Artificial Intelligence II Homework 3

Comments & Model Performance results on Questions 1 & 2

Pavlos Spanoudakis (sdi1800184)

Basic Execution flow

1. Before creating the train & validation sets, we read the `.csv` files the usual way (with `pd.read_csv`), to check if all the samples have the expected format without missing values, and get the number of samples in the validation set (which will be needed later).

2. We create the Train and Validation sets, using `TabularDataset` objects:

- We preprocess each tweet with `customPreprocessor` and tokenize it using `nltk.TweetVectorizer`.
- We will access the two sets in batches, using `BucketIterator` objects. We initialize the validation set `BucketIterator` with `batch_size` equal to the validation set size, so essentially we access the set in "batch mode" (one batch for the whole set).

3. We create the embeddings matrix, where we store the **GloVe** vector for each word in the train set vocabulary.

4. We initialize the Model. Hyperparameters such as number of epochs, batch size, learning rate, number & size of layers, Dropout probability etc. can be modified on code cell #5.
   We use `CrossEntropyLoss` and the `Adam` optimizer.

5. We train the model: We use `numpy` arrays to store several performance stats during training, such as Loss and F1 score on Train and Validation set after each epoch.
   During each epoch:

   - For each batch given by the train set `BucketIterator`:
     - We make predictions on this batch
     - Extract the predicted labels & calculate the accuracy
     - Calculate & store the batch Loss
     - Perform backpropagation
   - After going through all the batches, we calculate the total Loss and the F1 score for the Train set
   - We make predictions on the Validation set
   - Calculate & store the Validation set Loss
   - We extract the predicted labels, calculate the accuracy and store the F1 score

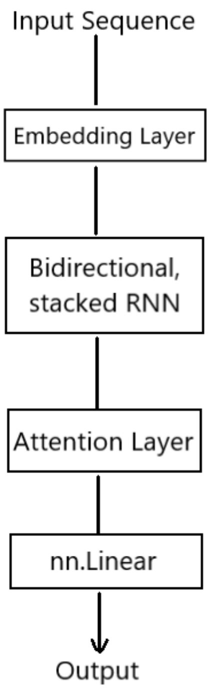
6. Displaying performance results: After the end of training we display:

   - The Confusion Matrices of the final model predictions on the Train and Validation sets.
   - The F1 Learning Curves for both sets, to demonstrate the performance of the model after each epoch of the training phase.
   - The F1, Precision and Recall scores for the final model on both sets.
     We use the corresponding **scikit-learn** routines for calculating the scores. We use `average='micro'` for F1 score and `average='macro'` for Precision & Recall scores.

  - Training phase Loss curves for both sets.
  - The ROC Curves for the Validation set predictions.
    We mirror the usage of `roc_curve` from **scikit-learn** for multiple classes, as demonstrated here.
    To create the curves, we apply the `softmax` function to the NN output vector, to convert it to possibility values that add-up to 1, and use the `softmax` output to create the curves.
    `roc_curve` applies generated possibility thresholds to create the curves, therefore if we provided it with just the predicted labels, it would only apply 3 thresholds to each result, which is insufficient to create useful ROC curves.

Model types used during development

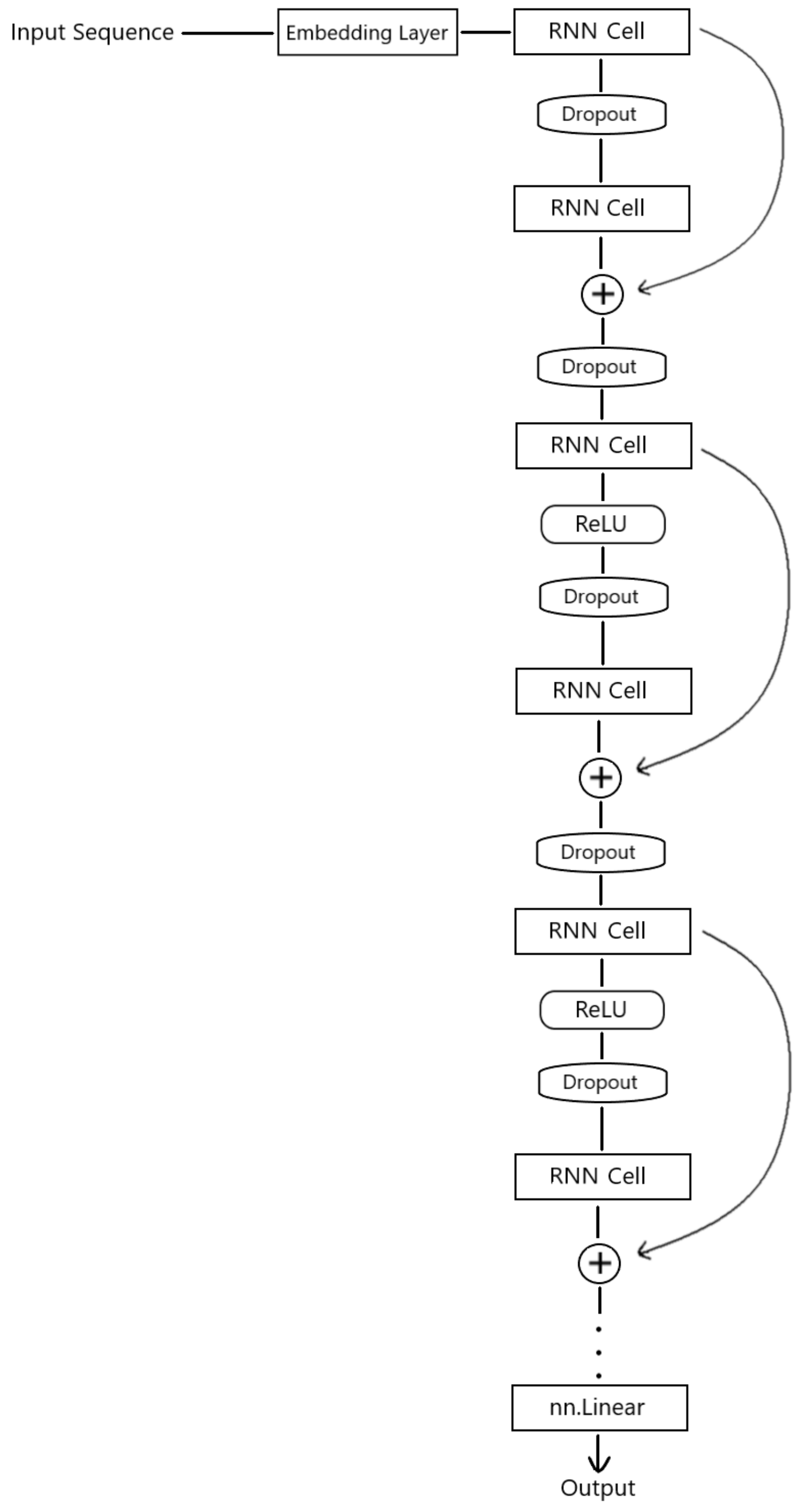
- Models 1, 2 have the following architecture:



  (Model 2 does not use attention layer, but other than that the structure is the same.
  Model 1 is the chosen model, and can be used with or without attention layer. See the next sections for the performance comaprison.)

  - We use bidirectional, stacked RNN, without skip connections.
  - If desired (optionally in Model 1), we can pass the RNN output through an Attention Layer, to help the model focus in critical parts of each tweet and prevent the vanishing gradient in faraway hidden states.
  - Finally, we add a Linear layer to output a vector of the desired size.

- Models 3, 4 use the following architecture:



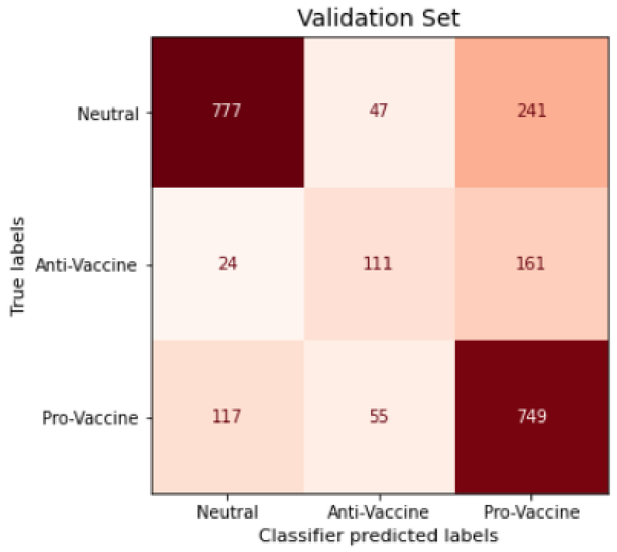
  - We use a stacked RNN, this time with skip connections. We apply a skip connection in every 2 layers (layers 1, 3, 5 etc. have a skip connection, and the output of layers 2, 4, 6 etc. is the destination of a skip connection). Skip connections are implemented simply using `torch.add`.
  - Before providing input to any layer (except for the first one) we apply a Dropout layer to it.
  - We apply a `ReLU` layer to the odd layer outputs, except for the first one.
  - Finally, we add a Linear layer to output a vector of the desired size.

Different models performance comparison

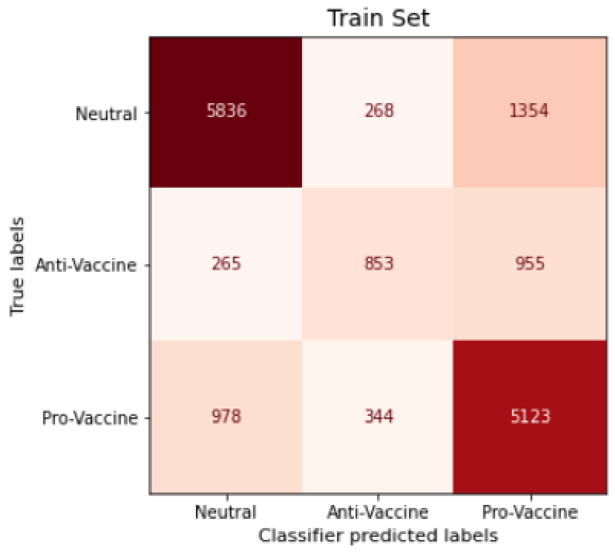
Notes on all models:

- The performance results displayed below have been produced without using GPU acceleration.
- When not using GPU, the results can be reproduced using `SEED = 42`.
- Unfortunately, GPU-accelerated model results could not be fully reproduced.
- The execution of each model using only CPU takes about 4-5 minutes. In case this is inconvenient, GPU can be enabled in code cell #6, but note that the results for all models will differ from the ones presented below.
- All models use the **GloVe** pre-trained word embeddings from `glove.6B.100d.txt`.

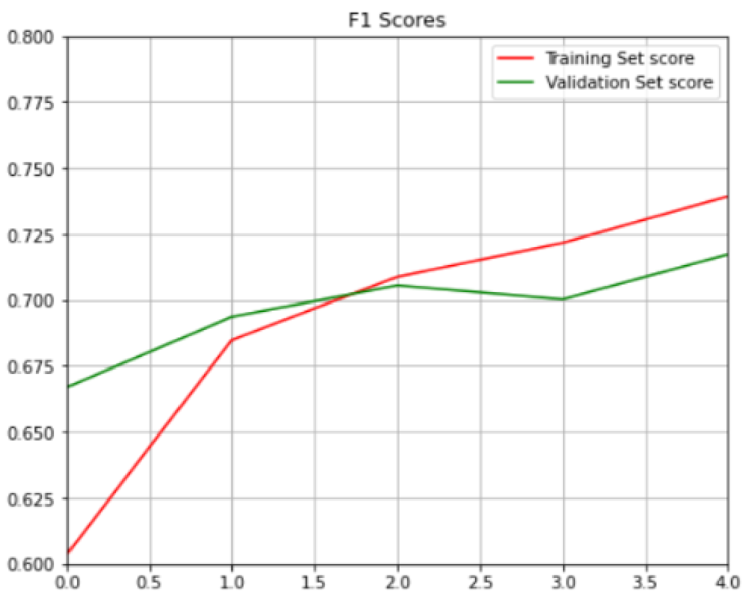
1. This is the preselected model in the interactive notebook. For this model, we use:
    - Learning rate: 0.005
    - Batch Size: 256
    - # Epochs: 5
    - RNN: Bidirectional, 4-layer **GRU**, with `hidden_size = 32`
    - Dropout probability: 0.5



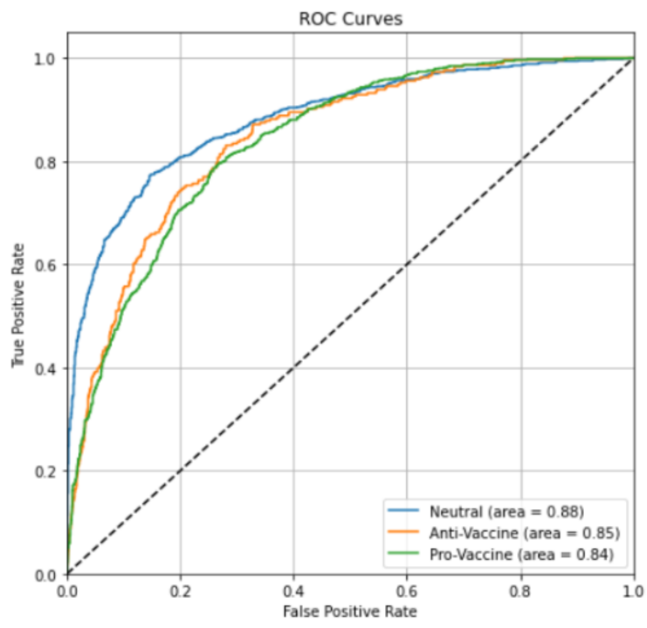
	Neutral	Anti-Vaccine	Pro-Vaccine
Neutral	777	47	241
Anti-Vaccine	24	111	161
Pro-Vaccine	117	55	749

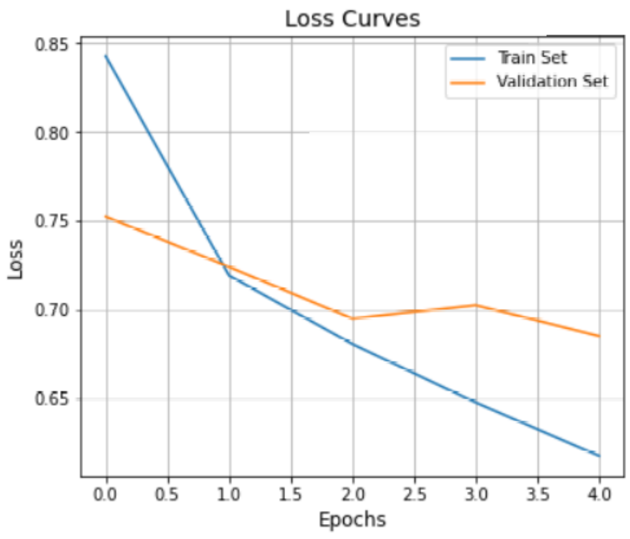


	Neutral	Anti-Vaccine	Pro-Vaccine
Neutral	5836	268	1354
Anti-Vaccine	265	853	955
Pro-Vaccine	978	344	5123



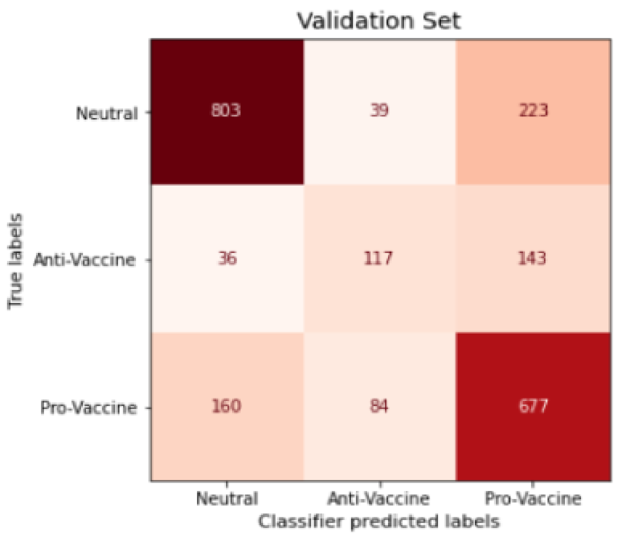
	F1	Precision	Recall
Train	0.7394	0.6987	0.6630
Validation	0.7174	0.6728	0.6393

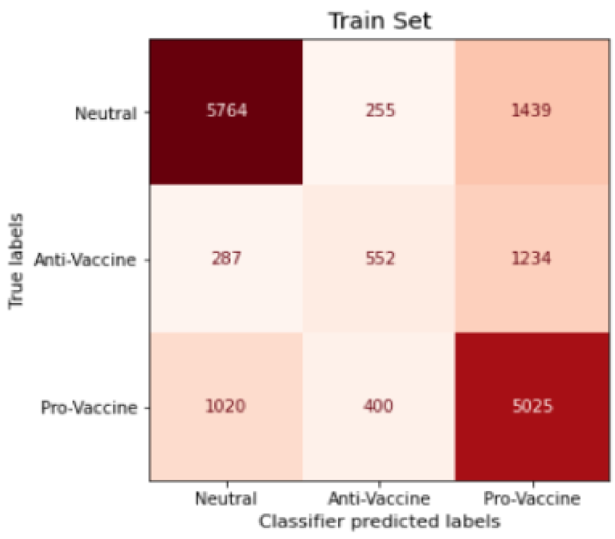


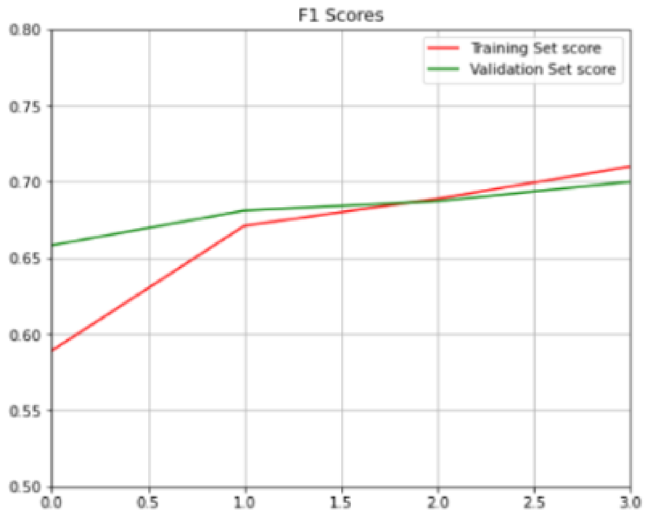


2. For this model, we use:

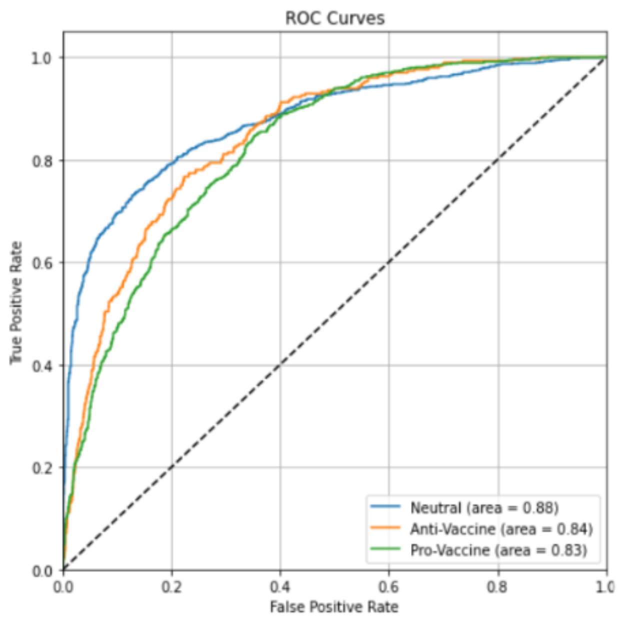
  - Learning rate: 0.005
  - Batch Size: 128
  - # Epochs: 4
  - RNN: Bidirectional, 4-layer **LSTM**, with `hidden_size = 32`
  - Dropout probability: 0.5

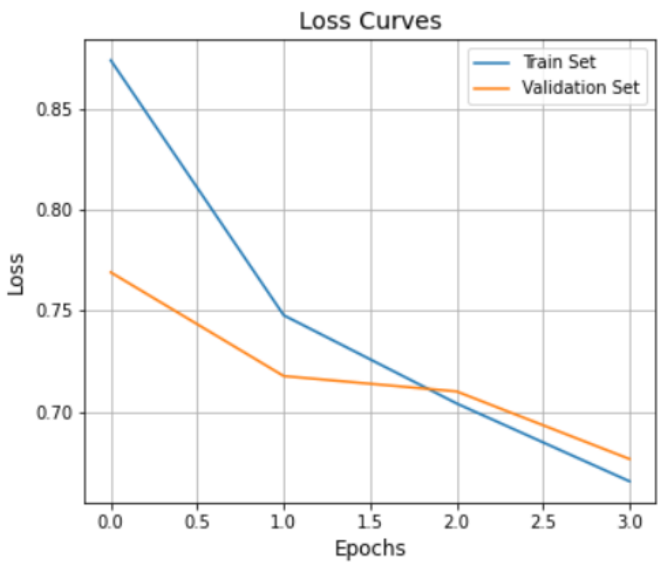






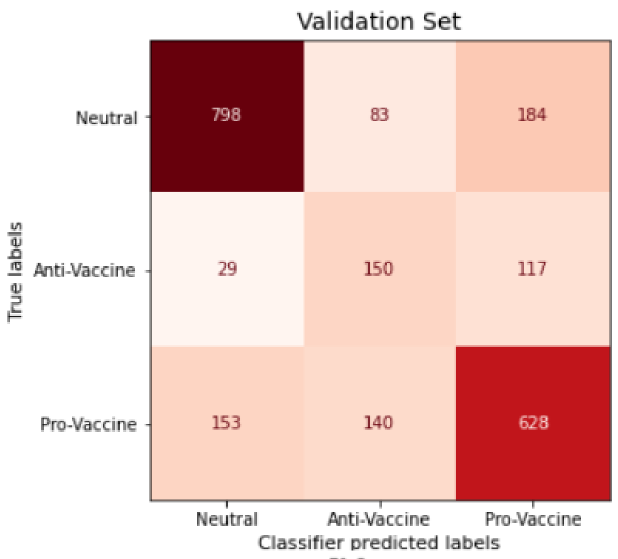
	F1	Precision	Recall
Train	0.7099	0.6418	0.6063
Validation	0.6998	0.6468	0.6281

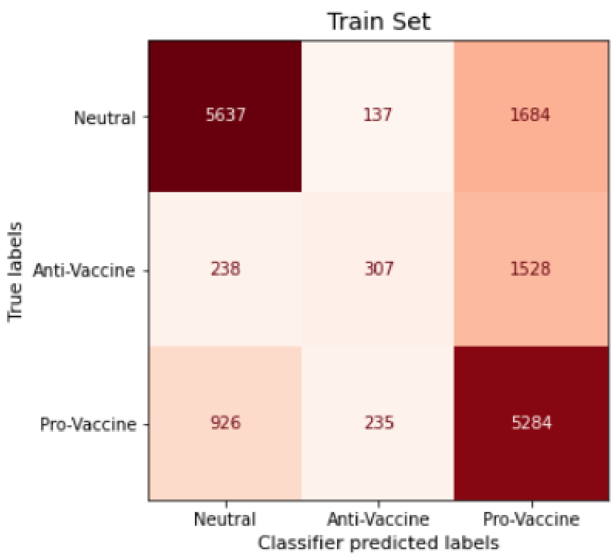


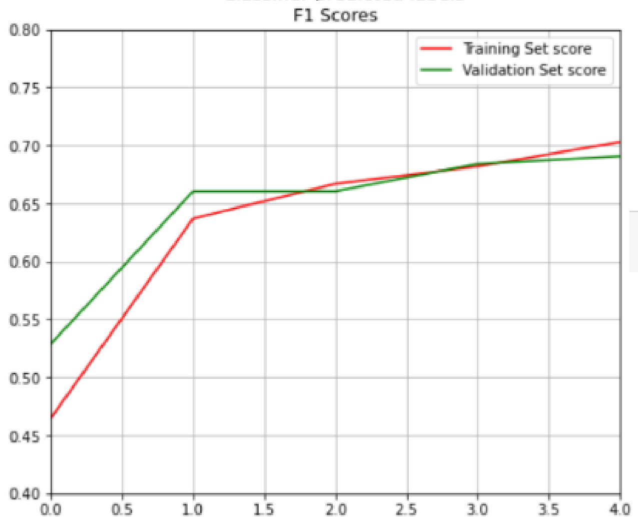


3. In this model, we use:

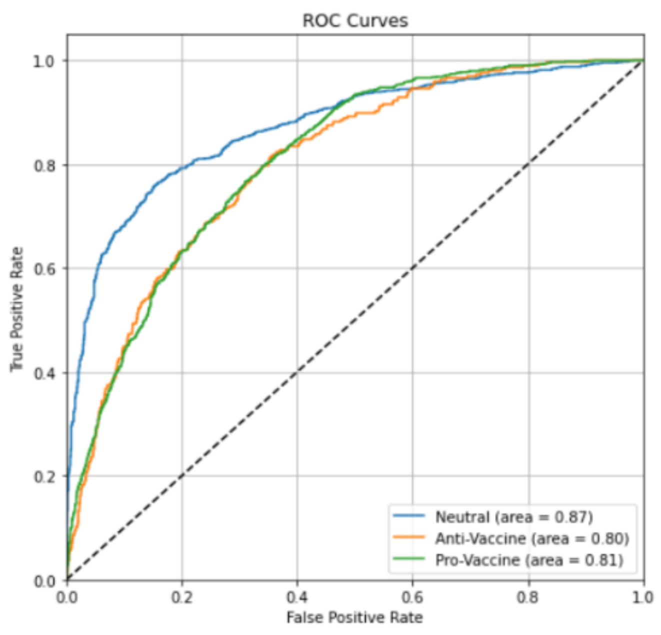
- Learning rate: 0.005
- Batch Size: 128
- # Epochs: 5
- RNN: 6-layer **GRU**, with `hidden_size = 64` and skip connections
- Dropout probability: 0.5

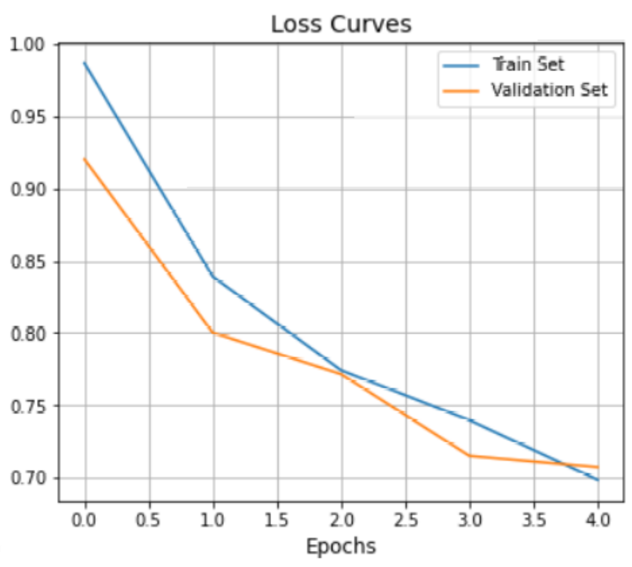






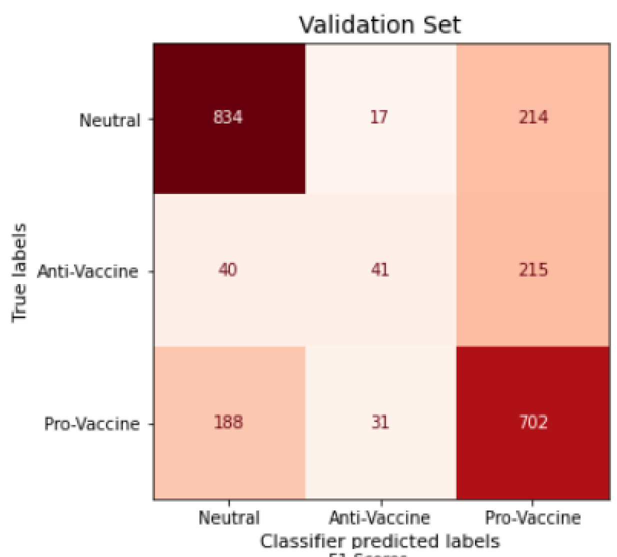
	F1	Precision	Recall
Train	0.7028	0.6343	0.5746
Validation	0.6906	0.6308	0.6460

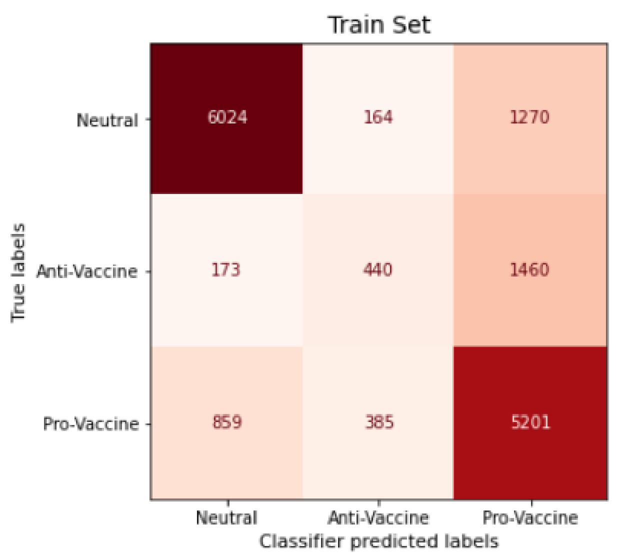


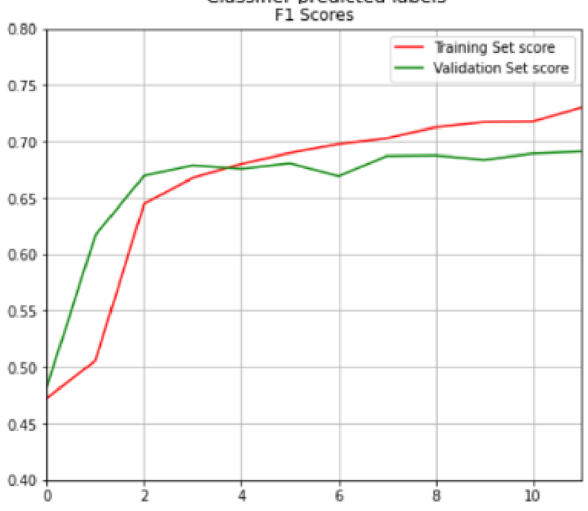


4. In the last model, we use:

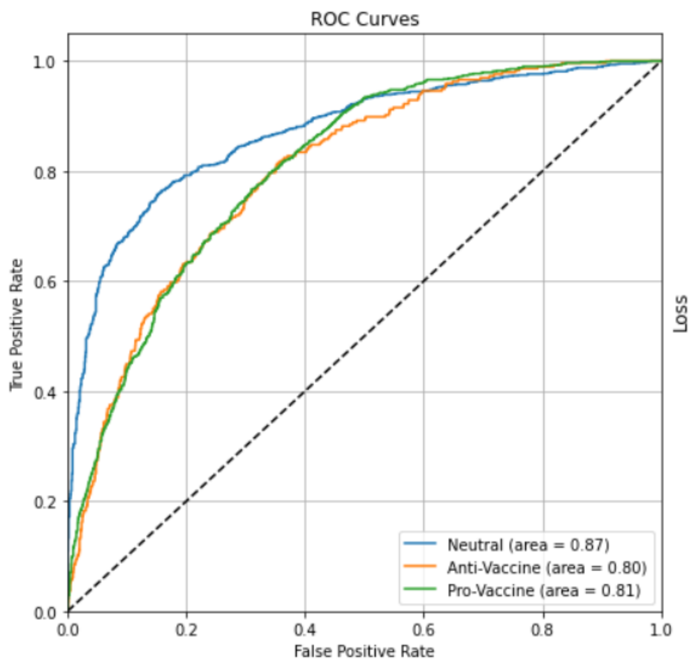
- Learning rate: 0.0055
- Batch Size: 256
- # Epochs: 12
- RNN: 3-layer **LSTM**, with `hidden_size = 64` and skip connections
- Dropout probability: 0.5

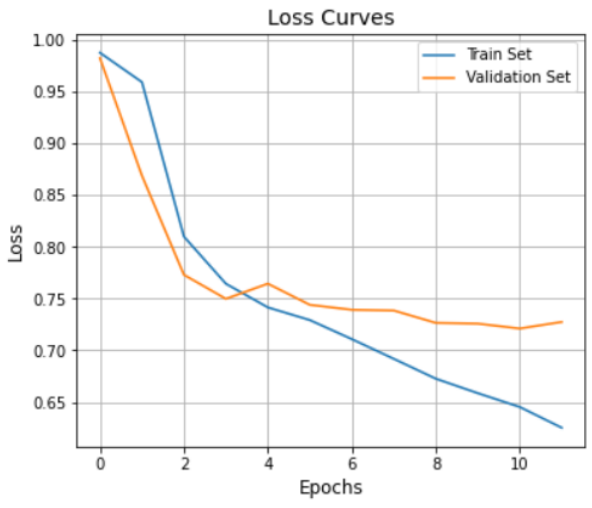






	F1	Precision	Recall
Train	0.7302	0.6515	0.6090
Validation	0.6911	0.6222	0.5613



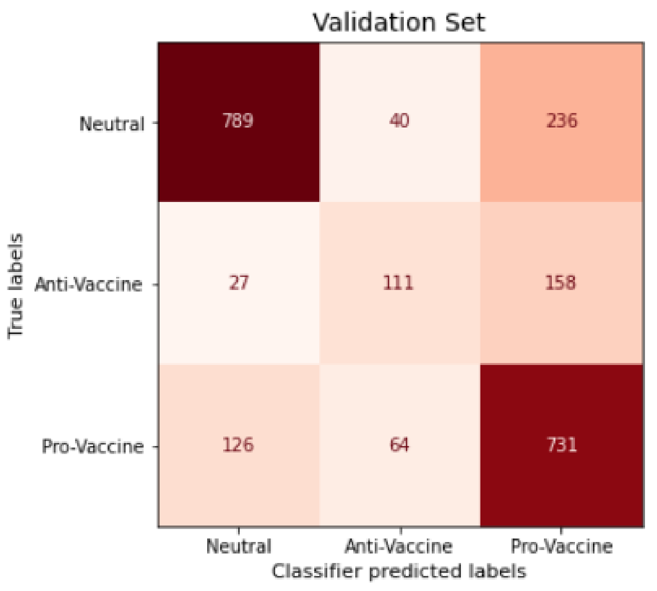


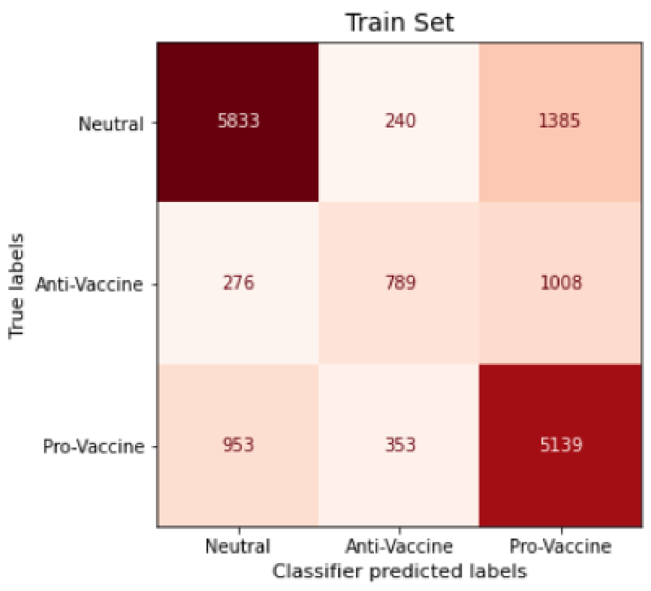
Comments/Observations on the models and their develpoment

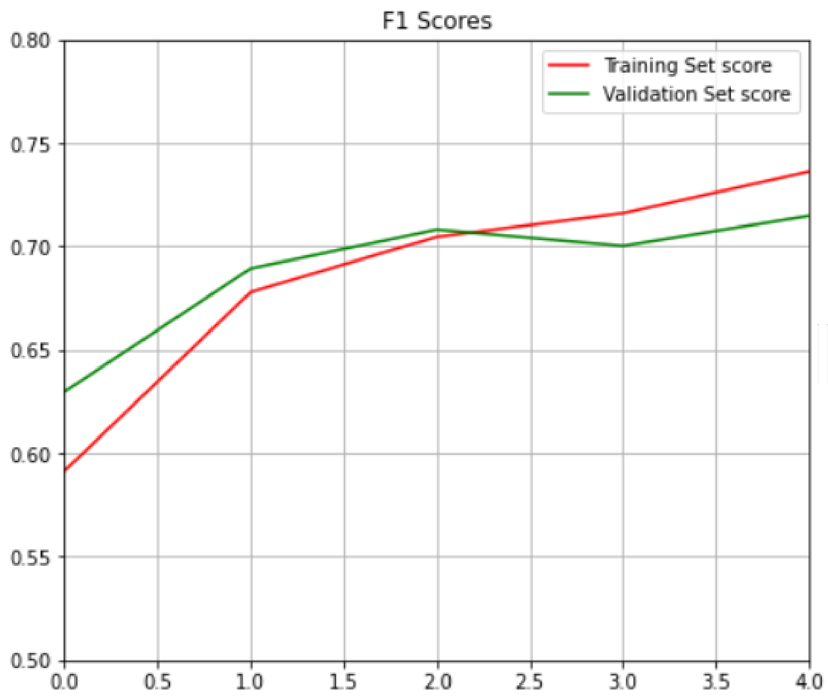
- We notice that bidirectional models tend to perform better. They learn more quickly (in terms of epochs) and perform better in each seperate class, which affects the **Precision** and especially the **Recall** score.
- Skip connections did not seem to have any significant effect in the model performance.
- GRU models achieved slightly better performance, possibly due to their simplicity, which prevents overfitting in comparison to LSTM's.
- Decreasing the Dropout probability and/or the batch size usually resulted in higher tendency for overfitting.
- Using **GloVe** embeddings with vector size = 100 appeared to be just about right. Higher vector size would cause significant overfitting in most cases.
- Just like in the previous 2 homeworks, we see much better performance on `Neutral` and `Pro-Vaccine` tweets in all models, since the vast amount of train set tweets are labeled as such.
- However, the ROC curves show that while the model may have trouble classifying `Anti-Vaccine` tweets correctly, the possibility assigned to this class when the prediction is wrong is not small. In other words, the model may be **misclassifying** the `Anti-Vaccine` tweets, but **it is not very "confident"** in these cases.

Adding Attention Layer to Model 1

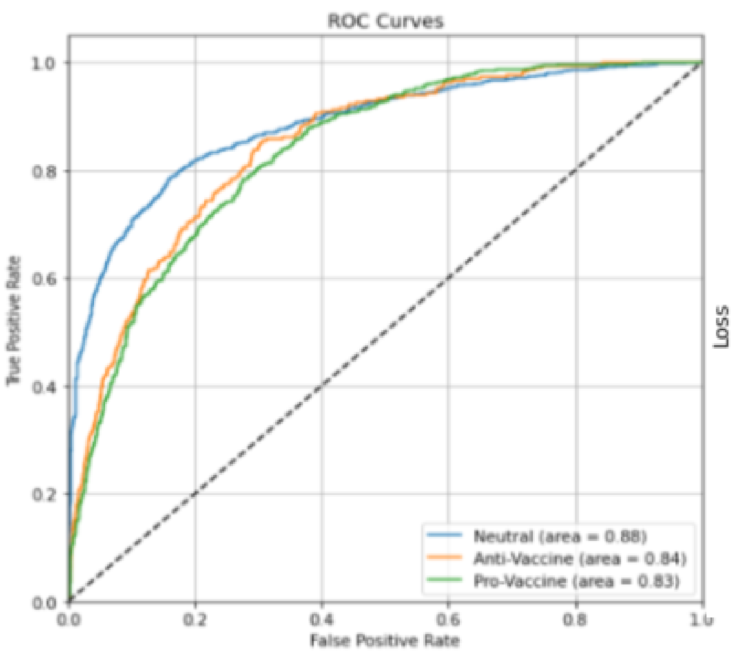
Adding the implemented Attention layer did not have any significant impact to the Model.
Below are the performance results of Model 1, after adding Attention:

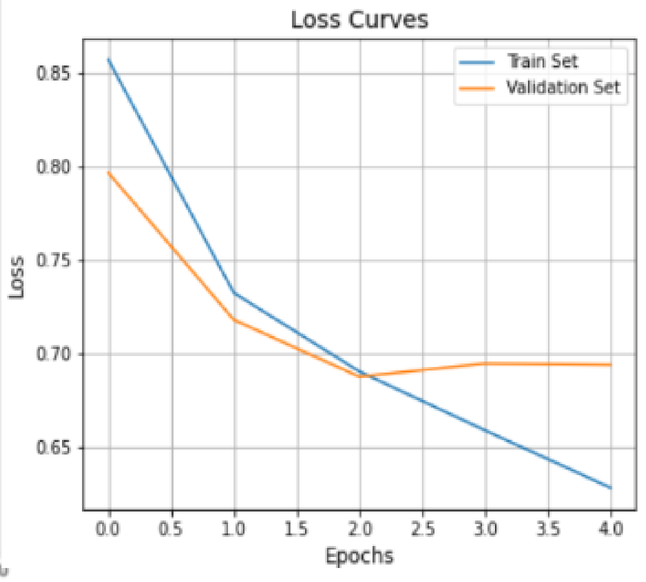






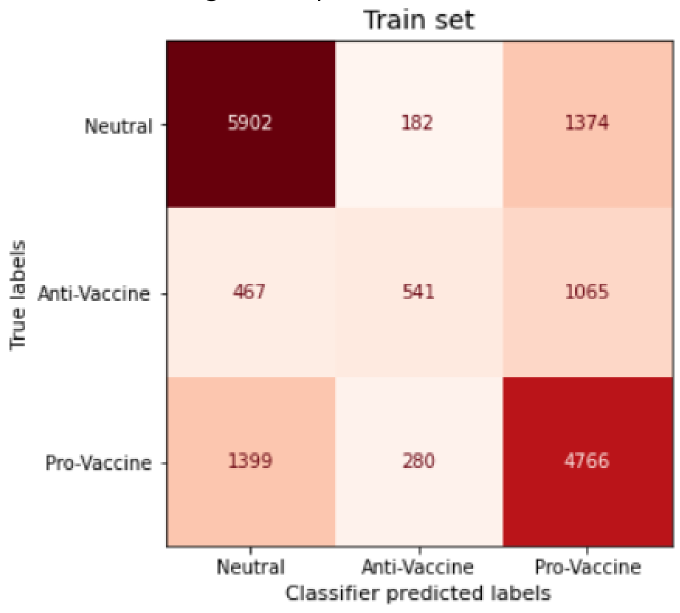
	F1	Precision	Recall
Train	0.7362	0.6931	0.6534
Validation	0.7147	0.6679	0.6365

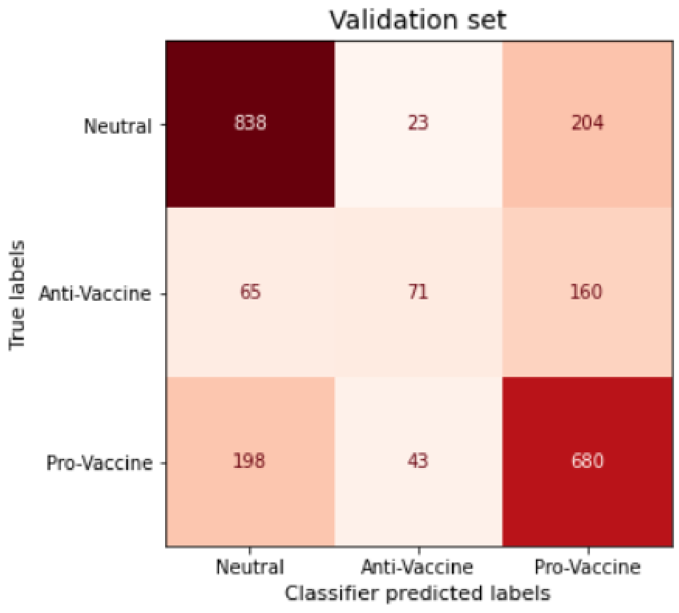


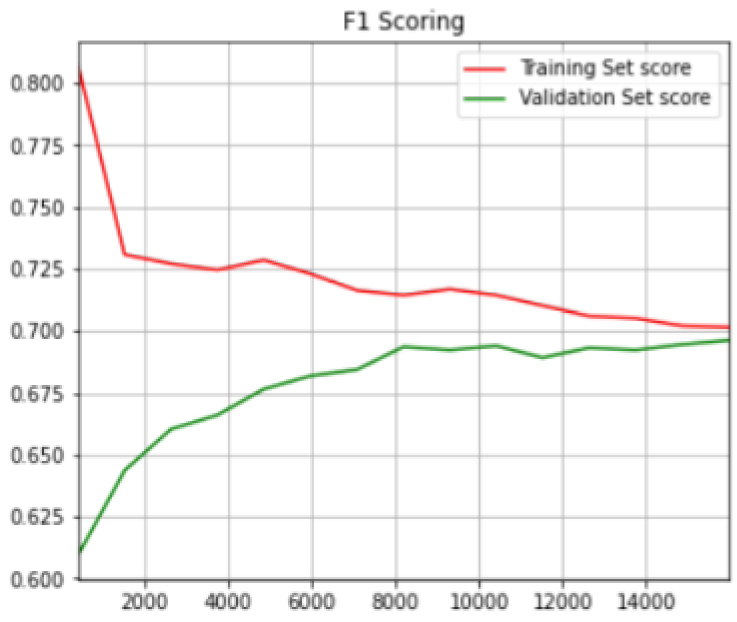


Comparison to HW1 Softmax Regression Model and HW2 Feed-Forward NN Model

HW1 SoftMax Regression performance results:

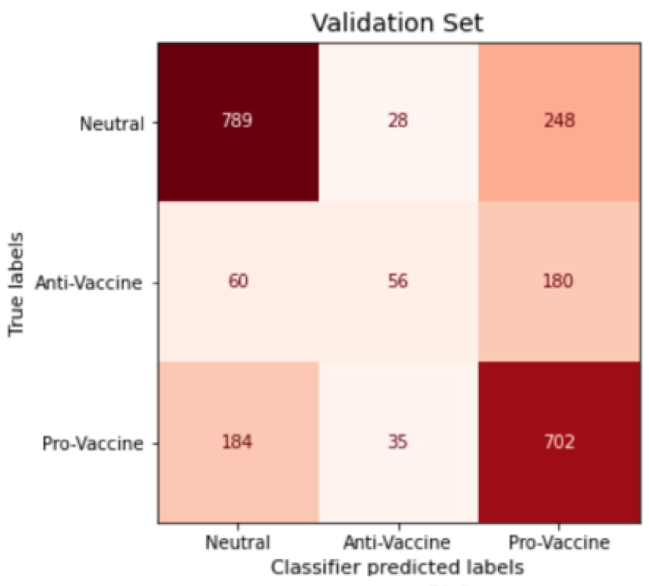


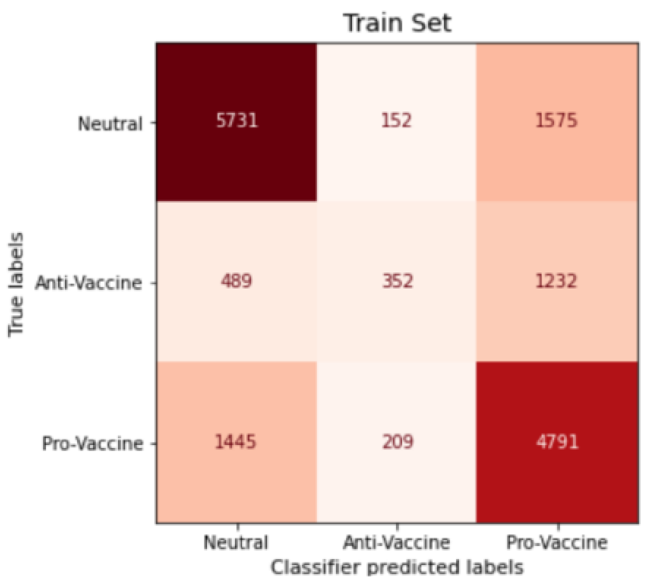


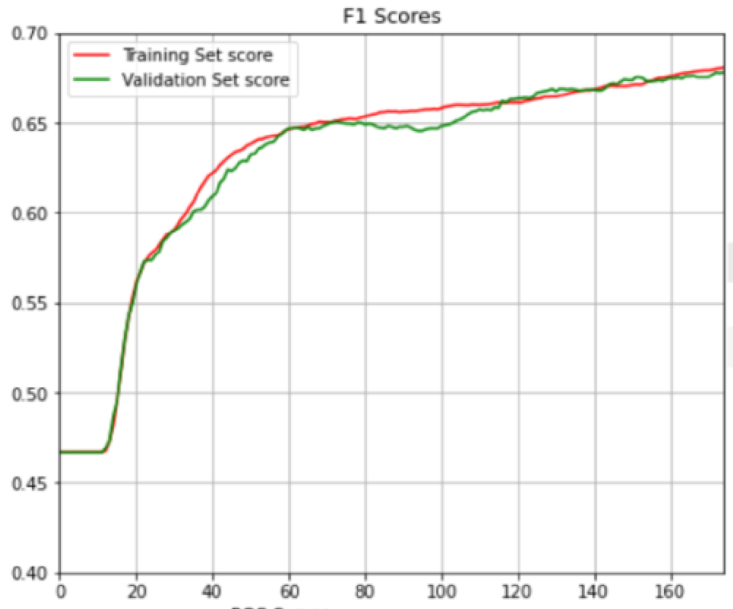


	F1	Precision	Recall
Train	0.7016	0.6536	0.5973
Validation	0.6963	0.6436	0.5883

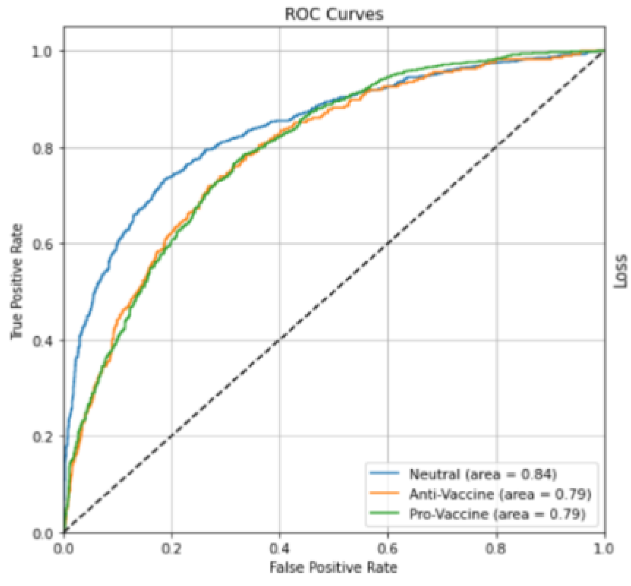
HW2 Feed-Forward NN performance results:

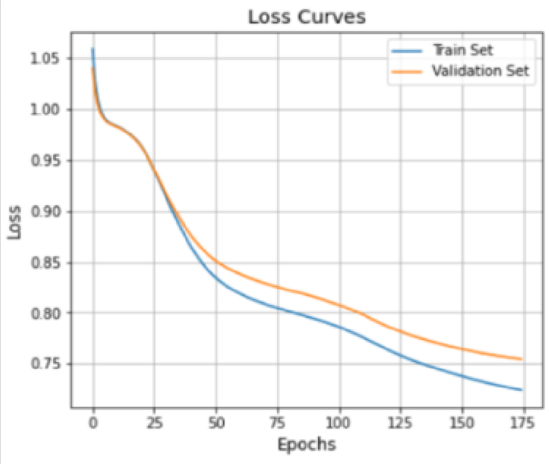






	F1	Precision	Recall
Train	0.6806	0.6240	0.5605
Validation	0.6779	0.6185	0.5641





While the macro accuracy and the F1 score have not improved a lot, we notice many performance improvements in comparison to the previous Models:

- The RNN Model performs much better in `Anti-Vaccine` tweets. This has an impact on multiple metrics:

    - The confusion matrix,
    - The Precision and especially the Recall score,
    - The ROC curves. The ROC Area for the `Anti-Vaccine` has risen to 0.85 (0.06 increase from 0.79 in the Feed-Forward Model)
- The loss value has been decreased below 0.70 (it was above 0.75 in the Feed-Forward NN).
- The RNN Model performs better in all seperate classes, as seen in the ROC Area scores.
- We can conclude that a bidirectional (or even a single-directional) RNN is a much more effective way to handle the word vectors (in comparison to the "naive" mean sentence vector used in HW2 Model), since it takes the word positions into account.

Development

The notebook has been developed mostly in Google Colab, but also in WSL Ubuntu 20.04, using Visual Studio Code & Python 3.8.10.
It has been tested successfully in the Google Colab environment, using both CPU-only and GPU-accelerated runtime engines.