The Analytics Edge Kaggle Competition (Team 16)

Members: Lucas Ng (1003478) Evan Sidhi (1003691) Tan Yin Ling (1003891)

Approach

We identified the competition problem to be a tweet sentimental analysis classification problem.

Our initial approach was to use classification models that we have learnt during class. However, in order to use these models, natural language processing (NLP) features must be engineered for the model to use as training data. We started off by creating a cleaned document-text matrix (DTM) using the text mining package *tm*. This was done by

- transforming text to be raw (changing all to lower case, remove punctuations and URLs),
- removing stop-words and
- identifying each word's stem

Further pre-processing was done before we ran the Random Forest model. For instance, we only included terms with a minimum of 20 occurrences in the DTM and removed some sparse terms from the training set. Using a training-test split ratio of 0.7, we ran the Random Forest model.

The best accuracy that we obtained via this model, however, was only 86.6% on the test set. While finetuning the model by changing the sparsity of words removed improved the results of the validation set, it did not improve results on the test set.

Multilayer perceptron

Next, we embarked on using a multilayer perceptron to improve the quality of our accuracy. Neural Networks are sets of algorithms, designed to recognise patterns. They can recognize numerical patterns that are contained in vectors. Despite its seemingly limited input type, real-world data such as images, sound, text or time series can all be converted into these numerical patterns. Artificial neural networks are composed of a large number of highly interconnected processing elements (neurons) working together to solve a problem. These processors usually operate in parallel and are arranged in layers. The first layer would receive raw input information and pass the output as an input for the succeeding layer. Finally, the last layer would produce the output of the system.

Similar to the feature engineering done above, our approach began by vectorising the word set given in the training data. We began by tokenising the words using *tokenizers*, stripping away URLs, punctuations and ensuring all words are lowercased.

Next, we create our own 'vocabulary' through fitting the text vectorization layer to our tokened tweets by adapting it. When this layer is adapted, it will analyse the dataset, determine the frequency of individual string values, and create a 'vocabulary' from them. This vocabulary can have unlimited size or be capped, depending on the configuration options for this layer; if there are more unique values in the input than the maximum vocabulary size, the most frequent terms will be used to create the vocabulary.

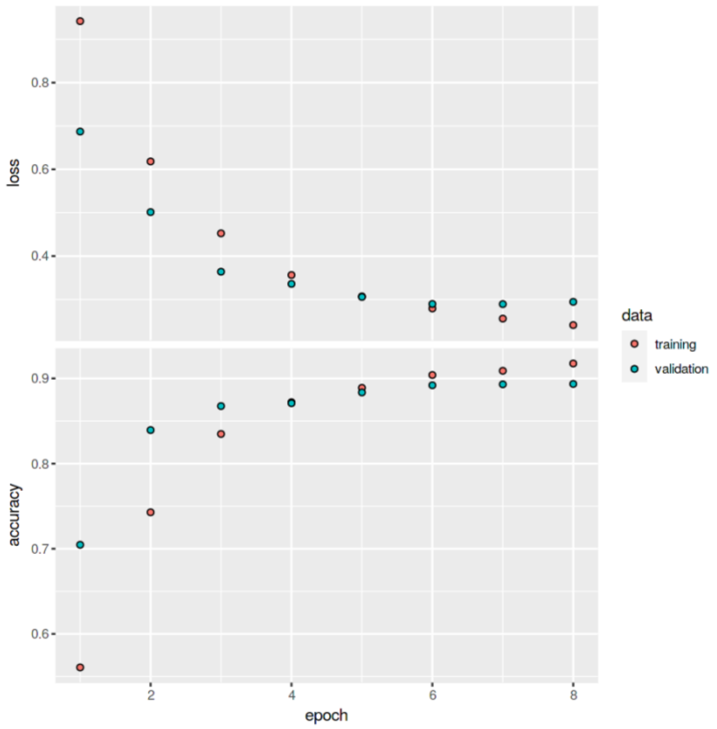
Next, we create the training, validation and test set with a 60%, 20%, 20% split on the given train data respectively.

We constructed the network architecture in the following way:

- Text vectorisation: we first vectorized the tokenized tweets using the same text vectorization layer as above and input them into the embedding layer.
- Embedding layer: This layer is especially useful to help us capture some of the semantics of the input by placing semantically similar inputs close together in the embedding space. It reduces the dimensionality of the categorial variables and represent these categories more meaningfully in the transformed space.
- Global average pooling layer: since we have a new representation of the input, a global average pooling layer is added to get the average value of these representations of the tokens for each of these representations.
- Tanh activation layer: these are then fed into a dense layer of 64 neurons with 'tanh' activation.
- Dropout layer: the dropout layer helps to prevent overfitting especially for our small sized dataset by randomly dropping out a number of output features of layers during training. For this layer, we used a dropout rate of 0.80.
- Output layer: Our output layer consisted of 3 neurons with 'softmax' activation. Softmax assigns the maximal element largest portion of the distribution with other smaller elements getting smaller part of the distribution which works well when we are getting the probability for different classes in the output.
- The model is then complied with the most common and best working optimizer for most 'adam', and the sparse categorical cross as the loss function.

After creating our desired model, we fit and validate with the validation data. The parameters we finetune here are the number of epochs and the batch size. Choosing too many epochs will result in overfitting of the data, resulting in higher validation loss. Larger batch sizes result in faster progress in training, but don't always converge as fast. Smaller batch sizes train slower, but can converge faster. Hence we added a call-back function here which would monitor the validation loss and stop the model from training on further data once the validation loss starts to increase upon further epochs.

After fitting the model, we plot the results of the accuracy (on the validation data) and the validation loss. This allows us to determine the appropriate number of epochs before the model starts to overfit.



Finally, we test the model's prediction on the test dataset and calculate the accuracy by taking the sum of the true positives with the total predicted values. This gave us a value of 88.5%

Results & Analysis

The model gave us a final accuracy of 89.86%. Given that a dummy model (that randomly classifies tweets) would have achieved a baseline accuracy of 33.3%, an accuracy of 89.86% gives us statistical power of about 56.53%. Hence, this is significant for our problem at hand.

Limitations & further work

Our model has little data to work with. Due to the time constraint, we didn’t have the time to explore further feature engineering. Instead, we were fine tuning the model (such as changing network architecture or optimiser) in order to achieve higher accuracy on our validation set. Unbeknownst to us, by finetuning, we were leaking information from the validation data into our model, thereby overfitting our dataset to the validation set (and eventually our own test set).

Considering that many tweets have key features (e.g. hashtags, @mentions), we could have engineered more features to allow the model to have a higher accuracy. Emojis were also present that could serve as datapoints that were telling of the sentiment of the tweet. We could have also modified the model of have a smoother learning rate that could give us the best optimal set of weights.

Conclusion

The data competition taught us much about the importance of building and fine tuning our models. The Kaggle platform provided for friendly competition that fostered excellence and pushed us to best each other.

References

Chollet, F. (2018). *Deep learning with Python*. Shelter Island, NY: Manning Publications.