Deep Learning

FNN & RNN for Binary- and Multiclass Classification

- Marri Bharadwaj (B21CS045)

BINARY CLASSIFICATION

Dataset

```
                                              review  class
0  People expect no less than brilliant when Stev...      1
1  I just saw "Valentine" and I have to say that ...      1
2  I grew up in Brazil and I used to visit and ma...      1
3  I am not saying that Night of the Twisters was...      0
4  Whoever made this nonsense completely missed t...      0
```

Input Features, Tokenisation and One-Hot Encoding

I began by setting a seed and made PyTorch's *cudnn* backend deterministic for consistent results. I used the *SpaCy* English tokeniser to tokenise the text data, ensuring that words with a frequency of less than five were replaced with the special token *UNK*. This tokenisation ensures that rare words do not negatively impact the model. During testing, any word not found in the vocabulary is automatically replaced with *UNK*. For padding, I added a *PAD* token to handle sentences shorter than the average length of the corpus, while longer sentences were truncated. I implemented one-hot encoding of the tokens. For example, each word in the vocabulary was represented as a one-hot vector, where the length of the vector corresponds to the vocabulary size. This means that if a sentence has eight unique words, each word is encoded as an 8-dimensional vector with a 1 in the position corresponding to the word and 0s elsewhere.

I used *PyTorch*'s *TabularDataset.splits()* to create the training, validation, and test datasets, reading from the preprocessed *.csv* files. The datasets are then tokenized, encoded, and batched using iterators, ensuring that the sequences are padded to a fixed length (based on the average length of the corpus). I also limited the vocabulary to the top 1,000 most frequent words, and any word appearing fewer than five times was handled as

UNK. This step is crucial for controlling the model's complexity and preventing overfitting on rare words. The final vocabulary and label fields were built from the training data, and the data was loaded into batches for efficient processing during training and evaluation.

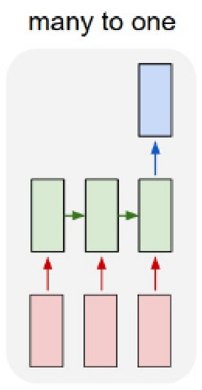
```
       The model has 322,817 trainable parameters.
```

```
Number of training examples is 20000
Number of validation examples is 2500
Number of testing examples is 2500

Number of nique tokens in Sentence vocabulary is 1002
Number of unique tokens in label vocabulary is 2
```

RNN for binary classification

Architecture



I used the above architecture for my RNN model. In this RNN model, I implemented a sequence processing network for binary sentiment classification. The architecture consists of an RNN layer followed by a fully connected linear layer:

- *RNN Layer*: This processes sequences of one-hot encoded input vectors, where each word in a sentence is represented as a one-hot vector. The RNN has a hidden size of 256 and processes the input sequence step-by-step. I ensured batch processing was efficient by using `batch_first=True`.
- *One-Hot Encoding*: I implemented a function to convert the tokenized text into one-hot vectors based on the vocabulary size. This is crucial for maintaining a consistent representation of tokens in the input.

- *Packed Sequences*: To efficiently handle variable-length sentences, I used `pack_padded_sequence` to ignore padding during RNN computation, ensuring the model focuses only on meaningful tokens in the sequence.
- *Linear Layer*: The hidden state from the last time step is passed to a fully connected layer, which **outputs a single value** representing the classification result (positive or negative).

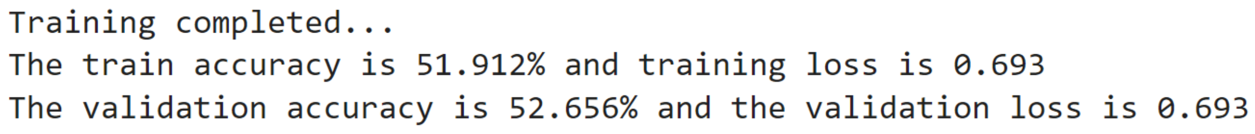
The model I created above uses *SGD* as the optimiser and *BCEWithLogitsLoss* for binary classification. By combining the *RNN*'s ability to capture sequence dependencies and one-hot encoding, my model is well-suited for sentiment analysis tasks.

Training, Validation and Metrics

Training Function: My function performs a single epoch of training. It iterates over batches of data, computes predictions, calculates the loss, and updates the model weights using backpropagation. During each iteration, I also calculated and accumulated the metrics (accuracy, precision, recall, F1-score) for later evaluation.

Evaluation Function: My this function is similar to but runs without updating the model weights. It evaluates the model on the validation set, computing the same metrics as during training. The model is put in evaluation mode using *model.eval()*, and gradient computation is disabled with *torch.no_grad()* to speed up validation.

Training Loop: The model was trained for 10 epochs. After each epoch, I compared the validation loss to the best loss so far, saving the model if it improves. Additionally, I store accuracy, precision, recall, F1-score, and loss for both training and validation sets after each epoch. This setup ensures that I track all key metrics during training, allowing me to monitor the model's performance and choose the best model based on validation loss.

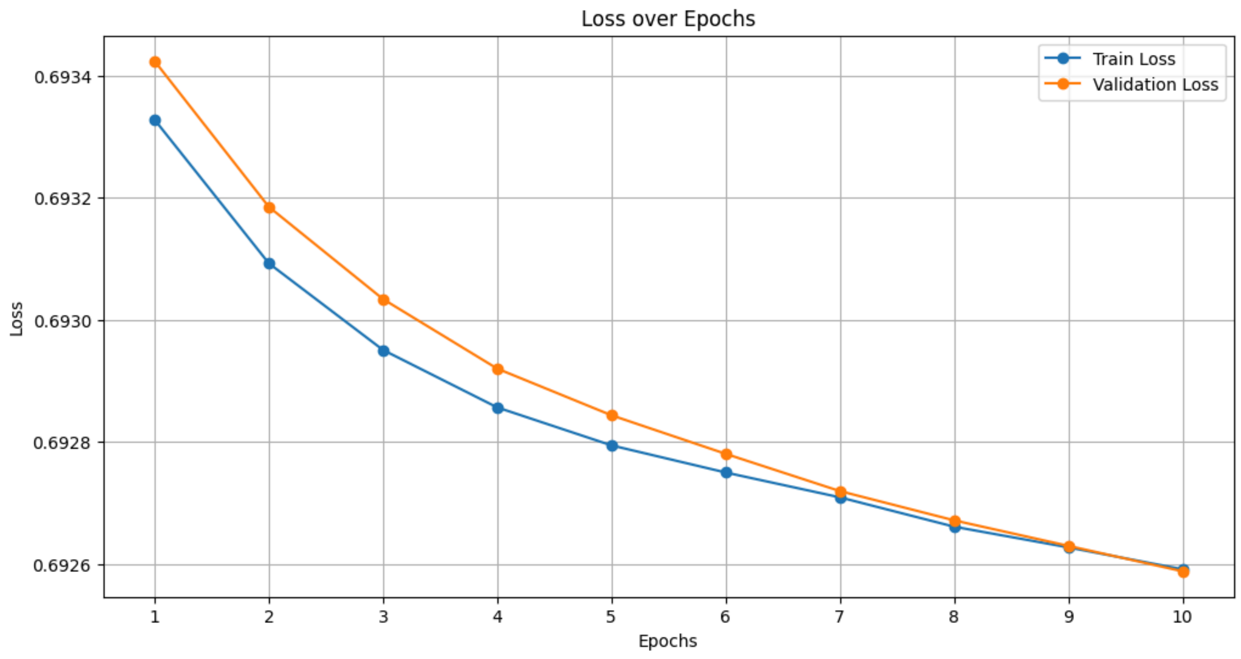


```
Training completed...
The train accuracy is 51.912% and training loss is 0.693
The validation accuracy is 52.656% and the validation loss is 0.693
```

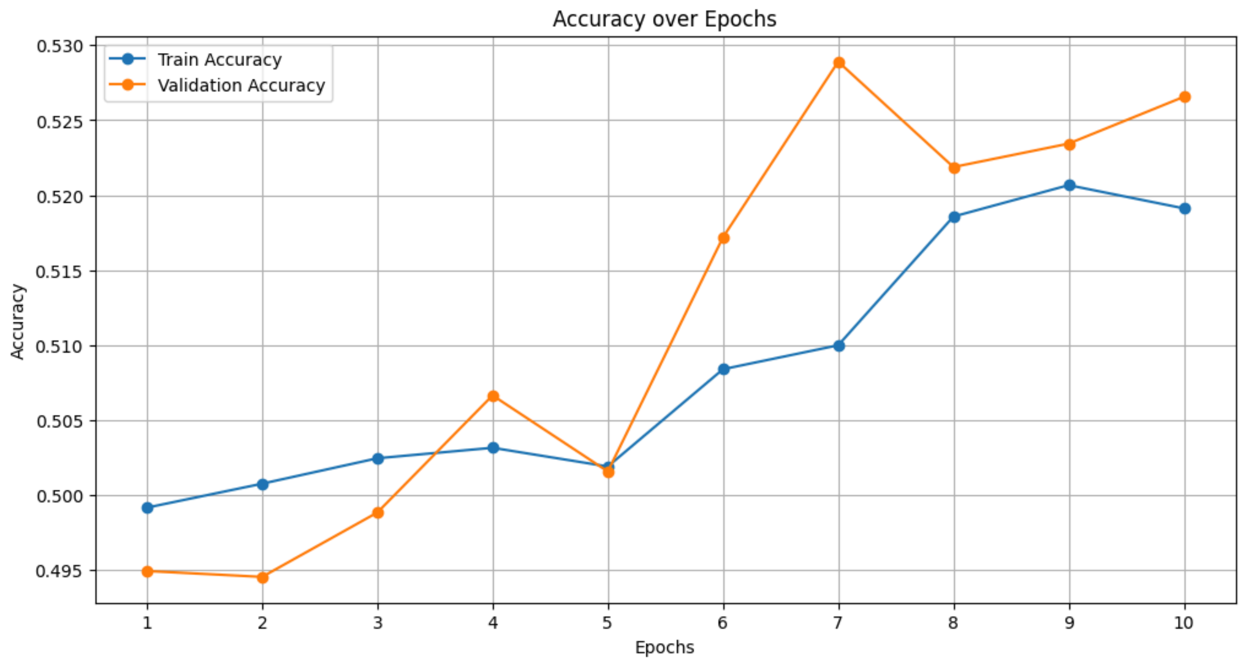
Thus, after training, I received the above metric calculations for my RNN on binary classification.

Testing and Plots

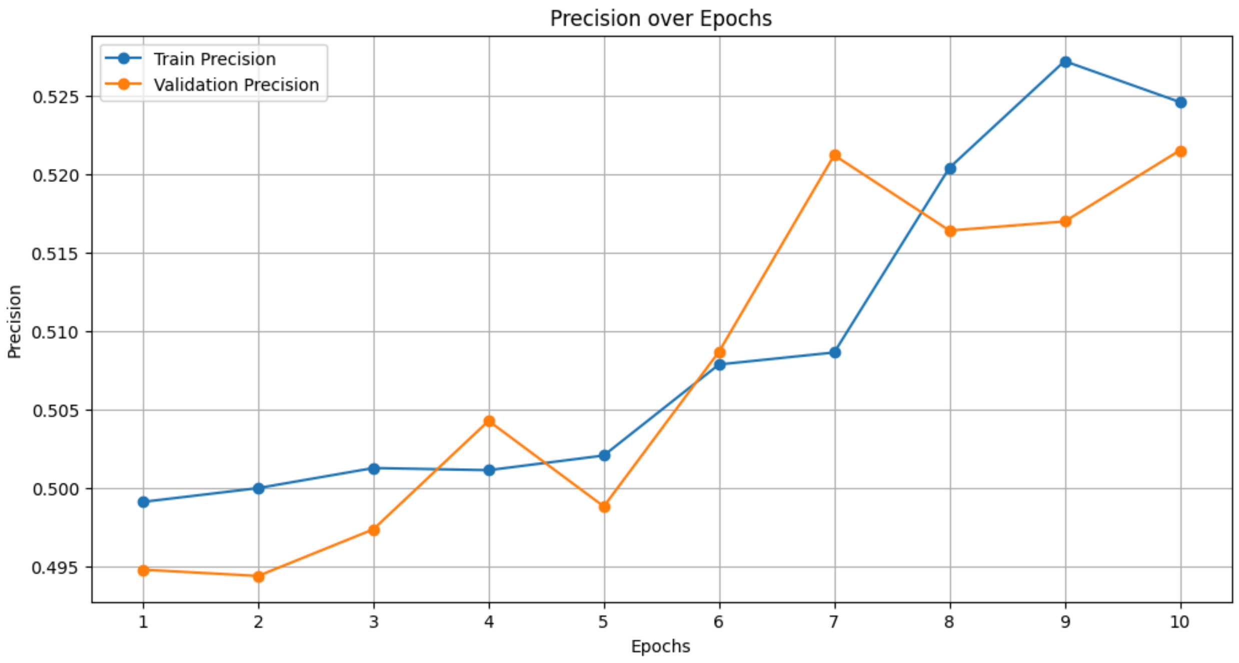
- **Loss** plot



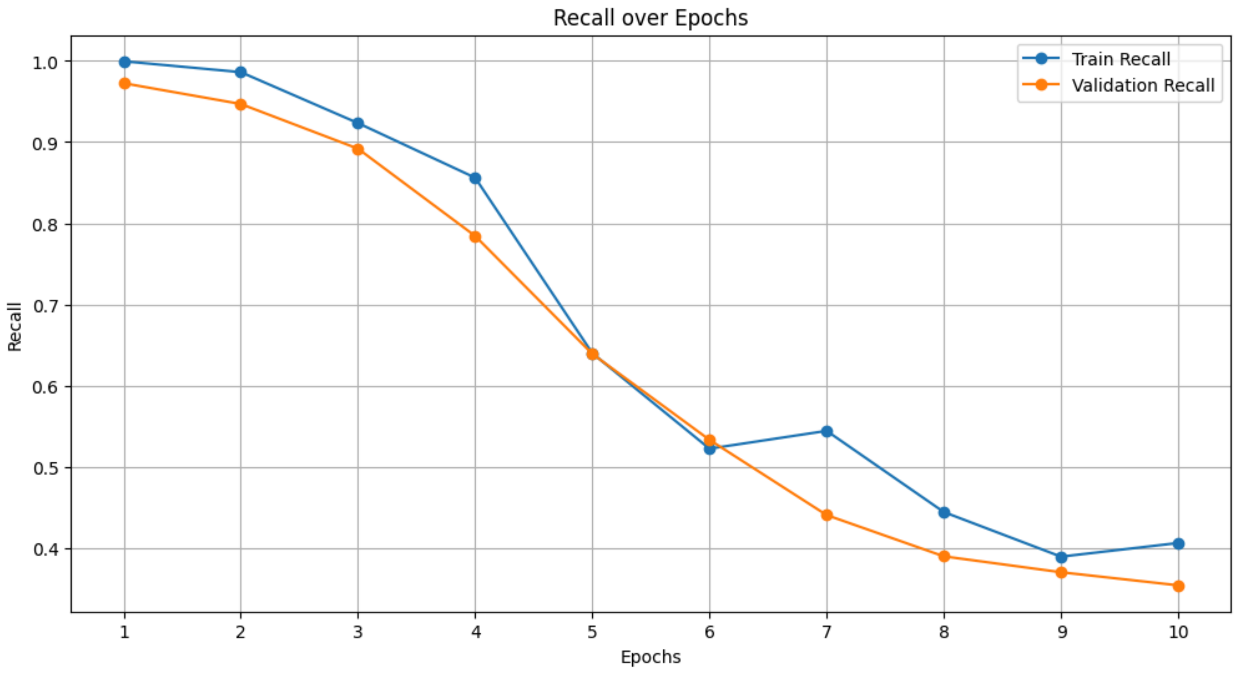
- **Accuracy** plot



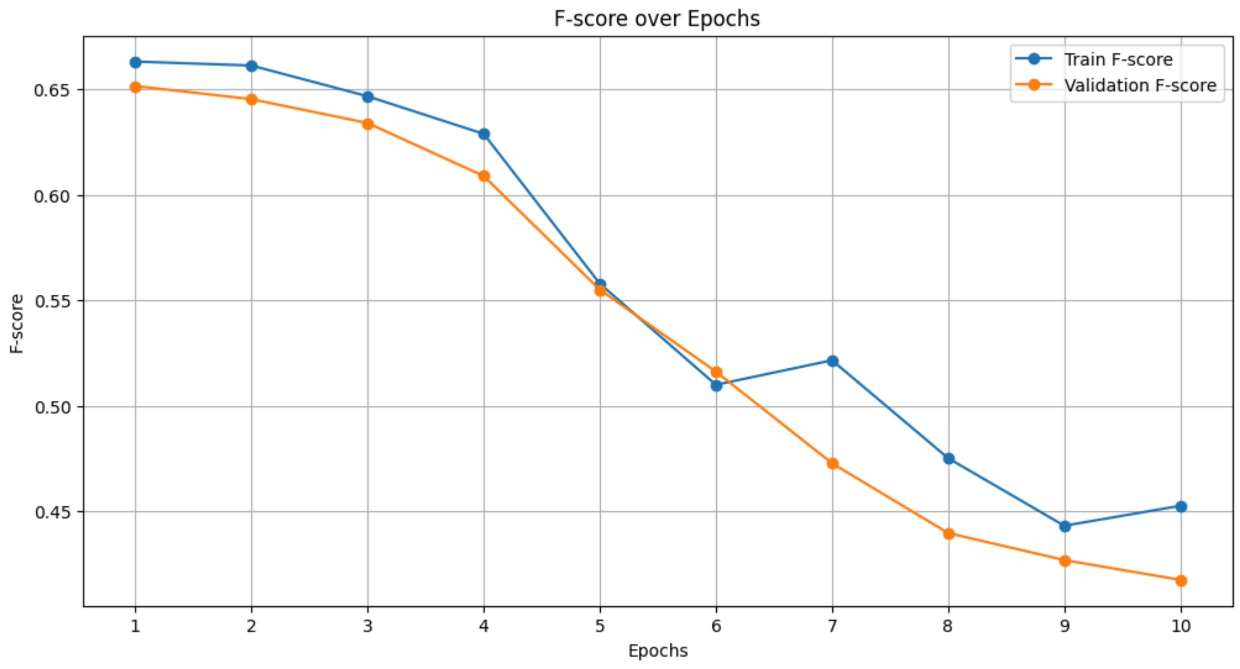
- **Precision** plot



- **Recall** plot



- **F1-Score** plot

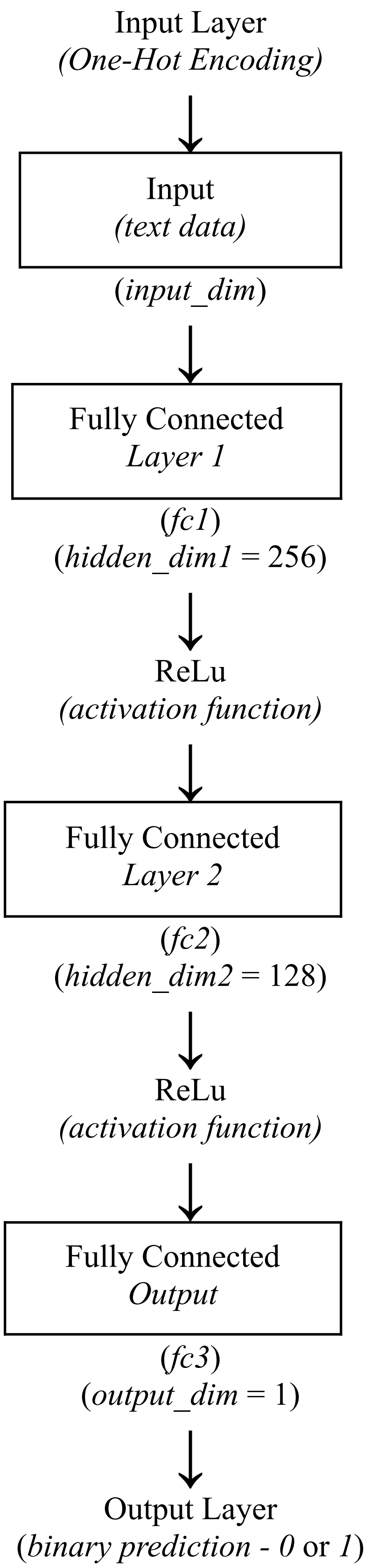


After plotting the metrics, I loaded the best model parameters saved during training using *torch.load*. This ensures that I evaluated the model that achieved the best validation performance. Finally, I evaluated the model on the testing dataset, calculating the test loss and key metrics such as accuracy, precision, recall, and F1-score as display below-

```
Testing done...
The testing accuracy is 53.75 %
The testing loss is 0.692
The testing precision is 0.564
The testing recall is 0.397
The testing F1 score is 0.46
```

FFN for binary classification

Architecture


Input Layer
(One-Hot Encoding)
Input
(text data)
(*input_dim*)
Fully Connected
Layer 1
(*fc1*)
(*hidden_dim1* = 256)
ReLu
(activation function)
Fully Connected
Layer 2
(*fc2*)
(*hidden_dim2* = 128)
ReLu
(activation function)
Fully Connected
Output
(*fc3*)
(*output_dim* = 1)
Output Layer
(*binary prediction - 0* or *1*)

I used the above architecture for my FFN model. It has the following components-

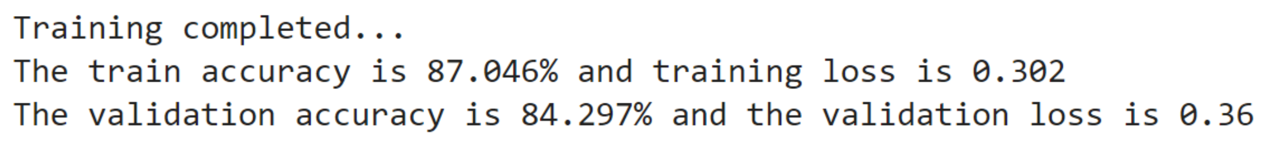
- *Input Layer*: This layer receives the one-hot encoded input vectors of the text data, where the dimension is determined by the vocabulary size.
- *Fully Connected Layer 1*: This layer consists of 256 neurons and transforms the input data into a higher-dimensional space, learning to extract features from the input. The *ReLU* activation function is then applied to this layer!
- *Fully Connected Layer 2*: This layer has 128 neurons and further refines the learned features from the previous layer. The *ReLU* activation function is then applied to this layer!
- *Fully Connected Output Layer*: This layer produces a single output, which represents the predicted sentiment (positive or negative) for the fed text.

Training, Validation and Metrics

In the same way, I implemented the training and evaluation loops for the Feed-Forward Neural Network (FFN) for binary classification. The *train* function handles the training process for each epoch by tracking metrics like loss, accuracy, recall, precision, and F1-score. It processes data in batches, calculates predictions and loss, and updates model parameters via backpropagation.

The *evaluate* function assesses performance on the validation set by calculating loss and metrics without updating the model.

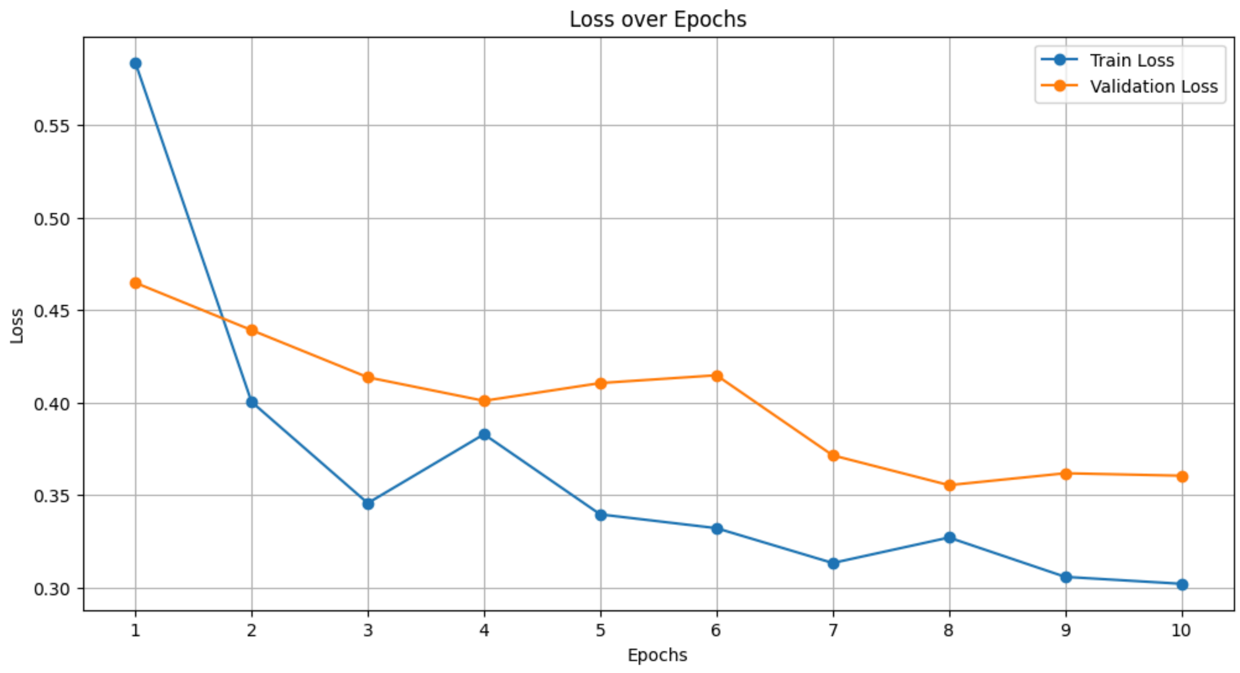
With 10 epochs, the model was trained and evaluated, saving the best model based on validation loss. Metrics, such as accuracy and loss, were stored for analysis, and final results were printed, offering insights into the model's learning and generalisation performance.



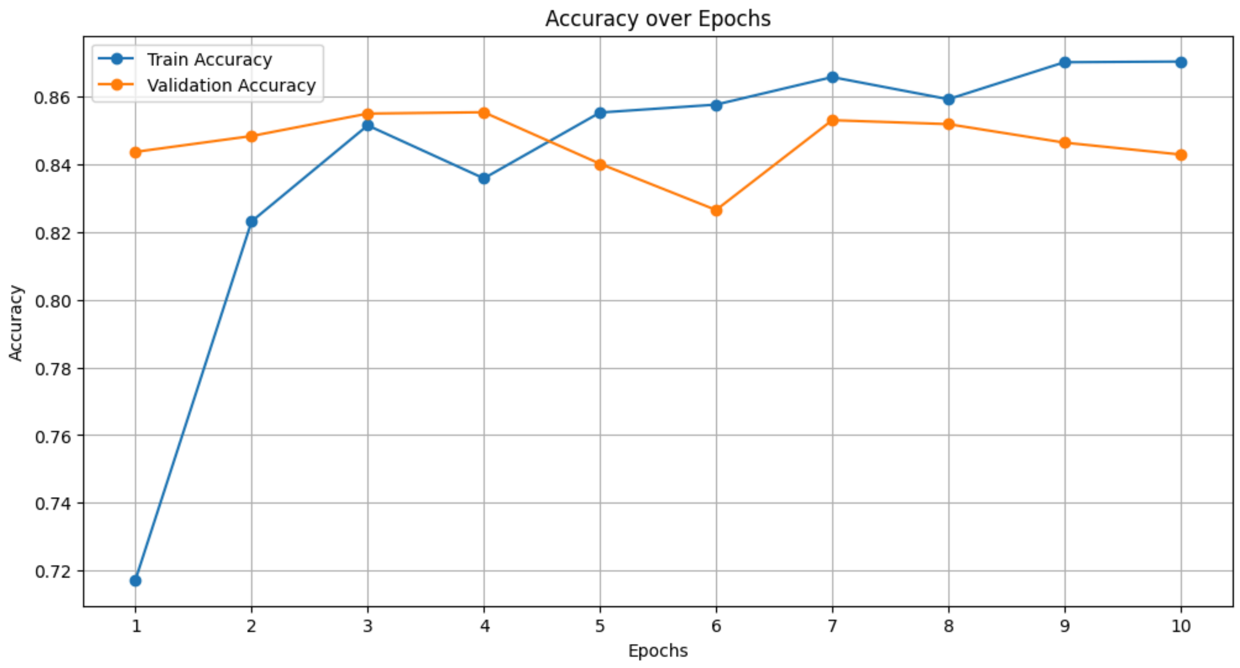
```
Training completed...
The train accuracy is 87.046% and training loss is 0.302
The validation accuracy is 84.297% and the validation loss is 0.36
```

Testing and Plots

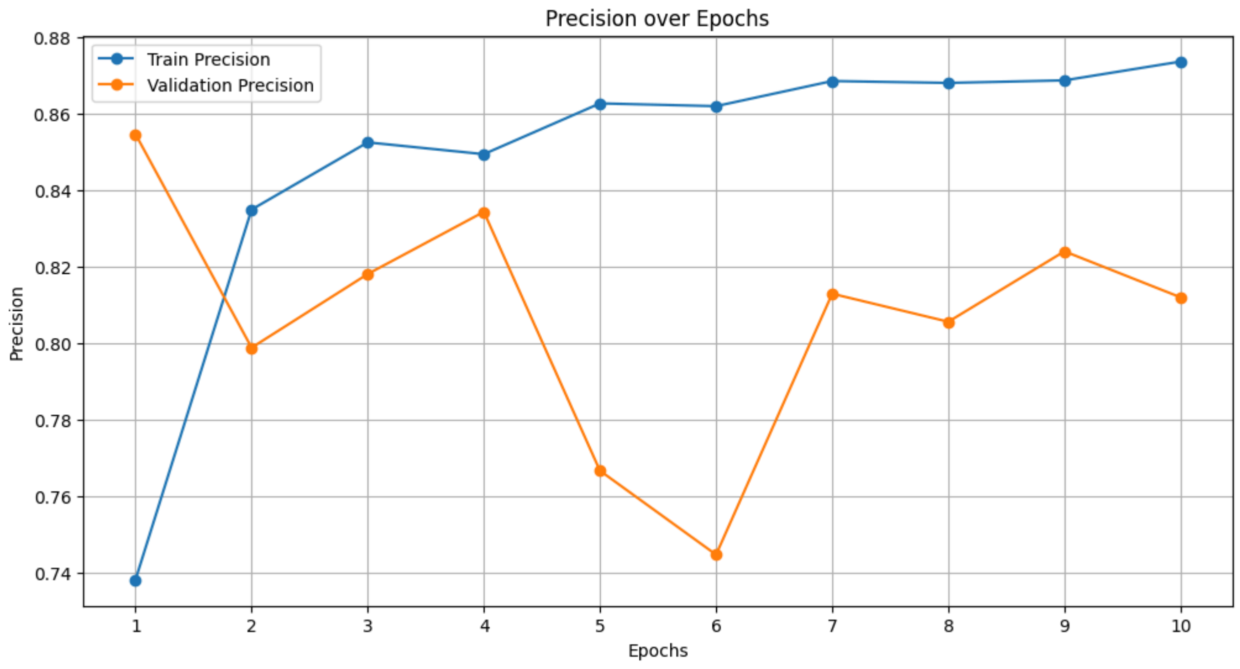
- **Loss** plot



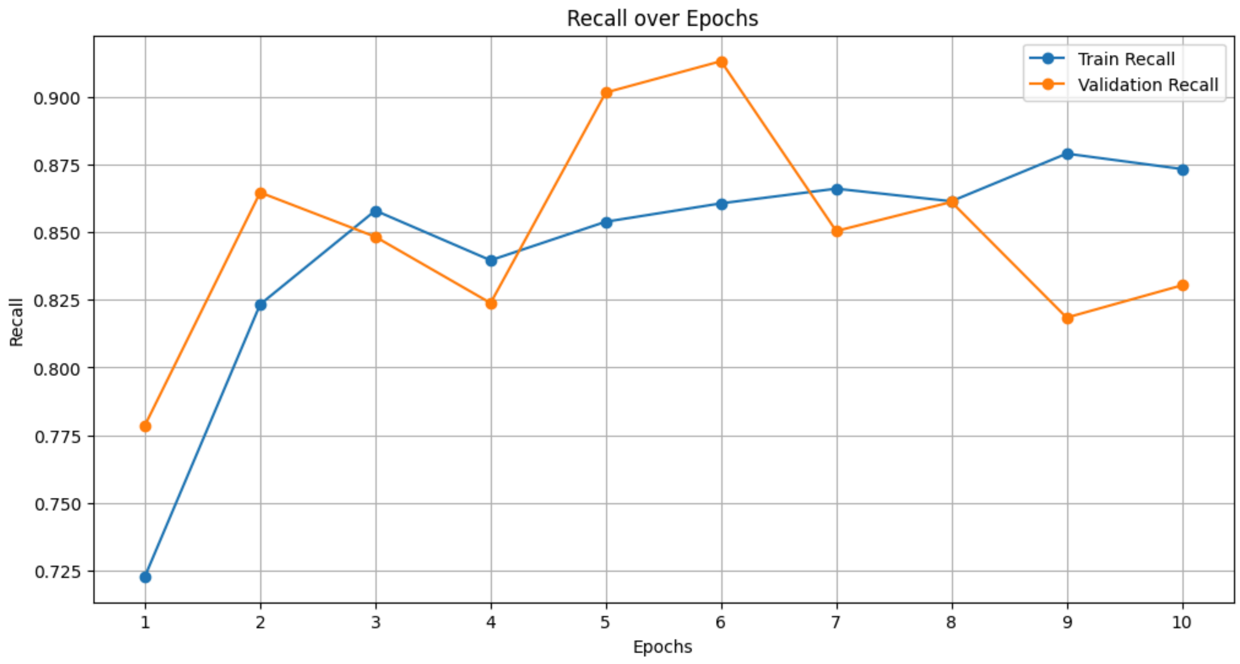
- **Accuracy** plot



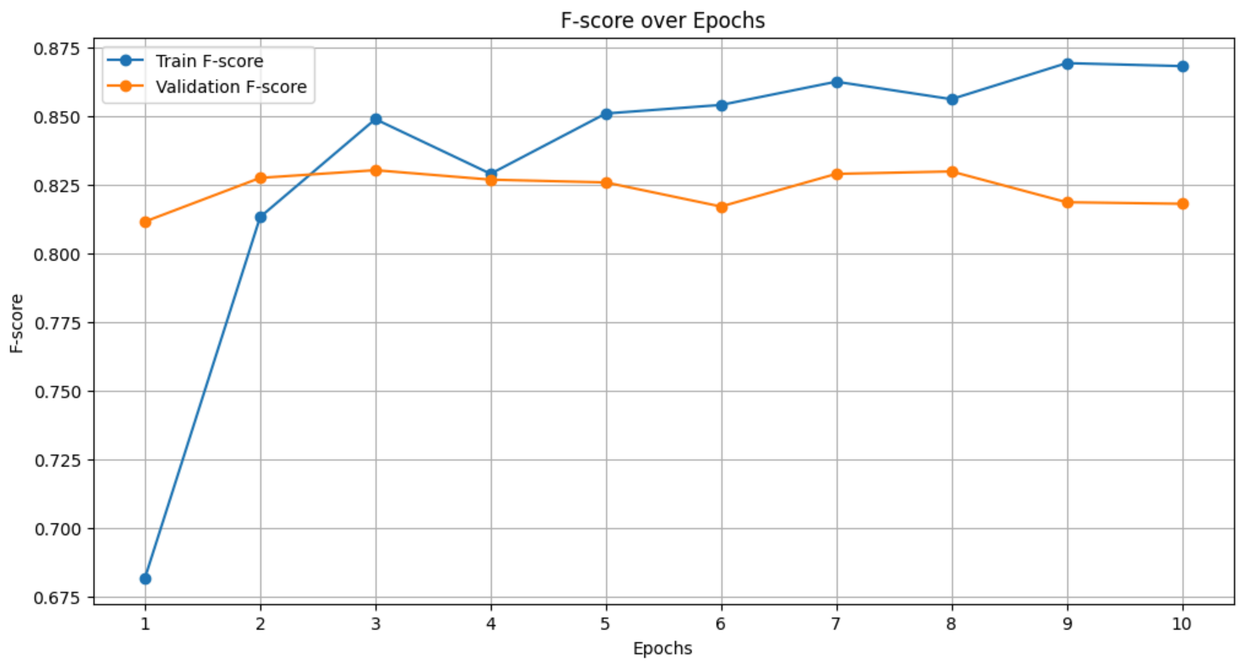
- **Precision** plot



- **Recall** plot



- **F1-Score** plot



```
Testing done...
The testing accuracy is 86.484 %
The testing loss is 0.342
The testing precision is 0.839
The testing recall is 0.901
The testing F1 score is 0.867
```

MULTICLASS CLASSIFICATION

Dataset

```
   label                                               text
0      1    One Night like In Vegas I make dat Nigga Famous
1      1  Walking through Chelsea at this time of day is...
2     -1  looking at the temp outside....hpw did it get ...
3     -1  I'm stuck in London again... :( I don't wanna ...
4      1  It's 2.32 a.m. here in Italy now :) Tomorrow o...
```

Input Features, Tokenisation and One-Hot Encoding

Next, I prepared the input features for the multiclass classification task by implementing tokenisation and one-hot encoding. Using *SpaCy*'s English tokeniser, like earlier, I defined a *tokenise* function that converts raw text input into a list of tokens. This function is crucial as it facilitates the transformation of text data into a format suitable for further processing in the model. Then, I created two fields for the input data: *text* for the input text and *label* for the sentiment labels. The *text* field was configured to use sequential data, build a vocabulary based on the tokenized input, and include sentence lengths with padding for sequences. The *label* field was defined to hold the sentiment labels as floating-point values.

To load the preprocessed training, validation, and test datasets, I utilised *TabularDataset.splits()*, which automatically handles the structure of the data and prepares it for processing in the model. After loading the data, I printed a sample of the first entry and the counts of training, validation, and test examples to verify the successful loading of the datasets.

Eg.- *{'text': ['The', 'earth', 'is', 'run', 'by', 'psychopathsKids', 'could', 'be', 'exposed', ' to', 'this#pornharms', '#', 'draintheswamp', 'https://t.co/m3RUDsWS8s'], 'label': '-1'}*

Next, I built the vocabulary for the *text* field, limiting it to the top 1,000 most frequent tokens while also processing the labels to create a corresponding vocabulary. This step helps manage the model's complexity by ignoring rare words, ensuring a more efficient training process. To facilitate efficient training and evaluation, I created iterators for the training, validation, and test datasets. These iterators were set to batch the data and sort it by text length to minimise padding, allowing shorter sequences to be processed together. This strategy optimises memory usage and computational efficiency during training.

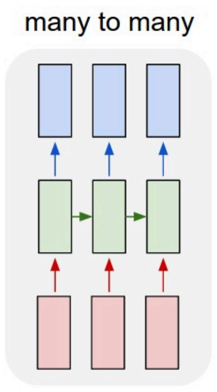
```
The model has 323,331 trainable parameters.
```

```
Number of training examples is 20000
Number of validation examples is 2500
Number of testing examples is 2500

Number of unique tokens in Sentence vocabulary is 1002
Unique tokens in label vocabulary is 3
```

RNN for mutliclass classification

Architecture



I reused the previous *RNN class* here with modifications while declaring the model. I began by defining key parameters for the model, including *input_dimensions*, which corresponds to the size of the vocabulary derived from the tokenised text. I set the *EMBEDDING_DIM* to 1000, indicating that each word in the vocabulary would be represented by a dense vector of this size. The *hidden_dimensions* was set to 256, which determines the number of hidden units in the RNN layer, allowing the model to capture complex patterns in the sequential data. Finally, the *output_dimensions* was specified as 3, reflecting the three possible sentiment classes: positive, negative, and neutral.

Next, I instantiated the RNN model using the defined parameters. To ensure the model's performance could be effectively monitored, I used the earlier *count_parameters* function to count the number of trainable parameters in the model.

For optimisation, I employed Stochastic Gradient Descent (SGD) with a learning rate of 0.001. Additionally, I specified the loss function as *CrossEntropyLoss*, which is suitable for multiclass classification tasks as it combines *softmax* activation and *negative log-likelihood loss*.

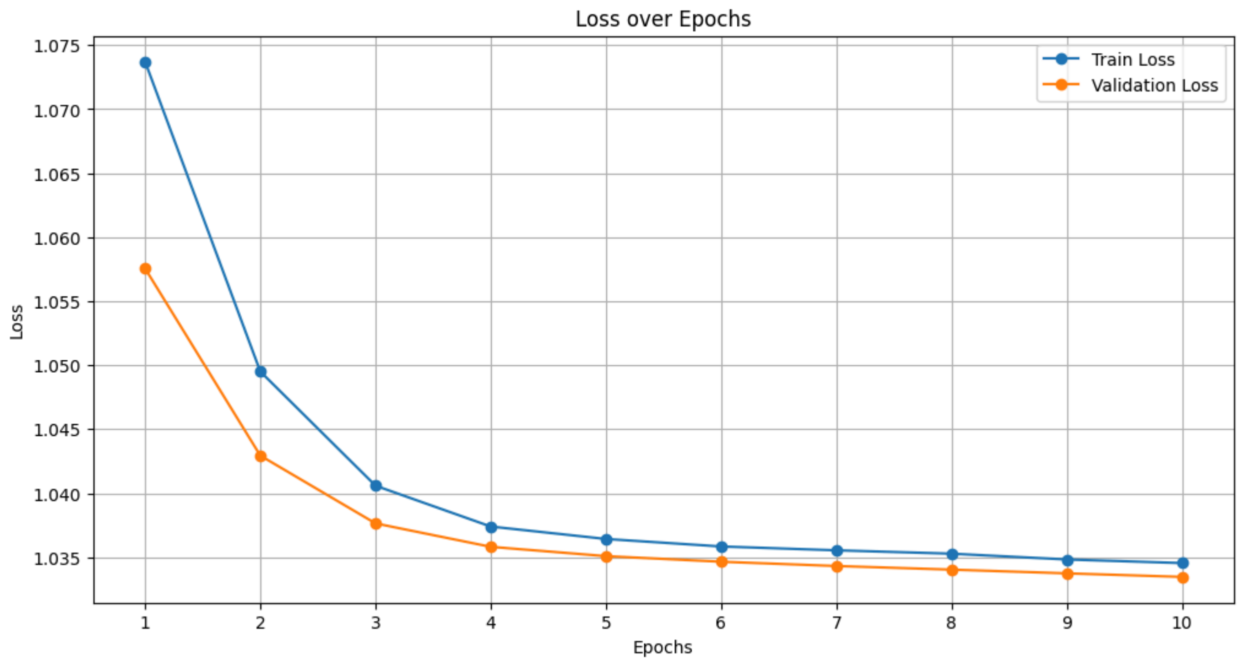
Training, Validation and Metrics

(same way as before)

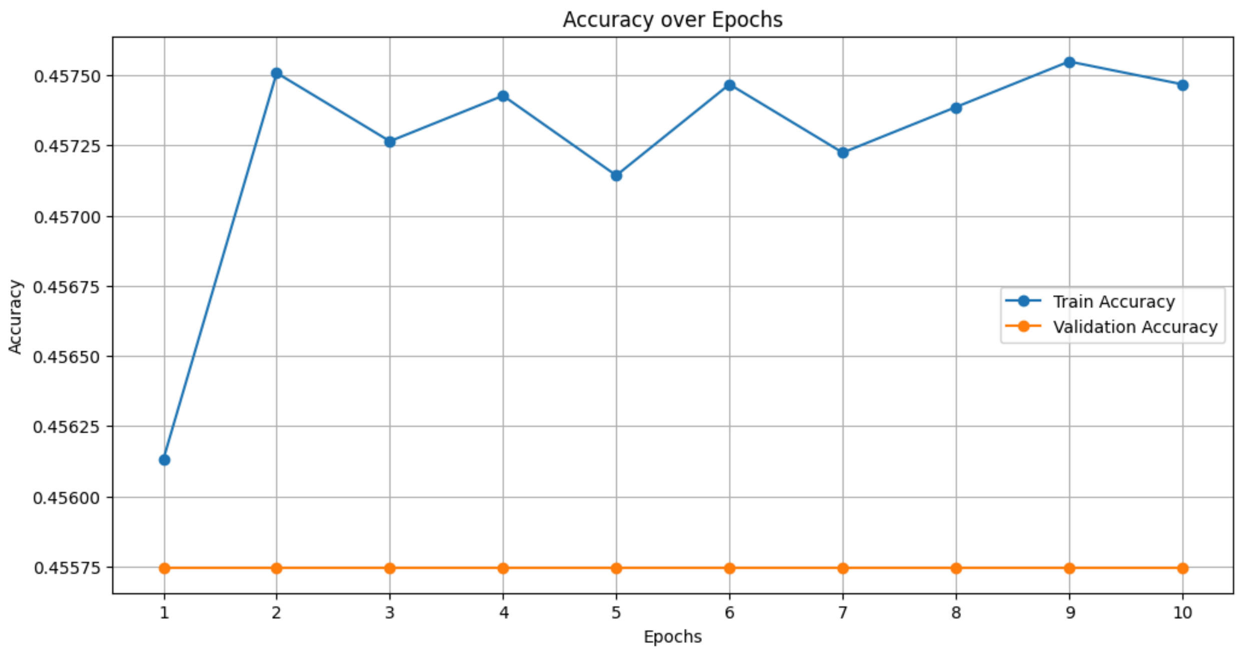
```
Training completed...
The train accuracy is 45.747% and training loss is 1.035
The validation accuracy is 45.575% and the validation loss is 1.033
```

Testing and Plots

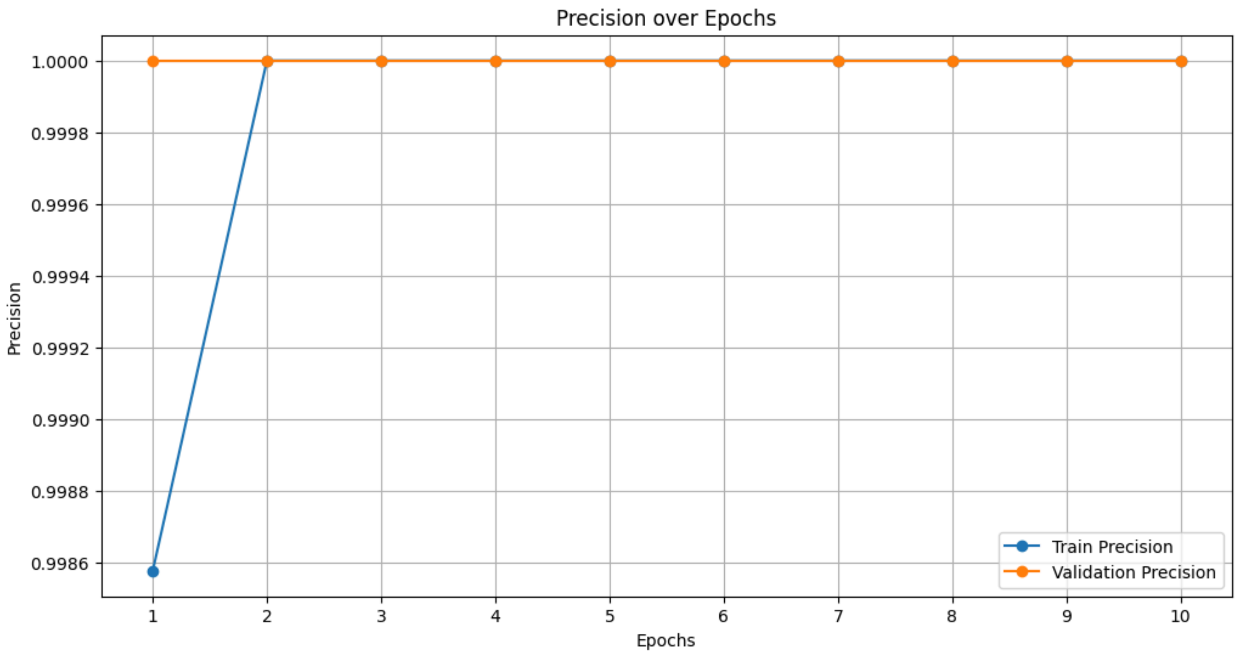
- **Loss** plot



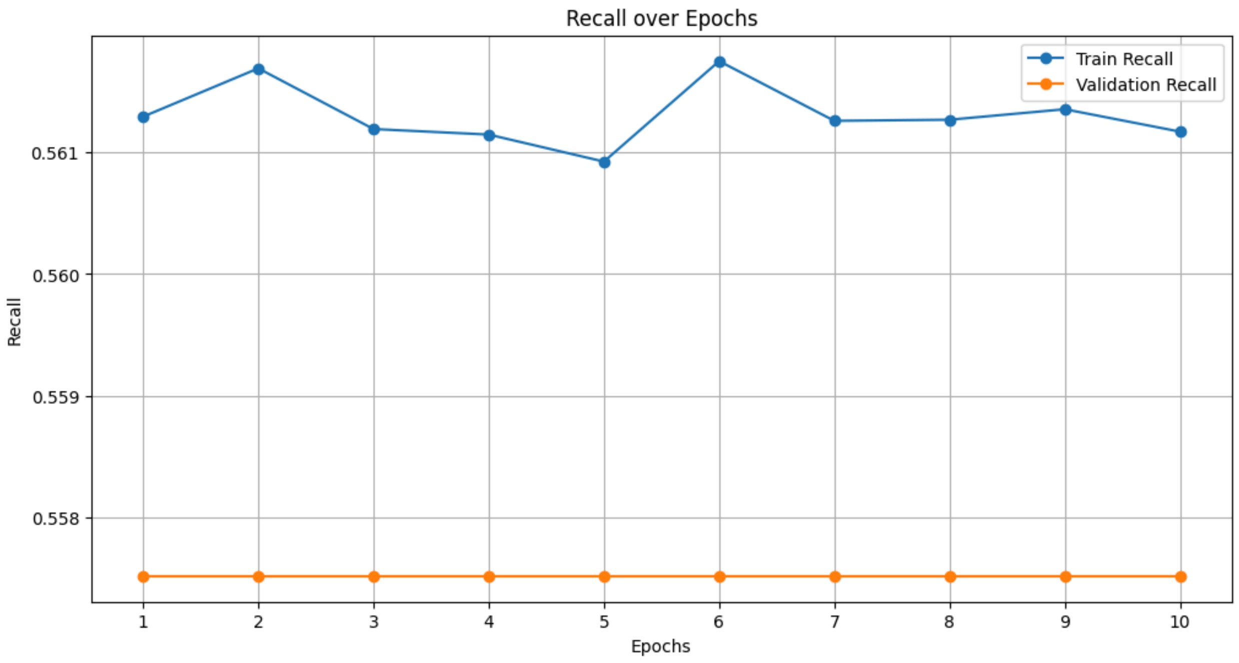
- **Accuracy** plot



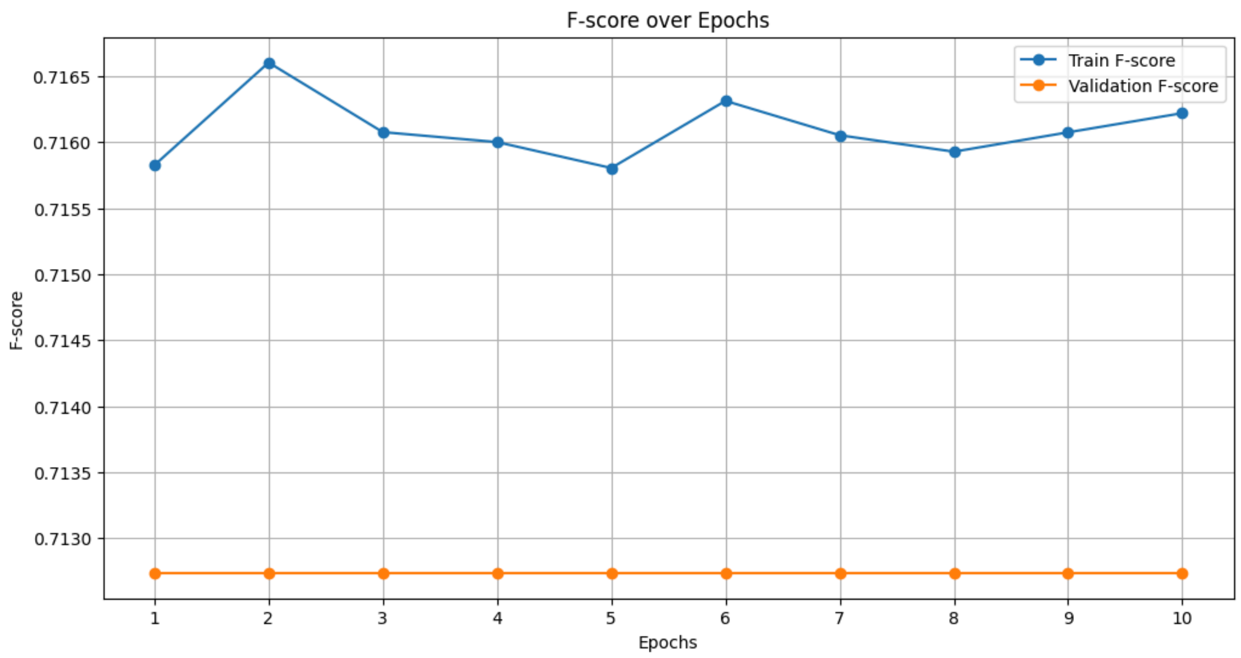
- **Precision** plot



- **Recall** plot

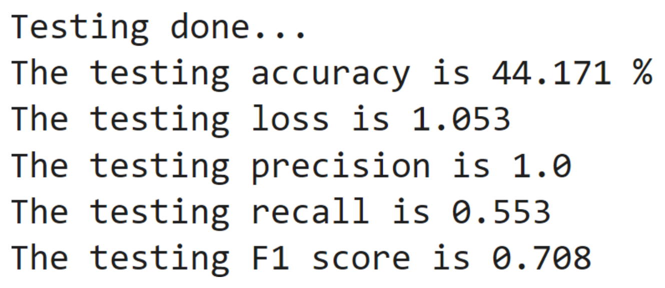


- **F1-Score** plot



In the same way as above, I defined functions for calculating key metrics: accuracy, precision, recall, and F1-score. These metrics help measure how well the model performs, particularly in terms of correctly predicting class labels and balancing precision and recall across classes. The *train* function manages the model's training for one epoch. For each batch, the model generates predictions, computes the loss, and updates its parameters using backpropagation. Metrics like accuracy, precision, recall, and F1-score are calculated for each batch and averaged over the epoch. The function returns the average loss and metrics for the entire epoch.

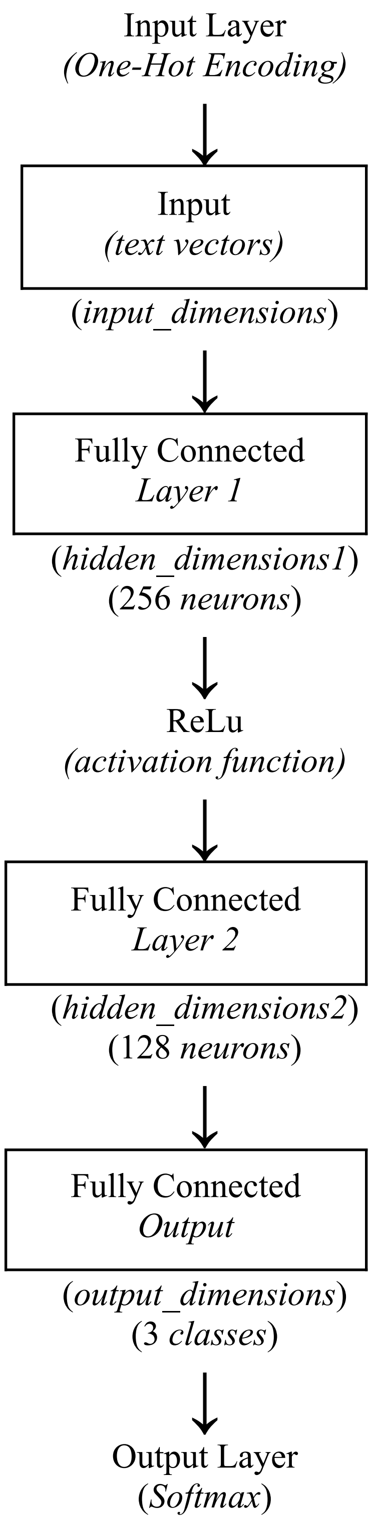
The *evaluate* function is used for model validation. It operates similarly to the training function but does not update model parameters. It calculates the same metrics as in training, giving a snapshot of how well the model generalises to unseen data. The model was trained for a specified number of epochs, with both training and validation metrics being calculated and stored. If the validation loss improved, the model's state was saved. After each epoch, the training and validation metrics, such as accuracy and loss, were printed to track the model's progress.



```
Testing done...
The testing accuracy is 44.171 %
The testing loss is 1.053
The testing precision is 1.0
The testing recall is 0.553
The testing F1 score is 0.708
```

FFN for multiclass classification

Architecture



I reused the previous *FFN class* here with modifications while declaring the model. The components of this FFN are:

1. *Input Layer:* The input layer receives vectors representing the text data, where the dimension of this layer is equal to the vocabulary size (denoted as *input_dimensions*). Each input vector corresponds to a one-hot encoded representation of the tokens in the text.

2. *Hidden Layers*:

   ○ *First Hidden Layer*: This layer consists of 256 neurons (*hidden_dimensions1*), allowing the model to learn complex features from the input data. It transforms the input from the input dimension to a higher-dimensional space.

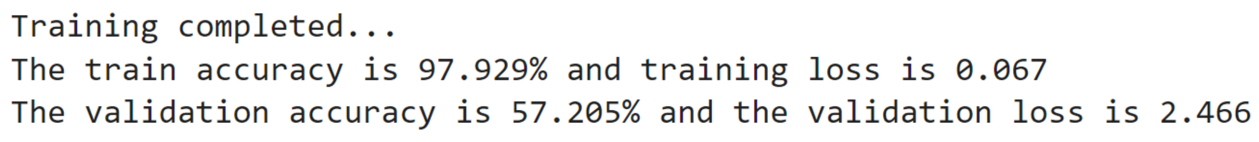
   ○ *Activation Function*: A Rectified Linear Unit (*ReLU*) activation function is applied after the first hidden layer to introduce non-linearity into the model. This helps the network capture complex patterns in the data.

   ○ *Second Hidden Layer*: The second hidden layer has 128 neurons (*hidden_dimensions2*), which further refines the features extracted from the first hidden layer, enabling the model to learn more intricate representations.

3. *Output Layer*: The output layer produces a probability distribution over the classes using the softmax function. The output dimension is set to 3 (*output_dimensions*), corresponding to the three sentiment classes: positive, negative, and neutral.

Training, Validation and Metrics

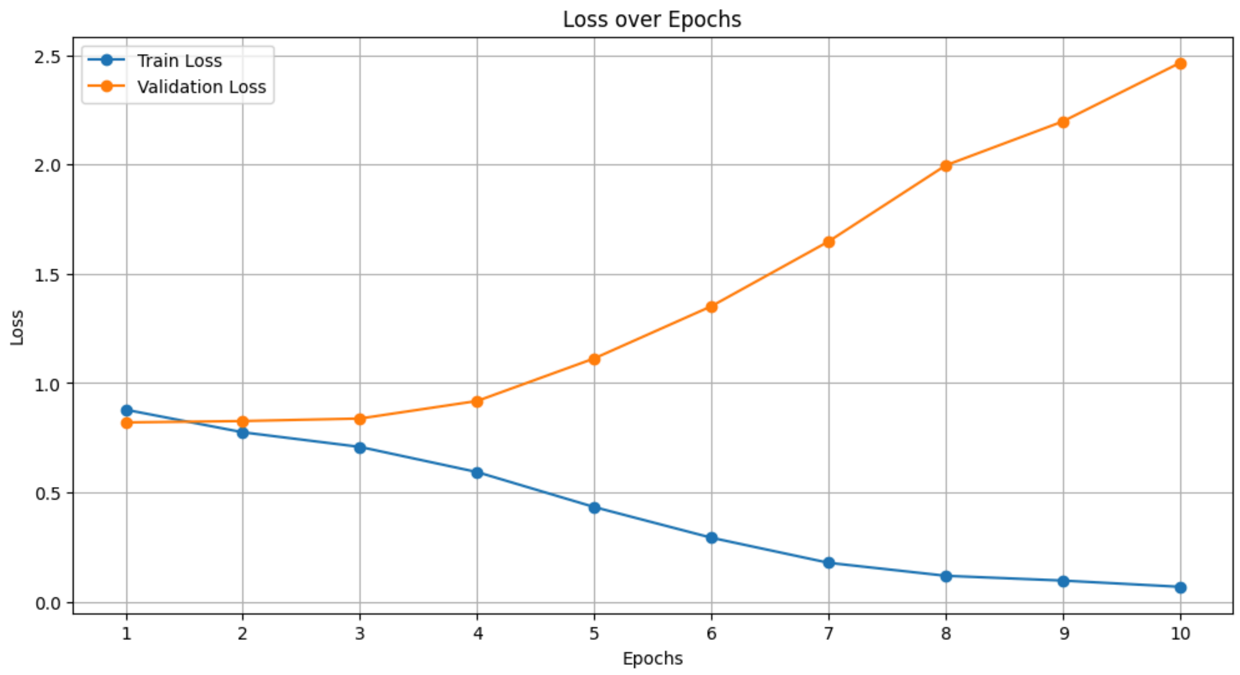
(same way as before)



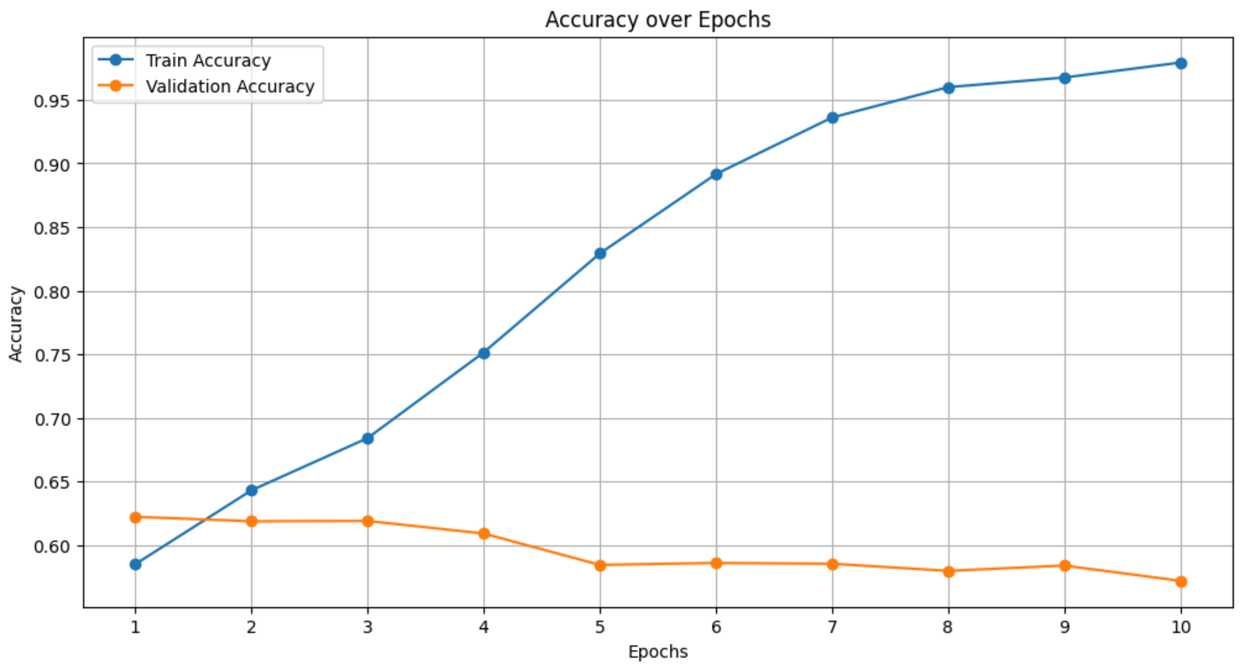
```
Training completed...
The train accuracy is 97.929% and training loss is 0.067
The validation accuracy is 57.205% and the validation loss is 2.466
```

Testing and Plots

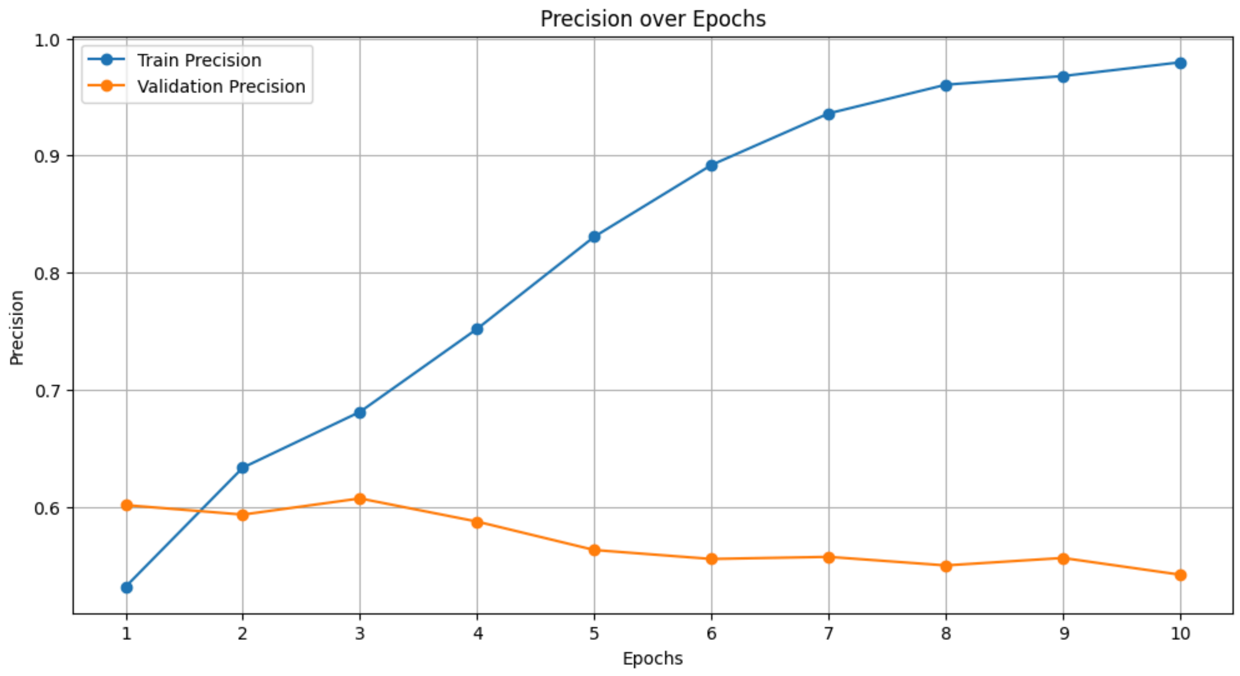
- **Loss** plot



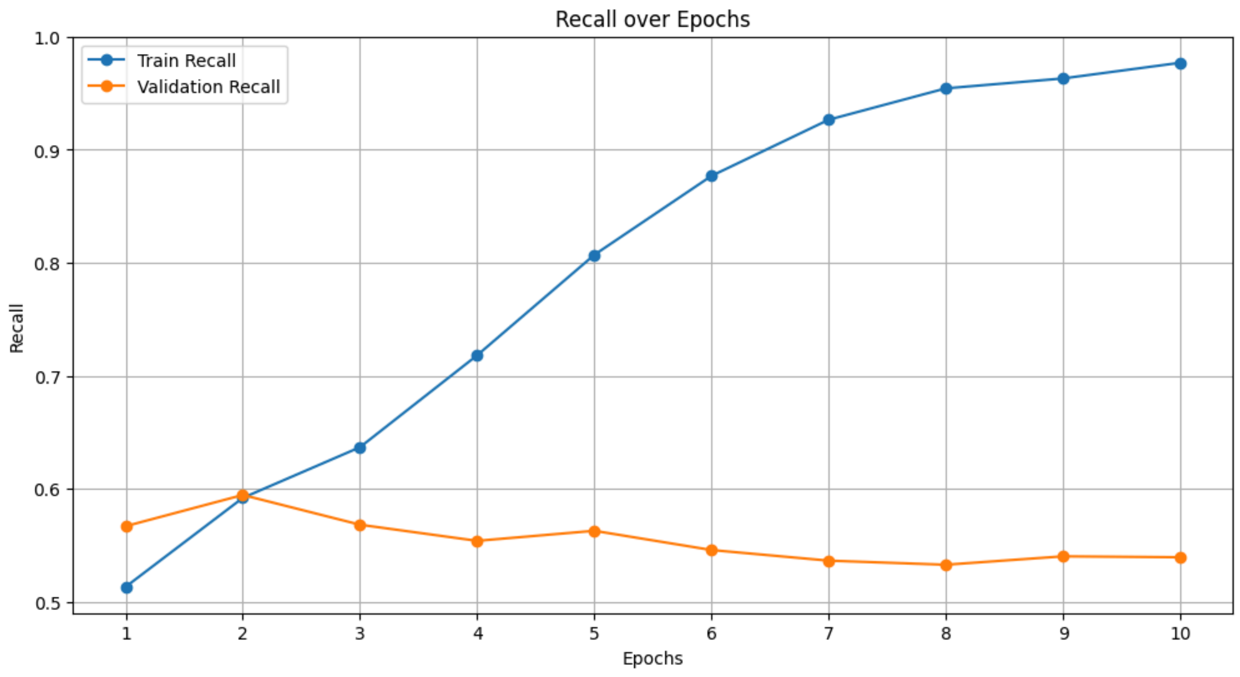
- **Accuracy** plot



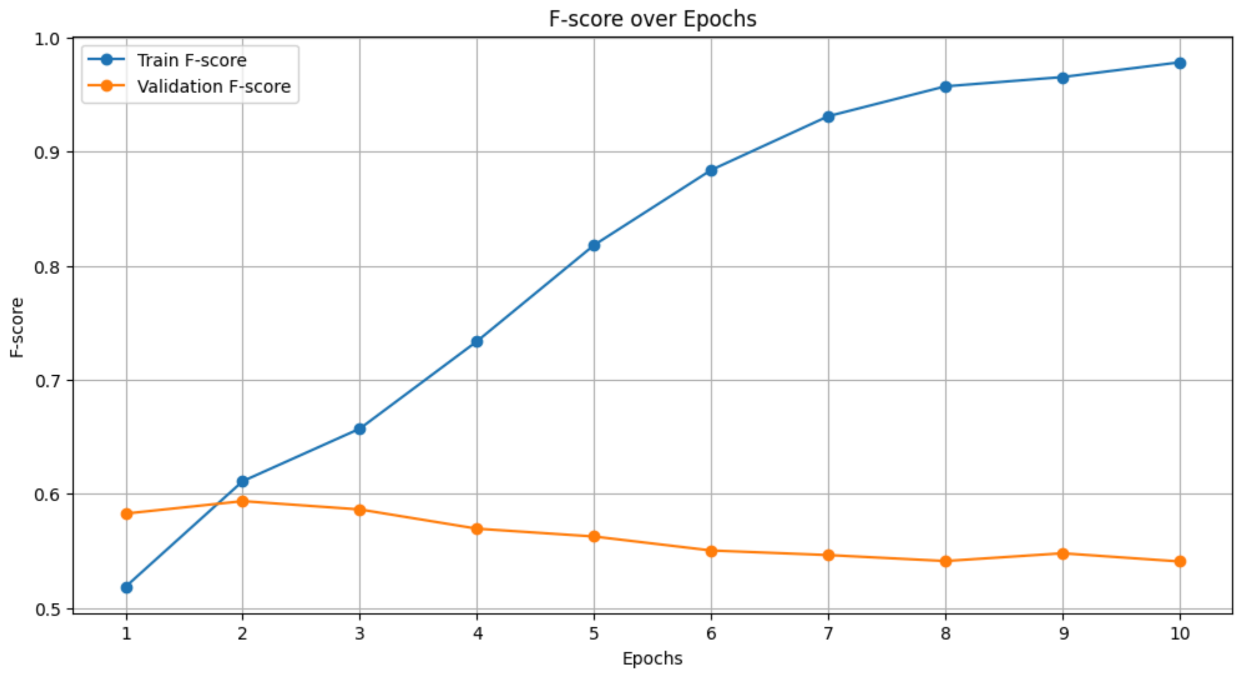
- **Precision** plot



- **Recall** plot



- **F1-Score** plot



In the same way as above, reiterating, I focused on visualising the performance of the Feed-Forward Neural Network (FFN) by plotting the key metrics during training and validation. I plotted graphs for training and validation loss, accuracy, precision, recall, and F1-score across all epochs. These plots helped to assess how well the model was learning and how effectively it was generalising to the validation set. After training, I loaded the best-performing model based on validation loss (saved during the training process) and evaluated it on the test dataset. The model's performance on unseen data was then calculated, providing test metrics such as test loss, test accuracy, precision, recall**, and F1-score. These metrics were printed to offer a final assessment of the model's ability to generalise beyond the training and validation sets. This comprehensive evaluation ensured that the model performed well not only during training but also on real-world data. Concluding, I got the following results for my FFN model on the testing dataset-

```
Testing done...
The testing accuracy is 63.119 %
The testing loss is 0.822
The testing precision is 0.642
The testing recall is 0.582
The testing F1 score is 0.609
```