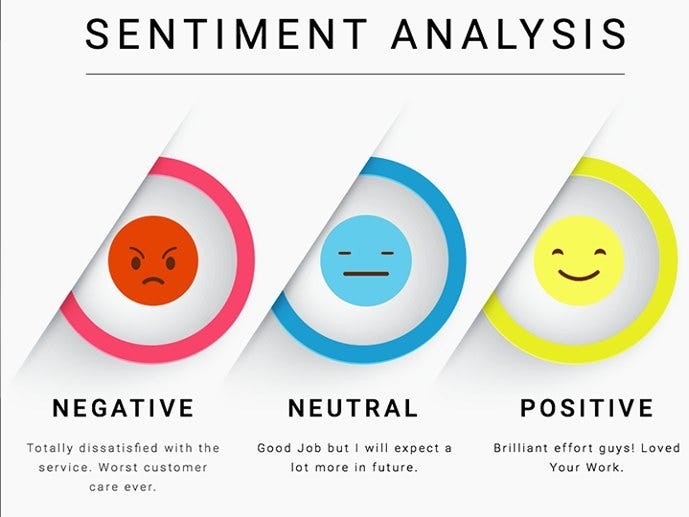


**FFN and RNN for Sentiment Analysis**

**Introduction:**

***Sentiment analysis*** is a fundamental task in natural language processing (NLP) that focuses on identifying and categorizing sentiments expressed in textual data. It plays a vital role in applications such as opinion mining, customer feedback analysis, and social media monitoring.



The goal of this assignment was to implement two DL models, a Feed-Forward Neural Network (FFNN) and a Recurrent Neural Network (RNN), for binary and multi-class sentiment analysis tasks. These models were applied to understand how well they can classify sentiments based on textual inputs.

**Dataset details:**

**1. IMDB Dataset:** The IMDB dataset contains movie reviews labeled as either **positive or negative**, making it suitable for **binary sentiment classification**. This dataset is widely used due to its large volume of text data, providing a variety of writing styles, opinions, and contexts.

* + The dataset includes both testing and training set, total **50k** sentences with their labels, **25k** (12k positive, 12k negative) for training and testing each.
  + Each review varies in length, making it challenging for models to handle both short and long text inputs.

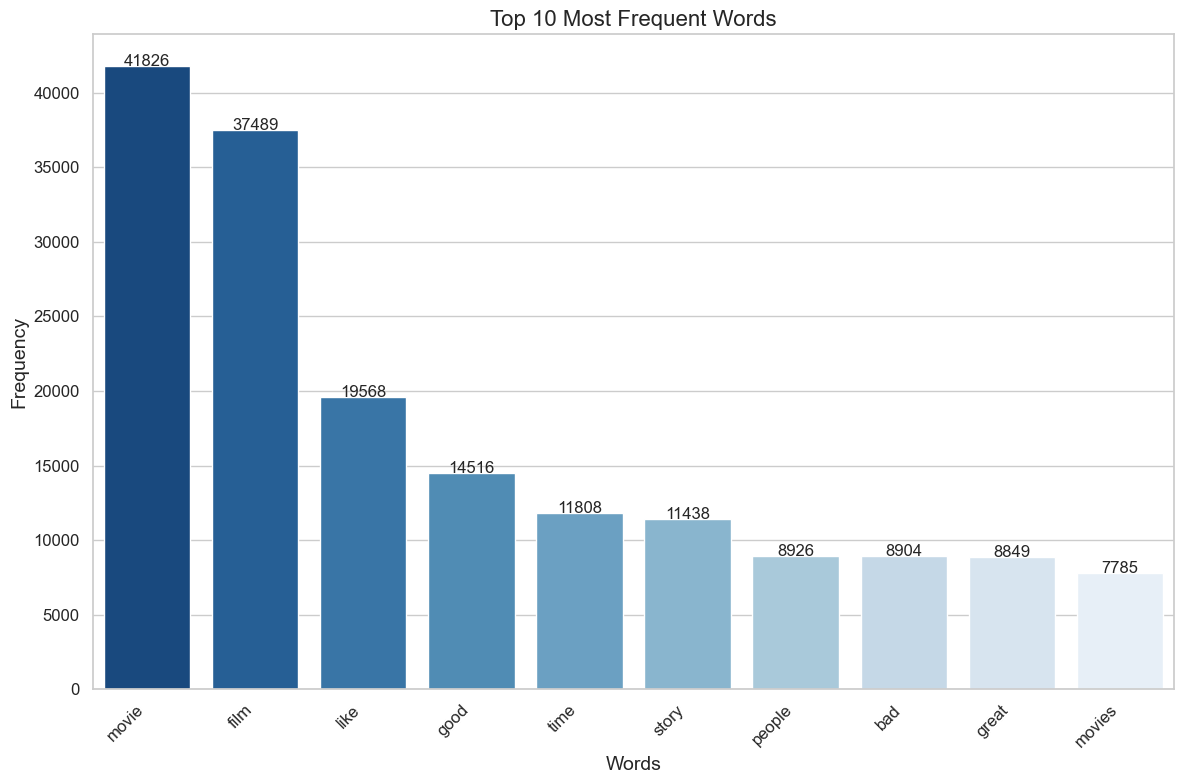
**2. SemEval-2013 Dataset:** The SemEval-2013 Twitter dataset is used for **multi-class sentiment classification**, where tweets are categorized as **positive, negative, or neutral**. Tweets are typically short and contain informal language, adding complexity to the classification task.

* + The dataset consists of short tweets, often containing slang, abbreviations, and emoticons.
  + It is designed for multi-class classification, with tweets labeled into three sentiment classes: positive, negative, and neutral. The distribution of data is **imbalanced**. The dataset has around 10k samples in training set, 1.5k in validation and 3.5k in the test set.

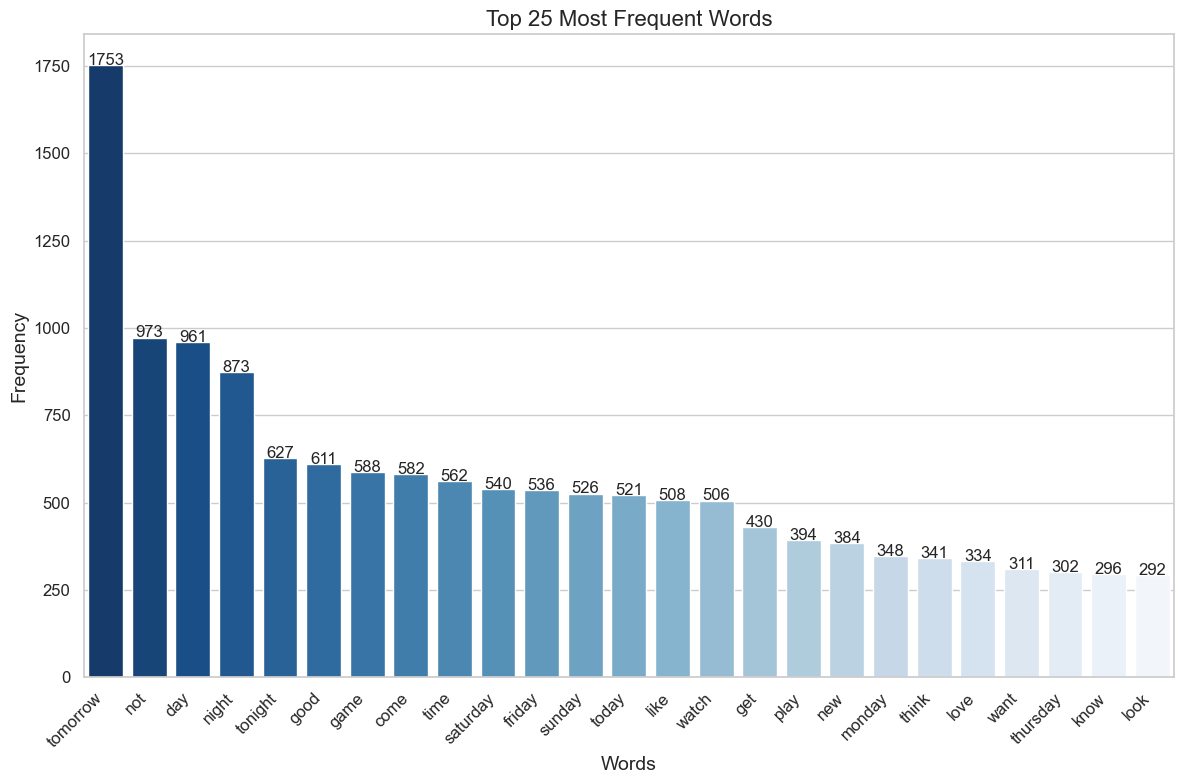
**Preprocessing Methodology:**

the preprocessing steps applied to the IMDB and SemEval datasets to prepare the textual data for sentiment analysis using both a Feed-Forward Neural Network (FFNN) and a Recurrent Neural Network (RNN).

1. **Text Cleaning**:
   * Each review was cleaned by removing **HTML tags** and **non-alphabetic characters** using regular expressions.
   * All text was converted to lowercase to ensure uniformity.
2. **Tokenization**:
   * Tokenization and lemmatization were done using the spaCy library (en\_core\_web\_sm model). During this step:
     + Stop words (common words like "the" or "and") and punctuation were removed.
     + Only alphabetic characters were retained for further analysis.
3. **Building Vocabulary**:
   * After tokenization, a vocabulary was built based on word frequency, with *a minimum frequency threshold of 5 words*. Words that occurred fewer times were discarded to reduce noise.
   * After testing for bigger vocabulary sizes (the kernel crashes for big vocabulary size), the vocabulary size was capped (**17,500** vocab size for **IMDB** dataset**)** to get unique tokens for efficient model training. (+2 for UNK and PAD tokens).
   * Words that were **too short** (1-2 characters), the unwanted tokens were also removed.



Top 10 frequent words in **IMDB dataset.**



Top 25 frequent words in **Semeval dataset.**

* + Special tokens like PAD (for padding shorter sentences) and UNK (for unknown words) were added to handle variable-length inputs and out-of-vocabulary words.

1. **Padding and Truncation**:
   * To ensure all reviews and tweets are of equal length, we calculated the **average sentence length** from the training data.

*Avg sentence length of IMDB dataset:* ***105 tokens***

*Avg sentence length of Semeval dataset:* ***11 tokens***

* + Sentences longer than this average were truncated, and shorter sentences were padded with the **PAD** token to achieve the same length for all inputs.

1. **Data Export**:
   * Preprocessed and tokenized reviews were saved into separate **CSV files** for training and testing datasets, which contained both tokenized and padded versions of the data.

This preprocessing pipeline ensured that the text data was cleaned, tokenized, and padded to a standard format, ready for input into the deep learning models.

**Model Architectures:**

**2. LSTM With Heatmap**

**Architecture Overview**: The model was tested with different LSTM-based architectures to improve sentence-level sentiment classification. We incorporated an attention mechanism to identify which words are most important in determining the sentiment of a sentence. The architecture used is described below:

**Embedding Layer**:

* **Size**: Embedding dimension of 128
* **Purpose**: This layer converts each word in the padded sequence (of max length 105) into a 128-dimensional embedding vector.
* **Variants**: In one variant, the embedding layer was replaced with a simple linear layer of 256 dimensions to evaluate its impact on performance.

**LSTM Layer**:

* **Type**: Long Short-Term Memory (LSTM) layer with an attention mechanism
* **Size**: 256 hidden units
* **Number of Layers**: 1 (no stacking of LSTM layers)
* **Output**: The LSTM processes the entire sequence of 105 words, outputting a hidden state at each time step.

**Attention Mechanism**:

* **Purpose**: After processing the sequence with the LSTM, an attention mechanism is applied to compute attention weights for each time step (word). This enables the model to focus on the most important words for sentiment prediction.
* **Implementation**: The attention mechanism calculates a weight for each word’s hidden state, which is then used to create a weighted sum (context vector). The context vector is passed to the output layer, representing the entire sentence with an emphasis on the key words identified by the attention mechanism.

**Last Hidden State with Attention**:

* **Description**: Instead of only using the last hidden state of the LSTM, we employ the weighted context vector created by the attention mechanism. This vector is an enriched representation that considers the importance of each word in the sequence for sentiment classification.

**Output Layer**:

* **Size**: 3 neurons (for multiclass classification: negative, neutral, and positive)
* **Activation Function**: Softmax is applied to the output layer to generate probabilities for each class, and CrossEntropyLoss is used during training for optimization.

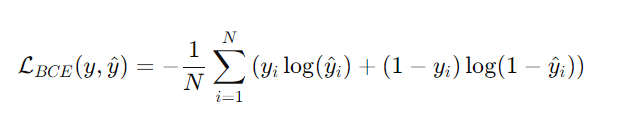
**Attention Visualization with Heatmap**:

* **Heatmap Visualization**: The attention weights for each word in the input sequence are visualized using a heatmap. This allows us to interpret the model's focus within the sentence and understand which words contributed most to the final prediction.
* **Example**: For a sentence such as "The movie had excellent acting but a poor plot," the attention heatmap highlights "excellent," "acting," and "poor," showing that the model focuses on these words when predicting sentiment.
* **Benefit**: The attention heatmap provides insights into the interpretability of the model, enabling a better understanding of the decision-making process and validating that the model is learning relevant patterns in the data.

**Loss Function for Binary Classification (IMDB dataset)**:

For the Feed-Forward Neural Network (FFNN) and RNN we use Binary Cross-Entropy (BCE) loss since the task is **binary classification** (positive or negative sentiment).

The BCE loss is defined as:



Where:

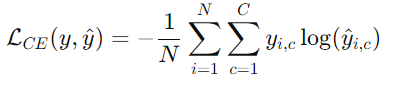
* *yi*​ is the true label (0 or 1).
* *y^i* ​ is the predicted probability of the positive class (output from the Sigmoid function).
* N is the total number of samples.

This loss function works by penalizing the model based on how far its predicted probabilities (y^​i​) are from the true binary labels (yi​). When the prediction is close to the true label, the log term is small, minimizing the loss. If the prediction is far from the true label, the log term becomes large, increasing the loss.

**Loss Function for Multiclass Classification (Semeval Dataset)**:

For the **SemEval dataset**, the task involves **multi-class classification** with three classes (positive, neutral, negative sentiments). The appropriate loss function for this setting is **Cross-Entropy Loss**, which works well for multi-class classification problems.

The **Cross-Entropy Loss** is defined as:



Where:

* yi (c)​ is the true label for class c (encoded as a one-hot vector).
* y^​i (c)​ is the predicted probability of class c (output from the Softmax function).
* N is the total number of samples.
* C is the total number of classes (in this case, **C=3**).

In this case, for the SemEval dataset, the three possible classes are:

1. Positive (Class 1)
2. Neutral (Class 2)
3. Negative (Class 0)

The **Softmax function** ensures that the output probabilities sum to 1 across the three classes, distributing the probability mass across the possible sentiment categories.

Cross-Entropy Loss calculates how far the predicted probabilities ***(y^​i,c​)*** are from the true one-hot encoded labels ***(yi,c​)*** and penalizes incorrect predictions accordingly.

This loss is particularly effective for **multi-class classification** because it encourages the model to not only assign a high probability to the correct class but also to reduce the probabilities for the incorrect classes.

**Note:**

Both architectures use non-linearities (ReLU in FFNN and tanh in RNN). Since in initial testing, the model did overfit the model so I included **dropout** and **early stopping** to reduce overfitting, while the choice of loss functions ensures that the models can handle binary classification effectively.

**Implementation:**

A variety of ways were tried to get the best model for each of the datasets by varying the features of the datasets and the models.

**1. Loading the Data:**

**1.1 Custom Dataset Class**

The IMDB dataset, which has been preprocessed and saved into CSV files. A custom SentimentDataset class is created, inheriting from torch.utils.data.Dataset. This class:

* Loads the padded reviews and corresponding labels from the dataset.
* Clamps word indices to ensure they do not exceed the specified vocabulary size (17,502).
* Converts each word index into a **one-hot encoded vector** for the FFNN model, and returns the processed review and label when accessed.

**1.2 DataLoader Creation**

PyTorch DataLoader objects are created for batching and iterating through the training, validation (dev), and test datasets. Each DataLoader batches data into smaller chunks (batch size = 16) to make the training process more efficient. The number of batches is printed for each dataset.

**2 Training the LSTM**

The LSTM is trained using the **Adam optimizer** and **Binary Cross-Entropy Loss** (BCELoss()), which is suited for binary classification. The training process:

* Iterates over multiple epochs. (no of epoch = **10**).
* Computes the training loss and accuracy after every batch.
* At the end of each epoch, validation loss and accuracy are calculated.

**Early Stopping and Validation**

To prevent overfitting, **early stopping** is implemented. If the validation loss does not improve for **3** consecutive epochs, training halts, and the best-performing model (based on validation loss) is restored. This ensures the model does not overfit to the training data.

**LSTM Evaluation and Plotting Results**

Once the FFNN is trained, it is evaluated on the test set using metrics such as **accuracy**, **precision**, **recall**, and **F1-score**.

The model's performance is printed, including class-wise metrics for positive and negative sentiments. Training and validation losses, as well as accuracies, are plotted using matplotlib to visualize the model's progress over time.

**Results**

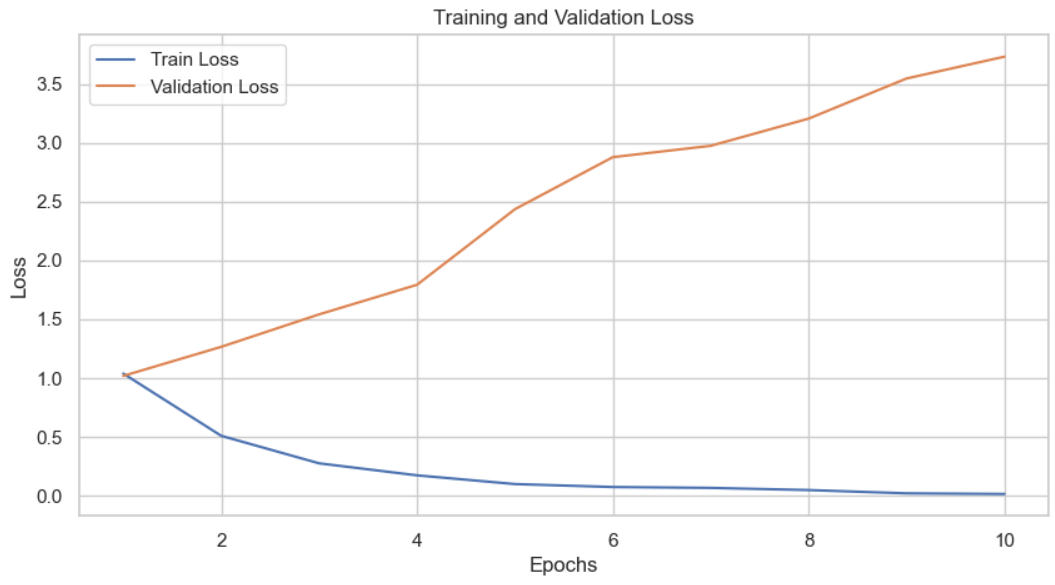
Here are the models with their configurations that were tried to get the best results.

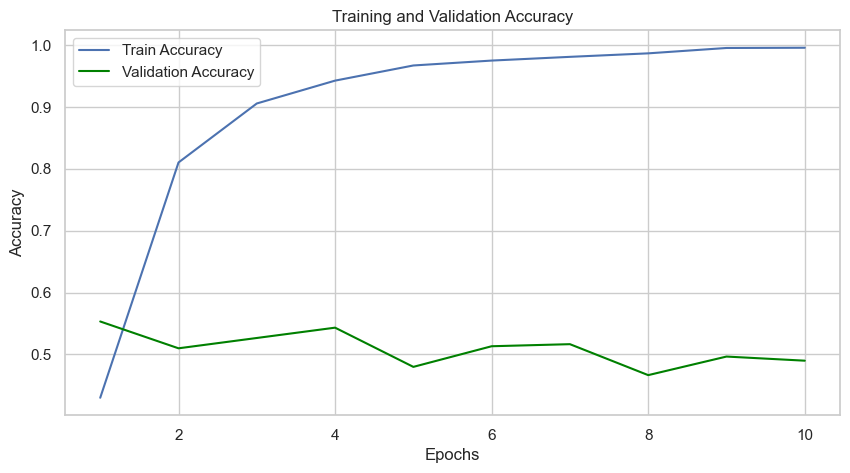
**IMDB dataset:**

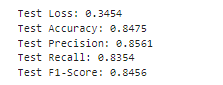
**-> Using subset of full vocabulary and front 100 words with one hot encoding.**

The vocab size was taken as 10k (from 30k) and the input was truncated. Avg sentence length: 268 words reduced to top 100 words (to avoid crashing and memory error).

**(i)** Results for LSTM with attention model:

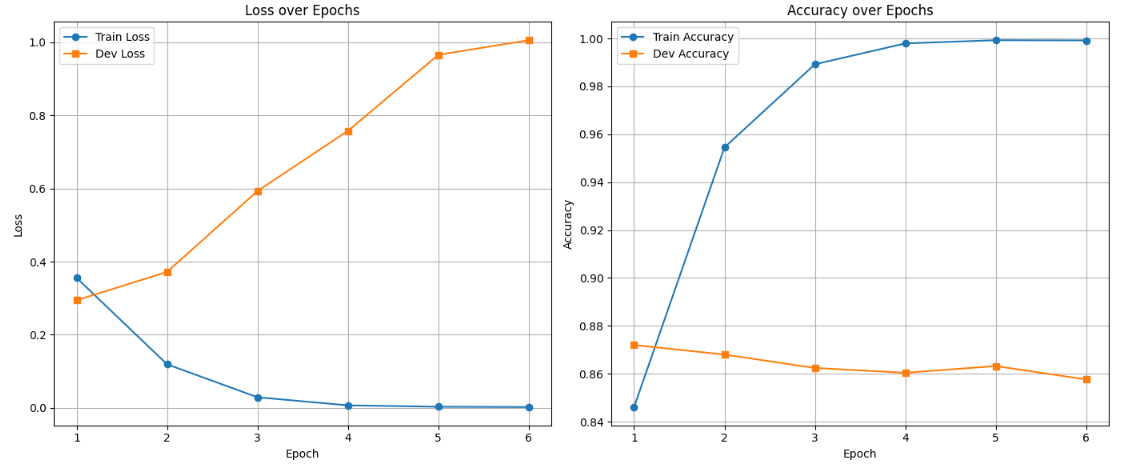
****

**** Training loss vs epoch graph and Dev Accuracy vs epoch graphs

****

Evaluation metrices for the model

(iv) **LSTM** **model with linear layer**

****

Training loss vs epoch graph and Dev Accuracy vs epoch graphs

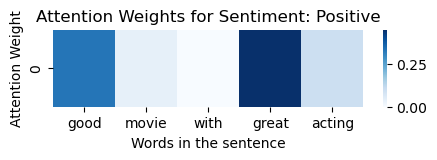
**Example statement:**

****

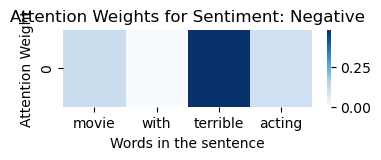
We can see that the model predicts very well on the given statements.

**Visualizing attention:**

sentence = "good movie with great acting"

****

sentence2 = "movie with terrible acting"

****

**Conclusion:**

We can see that using Attention Layer with LSTM’s on the new dataset (with top words) gave the best accuracy of **84.85%** over **25k** samples test set. The model also achieved a high F1-score of **84.66%**, showing a good balance between precision and recall. This means the RNN’s, GRU’s and LSTM’s were effective at understanding the relationships between words in the text, making it the best-performing model for this dataset.

**Semeval dataset:**

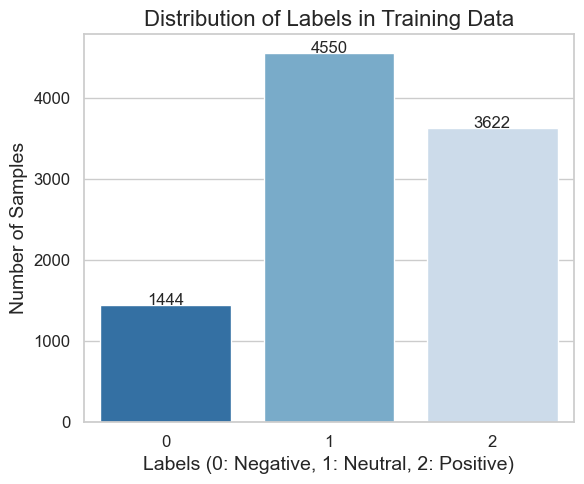
**-> Full vocabulary with one hot encoding.**

The vocab size for this dataset was 3083 and the input was truncated.   
 Avg sentence length of the tweets: 26.

This did not give good results as the accuracy was around 36% only. but it established a baseline for our model. Also, our model did not predict class 0, which implies there may be a **class imbalance** in the dataset**.**

Test accuracy: **0.36**

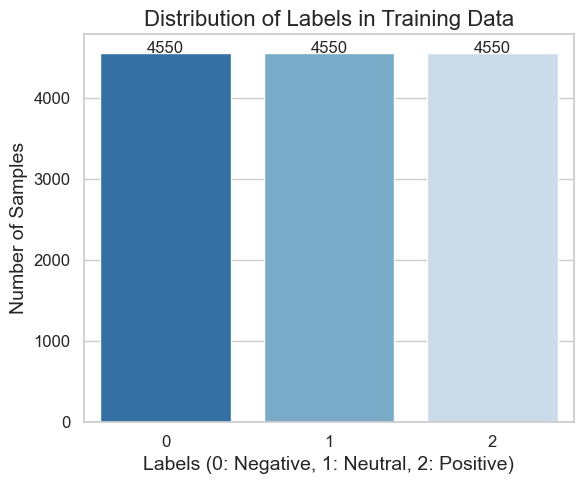
# **Removing Imbalance from the data:**



From the figure, we can clearly see an imbalance in the number of samples of class 0 due to which our model did not give good predictions. Hence I applied two methods for correcting it:

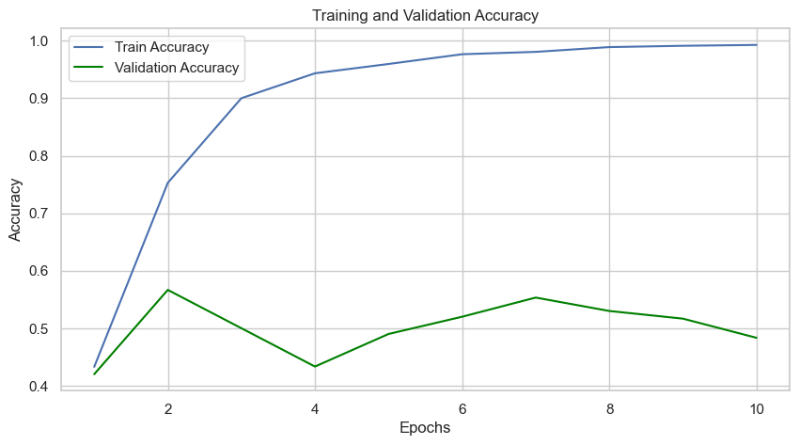
(a) Giving more weights to undersampled class.

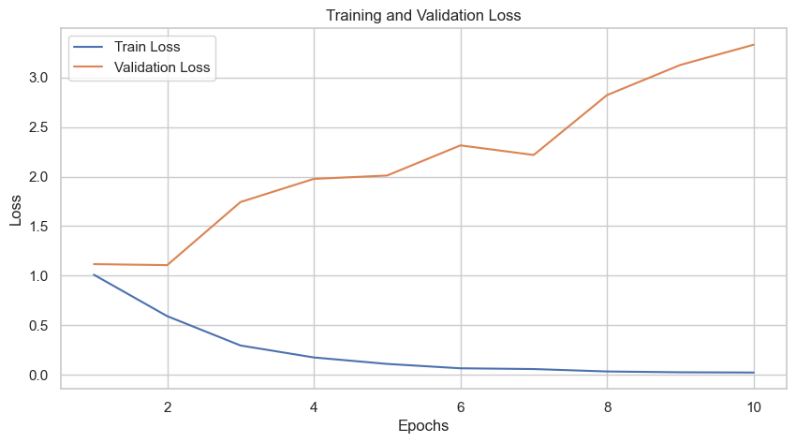
(b) **Oversampling** the class with less samples.



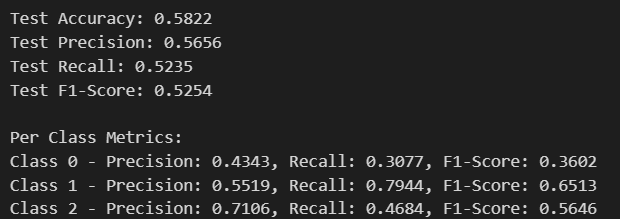
Distribution of training data after oversampling.

(iii) LSTM **model with attention:**





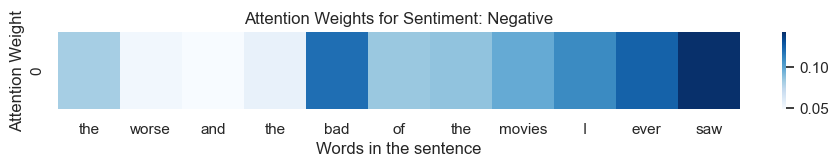
Training loss vs epoch graph and Dev Accuracy vs epoch graphs



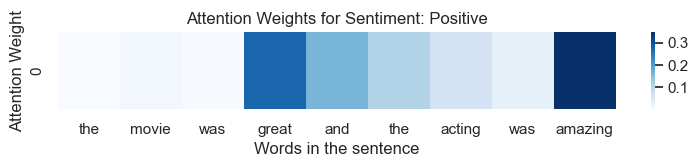
Evaluation metrices for the model

**Visualizing attention:**

**Eg:** sentence = "the worse and the bad of the movies I ever saw"

****

sentence = "the movie was great and the acting was amazing"

****

**Conclusion:**

Initially, the model struggled with class imbalance, achieving a low accuracy and failing to predict class 0. After identifying the imbalance, I applied two techniques to address it: **weighting the undersampled class and oversampling the minority class.**

Furthermore, by modifying the dataset with the top 1502 frequent words and applying tokenization accordingly, the LSTM with attention model showed better results.

By modifying the dataset and removing class imbalance, this significantly improved the model's performance, as seen in the improved accuracy to **49%** and **55%** respectively.

The LSTM model with an embedding dimension of 128 performed particularly well. Finally, replacing the embedding layer with a linear layer also produced strong results with accuracy of **58%** and f1-score of **52.5%**, demonstrating the effectiveness of the LSTM with attention mechanism in capturing relationships between tokens for this dataset.

*--------------------------------------------------------Report Ends-----------------------------------------------------------*

.