CSE440: Natural Language Processing II

Project Report

Project Title: Movie Review Classifier

Group No: 02, Section: 02, Fall 2023	
ID	**Name**
21301648	Raiyan Wasi Siddiky
21301610	Shihab Muhtasim

Introduction

At first glance, NLP can seem daunting. Multidimensional vectors using vector spaces still not entirely understood, datasets with sizes ranging in the millions, rows and rows of numbers that stretch into the horizon, and let's not even get started on the mathematical calculations and machine learning algorithms to make it all work. This project tries to break down and show the individual steps that go into making a fully functional NLP project that deals with the classification of movie reviews and is the first step into the practical side of the world of NLP.
This project is a Movie Review Classifier that deals with sentiment learning by training an algorithm using a dataset of reviews on movies that it then classifies as positive or negative. We use three different models for training-testing; a shallow layer neural network with three layers; an LSTM model and finally a bidirectional LSTM model. The entire project was done using the Google Colab services by Google available for free online.

Instructions (bonus)

As this project was done on google colab, we recommend the code to also be run on google colab, as some cells may not work on local IDEs. If you do still plan on running it on a local IDE, please make sure to comment out the uploader cells and handle file input manually.
The two files we are using are imdb_440.zip - Google Drive for the IMDB dataset file and glove.6B.100d.txt - Google Drive for our GloVe embeddings file.
We have included a way to upload the necessary files from your local device. After uploading, the files will automatically be moved to your google drive. However, if you already have the necessary files in your google drive and you don't need to upload, then you can just input 'yes' and paste the file location. Please note that uploading such large files through colab takes a significant amount of time and we recommend just uploading the files to your drive separately first and then just pasting the file location.

Loading the dataset

This step was done simply by first importing pandas and using the pandas.read_csv function to read the CSV file containing all the movie reviews and their classification as either positive or negative. For confirmation, we then used the .head and .iloc commands to show the data frame and confirm the format. The shape of the dataset came out to be (50000, 2) meaning there were 50000 rows of data, and 2 columns, one for the reviews and one for the classification or sentiment to be used for training. After a deeper checking of the dataset using .describe, we noticed that there were a negligible amount of duplicate values and the 'sentiment' class was equally distributed so there was no need for duplicate removal or downsampling. Additionally, we also noticed there were no None type values in either of our columns so there was no need for removing None type

values either. Finally, we created a new column in our dataset which checked our sentiment column and placed a 1 if the sentiment in the corresponding row was positive and a 0 if negative.

Converting the reviews into vectors

The next step was getting our data frame into a usable format as our computers can't really understand words from the get-go. We started by first tokenizing our data using the Tokenizer module from Keras.preprocessing.text and fitting all our reviews into our tokenizer to get a word index, which is just a dictionary that holds an integer for each unique word in our corpus. We then convert each review into a list of numbers according to the index from the dictionary. At this stage a major problem appears that each review is of a different length and a machine learning algorithm cannot run for variable lengths of data. So we rectify this by using pad_sequences from the keras.preprocessing.sequence module which converts every list representing a word into a fixed length, which it achieves by cutting down for larger lists and padding with 0s for smaller lists. Technically the option to get all data captured here would be to convert every list into the length of the largest review in our data which was 2493, but we chose to use 256 to vastly reduce computational time. Finally, we used a GloVe vector representation from GloVe developed by the Stanford NLP group to create a vector embeddings matrix as it is a pre-trained dictionary that holds correlational data between words, and we compared that to our tokenizer dictionary and only kept the embedding vectors that we needed in our matrix, in other words our embedding matrix only kept the vectors of the words that were present in our dataset.

Splitting the dataset

We simply used the train_test_split from the sklearn.model_selection module to do an 80-20 split of our data; as in 80 percent of the data was to be used for training and 20 percent was to be used for testing.

Training the shallow layer

We started off with a neural network model with 3 layers. The first layer used was an Embedding layer which took inputs of length 256 as previously set and dimensions equal to the size of the vocabulary dictionary, and converted our data into 100-length vectors(the size of our embedding matrix dimensions). We also used our pre-trained dataset, or our embedding matrix as weights and kept trainable=False so our pre-trained weights did not change. There was another layer used to flatten the output from this layer to be able to connect with the dense layers. The second layer was a dense layer of size 10 or rather 10 neurons which used a ReLU(rectified linear unit) function to process our input data. The choice of the ReLU activation function in the second dense layer introduces non-linearity, enabling the neural network to learn complex patterns and representations in the data. Finally, the output layer is a dense layer of size 1 which uses a sigmoid function to

give our predicted output. The sigmoid function in the output layer is chosen for binary classification tasks, as it squashes the output between 0 and 1 which makes it suitable for determining the likelihood of a review being positive or negative. Furthermore, we set our optimizer as the Adam optimizer and our loss calculation as cross-entropy loss to reduce loss through back propagation. We run this neural network on our data for 20 epochs with a batch size of 16 while initializing our weights as the weights from our GloVe vector representation and also using the inbuilt keras validation data argument to get both training and testing accuracies.

LSTM

a.)
To improve over our shallow model, we implemented a Unidirectional LSTM model aimed to leverage sequential dependencies within the movie reviews for improved sentiment analysis. In the first embedding layer, we again utilized pre-trained GloVe word embeddings, which provided a vector representation for each word in the corpus and pretty much kept everything the same. This time however, we replaced the second layer with a single unidirectional LSTM layer with 10 neurons, facilitating the capture of sequential patterns within the reviews. The output layer comprised a similar dense layer with a sigmoid activation function for binary sentiment classification. The model also kept the same batch size as well as the same optimizer and loss functions to maintain a fair test and was similarly trained for 20 epochs with a focus on optimizing binary cross-entropy loss.

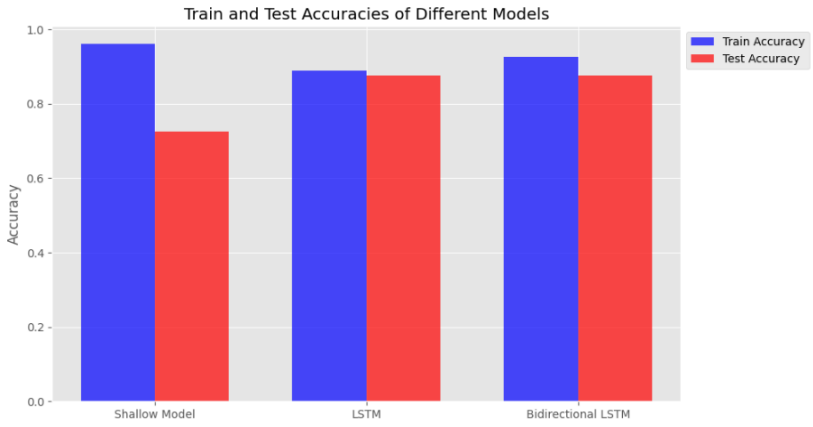
b.)
The Bidirectional LSTM model represents an extension of the Unidirectional LSTM, incorporating bidirectional information flow for a more comprehensive understanding of the context within movie reviews. The Bidirectional LSTM model shares similarities with the Unidirectional LSTM model's structure, meaning we kept the input embedding layer and the output sigmoid layer the same as well as the same batch, optimizer and loss functions, with the only difference being in the second layer where we have implemented a bidirectional LSTM layer with 10 neurons instead, enabling the model to capture dependencies from both past and future contexts.

(Note - after running each model we used clear_session from keras.backend to make sure to keep the test fair for all three models)

Analysis
Performance Analysis of Models: Shallow vs. unidirectional LSTM vs. bidirectional LSTM

For all 3 models, we tested multiple batch sizes and ended up settling on a batch size of 16 which gave us the best combinations of accuracies within a reasonable runtime. The shallow model performs significantly better than expected and provides an astonishing train accuracy of 94.83%

but severely overfits to give an accuracy of 73.19% on the test set with a loss surpassing 100%, showing reasonable room for improvement. The LSTM (Unidirectional) model shows a slightly worse train accuracy of 89.27%, but blows the test accuracy out of the water with an 87.41% on the test set showing little to no overfitting, demonstrating better learning capability and sacrificing a tiny bit of bias for a huge improvement in variance. Finally, the Bidirectional LSTM model achieves an impressive 91.36% accuracy on the training set but shows a slightly lower accuracy of 86.40% on the test set, which we can infer as model complexity leading to some overfitting. One thing that is notable is that the bidirectional LSTM model manages to achieve a very high training and testing accuracy much much quicker than the other 2 models. However, it also needs to be mentioned that the runtimes double for each model, with unidirectional LSTM being twice that of the shallow model, and bidirectional LSTM being twice that of the unidirectional LSTM model. Overall, we can say that either the unidirectional or bidirectional model would be a better choice due to the higher test accuracies.



To make improvements, especially for the shallow and bidirectional LSTM models, we can introduce many regularization techniques such as lasso or ridge regression to mitigate overfitting by introducing a penalty term. We could also add dropout or pooling layers to drop neurons randomly which would also reduce overfitting. We could also use early stopping which would stop the model from running after reaching a flatline in improvement to yet again prevent overfitting. In a different breath, we could also implement attention mechanisms to focus on relevant parts of the input sequence, potentially improving the model's understanding of critical information and even go further and use even better models, such as a transformer which would just be an overall upgrade, or a GRU(Gated Recurrent Unit) which should still display similar performance to the LSTM models but greatly improve runtime due to a more compact infrastructure and optimization of the cellstate layer. Lastly we would like to mention that further experimentation with different hyperparameter configurations could also optimize model performance, for example, changes to learning rates and batch sizes.