# Deep Learning for Natural Language Processing

## Project II

Chalkias Spyridon

December 2021

## Contents

**1 Vaccine Sentiment Classifier** **3**
1.1 Task . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.2 Data Preprocessing . . . . . . . . . . . . . . . . . . . . . . . . 3
1.2.1 Data Cleaning . . . . . . . . . . . . . . . . . . . . . . . 3
1.2.2 Data Reduction . . . . . . . . . . . . . . . . . . . . . . 4
1.2.3 Data Transformation . . . . . . . . . . . . . . . . . . . 4
1.3 Tackling the imbalanced classes problem . . . . . . . . . . . . 4
1.4 Defining the Neural Network . . . . . . . . . . . . . . . . . . . 5
1.5 Hyperparameter Tuning . . . . . . . . . . . . . . . . . . . . . . 6
1.5.1 Framework . . . . . . . . . . . . . . . . . . . . . . . . . 6
1.5.2 Number of hidden layers . . . . . . . . . . . . . . . . . 6
1.5.3 Number of units in each layer . . . . . . . . . . . . . . 6
1.5.4 Dropouts . . . . . . . . . . . . . . . . . . . . . . . . . . 7
1.5.5 Learning rate . . . . . . . . . . . . . . . . . . . . . . . 7
1.5.6 Optimizer . . . . . . . . . . . . . . . . . . . . . . . . . 7
1.5.7 Loss function . . . . . . . . . . . . . . . . . . . . . . . 8
1.5.8 Activation function . . . . . . . . . . . . . . . . . . . . 8
1.6 Which model generalizes the best? . . . . . . . . . . . . . . . . 9
1.6.1 BoW input representation . . . . . . . . . . . . . . . . 9
1.6.2 TF-IDF input representation . . . . . . . . . . . . . . 10
1.6.3 Word Embeddings . . . . . . . . . . . . . . . . . . . . 10
1.7 Scores . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
1.8 Best Performing Model . . . . . . . . . . . . . . . . . . . . . . 23
1.9 Comparison with Softmax Regression . . . . . . . . . . . . . . 23

**2 Sources** **24**

# 1 Vaccine Sentiment Classifier

## 1.1 Task

The task is to perform sentiment analysis on a dataset consisting of tweets and be able to infer whether a new tweet is:

1. Neutral
2. Anti-vax
3. Pro-vax

by constructing a *Feed Forward Neural Network*.

## 1.2 Data Preprocessing

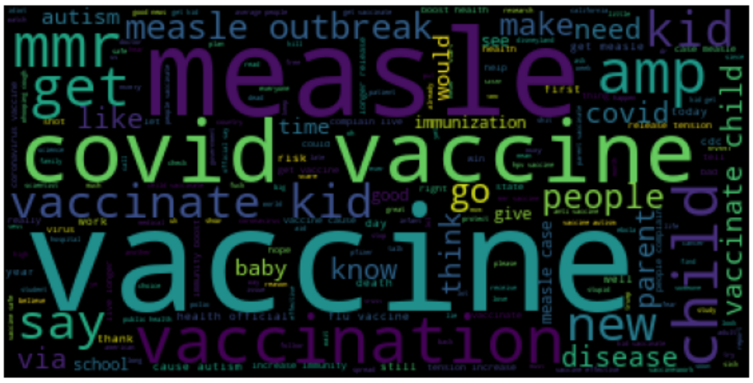
Real-world data tend to be incomplete, noisy, and inconsistent. This can lead to a poor quality of collected data and further to a low quality of models built on such data. In order to address these issues, Data Preprocessing provides operations which can organise the data into a proper form for better understanding in data mining process.
Some of the methods applied in order to prepare the data for analysis are:

- Data Cleaning
- Data Reduction
- Data Transformation

### 1.2.1 Data Cleaning

The cleaning process involved two (3) operations:

- Visualizing the tweets using **Wordclouds**, in order to expose the dataset's keywords. An indicative wordcloud is shown below:



- Checking whether there's need to drop data rows that contain *null* values. Luckily, the data had no such rows.
- Cleaning tweets from
  - URLs.
  - Noise contained in '< >'parenthesis.
  - The phrase 'RT' (meaning retweet).
  - '@' and the following twitter account name.
  - Any punctuation.
  - '\r\n' which is present in some strings.
  - Numbers, capitalization and white space.

  which are totally useless and add noise to the data.

### 1.2.2 Data Reduction

Data had a significant reduction process through *Tokenization and Lemmatization*, in order to group together the inflected forms of the words so they can be analysed in less items which are more independent and have greater semantic content. Also, all the stopwords were removed, so that data delivers more semantic content with as few words as possible. The stopwords container used was provided by *NLTK* library and the lemmatization process was supported by *spaCy* library.

### 1.2.3 Data Transformation

The data was transformed by testing three (3) representations:

1. **Bag of Words (BoW)** method in which the input is nothing but an array with size of vocabulary in the corpus and values that are the frequencies of words in corpus with index being the unique id of the word.
2. **Term Trequency-Inverse Document Frequency (TF-IDF)** representation using sklearn's **TfidfVectorizer()**.
3. **Word Embeddings** using pre-trained vectors from GloVe's twitter dataset.

## 1.3 Tackling the imbalanced classes problem

An imbalanced classification problem is an example of a classification problem where the distribution of examples across the known classes is biased or skewed. The distribution can vary from a slight bias to a severe imbalance where there is one example in the minority class for hundreds, thousands, or millions of examples in the majority class or classes.

In our case, the dataset suffers from the aforementioned problem, since the *Neutral* class consists of 7458 samples, the *Anti-Vax* class consists of 2073 samples and the *Pro-Vax* class consists of 6445 samples.

There are three (3) methods that have been tested in order to reduce the class imbalance:

- Upsampling by duplicating samples in minor class.
- Downsampling by cutting down samples from the major classes.
- Upsampling by generating synthetic samples from the minor class.

In the dataset provided, the first method (*Upsampling by duplicating*) proved to perform the best in Word Embeddings model compared to all the other methods. The aforementioned method resulted in greater weighted f1-score and accuracy in the minor class, as the duplicate samples provided more support and added generalization confidence to the model.

In the case of BoW model, no method was as efficient as the raw data themselves.

In the given *.ipynb* notebook, there has not been applied any imbalance tackling method to the dataset.

### 1.4 Defining the Neural Network

A simple PyTorch neural network is defined, that accepts a list of layers and a list of dropouts.

**torch.nn.Sequential** is used in order to stack the layers together. More specifically:

- **torch.nn.Linear** is used to define the network's layers.
- **torch.nn.Sigmoid** is used as an activation function.
- **torch.nn.Dropout** is used as a dropout layer.
- **torch.nn.functional.log_softmax** is applied to the output layer. Keep in mind that Log Softmax is advantageous over Softmax for numerical stability, optimisation and heavy penalisation for highly incorrect class.

> **Note:** When using **nn.CrossEntropyLoss**, log softmax is being internally performed, so there's no need to manually apply it to the output layer.

For example given the following input:

- Layers: [10, 5, 3]
- Dropouts: [0.3, 0.1]

a Neural Network with input size 10, a hidden layer of 5 units and an output layer with 3 units will be defined. Also, there will be 2 dropout layers of 30% and 10% activation probabilities respectively. Dropouts can also be empty.

## 1.5 Hyperparameter Tuning

### 1.5.1 Framework

The framework used for the neural network's tuning is **Optuna**. Optuna is an open source hyperparameter optimization framework that automates hyperparameter search.

In this project, several features were tested and fine-tuned in order maximize the network's generalization capabilities, which are discussed below.

**IMPORTANT NOTE:** In the notebook provided with this report, the tuning function only tunes:

- The number of network layers.
- The number of units in every layer.
- The percentages of dropouts in each layer.

That is, because the activation function, the loss function and the optimizer (along with the learning rate) proved to be optimal for this project since the very first tunings, so they were removed from the tuning process in order to significantly reduce the time taken by the tuning process and focus on a more extensive layer-unit-dropout tuning.

### 1.5.2 Number of hidden layers

The optimal number of hidden layers seemed to be between two (2) and three (3) hidden layers, with the statistically best results produced by three (3) hidden layers.

### 1.5.3 Number of units in each layer

The unit number for each layer that performed best has a maximum of 100 units per layer.

### 1.5.4 Dropouts

Dropouts with activation probability close to 70% were optimal when applied to the input layer, whereas dropouts from 0% to 50% were most suitable, when applied to the inner layers.

### 1.5.5 Learning rate

As we will discuss extensively below, the two best performing optimizers were *Adam* and *Stochastic Gradient Descent*. Adam optimizer sets its own learning rates during the training session, so there's no unique optimal learning rate associated with it. However, when running SGD ,the optimal order of magnitude for the learning rate is $10^{-1}$.

### 1.5.6 Optimizer

The number of optimizers tested was three (3):

- Adam
- RMSprop
- Stochastic Gradient Descent (SGD)

SGD is computationally fast as only one sample is processed at a time. Also, due to frequent updates, the steps taken towards the minima of the loss function have oscillations that can help to get out of the local minimums. However, due to frequent updates, the steps taken towards the minima are very noisy. This can often lean the gradient descent into other directions. Also, due to noisy steps, it may take longer to achieve convergence to the minima of the loss function.

In a few words, RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numinator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. Experiments showed that bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser.

Insofar, **Adam** and **SGD** performed the best in most of the training sessions, with Adam being a tiny step ahead, mostly because of its higher weighted average f1-score and faster learning process. So, **Adam was the best overall choice**.

### 1.5.7 Loss function

The loss functions tested were three (3):

- Cross Entropy Loss
- Negative log likelihood loss
- Multi Margin Loss

Cross Entropy Loss and Negative log likelihood loss are essentially the same thing, except the former performs log softmax internally, while the latter expects you to manually use such function to the output layer.

The main difference between the Multi-Margin loss and the Cross Entropy loss is that the former arises from trying to maximize the margin between our decision boundary and data points - thus attempting to ensure that each point is correctly and confidently classified, while the latter comes from a maximum likelihood estimate of our model's parameters.

The softmax function, whose scores are used by the cross entropy loss, allows us to interpret our model's scores as relative probabilities against each other. For example, the cross-entropy loss would invoke a much higher loss than the multi margin loss if our (un-normalized) scores were [10, 8, 8] versus [10, -10, -10], where the first class is correct.

In fact, the multi margin loss would recognize that the correct class score already exceeds the other scores by more than the margin, so it will invoke zero loss on both scores. Once the margins are satisfied, the SVM will no longer optimize the weights in an attempt to "do better" than it is already.

That way, Cross Entropy Loss is considered more efficient and suitable for the current project.

### 1.5.8 Activation function

The number of activation functions tested was four (4):

- Sigmoid
- Tanh
- Rectified Linear Unit (ReLU)
- Leaky ReLU

Generally speaking, sigmoids saturate and **kill gradients**. Also, sigmoid outputs are not zero-centered.

The tanh non-linearity squashes a real-valued number to the range $[-1, 1]$. Like the sigmoid neuron, its activations **saturate**, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity.

Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero. Unfortunately, ReLU units can be fragile during training and can "**die**". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron **will never activate** on any datapoint again.

Leaky ReLUs are one attempt to fix the "dying ReLU" problem. Instead of the function being zero when $x < 0$, a leaky ReLU will instead have a small positive slope (of 0.01, or so). However, the results are not always consistent.

In the current project, the models that performed the best amongst all the other ones are:

- **Adam** optimizer with **Sigmoid** activation function.
- **SGD** optimizer with **LeakyReLU** activation function.

While **LeakyReLU** performed slightly better than ReLU and ReLU performed better than any other activation function when having SGD as an optimizer, **Sigmoid** was chosen as the final model's activation function, because the overall best performing optimizer was Adam.

## 1.6 Which model generalizes the best?

### 1.6.1 BoW input representation

One pretty interesting approach is the BoW model. The input in this model is being transformed to an array with the size of vocabulary in the corpus and values that are the frequencies of words in corpus with index being the unique id of the word. That way, one can extract the vocabulary of the corpus and start training the model simply from the frequency of words within the corpus.

Some of the key advantages of this model are:

- It can be fully trained (and not overfit) in a very small number of epochs.
- It is very understandable and has intuitive and simple design.

The aforementioned model proved to perform the highest in contrast to all the other models, by scoring a weighted average f1-score of 73%.

### 1.6.2 TF-IDF input representation

TF-IDF model uses *sklearn's* **TfidfVectorizer** in order to transform the data and to reflect how important some words and phrases are to a document in the provided corpus.

TF-IDF transformed model clearly didn't stand a chance, since its enormous input size made it impossible to run any mini-batch model in Google Colab. However, it is being kept in the provided notebook, since it is of great educational value and it can perform pretty well with smaller and more manageable datasets.

### 1.6.3 Word Embeddings

A pretty efficient and manageable model was the Word Embeddings model. In order to train the model, GloVe's pre-trained Twitter dataset was downloaded and used.

Some of the key advantages of this model are:

- Its small input size. The input size of the aforementioned model can be at max 200 units, since GloVe's pre-trained vectors can be maximum 200-dimensional. That way, the training sessions are way faster than every other model's.
- It enforces the word vectors to capture sub-linear relationships in the vector space.
- GloVe does not rely just on local statistics (local context information of words), but incorporates global statistics (word co-occurrence) to obtain word vectors.

The Word Embeddings model proved to performed pretty high, by scoring a weighted average f1-score of 70%.

## 1.7 Scores

The following diagrams are showcasing **only a few** of the most important training sessions that, apart from tuning, were critical for the choosing of the overall best performing models.

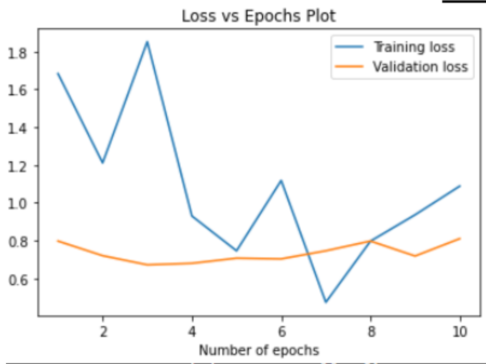
Note that the training sessions corresponding to the following diagrams do not appear in the provided notebook, because that would make it huge. The notebook only showcases the best performing version of every model.

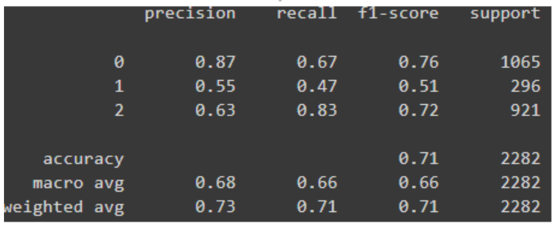
**<u>BoW</u>**

**<u>SGD + ReLU + lr=0.01</u>**

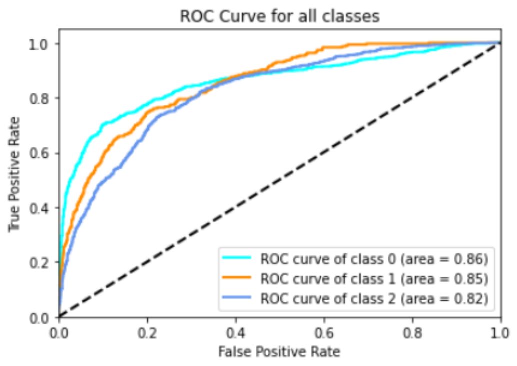
**<u>4 hidden layers</u>**

**<u>Upsampled</u>**



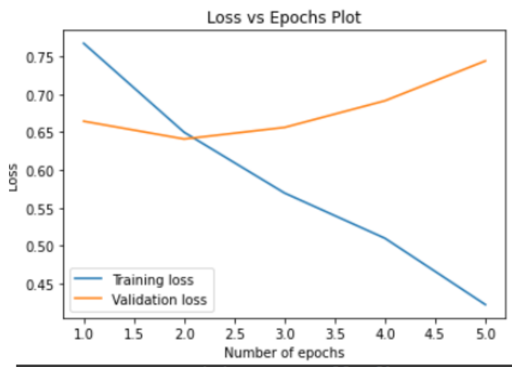


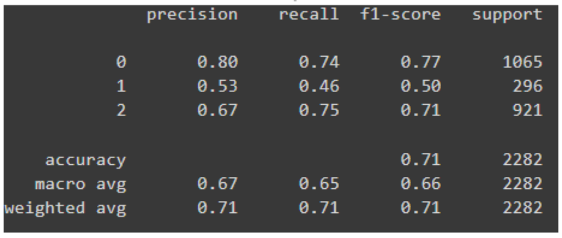
```
              precision    recall  f1-score   support

           0       0.87      0.67      0.76      1065
           1       0.55      0.47      0.51       296
           2       0.63      0.83      0.72       921

    accuracy                           0.71      2282
   macro avg       0.68      0.66      0.66      2282
weighted avg       0.73      0.71      0.71      2282
```



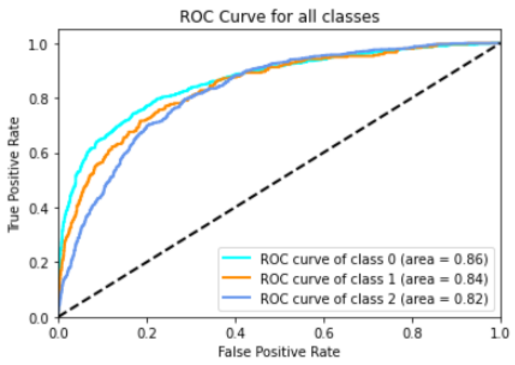
## SGD + ReLU + lr=0.01

## 1 hidden layer





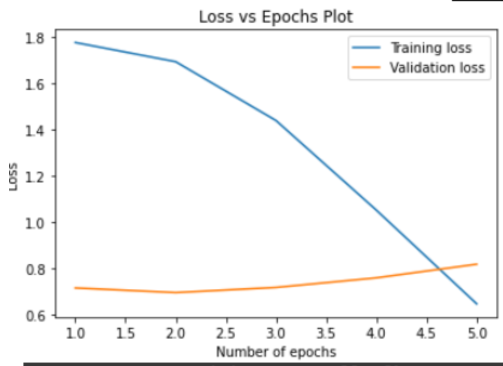
```
              precision    recall  f1-score   support

           0       0.80      0.74      0.77      1065
           1       0.53      0.46      0.50       296
           2       0.67      0.75      0.71       921

    accuracy                           0.71      2282
   macro avg       0.67      0.65      0.66      2282
weighted avg       0.71      0.71      0.71      2282
```

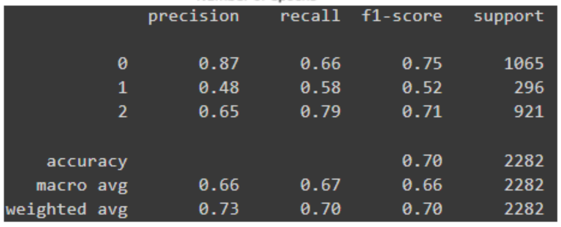


## SGD + ReLU + lr=0.01

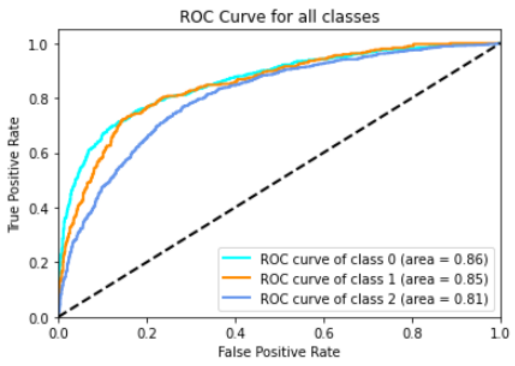
## 1 hidden layer

## Upsampled

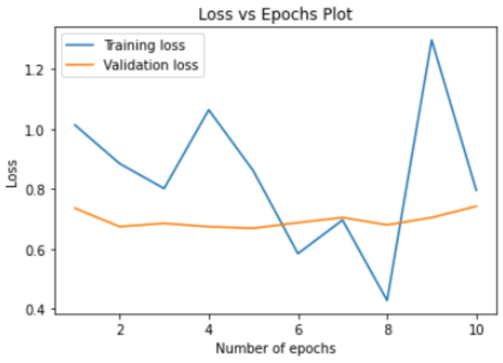


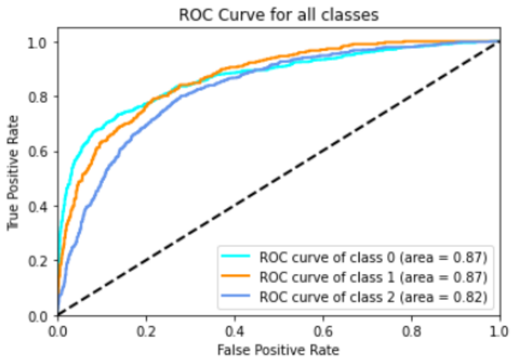


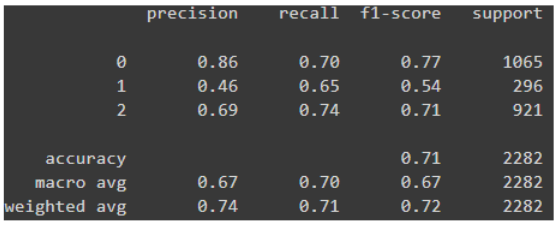
```
              precision    recall  f1-score   support

           0       0.87      0.66      0.75      1065
           1       0.48      0.58      0.52       296
           2       0.65      0.79      0.71       921

    accuracy                           0.70      2282
   macro avg       0.66      0.67      0.66      2282
weighted avg       0.73      0.70      0.70      2282
```



## SGD + Leaky ReLU + lr=0.01

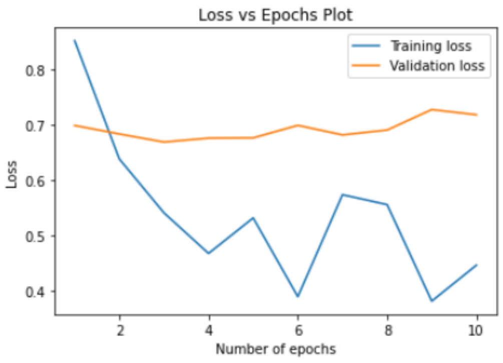


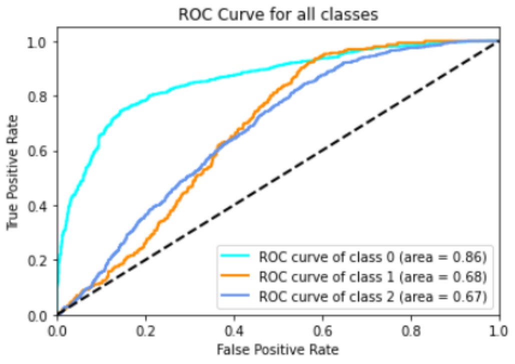


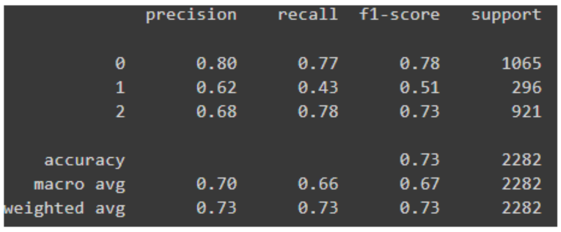


```
              precision    recall  f1-score   support

           0       0.86      0.70      0.77      1065
           1       0.46      0.65      0.54       296
           2       0.69      0.74      0.71       921

    accuracy                           0.71      2282
   macro avg       0.67      0.70      0.67      2282
weighted avg       0.74      0.71      0.72      2282
```

## Adam + ReLU



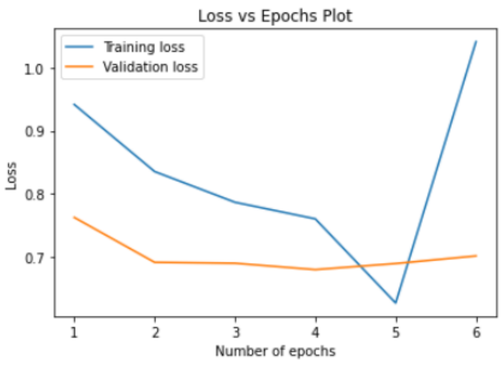


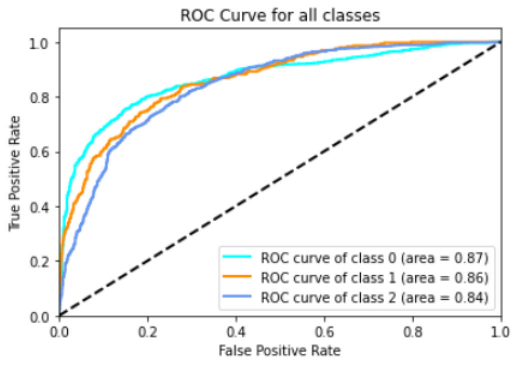


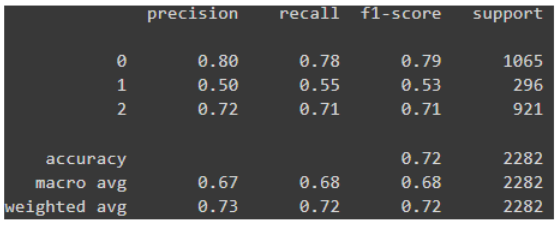
```
              precision    recall  f1-score   support

           0       0.80      0.77      0.78      1065
           1       0.62      0.43      0.51       296
           2       0.68      0.78      0.73       921

    accuracy                           0.73      2282
   macro avg       0.70      0.66      0.67      2282
weighted avg       0.73      0.73      0.73      2282
```

## Adam + Sigmoid

## Upsampled

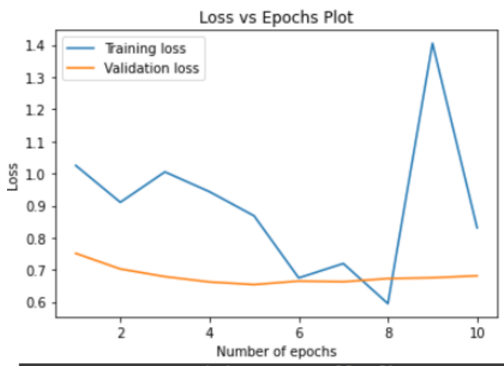


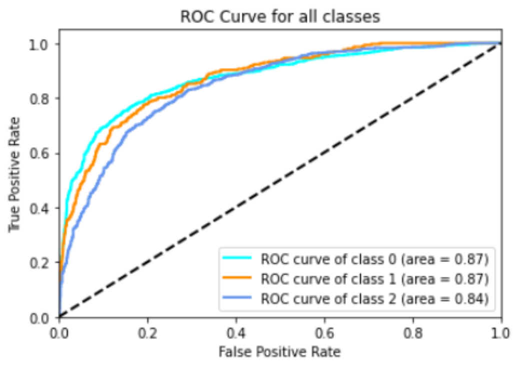


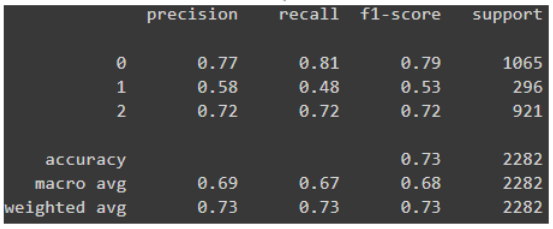


```
              precision    recall  f1-score   support

           0       0.80      0.78      0.79      1065
           1       0.50      0.55      0.53       296
           2       0.72      0.71      0.71       921

    accuracy                           0.72      2282
   macro avg       0.67      0.68      0.68      2282
weighted avg       0.73      0.72      0.72      2282
```

## Adam + Sigmoid







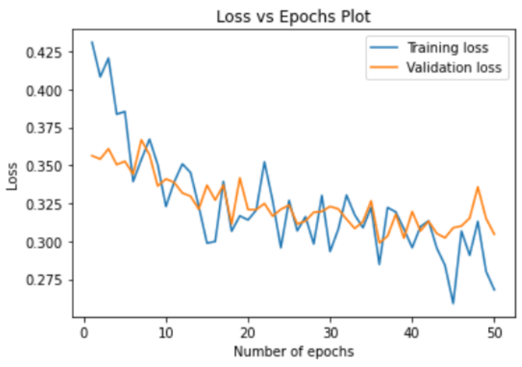
```
              precision    recall  f1-score   support

           0       0.77      0.81      0.79      1065
           1       0.58      0.48      0.53       296
           2       0.72      0.72      0.72       921

    accuracy                           0.73      2282
   macro avg       0.69      0.67      0.68      2282
weighted avg       0.73      0.73      0.73      2282
```

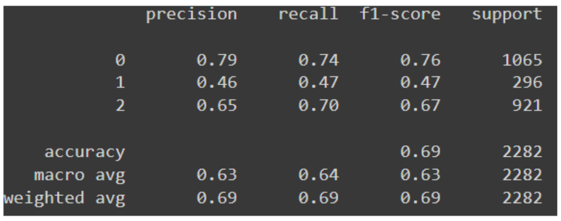
Overall, SGD often tends to have less noisy diagrams compared to Adam, but Adam is a very fast learner compared to SGD. Also, heavy dropout layers where used in order to avoid overfitting, so it is very reasonable for the results to be more noisy or vary during multiple training sessions of the same model. Following the previous statements, Adam with sigmoid as an activation function resulted to the best fit in the ROC curve, by producing very truthful predictions and to the overall best classification report.

## Word Embeddings

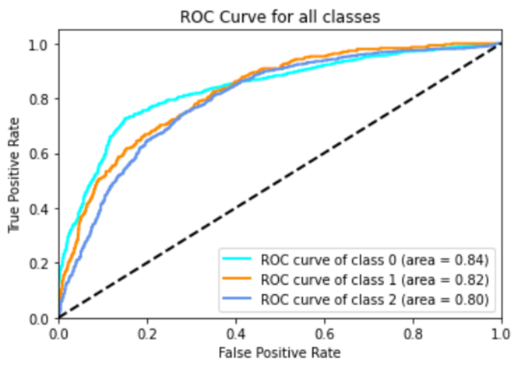
### Loss functions:

**SGD + Multi Margin Loss with learning rate=0.01**



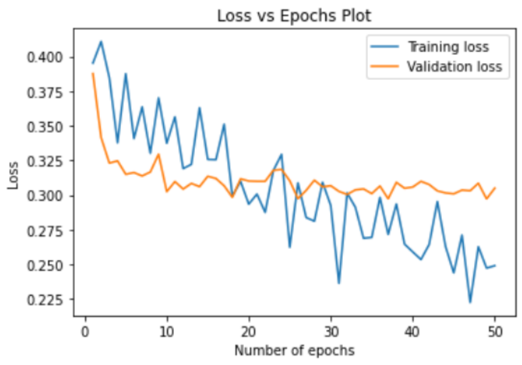


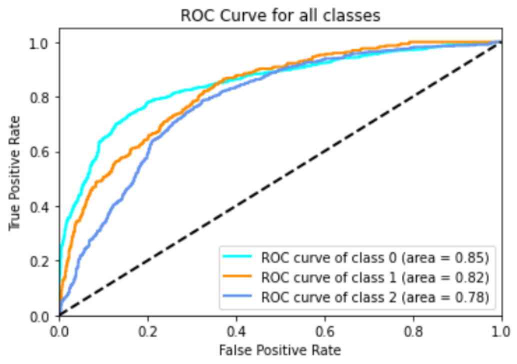
```
              precision    recall  f1-score   support

           0       0.79      0.74      0.76      1065
           1       0.46      0.47      0.47       296
           2       0.65      0.70      0.67       921

    accuracy                           0.69      2282
   macro avg       0.63      0.64      0.63      2282
weighted avg       0.69      0.69      0.69      2282
```

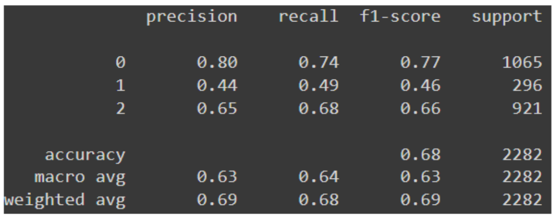


The diagram show above is pretty noisy, but it produces pretty truthful results (ROC curve) and has a high classification report.

## Adam + Multi Margin Loss



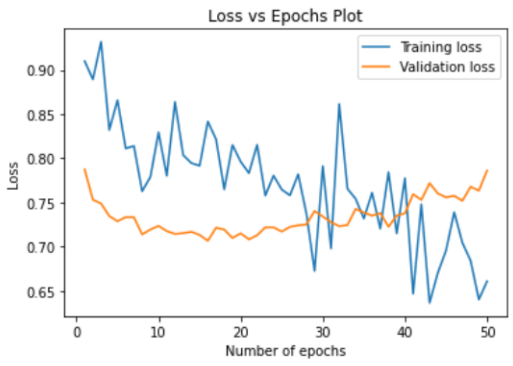


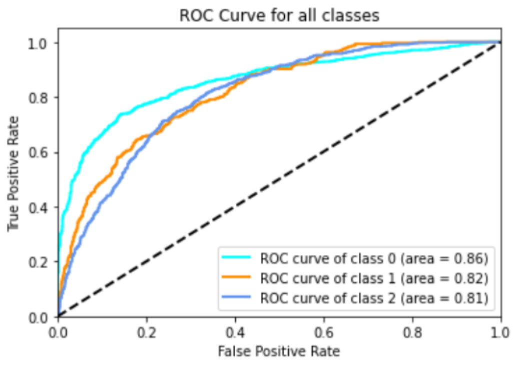


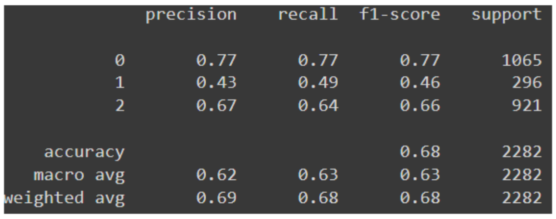
```
              precision    recall  f1-score   support

           0       0.80      0.74      0.77      1065
           1       0.44      0.49      0.46       296
           2       0.65      0.68      0.66       921

    accuracy                           0.68      2282
   macro avg       0.63      0.64      0.63      2282
weighted avg       0.69      0.68      0.69      2282
```

The above model seems to fit well, but the resulting Loss-to-Epochs diagram is pretty noisy.

## Adam + NLLLoss





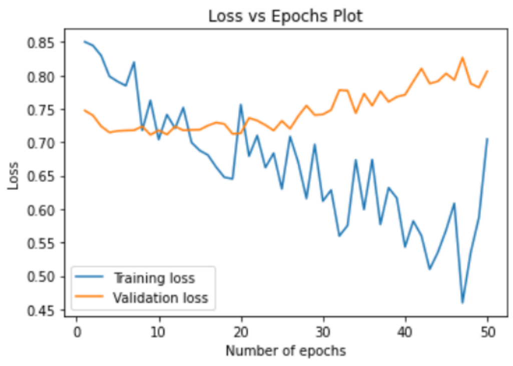


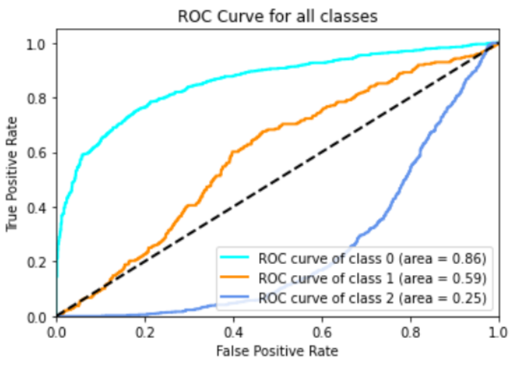
```
              precision    recall  f1-score   support

           0       0.77      0.77      0.77      1065
           1       0.43      0.49      0.46       296
           2       0.67      0.64      0.66       921

    accuracy                           0.68      2282
   macro avg       0.62      0.63      0.63      2282
weighted avg       0.69      0.68      0.68      2282
```

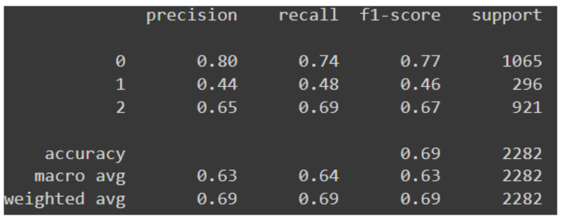
Similarly to the previous model, the outlook of the training process is very noisy.

**Activation functions:**

**Adam + LeakyReLU**



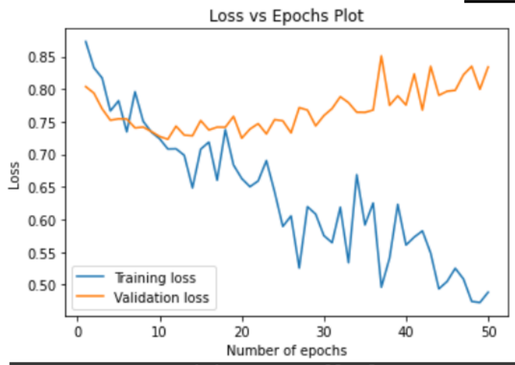


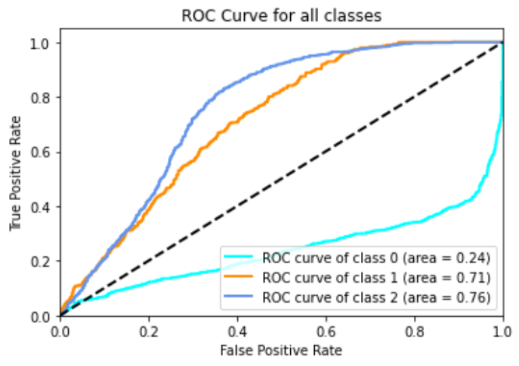


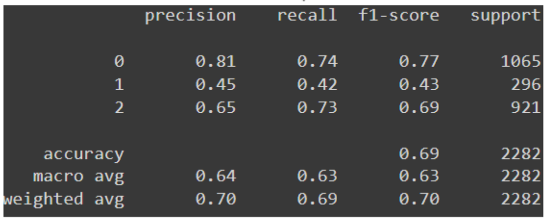
```
              precision    recall  f1-score   support

           0       0.80      0.74      0.77      1065
           1       0.44      0.48      0.46       296
           2       0.65      0.69      0.67       921

    accuracy                           0.69      2282
   macro avg       0.63      0.64      0.63      2282
weighted avg       0.69      0.69      0.69      2282
```

The above diagram is very noisy, overfits quickly and the ROC curve has loads of false estimations.

**Adam + ReLU**



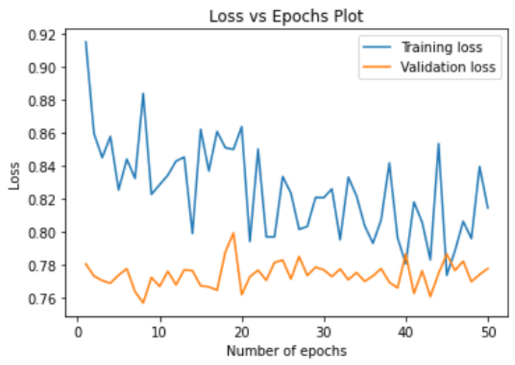


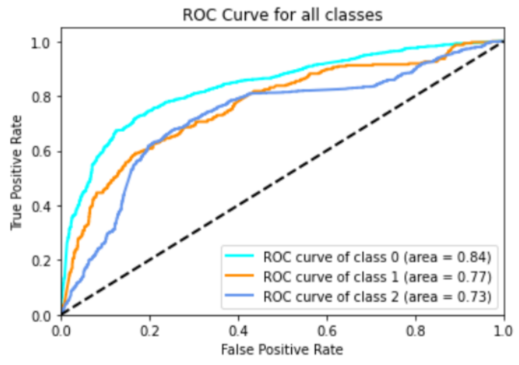


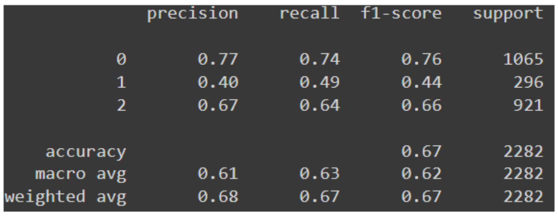
```
              precision    recall  f1-score   support

           0       0.81      0.74      0.77      1065
           1       0.45      0.42      0.43       296
           2       0.65      0.73      0.69       921

    accuracy                           0.69      2282
   macro avg       0.64      0.63      0.63      2282
weighted avg       0.70      0.69      0.70      2282
```

The above diagram is pretty **noisy** and the ROC curve's outlook is pretty disappointing considering class 0. Also, it is obvious that the model is prone to overfitting.

**Adam + Tanh**



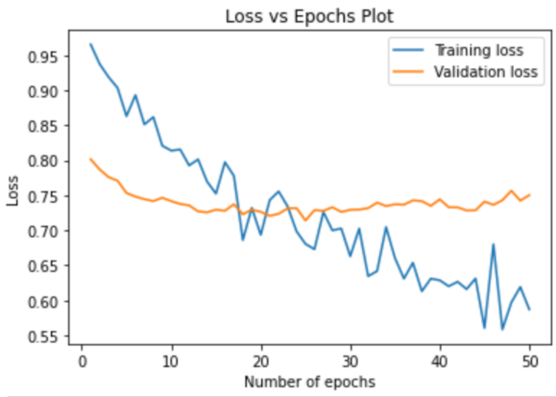


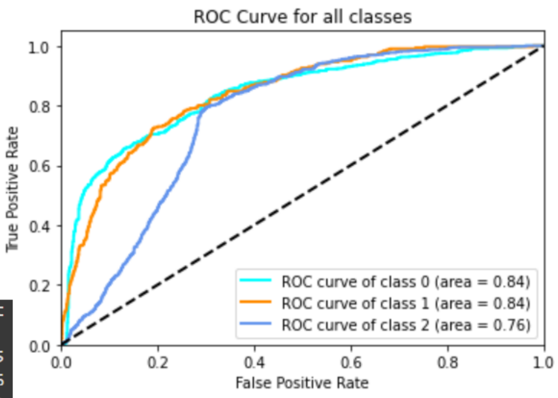


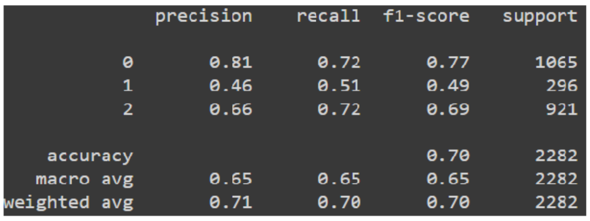
```
              precision    recall  f1-score   support

           0       0.77      0.74      0.76      1065
           1       0.40      0.49      0.44       296
           2       0.67      0.64      0.66       921

    accuracy                           0.67      2282
   macro avg       0.61      0.63      0.62      2282
weighted avg       0.68      0.67      0.67      2282
```

*tanh* activation function is not optimal, since the training process is very noisy and does not seem to take advantage of the information provided.

**Adam + Sigmoid**





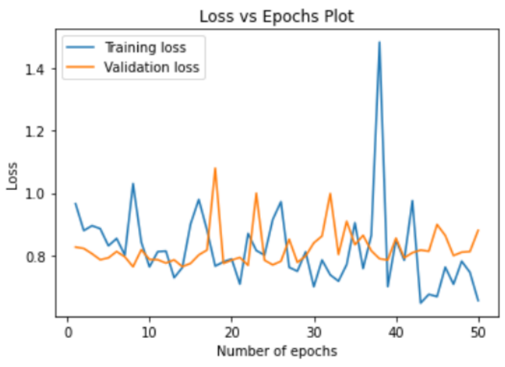


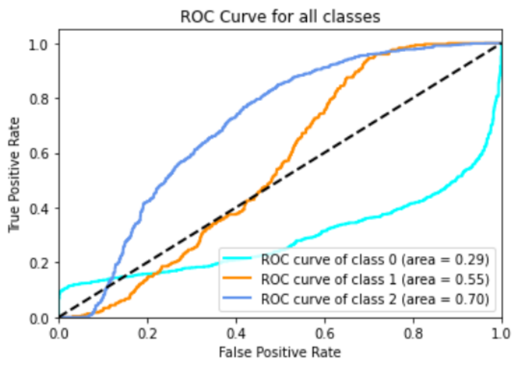
```
              precision    recall  f1-score   support

           0       0.81      0.72      0.77      1065
           1       0.46      0.51      0.49       296
           2       0.66      0.72      0.69       921

    accuracy                           0.70      2282
   macro avg       0.65      0.65      0.65      2282
weighted avg       0.71      0.70      0.70      2282
```

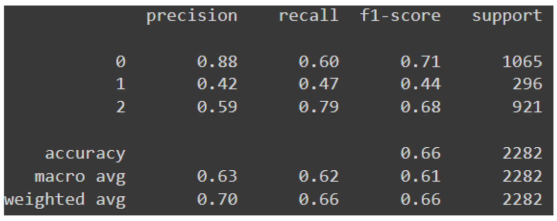
The above model fits pretty well, as the ROC curve has good looking results and accurate estimations, the Loss-to-Epochs diagram is not very noisy and the classification report is top of the heap compared to the other reports produced. The only existing downside, is that the model is prone to overfitting, so have to be aware of the training epochs we use.

**Testing RMSprop:**

**RMSprop with learning rate=0.003**





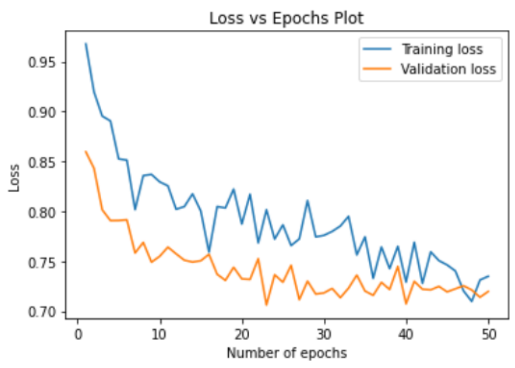


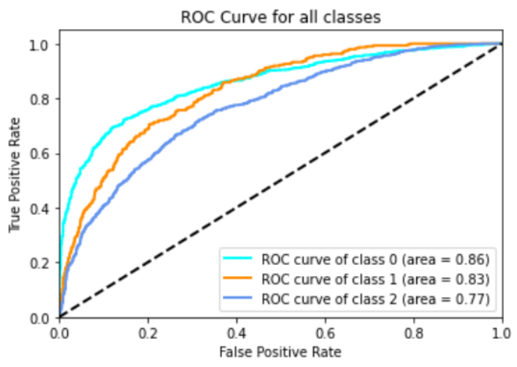
```
              precision    recall  f1-score   support

           0       0.88      0.60      0.71      1065
           1       0.42      0.47      0.44       296
           2       0.59      0.79      0.68       921

    accuracy                           0.66      2282
   macro avg       0.63      0.62      0.61      2282
weighted avg       0.70      0.66      0.66      2282
```

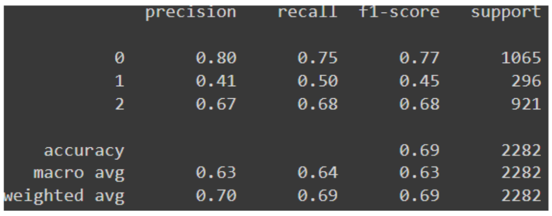
RMSprop does not fit to the current project's data, as both training and validation losses just bounce up and down instead of decreasing as the epochs go by. Also, the corresponding ROC curve has loads of false estimations.

**Trying out the optimal learning rate for SGD:**

## SGD with learning rate=0.003

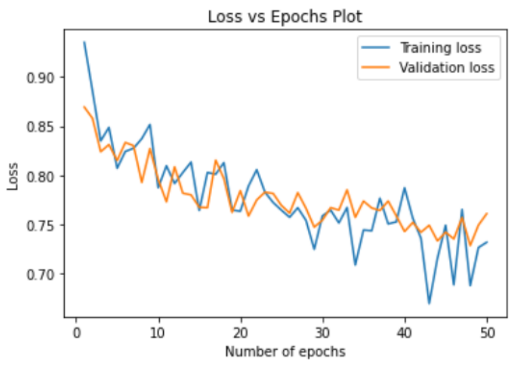


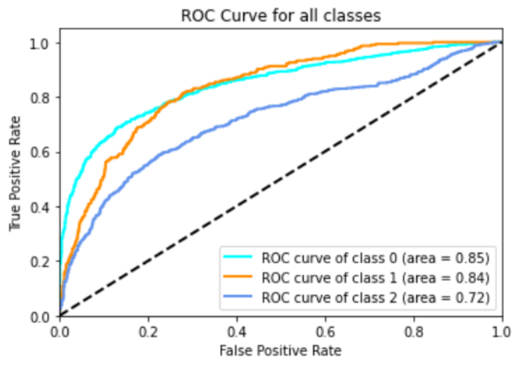


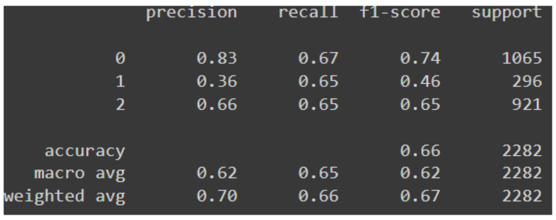


```
              precision    recall  f1-score   support

           0       0.80      0.75      0.77      1065
           1       0.41      0.50      0.45       296
           2       0.67      0.68      0.68       921

    accuracy                           0.69      2282
   macro avg       0.63      0.64      0.63      2282
weighted avg       0.70      0.69      0.69      2282
```

## SGD with learning rate=0.01

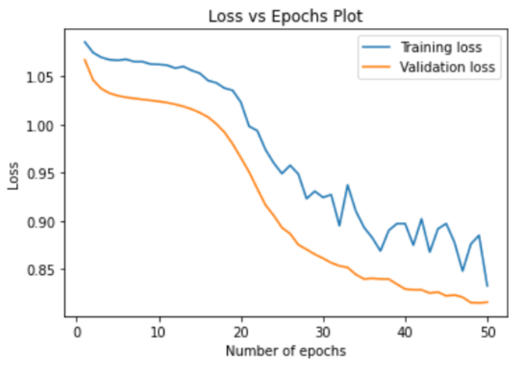


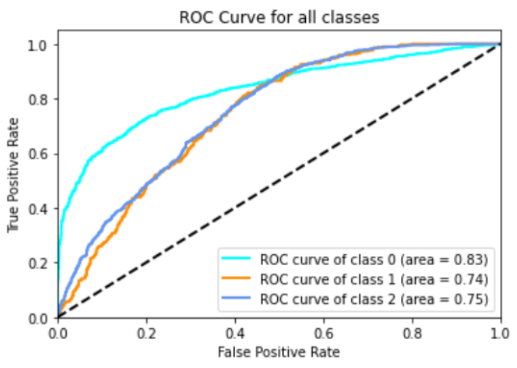


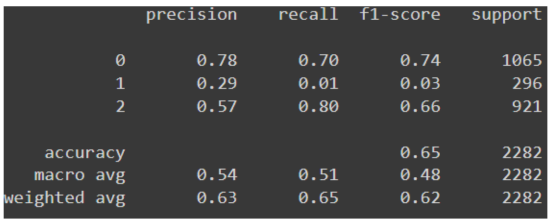


```
              precision    recall  f1-score   support

           0       0.83      0.67      0.74      1065
           1       0.36      0.65      0.46       296
           2       0.66      0.65      0.65       921

    accuracy                           0.66      2282
   macro avg       0.62      0.65      0.62      2282
weighted avg       0.70      0.66      0.67      2282
```

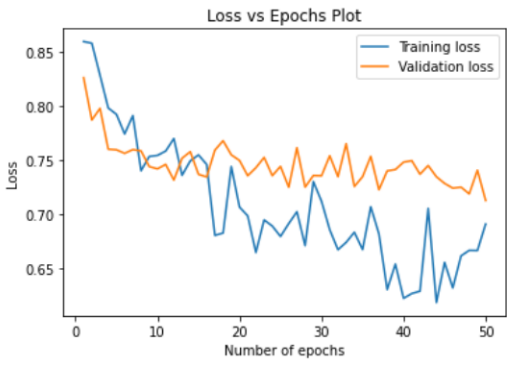
**SGD with learning rate=0.00001**

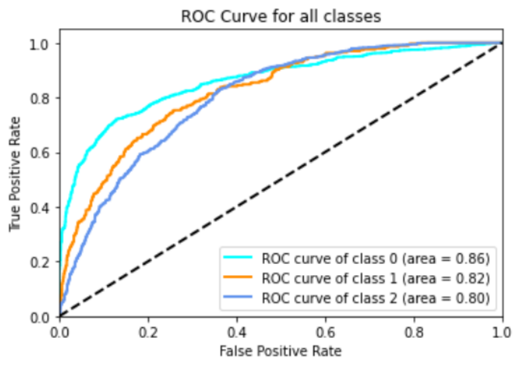


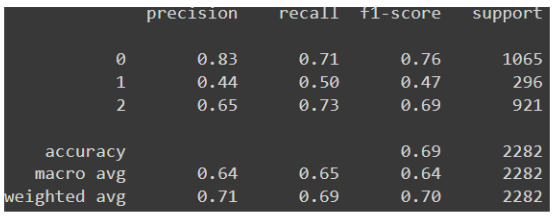



```
              precision    recall  f1-score   support

           0       0.78      0.70      0.74      1065
           1       0.29      0.01      0.03       296
           2       0.57      0.80      0.66       921

    accuracy                           0.65      2282
   macro avg       0.54      0.51      0.48      2282
weighted avg       0.63      0.65      0.62      2282
```

**SGD with learning rate=0.02**






```
              precision    recall  f1-score   support

           0       0.83      0.71      0.76      1065
           1       0.44      0.50      0.47       296
           2       0.65      0.73      0.69       921

    accuracy                           0.69      2282
   macro avg       0.64      0.65      0.64      2282
weighted avg       0.71      0.69      0.70      2282
```

As we can see from all of the above diagrams, the optimal learning rate for SGD is 0.01 since it is producing a model that does not overfit, has overall truthful estimations as epochs go by (ROC curve) and has relatively high precision, recall and f1-score for all of the 3 classes.

## 1.8 Best Performing Model

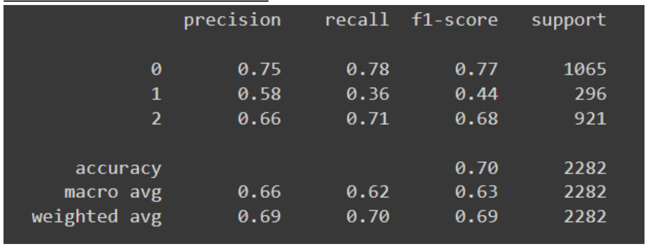
Based on all the testing, tuning and experimenting, the overall **best performing model** is a BoW Feed Forward Neural Network consisting of:

- An input layer consisting of the dataset's vacabulary size.
- An output layer consisting of 3 units.
- 3 hidden layers consisting of 100, 80 and 20 units respectively.
- Sigmoid activation function to the hidden layers.
- 2 dropout layers having 70% and 50% dropout probabilities respectively.
- Adam optimizer.
- Cross Entropy loss function.
- 10 epochs of training.

## 1.9 Comparison with Softmax Regression

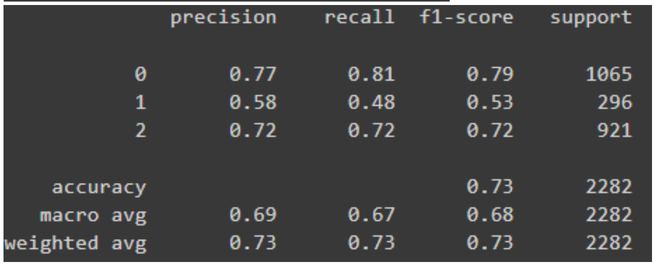
Neural networks are somewhat related to logistic regression. Basically, we can think of logistic regression as a one layer neural network. In fact, it is very common to use logistic sigmoid functions as activation functions in the hidden layer of a neural network.

**Softmax Classifier**



```
              precision    recall  f1-score   support

           0       0.75      0.78      0.77      1065
           1       0.58      0.36      0.44       296
           2       0.66      0.71      0.68       921

    accuracy                           0.70      2282
   macro avg       0.66      0.62      0.63      2282
weighted avg       0.69      0.70      0.69      2282
```

**Feed Forward Neural Network**



```
              precision    recall  f1-score   support

           0       0.77      0.81      0.79      1065
           1       0.58      0.48      0.53       296
           2       0.72      0.72      0.72       921

    accuracy                           0.73      2282
   macro avg       0.69      0.67      0.68      2282
weighted avg       0.73      0.73      0.73      2282
```

In this case, we can see that the Feed Forward Neural Network we constructed is more powerful than the previous model based on Softmax Classification. More specifically the newer model **excels in every metric of every class!**

Overall, by looking at the weighted averages of each metric, our new model has:

- Better precision.
- Better recall.
- Better f1-score.

which makes it more efficient compared to the initial one.

## 2 Sources

- https://towardsdatascience.com/methods-for-dealing-with-imbalanced-data-5b761be45a18
- https://optuna.org/#code_PyTorch
- https://deepdatascience.wordpress.com/2020/02/27/log-softmax-vs-softmax/
- https://www.machinecurve.com/index.php/2021/07/19/how-to-use-pytorch-loss-functions/
- https://ruder.io/optimizing-gradient-descent/index.html
- https://rohanvarma.me/Loss-Functions/
- https://cs231n.github.io/neural-networks-1/