Text Sentiment Analysis

Deep Learning Semester Project

Authors:

Nikolaos Kyriakakis

Alexandros Chantzaras

Abstract

In the context of this project, our objective was to utilize different text representations and use them as input to different neural network architectures, in order to evaluate their performance on sentiment analysis problems. Moreover, we wanted to compare the performance of our models when tested on different datasets than those that they had been previously trained on. The main components of this project are four python modules and one notebook file. The notebook file (main.ipynb) acts as a command-and-control file where all objects and function calls take place, as well as plots and results are shown. The rest of the modules contain different class declarations, with those being outlined in more detail below.

Vocabulary

The Vocabulary class is responsible for the mapping of words in our dataset to unique integer values and vice versa. We have added functionality to handle words that have not been seen from the model during the training stage by assigning a special <UNK> token to those. Similarly, should we wish to apply padding to create equal-sized sequences, the <PAD> token can be used.

We have also implemented three methods to achieve the above functionality.

add_token:
This method checks if the given token(i.e., word) is already present in the vocabulary, and should this be true, it immediately returns the corresponding index. Otherwise, the new token is added to a dictionary as a new key, the size of the vocabulary automatically increases, and an integer is assigned to this token. In addition, another dictionary with the reverse functionality is also used in order to be able to reference tokens using their indices.

lookup_index:
This method is used to take an index as input and return the token that this index is associated with. If the given index is not present in the *token_to_index* dictionary, a KeyError exception will be raised.

lookup_token:
This method takes as input a word and returns the respective index. It also checks whether the <UNK> index should be returned if a token is not located.

Vectorizer

The Vectorizer class uses two internal vocabularies, one for the data and another for the labels. A vectorizer object transforms the data into an appropriate representation for each model.

The most important methods of the above classes are the following:

from_dataframe:
This is a factory method used to create and instantiate vectorizer objects. Internally, a threshold value is used to filter only the tokens with at least a predefined frequency of occurrence, while also performing a basic cleaning of punctuation symbols and words.

load_pretrained_embed:
This method takes as input a substantial collection of two million words, represented in vectors of 300 dimensions. It loops over the respective file and searches for matches between our vocabulary and the used file. Finally, a mapping between the existing tokens and their associated pre-trained embeddings is returned.

vectorize:
Abstract method that converts tokens to an appropriate encoded representation, depending on the subclass that inherits and implements it.

Text Dataset

The TextDataset class is used to load and split the data in train, validation, and test parts in an 80-10-10 percent ratio respectively, generate batches, vectorize the data based on the ***vectorizer mode*** that users choose, and finally apply to preprocess in order to ignore symbols, punctuations, and stop words.

generate_batches:
This method loads the data in batches, which are then sent to the GPU or CPU depending on the available underlying hardware.

load_dataset_and_make_vectorizer:
This method is responsible for receiving the contextual dataset, setting the sequence length to the predefined dimension of 359 if the maximum sequence length of a dataset exceeds the number of 1000 words, and finally returning the converted representation of the text by calling the **from_dataframe** method from the vectorizer class.

string_processing:
This method takes as input a string, in our case a single review, and implements the basic preprocessing needed for natural language data such as removing non-word characters and stop words.

Classifier

The Classifier class is used to outline the basic behavior of a model such as to configure the metrics produced through the training, validation, and test stages regarding the classification task we are facing, binary or multi-class. In the case of a binary classification task, the appropriate methods are called to define the loss function that will be used to compute loss, either Binary Cross-Entropy loss or Cross-Entropy loss. Moreover, in the parent **Classifier** class, the **fit**, **eval_net,** and **train_net** methods are used to determine the respective stages of actually training the model. Last but not least, another method called **plot_logs** is used to plot the collected metrics' accuracy, loss, and F1 score.

The child classes BOW Classifier, CNN Classifier, and LSTM Classifier expand the aforementioned class in order to specify their special characteristics regarding their topology and forward method, which mostly depend on the type of the Neural Network model we apply to our data.

Bag Of Words Classifier

This class represents the simplest model that was used for the classification tasks which were evaluated. The data are transformed from their textual representation to a large onehot-encoded matrix, which is then provided as input to a shallow neural network. The latter consists of two linear layers, a dropout layer, and a relu activation function.

CNN Classifier

This class is the implementation of a simple convolutional neural network that utilizes pre-trained embeddings from the fast-text repository. Textual data is mapped to its appropriate vectors using the pre-trained embeddings and is also padded so all sequences comprise of the same length.

In the case of the CNN classifier, the topology of the network comprises three one-dimensional convolution layers, each followed by a relu activation function and a max-pooling layer. Finally, a linear layer and a dropout layer have been placed to perform the classification task.

LSTM Classifier

Similarly, as in the case above, we use the fast-text pre-trained embeddings and padded sequences as input to the LSTM model, and afterward, the result was provided to two linear layers with a relu activation function in between.

Results

Bag of Words Classifier

Each classifier was trained in the Tweets dataset for five epochs. Bag Of Words model took as input the data in one-hot-encoding format. The test results in the Tweets dataset were 79.14% accuracy score, 79.00% f1 – score, and loss of 0.45 whereas in the IMDB dataset when the model has been trained the on Tweets dataset, the accuracy score is 51.09%, 39.94% f1 – score, and 5.51 loss. The following plots illustrate the training and validation logs for the Bag of Words model.

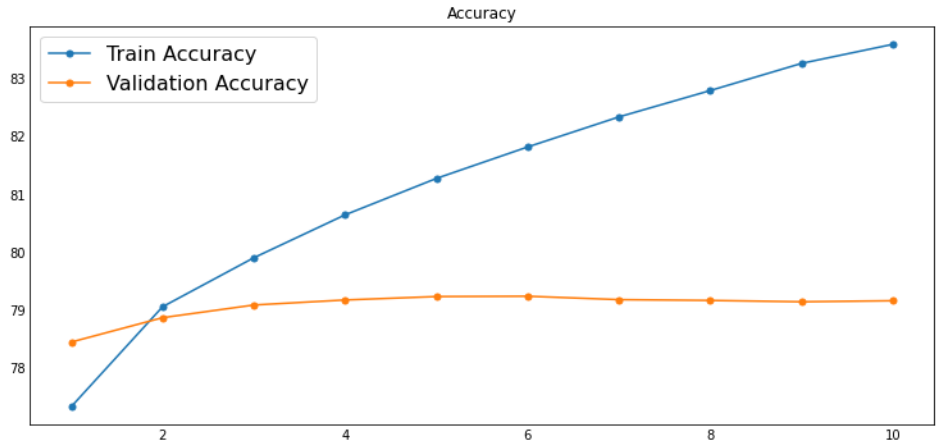

Figure 1: The training and validation accuracy of the bow classifier

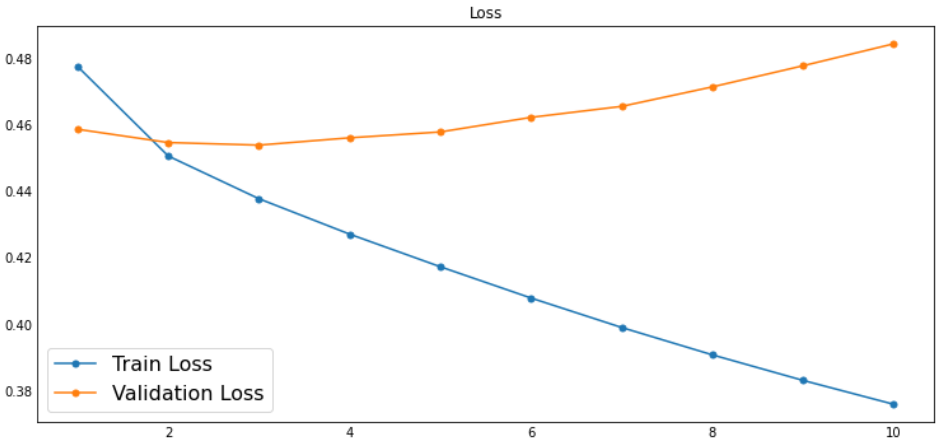

Figure 2: The training and validation loss for the bow classifier over 10 epochs

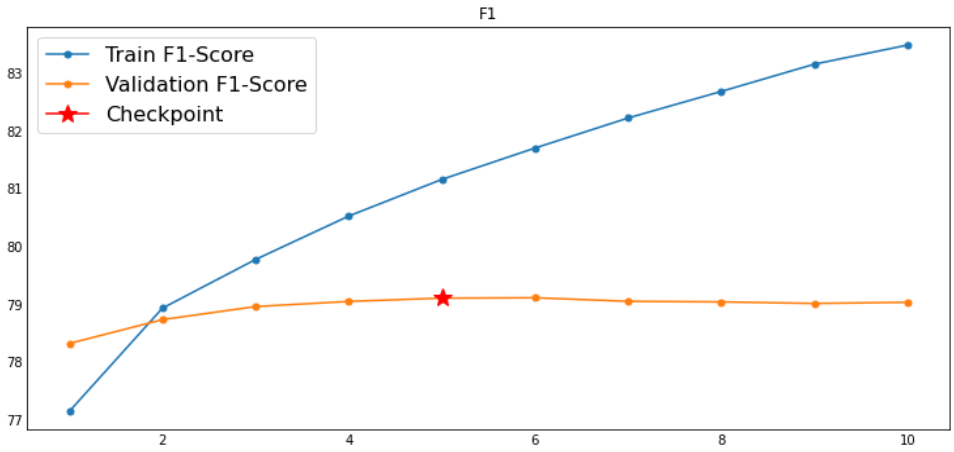

Figure 3: The training and validation F1 scores for the bow classifier over 10 epochs

CNN Classifier

The CNN classifier received data in padded sequences, and we also used pre-trained embeddings to represent our vocabulary. In this case, the results for the Tweets dataset were 79.08% accuracy score, 78.94% f1 – score, and loss of 0.46 whereas in the IMDB dataset 63.39% accuracy score, 61.08% f1 – score, and 0.63 loss. The following plots illustrate the training and validation logs for the CNN model.

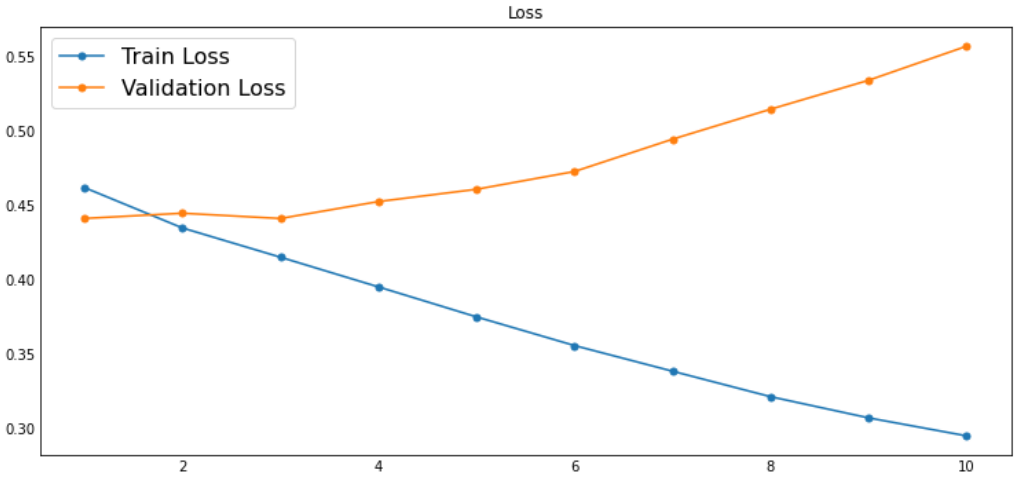

Figure 4: Training and validation loss for the CNN classifier over 10 epochs

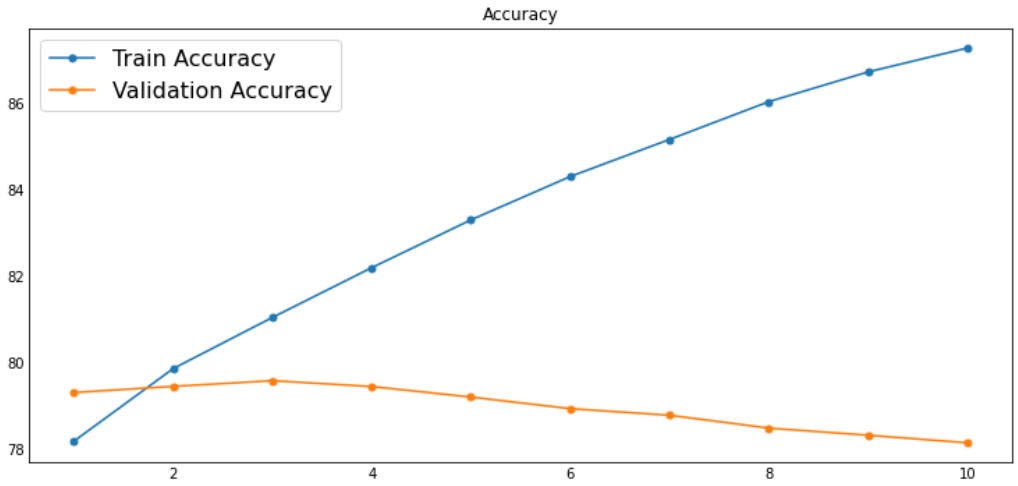

Figure 5: Training and validation accuracy for the CNN classifier over 10 epochs

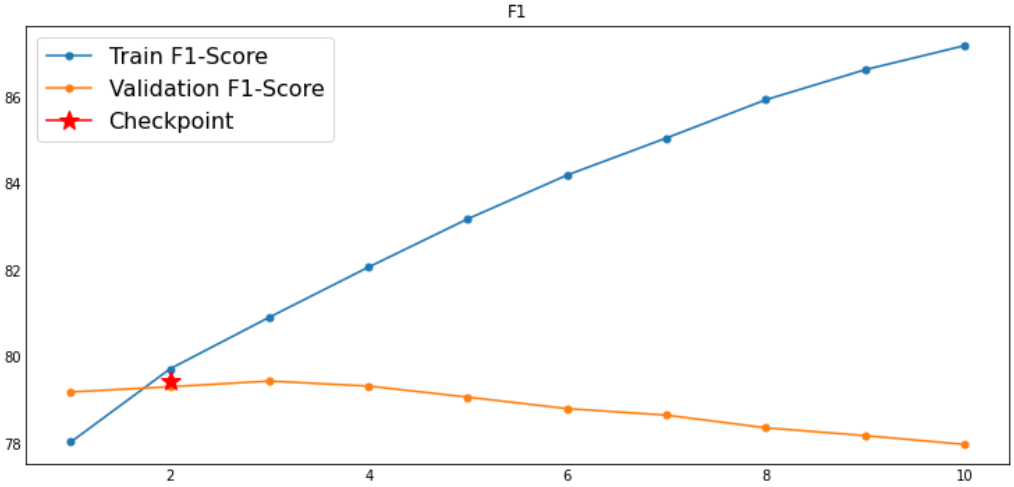

Figure 6: Training and validation F1 scores for the CNN classifier over 10 epochs

LSTM Classifier

The data in the LSTM classifier were fed in the same way as in the CNN model. The scores in the Tweets dataset were 78.08% accuracy score, 78.73% f1 – score, and a loss of 0.44. The scores in the IMDB dataset were 61.01% accuracy score, 60.34% f1 – score, and loss of 0.65 respectively.

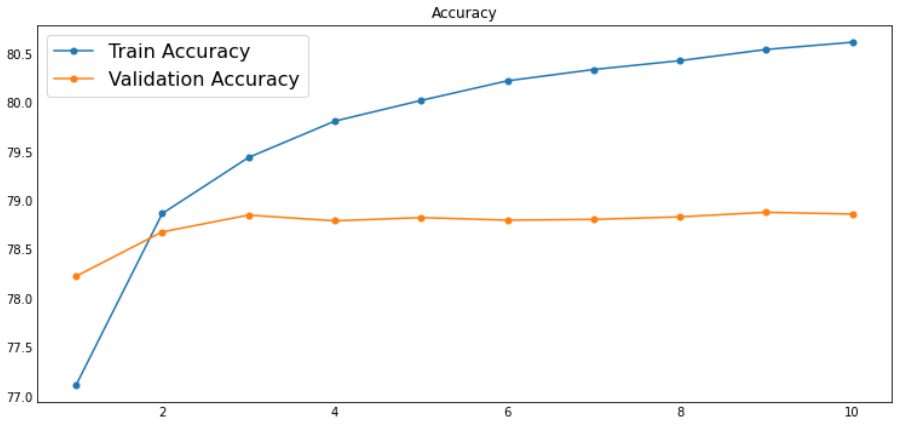

Figure 7: Training and validation accuracy for the LSTM classifier

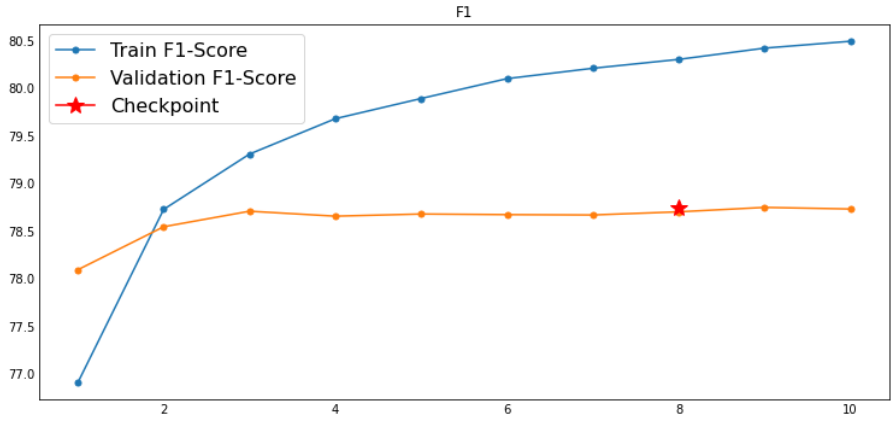

Figure 8: Training and validation F1 scores for the LSTM classifier

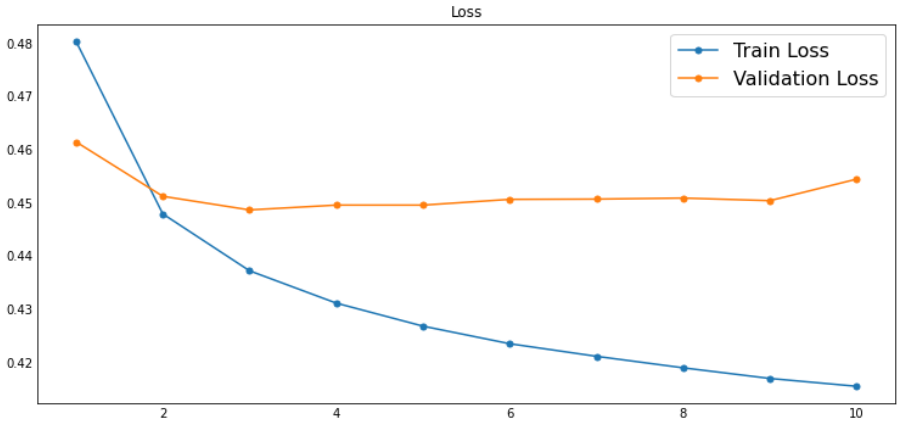

Figure 9: Training and validation loss for the LSTM classifier

Judging from the above, we can easily spot a significant decrease in performance in all networks when tested on a new dataset. However, comparing CNN with the simpler first neural network, we observe a vast difference in favor of the former. The same can be said about the LSTM, however, we should also point out that both CNN and LSTM networks use the **fastText** pre-trained embeddings while also being more complex by design. Therefore, they are able to capture patterns more efficiently when tested with not previously seen data.

Future Work

As an extension of this work, it would be interesting to perform training on a review-oriented dataset such as the ones used already and evaluate the trained models on a textual dataset that refers to a totally different topic.

Conclusion

Given the results of our research, more complex models such as CNN and LSTM achieve better generalization when tested on different datasets than the ones that were trained. We have to mention though that these models exploited large-scale pre-trained embeddings which obviously boosted their performance.