# ML1-Sentiment-analysis-and-emotion-detection

This repository contains a text classification model trains a Recurrent Neural Network (RNN) on the Yelp Polarity Reviews dataset for sentiment analysis. It is a project under the IITISoc programme, IIT Indore.

Created by Zhang et al. in 2015, the Yelp Polarity Reviews Dataset contains 1,569,264 samples from the Yelp Dataset Challenge 2015. This subset has 280,000 training samples and 19,000 test samples in each polarity (positive and negative reviews). Dataset from FastAI's website., in the English language. Containing 1,569,264 in CSV file format.

The repository contains two files:

1. Sentiment\_Analysis(Yelp\_Polarity\_reviews).ipynb: the initial model(Kindly, refer to this first)
2. (Robust)Sentiment Analysis(Yelp-Polarity reviews).ipynb-the model with overfitting overcome by using more dropout layers and callback function.
3. Explanation-Sentiment Analysis.doc-An explanation of the various libraries and functions used to create, train and test the model.

Salient Features of the initial model (Sentiment\_Analysis (Yelp\_Polarity\_reviews).ipynb):

* The raw text loaded by tfds needs to be processed before it can be used in a model. The simplest way to process text for training is using the experimental.preprocessing.TextVectorization layer. This layer will standardize, tokenize, and vectorize the data. Standardization refers to pre-processing the text, typically to remove punctuation or HTML elements to simplify the dataset. Tokenization refers to splitting strings into tokens (for example, splitting a sentence into individual words by splitting on whitespace). Vectorization refers to converting tokens into numbers so they can be fed into a neural network.
* The .prefetch() function is applied to overlap data pre-processing and model execution while training.

- The layers are stacked sequentially to build the classifier:

1. This model is built as a tf.keras.Sequential model.
2. The first layer is the encoder, which converts the text to a sequence of token indices.

After the encoder, there is an embedding layer. The Embedding layer takes the integer-encoded vocabulary and looks up the embedding vector for each word-index. When called, it converts the sequences of word indices to sequences of vectors. These vectors are trainable. After training (on enough data), words with similar meanings often have similar vectors. This index-lookup is much more efficient than the equivalent operation of passing a one-hot encoded vector through a tf.keras.layers.Dense layer.

1. Then come the RNN layers.

A recurrent neural network (RNN) processes sequence input by iterating through the elements. RNNs pass the outputs from one timestep to their input on the next timestep.

The tf.keras.layers.Bidirectional wrapper is used with two RNN layers. These layers propagate the input forward and backwards through the RNN and then concatenate the final output.

The first layer has the return\_sequences constructor argument that is set to true to return full sequences of successive outputs for each timestep (a 3D tensor of shape (batch size, timesteps, output features)).

1. Further there are two dense layers:
2. The third last dense layer consists of 64 neuron units with a rectified linear unit (relu) activation.
3. The second last output layer is a dropout layer that randomly drops out some fraction(50%) of a layer's input units every step of training, making it much harder for the network to learn those spurious patterns in the training data. Instead, it has to search for broad, general patterns, whose weight patterns tend to be more robust thereby countering overfitting.
4. The last layer is densely connected with a single output node or neuron as we the model determine a single output, i.e., either positive or negative review.

After the RNN has converted the sequence to a single vector, the two layers.Dense do some final processing, and convert from this vector representation to a single logit as the classification output. We define custom functions to calculate the metrics-precision, recall and the F1 Score. We call these functions from within model.compile() in the the next code cell.

* Since this is a binary classification problem and the model outputs single modified value (positive or negative), we use losses.BinaryCrossentropy loss function and adam as optimizer.
* We then train the model for 10 epochs with a batch size of 128 at a time and 30 validation steps(batches) after each epoch.

After training the model, the following results are obtained:

loss: 0.1396, accuracy: 0.9422, f1\_m: 0.9407, precision\_m: 0.9613, recall\_m: 0.9215, val\_loss: 0.1554, val\_accuracy: 0.9382, val\_f1\_m: 0.9383, val\_precision\_m: 0.9518, val\_recall\_m: 0.9256

- Testing the model, on the test data we obtain the following results:

Test Loss: 0.14759980142116547

Test Accuracy: 0.9415000081062317

Test Precision: 0.9503315091133118

Test Recall: 0.9313362836837769

Test F1 Score: 0.9405220746994019

We observe that in the later epochs the validation loss and loss are far apart and the validation loss shows sign of levelling off or a slight increase. This indicates the there has been overfitting in the model. We have tried to overcome that in the second file.

* -Finally, we have saved the model under Keras to access it in future.

The second file contains the more robust model. All the features of this model are same as the initial model except for the following changes:

(a)Citing the overfitting in the initial model, we added an earlystopping call back function (paticence=5 epochs and min\_delta=0.01), whose objective was to check overfitting.

As soon as the validation loss starts increasing or decreases by a value less than 0.001 per epoch, this function is called and it stops the training there, restoring the best weights.

(b)Also to prevent the model from deviating by learning noisy interpretations or spurious data, we added another drop out layer after the 2nd RNN layer.

The following are the results obtained:

After training:

loss: 0.1766 - accuracy: 0.9269 - f1\_m: 0.9248 - precision\_m: 0.9482 - recall\_m: 0.9033 - val\_loss: 0.1791 - val\_accuracy: 0.9264 - val\_f1\_m: 0.9268 - val\_precision\_m: 0.9397 - val\_recall\_m: 0.9147

After Testing:

Test Loss: 0.17096687853336334

Test Accuracy: 0.93039470911026

Test Precision: 0.9406332969665527

Test Recall: 0.9187859892845154

Test F1 Score: 0.929294764995575

The trends in loss and accuracy can be referred to the graphical plots in the colab file.

In spite of the slight decrease in accuracy, this model is more robust than the initial model. It allows better generalization for the problem and hence, ensures better accuracy in predictions over unseen data.