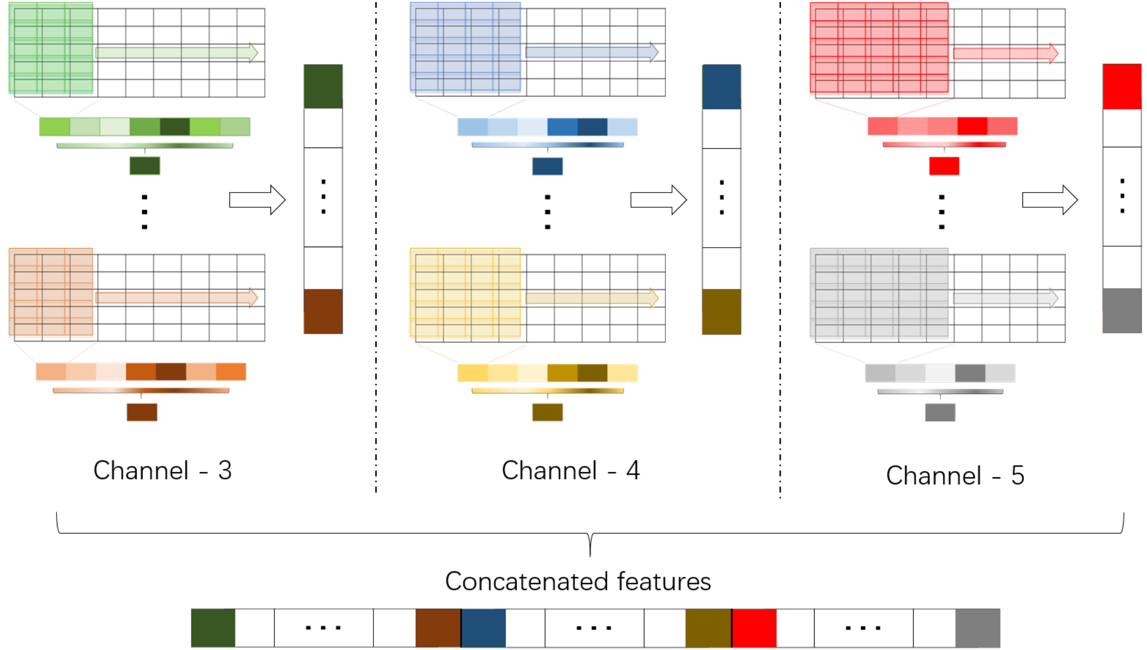

Figure 3.5: Multi-channel Convolution Layer

number m represents the number of features.

$$f^{m'}_{l=1:L} = \sigma(k_m * [v_{l-r/2+1}, ..., v_l, .., v_{l+r/2}]), \quad \sigma(x) = \max\{0, x\}$$

where m is the kernel number and σ is the activation function. The convolution operation $*$ works like Figure 3.6. We calculate the element-wise product and then sum them together. After that, we calculate the activation function value of the sum.

2. Then the pooling step chooses the maximum as the abstract feature from all the convolutions

$$f^m = \max\{f^{m'}_{l=1}, ..., f^{m'}_{l=L-r+1}\}$$

The convolution unit outputs the sequence of features which are as many as

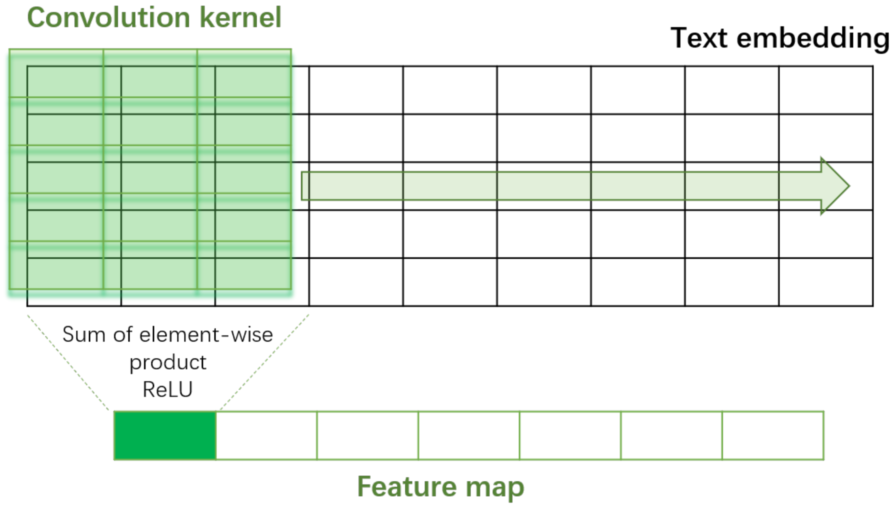

Figure 3.6: 1-D Convolution

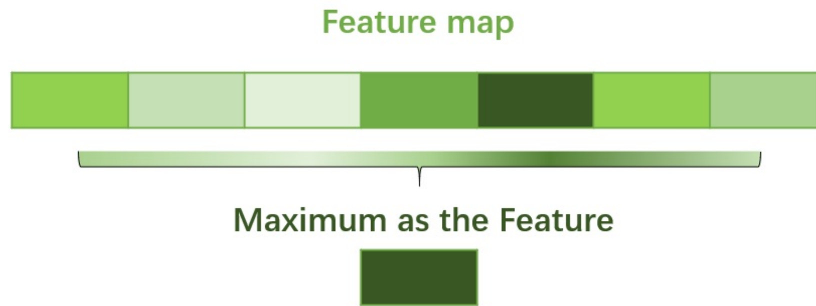

Figure 3.7: Max-Pooling

the kernel number. We extract one feature from one convolution kernel. The output can be seen as the features extracted within a certain size of context window

$$F = (f^1, ..., f^M)$$

The convolution between two matrices is just like the matrix version of dot product. Hence we can imagine that the convolution kernels are very like the embeddings of some certain n-grams as features. The features are thus the convolution results as scores of the existence of these n-grams. Among all the convolution results, we

only pick the largest value. Because the largest value correspond to the closest match between the sentence and the n-gram we are looking for.

Each kernel-pooling unit in the CNN works as above. In our system, we have three channels with different kernel sizes of 3, 4, and 5. Each channel has $M = 512$ kernels. We concatenate all three feature maps to one single vector. This single vector is later used as the features for classification. The code we used can be found at: https://github.com/JeromWang/UMDStatProjectAA

3.3 Classification

The third part of our model works in a similar way as most authorship attribution models. After we generate features of each text, we use these vectors as input to a classifier and expect to get authorship predictions as output. Our approach to solve the authorship attribution problem consists of two steps. In the training step, we use batch gradient descent and back propagation to update the parameters. Then we use test set (a held out test set to evaluate the accuracy of the classifier) to test the accuracy of the classifier. The classifier we use is a neural network classifier. It consists of two basic fully connected neural networks and one soft-max function. It is able to fit the function which takes the features and output the classification result.

We implement the training and testing of the classifier with scikit-learn library [26] in Python.

3.3.1 Other Classifiers

For the doc2vec system, we also test other classifiers including Random Forest, Support Vector Machine (SVM), and Logistic Regression classifiers as alternatives, since they are generally robust classifiers. Neural network is the most accurate by

comparison.

For random forest classifier, we implemented it with scikit-learn package in Python. We used the default parameters for building the random forest, but we could not get a satisfactory result. We explain this by the behavior of decision tree. Since each decision tree creates one boundary which is parallel to one axis in the feature space, the cost and accuracy decrease when the dimension of features grows. In our embedding, one document is represented by a feature vector with 300 dimensions, which is beyond the best zone of random forest.

For SVM, one problem is the parameter optimization. We tried to solve the problem and avoid the tedious work by using previous work that can automatically select the parameters for SVM models. We implemented this work by using the libsvm package in Python [27]. However, the best outcome of SVM model with this package is still not as good as that of CNN. Our explanation for this is SVM models perform not as well in multi-classes problems, especially when the feature number is so huge.

For logistic regression, we used the one-versus-rest (OvR) scheme and implemented with the sklearn package in Python. This OvR logistic regression performs better than Random Forest and SVM, but not as good as the neural network. See the comparison of results of different classifiers in Chapter 4 Experimental Results.

3.3.2 Embedding as Dimension Reduction Method

For general classification problems, a short and efficient list of features is always what researchers want as the classifier input. This becomes a popular strategy in many problems when longer vectors does not work out very well. Inspired by the previous work [28], in our doc2vec system we try to add a step of dimension reduction before we apply our classifiers on the document embedding.

We use the traditional principal component analysis (PCA) for the dimension reduction. PCA finds an orthogonal transformation between the original feature set and the new feature set and sorts the new feature set by the variances on feature dimension. By doing this, we can select the most useful features in the new feature set by setting threshold for variance proportion. Usually it can improve classifier performance by gleaning out the less useful features. We implement the PCA and graph with the PCA function from matplotlib package [29] in python. It automatically determines the transformation of features. We compare the fracs attributes which are the proportion of variance of each principal component to get the importance of the features (see the table 3.1).

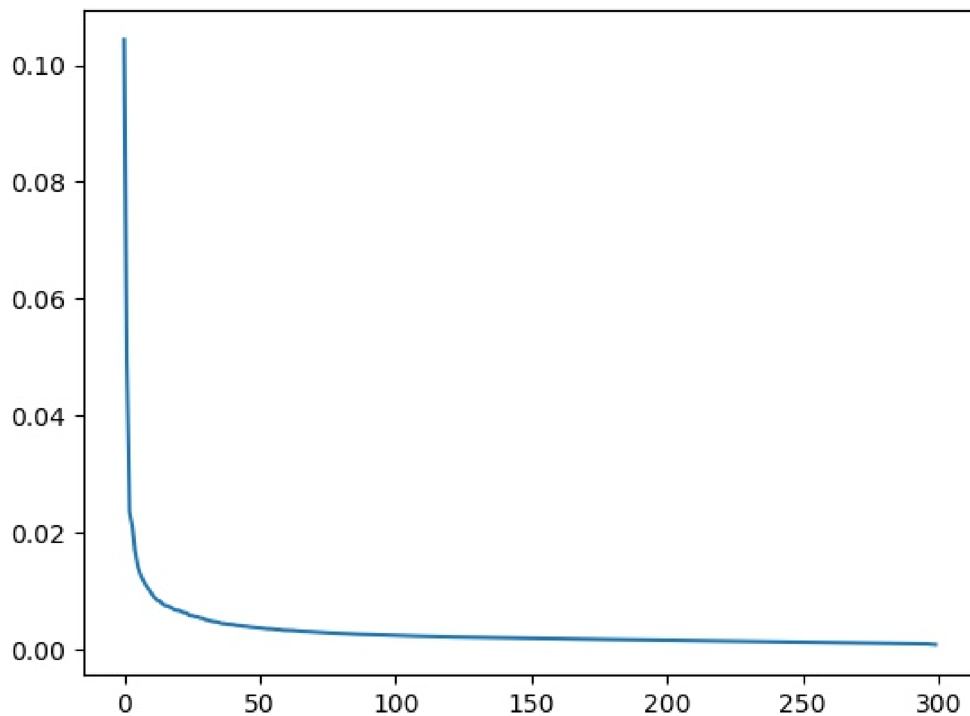

Figure 3.8: % of importance of features after PCA

No PCA	PCA-100	PCA-95	PCA-90
0.411	0.3845	0.3812	0.3816

Table 3.1: Accuracy for keeping different % of importance of feature set for 50 authors with 1,000 tweets each

The fracs curve has a signifincant drop. However, the drop happens too early, leaving most of the importance in the long tail. This could mean that each feature in the set is useful to the same degree. In order to see if further reductions can be made on this feature list, we try to glean out the last features to keep 95% and 90% of the importance in the feature set. However this harms the performance of the classifier. This means the 300 dimensions are all informative and cannot be reduced any more.

In this work [5], Gittens et al. shows Skip-Gram model can be interpreted as a heuristic method of original Sufficient Dimensionality Reduction (SDR) method. The difference between these two is whether to fit the conditional distribution as the former does or the entire probability mass functions as the latter does. This interpretation of word embedding bridges the idea of word embedding and co-occurrence feature, unveiling the mystery of word embedding. Our experiments corroborate this point of view.

4 Experimental Results

4.1 Experiment Settings

We are interested in the performance of our model when the number of authors is large, which is more likely to be the actual case on Twitter. We use a subset of our dataset from [19] [6]. The subset of data for our experiment contains 1,800 authors and 200 tweets each. The 1,800 authors along with their tweets are separated into four groups with sizes 100, 200, 500 and 1,000. We conduct experiment for all four of our systems and compare their results with that of a state-of-the-art method on the same dataset.

4.2 Hyperparameter Details

We use all the tweets in the dataset collectively for training the byte pair encoding algorithm in 2,000 iterations. Each iteration of BPE algorithm generates a new character n-gram in the vocabulary. Hence our sub-word vocabulary size is around 2,000. We embed each sub-word with a vector with dimension $D = 300$. In our concatenated convolutional neural network, each unit has $M = 128$ filters with filter sizes and max pooling size $r = \{3, 4, 5\}$. For comparison, we also build character based system containing not only the ascii characters with vocabulary size $|V| = 200$ but also a word based system with vocabulary size $|V| = 10,000$. In order to compare the efficiency of different embedding levels, we set embedding dimension on all levels as

Embedding	$\|V\|$	D	M	r
Word	10K			
Sub-word	2K	300	500	3, 4, 5
Character	200			

Table 4.1: Network parameter settings

$D = 300$. We also use the same network structure for all different text representation methods. These parameter settings are shown in Table 4.1.

We use Adaptive Moment Estimation [30] with a learning rate $\alpha = 10^{-4}$ for parameter training. Each model is trained for 100 epochs on data batches with size 32. We add two dropout layers [31] after the embedding layer and convolution layer respectively with droping rate 0.25 in order to prevent over-fitting. Apart from that, an early stopping strategy is also applied for over-fitting prevention. We set the early stopping patience to be 10, which means the training process will be terminated after 10 epochs, accumulatively, without reduction of the loss. This causes the training to stop after about 30-40 epochs. We save our models after each training epoch as checkpoints and choose the one which has the highest accuracy for comparison. We evaluate all the model with cross validation method with the number of folds equal to 10.

4.3 Results and Evaluation

We check the training log to make sure that the training converges. From the training accuracy plot in Figures 4.1 and 4.2, we can see that both system converges as we have more training epochs. The curves are generally going just as we expect, where training accuracy increases but test accuracy gradually plateaus.

The final results are in Table 4.2. In the table, Char means character embedding and Char-2 means character bi-gram, sub for sub-word, d2v for doc2vec, and BOW-

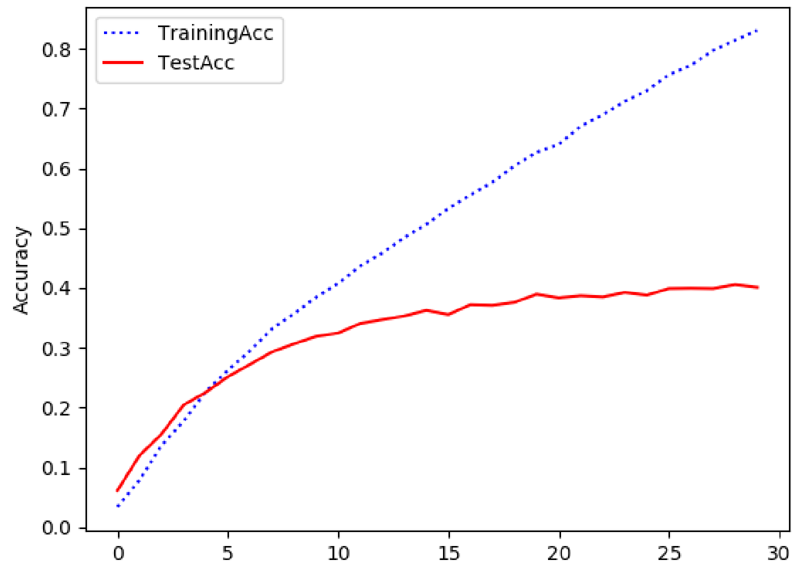

Figure 4.1: Accuracy versus training epochs for sub-word system

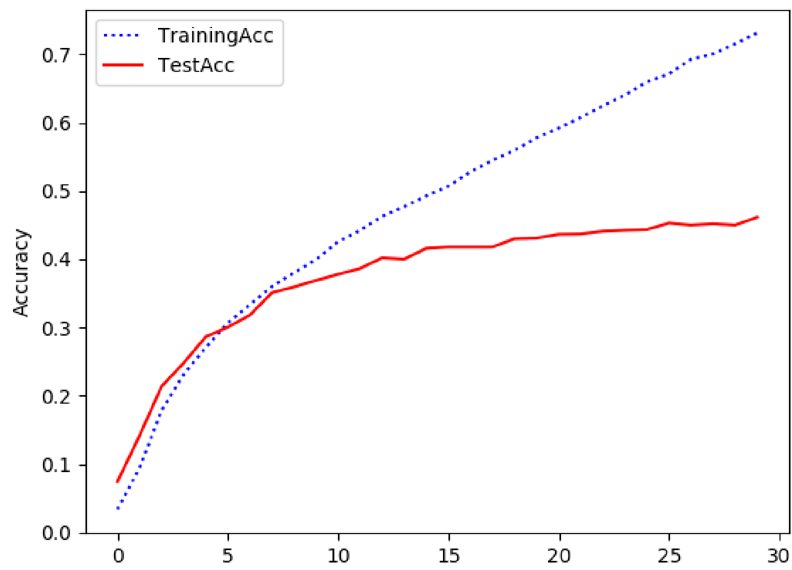

Figure 4.2: Accuracy versus training epochs for character system

	CNN-Char	CNN-Sub	CNN-Word	CNN-d2v	CNN-Char-2	BOW-LR
100	**0.508**	0.407	0.241	0.338	0.506	0.445
200	0.473	0.393	0.208	0.256	**0.482**	0.442
500	0.417	0.353	0.161	0.218	**0.422**	0.384
1000	0.359	0.284	0.127	0.154	**0.365**	

Table 4.2: Accuracy for different # of authors with 200 tweets each

LR for the bag of words logistic regression baseline. The result of the baseline for the largest experiment is missing because of the limit of our computing resources, but we believe that the number will not surprise us. The results show that the multi-channel based on character and character n-grams are the most accurate, followed by the sub-word, doc2vec and word embedding. The character embedding methods improve the second best sub-word methods by about 6-10%, which is significant.

We explain the reasons why character embedding is the best by comparing it to all other embeddings. Comparing to word embedding and document embedding which are also based on words, the success of character embeddings can be explained since tweets language are generally irregular. This means that many words are variations from their standard forms, like yeees vs. yes. In these cases, word embeddings can not represent tweet texts effectively but characters can do. Because character embedding system has this better representation, it can extract better features and get higher accuracy.

To compare the character and sub-word systems, we first want to see what are the sub-words extracted from byte-pair-encoding. We get a list of the most frequent sub-words in the 1,000 authors experiment shown in Table 4.3

From this list, we can gain some understanding of sub-words. The sub-words segmented from the byte-pair-encoding algorithm can be seen as frequent characters or their n-grams. They can be seen as n-grams-level features learned without supervi-

Most frequent sub-words				
(name)	i	a	the	to
in	(url)	s	(number)	and
.	on	t	is	it
you	e	d	,	u
!	for	my	-	?
y	of	me	...	#
n	n	m	k	p
be	o	c	that	at
l	:	b	an	re
g	ing	er	st	”

Table 4.3: Most frequent 50 sub-words in 200,000 tweets from 1,000 authors

sion from character-level. When we do our experiments, we assume that sub-word will give us better results than character because the sub-word segmentation can combine the frequent and widely used character n-grams into one sub-word and leave the rare and indicative patterns for the CNN to extract. Consider the example of the sub-word *pre*. It is frequently used among all authors, and so it can be easily extracted from characters and have this prefix as a sub-word rather having the CNN extracting it. But to our surprise, sub-word does not improve the CNN feature extraction. We explain this by the fact that there are also cases when some character n-grams are frequent but only used by certain users. When we combine these character into sub-words, we actually remove these decisive features from the system. We expect that these two effects both exist in sub-word system and in this dataset, sub-word segmentation seems to make more negative effects.

It is important to note that these methods are effective, and the rankings among them are very likely to change from one dataset to another. It is very interesting to study the reasons why a certain method works well on one data set (or not) in the future.

As mentioned earlier, we use four classifiers in the doc2vec configuration experi-

ments and find Convolutional Neural Network as the most accurate one. Table 4.3 shows the accuracies we get from the four classifiers which are Random Forest(RF), Support Vector Machine (SVM), Logistic Regression (LR) and Convolutional Neural Network (CNN), and we include the bag of words baseline (BOW). We implent them with sk-learn and Keras library in Python. We use a relatively small set containing less authors for the comparison of classifiers and we expect the results will generalize to larger datasets. Note that the accuracy varies greatly because of their different characteristics. But no matter what classifier we try with doc2vec, the performance is still a lot worse than the bag of words baseline. Because the first four results are from combinations of doc2vec embeddings and classifier, this means that doc2vec features are still not good enough for the authorship attribution task.

RF	SVM	LR	CNN	BOW
0.213	0.216	0.384	0.411	**0.739**

Table 4.4: Accuracy for 50 authors with 1,000 tweets each

5 Conclusions and Future Work

5.1 Conclusions

In this work, we evaluated four different embedding methods including character, sub-word, word and document embedding on a Tweets Authorship Attribution dataset with carefully designed experiments. We compare the results we get with state-of-the-art methods. Our results show that all of them are accurate methods compared to random baseline while the character embedding based system is the most accurate of these. This means that for corpora on twitter which are generally irregular, using character n-grams embeddings for text representation can improve performance over word embedding based methods. Using all character bi-grams for feature extraction is better than using sub-words, which are the most frequent varying-length character n-grams extracted by byte-pair-encoding algorithm. Finally, we observe that the bag of words (BOW) baseline is a solid method. It gives both good performance and understandability in this task.

5.2 Future Work

In this work, we test different text embedding methods on a Tweets Authorship Attribution task. Since we only do experiment on one dataset, the conclusions we draw about these embedding methods may not hold true on other datasets. In the future, we will test our systems on other different tasks and datasets. The datasets

we have already found include amazon reviews, sogou news, yahoo question answers and yelp reviews. They are all very good datasets with reasonable sizes for text classification tasks. Theyare shared by previous researchers and can be found in the following link:

https://drive.google.com/drive/folders/0Bz8a_Dbh9QhbfII6bVpmNUtUcFdjYmF2SEpmZUZUcVNiMUw1TWN6RDV3a0JHT3kxLVhVR2M?usp=sharing

It can be observed in our experiments that sub-word embeddings do not always work better than word or character embeddings. Our next goal is to have a deeper understanding of the reasons why sub-word works or does not. Then we can decide when to use sub-word. This work is currently in progress (see Figure 5.1 and 5.2). We believe that one of the key factors is the discrepancy between distributions of words in the corpus and sub-words generated by our segmentation algorithm.

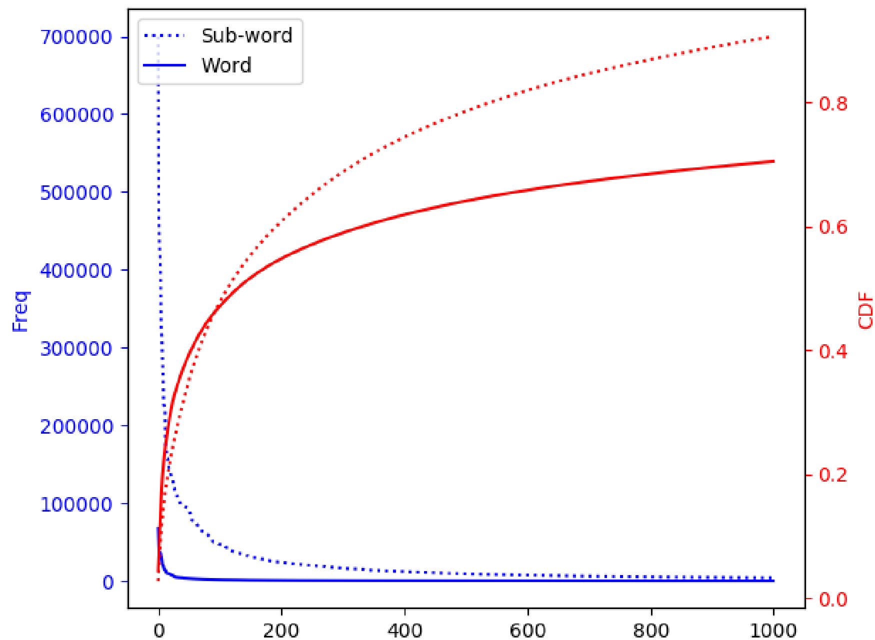

Figure 5.1: Frequency and Cumulative distribution function of word and sub-word on Sentiment Analysis dataset (2)

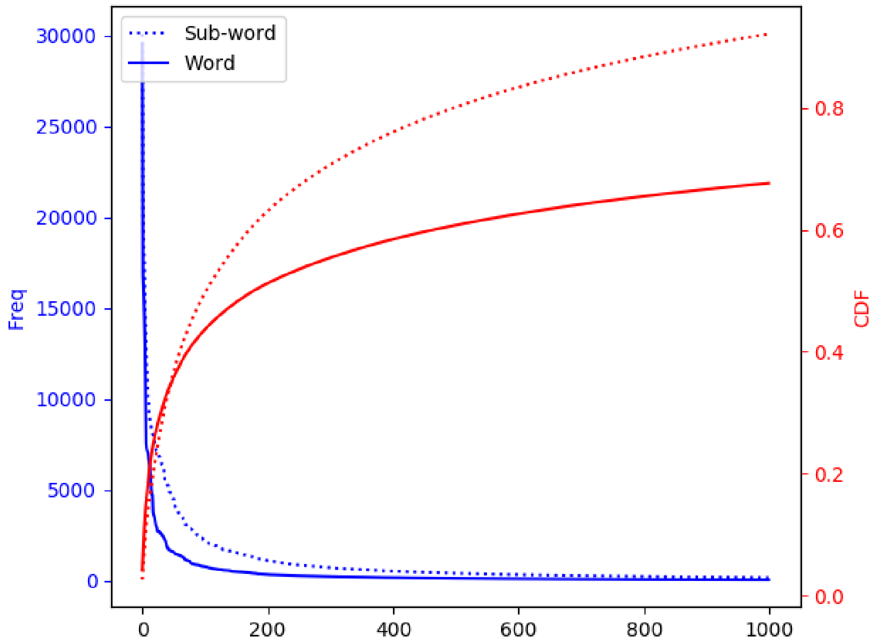

Figure 5.2: Frequency and Cumulative distribution function of word and sub-word on Tweet Authorship Attribution dataset (50)

In Figure 5.1 and Figure 5.2, we compare the frequency and cumulative distribution function (CDF) of the most frequent 1000 tokens in both word and sub-word system. The two datasets being compared are for Sentiment Analysis and Authorship Attribution, respectively. Sentiment Analysis problem is a binary classification task with relatively long text, while this authorship attribution task is a 50-class classification with short text. In Figure 5.1, the most frequent sub-words are approximately 7 times frequent than words. But in Figure 5.2, this does not happen. The result is that sub-word embeddings improve over word embeddings significantly in the task of Figure 5.1 but not so much in the task of Figure 5.2. However we are not sure about how that will affect the accuracy of sub-word embedding at this point. Also note that from these two figures, we can see from their CDFs that with sub-word embeddings, we are able to represent text with smaller vocabulary. Because with the same amount

(1000) of words or sub-words, we can represent a higher proportion of the text using sub-words (≈ 0.85) than words (≈ 0.65).

We are also interested in making our sub-word segmentation algorithm (byte-pair-encoding, BPE) better. From previous experiments, we notice that not all sub-word are segmented in the way that we imagine that follows the word morphologies. Here is an example we found:

Before BPE:

jacksonville, tiger woods is closer to competing again. woods is back home after a week of family counseling?

After BPE:

jack-son-vil-le, ti-ger woo-ds is clo-s-er to com-pe-ting again. woo-ds is back home after a week of family coun-sel-ing?

From this example, we can see segmentations like "com-pe-ting" that make sense but also "woo-ds"which seems less reasonable. Also there are long patterns that appear frequent enough to escape being segmented e.g., "yesterday". These go against our motivation for sub-word segmentation which is to represent text while making use of word morphology information. Hence in the future we will continue our study of the possible constraints that can be made on byte-pair-encoding algorithm. With several constraints on for example, length, vowel, we believe that we can minimize bad segmentations and thus improve the sub-word embedding even more. We are trying to get morphological information "cheaply" without doing morphological parsing.

References

[1] F. Mosteller and D. L. Wallace. "Inference in an authorship problem: A comparative study of discrimination methods applied to the authorship of the disputed Federalist Papers". In: *Journal of the American Statistical Association* 58.302 (1963), pp. 275–309 (cit. on pp. 1, 4).

[2] T. Joachims. "Text categorization with Support Vector Machines: Learning with many relevant features". In: *Machine Learning: ECML-98: 10th European Conference on Machine Learning Chemnitz, Germany, April 21–23, 1998 Proceedings*. Ed. by C. Nédellec and C. Rouveirol. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 137–142. ISBN: 978-3-540-69781-7. DOI: 10.1007/BFb0026683. URL: https://doi.org/10.1007/BFb0026683 (cit. on p. 7).

[3] A. Genkin, D. D. Lewis, and D. Madigan. "Large-Scale Bayesian Logistic Regression for Text Categorization". In: *Technometrics* 49.3 (2007), pp. 291–304. ISSN: 00401706. URL: http://www.jstor.org/stable/25471349 (cit. on p. 7).

[4] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems*. 2013, pp. 3111–3119 (cit. on p. 10).

[5] A. Gittens, D. Achlioptas, and M. W. Mahoney. “Skip-Gram - Zipf + Uniform = Vector Additivity”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. July 2017 (cit. on pp. 11, 41).

[6] P. Shrestha, S. Sierra, F. Gonzalez, M. Montes, P. Rosso, and T. Solorio. “Convolutional neural networks for authorship attribution of short texts”. In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. Vol. 2. 2017, pp. 669–674 (cit. on pp. 12, 17, 20, 42).

[7] Y. Xiao and K. Cho. “Efficient character-level document classification by combining convolution and recurrent layers”. In: *arXiv preprint arXiv:1602.00367* (2016) (cit. on pp. 12, 15, 17).

[8] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush. “Character-Aware Neural Language Models.” In: *AAAI*. 2016, pp. 2741–2749 (cit. on pp. 12, 17).

[9] J. Kiefer and J. Wolfowitz. “Stochastic estimation of the maximum of a regression function”. In: *The Annals of Mathematical Statistics* (1952), pp. 462–466 (cit. on p. 14).

[10] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. “Enriching Word Vectors with Subword Information”. In: *CoRR* abs/1607.04606 (2016). URL: http://arxiv.org/abs/1607.04606 (cit. on p. 15).

[11] Q. V. Le and T. Mikolov. “Distributed Representations of Sentences and Documents”. In: *arXiv preprint arXiv:1405.4053* (2014) (cit. on pp. 15, 18, 32).

[12] M.-T. Luong and C. D. Manning. “Achieving open vocabulary neural machine translation with hybrid word-character models”. In: *arXiv preprint arXiv:1604.00788* (2016) (cit. on p. 15).

[13] J. Botha and P. Blunsom. “Compositional morphology for word representations and language modelling”. In: *International Conference on Machine Learning*. 2014, pp. 1899–1907 (cit. on p. 15).

[14] S. Qiu, Q. Cui, J. Bian, B. Gao, and T.-Y. Liu. “Co-learning of Word Representations and Morpheme Representations.” In: *COLING*. 2014, pp. 141–150 (cit. on p. 15).

[15] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. “Bag of tricks for efficient text classification”. In: *arXiv preprint arXiv:1607.01759* (2016) (cit. on pp. 16, 18, 33).

[16] J. H. Lau and T. Baldwin. “An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation”. In: *CoRR* abs/1607.05368 (2016). arXiv: `1607.05368`. URL: `http://arxiv.org/abs/1607.05368` (cit. on p. 18).

[17] J. Pennington, R. Socher, and C. D. Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)* 12 (2014) (cit. on p. 19).

[18] R. Caruana. “Multitask learning”. In: *Learning to learn*. Springer, 1998, pp. 95–133 (cit. on p. 20).

[19] R. Schwartz, O. Tsur, A. Rappoport, and M. Koppel. “Authorship attribution of micro-messages”. In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. 2013, pp. 1880–1891 (cit. on pp. 20, 42).

[20] R. Řehůřek and P. Sojka. “Software Framework for Topic Modelling with Large Corpora”. English. In: *Proceedings of the LREC 2010 Workshop on New Chal-*

lenges for NLP Frameworks. http://is.muni.cz/publication/884893/en. Valletta, Malta: ELRA, May 2010, pp. 45–50 (cit. on p. 32).

[21] D. Jurafsky. “Speech and language processing: An introduction to natural language processing”. In: *Computational linguistics, and speech recognition* (2000) (cit. on p. 32).

[22] F. Chollet et al. *Keras*. https://keras.io. 2015 (cit. on p. 33).

[23] A. Kilgarriff. “Language is never, ever, ever, random”. In: *Corpus linguistics and linguistic theory* 1.2 (2005), pp. 263–276 (cit. on p. 33).

[24] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. *Byte Pair encoding: A text compression scheme that accelerates pattern matching*. Tech. rep. Technical Report DOI-TR-161, Department of Informatics, Kyushu University, 1999 (cit. on p. 33).

[25] R. Sennrich, B. Haddow, and A. Birch. “Neural machine translation of rare words with subword units”. In: *arXiv preprint arXiv:1508.07909* (2015) (cit. on p. 34).

[26] 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”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 38).

[27] C.-c. Chang and H. Lin. “A library for support vector machines”. In: (2007) (cit. on p. 39).

[28] H. Kim, P. Howland, and H. Park. "Dimension reduction in text classification with support vector machines". In: *Journal of Machine Learning Research* 6.Jan (2005), pp. 37–53 (cit. on p. 39).

[29] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55 (cit. on p. 40).

[30] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 43).

[31] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958 (cit. on p. 43).