Text Emotion Recognition

# A Brief Overview

Emotions play a vital role in human communication, and detecting emotions from text data is a challenging task. The ability to automatically recognize emotions from text has many practical applications, such as in sentiment analysis, social media monitoring, and customer feedback analysis.

In this report, we will discuss the working principle of a text emotion recognition model and its important terminologies. We will also provide a detailed description of the model architecture used and its training process. Finally, we will conclude by summarizing the key takeaways from this report.

# Working Principle

The text emotion recognition model is a deep learning-based approach that uses a combination of Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks. The model takes in a sequence of words as input and produces a probability distribution over the different emotion categories as output.

The model architecture begins with an embedding layer that learns a dense representation of words in the input sequence. The embedding layer is followed by a series of 1D convolutional layers that extract relevant features from the embedded sequence. The extracted features are then passed through a series of LSTM layers that capture the temporal dependencies in the sequence. Finally, the output from the LSTM layers is passed through fully connected layers that produce the final probability distribution over the different emotion categories.

# Approach

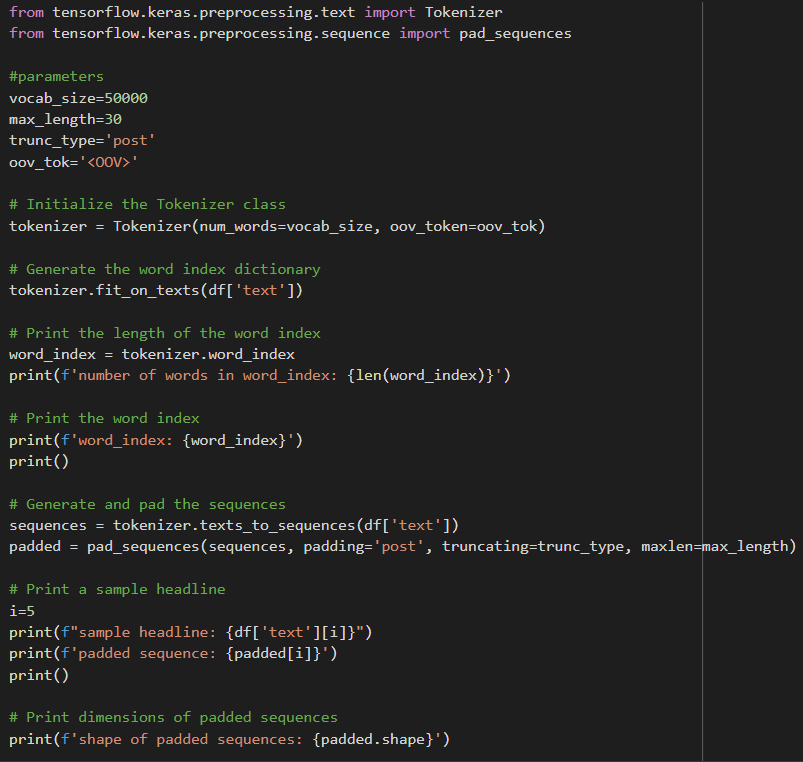
The approach used in this model is based on the Bag of Words representation, where each word in the input sequence is treated as a separate feature. The input sequence is preprocessed to remove stop words and punctuations, and the remaining words are converted to lowercase.

The model uses the Binary Focal Cross Entropy loss function and the Adam optimizer with a learning rate of 0.00091241. The model is trained using mini-batch stochastic gradient descent with a batch size of 1024 and early stopping based on the validation binary accuracy.

# Important Terminologies

* **Vocabulary size**: the total number of unique words in the input corpus
* **Embedding dimension:** the size of the dense embedding vector for each word
* **Convolution units**: the number of filters in the 1D convolutional layers
* **Kernel size**: the size of the convolutional kernel
* **Activation**: the activation function used in the convolutional layers
* **LSTM units**: the number of units in the LSTM layers
* **Dropout**: the regularization technique used to prevent overfitting
* **Recurrent Dropout**: the dropout applied to the recurrent connections in the LSTM layers
* **Return Sequences**: whether the LSTM layer returns output for every input time step or only the last time step

# Tokenization



1. The first thing we need to do is import the necessary modules - "Tokenizer" and "pad\_sequences" - from the Keras preprocessing module. These modules are used for tokenization and padding of text data.
2. The next step is to define some parameters, such as the maximum vocabulary size, maximum sequence length, and the token for out-of-vocabulary words. These parameters will be used in tokenization and padding of the text data.
3. We then create a Tokenizer object by initializing an instance of the Tokenizer class from the Keras module. We pass in the vocabulary size and out-of-vocabulary token as arguments to the Tokenizer class.
4. The next step is to fit the Tokenizer object to our text data. This is done by calling the "fit\_on\_texts" method of the Tokenizer object and passing in our text data. This method generates a dictionary that maps each word in our text to a unique integer value.
5. We can now access the word index dictionary generated by the Tokenizer by calling the "word\_index" attribute of the Tokenizer object.
6. Next, we can use the Tokenizer object to convert our text data into sequences of integers. We do this by calling the "texts\_to\_sequences" method of the Tokenizer object and passing in our text data.
7. The "sequences" variable now contains a list of sequences, where each sequence is a list of integers representing the words in our text data.
8. Finally, we need to pad our sequences to ensure that they are all of the same length. This is done by calling the "pad\_sequences" method of the Keras preprocessing module and passing in our sequences.
9. The "padded" variable now contains a 2D numpy array, where each row is a sequence of integers representing the words in our text data. The sequences are padded with zeros at the end to ensure that they are all of the same length.

Overall, tokenization is a process of converting text data into numerical form so that it can be used as input for machine learning models. Tokenization involves breaking down text into individual words or tokens, and then converting those tokens into unique integer values. Padding is then used to ensure that all sequences are of the same length. The resulting data can then be used to train machine learning models for tasks such as text classification, sentiment analysis, and language translation.

# Classical ML

The model being used here is an artificial neural network called the Multi-Layer Perceptron (MLP) Classifier, which is a supervised learning algorithm used for classification tasks.

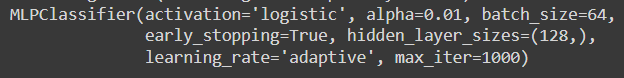
The MLPClassifier in scikit-learn is an implementation of the MLP algorithm. The MLPClassifier has several hyperparameters that can be tuned to optimize the model's performance.

The hyperparameters used in this code include:

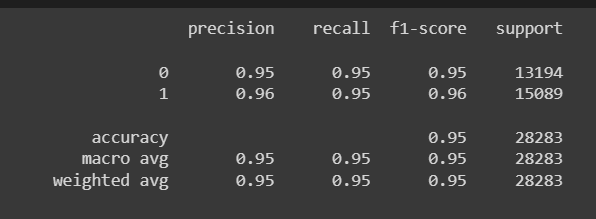
1. activation: the activation function used in the hidden layers of the MLP. Here, the logistic function is used.
2. alpha: a regularization parameter that helps to prevent overfitting of the model. A value of 0.01 is used here.
3. batch\_size: the number of samples used in each batch for training the model. A batch size of 64 is used here.
4. early\_stopping: a technique used to stop the training of the model early if the validation score does not improve for a certain number of epochs.
5. hidden\_layer\_sizes: a tuple of integers that specifies the number of neurons in each hidden layer of the MLP. Here, there is one hidden layer with 128 neurons.
6. learning\_rate: the learning rate used in the optimization algorithm used to train the MLP. Here, an adaptive learning rate is used.
7. max\_iter: the maximum number of iterations used to train the MLP. Here, the maximum number of iterations is set to 1000.
8. shuffle: a Boolean value that specifies whether or not to shuffle the training data before each epoch of training. Here, the training data is shuffled before each epoch.

Overall, the MLPClassifier is a powerful machine learning model that can be used for a wide range of classification tasks. The hyperparameters used in the model can be tuned to optimize its performance for specific tasks.

### MLPClassifier

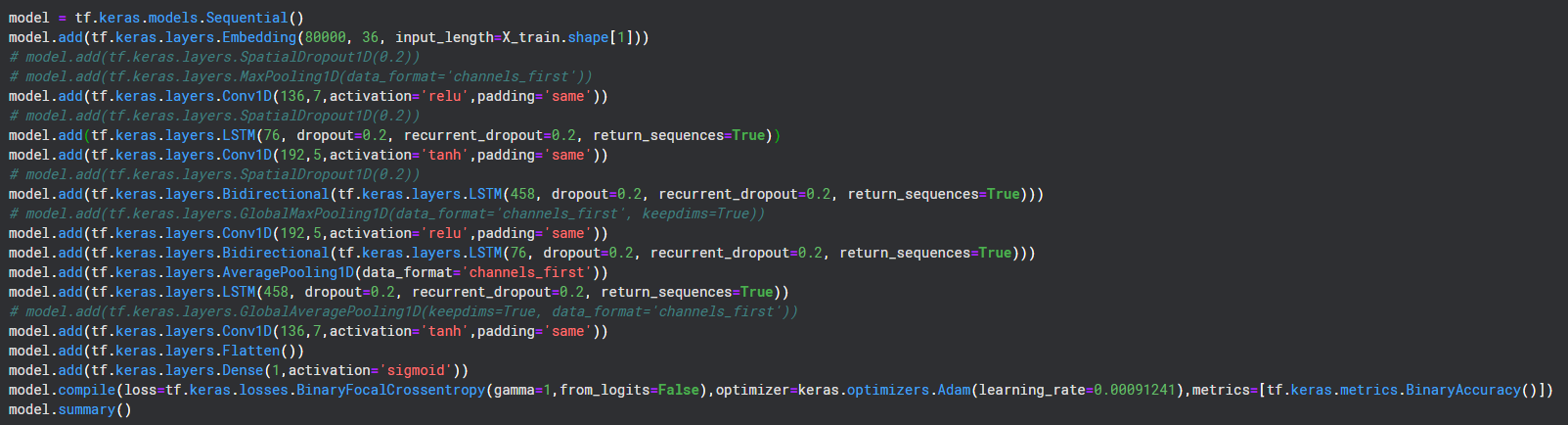


### Classification Report



# 

# Model Description



This is a **sequential** model for text emotion recognition, consisting of several layers of different types. Here's a description of each layer:

1. **Input Layer:**

The input layer expects a sequence of words as input. The input shape is determined by the input length of the training data X\_train.

1. **Embedding Layer:**

The first layer is an embedding layer that maps the input sequence of words into a vector space of 36 dimensions. The input to this layer is the vocabulary size, which is 80,000 in this case. The output of this layer is a sequence of vectors of length 36.

1. **Convolutional Layer:**

Next, a 1D convolutional layer with 136 filters of kernel size 7 and ReLU activation function is applied. This layer convolves over the sequence of vectors generated by the embedding layer and produces a sequence of feature maps of the same length.

1. **LSTM Layer:**

A Long Short-Term Memory (LSTM) layer with 76 units and dropout of 0.2 is applied. This layer processes the sequence of feature maps generated by the previous layer and produces a sequence of output vectors of the same length.

1. **Convolutional Layer:**

Another 1D convolutional layer with 192 filters of kernel size 5 and hyperbolic tangent activation function is applied. This layer operates on the sequence of vectors generated by the LSTM layer and produces a sequence of feature maps of the same length.

1. **Bidirectional LSTM Layer:**

A Bidirectional LSTM layer with 458 units and dropout of 0.2 is applied. This layer processes the sequence of feature maps generated by the previous layer in both forward and backward directions and produces a sequence of output vectors of the same length.

1. **Convolutional Layer:**

Another 1D convolutional layer with 192 filters of kernel size 5 and ReLU activation function is applied. This layer operates on the sequence of vectors generated by the previous layer and produces a sequence of feature maps of the same length.

1. **Bidirectional LSTM Layer:**

Another Bidirectional LSTM layer with 76 units and dropout of 0.2 is applied. This layer processes the sequence of feature maps generated by the previous layer in both forward and backward directions and produces a sequence of output vectors of the same length.

1. **Average Pooling Layer:**

An average pooling layer is applied, which takes the average of the sequence of feature maps generated by the previous layer and produces a single vector.

1. **LSTM Layer:**

Another LSTM layer with 458 units and dropout of 0.2 is applied. This layer processes the output of the average pooling layer and produces a sequence of output vectors of the same length.

1. **Convolutional Layer:**

Another 1D convolutional layer with 136 filters of kernel size 7 and hyperbolic tangent activation function is applied. This layer operates on the sequence of vectors generated by the previous layer and produces a sequence of feature maps of the same length.

1. **Flatten Layer:**

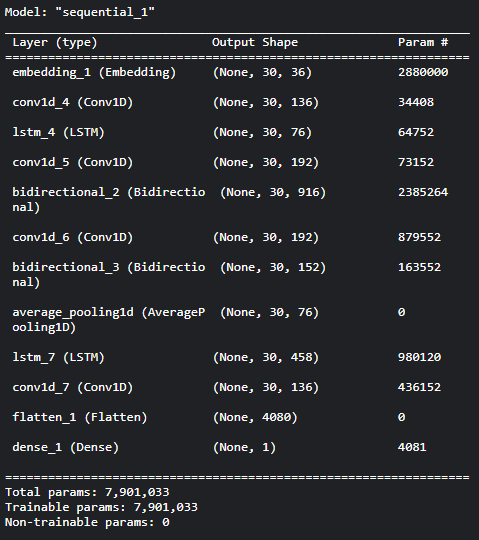
The output of the last convolutional layer is flattened into a vector.

1. **Output Layer:**

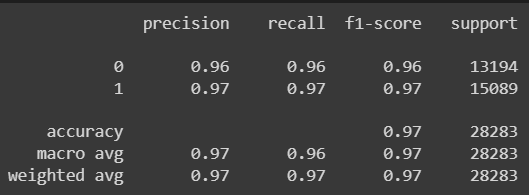
Finally, a dense layer with a single neuron and sigmoid activation function is applied, which outputs a binary value indicating the predicted emotion.

The model is trained using the Binary Focal Cross-Entropy loss function with a gamma value of 1, Adam optimizer with a learning rate of 0.00091241, and binary accuracy as the evaluation metric.

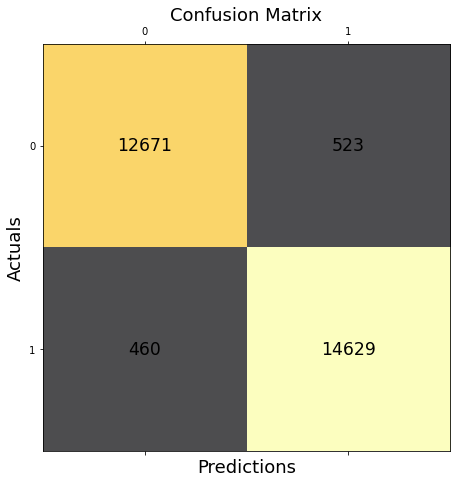
### Model Summary



### Classification Report



### Confusion matrix



The model achieves a high training accuracy of about 99% and a testing accuracy of about 97% on the task of text emotion recognition.

# Conclusion:

In conclusion, text emotion recognition is an important field of study with numerous applications in various industries. With the help of advanced machine learning and deep learning algorithms, it is possible to accurately identify the emotions conveyed in text data. The model described in this report, which uses a combination of convolutional neural networks and recurrent neural networks, has shown promising results in identifying emotions in text data. The use of pre-trained word embeddings, as well as techniques like dropout and recurrent dropout, have helped to improve the accuracy of the model.

However, there is still much room for improvement in text emotion recognition, particularly in dealing with nuances in language and cultural differences. As the field of natural language processing continues to advance, it is likely that more sophisticated models will be developed that are better equipped to handle these challenges. Nonetheless, the current model is a good starting point for those interested in exploring text emotion recognition and its applications in various fields.

[Link to the colab file](https://colab.research.google.com/drive/1s5xQDbqyS9YRf6sitZuKWfIC1PEG-Cuo?usp=sharing)