SENTIMENT ANALYSIS ON TWEETS USING DOCUMENT EMBEDDINGS

Qasim Nawaz
Supervisor: Professor Martin Russell

Submitted in conformity with the requirements
for the degree of BSc in Computer Science

School of Computer Science
University of Birmingham

Abstract

Sentiment Analysis is one of the primary areas of natural language processing and information retrieval being tackled by researchers to date, and for good reason; the internet. The internet is a mostly untapped source of rich amounts of data that can be used to gauge the opinions of people, in reference to any number of topics. Twitter is one such platform designed for people to voice their opinions in the form of tweets about any topic they desire. My project will set out to investigate the best way to be able to analyse the sentiment of these aforementioned tweets using machine learning techniques. I will be training word vector-based, and paragraph vector-based models on a dataset consisting of 1.6 million tweets, in conjunction with various classifiers in order to find the best performing method in which to obtain the sentiment of tweets.

Acknowledgements

I would like to express my thanks to all those who made this project possible, and in the standard that it is right now. I would like to give a special thanks to my project supervisor, **Professor Martin Russell**, whose advice and knowledge put me on track, and ensured the completion of this project. His suggestions and advice resonate throughout the content of this report and throughout my project. I would also like to thank my project inspector, **Professor Chris Baber**, whose comments and advice were invaluable in making this project what it is. I would also like to thank all the researchers, for whose work I referenced, for all their hard work.

Chapter 1 - **Introduction**

In this project, I have set out to design and create a model that will find the sentiment of any given tweet with the highest degree of accuracy. In order to do this, however, I will need to employ a technique called sentiment analysis. Sentiment analysis is being able to gauge the opinion, sentiment or subjectivity of text (Pang and Lee, 2008). This is to say that given a piece of text, we want to find out whether it is positive or negative. This is something we, as humans, do every day whether we know it or not. We are constantly taking in information in many different forms, and then using this data to build up our view of the world around us.

Twitter is now a huge platform, designed for people to be able to share their opinions on any topic, person, product or anything they wish. It is now so big in-fact, that it boasts over 330 million monthly active users, of which 145 million are daily active users (Lin, 2019). This is a lot of potential data.

Being able to tweet about a specific topic and then tagging it with a #, for instance #UoB, allows all the different opinions by all the different people sharing them to be grouped together and easily searched. This has then made it a very effective method of collating all this data and analysing the opinions of the general population. This would be very valuable for, take for example, large corporations where they are able to analyse market opinion about a specific product or service they offer. These companies would then be able to look at this data, and make informed decisions about how to strategize a marketing campaign to increase sales, or about how to make changes to the development of the product etc. This can be useful in many more ways however, for instance, for political parties or agencies where they are able to gauge how popular a particular candidate or party is relative to another, or even for a high-profile individual wanting to know what people think in response to a publicized event they were involved in.

It would be a long and tedious task to have to go through each and every tweet, and manually figure out if the sentiment is positive or negative. Hence, using an automated method, in which a tweet is taken in and sentiment is calculated with a high degree of accuracy would prove to be very useful.

My project is an investigation into the various methods that could be used to solve this task and build a model that will yield the most accurate results.

Chapter 2 - **Background**

2.1 Supervised Learning

A computer is said to have learned something if it has learned from experience, E, with respect to some class of tasks, T, and performance measure, P, if its performance, P, at tasks in T improves with experience, E (Mitchell, 1997). Supervised Learning is a method of machine learning in which the machine is assisted by a 'teacher'. It is called supervised learning because the process of the algorithm learning from the training set is assisted in such a way that it can be thought of as a teacher supervising the learning process (Brownlee, 2019). We, as the teachers, know the correct outputs, and we iteratively correct the algorithm as it makes its own predictions on what the output should be. We usually stop this learning process when the error rate is low enough to be considered acceptable. More formally we describe supervised learning in the following way:
A 'teacher' gives the machine learning model some labelled data:

$$(x^1, y^1), \dots, (x^n, y^n) \sim D$$

Where x = feature, y = class, and D = Distribution of data. The goal of the algorithm is to predict the class, given some features:

$$p(y|x) = \frac{Pr_{(X,Y)\sim D}(X = x \wedge Y = y)}{Pr_{(X,Y)\sim D}(X = x)}$$

In layman's terms, given that some features have been observed, what is the class that they belong to?

2.2 Unsupervised Learning

Unsupervised learning is a method of machine learning in which you input some data into a system, but there is no output to speak of. The goal of unsupervised learning is to be able to model the underlying data, or learn more about the distribution, so that we can find out more about the data (Brownlee, 2019). It is called unsupervised learning because there is no 'teacher' to speak of, not even any outputs, because the algorithm is left on its own and is expected to learn more about the data. One popular application of unsupervised learning is clustering, particularly k-means clustering. More formally we describe supervised learning in the following way:
We observe a dataset:

$$x^1, \dots, x^n \sim D$$

Where x = feature, and D = Distribution of data. The goal of the algorithm is to learn something about the distribution of the data:

$$p(x) = Pr_{X\sim D}(X = x)$$

In layman's terms, what is the probability of observing a given datapoint, x?

Chapter 3 - **Design**

3.1 Word Vectors

Word2Vec is an unsupervised algorithm, or rather a collection of algorithms, which learns vector representations for each word, relative to other words in a corpus in a multi-dimensional vector space using neural networks (NSS, 2017). These vector representations that we obtain from these neural networks are of particular interest because the learned vectors explicitly encode many linguistic regularities and patterns (Mikolov et al, 2013). This is to say that the computer is able to compute and understand the meaning behind each word. What this means is that two words with very similar meanings will have vector representations that are very close to each other, in contrast to two words that have very dissimilar meanings, that will have very differing vector representations relative to each other. A surprising property of word vectors is that they have the ability to solve word analogies using vector arithmetic (El Boukkouri, 2018). For instance, if we take the representations for 'Man', 'Woman, 'King', and 'Queen' then the following is observed to be true: 'King' – 'Man' + 'Woman' = 'Queen'. Another important characteristic to keep in mind is that all word vector representations are unique, and that WordVector1 can only be equal to WordVector2 if, and only if, Word1 is equal to Word2.

Word2Vec, as mentioned before, is a collection of algorithms and it consists of the Continuous Bag-of-words (CBOW) and Skip-gram algorithms. Both of these are shallow neural networks which map a word/words to a target variable which is also a word/words and both of these techniques learn weights which are effectively our word vector representations (NSS, 2017).

3.1.1 Continuous Bag-of-words (CBOW)

The CBOW algorithm is one method in which to implement the Word2Vec algorithm and obtain word vector representations. Given a corpus, this method takes the context of each word (the surrounding words) and tries to predict the focus word (the word missing from this context), which is the word for which we are trying to obtain a representative vector (Karani, 2018). On a high level, what is essentially happening is that we are taking a window of context words, of a given size, and then inputting these words into the network. We then feed these inputs, combined with their weights, into the hidden layer. The hidden layer is then combined with the weights between the hidden and the output layer, a softmax activation function is applied and then finally, the focus word is the output. The embedding is obtained by taking the weights between the hidden layer and the output layer (NSS, 2017). The embedding is effectively our word vector representation for our focus word. Intuitively, what we have here is a 'fill in the gap' problem where we are asking the question: For the following context of words, what is the missing word? 'The man drove his _ home from work'. We would expect the most probable word here to be 'car'.

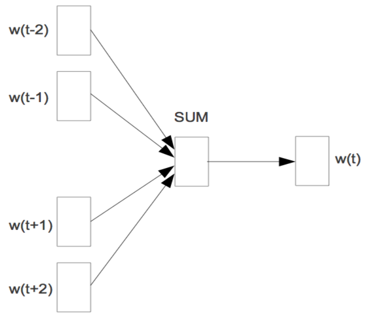


Figure 1: CBOW Model

3.1.1.1 Single-Word Context

In order to explain how CBOW works, we will start by taking a look at the simplest CBOW implementation, where we take a single context word and try to predict a single focus word (Rong, 2016).

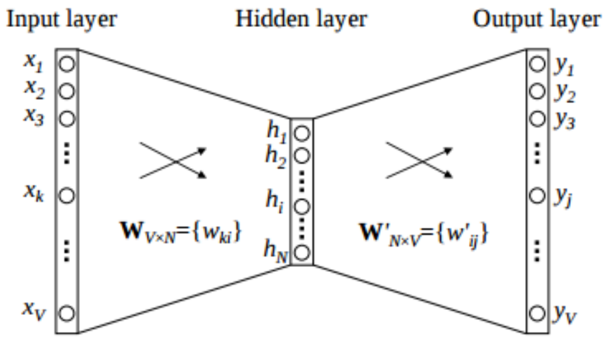


Figure 2: Single-Word Context

We have an input layer and an output layer, which are both one-hot encoded vectors of size (1 x V), where V is the size of the vocabulary. This means that the encoding for these words is in a format where all of the values in the vector are 0, except for the value corresponding to the position of the word that the vector is representing, which will be 1. The weights between the input layer and the hidden layer, the Input-Hidden weights, are represented by a (V x N) matrix, W, where N is the dimensionality of the resulting word vector representation and is a hyper-parameter that is defined when designing the CBOW model. In Figure 2, we can see that x_k is highlighted and so we will take this to be the element in the vector with value, 1. The input vector is multiplied by the weight matrix, W, and the resulting vector is simply the corresponding row, of the weight vector, relative to the position of the word in the vocabulary. What we get is a (1 x N) matrix, which is in effect our hidden layer. This implies that there is a linear activation function between the input and hidden layers; that it simply passes on the weighted sum of the input layer to the hidden layer.

$$\boldsymbol{h} = \boldsymbol{W}^T\boldsymbol{x} = \boldsymbol{W}^T_{(\boldsymbol{k},.)}$$

The weights between the hidden layer and the output layer, the Hidden-Output weights, are represented by a (N x V) matrix, W'. The hidden layer is multiplied by the j-th column in the weight matrix, W', and the resulting vector is u, where u is the score that is computed for each and every word in the vocabulary (Rong, 2016).

$$\boldsymbol{u} = \boldsymbol{W'}^T_{(.,\boldsymbol{j})}\,\boldsymbol{h}$$

Then we have a softmax activation function to obtain a multinomial distribution of words:

$$p(w_O|w_I) = \frac{\exp(u_j)}{\sum_{j'=1}^{V} \exp(u_{j'})}$$

Where w_O = output word, and w_I = input word. We then, find the output that maximises this function, and that is the focus word corresponding to our input context word. We then, calculate the error for the output and use stochastic gradient descent, and backpropagation to train this shallow neural network. The word embedding is the corresponding column for the focus word in the final weight matrix, W', after all training has converged to a solution. This word embedding of size (1 x N) is our word vector representation for our focus word.

3.1.1.2 Multiple-Word Context

We saw how CBOW works for a single-word context, however this isn't how the algorithm is implemented in practise. In reality, we use multiple context words. Let's take a look at a CBOW implementation where we take a window of three context words and try to predict, again, a single focus word.

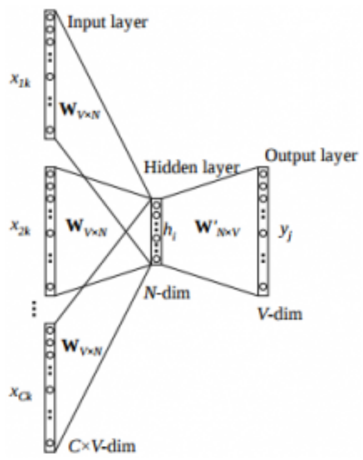


Figure 3: Multiple-Word Context

The three input vectors are multiplied by their corresponding weight matrices, W, and the resulting vectors, are again, simply the corresponding rows, of the weight vectors, relative to the position of their corresponding words in the vocabulary. What we get are 3 (1 x N) matrices. Next, what we do is take the element-wise average of all three matrices and this forms our hidden layer. The rest of the process is the same as with using a single-word context.

3.1.2 Skip-gram

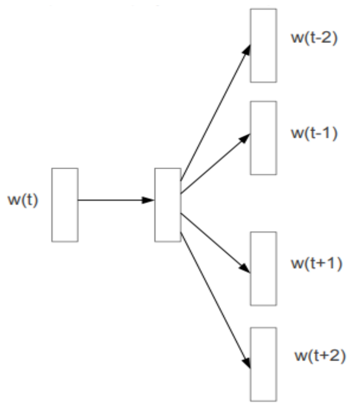


Figure 4: Skip-gram Model

The Skip-gram algorithm is another method in which to implement the Word2Vec algorithm and obtain word vector representations. It follows the same topology as CBOW, and just flips the CBOW architecture around (NSS, 2017). Given a corpus, this method takes a focus word (which is the word for which we are trying to obtain a representative vector) and tries to predict the context (the words that most often co-occur with the focus) (NSS, 2017). On a high level, what is essentially happening is that we are taking a focus word and input this into the network. We then feed this input, combined with its weight matrix, into the hidden layer. The hidden layer then is then combined with the weights between the hidden layer and the output layers, a softmax activation function is applied to each output and then finally the context words are the output. The embedding is obtained by taking the weights between the input layer and the hidden layer (NSS, 2017). The embedding is effectively our word vector representation for our focus word. Intuitively, what we have here is a distribution of context words that are most likely to co-occur with the focus word, relative to the corpus. This is effectively a probability distribution where w(t+1/-1) are the most probable context words to co-occur with the focus word.

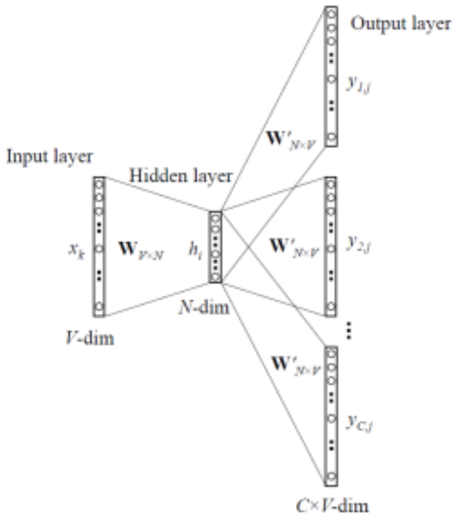


Figure 5: Skip-gram Diagram

We have an input layer and C output layers, which are both one-hot encoded vectors of size (1 x V) and C x (1 x V) respectively, where V is the size of the vocabulary. The weights between the input layer and the hidden layer, the Input-Hidden weights, are represented by a (V x N) matrix, W. In figure 5, we can see that x_k is highlighted and so we will take this to be the element in the vector with value, 1. The input vector is multiplied by the weight matrix, W, and the resulting vector is simply the corresponding row, of the weight vector, relative to the position of the word in the vocabulary. What we get is a (1 x N) matrix, which is in effect our hidden layer. This implies that there is a linear activation function between the input and hidden layer; that it simply passes on the weighted sum of the input layer to the hidden layer.

$$\boldsymbol{h} = \boldsymbol{W}^T\boldsymbol{x} = \boldsymbol{W}^T_{(\boldsymbol{k},.)}$$

Thus far, the process has been exactly the same as a single-word context CBOW model. Now is where the differences become more apparent. The weights between the hidden layer and the output layers, the Hidden-Output weights, are represented by C lots of (N x V) matrices which we denote as W'. The hidden layer is multiplied by the j-th column in the Hidden-Output matrices of the C-th output, for outputs 1 through C, and the resulting vector is $\boldsymbol{u_c}$, where $\boldsymbol{u_c}$ is the score that is computed for each and every word in the vocabulary, for every output (Rong, 2016).

$$\boldsymbol{u_c} = \boldsymbol{W'}^T_{(.,\boldsymbol{j})} \,.\boldsymbol{h}, \text{ for c = 1, 2, …, C}$$

Then we have a softmax activation function to obtain a multinomial distribution of words, for each output:

$$p(w_{O,c}|w_I) = \frac{\exp\,(u_{c,j})}{\sum_{j'=1}^{V} \exp\,(u_{j'})}, \text{for c } = 1, 2, \dots, \text{C}$$

Where $w_{O,c}$ = the C-th output word, and w_I = input word. We then find the output that maximises this function, and that is the context word, for that specific output corresponding to our input focus word. We then calculate the error for each output, sum them and use stochastic gradient descent, and backpropagation to train this shallow neural network. The word embedding is the corresponding row for the focus word in the final weight matrix, W, after all training has converged to a solution. This word embedding of size (1 x N) (after it has been transposed) is our word vector representation for our focus word.

3.2 Paragraph Vectors

Now that we have seen how both iterations of Word2Vec work, we can now explore Doc2Vec algorithms. Doc2Vec algorithms learn vector representations for each document, relative to other documents in a corpus in a multi-dimensional vector space, similarly to Word2Vec algorithms for word representations. Each document in the trained-upon corpus is represented as a dense vector which is used in a modified Word2Vec algorithm, to predict words in a specific document as opposed to the entire corpus (Mikolov and Le, 2014). These so-called paragraph vectors are fixed-length feature representations from variable-length pieces of text (Mikolov and Le, 2014). It is the property that paragraph vectors can be constructed from variable-length pieces of text, anything from a phrase or sentence to a large document, that makes it such a suitable method to make document embeddings for tweets (Mikolov and Le, 2014). Similarly, to word vectors, two documents that have similar meaning to each other will have document vector representations that are very similar to each other, whilst two documents that have very differing meanings to each other will have very different document vector representations. Also, no two documents will have the same paragraph vector unless the following condition is met: they must have the exact same words and they must occur the same number of times. Essentially, in order for ParagraphVector1 to be equal to ParagraphVector2, Document1 must be equal to Document2.

Doc2Vec consists of the Distributed Memory (DM) and Distributed Bag-of-words (DBOW) algorithms. Both of these are shallow neural networks and both of these techniques learn paragraph vectors, via stochastic gradient descent and backpropagation until convergence, which are effectively our document vector representations.

3.2.1 Distributed Memory (DM)

The DM algorithm is one of the two methods in which to implement the Doc2Vec algorithm in order to obtain document vector representations. It is based on the Word2Vec algorithm, CBOW. Essentially, given a document we take a fixed number of b context words that are sampled from a sliding window, and train the network to find an output context word. We, however, make a small addition and add another matrix, D, which we will call the paragraph ID. This additional matrix can be thought of as another context word, and its role is to be able to aid in the prediction task and provide additional information to help the network predict the next word in the context. Over numerous iterations and sampled contexts, the paragraph ID will eventually encode information that describes all the information in the document in an n-dimensional vector space. It can be thought of as a memory that remembers what is missing from the current context (Mikolov and Le, 2014). Every document is mapped to a unique vector that is represented as a column in matrix D (Mikolov and Le, 2014). Word ordering is maintained when training the network to obtain a paragraph vector in this way.

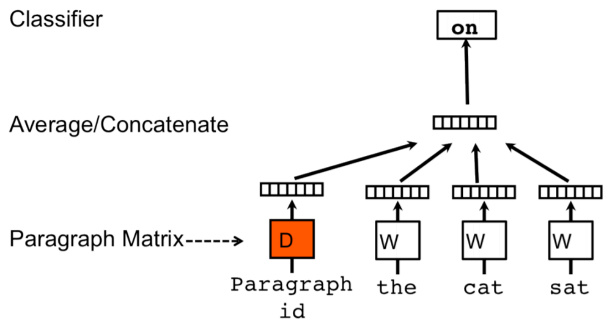


Figure 6: Distributed Memory Model

The way this works is exactly the same as a multiple-word context CBOW model, with a few additions and exceptions. At the input layer, instead of taking the element-wise average of the input word representations (multiplied with their weights), what we do is take the element-wise average of these input words representations (multiplied with their weights) and the relevant column in the paragraph ID. Another option is to concatenate instead of averaging. Hence, h is constructed from W and D (Mikolov and Le, 2014):

$$\boldsymbol{h} = \frac{\boldsymbol{W}^T \boldsymbol{x}, n + \boldsymbol{D}}{b+1} = \frac{\boldsymbol{W}^T_{(\boldsymbol{k},.),\boldsymbol{n}} + \boldsymbol{D}}{b+1}, \text{for n} = 1, 2, \dots, \text{b}$$

The rest of the process is the same as CBOW. The paragraph and word vectors are trained using stochastic gradient descent and backpropagation. When we want to infer an unseen paragraph vector, we add an additional column to D to represent the new document. We then fix the values of the word vectors, and the softmax weights and train the paragraph vector, via stochastic gradient descent and backpropagation, until convergence. (Mikolov and Le, 2014).

3.2.2 Distributed Bag-of-words (DBOW)

The DBOW algorithm is the other of the two methods in which to implement the Doc2Vec algorithm in order to obtain document vector representations. It is based on the Word2Vec algorithm, Skip-gram. Essentially, what we are doing is taking the Skip-gram architecture but instead of inputting a focus word representation, we are inputting the paragraph ID; the relevant column in matrix D. What this does is, force the model to predict the set of context words that are randomly sampled from the specific paragraph corresponding with the input paragraph vector (Mikolov and Le, 2014). At each iteration, we keep sampling a random window of context words, and perform stochastic gradient descent using backpropagation. What this does is encode all the information from these random samplings, and when the paragraph vector converges, will be able to randomly sample a window of words from that paragraph, and that paragraph only. Intuitively, what this means is that the paragraph vector will learn and encode all of the information that makes the document in question different from all other documents. Every document is again mapped to a unique vector that is represented as a column in matrix D (Mikolov and Le, 2014). Word ordering is not maintained when training the network to obtain a paragraph vector in this way. Also, this model does not train any word vectors, unlike DM, and so does not need to store any, requiring the model to store less data making it much smaller in size than a DM model (Mikolov and Le, 2014).

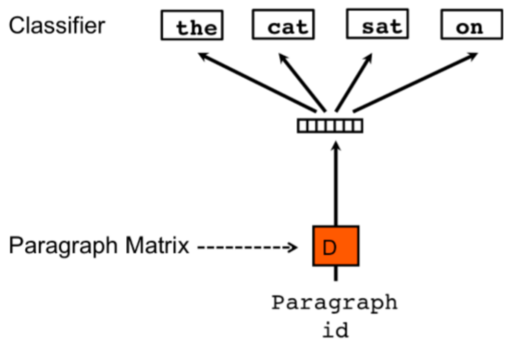


Figure 7: Distributed Bag-of-words Model

The way this works is exactly the same as a Skip-gram model, with a few additions and exceptions. At the input layer, instead of taking the element-wise average of the input word representations (multiplied with their weights), what we do is take the paragraph ID, which if we remember correctly is the corresponding column in matrix D for the document in question, and multiply this with the weight and then this forms our hidden layer.

$$\boldsymbol{h} = \boldsymbol{W}^T\boldsymbol{D}$$

Where D = paragraph ID. The rest of the process is the same as Skip-gram. The paragraph vector is trained using stochastic gradient descent and backpropagation. When we want to infer an unseen paragraph vector, we add an additional column to D to represent the new document. We then fix the values of the softmax weights and train the paragraph vector until convergence via stochastic gradient descent and backpropagation (Mikolov and Le, 2014).

3.3 Classifiers

In order to be able to predict the sentiment of our tweets using our document representations, we need to be able to classify them. There are various ways to do this, each with their own performance benefits and trade-offs, and we are going to explore each of these. In this project, I am going to use a number of classifiers. A classifier is a machine learning algorithm or function that is used to classify particular data points (Raschka, no date). It does this by utilizing some form of training data to understand how any given input variable relates to a class (Asiri, 2018). The classification algorithms I decided to use for this project are Logistic Regression, Cosine Similarity, Linear Discriminant Analysis, and a Multi-Layer Perceptron. All of these methods will help separate our positive tweets, from the negative ones and allow us to be able to then classify unseen tweets.

3.3.1 Logistic Regression

Logistic regression is a parametric classification model that, contrary to what the name implies, given an input, is able to output a categorical prediction (Zornoza, 2020). This means that the kind of data that you are trying to represent has an independent variable (input) that is continuous and a dependent variable (output) that is discrete. In this case, we will be using binary logistic regression which means that there will only be two output classes; 1 and 0, or in our case, positive and negative. So, in layman terms, given a data-point, we are going to predict whether it belongs to one class or the other.

The way this works is by using something called a sigmoid function:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Where x is the weighted input feature (the hypothesis) (Swaminathan, 2018):

$$x = \theta a + b$$

Where a = input feature. So, given some data that is split into discrete classes, we want to fit a sigmoid function to the datapoints. For every given feature, you find the weighted sum, and then input this into the sigmoid function to obtain your classification. Since sigmoid functions are continuous between 0 and 1, it is always going to be the case that our output of the sigmoid function is going to be between 0 and 1, but never actually 0 or 1. This forms a probability distribution where the output essentially corresponds to the probability of the datapoint belonging to the classification corresponding to 1. A cut-off is usually defined at 0.5. For any value over 0.5, we would say the datapoint belongs to class 1, and 0 otherwise. The sigmoid function is fit to the data using maximum likelihood estimate (MLE), where the following parameters are trained: θ and b (Brownlee, 2016a). We then use these values to construct x, for which we can apply it to new datapoints and calculate the classification using the sigmoid function.

3.3.2 Cosine Similarity

Cosine Similarity measures the similarity of two vectors in a vector space, and it is done by taking the cosine of the angle between two vectors, thus determining how 'similar' they are (Han et al, 2012). The approach I am taking with this is a simple one and it essentially consists of 3 steps:

- Find the average vector for all positive tweets (AvgPos) and all negative tweets (AvgNeg).
- Compare any given tweets vector representation with the AvgPos and AvgNeg vectors using cosine similarity.
- If the similarity value with AvgPos is greater than the similarity value with AvgNeg then mark as positive, or else negative otherwise.

The way cosine similarity in itself works is by taking two vectors, normalising them and then returning a measure of how similar they are based on how closely they point in the same direction. It does this by working out the angle between the two vectors. We do this by using the dot product formula (Prabhakaran, 2018).

$$\vec{a} \cdot \vec{b} = ||\vec{a}||\; ||\vec{b}|| \cos\theta$$

We then solve for $\cos\theta$:

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{||\vec{a}||\; ||\vec{b}||}$$

$\cos\theta$ is our measure of how similar two vectors are.

3.3.3 Linear Discriminant Analysis

Linear Discriminant Analysis is a dimensionality reduction technique, and as is implied by the name of the technique, reduces the number of dimensions in a dataset whilst retaining as much info as possible (Maklin, 2019). The goal here is to project a dataset onto a lower-dimensional space that has the highest class-separability (Raschka, 2014). This is to say that, in our case, we want to reduce a complex multi-dimensional dataset into a single dimension, and then separate these into two classes, from which we will then be able to determine the class allocation of unseen data. The way that we do this is that, in the case of two classes, we find an axis that maximises the distance between the means of the two classes (between-class variance), whilst simultaneously minimizing the scatter/range of each of the classes (within-class variance).

In order to work out the within-class variance, we use the following formula:

$$S_W = \sum_{i=1}^{c} S_i$$

Where S_i is the scatter matrix for each class, and c corresponds to the number of classes:

$$S_i = \sum_{x \epsilon D_i}^{n} (x - m_i)(x - m_i)^T$$

To work out the scatter matrix for each class what we need to do is firstly calculate the mean of each class using:

$$m_i = \frac{1}{n_i} \sum_{x \epsilon D_i}^{n} x_k$$

We then, for each datapoint, work out the Euclidean distance between the mean, and that given datapoint resulting in an n-dimensional vector, where n is the number of features/dimensions that describe our data. We then multiply the vector by its transpose, to get a n x n matrix. We then sum all of these matrices up, corresponding to each datapoint to get our S_i matrix. We then further sum up all of the scatter matrices for each class and we are left with our S_W matrix which is our within-class variance.

In order to work out the between-class variance, we use the following formula:

$$S_B = \sum_{i=1}^{c} N_i (m_i - m)(m_i - m)^T$$

What we are essentially doing here is calculating the mean values for each class, as before, but also calculating a global mean value using:

$$m = \frac{1}{n} \sum_{i}^{n} x_i$$

What we then do is, for each class, we work out the Euclidean distance from the mean corresponding to the class and the global mean, and then multiply that with its transpose to get a n x n matrix. We then sum all of these matrices up, corresponding to each class to get our S_B matrix.

We combine both of these:

$$S_W^{-1} S_B$$

We can then use the resulting matrix to calculate our projection vector using eigenvectors. We want to find the axis that maximises the between-class variance, whilst minimizing the within-class variance because intuitively what we want is an axis that has the mean values of both classes as far apart as possible, whilst keeping the scatter to a minimum so there is reduced overlapping, which in turn allows for good separability. Keeping this in mind, what we then do is find the eigenvector of our matrix that corresponds to the largest eigenvalue. This satisfies our conditions and gives us our projection vector, W, and we can plug that into the following equation:

$$Y = X \cdot W$$

Where X is our input data, and Y is our new feature space (Maklin, 2019). This is how we get our low-dimensional feature representation where class separability is at a maximum. In order to actually make predictions, we use Bayes theorem to make probabilistic classifications (Brownlee, 2016b).

3.3.4 Multi-Layer Perceptron

A multi-layer perceptron is a type of neural network, that consists of at least 3 layers including, an input layer, n-hidden layers, and an output layer (Kang, 2017). Each layer in this inter-connected network consists of a number of nodes, and the job of these nodes is to propagate data forward through the network, whilst being modified by weights and activation functions along the way, so that something can be output. In our use case, we will be using these networks for classification, and so our network will have an output layer consisting of one single node that will output a categorical classification, given some data. The reason neural networks like this are being increasingly used for classification tasks is because of accuracy. Not all classification problems have datasets that are separable linearly, for example the XOR problem. Neural networks are able to overcome this problem by defining and learning a decision boundary that is non-linear (Rocca, 2018). So essentially how a multi-layer perceptron works is by having some data being input into the input layer, which will have m-nodes. The data input into each node will then be augmented with some parameters; a weight value, w, and in some cases a bias, b, too:

$$\sum_{i=1}^{m}(x_i w_i) + b$$

This will be then carried forward to a node in the next layer, for each and every input node. Every node in the hidden layer will then be updated in this way, where each node will have a distinct bias value. We then come across something called an activation function. The nature of these activation functions varies depending on the kind of problem you are attempting to solve, however their role is very simple; to govern a threshold at which a node is activated (Brownlee, 2016c). This is to say that we define a minimum value that a node has to meet in order to be further propagated through the network, otherwise it is not and generally this is handled by simply setting the output of this node to 0 (Brownlee, 2016c). This is then repeated over and over for every hidden layer, which can have varying sizes of nodes. Then, we reach the output layer where, similarly to before we calculate an output, via an activation function. This tends to be the sigmoid function for binary classification problems (Brownlee, 2016c). We then take this computed output, compare it against the expected output and calculate an error value. This value can be used to perform something called backpropagation, which is the 'most fundamental building block in a neural network' (Kostadinov, 2019). What this means is that for every forward pass through the network, backpropagation, propagates the error values back through the network, to be able to update the network parameters and to essentially help the network learn. We want to minimise this error as much as possible and we do that with a process called gradient descent. We can also implement optimisation algorithms such as Adam, which is 'a method for efficient stochastic optimization that only requires first-order gradients' (Kingma and Ba, 2015).

Chapter 4 - **Experiments**

4.1 The Dataset

I'm using a dataset compiled by Stanford University researchers, that was used to train their own closed-source project called 'Sentiment140'. It contains 1.6 million tweets; 800,000 of which are positively tagged, and 800,000 of which are negatively tagged. The data was collected via the Twitter API, and the query function was taken advantage of to be able to collate mass amounts of positive/negative tweets. The data was collected with minimal human input and each tweet was tagged via the emoticon data associated with the tweet. 'In the Twitter API, the query ":)" will return tweets that contain positive emoticons, and the query ":(" will return tweets with negative emoticons' (Go et al, 2009). The tweets that corresponded with ':)' were marked as positive, and the tweets that corresponded with ':(' were marked as negative. If a tweet contained both, it was simply discarded (Go et al, 2009). The emoticons, retweets, and duplicates are then removed. Each tweet in the dataset has a corresponding tag; 0 if negative, and 4 if positive. The average length of a tweet in this dataset is 78 characters, or 14 words.

4.1.1 Dataset Pre-Processing

In order to train the models on this data, a lot of processing needed to be done in order to get the data in such a state that it would be suitable to be used to build document embeddings. For each tweet I did the following:

- Removed @ mentions.
- Removed links to websites in both the 'http:' and 'www.' formats.
- Used BeautifulSoup (with lxml parser) to remove all HTML encoding and convert this to raw text.
- Removed UTF-8 BOM/SIG characters that were present in some tweets.
- Converted all text to lower case.
- Removed all characters that did not satisfy the following bounds: A-Z and a-z.
- Set minimum length of words in the text to 2. All words with length less than 2 were discarded.
- Tokenized all tweets, and then re-appended in order to get rid of any unusual white space that was sometimes present.
- I also changed the positive tag from 4 to 1, since it was more intuitive to work with.

I, then, re-organised the CSV file, such that it only consisted of the, now processed, tweet and its corresponding sentiment tag; 0 or 1. All other information was simply discarded.

4.2 Experiment 1 - Varying the number of tweets used to build the Doc2Vec models

In this experiment, I want to investigate whether varying the amount of data that I use to build the paragraph vector representations will affect the accuracy of predicting the sentiment of any given tweet. I will use logistic regression, cosine similarity, and linear discriminant analysis to evaluate these models. I did this for both the DBOW and DM models, as well as a combined model which concatenates both models together. The way in which I did this was to keep all other hyper-parameters constant when training the Doc2Vec models. In my case, I did the following:

- I set the size of the dimension of the vector to 100.
- I trained each model on 30 epochs.
- I used the same hardware to train these models, and hence the same number of cores.
- I also set the initial alpha value (the learning rate) to 0.065, decreasing by 0.002 after each epoch.

Hence, this way the only variable that will change for each model is the amount of data that we use to train them. I decided to use the following numbers of tweets, where the proportion of positive:negative tweets are 50:50: 1,000; 10,000; 100,000; 250,000; 500,000; 1 million and finally, the entire dataset of 1.6 million tweets. I randomly allocated all of the tweets to their respective datasets, from either a pool of positive or negative tweets, and then combined them. My hypothesis for this experiment is that, as the amount of data used to train the data increases, the accuracy of predicting the sentiment will also increase. This, in my opinion, will only happen up to a point, after which any additional data thrown at the models will have a negligible effect. This is because, I feel that as once the models have enough data, the amount of data will no longer be the limiting factor. I expect to see a plateau form after 500,000 tweets. I expect to see that the Combo model will perform the best from all three models since it holds the most encoded information for each tweet.

4.3 Experiment 2 - Varying the dimensionality of the Doc2Vec models

In this experiment, I want to investigate whether varying the dimensionality of the paragraph vector representations will affect the accuracy of predicting the sentiment of any given tweet. I will use logistic regression, cosine similarity, and linear discriminant analysis to evaluate these models. I did this for both the DBOW and DM models, as well as a combined model which concatenates both models together. The way in which I did this was to keep all other hyper-parameters constant when training the Doc2Vec models. In my case, I did the following:

- I set the number of tweets I used to 1.6 million; 800,000 positive and 800,000 negative.
- I trained each model on 30 epochs.
- I used the same hardware to train these models, and hence the same number of cores.
- I also set the initial alpha value (the learning rate) to 0.065, decreasing by 0.002 after each epoch.

Hence, this way the only variable that will change for each model is the dimensionality of the resulting paragraph vectors. I decided to train Doc2Vec models in the following dimension sizes: 10; 25; 50; 75; 100 and finally, 200. My hypothesis for this experiment is that, as the best number of dimensions to use will be 50. This is because I feel that, around 50 dimensions is enough for all the data for any given tweet to be effectively encoded, since they are very short documents that consist of, as previously mentioned, an average of 14 words. I hypothesise that as I increase the number of dimensions beyond this, the accuracy will fall due to picking up a lot of 'noise' that will have an adverse effect on the integrity of the data encoded in the paragraph vectors. I expect to see similar behaviour from both models, and I expect to see that there will be a bell curve where the accuracy will steadily increase until 50 dimensions, and then fall. I, again, expect to see that the Combo model will perform the best from all 3 models since it holds the most encoded information for each tweet.

4.4 Experiment 3 – Using averaged word vectors and comparing with Doc2Vec

In this experiment, I want to investigate how much of a difference in performance there is in using averaged word vectors to represent tweets, instead of the more specialised paragraph vector representations. I will use logistic regression, cosine similarity, and linear discriminant analysis to evaluate these representations. Instead of training a Word2Vec model from scratch, I extracted the word vectors from some pre-trained DM and DBOW Doc2Vec models. The way in which I did this was to extract the word vectors directly from the model, by indexing the words in a given tweet directly against the model:

$$wordvector = doc2vecmodel[word]$$

This works differently for DM and DBOW. For DM, the word vectors are co-trained alongside the paragraph vectors, and so the extracted word vectors are raw representations. For DBOW however, word vectors are not co-trained alongside the paragraph vectors and so these extracted word vectors are inferred by the model and are not raw representations. I, then, averaged all the word vectors in a given tweet and returned this as a paragraph representation. The Doc2Vec models from which these word vectors were extracted met the following conditions:

- I set the number of tweets I used to 1.6 million; 800,000 positive and 800,000 negative.
- I set the size of the dimension of the vector to 100.
- I trained each model on 30 epochs.
- I used the same hardware to train these models, and hence the same number of cores.
- I also set the initial alpha value (the learning rate) to 0.065, decreasing by 0.002 after each epoch.

My hypothesis from this experiment is that the averaged word vector-based paragraph representations extracted from DM will outperform, its DBOW counterpart. This is because in the DM model, the word vectors are explicitly trained, whereas in DBOW they are not. Because of this, my expectation would be that the DM-based word vectors would more accurately encode the information required to represent a word. After averaging, the DM-based word vector-based paragraph representations should be able to outperform the DBOW equivalent. However, I feel that neither of these models will be able to outperform paragraph vectors in being able to represent documents, and hence the accuracy of sentiment prediction will be expected to be significantly lower.

4.5 Experiment 4 – Varying depth and size of hidden layers in Multi-Layer Perceptron classifiers

In this experiment, I am going to investigate the what the best configuration is for a multi-layer perceptron network, that will be used to predict the sentiment of any given tweet. I will be running the neural network classifier on both the DBOW and DM models, as well as a combined model which concatenates both models together (Combo model). The way in which the networks will be implemented is by setting up the perceptron in the following way:

- An input size of 100 nodes will be used which corresponds to the dimensionality of the DM and DBOW models, and an input of size 200 will be used for the Combo model.
- A Rectified linear unit (ReLu) activation function will be used for the input, and all hidden layers.
- The output layer will use a sigmoid activation function and will have a single node.
- An Adam optimizer will be used, as well as a Binary Cross-Entropy loss function. The Binary Cross-Entropy loss function will be used because it is most suitable for multi-label classification (Gomez, 2018).

The Doc2Vec models which were used with the multi-layer perceptron classifiers met the following conditions:

- I set the number of tweets I used to 1.6 million; 800,000 positive and 800,000 negative.
- I set the size of the dimension of the vector to 100.
- I trained each model on 30 epochs.
- I used the same hardware to train these models, and hence the same number of cores.
- I also set the initial alpha value (the learning rate) to 0.065, decreasing by 0.002 after each epoch.

I will be using multiple configurations including: 256x3, 128x3, 256x1 and finally 128x1, where the format AxB denotes [Number of nodes per layer]x[Number of layers]. My hypothesis for this experiment is that the 256x3 model will perform the best, followed by the 128x3, then the 256x1 and then the 128x1 model. My reasoning for this is that the more layers we have, the more complex decision boundaries can be established, and overfitting will be reduced (Goodfellow, Bengio and Courville, 2016), and so the better accuracy we will get when calculating sentiment of an unseen tweet. I think the increase in number of nodes will increase performance due to the ability to further abstract the data as much as possible and using this level of abstraction to pinpoint key properties about the dataset that will allow for better separability.

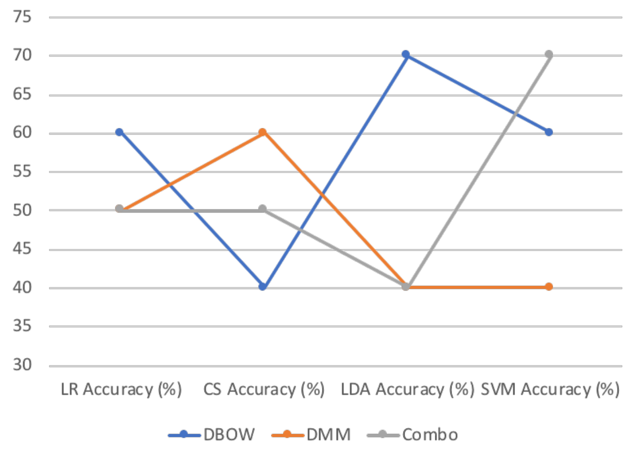
Chapter 5 – **Results and Discussion**

5.1 Experiment 1 - Varying the number of tweets used to build the Doc2Vec models

These are the results I collected for this experiment. I ran the code three times, and if the results differed for each run, then I took the average of the three results. The graph axes have been scaled to best show the differences between the variables.

For 1,000 tweets:

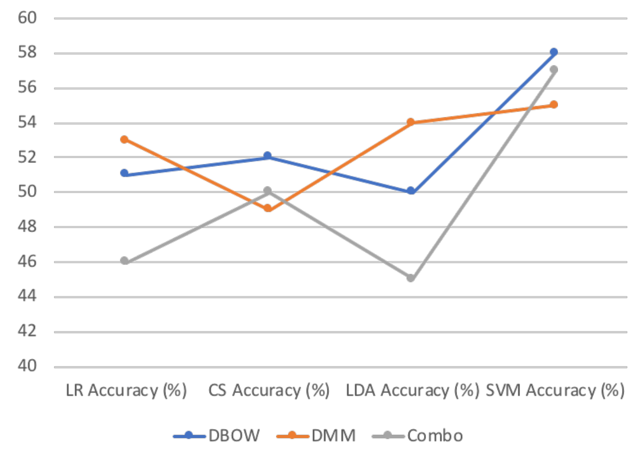
Model	LR Accuracy (%)	CS Accuracy (%)	LDA Accuracy (%)	SVM Accuracy (%)
DBOW	60	40	70	60
DMM	50	60	40	40
Combo	50	50	40	70



I expected the results for 1,000 tweets to be poor, and we can see that the results are very erratic, ranging from 40% accuracy all the way up to 70% accuracy. We can see that using 1,000 tweets was far too low for the model to actually construct any meaningful representations, and all the results are due to chance, hence the erratic nature of the results. I also decided to use an SVM classifier for this, since the dataset was so small. I used an SVM with hyperparameters tuned by Optuna, a hyperparameter optimization framework.

For 10,000 tweets:

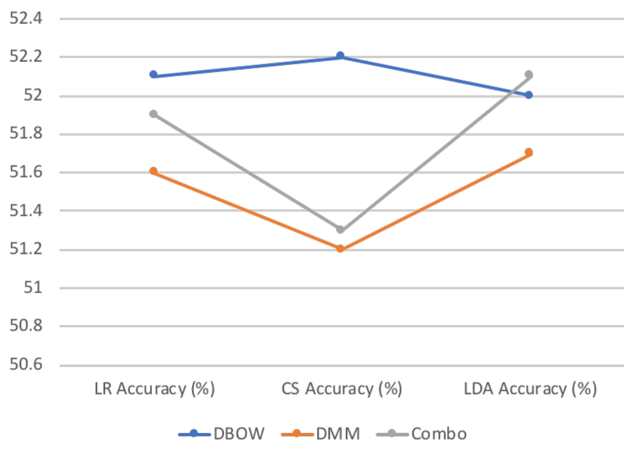
Model	LR Accuracy (%)	CS Accuracy (%)	LDA Accuracy (%)	SVM Accuracy (%)
DBOW	51	52	50	58
DMM	53	49	54	55
Combo	46	50	45	57



Here we can see that the results are starting to become more legible, and that some patterns can start to be observed. We can see that, in general, the DM and DBOW models perform similarly, whilst the Combo model performs the worst. This goes against my expectation that the Combo model would perform the best. I believe that the reason for this is because there simply isn't enough data for the additional data the Combo model possesses to have any positive effect on accuracy. Instead it only adds 'noise' which harms accuracy. I, again, used the SVM classifier and as expected it performed the best. This is because an SVM classifier can define non-linear decision boundaries, which I believe helped significantly in this case.

For 100,000 tweets:

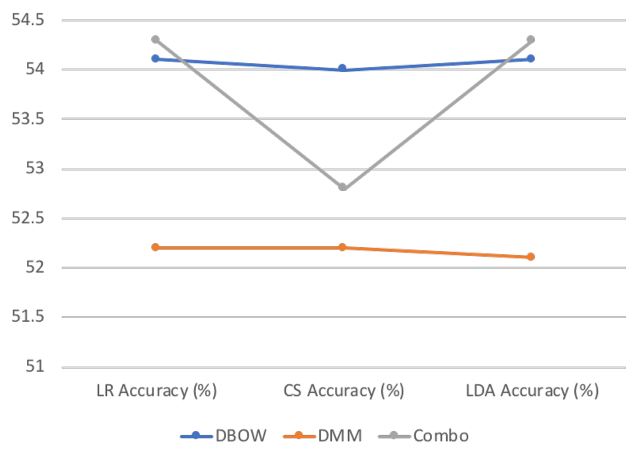
Model	LR Accuracy (%)	CS Accuracy (%)	LDA Accuracy (%)
DBOW	52.1	52.2	52.0
DMM	51.6	51.2	51.7
Combo	51.9	51.3	52.1



Here we can see that in general, the results are an improvement over using just 10,000 tweets, and as expected, thus far, accuracy has increased. We can see that, in general, the DBOW results are best, with it also having the highest accuracy with the CS classifier. The other results, however, suggest that this is due to chance since, as expected, the other models perform better with the more sophisticated LR and LDA classifiers. In general, we can see that LDA performed the best, with LR closely behind with CS performing the worst, if we ignore the anomalous result. An SVM was unable to be used here due to me not

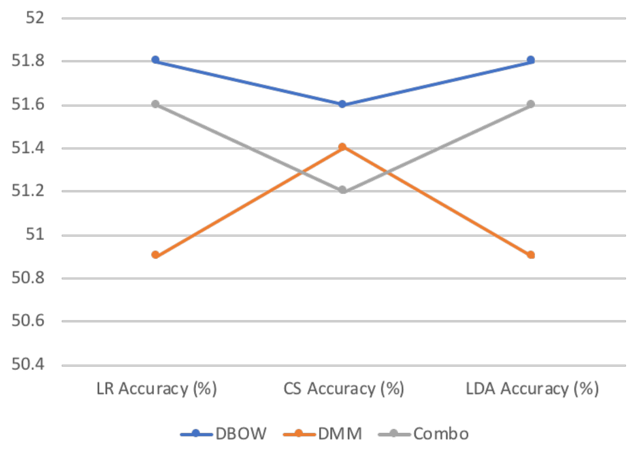
having access to the computational power needed to train the classifier on a dataset this vast. I did not use any SVMs in any experiments beyond this due to lack of computational resources required to train them.
<u>For 250,000 tweets:</u>

Model	LR Accuracy (%)	CS Accuracy (%)	LDA Accuracy (%)
DBOW	54.1	54.0	54.1
DMM	52.2	52.2	52.1
Combo	54.3	52.8	54.3



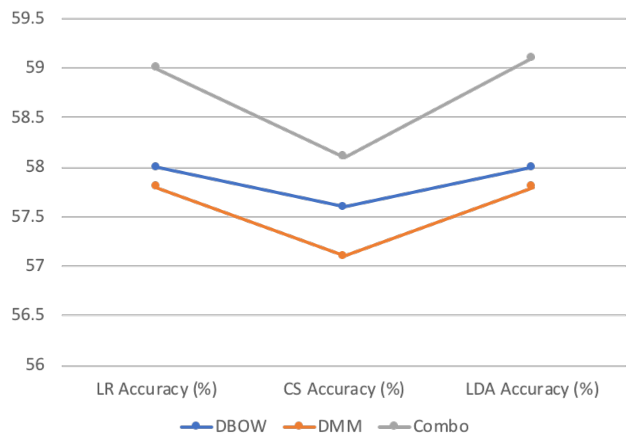
Here we can see that in general, the results are, again, a slight improvement over using just 100,000 tweets, and as expected, thus far, accuracy has increased again. We can see that the Combo results are best, with a dip in accuracy with the CS classifier. This is as expected, since the CS classifier is a much more naïve classifier compared to the others. The DBOW results are more consistent, as are the DM results. There is, however, a significant drop in accuracy with the DM model, and this is because of the nature of the dataset and I expect this to be consistent through all further experiments since DM models tend to perform better on longer documents, whereas DBOW performs better with shorter documents (Ai et al, 2016).
<u>For 500,000 tweets:</u>

Model	LR Accuracy (%)	CS Accuracy (%)	LDA Accuracy (%)
DBOW	51.8	51.6	51.8
DMM	50.9	51.4	50.9
Combo	51.6	51.2	51.6



The results here are surprising to me, since I had anticipated that the results at this point would represent the point at which performance would peak and plateau off, whilst we can see that this is definitely not the case and goes against my hypothesis. What is more surprising to see is that the results, as a whole are lower than all the previous results and are the worst results thus far. We can see that here the DBOW model performed the best, whilst again, the DM model performed the worst, with the Combo model sitting comfortably in-between the two. The CS classifier results are going against the norm, further suggesting that there is not enough data to be able to accurately represent the tweets, and hence classify them.
<u>For 1 Million tweets:</u>

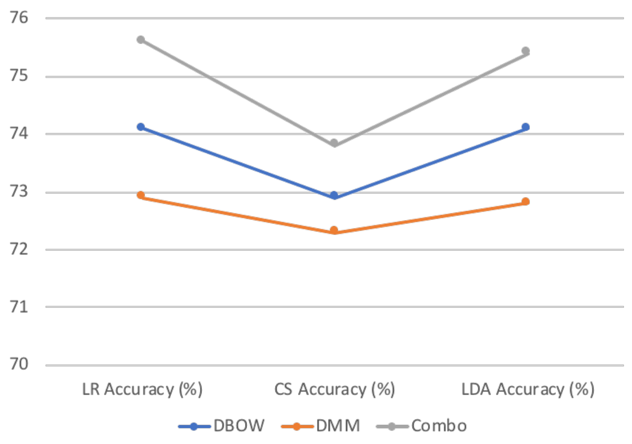
Model	LR Accuracy (%)	CS Accuracy (%)	LDA Accuracy (%)
DBOW	58.0	57.6	58.0
DMM	57.8	57.1	57.8
Combo	59	58.1	59.1



Here we can see that in general, the results are, a significant improvement over the previous datasets, and against my hypothesis and expectations, accuracy has increased again. We can see that, in general, the Combo results are best across the board. This is expected and suggests to me that we finally have enough data to accurately represent the dataset, since the Combo results are no longer hindered. The DBOW results are more consistent, as are the DM results.

For 1.6 Million tweets:

Model	LR Accuracy (%)	CS Accuracy (%)	LDA Accuracy (%)
DBOW	74.1	72.9	74.1
DMM	72.9	72.3	72.8
Combo	75.6	73.8	75.4

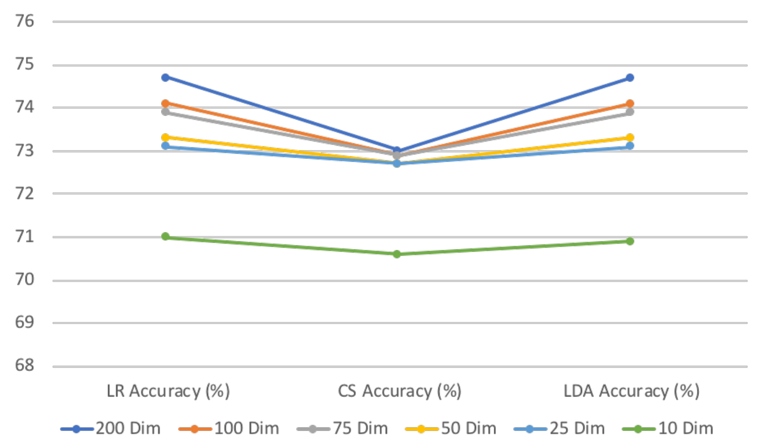


Finally, here we can see that the results are the best so far. We can see that we have obtained an accuracy score of 75.6% for the Combo model, using the LR classifier which is by far the best result we have seen so far. The results across the board are better and more consistent than what we have seen before, and this suggests to me that we finally have enough data to be able to construct a model that accurately represents the tweets in such a way that they are more easily separable, in terms of sentiment. These results go against my initial hypothesis that the models would improve, given a larger dataset, until 500,000 tweets and then plateau off. I can infer that the point at which the results would plateau off would be somewhere between 1 million, and 1.6 million tweets, inclusive.

5.2 Experiment 2 - Varying the dimensionality of the Doc2Vec models

These are the results I collected for this experiment. I ran the code three times, and if the results differed, I took the average. Graph axes have been scaled to best show the differences between the variables.
For DBOW:

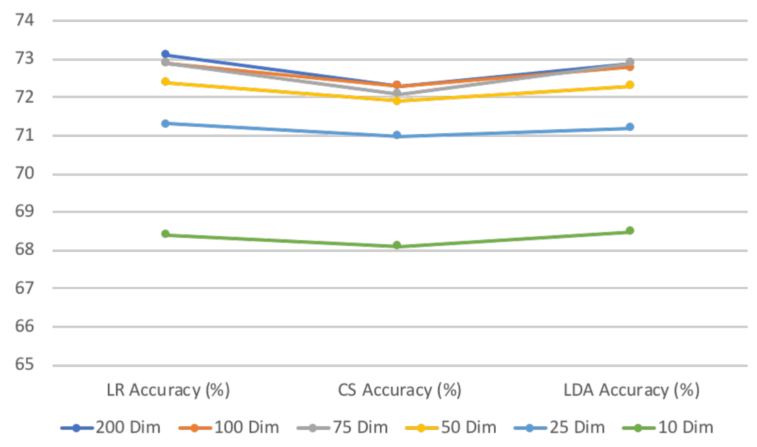
Model	LR Accuracy (%)	CS Accuracy (%)	LDA Accuracy (%)
200 Dim	74.7	73.0	74.7
100 Dim	74.1	72.9	74.1
75 Dim	73.9	72.9	73.9
50 Dim	73.3	72.7	73.3
25 Dim	73.1	72.7	73.1
10 Dim	71.0	70.6	70.9



Here we can see that all the results seem to be close together, with there only being a significant difference with the 10-dimensional vector. All other performance benefits after this seem to be marginal. We can see here that the LR and LDA classifiers perform identically, with there being a significant accuracy decrement with CS, as expected. The results, here, go against my hypothesis in that the accuracy scores for dimensions higher than 50 continue to increase, although not by any significant margin,

For DM:

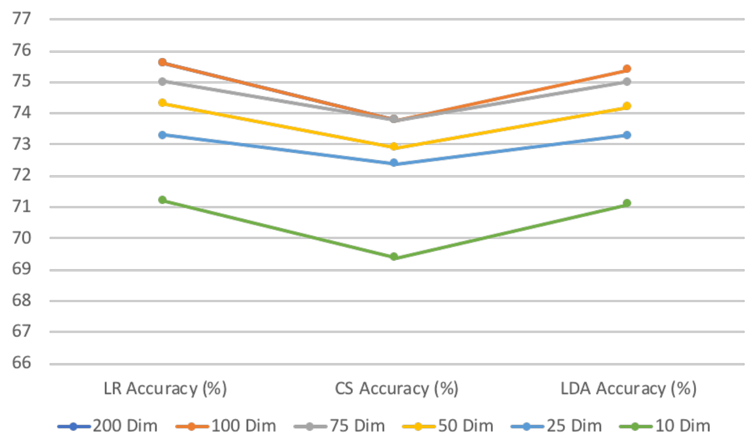
Model	LR Accuracy (%)	CS Accuracy (%)	LDA Accuracy (%)
200 Dim	73.1	72.3	72.9
100 Dim	72.9	72.3	72.8
75 Dim	72.9	72.1	72.9
50 Dim	72.4	71.9	72.3
25 Dim	71.3	71	71.2
10 Dim	68.4	68.1	68.5



Here we can see that all the results seem to be very similar to the DBOW model, in terms of the trends and structure. There does, however, seem to be a larger difference in accuracy with the 25-dimensional vector. Overall however, the results are slightly lower than for DBOW. The results, again, go against my hypothesis in that the accuracy scores for dimensions higher than 50 continue to increase.

For Combo (DBOW + DM):

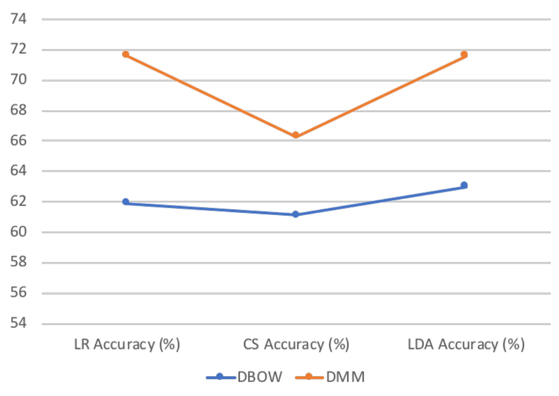
Model	LR Accuracy (%)	CS Accuracy (%)	LDA Accuracy (%)
200 Dim	75.6	73.8	NULL
100 Dim	75.6	73.8	75.4
75 Dim	75.0	73.8	75.0
50 Dim	74.3	72.9	74.2
25 Dim	73.3	72.4	73.3
10 Dim	71.2	69.4	71.1



Here we can see, again, that all the results seem to be very similar to the other models, in terms of trends and structure. There does, however, seem to be a larger disparity in accuracy with the different vectors than before. Overall however, the results are slightly higher than for the other two models. We can also see that we get a NULL value for LDA on the 200-dimensional vector. This value could not be obtained, despite numerous attempts, because it kept crashing the machine that I was training the classifier on. Although, we can infer from the other results that this would be 75.4% - The exact same result as the 100-dimensional vector.

5.3 Experiment 3 - Using averaged word vectors and comparing with Doc2Vec

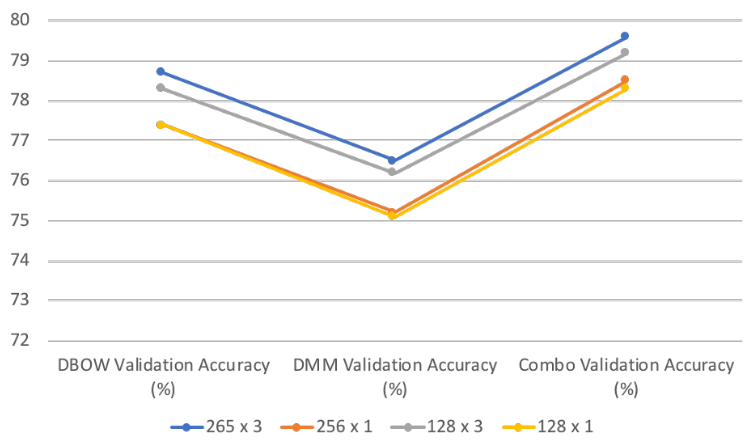
Model	LR Accuracy (%)	CS Accuracy (%)	LDA Accuracy (%)
DBOW	69.1	61.1	63
DMM	71.6	66.3	71.6



These results support my hypothesis completely, in that the DM model outperforms the DBOW model, by quite some significant margin. My hypothesis also stated that neither of these models will be able to outperform true paragraph vectors, which can be shown to be true as both of these models have significantly less accuracy than true paragraph vector models.

5.4 Experiment 4 - Varying depth and size of hidden layers in Multi-Layer Perceptron classifiers

Nodes x Layers	DBOW Validation Accuracy (%)	DMM Validation Accuracy (%)	Combo Validation Accuracy (%)
265 x 3	78.7	76.5	79.6
256 x 1	77.4	75.2	78.5
128 x 3	78.3	76.2	79.2
128 x 1	77.4	75.1	78.3



These results support my hypothesis completely, in that the 256x3 network performed the best, followed very closely by the 128x3 network. The 256x1 network, and the 128x1 networks follow, with a fairly significant drop in accuracy. The 256x3 and 128x3 networks are almost identical in performance, whilst the 256x1 and 128x1 networks are almost identical too. This suggests that the number of nodes has a very insignificant effect on accuracy, and that the number of layers play a much bigger role, as I had hypothesised. In general, the Combo model performed best, followed by the DBOW model, and then the DM model as expected. This experiment has also given me the best performance I have achieved so far in any classifier/model combination, of an accuracy score of 79.6% with the Combo model (1.6 million tweets, 100 dimensions), and 256x3 MLP classifier.

Chapter 6 - **Conclusion**

This paper has explored methods that varied how I could apply different document embedding techniques, and it explored the different classification techniques that I employed in order to define a decision boundary on various datasets that would allow me to calculate the sentiment of unseen tweets. I have looked at a number of methods on how I could go about deciding what the best document embedding method was, that could accurately and reliably represent my huge dataset of 1.6 million tweets. I concluded that the best way to do this was to use a combined vector representation of each tweet, that consisted of a DBOW representation concatenated with a DM representation. I also concluded that the best dimensionality to use was 100, since it offered the best accuracy and wasn't as computationally expensive to use as the 200-dimensional vectors, which offered very little in the way of performance benefits anyway. I also found that the best classifier to use was a multi-layer perceptron consisting of 3 hidden layers, and 256 nodes in each layer. This gave me an accuracy of 79.6%. Although the 128x3 classifier was only 0.4% behind in accuracy, at 79.2% and is less computationally expensive to train and use, this small difference could be worth £Millions to companies for who this kind of data is of value.

I think this was an interesting problem to investigate, since typically sentiment analysis investigations tend to focus on corpora that tend to have much, much longer documents. I decided to work with tweets, which means that the documents I was working with had a maximum size of 280 characters, whilst in my dataset the average length was a mere 78 characters. Tweets are also unique in the way that the physical size of the corpus is almost limitless and continues to grow on the order of 10s of millions of new tweets every single day. Tweets also don't follow a traditional language model, where misspellings, abbreviations, emoticons, and slang are the norm. Also, unlike traditional corpora, there is no fixed domain or subject matter and tweets can quite literally be about anything. All of these factors, and more, make tweets a more difficult and more interesting type of data to work with on these types of problems.

I think some ways in which I could have improved my accuracy was by using a different dataset, since the dataset I used was so crudely annotated. I think using a dataset of this type put a cap on the kind of accuracy I could expect to receive, since if there are errors in the tagging of the dataset, this will be propagated into my models. I found that in manually testing some tweets, the most success I had was in using tweets that were obviously positive, or obviously negative. When I tried to use tweets that were not so obviously either positive or negative, the system tended to struggle. For example, when I used tweets that exhibited elements of sarcasm. I attributed this to the dataset I used to train the system. I estimate that the accuracy of the dataset itself was around 85%, thus capping my maximum potential accuracy at 85%. In this context, the results I achieved seem even more impressive.

Overall, I believe I was successful in solving the problem I set out to. I found the best way to represent tweets and found the best way to classify them with a high degree of accuracy. This was, in my case, using a Combo Doc2Vec model (DBOW + DM) trained on 1.6 million tweets, with a dimensionality of 100, trained on a 256x3 MLP classifier to get an accuracy of 79.6%.

References

Ai, Q., Yang, L., Guo, J., and Croft, W.B. (2016) *Analysis of the Paragraph Vector Model for Information Retrieval.* Massachusetts, USA: University of Massachusetts Amherst.

Asiri, S. (2018) *Machine Learning Classifiers.* Available at: https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623 (Accessed: 9 March 2020)

Brownlee, J. (2016a) *Logistic Regression for Machine Learning.* Available at: https://machinelearningmastery.com/logistic-regression-for-machine-learning/ (Accessed: 20 March 2020)

Brownlee, J. (2016b) *Linear Discriminant Analysis for Machine Learning.* Available at: https://machinelearningmastery.com/linear-discriminant-analysis-for-machine-learning/ (Accessed: 21 March 2020)

Brownlee, J. (2016c) *Crash Course on Multi-Layer Perceptron Neural Networks.* Available at: https://machinelearningmastery.com/neural-networks-crash-course/ (Accessed: 23 March 2020)

Brownlee, J. (2019) *Supervised and Unsupervised Machine Learning Algorithms.* Available at: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/ (Accessed: 8 April 2020)

El Boukkouri, H. (2018) *Arithmetic Properties of Word Embeddings.* Available at: https://medium.com/data-from-the-trenches/arithmetic-properties-of-word-embeddings-e918e3fda2ac (Accessed: 15 March 2020)

Go, A., Bhayani, R., and Huang, L. (2009) *Twitter Sentiment Classification using Distant Supervision.* California, USA: Stanford University.

Gomez, R. (2018) *Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names.* Available at: https://gombru.github.io/2018/05/23/cross_entropy_loss/ (Accessed at: 5 April 2020)

Goodfellow, I., Bengio, Y., and Courville, A. (2016) *Deep Learning.* Massachusetts, USA: MIT Press.

Han, J., Kamber, M., and Pei, J. (2012) *Data Mining: Concepts and Techniques.* Massachusetts, USA: Morgan Kaufmann.

Kang, N. (2017) *Multi-Layer Neural Networks with Sigmoid Function— Deep Learning for Rookies.* Available at: https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f (Accessed: 2 April 2020)

Karani, D. (2018) *Introduction to Word Embedding and Word2Vec.* Available at: https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa (Accessed: 24 March 2020)

Kingma, D.P., and Ba, J.L. (2015) *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION.* Toronto, Canada: University of Toronto.

Kostadinov, S. (2019) *Understanding Backpropagation Algorithm.* Available at: https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd (Accessed: 23 March 2020)

Lin, Y. (2019) *10 Twitter Statistics Every Marketer Should Know in 2020.* Available at: https://www.oberlo.co.uk/blog/twitter-statistics (Accessed: 7 March 2020)

Maklin, C. (2019) *Linear Discriminant Analysis in Python.* Available at: https://towardsdatascience.com/linear-discriminant-analysis-in-python-76b8b17817c2 (Accessed: 21 March 2020)

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013) *Distributed Representations of Words and Phrases and their Compositionality.* California, USA: Google Research.

Mikolov, T., and Le, Q. (2014) *Distributed Representations of Sentences and Documents.* California, USA: Google Research.

Mitchell, T, M. (1997) *Machine Learning.* New York, USA: McGraw-Hill.

NSS. (2017) *An Intuitive Understanding of Word Embeddings.* Available at: https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/ (Accessed: 14 March 2020)

Pang, B., and Lee, L. (2008) *Opinion mining and sentiment analysis.* New York, USA: Cornell University.

Prabhakaran, S. (2018) *Cosine Similarity.* Available at: https://www.machinelearningplus.com/nlp/cosine-similarity/ (Accessed: 2 April 2020)

Raschka, S. (no date) *Machine Learning FAQ.* Available at: https://sebastianraschka.com/faq/docs/difference_classifier_model.html (Accessed: 17 March 2020)

Raschka, S. (2014) *Linear Discriminant Analysis.* Available at: https://sebastianraschka.com/Articles/2014_python_lda.html (Accessed: 17 March 2020)

Rocca, J. (2018) *A gentle journey from linear regression to neural networks.* Available at: https://towardsdatascience.com/a-gentle-journey-from-linear-regression-to-neural-networks-68881590760e\ (Accessed: 23 March 2020)

Rong, X. (2016) *word2vec Parameter Learning Explained.* Michigan, USA: University of Michigan.

Swaminathan, S. (2018) *Logistic Regression - Detailed Overview.* Available at: https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc (Accessed: 20 March 2020)

Zornoza, J. (2020) *Logistic Regression Explained.* Available at: https://towardsdatascience.com/logistic-regression-explained-9ee73cede081 (Accessed: 20 March 2020)

Appendix A – **Machine Specifications**

Primary Machine Specs:

CPU	AMD Ryzen 5 2600
GPU	AMD RX 580 8GB
RAM	16GB
OS	Windows 10 Home

Secondary Machine Specs:

CPU	Intel Core i5 7360U
GPU	Intel Iris Plus 640
RAM	8GB
OS	macOS High Sierra

I also used Google Colaboratory which allowed me to utilize some Google hardware to train my models. The specs for this machine varied but most of the time it was:

CPU	Intel-based Xeon
GPU	Nvidia Tesla K80
RAM	13GB
OS	Linux

Appendix B – **How to Run**

You should have ALL of the files in the same directory/folder. This includes all models, datasets, and the main Project.ipynb file. See Appendix C for more.

You should then run and launch Jupyter Notebook on your computer. If you do not have this then you should install this. You can install this using the following instructions: https://jupyter.org/install

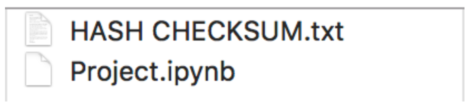
Once you have installed and have opened Jupyter Notebook, you should navigate to and open the Project.ipynb file.

You should then run all of the cells one by one, in order, and review the outputs.

PLEASE NOTE: Some of the outputs will not match exactly, and I don't really know what the reason for this is. This particularly applies to my MLP experiments, where I'm getting slightly different outputs on my MacBook and on my Desktop PC. You can find the specifications of the PC I used to work on the project in Appendix A.

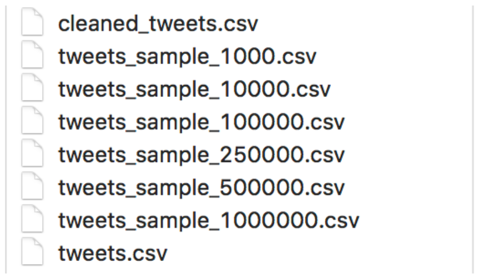
Appendix C – **File Structure**

In the PROJECT.zip file you should find:



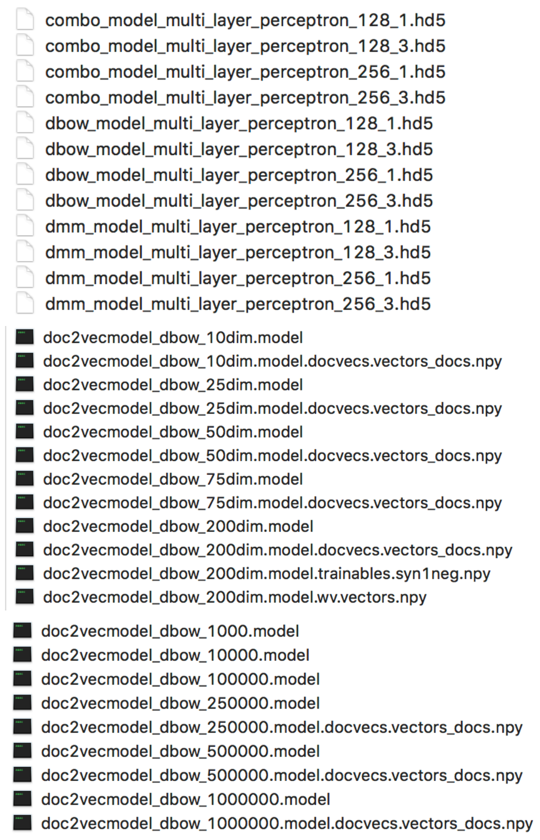
HASH CHECKSUM.txt
Project.ipynb

In the TWEETS.zip file you should find:

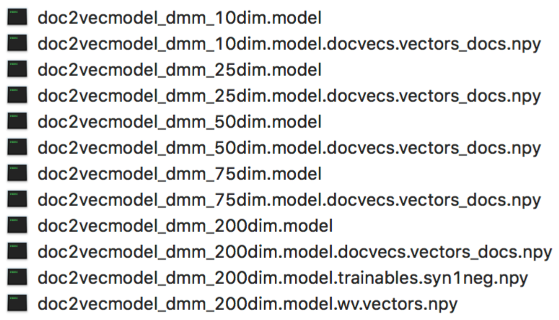


cleaned_tweets.csv
tweets_sample_1000.csv
tweets_sample_10000.csv
tweets_sample_100000.csv
tweets_sample_250000.csv
tweets_sample_500000.csv
tweets_sample_1000000.csv
tweets.csv

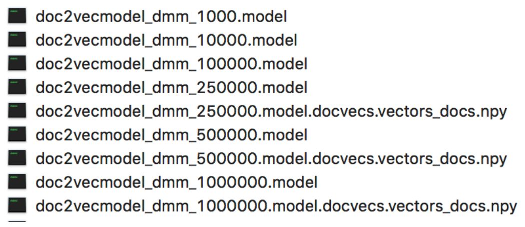
In the MODELS.zip file you should find:



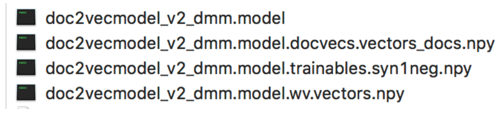
combo_model_multi_layer_perceptron_128_1.hd5
combo_model_multi_layer_perceptron_128_3.hd5
combo_model_multi_layer_perceptron_256_1.hd5
combo_model_multi_layer_perceptron_256_3.hd5
dbow_model_multi_layer_perceptron_128_1.hd5
dbow_model_multi_layer_perceptron_128_3.hd5
dbow_model_multi_layer_perceptron_256_1.hd5
dbow_model_multi_layer_perceptron_256_3.hd5
dmm_model_multi_layer_perceptron_128_1.hd5
dmm_model_multi_layer_perceptron_128_3.hd5
dmm_model_multi_layer_perceptron_256_1.hd5
dmm_model_multi_layer_perceptron_256_3.hd5
doc2vecmodel_dbow_10dim.model
doc2vecmodel_dbow_10dim.model.docvecs.vectors_docs.npy
doc2vecmodel_dbow_25dim.model
doc2vecmodel_dbow_25dim.model.docvecs.vectors_docs.npy
doc2vecmodel_dbow_50dim.model
doc2vecmodel_dbow_50dim.model.docvecs.vectors_docs.npy
doc2vecmodel_dbow_75dim.model
doc2vecmodel_dbow_75dim.model.docvecs.vectors_docs.npy
doc2vecmodel_dbow_200dim.model
doc2vecmodel_dbow_200dim.model.docvecs.vectors_docs.npy
doc2vecmodel_dbow_200dim.model.trainables.syn1neg.npy
doc2vecmodel_dbow_200dim.model.wv.vectors.npy
doc2vecmodel_dbow_1000.model
doc2vecmodel_dbow_10000.model
doc2vecmodel_dbow_100000.model
doc2vecmodel_dbow_250000.model
doc2vecmodel_dbow_250000.model.docvecs.vectors_docs.npy
doc2vecmodel_dbow_500000.model
doc2vecmodel_dbow_500000.model.docvecs.vectors_docs.npy
doc2vecmodel_dbow_1000000.model
doc2vecmodel_dbow_1000000.model.docvecs.vectors_docs.npy



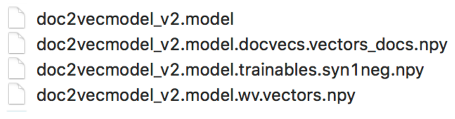
```
doc2vecmodel_dmm_10dim.model
doc2vecmodel_dmm_10dim.model.docvecs.vectors_docs.npy
doc2vecmodel_dmm_25dim.model
doc2vecmodel_dmm_25dim.model.docvecs.vectors_docs.npy
doc2vecmodel_dmm_50dim.model
doc2vecmodel_dmm_50dim.model.docvecs.vectors_docs.npy
doc2vecmodel_dmm_75dim.model
doc2vecmodel_dmm_75dim.model.docvecs.vectors_docs.npy
doc2vecmodel_dmm_200dim.model
doc2vecmodel_dmm_200dim.model.docvecs.vectors_docs.npy
doc2vecmodel_dmm_200dim.model.trainables.syn1neg.npy
doc2vecmodel_dmm_200dim.model.wv.vectors.npy
```



```
doc2vecmodel_dmm_1000.model
doc2vecmodel_dmm_10000.model
doc2vecmodel_dmm_100000.model
doc2vecmodel_dmm_250000.model
doc2vecmodel_dmm_250000.model.docvecs.vectors_docs.npy
doc2vecmodel_dmm_500000.model
doc2vecmodel_dmm_500000.model.docvecs.vectors_docs.npy
doc2vecmodel_dmm_1000000.model
doc2vecmodel_dmm_1000000.model.docvecs.vectors_docs.npy
```



```
doc2vecmodel_v2_dmm.model
doc2vecmodel_v2_dmm.model.docvecs.vectors_docs.npy
doc2vecmodel_v2_dmm.model.trainables.syn1neg.npy
doc2vecmodel_v2_dmm.model.wv.vectors.npy
```



```
doc2vecmodel_v2.model
doc2vecmodel_v2.model.docvecs.vectors_docs.npy
doc2vecmodel_v2.model.trainables.syn1neg.npy
doc2vecmodel_v2.model.wv.vectors.npy
```

You should take all these files and put them into one folder.