Analysing the Time Complexity and Performance of Text-to-Image Sentiment Analysis

Student: Sean Betts

Student Number: 08865434

Supervisor:

Wageeh Boles, Mahsa Baktashmotlagh, Samuel Cunningham-Nelson

1 Executive Summary

This document discusses the project that has focused on determining the effectiveness of sentiment analysis on text data that has been converted to an image. It will present a review of the literature review, as well as any changes that have been made to the design plan. A completed project timeline is also provided.

There has been no research performed on sentiment analysis that converts data into different mediums. As such throughout this project a model will be derived to convert from the text medium to an image medium and determine if there is any effect on performance or whether there is any more information that could be acquired from this process. As such, this project could offer a technique that improves upon the efficiency of sentiment analysis as well as providing a method of extracting more information about a set of text while it is in an image medium.

Overall the results of the classifiers did not show the ability to accurately predict the sentiment of text that had been converted into an image. None of the image-based classifiers were able to achieve a higher or equal accuracy to that of the text-based classifiers. This may be due to the generated images being too similar in content. This in turn led to the generation of binary codes that were the same as text samples of opposite sentiment, which would have greatly affected the accuracy of the classifiers. As such, it is believed that these generated images must have unique features, before the accuracy of the classifiers can improve.

2 Background and Literature Review

2.1 Introductory Statement

This project has focussed upon the efficiency of sentiment analysis using text that has been converted to an image. The project has been focused on determining the efficiency of a sentiment classifier that was fed different mediums. These different mediums included the text data and images, in a hash representation and in an image medium.

Sentiment analysis is used to determine the attitude of a topic. This analysis can be performed on either image or text mediums, or the combination of the two. No research has been performed on text that has been converted to an image medium. As such, this project has focused upon the development of a method to perform sentiment analysis on text that has been converted to an image. This has focused on performing classification on images and binary codes that have been generated from the images.

https://github.com/sbett7/Text_to_Image_Classifier

2.2 Literature Review

2.2.1 Neural Network Sentiment Analysis

An analysis of three different Convolutional Neural Network (CNN) models that integrate sentiment information with semantic word representation was presented in [1]. The three models that were presented included the Mixed-Sentiment Word Vector, the Combined Sentiment Word hesemantic level and semantic level within the word vector when performing sentiment analysis. From the results of the study it was found that injecting the sentiment information into the different layers of the models improved the ability to identify the sentiment polarity of a word vector in comparison to that of a generalised CNN architecture [1].

A study was performed by [2] on visual sentiment analysis on the images that were used in Chinese social media. This was done by using a sentiment analysis framework consisting of a Deep Convolutional Neural Network (DNN). The model consists of 22 layers that applied transfer learning as well as weights and biases from a pretrained third-party sentiment analyser framework known as GoogLeNet. The results of this model indicated that the precision of the proposed model was greater than the CNN and a PCNN model that was used to benchmark their model. [2] performed a comparison against another classifier known as the AlexNet classifier. It was determined that the performance of their model was greater than AlexNet's model by 9%. This model did contain several misclassified images which was suggested to be due to a lack of training material [2].

Sentiment analysis was performed by [3] with CNN's on Chinese microblogs that contained both textual and image content. A CNN was used to analyse sentiment within text, a DNN was used to perform analysis on images, and a combination of text and images using by combining the two neural networks with a logistic regression. It was tested using two-class and three-class evaluations. It was determined that a combination of the text and image classifiers provided greater prediction rates than the use of the individual media types. It also indicated that the use of a two-class evaluation classifier had higher precision than the three-class evaluation classifier by 6%. Similar results were presented in [3]. However, [3] also displayed that the precision of the text analysis was greater than the analysis of the image medium.

This research presented methods of performing sentiment analysis on different mediums. It displayed that the precision of using a text medium was greater than using an image medium in one study. However, it did not consider converting data between mediums and how it might affect performance or precision. No material covered the performance of the using an image sentiment classifier and a text sentiment classifier. Addressing these gaps will determine if there is any value in converting data between mediums and will also provide evidence as to which medium can be processed faster.

There have also been other sentiment analysis methods that have used hashing functions to provide the classifier's input. One method uses hash embeddings to manage large vocabularies with and without a dictionary. Another method used a hashing technique called BitHash that was tested in two different applications; sentiment analysis and image retrieval.

BitHash, short for Bitwise Min-Hash, is a hashing technique that is based upon the Min-Hash function. It generates more compact hash codes than other hashing functions. This technique was used in [4] to demonstrate its effectiveness in both image retrieval and sentiment analysis. The process that was used for the sentiment classification is shown in Figure 1. It can be seen that the BitHash function is used as a pre-processing method that converts the text data into a set of binary codes that is fed to the classifier. It was found that as the number of bits within the binary representation increased, the accuracy of the classifier also increased.

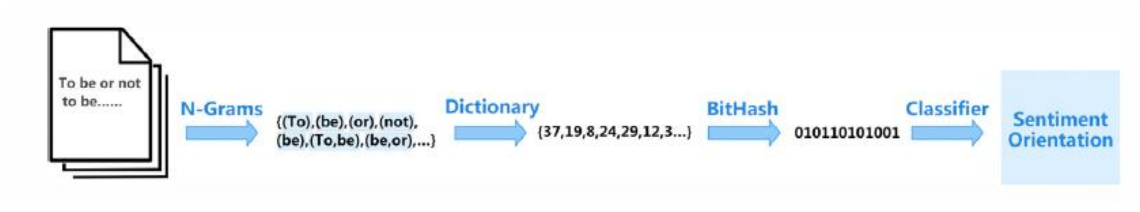


Figure 1: The general framework of sentiment classification with BitHash Technique [4]

Hash embeddings is the process of representing words in a continuous vector form. It was found within [5], that using a random hash function upon word embeddings allowed the classifier to process huge vocabularies. The process involves performing *k* number of hash functions on an input token and combining the chosen vectors. It was found that the function was capable of producing state-of-the-art accuracy when used in sentiment analysis.

Linear SVM's were used when performing sentiment analysis on an IMDb movie review dataset, as described by [6]. Three different classifiers were used, these included the Linear SVM, a Naïve Bayes classifier, and a Synthetic words classifier. The dataset was tokenised and placed through a Porter algorithm to remove suffixes from words. From their results they were able to determine that the Linear SVM resulted in the highest accuracy.

2.2.2 Hashing Techniques

A comparison of different hashing techniques such as Locality Sensitive Hashing (LSH), and Spectral Hashing (SH) by [7]. The aim of this paper was to evaluate spectral hashing and determine the limitations of the technique on larger sets of data. The study found that the choice of the outputted binary code length would impact the search quality and the performance of the technique. It was shown that as the dataset size increased, so too did the computational time as the binary code length grew. This was further covered by [8], and [9]. It is also stated that a drawback of the approach is the memory space that is required to compute the Hamming matrix for large datasets [7].

Other related work has considered how to optimise the spectral hashing algorithm. [10] developed a hashing technique that included a process that would optimise the Laplacian graph with the use of user-provided tags instead of generating a distance metric that would be used to perform the initial construction of the graph. This model was compared against other models such as SH and LSH and was found to be more accurate in finding images related to the inputted text. [10]

Another model, that was developed by [11], that is the reverse of the spectral hashing model. This model considered how to improve upon the defective mapping issue that maps dissimilar points to adjacent hash codes. The main difference between the ReSH and the SH models is that the position of the input data point $x'_i s$ has been switched with the outputted hash code $y'_i s$ within the model. The evaluations of the reversed spectral hashing found it outperformed all other spectral hashing models that were tested, such as LSH and SpH [11]. It was found that the effectiveness of this new model was between 132% and 33% better than the baseline technique SH.

[12] considered the balance between the computational cost of the spectral hashing model and its retrieval performance while achieving the criterion for entropy maximisation. Their proposed

model was based upon the SpH model; however, its base assumption considers that any distribution of one-dimensional data can be mapped with a uniform distribution. It was found that the SFSpH technique had the least computational cost as it reduced costs by 25% across tested techniques [12].

Deep hashing is another hashing method that makes use of a deep neural network that learns multiple hierarchical nonlinear transformations that assists in providing compact *exactly* binary codes for large-scale image retrieval.

The deep hashing method presented in [13], HashNet, introduces a new architecture that addresses the data imbalance and gradient issues the optimisation problem. The HashNet algorithm is a deep learning to hash through continuation model. The model takes a pair of images as input, and passes it through a CNN, a fully-connected hash layer, a sign function to binarize the results, and a weighted cross-entropy loss function for similarity preservation. It was shown that this method was able to produce *exactly* binary codes from a set of image data. The report found that using the continuation method, and optimising the entropy-loss function, the HashNet model could yield state-of-the-art media retrieval.

Extending upon the deep hashing technique discussed in the previous section, [14] describes an end-to-end process that is used to reduce searching in large sized data and maintain state-of-the-art accuracy. This is done by using quantizable representations which are used to optimise and improve the quality of the network embedding representation for the specified binary code representation. They further move to prove that the solutions can be determined by solving an equivalent minimum cost flow problem. Using this model, they were able to maintain search accuracy in comparison to other linear search algorithms.

2.3 Research Problem

The research problem that has been focused upon within this project has considered the possibility of performing sentiment analysis on text that has been converted to an image. It also considers how effective it is in comparison to the analysis that can be performed on normal text and image data. The features of this analysis will also move to determine if applying hashing functions to these images will result in any changes to the accuracy of such a classifier.

There is significance in researching this topic as this method has not been publicly discussed. As such it would provide a clear indication of how viable this option is when performing sentiment analysis. Current text classification methods use word tokenisation and other steps to strip away unnecessary data from a set of text. With this method it may be possible to take the entire text string and build an image with it so that there is no loss in data. This research may also provide further research opportunities to investigate how such an analysis method could be used in other fields.

Sentiment analysis is used to process data and determine the opinion of that text. This text data can focus on reviews of a product or based upon a specific topic. Many different approaches in performing sentiment analysis have been investigated. However, this research topic may be important to such occupations as it may be more efficient or provide greater information about a set of text.

As such there are several questions that have been focused upon within this project.

1. Are there any useful features within a binary image constructed from a string of text?

2. Is it possible to perform accurate sentiment analysis on text that has been converted to an image?
3. How efficient and accurate is this method in comparison to models that make use of the raw data?

3 Program and Design of the Research Investigation

3.1 Objectives, Methodology and Research Plan

There are several tasks that were performed to address the questions that were presented in Section 2.3. These tasks included the development of a program that will convert a text dataset to an image-based dataset, inputting text and image data to a hashing function, and using binary codes with sentiment classifiers. The initial tasks that have been completed within these projects include the development of a script to convert text to images, the testing of hashing techniques, and testing the functionality of sentiment classifiers that make use of binary codes. Changes have been made to the project, focusing on the generation of new image types such as colour and grayscale images.

The project has also focused on the testing of these images with other sentiment classifiers that take different mediums. These mediums will include images, binary codes generated from hashed text data, and binary codes generated from hashed image data. The accuracy and overall efficiency during the training and testing modes of each model have been compared to determine if there are any advantages to using the developed model.

The datasets that were used consisted of a Twitter dataset that was taken from [15], and an IMDB movie review dataset that was taken from [16]. The two different datasets were used to determine the effectiveness of the technique on two distinct types of data. The Twitter dataset contains short text samples that do not very greatly in length and contain similar phrases. The IMDB dataset contain samples of various lengths and consist of distinct contents.

As such it was expected that there would be a decrease in the accuracy of the classification with regard to the twitter dataset as each a single tweet was originally limited to 140 characters. However, this limit has increased to 280 characters as of early 2018 [17]. The IMDB movie review dataset is limited to have a minimum of 50 characters per review and have no maximum character limit [18]. As such, the IMDB movie review dataset is likely to contain a higher degree of unique data.

3.1.1 Environment and Testing Data

The environment that was used to perform the testing of this project consisted of a machine with a Window 10 Operating System, 48GB of RAM, an Intel i7-6700K CPU @ 4.0GHz, and a NVIDIA GeForce GTX graphics card. A Xubuntu 16.04 Virtual Machine (VM), allocated with 8GB RAM, and 2 CPU cores, was used to perform several of the binary code generation techniques.

3.1.2 Pre-Processing Steps

3.1.2.1 Image Creation

Before passing the data to the classifier, the text data need to be converted into image data and placed through a hashing technique to retrieve a set of binary codes. The process can be seen in Figure 2.

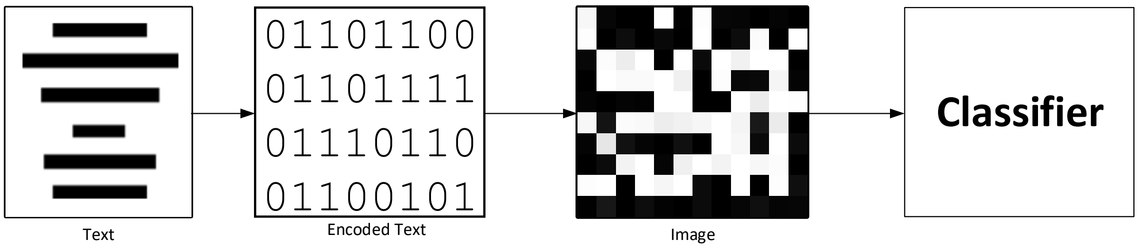


Figure 2: The text is first encoded to a binary or hexadecimal string. This encoded string is then used to fill an image with a specific shape. After being generated. The images can be passed to a hashing function or a Sentiment classifier.

The text data is converted to an encoded string and each individual bit is used to create an image of user specified size. Several different images types were created to test the effectiveness of the classifying technique. These types included binary images, grayscale images, and colour images.

The binary images are created by taking the binary string representation of the given text and each bit within the string to comprise one pixel within the image. Two different options were used to fill the images that were generated. The first was designed to assign the length of the binary string to the image. The other method involved iterating through the binary string until the contents of the image was full. This was done to determine if an image with more encoded data would be more accurate than image with less data.

A set of RGB images were also generated and tested by converting the input text into hexadecimal notation. Each hexadecimal value in the string would contribute to an RGB pixel of the generated image. As with the other techniques described previously, the colour images were also generated that had been filled with text data, as well as images that were only filled with the length of the input text's hexadecimal string. A general overview as to how the image is generated can be seen in Figure 3.

```
Create Filled Image
  # initialise variables
   text_string = "test_string";
   image_dimensions = [64, 64];
   number_of_elements = image_dimensions[0] * image_dimensions[1];
   image_array = new array[dimensions[0], dimensions[1]];
   encoded_string = format(text_string, 'ENCODE_TYPE');

   # initialise counters
   current_row = 0;
   current_column = 0;
   counter = 0;

   While counter > number_of_elements do
        For bit in binary_string
               If current_column < image_dimensions[1]
                       current_row++;
                       current_column = 0;
               If current_row >= image_dimensions[1]
                       Break;  # leave loop at the end of image
               image_array[current_row, current_column] = 255 * bit;
               counter++;
   return image_array;
```

Figure 3: The image is filled with a type of encoded string and filled with each value of the encoded string being given to a single pixel. The types of encoding that are allowed includes Binary and Hexadecimal.

It is believed that there will be some variability in the classification accuracy across the different image types. It is believed that RGB images will have higher accuracy than grayscale and binary images due to the extra dimension and new values that can be used to represent the text data. Binary and grayscale images can only represent data within a certain range. RGB images provide an additional dimension that increases the range of ways that a set of data that can be represented.

It is also believed that there will be a difference between the accuracy of images that have been filled with text data and images has only been filled with the text string. By creating images that only filled with a single text string, the length of the text would become an important feature in the classification of the text data. The images that are generated using the text string until they are filled remove the length as an important feature. It also adds an unspecified set of redundant data into the image which could affect the accuracy of the classifier.

Four different image sizes were generated from the IMDB dataset to test the accuracy of this method. These sizes included; 48x64, 64x64, 128x64, and 128x128 images. These images sizes were chosen as in most cases they were capable of storing all of the text data within the images. Larger images required more resources than were available within the testing environment. As such, the generation was limited to images with a maximum size of 128x128.

A set of images were also created for the Twitter dataset. Three different image sizes were created, these sizes included; 28x80, 40x28, and 35x32 images. These image sizes were used as the 40x28 and 35x32 images would be able to store the binary string of a 140-character tweet. The 28x80 images were used as it would be able to store a full binary string of a 280-character tweet.

3.1.2.2 Binary Codes

Binary codes were developed using the implementations discussed by [8], [19], and [20]. [20]'s implementation was used to convert the raw dataset into a binary code. The other two implementations were used to generate the binary codes from the generated images.

For the image-based hashing techniques, an indexer was required to generate the binary codes from the input images. The indexers could be used to generate binary codes of various lengths. The lengths of the binary codes that were tested include 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit. The index was built by generating images from the dataset and passing the equivalent image vectors to the index builder. With the index built, the indexer can be used to hash a generated image.

3.1.2.3 Text-Based Binary Codes

Text-based binary codes will be used as a baseline for determining the effectiveness of the image-based classifiers. Generating binary codes from continuous text was done by using the DataSketch package. This is done by passing a string of text to a MinHash algorithm. This algorithm iterates across every character in a string to generate the binary code. The size of the binary codes that were generated included 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit binary codes. The code block displayed in Figure 4 displays the method that was used to generate the binary codes.

```
Generate Text-Based Binary Codes
   text_string = "test_string";
   hash_size = 128;
   minhash = MinHash(num_perm=hash_size); # set the expected

  For character in text_string do
      minhash.update(character); # update hash for every character
  return minhash;
```

Figure 4: Every character in a text string is iterated across and given to the MinHash function to update the current hash value.

3.1.3 Sentiment Classifier

Several different types of classifiers were tested to determine if there were any changes in accuracy across the different data types. These classifiers included a Deep Hashing classifier, a Hash Embedding classifier, a KNN classifier, and a Linear and Gaussian SVM. These classifiers take binary codes that are generated from the LSH and SH hashing techniques be fed into the sentiment classifiers as the feature vector. Previous work to determine the effectiveness of the SH and LSH hashing functions can be seen in Section 9.1.

3.1.3.1 Hash Embedding Classifier

The hash embedding classifier that was discussed by [5], was used with the binary codes that were generated from the SH and LSH techniques. [5] makes use of the encodings of individual words and not those generated from images. These encodings are also done using a SHA-1 algorithm, which makes the hash of similar values to be significantly different, which is different from the LSH and SH hashing functions operate.

This classifier was used to test its accuracy with the raw Twitter dataset, as well as the LSH binary codes generated from the raw text, and the LSH and SH binary codes that were generated from the images created from the text. The raw text and the LSH binary codes generated from the text were used as a baseline for the accuracy of the binary codes generated from the images. No altercations were made to the hash embedding classifier when testing the raw text classifying accuracy.

Several changes were made to their implementation to use the binary codes that were generated by the LSH and SH hashing techniques. These binary codes were used to replace the SHA-1 word encodings that were previously used.

3.1.3.2 KNN

The K-Nearest Neighbour (KNN) algorithm was implemented using [21] as a base. The classifier used the binary codes as the input feature vector. The labels that were used comprised of '0', and '1' to indicate negative and positive labels respectively. The classifier will be used to determine the accuracy of the text-based binary codes, and the image binary codes.

Several different configurations of the KNN classifier were used to determine changes in the accuracy of the classifier. These included the metrics to determine the distance between neighbours and the difference between using Euclidean and Hamming distance as a metric for the classifier. The different neighbour metrics that were used consisted of 1, 4, 20, 50, 100, and 250 neighbours. This was done to determine if there was an effective number of neighbours that will affect the accuracy of the classifier.

3.1.3.3 SVM

Two different SVM classifiers were used to determine the effectiveness of the SVM model with the data. These classifiers include the Gaussian SVM and the Linear SVM. The sentiment classifiers were used to determine of the accuracy of the image-based binary codes. Both SVM models use the binary codes as the feature vector for the model. the labels given to the models are the numeric values '1' for positive, and '0' for negative sentiment.

As with the other sentiment classifiers, the SVM models take the binary codes as the feature vector. As with the KNN classifier, the labels given to the Linear SVM are the numeric values '1' for positive, and '0' for negative sentiment. Both models were configured to determine if the C-Value was changed for both to determine if the value would affect the fitting of the data. The values for the C-Value was varied from 0.001 to 1500.

3.1.3.4 Deep Hashing

The Deep Hashing implementation that was provided by [13] was used to test the generated images. Of the five available Deep Hashing algorithms that were available, the Deep Cauchy Hashing (DCH) algorithm was used. This method was used due to the speed which it took to train and validate the method in comparison to other models. This model was able to train and validate models within 60 minutes, in comparison to other models which took a higher amount of time.

No changes were made to the implementation of the Deep Hashing algorithm. The algorithm was set to use a learning rate of 0.005, and a decay factor of 0.1. All image types and shapes that were discussed in Section 3.1.2.1 were tested. The algorithm outputs the mAP, precision and recursion of the model. An example of the images that were used can be seen in Figure 5.

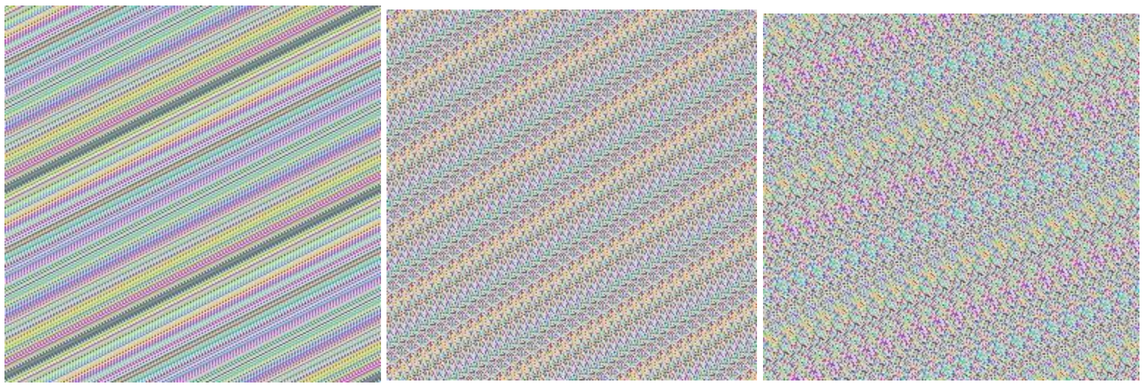


Figure 5: Colour Images that were generated from the IMDb dataset.

4 Results

The results of the accuracy tests for the sentiment classifiers can be seen in Figure 6. It can be seen that the results of all of the neural network classifiers reach a maximum of 60%, overall these values averaged approximately 54%. These results were taken by feeding 50% of datasets to the classifiers to train the model and using the other 50% to test the model. This indicates that the classifiers are not capable of accurately determining the classification of a set of text that has been converted into an image.

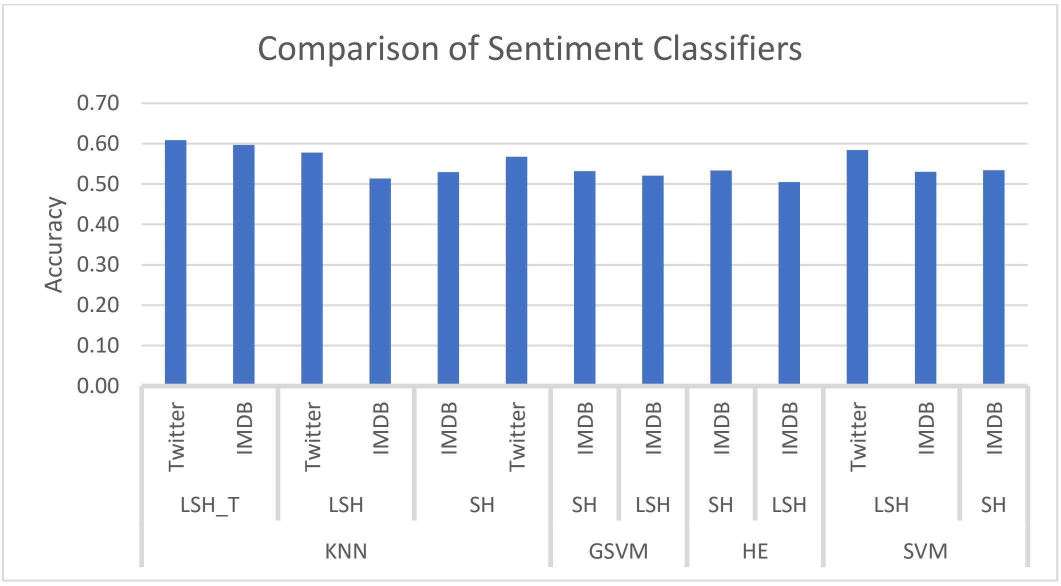


Figure 6: The accuracy of the different sentiment classifiers and the hashing results. The baseline LSH Text reaches a maximum of 60% accuracy, while the other classifiers range from 50% to 58%.

The results from the text-based binary codes between the two datasets show that the average accuracy for the classifier was 56.7% and 54.5% for the Twitter and IMDB review dataset respectively. The maximum accuracy for the Twitter dataset was the 60.8%, and the IMDB dataset's maximum accuracy was 60.1%. This indicates that the classification methods and images are below the baseline accuracy. The maximum for the IMDB dataset occurred using a KNN classifier with a binary code size of 512. The maximum for the Twitter dataset occurred using a KNN classifier with the LSH binary codes, which results in 58.4% accuracy. This data can be seen in Table 3 in the Appendix. The results from Figure 16 and Figure 17 from the Appendix also indicate that the size of the binary code and the appear to affect the accuracy of classifiers across different image types.

From the results of the Deep Hashing algorithm it can be seen that the classifiers for the IMDB review can only average an accuracy of 50%, this can be seen in Figure 18 in the Appendix. The deep hashing algorithms which take the images as input, would indicate that the images do not provide enough differences for the classifier to differentiate between the two sentiment classifications. As such, the process for generating images may not provide enough unique features to accurately classify the sentiment of new images.

The lack of change in the accuracy of the image-based classification methods between the two different datasets is believed to be due to the similarities between the generated images. The two hashing techniques were originally designed to provide similar binary codes to images of similar content. Considering the images presented in Figure 7, it can be seen that while the content differs slightly between the images, there is little difference between their binary codes. As such it could be suggested that generating images directly from the hexadecimal and binary data of a set of input text does not provide images with enough differing features to provide unique binary codes to perform accurate sentiment analysis.

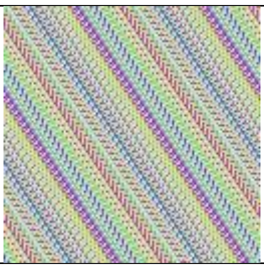
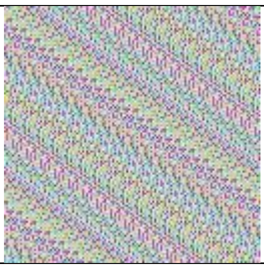
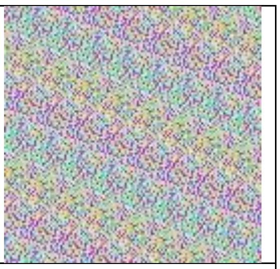
Image			
Binary Code	[233 177 152 203]	[233 177 152 83]	[233 177 152 83]
Sentiment	0	1	0

Figure 7: Images created from the IMDb dataset, with their related binary codes. It can be seen the Image 2 and Image 3 have the same binary code but are opposite in sentiment.

While the difference between the accuracies of the individual classifiers varied by 1%-2%, there were some slight trends that could be seen from the data. The other properties that were considered, such as the image shape and binary code length did not appear to make much difference in the accuracy of the classifiers. It can be seen in several different classifiers, such as the KNN, Linear SVM, and GSVM have specific image types that will result in a higher accuracy. For the GSVM and Linear SVM, RGB images that were filled resulted in the highest accuracy for SH images. For the KNN classifier, both types of colour images resulted in the highest accuracy.

The LSH binary codes provided different results for the accuracy in comparison to the SH binary codes. The GSVM classifier had greater accuracies for RGB images constructed from a single string of text. The Linear SVM classifier also had greater accuracies for RGB images that were filled with the sample text string. The KNN classifier had the greatest accuracy with any image type that had been constructed from a single string of text.

Table 1: Maximum Classifier Fitting Time

Dataset	Hash Type	Classifier	Hash Size	Image Shape	Image Type	Fit Time (s)
IMDB	LSH	GSVM	128	(128, 128)	RGB String	2043.41
IMDB	LSH	Hash Embedding	64	(64, 64)	RGB String	193.95
IMDB	SH	Hash Embedding	128	(128, 128)	RGB Full	180.82
Twitter	LSH	SVM	128	(35, 32)	Grayscale String	37.83
IMDB	SH	GSVM	128	(64, 64)	Binary String	34.79
Twitter	LSH	KNN	32	(28, 80)	Grayscale Full	9.53
IMDB	LSH	SVM	128	(128, 128)	RGB String	7.05
IMDB	SH	SVM	128	(64, 64)	Grayscale String	2.67
Twitter	SH	KNN	32	(40, 28)	RGB String	2.10
IMDB	LSH	KNN	128	(128, 128)	RGB String	0.95
IMDB	SH	KNN	8	(64, 64)	Binary Full	0.16

A sample of the time taken by the classifiers to fit and query the data can be seen in Table 1. It can be seen that overall, it takes the GSVM classifier takes the longest period of time to fit the data, the maximum time being 2043.41 seconds for LSH binary codes, and 34.79 seconds for SH binary codes. The Hash Embedding also takes longer than the LSVM and KNN classifiers, taking 180.82 seconds to fit the data using SH binary codes and 193.95 seconds for LSH binary codes.

It can also be seen that larger binary code lengths will result in higher fit times and for colour images.

The fastest to fit the data was the KNN classifier at 0.04 seconds for SH binary codes and 0.021 for SH binary codes. It can be seen in Table 2, that the fastest classifiers will usually occur for smaller binary code lengths, and binary or grayscale images. The majority of the classifiers will perform faster for smaller images as well.

Table 2: Minimum Classifier Fitting Time

Dataset	Hash Type	Classifier	Hash Size	Image Shape	Image Type	Fit Time (s)
IMDB	SH	KNN	16	(128, 128)	Binary String	0.004
IMDB	LSH	SVM	8	(48, 64)	RGB Full	0.009
Twitter	SH	KNN	128	(35, 32)	Grayscale String	0.021
Twitter	LSH	SVM	8	(35, 32)	Binary String	0.097000122
IMDB	LSH	KNN	64	(48, 64)	Binary Full	0.125
IMDB	SH	SVM	8	(64, 64)	Grayscale Full	0.43
Twitter	LSH	KNN	8	(40, 28)	Grayscale String	0.519
IMDB	SH	GSVM	8	(48, 64)	Binary Full	2.80
IMDB	LSH	GSVM	8	(128, 64)	Grayscale Full	4.140
IMDB	SH	Hash Embedding	8	(128, 128)	Binary String	5.764
IMDB	LSH	Hash Embedding	8	(128, 128)	RGB String	11.433

The difference in fitting times between the SH binary codes and the LSH binary codes are due to the difference in the number of features between the SH and LSH binary codes. The LSH binary code is returned as a binary string, where each bit constitutes a feature, whereas the SH binary code is in ASCII format and each ASCII value is taken as a feature.

5 Future Work

Further work in this area could focus on the development of a new text-to-image technique. It is believed that the failure to perform accurate classifications with these generated images is due to a lack of differing features within the generated images. Further implementations could convert the text data into a different type of string, rather than a binary string or hexadecimal string. This format could possibly take the form of a SHA hash function as it would create samples of the same size and similar text samples would result in significantly different SHA codes. However, as seen with the MinHash text binary codes, it can be seen that codes that are too different from may cause a loss in accuracy.

As stated previously, there are no significant set of features that can be seen when viewing the images. If there were specific sections of each image to display a meaningful characteristic of the set of text, the accuracy may improve. These could include setting specific areas of an image to house occurrences of certain words, such that one corner of an image would contain multiple occurrences of the same word in a set of text.

Another area that could be focused upon would be the MinHash text binary codes that were generated from continuous text. The method of generating these binary codes could be changed to change the size of a text string that is used to create and update a MinHash binary code, as passing a full string of text to the MinHash algorithm provides significantly different binary codes than updating a MinHash binary code by iterating through each character in a text string.

6 Limitations

There were several limitations that were present throughout the course of this project. These limitations focused upon the expensive computations that several of the algorithms had to perform. These algorithms include the Deep Hashing algorithms, and the GSVM classifier. Of the five Deep Hashing algorithms that were provided by [13], only one could be used as the time and resources that it required to run the algorithms were too great.

Testing of larger images could also not be done due to memory errors that were prevalent when the algorithms were performing dot products. As such, the images that could be tested were limited to 128x128 images. Further testing could not be done with the IMDB movie review set to determine whether there would be a change in accuracy for larger images.

7 Conclusion

This report has discussed the findings that were found with regarding the effects of converting text to images with respect to sentiment classification. The text was converted to binary and hexadecimal strings and used to generate different image types. The classifiers that were used in conjunction with the different image types. These classifiers included a Hash Embedding classifier, a KNN classifier, a Deep Hashing classifier, and a Linear and Gaussian SVM classifier.

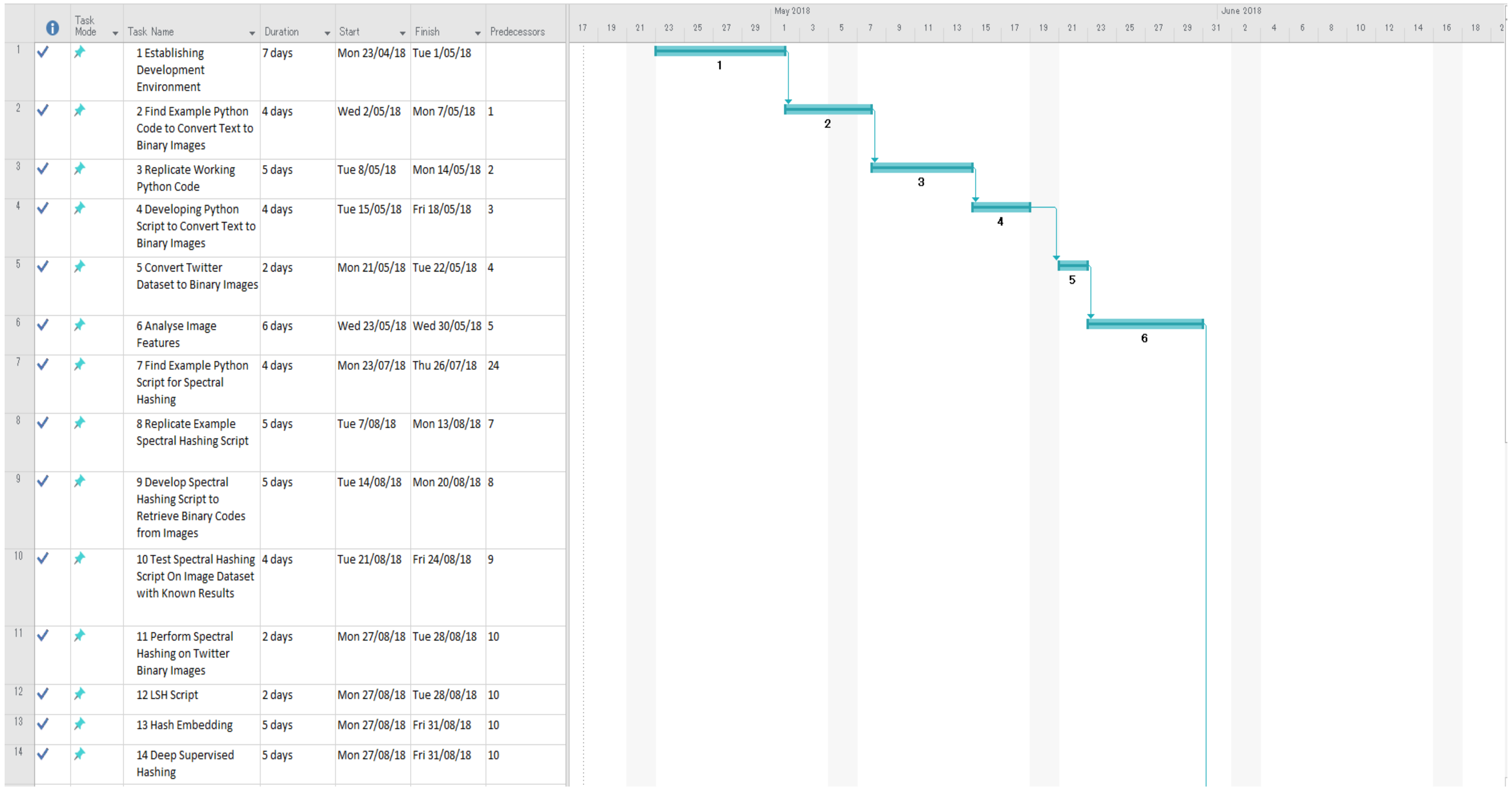
It was found that the overall accuracy of the classifiers indicate that the generated images are not unique enough to provide accurate classifications. The highest accuracy came from KNN classifier while using the Twitter dataset. The LSH and SH binary codes that were used with these classifications were not unique across the entire dataset. Many entries of different sentiment resulted in the same binary code, and as such the classifiers were unable to accurately differentiate between samples.

There are several different areas that could be worked upon to further the knowledge in this area. These areas include generating images using a different methodology, such as providing structure to the image by assigning meaning to certain areas. It is believed that by doing so will result images that are more distinguishable. Another option may also be to convert the data into some other format, such as a SHA hash value. These changes may generate images that are more unique than the ones that were generated in this project, and thus could improve the accuracy of the other classification methods.

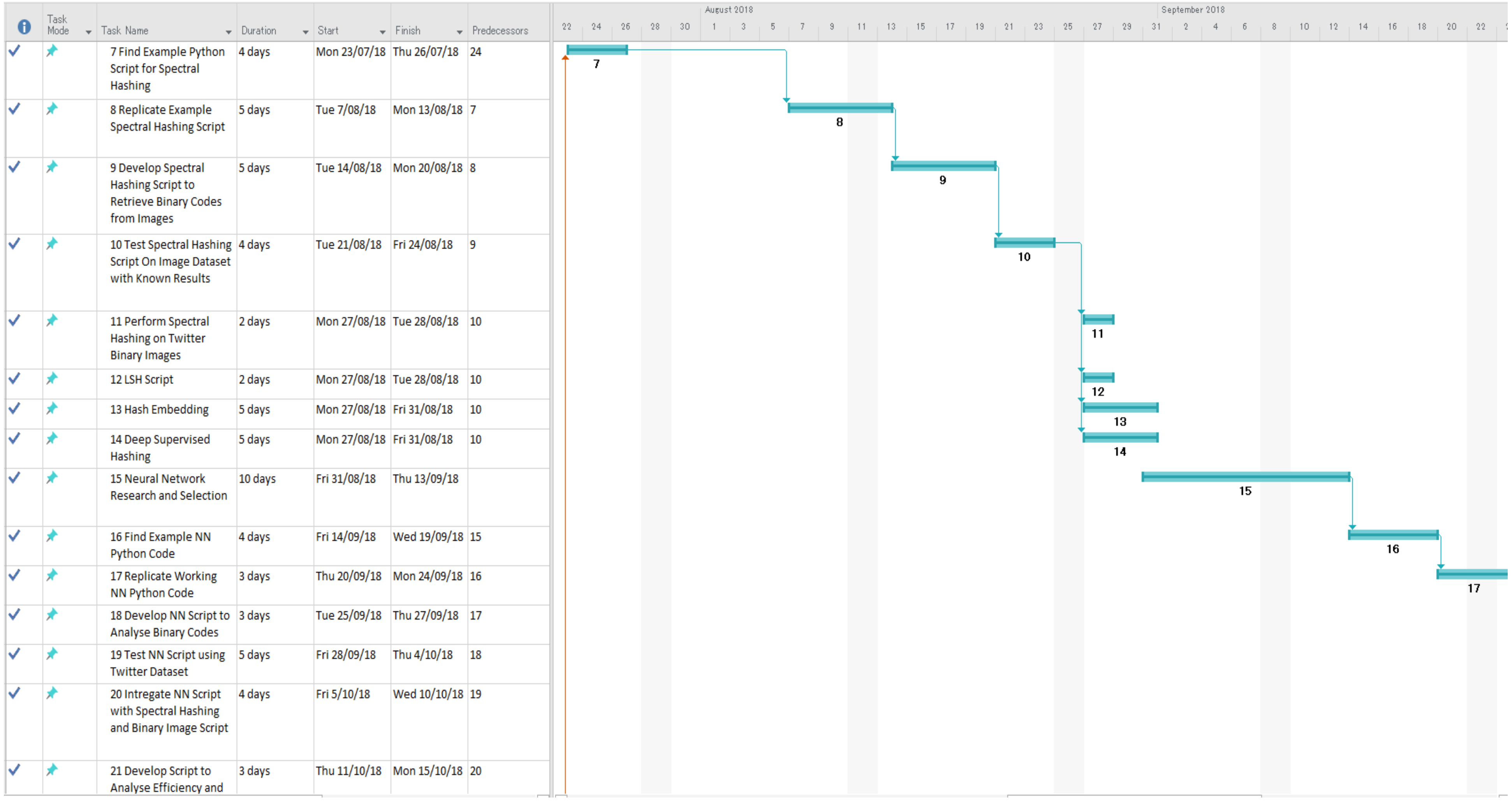
7.1 Timeline for Completion

Task Mode	Task Name	Duration	Start	Finish	Predecessors	Task Mode
Manually Scheduled	Establishing Development Environment	7 days	Mon 23/04/18	Tue 1/05/18		Manually Scheduled
Manually Scheduled	Find Example Python Code to Convert Text to Binary Images	4 days	Wed 2/05/18	Mon 7/05/18	1	Manually Scheduled
Manually Scheduled	Replicate Working Python Code	5 days	Tue 8/05/18	Mon 14/05/18	2	Manually Scheduled
Manually Scheduled	Developing Python Script to Convert Text to Binary Images	4 days	Tue 15/05/18	Fri 18/05/18	3	Manually Scheduled
Manually Scheduled	Convert Twitter Dataset to Binary Images	2 days	Mon 21/05/18	Tue 22/05/18	4	Manually Scheduled
Manually Scheduled	Analyse Image Features	6 days	Wed 23/05/18	Wed 30/05/18	5	Manually Scheduled
Manually Scheduled	Find Example Python Script for Spectral Hashing	4 days	Mon 23/07/18	Thu 26/07/18	24	Manually Scheduled
Manually Scheduled	Replicate Example Spectral Hashing Script	5 days	Tue 7/08/18	Mon 13/08/18	7	Manually Scheduled
Manually Scheduled	Develop Spectral Hashing Script to Retrieve Binary Codes from Images	5 days	Tue 14/08/18	Mon 20/08/18	8	Manually Scheduled
Manually Scheduled	Test Spectral Hashing Script On Image Dataset with Known Results	4 days	Tue 21/08/18	Fri 24/08/18	9	Manually Scheduled
Manually Scheduled	Perform Spectral Hashing on Twitter Binary Images	2 days	Mon 27/08/18	Tue 28/08/18	10	Manually Scheduled
Manually Scheduled	LSH Script	2 days	Mon 27/08/18	Tue 28/08/18	10	Manually Scheduled
Manually Scheduled	Hash Embedding	5 days	Mon 27/08/18	Fri 31/08/18	10	Manually Scheduled
Manually Scheduled	Deep Supervised Hashing	5 days	Mon 27/08/18	Fri 31/08/18	10	Manually Scheduled
Manually Scheduled	Neural Network Research and Selection	10 days	Fri 31/08/18	Thu 13/09/18		Manually Scheduled
Manually Scheduled	Find Example NN Python Code	4 days	Fri 14/09/18	Wed 19/09/18	15	Manually Scheduled
Manually Scheduled	Replicate Working NN Python Code	3 days	Thu 20/09/18	Mon 24/09/18	16	Manually Scheduled
Manually Scheduled	Develop NN Script to Analyse Binary Codes	3 days	Tue 25/09/18	Thu 27/09/18	17	Manually Scheduled
Manually Scheduled	Test NN Script using Twitter Dataset	5 days	Fri 28/09/18	Thu 4/10/18	18	Manually Scheduled
Manually Scheduled	Intregate NN Script with	4 days	Fri 5/10/18	Wed 10/10/18	19	Manually Scheduled

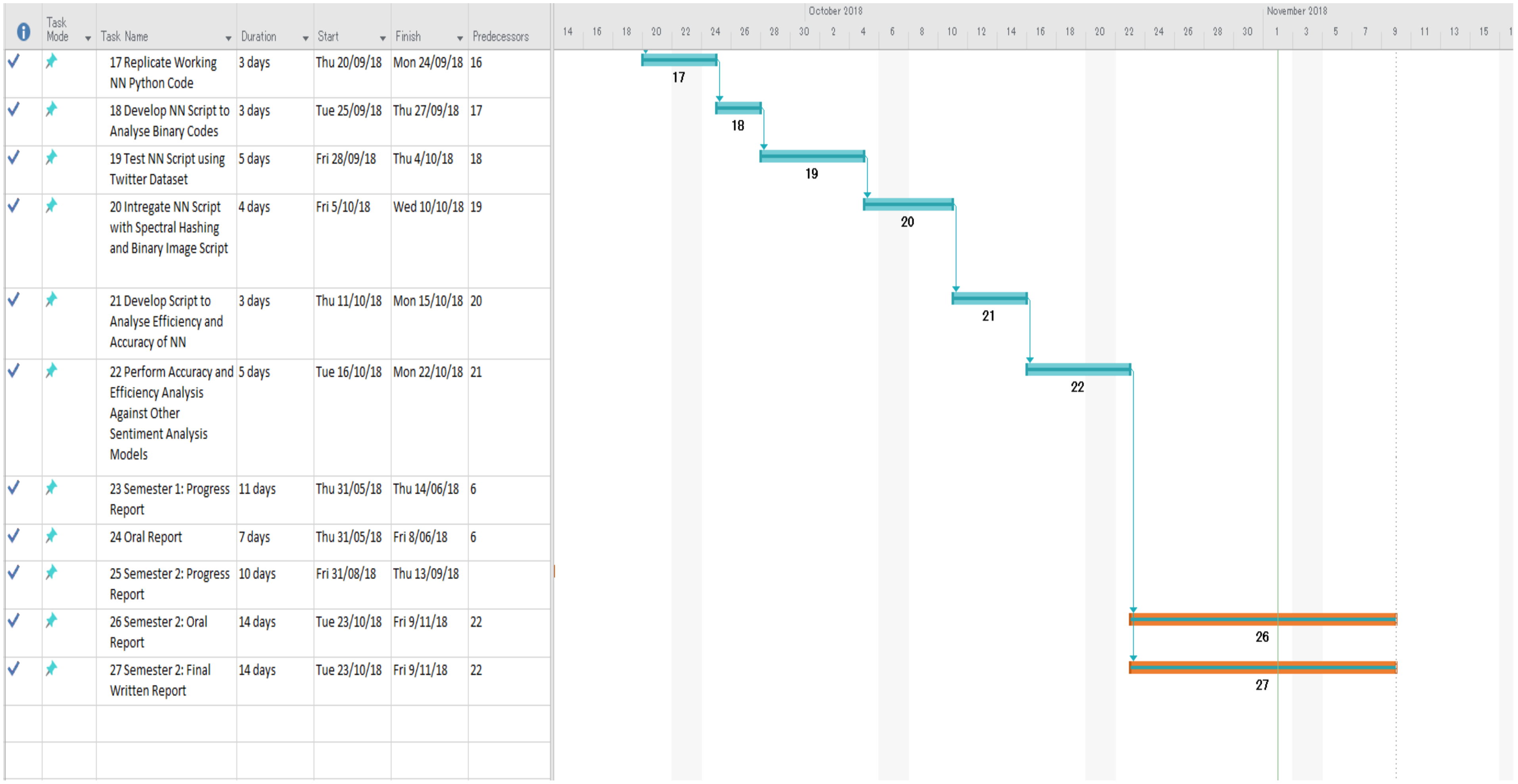
	Spectral Hashing and Binary Image Script					
Manually Scheduled	Develop Script to Analyse Efficiency and Accuracy of NN	3 days	Thu 11/10/18	Mon 15/10/18	20	Manually Scheduled
Manually Scheduled	Perform Accuracy and Efficiency Analysis Against Other Sentiment Analysis Models	5 days	Tue 16/10/18	Mon 22/10/18	21	Manually Scheduled
Manually Scheduled	Semester 1: Progress Report	11 days	Thu 31/05/18	Thu 14/06/18	6	Manually Scheduled
Manually Scheduled	Oral Report	7 days	Thu 31/05/18	Fri 8/06/18	6	Manually Scheduled
Manually Scheduled	Semester 2: Progress Report	10 days	Fri 31/08/18	Thu 13/09/18		Manually Scheduled
Manually Scheduled	Semester 2: Oral Report	14 days	Tue 23/10/18	Fri 9/11/18	22	Manually Scheduled
Manually Scheduled	Semester 2: Final Written Report	14 days	Tue 23/10/18	Fri 9/11/18	22	Manually Scheduled



	ℹ	Task Mode	Task Name	Duration	Start	Finish	Predecessors
1	✓	📌	1 Establishing Development Environment	7 days	Mon 23/04/18	Tue 1/05/18	
2	✓	📌	2 Find Example Python Code to Convert Text to Binary Images	4 days	Wed 2/05/18	Mon 7/05/18	1
3	✓	📌	3 Replicate Working Python Code	5 days	Tue 8/05/18	Mon 14/05/18	2
4	✓	📌	4 Developing Python Script to Convert Text to Binary Images	4 days	Tue 15/05/18	Fri 18/05/18	3
5	✓	📌	5 Convert Twitter Dataset to Binary Images	2 days	Mon 21/05/18	Tue 22/05/18	4
6	✓	📌	6 Analyse Image Features	6 days	Wed 23/05/18	Wed 30/05/18	5
7	✓	📌	7 Find Example Python Script for Spectral Hashing	4 days	Mon 23/07/18	Thu 26/07/18	24
8	✓	📌	8 Replicate Example Spectral Hashing Script	5 days	Tue 7/08/18	Mon 13/08/18	7
9	✓	📌	9 Develop Spectral Hashing Script to Retrieve Binary Codes from Images	5 days	Tue 14/08/18	Mon 20/08/18	8
10	✓	📌	10 Test Spectral Hashing Script On Image Dataset with Known Results	4 days	Tue 21/08/18	Fri 24/08/18	9
11	✓	📌	11 Perform Spectral Hashing on Twitter Binary Images	2 days	Mon 27/08/18	Tue 28/08/18	10
12	✓	📌	12 LSH Script	2 days	Mon 27/08/18	Tue 28/08/18	10
13	✓	📌	13 Hash Embedding	5 days	Mon 27/08/18	Fri 31/08/18	10
14	✓	📌	14 Deep Supervised Hashing	5 days	Mon 27/08/18	Fri 31/08/18	10



ℹ	Task Mode	Task Name	Duration	Start	Finish	Predecessors
✓	📌	7 Find Example Python Script for Spectral Hashing	4 days	Mon 23/07/18	Thu 26/07/18	24
✓	📌	8 Replicate Example Spectral Hashing Script	5 days	Tue 7/08/18	Mon 13/08/18	7
✓	📌	9 Develop Spectral Hashing Script to Retrieve Binary Codes from Images	5 days	Tue 14/08/18	Mon 20/08/18	8
✓	📌	10 Test Spectral Hashing Script On Image Dataset with Known Results	4 days	Tue 21/08/18	Fri 24/08/18	9
✓	📌	11 Perform Spectral Hashing on Twitter Binary Images	2 days	Mon 27/08/18	Tue 28/08/18	10
✓	📌	12 LSH Script	2 days	Mon 27/08/18	Tue 28/08/18	10
✓	📌	13 Hash Embedding	5 days	Mon 27/08/18	Fri 31/08/18	10
✓	📌	14 Deep Supervised Hashing	5 days	Mon 27/08/18	Fri 31/08/18	10
✓	📌	15 Neural Network Research and Selection	10 days	Fri 31/08/18	Thu 13/09/18	
✓	📌	16 Find Example NN Python Code	4 days	Fri 14/09/18	Wed 19/09/18	15
✓	📌	17 Replicate Working NN Python Code	3 days	Thu 20/09/18	Mon 24/09/18	16
✓	📌	18 Develop NN Script to Analyse Binary Codes	3 days	Tue 25/09/18	Thu 27/09/18	17
✓	📌	19 Test NN Script using Twitter Dataset	5 days	Fri 28/09/18	Thu 4/10/18	18
✓	📌	20 Intregate NN Script with Spectral Hashing and Binary Image Script	4 days	Fri 5/10/18	Wed 10/10/18	19
✓	📌	21 Develop Script to Analyse Efficiency and	3 days	Thu 11/10/18	Mon 15/10/18	20



	Task Mode	Task Name	Duration	Start	Finish	Predecessors
✓	📌	17 Replicate Working NN Python Code	3 days	Thu 20/09/18	Mon 24/09/18	16
✓	📌	18 Develop NN Script to Analyse Binary Codes	3 days	Tue 25/09/18	Thu 27/09/18	17
✓	📌	19 Test NN Script using Twitter Dataset	5 days	Fri 28/09/18	Thu 4/10/18	18
✓	📌	20 Intregate NN Script with Spectral Hashing and Binary Image Script	4 days	Fri 5/10/18	Wed 10/10/18	19
✓	📌	21 Develop Script to Analyse Efficiency and Accuracy of NN	3 days	Thu 11/10/18	Mon 15/10/18	20
✓	📌	22 Perform Accuracy and Efficiency Analysis Against Other Sentiment Analysis Models	5 days	Tue 16/10/18	Mon 22/10/18	21
✓	📌	23 Semester 1: Progress Report	11 days	Thu 31/05/18	Thu 14/06/18	6
✓	📌	24 Oral Report	7 days	Thu 31/05/18	Fri 8/06/18	6
✓	📌	25 Semester 2: Progress Report	10 days	Fri 31/08/18	Thu 13/09/18	
✓	📌	26 Semester 2: Oral Report	14 days	Tue 23/10/18	Fri 9/11/18	22
✓	📌	27 Semester 2: Final Written Report	14 days	Tue 23/10/18	Fri 9/11/18	22

8 References

[1] M. Lan, Z. Zhang, Y. Lu and J. Wu, “Three Convolutional Neural Network-based Models for Learning Sentiment Word Vectors towards Sentiment Analysis,” IEEE, Shanghai, 2016.

[2] J. Islam and Y. Zhang, “Visual Sentiment Analysis for Social Images Using Transfer Learning Approach,” IEEE, Atlanta, 2016.

[3] Y. Yu, H. Lin, J. Meng and Z. Zhao, “Visual and Textual Sentiment Analysis of a Microblog Using Deep Convolutional Neural Networks,” Dalian University of Technology, Dalian, 2016.

[4] W. Zhang, J. Ji, J. Zhu, J. Li, H. Xu and B. Zhang, “BitHash: An Efficient Bitwise Locality Sensitive Hashing Method With Applications,” Elsevier, Beijing, 2016.

[5] D. Svenstrup, J. M. Hansen and O. Winther, “Hash Embeddings for Efficient Word Representations,” 31st Conference on Neural Information Processing Systems, Long Beach, 2017.

[6] S. Rana and A. Singh, “Comparative Analysis of Sentiment Orientation,” International Conference on Next Generation Computing Technologies, Dehradun, 2016.

[7] L. Karbil, I. Daoudi and H. Medromi, “A Comparative Experimental Study of Spectral Hashing,” IEEE, 2017.

[8] J. Wan, S. Tang, Y. Zhang, J. Li, P. Wu and C. Hoi, “HDIdx: High-Dimensional Indexing for Efficient Approximate Nearest Neighbor Search,” Beijing, 2015.

[9] B. Revathi and G. Sudha, “Retrieval performance analysis of multibiometric database using optimized multidimensional spectral hashing based indexing,” King Saud University, Puducherry, 2018.

[10] P. Li, M. Wang, J. Cheng, C. X. Xu and H. Lu, “Spectral Hashing With Semantically Consistent Graph for Image Indexing,” IEEE, 2013.

[11] Q. Liu, G. Liu, L. Li, X.-T. Yuan, M. Wang and W. Liu, “Reversed Spectral Hashing,” IEEE, 2017.

[12] Y. Wang, S. Tang, Y. Zhang, J. Li and D. Chen, “Fitted Spectral Hashing,” Barcelona, 2013.

[13] Z. Cao, M. Long, J. Wang and P. S. Yu, “HashNet: Deep Learning to Hash by Continuation,” University of Illinois, Chicago, 2017.

[14] Y. Jeong and H. O. Song, “Efficient End-to-End Learning for Quantizable Representations,” International Conference on Machine Learning, Stockholm, 2018.

[15] Kaggle, “Twitter Sentiment Analysis 2,” Kaggle, [Online]. Available: https://www.kaggle.com/c/twitter-sentiment-analysis2/data.

[16] Kaggle, “Keras IMDb,” Keras, [Online]. Available: https://www.kaggle.com/pankrzysiu/keras-imdb.

[17] Y. Heisler, “Twitter’s 280 character limit increased engagement without increasing the average tweet length,” BGR, 8 February 2018. [Online]. Available: https://bgr.com/2018/02/08/twitter-character-limit-280-vs-140-user-engagement/.

[18] IMDb General Support , “User review guidelines,” IMDb, 2018. [Online]. Available: https://help.imdb.com/article/contribution/contribution-information/user-review-guidelines/GABTWSNLDNFLPRRH#.

[19] B. Roberts, “SparseLSH,” 2018. [Online]. Available: https://github.com/brandonrobertz/SparseLSH. [Accessed 2018].

[20] E. Zhu, “MinHash LSH,” GitHub, 2018. [Online]. Available: https://ekzhu.github.io/datasketch/lsh.html.

[21] B. Klein, “k-Nearest-Neighbor Classifier,” Python Machine Learning Tutorial, 2018. [Online]. Available: https://www.python-course.eu/k_nearest_neighbor_classifier.php.

[22] A. Kumar and R. Rani, “Sentiment Analysis Using Neural Network,” International Conference on Next Generation Computing Technologies, Dehradun, 2016.

[23] J. Wehrmann, W. Becker, H. E. L. Cagnini and R. Barros, “A Character-based Convolutional Neural Network for Language-Agnostic Twitter Sentiment Analysis,” IEEE, Porto Alegre, 2017.

[24] M. Katsurai and S. Satoh, “Image Sentiment Analysis Using Latent Correlations Among Visual, Textual, and Sentiment Views,” IEEE, Tokyo, 2016.

[25] N. Maslova and V. Potapov, “Neural Network Doc2vec in Automated Sentiment Analysis for Short Informal Texts,” Springer International Publishing, Moscow, 2017.

[26] P. D. Purnamasari, M. Taqiyuddin and A. A. P. Ratna, “Performance Comparison of Text-based Sentiment Analysis using Recurrent Neural Network and Convolutional Neural Network,” DOI, Depok, 2017.

[27] X. Sun, H. Jiajin and C. Quan, “A Multi-granularity Data Augmentation based Fusion Neural Network Model for Short Text Sentiment Analysis,” IEEE, Hefei, 2017.

[28] S. Chen, J. Yang, F. Jia and Y. Gu, “Image Sentiment Analysis Using Supervised Collective Matrix Factorization,” IEEE, Shanghai, 2017.

9 Appendix

9.1 LSH and SH Index Classification

9.1.1 Locality Sensitive Hashing Sentiment Analysis

A script using LSH has been developed to perform basic sentiment analysis on a twitter dataset. This process first constructs an index based off of a set of data that is provided. After constructing the index, text can be provided to query the index to determine the sentiment based upon the returned results of the query.

This script first collects and stores the text and sentiment data from the dataset in a model that converts the text into an image. The LSH index is then created from the image vector and sentiment of each of these image models that is used. To add these image vectors to the index however, the vectors must all be the same size. As such, before constructing the index, each image vector is resized to match the length of the longest image vector. This can be seen in Figure 8. These image vectors are then used along with the sentiment of the text to build the index.

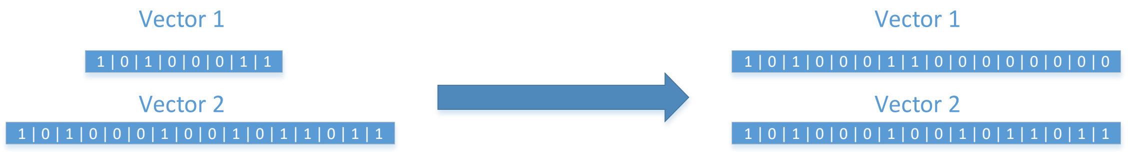


Figure 8: Image Vector Reshaping

The current sentiment classifier for this method uses the average sentiment from the returned matches, which is limited to five results. The average of these results is then compared against a threshold to determine if the queried text has positive or negative sentiment. The current threshold is 0.5, anything above this threshold will be considered positive, and anything below this threshold will be considered to have negative sentiment.

Testing has been done to determine whether there is any effect on the size of the index, and the height of the images. The results of these test can be seen in Figure 9 and Figure 10.

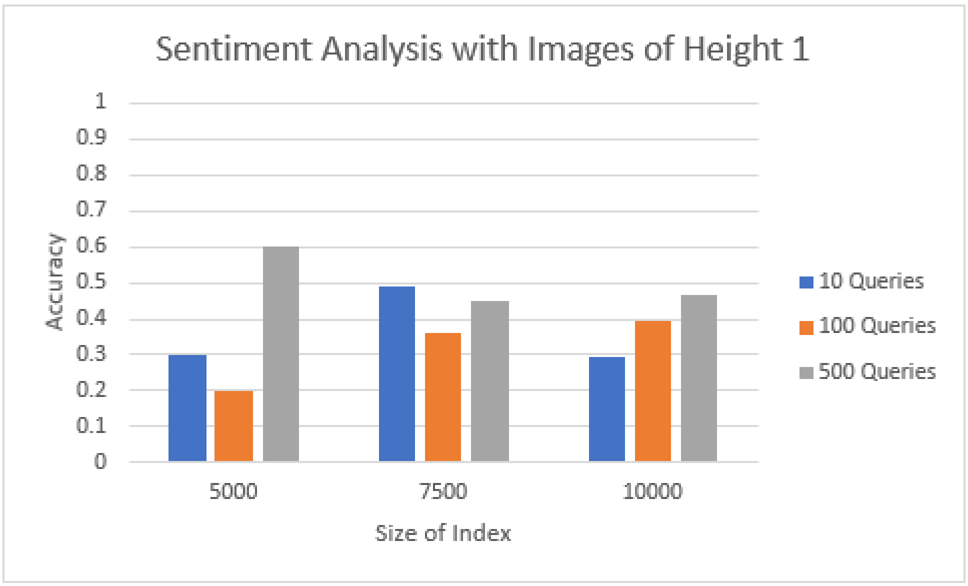


Figure 9: Sentiment Analysis For Images with 1 Row

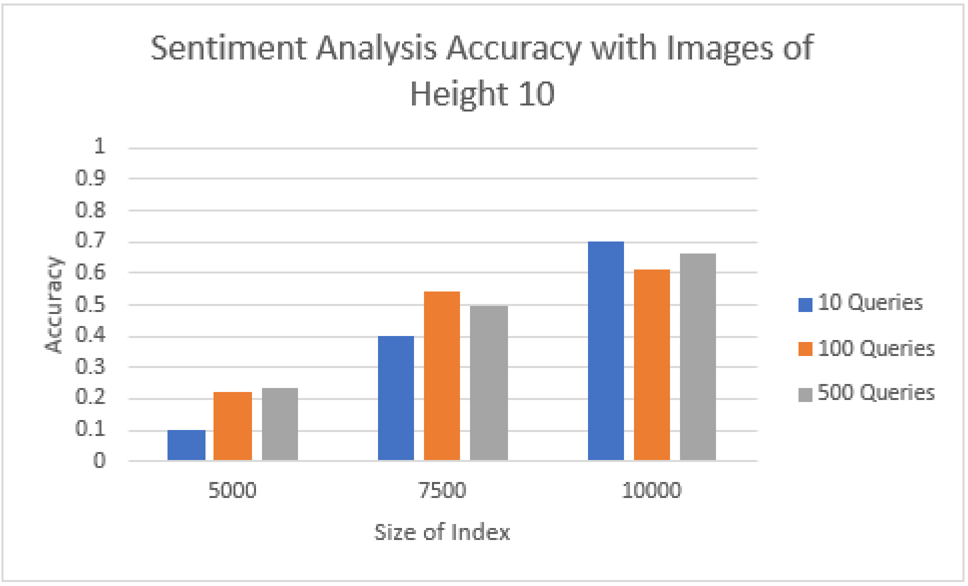


Figure 10: Sentiment Analysis for Images with 10 Rows

The data displayed in Figure 10 indicates that there is a difference in the accuracy of the sentiment classifier with regards to the size of the index and the height of the images that are used to build the index. Figure 10 indicates that the accuracy will increase as the size of the index increases. However, Figure 9 does not clearly indicate this trend. This may be due to the effect that the height of the image has on the search factor. It can be seen by comparing the results from Figure 9 and Figure 10, that the accuracy of the classifier will increase as the height of the image increases.

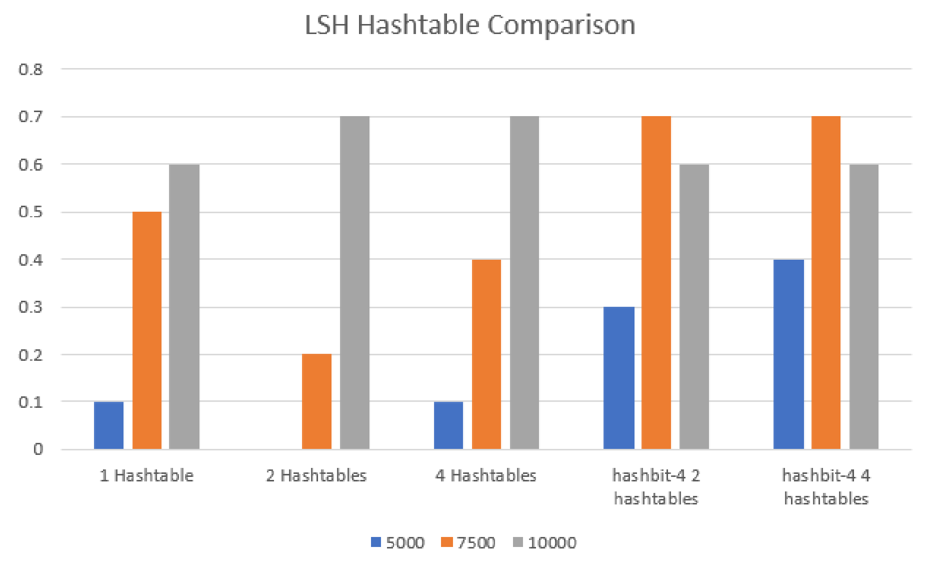


Figure 11: Modifying the number of hash tables and binary code lengths causes changes across the different index sizes.

Adjusting the number of hash tables that are used with the LSH indexer and modifying the length of the binary codes results in a difference in accuracy. Interpreting Figure 11, it can be seen that adjusting the number of hash tables and the length of the binary codes improves the accuracy of the index classifier.

9.1.2 Spectral Hashing

The SH function has also been implemented in a similar fashion to that of the LSH script that was described previously. It constructs an index which is queried to retrieve a list of similar records. The sentiment of these returned records is then averaged to determine if the sentiment of the queried text is positive or negative.

As with the LSH technique, the text is passed to a model that converts the text to an image and stores the related sentiment. The image vectors of these images are then reshaped so that each image vector has the same length. These image vectors are then used to construct the index.

Once this index has been built, it can be queried using text provided by the user. This text is converted to an image using the same process that was used by the LSH method.

Unlike with the LSH technique, the SH query request does not return the specific set of results. It will instead return the array index of all of the items that are similar to the; the number of results it will return can be limited to a set amount. The average sentiment is then calculated from the set of returned similar results and compared against a threshold value to determine if the queried text is positive or negative. This classification process is done in the same manner as the Locality Sensitive Hashing.

Figure 12 shows the accuracy of the Spectral Hashing technique. It can be seen that across the different index sizes, there is no change in the accuracy, excluding the index containing 5000 records. However, the accuracy of this model does not exceed 30%. The hashing technique used a 64-bit hash.

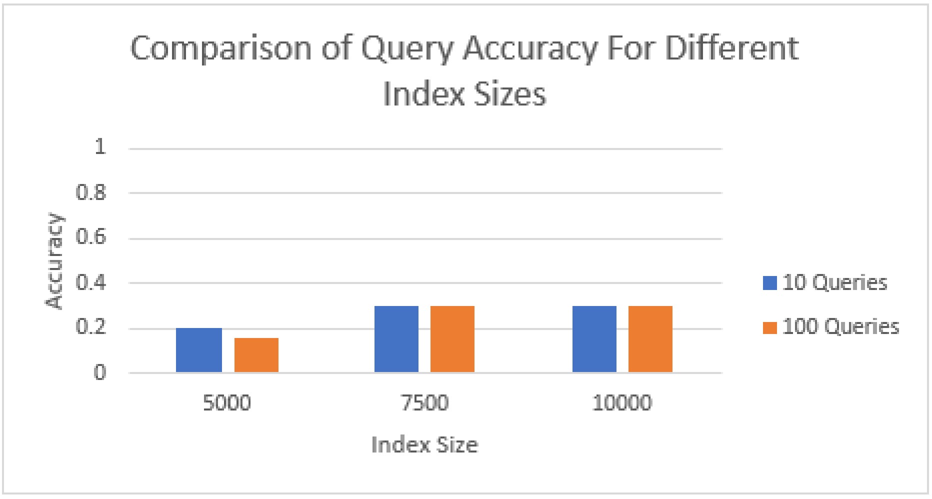


Figure 12: Comparison of Query Accuracy for Different Index Sizes for Images with 1 Row

Figure 13 indicates that the accuracy can be improved by increasing the size of the binary code. It can be seen that increasing the code above 64-bits and increasing the size of the index will result in an increase in the accuracy of the model.

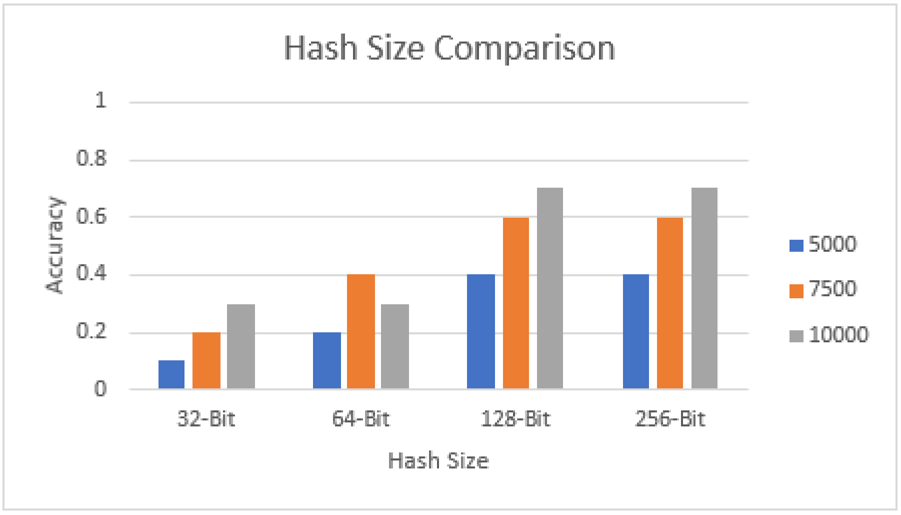


Figure 13: Hash Size Comparison

Figure 14 compares the difference in accuracy between the Spectral Hashing technique for indexes built using images with two different heights. Overall, there is little change between the results. Only the index with 7500 entries shows a difference, however further testing must be done to determine if this is an outlier.

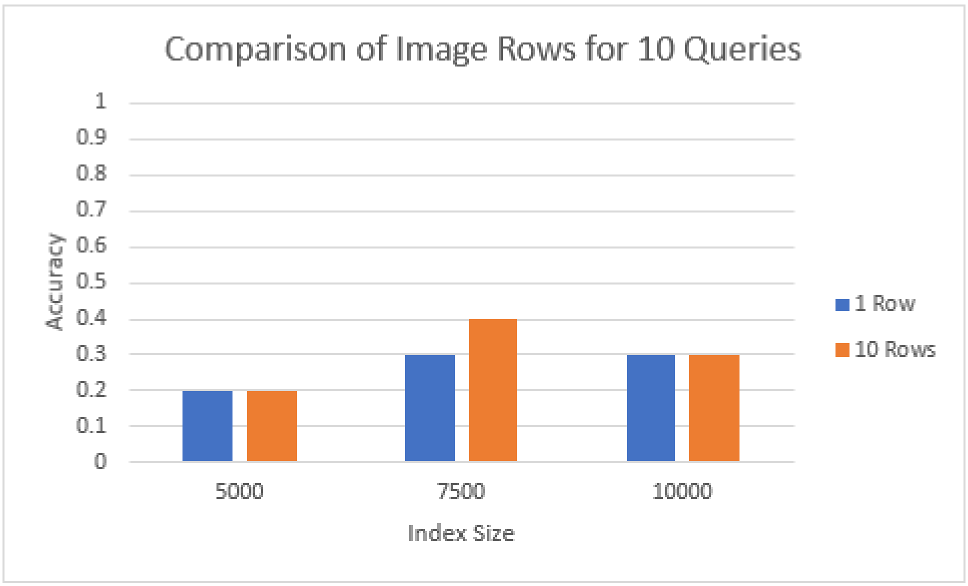


Figure 14: Comparison of Image Rows for 10 Queries

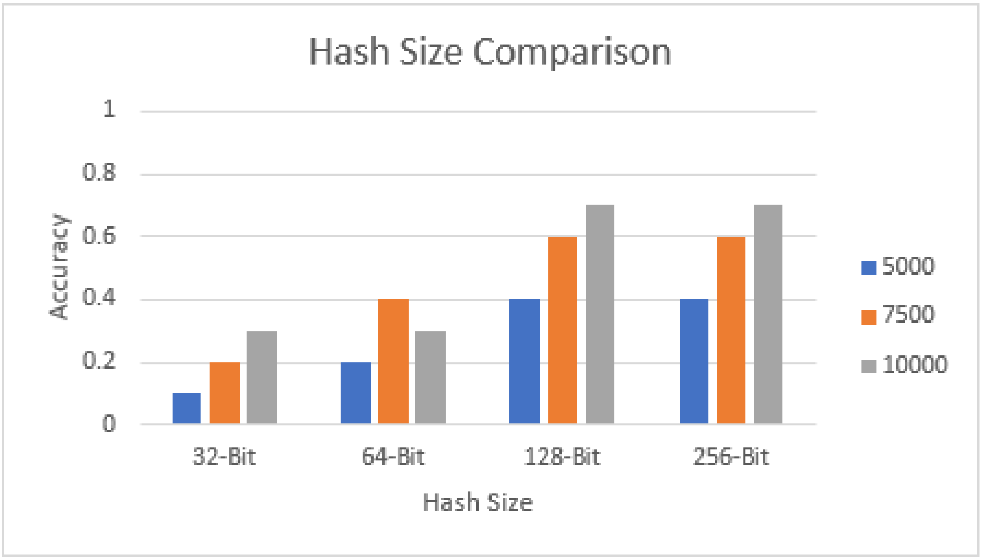


Figure 15: Increasing the binary code length of the index increases the accuracy of the index classifier.

Increasing the size of the binary code appears to increase the accuracy of the index classifier, as can be seen in Figure 15. It can be seen that the accuracy improves linearly as the bit-size and number of entries within the index increases. As such, it is believed that the binary code length has a significant effect on the classifier.

9.2 Results

9.2.1 Accuracy Results Table

Table 3: Results table for the accuracy of the different classifier types. The highest accuracy was attained by the KNN LSH Text classifier. The LSH text acted as a baseline for the other classifiers and shows that all image classifiers are below the baseline accuracy.

Dataset	Hash Type	Classifier	Hash Size	Image Shape	Image Type	Accuracy
Twitter	LSH_T	KNN	128			0.61
IMDB	LSH_T	KNN	512			0.60
Twitter	LSH	SVM	128	(40, 28)	Grayscale String	0.584
Twitter	LSH	KNN	16	(40, 28)	Grayscale String	0.58
Twitter	SH	KNN	64	(35, 32)	RGB String	0.57
IMDB	SH	SVM	8	(48, 64)	RGB Full	0.535
IMDB	SH	HE	32	(128, 64)	RGB Full	0.53
IMDB	SH	GSVM	8	(48, 64)	RGB Full	0.5316
IMDB	LSH	SVM	128	(128, 128)	RGB String	0.53
IMDB	SH	KNN	16	(48, 64)	RGB Full	0.53
IMDB	LSH	GSVM	128	(128, 128)	RGB String	0.521
IMDB	LSH	KNN	128	(64, 64)	RGB String	0.51
IMDB	LSH	HE	128	(128, 128)	RGB String	0.50

9.2.2 KNN Results

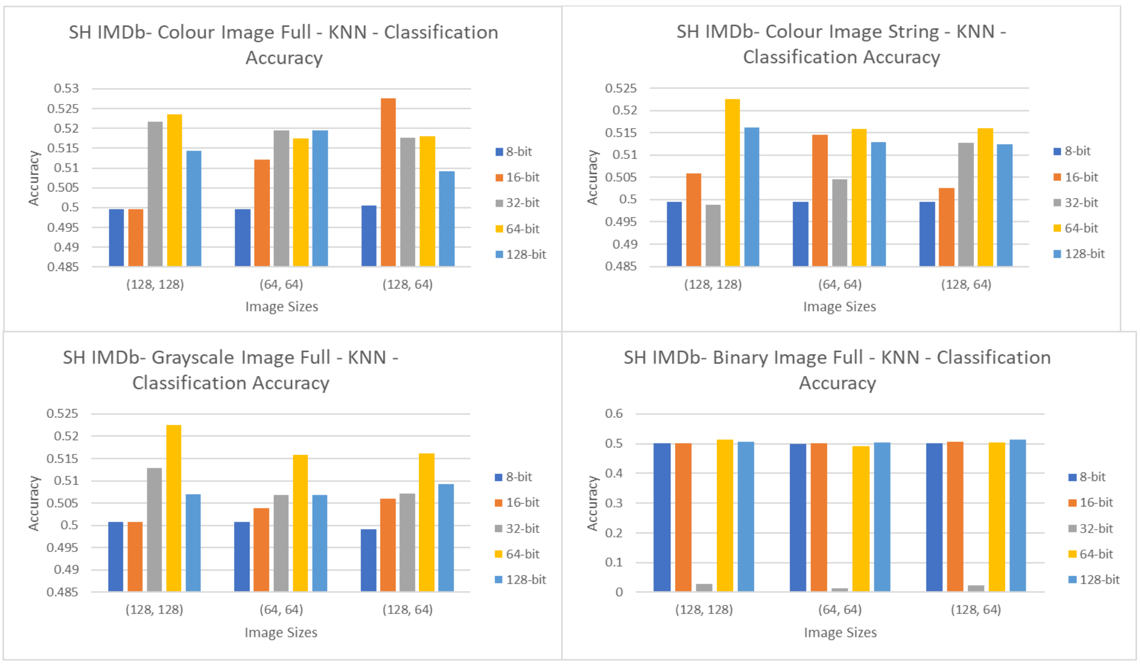


Figure 16: A comparison of the accuracy of the different image sizes and the binary code length for the SH IMDB KNN results for different image types. It can be seen that there are no real trends within the data, and it could be assumed that the image shape and binary code length may not affect the accuracy across different types of images.

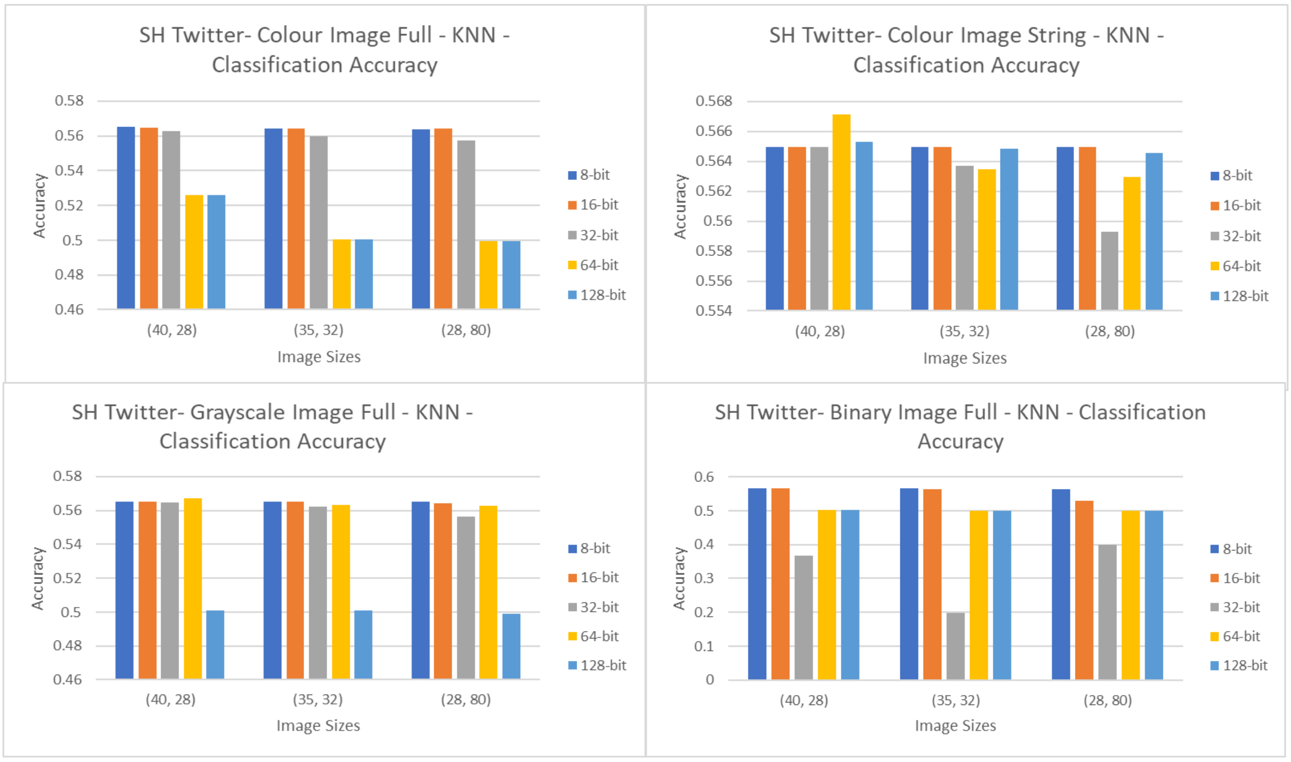
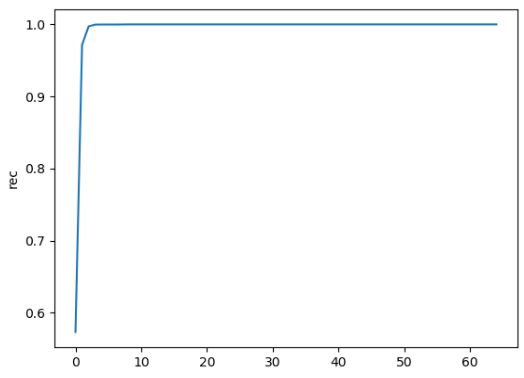
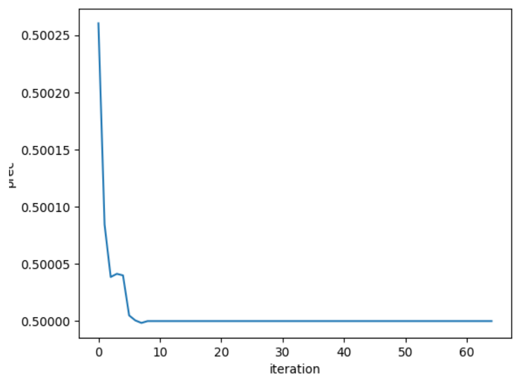


Figure 17: A comparison of the accuracy of the different image sizes and the binary code length for the SH Twitter KNN results for different image types. It can be seen that there are no real trends within the data, and it could be assumed that the image shape and binary code length may not affect the accuracy across different types of images.

9.2.3 Deep Hashing Results

All results retrieved from the Deep Hashing algorithm appear similar to the results presented below. To view more of these results, navigate to the results section at: https://github.com/sbett7/Text_to_Image_Classifier





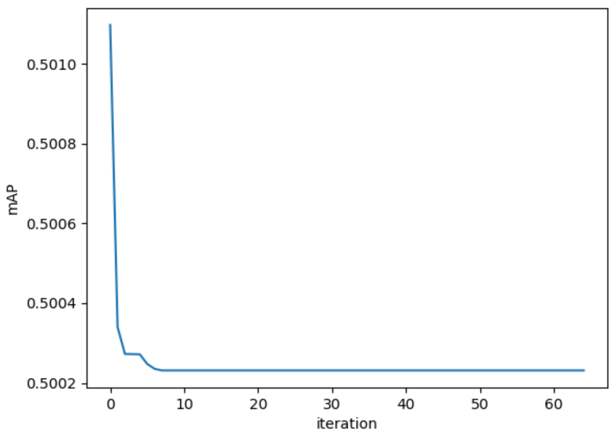


Figure 18: A set of results from the DCH algorithm. It can be seen that the mAP settles at 50% error. As such it can be seen that the Deep Hashing algorithm cannot accurately classify the generated images.

9.3 Code Samples

All code can be found at https://github.com/sbett7/Text_to_Image_Classifier.

9.3.1 Generate Images

```python
import numpy as np
import csv
import cv2 as cv

class ImageClass:
    _text = ""
    _image_shape = 1
    data = np.zeros((1, 1), dtype=int)
    sentiment = 0
    id = 0
    _full_image = False
    _is_colour = False
    _gray_scale = False

    # CONSTANTS
    _ROW = 0
    _COLUMN = 1

    ID = 0
    SENTIMENT = 1
    TEXT = 2

    _HEX = 16
    _TWO_CHARACTERS = 2

    def __init__(self, input_text, id_val=0, sentiment=0, image_shape=(32, 70), full_image=False,
                 is_colour=False, gray_scale=False):
        '''
        Converts a set of text to an image based upon the given set of image configurations.
        :param input_text:
        :param id_val: the associated ID for the text if it were to come from a set of text.
        This value is not required.
        :param image_shape: An array specifying the width and height of the image to generate.
        :param full_image: A boolean specifying whether the text will be set to fill the entire image (True) or the
         image will only contain a single iteration of the text string (False).
        :param is_colour: A boolean specifiying whether the generated image will be an RGB Image (True), or a Binary
        Image (False).
        :param gray_scale: A boolean that specifies whether an image will be generated as gray_scale or smoothed
(True),
        or if the image will be a binary image. If color_images is True, the resulting image will be smoothed.
        '''
        self._full_image = full_image
        self._is_colour = is_colour
        self._gray_scale = gray_scale
        self._image_shape = image_shape
        self._text = input_text
        self.id = id_val
        self.sentiment = sentiment

        # convert text to image
        self.initialise_array()
        self.convert_to_image()

    def convert_to_image(self):
        '''
        Converts stored text to an image based on the image configurations.
        :return: None
        '''
        if self._full_image:
            if self._is_colour:
                self.convert_string_to_colour_full_image()
            elif self._gray_scale:
                self.convert_string_to_binary_full_image()
                self.data = self.smooth_image()
            else:
                self.convert_string_to_binary_full_image()
        else:
            if self._is_colour:
                self.convert_string_to_colour_image()
            elif self._gray_scale:
                self.convert_string_to_binary()
                self.data = self.smooth_image()
            else:
                self.convert_string_to_binary()

    def convert_string_to_binary_full_image(self):
        '''
        converts the text string to a binary image by converting the text to a binary string and assigning the pixel
        value to 255 or 0 based upon the corresponding binary string character.
        The binary string will be iterated through until the image has been filled.
        :return: None
        '''
        # get binary string
        binary_string = ''.join(format(ord(x), 'b') for x in self._text.strip())

        counter = 0
        row = 0
        column = -1

        # until every element in the image array has been iterated over
        while counter < self.data.shape[self._ROW] * self.data.shape[self._COLUMN]:
            for character in binary_string:
                # increment  column and counter iterators
                column = column + 1
                counter = counter + 1

                # if the number of columns in the image matches the column iterator, move to the next row
                if column == self.data.shape[self._COLUMN]:
                    column = 0
                    row = row + 1
                # if the index has reached the end of the image, break from loop
                if row == (self.data.shape[self._ROW]):
                    break

                # set image value
                if character == '1':
                    self.data[row][column] = 255
                else:
                    self.data[row][column] = 0

    def convert_string_to_binary(self):
        '''
        converts the text string to a binary image by converting the text to a binary string and assigning the pixel
        value to 255 or 0 based upon the corresponding binary string character.
        The binary string will be iterated through once, all pixels in the image that have not been iterated over will
        remain as their initial value.
        :return: None
        '''
        # get binary string
        binary_string = ''.join(format(ord(x), 'b') for x in self._text.strip())

        row = 0
        column = -1
        for character in binary_string:
            # increment  column iterator
            column = column + 1

            # if the number of columns in the image matches the column iterator, move to the next row
            if column == self.data.shape[self._COLUMN]:
                column = 0
                row = row + 1

            # if the index has reached the end of the image, break from loop
            if row == (self.data.shape[self._ROW]):
                break

            # set image value
            if character == '1':
                self.data[row][column] = 255
            else:
                self.data[row][column] = 0

    def convert_string_to_colour_full_image(self, magnitude=2):
        '''
        converts the text string to an RGB image by converting the text to a hexadecimal string and assigning each
        hexadecimal value to a colour element of a pixel.
        The hexadecimal string will be iterated through until the image has been filled.
        :param magnitude: A scaling integer to multiply the value for each pixel.  This is used to make it easy to
view
        the pixel values in the generated image.
        :return: None
        '''
        # get hex string
        hex_string = self._text.strip().encode('hex')

        row = 0
        column = -1
        rgb_next = 0
        RED = 0
        BLUE = 1
        GREEN = 2
        red = 0
        blue = 0
        green = 0
        counter = 0
        characters = ""  # variable to store hex characters

        # until every element in the image array has been iterated over
        while counter < self.data.shape[self._ROW] * self.data.shape[self._COLUMN]:
            for character in hex_string:
                characters = characters + character
                # if there are two characters in characters variable, add to image and reset variable
                if len(characters) == self._TWO_CHARACTERS:
                    if rgb_next == RED:
                        rgb_next = BLUE
                        red = int(characters, self._HEX)
                    elif rgb_next == BLUE:
                        rgb_next = GREEN
                        blue = int(characters, self._HEX)
                    else:  # reset colour variable to RED, and set the values of the image element
                        rgb_next = RED
                        green = int(characters, self._HEX)
                        column = column + 1
                        counter = counter + 1

                        # if the number of columns in the image matches the column iterator, move to the next row
                        if column == self.data.shape[self._COLUMN]:
                            column = 0
                            row = row + 1
                        # if the index has reached the end of the image, break from loop
                        if row == (self.data.shape[self._ROW]):
                            break

                        # set values and multiply by the given magnitude.
                        self.data[row][column][RED] = red * magnitude
                        self.data[row][column][BLUE] = blue * magnitude
                        self.data[row][column][GREEN] = green * magnitude
                    characters = ""

    def convert_string_to_colour_image(self, magnitude=2):
        '''
        converts the text string to an RGB image by converting the text to a hexadecimal string and assigning each
        hexadecimal value to a colour element of a pixel.
        The hexadecimal string will be iterated through once, all pixels in the image that have not been iterated
        over will remain as their initial value.
        :param magnitude: A scaling integer to multiply the value for each pixel.  This is used to make it easy to
view
        the pixel values in the generated image.
        :return: None
        '''
        # get hex string
        hex_string = self._text.strip().encode('hex')

        row = 0
        column = -1
        rgb_next = 0
        RED = 0
        BLUE = 1
        GREEN = 2
        red = 0
        blue = 0
        green = 0

        characters = ""  # variable to store hex characters
        for character in hex_string:
            characters = characters + character

            # if there are two characters in characters variable, add to image and reset variable
            if len(characters) == self._TWO_CHARACTERS:
                if rgb_next == RED:
                    rgb_next = BLUE
                    red = int(characters, self._HEX)
                elif rgb_next == BLUE:
                    rgb_next = GREEN
                    blue = int(characters, self._HEX)
                else:  # reset colour variable to RED, and set the values of the image element
                    rgb_next = RED
                    green = int(characters, self._HEX)
                    column = column + 1
                    if column == self.data.shape[self._COLUMN]:
                        column = 0
                        row = row + 1

                    # if the index has reached the end of the image, break from loop
                    if row == (self.data.shape[self._ROW]):
                        break

                    # set values and multiply by the given magnitude.
                    self.data[row][column][RED] = red * magnitude
                    self.data[row][column][BLUE] = blue * magnitude
                    self.data[row][column][GREEN] = green * magnitude
                characters = ""

    def initialise_array(self):
        '''
        Initialises the image array based upon whether the image is a RGB or binary image
        :return: None
        '''
        if self._is_colour:
            self.data = np.zeros((self._image_shape[0], self._image_shape[1], 3))
        else:
            self.data = np.zeros((self._image_shape[0], self._image_shape[1]))

    def get_image_vector(self):
        '''
        Gets the image vector of the generated image and returns it.  The image vector is generated depending upon the
        type of image.
        :return: An image vector representation of the generated image.
        '''
        if self._gray_scale:
            self.data = self.smooth_image()

        if self._is_colour:
            img_vector = self._get_colour_image_vector()
        else:
            img_vector = self._get_binary_image_vector()

        return img_vector

    def _get_colour_image_vector(self):
        '''
        Converts the RGB image to an image vector.  This is done by averaging the RGB values of each element, and then
        averaging each column to generate the vector.
        :return: A 1-D image vector representation of the generated image.
        '''
        img_vector = np.zeros((1, self._image_shape[self._COLUMN]), dtype=float)
        gray_scale = np.zeros((self._image_shape[self._ROW], self._image_shape[self._COLUMN]), dtype=float)

        # get average RGB color across every pixel in image
        for i in range(0, self._image_shape[self._ROW] - 1):
            for j in range(0, self._image_shape[self._COLUMN] - 1):
                gray_scale[i, j] = np.mean(self.data[i, j])

        # get average values of each column
        for i in range(0, self._image_shape[self._COLUMN] - 1):
            img_vector[self._ROW][i] = np.mean(gray_scale[:, i])

        return img_vector

    def _get_binary_image_vector(self):
        '''
        Converts the binary image to an image vector.  This is done by averaging each column to generate the vector.
        :return: A 1-D image vector representation of the generated image.
        '''

        img_vector = np.zeros((1, self._image_shape[1]), dtype=float)

        # get average values of each column
        for i in range(0, self._image_shape[1] - 1):
            img_vector[0][i] = np.mean(self.data[:, i])
        return img_vector

    def smooth_image(self):
        '''
        Smooths the image using a 5x5 window.
        :return: The smoothed image
        '''
        kernel = np.ones((5, 5), np.float32) / 25
        return cv.filter2D(self.data, -1, kernel)

    def write_to_image(self, image_name):
        '''
        Writes the image data to an image with the specified name.  The location of the images will be stored in the
        Images folder in the current directory.
        :param image_name:  A string specifying the name of the image.
        :return: None
        '''
        cv.imwrite('Images/{}.jpg'.format(image_name), self.data)

    @staticmethod
    def convert_text_to_image(path_to_store, filename, image_shape=10, is_color=False,
                              full_image=False, gray_scale=False):
        '''
        Converts a set of text to images and writes the images to the specified location.
        :param path_to_store: A string specifying the location for the images to be stored.
        :param filename: A string specifying the name of the image.
        :param image_shape: An array specifying the width and height of the image to generate.
        :param is_color: A boolean specifiying whether the generated image will be an RGB Image (True), or a Binary
        Image (False).
        :param full_image: A boolean specifying whether the text will be set to fill the entire image (True) or the
         image will only contain a single iteration of the text string (False).
        :param gray_scale: A boolean that specifies whether an image will be generated as gray_scale or smoothed
(True),
        or if the image will be a binary image. If color_images is True, the resulting image will be smoothed.
        :return:
        '''

        # open file and read in data
        with open(filename, 'rt') as csvfile:
            reader = csv.reader(csvfile, delimiter=',', quotechar='|')
            for row in reader:
                if row[0] != "ItemID":
                    item = ImageClass(row[2], int(row[0]),
                                      sentiment=row[1], image_shape=image_shape,
                                              is_colour=is_color, full_image=full_image, gray_scale=gray_scale)
                        cv.imwrite(path_to_store + "TrainImage" + str(item.id) + '.jpg', item.data)
```

9.3.2 Get SH Codes for IMDB Dataset

```
from os import listdir
from os.path import isfile, join
from SH.HdidxClass_Test import HdidxClass
import numpy as np
import random

db_directory = 'F:¥Documents¥PycharmProjects¥SpectralHashing¥SH¥Data¥IMDB¥¥'
positive_file_location = 'F:¥Downloads¥¥aclImdb_v1(1)¥¥aclImdb¥¥train¥pos'
negative_file_location = 'F:¥Downloads¥¥aclImdb_v1(1)¥¥aclImdb¥¥train¥¥neg'
positive_files = [f for f in listdir(positive_file_location) if isfile(join(positive_file_location, f))]
num_pos_files = len(positive_files)
file_counter = 0
negative_files = [f for f in listdir(negative_file_location) if isfile(join(negative_file_location, f))]

files = positive_files + negative_files
iRows = [(128, 128), (64, 64), (48, 64), (128, 64)]
nbits_array = [8, 16, 32, 64, 128]
database_size = len(files)
sizeVals = len(files)
balance = int(sizeVals * 0.5)
positiveCounter = 0
negativeCounter = 0
color_image = False
full_image = False
gray_scale = False

IS_COLOR = 0
IS_FULL_IMAGE = 1
IS_GRAYSCALE = 2
PATH = 3

configuration = [[False, False, False, "blackwhite/string_image"],
                 [True, False, False, "color/string_image/not_smooth"],
                 [True, False, True, "color/string_image/smooth"],
                 [True, True, False, "color/full_image/not_smooth"],
                 [True, True, True, "color/full_image/smooth"],
                 [True, False, True, "color/string_image/smooth"],
                 [False, True, True, "grayscale/full_image"],
                 [False, True, False, "blackwhite/full_image"],
                 [False, False, True, "grayscale/string_image"]]

text_data = []
ran = random.sample(range(0, len(files)), len(files))
mid_point = database_size / 2
data = []
for text in files:
    if file_counter < num_pos_files:
        with open('{}/{}'.format(positive_file_location, text),mode='r') as data:
            text_data.append([data.read(), 1])
    else:
        with open('{}/{}'.format(negative_file_location, text), mode='r') as data:
            text_data.append([data.read(), 0])
    file_counter = file_counter + 1
text = []
for values in text_data:
    text.append(values[0])

# for every values
for config in configuration:
    isColor = config[IS_COLOR]
    isFull = config[IS_FULL_IMAGE]
    isGray = config[IS_GRAYSCALE]
    path = config[PATH]
    for iRow in iRows:
        for nbit in nbits_array:
            print("Config: {}       iRow: {}     nBit: {}".format(path, iRow, nbit))
            image_rows = iRow
            nbits = nbit
            hdidx = HdidxClass(text, 100, image_shape=image_rows, hash_size=nbits,
                               gray_scale=isGray, full_images=isFull, color_images=isColor)
            database = open('{}/{}bit/({}, {})/{}/database.csv'.format(db_directory, nbit, iRow[0], iRow[1], path),
                            mode='w')
            for value in ran:
                data = text_data[value]
                query = (hdidx.get_query_code(data[0], data[1], 0, rows=image_rows,
                                              gray_scale=isGray, full_image=isFull, color_image=isColor))[0]
                print(query)
                database.write(np.array2string(query) + ", {}¥n".format(int(data[1])))
```

9.3.3 KNN Classifier

```
import numpy as np
import csv
from sklearn.neighbors import KNeighborsClassifier
import time


db_path = 'F:¥Documents¥PycharmProjects¥SpectralHashing¥LSH¥Data¥Twitter'  # Path to where data is
stored
results_path = 'F:¥Documents¥PycharmProjects¥SpectralHashing¥LSH¥Data¥Twitter¥¥results_knn.csv'  #
Editing data
is_LSH = True
iRows_data = [(28, 80), (35, 32), (40, 28)]
nbits_array = [8, 16, 32, 64, 128]
neighbours_array = [1, 4, 20, 50, 100, 250]
types_array = ['euclidean', 'hamming']
num_data = 0

configuration = [[False, False, False, "blackwhite/string_image"],
                 [True, True, False, "color/full_image/not_smooth"],
                 [False, True, True, "grayscale/full_image"],
                 [False, True, False, "blackwhite/full_image"],
                 [False, False, True, "grayscale/string_image"]]


def split_string(is_lsh, data_val):
    '''
    Splits the data based on whether the data is space separated or not.
    :param is_lsh: A boolean that specifies whether the data being used is LSH.
    :param data_val: The string data that is to be processed.
    :return: Returns the split data string.
    '''
    if is_lsh:
        return str.split(data_val)[0]
    else:
        return str.split(data_val)


def get_length_of_data(data_row):
    '''
    Gets the size of the row data.
    :param data_row: The text sample with the data.
    :return: The number of entries per text sample.
    '''
    dat_val = data_row[0]
    dat_val = dat_val[1:len(dat_val) - 1]
    value = split_string(is_LSH, dat_val)
    value_array = []
    for vals in value:
```

```python
        value_array.append(int(vals))
    return len(value_array)


def process_data(rows, data_len, data_width):
    '''
    Converts all of the row data to a format that can be passed to classifier.
    :param rows: a list of rows with the data to process.
    :param data_len: The number of rows to be processed.
    :param data_width: The number of entries per entry.
    :return: The converted data array, and the label array.
    '''
    i = 0
    j = 0
    data_array = np.zeros((data_len, data_width), dtype=float)
    label_array = np.zeros((data_len, 2))
    label_temp = 0
    for row in rows:
        dat = row[0]
        dat = dat[1:len(dat) - 1]
        values = split_string(is_LSH, dat)
        j = 0
        for val in values:
            data_array[i, j] = val
            j = j + 1
        try:
            label_temp = int(row[1])
        except ValueError:
            label_temp = int(row[1][0])
        if label_temp == 1:
            label_array[i, 0] = 1
            label_array[i, 1] = 0
        else:
            label_array[i, 0] = 0
            label_array[i, 1] = 1
        i = i + 1
    return data_array, label_array

with open(results_path, mode='w') as result_data_file:
    field_names = ['Hash Size', 'Image Shape', 'Image Type', 'KNN Type', 'Neighbours', 'Accuracy', 'Fit
Time',
                                               'Average Query Time']
    writer = csv.DictWriter(result_data_file, fieldnames=field_names)
    writer.writeheader()
    for nbit in nbits_array:
        for iRows in iRows_data:
            for config in configuration:
                path = config[3]
                data_file = '{}/{}bit/({}, {})/{}/database.csv'.format(db_path, nbit, iRows[0],
iRows[1], path)

                with open(data_file, 'rt') as csvfile:
                    reader = csv.reader(csvfile, delimiter=',', quotechar='|')
                    len_of_data = len(list(reader))

                with open(data_file, 'rt') as csvfile:
                    reader = csv.reader(csvfile, delimiter=',', quotechar='|')
                    for row in reader:
                        num_data = get_length_of_data(row)
                        break

                with open(data_file, 'rt') as csvfile:
                    reader = csv.reader(csvfile, delimiter=',', quotechar='|')
                    data, labels = process_data(reader, len_of_data, num_data)

                # get data samples
                np.random.seed(42)
                indices = np.random.permutation(len(data))
                n_training_samples = int(data.shape[0] * 0.5)
                learnset_data = data[indices[:-n_training_samples]]
                learnset_labels = labels[indices[:-n_training_samples], 0]
                testset_data = data[indices[-n_training_samples:]]
                testset_labels = labels[indices[-n_training_samples:], 0]

                for neighbour in neighbours_array:
                    for types in types_array:
                        # Create and fit a nearest-neighbor classifier
                        knn = KNeighborsClassifier(algorithm='auto', leaf_size=30, metric=types,
                                                   metric_params=None,
                                                   n_jobs=1, n_neighbors=neighbour, weights='uniform')

                        # fit data and get time taken to fit
                        fit_start = time.time()
                        knn.fit(learnset_data, learnset_labels)
                        fit_time = time.time() - fit_start

                        # predict data and get query time
                        query_start = time.time()
                        results = knn.predict(testset_data)
                        query_end = time.time()
                        query_time = float(query_end - query_start) / float(len(testset_labels))

                        correctResults = 0
                        i = 0
                        for result in results:
                            if result == testset_labels[i]:
                                correctResults = correctResults + 1
                            i = i + 1
                        accuracy = float(correctResults) / float(len(testset_labels))

                        writer.writerow({
                            'Hash Size': nbit,
                            'Image Shape': '({}, {})'.format(iRows[0], iRows[1]),
                            'Image Type': path,
                            'KNN Type': types,
                            'Neighbours': neighbour,
                            'Accuracy': accuracy,
                            'Fit Time': fit_time,
                            'Average Query Time': query_time
                        })
                        print('Hash Size: {}, Image Shape: ({}, {}), Image Type: {}, KNN Type: {},
Neighbours: {},'
                              ' Accuracy: {}, Fit Time: {}, Average Query Time: {}'.format(nbit,
iRows[0], iRows[1],
path, types, neighbour,
accuracy, fit_time,
query_time))
```