ENGG 680: Introduction to Digital Engineering
Final Project

Adam Smith

December 14, 2022

1 Introduction

In this project, we will perform sentiment analysis on the IMDB movie review dataset using a deep learning approach [1]. This dataset consists of 50,000 movie reviews evenly split between training and testing sets. Each review is associated with a positive label, corresponding to a rating of 7/10 or greater, or a negative label, corresponding to a rating of 4/10 or less. The training and testing sets each contain an equal number of positive and negative labels. Here, we perform a simple data processing step to vectorize and embed the reviews, and develop a simple naive Bayesian classifier as well as a convolutional network for prediction. We then utilize the preprocessed and vectorized data provided by Keras to develop a simple fully-connected network. Results of the models are then compared.

2 Data Analysis and Processing

The deep learning library Keras offers easy access to a preprocessed and vectorized version of the IMDB movie review dataset. In their documentation, they state that words are indexed by their overall frequency in the dataset such that, for example, the integer 5 encodes the 5th most common word in the dataset. If we compute the number of unique words in their training and testing sets, we find these values to be 88,585 and 51,725 respectively. Since the datasets should have been drawn from the same distribution, we expect these values to be similar, but their difference suggests they are removing words from the testing set which are not present in the testing set, although they do not state this in their documentation. We can confirm this by computing the number of words in the testing set that are not present in the training set, and we find this value to be 0. To ensure I know the exact processing steps that are applied to the data, and to learn more about data processing for NLP applications, I decided to first try beginning from the unprocessed data and doing the vectorization myself.

Printing a few reviews, we see that they contain lots of unwanted punctuation and some HTML line break tags. Thus, we begin by removing these and lowercasing the reviews. We then tokenize the reviews using the `word_tokenizer` method provided by the python library NLTK. Stop words are words such as 'have', 'the', or 'over' which occur frequently in text data but do not add much semantic meaning to the text, at least as far as numerical models are concerned. Thus, we remove these next. The remaining words are then lemmatized with NLTK's `WordNetLemmatizer` class. Lemmatization is the process of converting a word into its root form. For example, 'working' becomes 'work', and 'better' becomes 'good'. To lemmatize, we must first obtain the part-of-speech (POS) of the word (i.e. noun, verb, etc.). We use the `synsets` method from NLTK's `wordnet` module to obtain all possible POS for a word, and choose the one that occurs most frequently. Below is an example of a review before and after processing:

This picture started out with good intentions, Bacon the scientist out to test the theory of invisibility, and Shue is cute as usual in her role. It all falls apart after that, it's your typical Hollywood thriller now, filmed on a soundstage with special effects galore, minus any kind of humour, wit or soul. In other words, don't waste your time watching this. Get the audiocassette tape with John DeLancie as the Invisible Man instead, also starring Leonard Nimoy. Now that was good, and HG Wells is well served, unlike with this mess.

picture start good intention bacon scientist test theory invisibility shue cute usual role fall apart typical hollywood thriller film soundstage special effect galore minus kind humour wit soul word waste time watch get audiocassette tape john delancie invisible man instead also star leonard nimoy good hg well well serve unlike mess

Running these text processing steps on the entire dataset takes roughly 90 seconds on my machine. However, it can be easily parallelized, so I use the python package `multiprocess` to run several processes concurrently. This module functions identically to the python standard library module `multiprocessing`, but serializes using `dill` rather than `pickle`, which is often necessary to function properly in IPython environments. Running the text processing concurrently reduces run time to < 20 seconds.

3 Models

3.1 Naive Bayesian

We first define a simple naive Bayesian classifier to set a baseline for the deep learning models. To vectorize the reviews, we use scikit-learn's `CountVectorizer` class, which returns a matrix representation of the data. Each row in the matrix is a vector with length equal to the size of the vocabulary and corresponds to a review. The elements in the vector are the counts of the corresponding word in the review. As the matrix contains many zeros, the matrix is represented by a `scipy.sparse._csr.csr_matrix`, which drastically reduces memory usage and speeds up training of the model.

The model is then created with scikit-learn's `MultinomialNB` class and fit on the training data. The accuracies of the trained model are 90.27% and 81.98% on the training and testing sets respectively. The model clearly overfits on the training data, but still does an alright job classifying the testing set reviews. As modelling language is a fairly complex task, I believe a deep learning model will be able to better approximate the sentiment-predicting function we would like to obtain. Moreover, deep learning models have more freedom and potential for customization that traditional machine learning models, so we should be able to fine-tune the network for our task and better apply regularization methods to prevent overfitting.

3.2 Convolutional Network

Since the semantic meaning of a word in a sentence is heavily influenced by its surrounding words, but less so by words further away in the sentence, it may be beneficial to have a method that considers local groups of words rather than the review as a whole. We should be able to achieve this with a convolution operation, which will run over each review sequentially and consider only local groups of words. We can then apply a maximum pooling operation for dimensionality reduction and to extract the most prominent local features. Once we have a feature-rich low-dimension representation of the review, we can apply a couple fully-connected layers to map the features into a sentiment prediction.

To produce a convolutional network model, we will use a slightly more sophisticated text embedding. We begin by vectorizing the reviews using Keras' `TextVectorization` layer. This layer transforms each word in a review to its index in the dataset vocabulary, and either zero-pads or truncates the vector so that it has a specified length. To determine what sequence length to supply, we plot a histogram of the number of words in each review in the training set, which is shown in Fig. 1. From this figure, we see that the distribution is skewed right and has a maximum of 1429, but the vast majority of reviews have a far lower word count. We will set

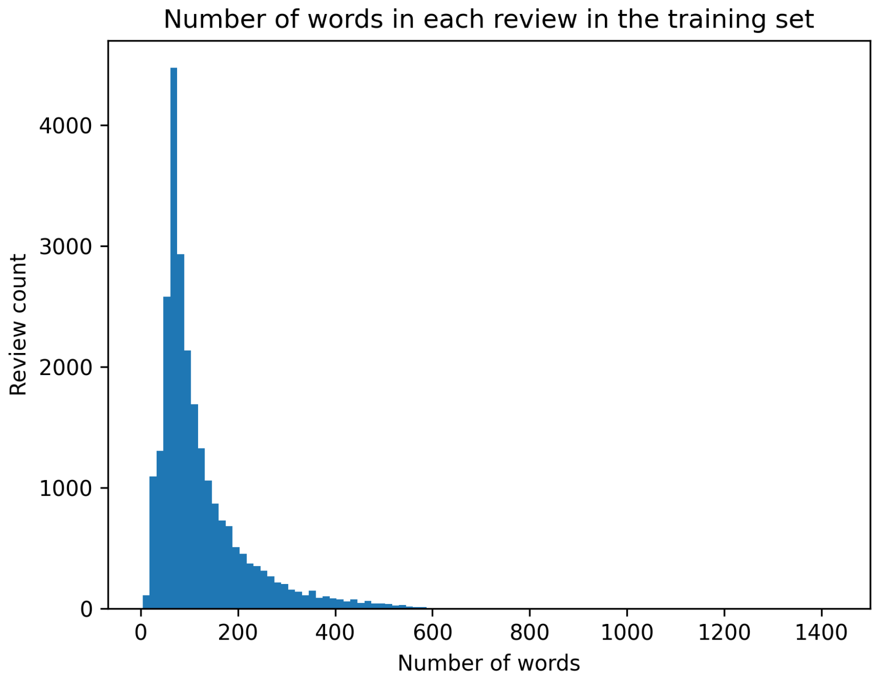

Figure 1: Histogram of the number of words in each review in the training dataset.

the sequence length to 300 so that we capture the entirety of the majority of reviews and a large portion of the remaining ones. Likely, the first 300 words of a long review is enough to determine sentiment.

Once we have the vectorized reviews, we can use a pre-trained word embedding model to embed the reviews into vectors with semantic meaning. The model we will use is the Global Vectors for Word Representation (GloVe) model [2]. These embeddings add structure to the word vectors such that semantically similar words are grouped together in the resulting vector space. We can load the word embeddings in from text files and choose between 50, 100, 200, or 300 dimensional representations. During model tuning, I found that the 100 dimensional representations performed best. Applying this vectorization and word embedding transforms the reviews into arrays of shape (300, 100). It is also possible to allow Keras to learn its own embedding during training via the `Embedding` layer. However, this adds a large amount of parameters to the model and it did not lead to a better classification accuracy compared to the GloVe embeddings, so we will stick with those.

Another thing to note is that words which do not appear in the training set are mapped to zero-vectors in the embedding layer. In an attempt to remedy this, I modified the embedding matrix (which maps word indices to GloVe embeddings) to include all words for which there is a GloVe representation rather than just words in the training set. This leads to a very large embedding matrix (400,000 by 100) which significantly increases memory usage and training time. Moreover, it had very little effect on the model's test set accuracy, so I found it better to stick with the reduced embedding matrix. However, we still have many words in the testing set which do not appear in the training set. Out of the 58,286 unique words in the testing set, 24,283 of them do not appear in the training set. This will lead to many zero-vectors in our embedded representation of the test reviews. With this kind of embedding, we cannot avoid this, but we can group the zero-vectors towards the end of the embedding rather than having them distributed throughout. This will hopefully make the embedding of a review in the test set seem more like a short review rather than a review with missing pieces. To do this, we remove all words in the testing set which do not appear in the training set. The sequence length of our embedding is still set at 300, but at least we will be able to fit more (or potentially all) of the words for which we do have an embedding into this 300 word cutoff.

After the embedding, the model architecture is as follows:

1. 1D convolution with 128 filters, a kernel of size 7, and ReLU activation;
2. 1D maximum pooling with a kernel size of 5 and stride of 5;
3. 1D convolution with 128 filters, a kernel of size 5, and ReLU activation;
4. 1D maximum pooling with a kernel size of 5 and stride of 5;
5. 1D convolution with 128 filters, a kernel of size 5, and ReLU activation;
6. 1D global maximum pooling;
7. dense layer with 32 neurons and a ReLU activation;
8. dropout layer with $p = 0.5$;
9. dense layer with a single neuron and no activation.

All convolution layers have a stride of 1 and no padding or dilation. We do not apply activation to the last layer since we can combine this with the binary crossentropy loss function by setting the argument `from_logits=True`. This allows Keras to combine the activation and loss into one calculation and apply the log-sum-exp trick for numerical stability.

This model is trained for 5 epochs with the Adam optimizer, a batch size of 256, and a learning rate of 0.001. The accuracy curves during training are shown in Fig. 2.

The model achieves a training accuracy of roughly 92%. Testing accuracy varies significantly between training runs, but is typically within the range of 83-86%. The model quite easily overfits on the training data, suggesting that it is possibly too complex to model effectively. However, reducing complexity or applying regularization techniques did not seem to improve model performance or overfitting. Thus, to achieve a better model, we likely need to overhaul the model architecture or process/vectorize/embed the data in a different way.

3.3 Fully-Connected Network

Since we were unable to achieve the project requirements on the previous model, we will create a simple fully-connected model to classify the data provided by Keras. Since this binary sentiment classification is a relatively simple NLP task, the previous convolutional model and data processing algorithm were likely overkill for the task. Thus, we will greatly simplify the model architecture and data processing here in hopes of producing a viable model.

We begin by loading the data from Keras and embedding the reviews with the technique suggested in the assignment. We set the vocabulary size to 10,000 such that we are only considering the top 10,000 most frequent words, and then embed each review into a vector of length 10,000 where each entry corresponds to the presence (1) or absence (0) of the word specified by its index. We then define a dense neural network with the following architecture:

1. dense layer with 32 neurons, ReLU activation, and L_2 regularization and with a coefficient of 0.002;
2. dropout layer with $p = 0.5$;
3. dense layer with 16 neurons, ReLU activation, and L_2 regularization and with a coefficient of 0.002;

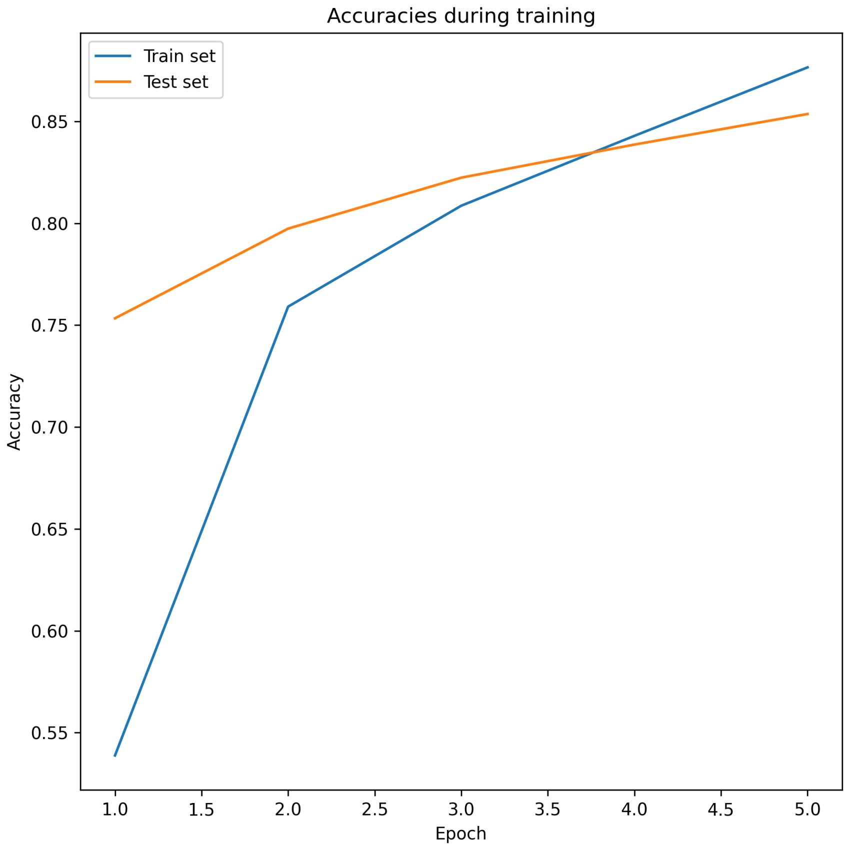

Figure 2: Training and testing accuracies of the model throughout training. Note that the training accuracies above are actually the averages of accuracies on each batch. Since the model improves throughout each epoch, these underestimate the true training accuracy at the end of the epoch. We can compute the true training accuracy after epoch 5 in this run to be 92.96%, showing that the model is, indeed, overfit.

4. dropout layer with $p = 0.5$;

5. dense layer with a single neuron and a sigmoid activation function.

This model is trained for 3 epochs with the Adam optimizer, a binary crossentropy loss function, a batch size of 256, and a learning rate of 0.0002. The accuracy curves are shown in Fig. 3.

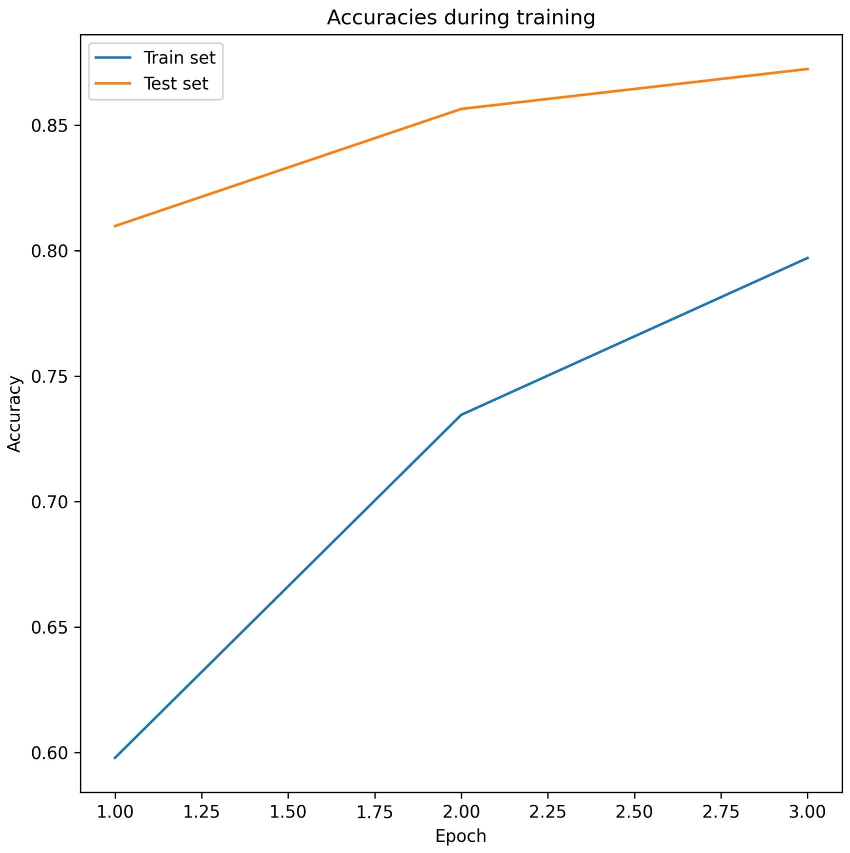

Figure 3: Training and testing accuracies of the fully-connected model throughout training. As before, the particularly low training accuracy is due to the fact that the testing accuracy is computed at the end of the epoch, whereas the training accuracy is computed as the average of the batch accuracies throughout the epoch. As this model improves a lot within a single epoch, the training accuracies shown are much lower than the true training accuracy at the end of the epoch.

4 Results

As the Bayesian network and convolutional models' accuracies were reported previously and they don't meet the assignment requirements, we will focus on the fully-connected model here. The final accuracies are computed to be 89.60% and 87.32% on the training and testing sets respectively. Even with the dropout layers and the simplicity of the model, training this model for more epochs quickly resulted in overfitting, so training for just 3 epochs provided the best combination of accuracy and low overfitting. To better analyze the performance of the model, we plot the confusion matrices in Fig. 4. From these confusion matrices, we see that the model

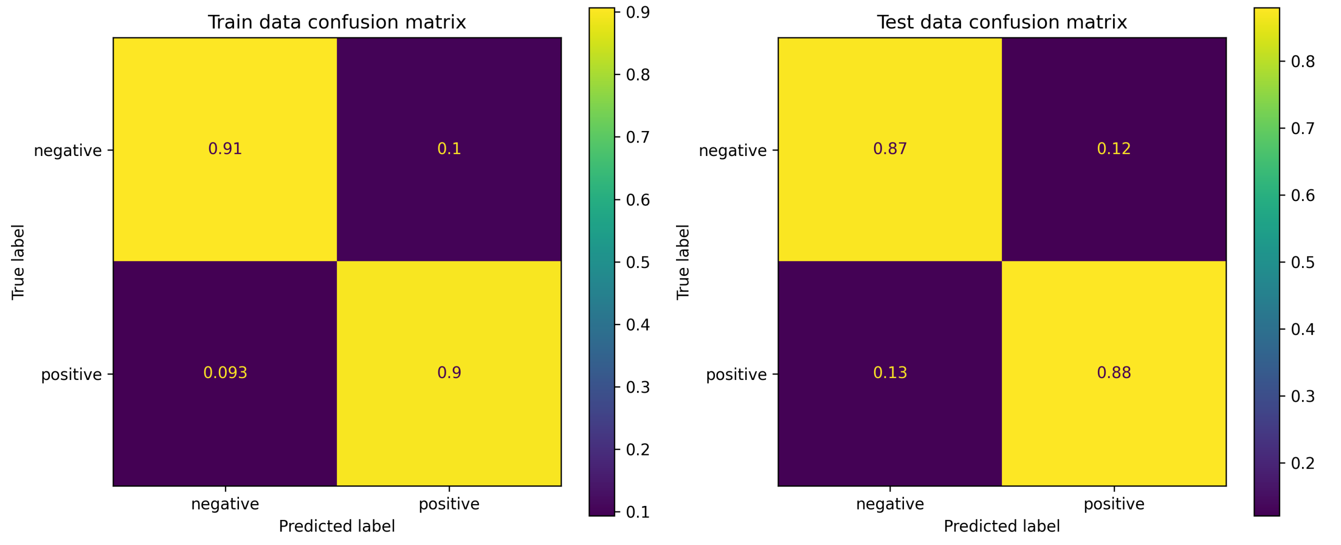

Figure 4: Confusion matrices of the fully-connected model on the training and testing datasets.

correctly and incorrectly classifies negative reviews about as often as positive reviews, so it does not seem to show bias. In general, it does a fairly good job of classifying the sentiment of the movie reviews. We can confirm the effectiveness of the classifier by computing the Matthews correlation coefficient (MCC). This coefficient is insightful for binary classification as it takes into account true and false positives and negatives and is generally regarded as a balanced measure of accuracy. This model achieves MCC scores of 0.799 on the training data and 0.752 on the testing data, which align with the performance previously noted from the accuracies and confusion matrices. In contrast, the Bayesian network achieves training and testing MCC scores of 0.806 and 0.604 respectively, which notes to its high variance. The convolutional model achieves MCC scores of 0.835 and 0.657 on the training and testing sets respectively. Thus, we can see that the fully-connected classifier is a much more well-rounded classifier than the other two.

5 References

[1] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). Learning Word Vectors for Sentiment Analysis. The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).

[2] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. (2014). GloVe: Global Vectors for Word Representation.

6 Note on Running the Code

I wrote the code in Jupyter Notebook rather than Google Colaboratory. This is simply because the data processing runs much faster on my machine than in Colab and I was running into memory constraints with the limited RAM available in the free version of Colab. That being said, the final code should be able to run in Colab. The first cell in the notebook contains code which downloads the required libraries and data files and should be uncommented and run when running the notebook in Colab. I will include the data files in the project submission, so if you are running it locally, you should be able to unzip the file and run the code from the resulting directory directly. The manual data processing for the first two models takes a few minutes to run. To run only the final fully-connected network and bypass the other two models and the data processing, run the cell with the imports at the top of the notebook and then all cells below the *Fully-Connected Model* section title.