**Chapter Four – Data Presentation and Analysis**

This chapter presents and analyze the Multicultural Sentiment Analysis Model by incorporating code snippets from the implementation of the model which is based on the DistilBERT architecture. The focus of the chapter will be on the key steps in model training, its evaluation, and the application of the model to classify sentiments in diverse languages.

**1. Model Setup and Libraries**

This model is built on top of the Transformers library from Hugging Face combined with the pre-trained transformer DistilBERT. Below are the installation of necessary libraries: torch, transformers, and accelerate:

A screen shot of a computer

Description automatically generated

Figure 1 - Necessary Libraries import

**2. Data Preparation**

The first step in this project is loading the data and preparing it for the model. The columns for this dataset are id, text, label, sentiment, and language. Label columns contain different categories of sentiment where 0 explains negative sentiments, 1 is for neutral sentiments, and 2 shows positive sentiments. A short description of this code and its purpose is given below:

A screenshot of a computer

Description automatically generated

Figure 2 - Load and prepare data

This code will load a CSV file containing text data with their corresponding sentiment labels. The first ten rows are displayed, showing key insights into the dataset. From the screenshot, we can see that the structure of the dataset consists of five columns: id, text, label, sentiment, and language. Each row represents a piece of text with a classified sentiment. For example:

* **Row 0:** The text "Cooking microwave pizzas, yummy" has been classified as positive with label 2.
* **Row 4:** The text "That sucks to hear. I hate days like that" is negative, assigned the label 0.

Additionally, the language column provides information about the language of the text. Once the data is loaded, it is split into training and testing sets using the train\_test\_split function:

**Data Processing and Label Mapping**

Then, map the labels to the dataset in numerical values so that they become usable for model training. The text data will be tokenized using the DistilBERT tokenizer: DistilBERT is a lighter and faster version of the BERT model. Tokenization breaks down the text into smaller units called tokens that can be fed into a model:

A close-up of a computer code

Description automatically generated

Figure 3 - Replacing label values

The code snippet maps the label column of this dataset from sentiment strings to integer values 0, 1, and 2. That would prepare the labels into an appropriate format for classification tasks.

**Train-Test Split**

Once the labels are mapped, the dataset is split into training and test sets:

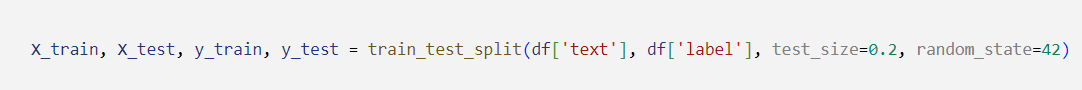


Figure 4 - Train Test Split

Here, the data is split into 80% for training, where X\_train and y\_train can be used, and 20% for testing, where X\_test and y\_test are available. In this way, the performance of the model on unseen data can also be checked.

**2. Model Training**

**Tokenization and Dataset Preparation**

Before training can happen on the model, text data must first be preprocessed by the DistilBERT tokenizer into tokens-numerical representations of text. This is through the code snippet below:

A screen shot of a computer code

Description automatically generated

Figure 5 - Tokenization Function

tokenize\_function: This is used to make sure all the text sequences are padded or truncated to a fixed size of 64 tokens. By this, the model sees consistent sizes for input. The training and test data gets tokenized. Finally, PyTorch datasets are created from the tokenized data so the model can access efficient batches for training and testing. A class, SentimentDataset, which defines how the data is organized for training:

A computer code on a white background

Description automatically generated

Figure 6 - Sentiment Dataset

After defining the dataset class, the training and testing data are wrapped in DataLoader objects for efficient batch processing during training.

A screen shot of a computer code

Description automatically generated

Figure 7 - Train and Test data

**Model and Training Arguments Setup**

Loading the DistilBERT and configuring it for training. Of course, this should be transferred to a GPU if available, to expedite the process. Setup of training arguments is then done. The definition of hyperparameters is done subsequently, including the learning rate, batch size, and the number of epochs. Training the model by using:

A screen shot of a computer

Description automatically generated

Figure 8 - Training Arguments

The Trainer object is responsible for managing the training loop. It handles tasks such as batching data, evaluating the model, and saving checkpoints:

A close-up of a computer screen

Description automatically generated

Figure 9 - Trainer Object

Training is initiated by calling the trainer.train() function, and the model is evaluated after every epoch to find the best-performing version based on accuracy:

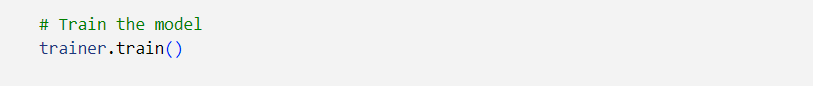


Figure 10 - Train the model

A screenshot of a graph

Description automatically generated

Figure 11 - Train Outputs

This screenshot shows the model training progress across 20 epochs, with accuracy, F1-score, precision, and recall metrics calculated for each epoch.

* **Training Loss** consistently decreases, indicating that the model is learning effectively.
* **Validation Loss** fluctuates, showing that the model may sometimes struggle to generalize to unseen data, but overall, it performs well.
* **Accuracy** reaches a stable value above 0.73 after a few epochs.

**5. Model Evaluation and Metrics**

After training is complete, the model is evaluated on the test set:

A screen shot of a computer program

Description automatically generated

Figure 12 - Compute Metrics Function

A close-up of a white background

Description automatically generated

Figure 13 - Evaluation Results

Accuracy, precision, recall, and F1-score are some of the important metrics to understand performance while working with a classification model. Herein, evaluation results:

* **Accuracy**: 0.7538, this means the model predicted the correct sentiment of 75.38% test samples.
* **F1-Score**: 0.7557-this value indicates the weighted average of the model's precision and recall.
* **Precision**: 0.7599, it is defined as the ratio of true positives to the sum of true positives and false positives.
* **Recall**: 0.7538, which is the ratio of actual positives that the model correctly classified.

Performance metrics have shown that the model should work but still leave room for further improvements. The minor difference of precision from recall would hint that this model might be slightly biased toward precision at the possible expense of recall, meaning it makes fewer incorrect positive predictions, probably missing some correct positives.

**6. Saving the Model**

Once training is complete, the model and tokenizer are saved for future use:

A computer code with text

Description automatically generated

Figure 14 - Save the model and tokenizer

**7. Testing the Model on New Data**

A custom function is then created to test the model on individual inputs. This allows for predictions on new text inputs, which are tokenized and passed through the model with its resulting sentiment predictions:

A screenshot of a computer code

Description automatically generated

Figure 15 - Testing the Model on new data

Sample test inputs can be run through the model to demonstrate its effectiveness in real-world scenarios:

A screenshot of a computer

Description automatically generated

Major steps towards the accomplishment of this text classification task involve loading and preprocessing the data, training the model, and finally evaluation. This technique of Sentiment Classification using DistilBERT for Sequence Classification has given quite good performance on the test set at an accuracy of 75.38%. Features that denote a well-performing model involve various evaluation metrics such as precision, recall, and an F1-score. Testing this on custom inputs further established its generalization capability on new text data. This can be further improved by reducing the variation in validation loss and ensuring better generalization over a number of epochs.