Sentiment Analysis for Code-Mixed Text with BERT and Transformers

By

Arjun Singh
Reg. no.: SRIP_2020_R16
Summer Research Intern
Domain: Language and Dialogue Processing

Summer Internship Report

Center for Cognitive Computing, IIIT-Allahabad

UNDER THE SUPERVISION OF

Prof. U. S. Tiwary
Professor
IIIT-ALLAHABAD

Table of Contents

Topic

Sentiment Analysis for Code-Mixed Text with BERT and Transformers

Introduction:

Sentiment analysis refers to the use of natural language processing, text analysis, computational linguistics, and biometrics to systematically identify, extract, quantify, and study affective states and subjective information. Sentiment analysis is the interpretation and classification of emotions (positive, negative and neutral) within text data using text analysis techniques. Sentiment analysis allows businesses to identify customer sentiment toward products, brands or services in online conversations and feedback.

I am going to perform the Sentiment Analysis task on Code-Mixed text, which will basically be the Social Media text, but can be further applied to more varied dataset of various distribution. Mixing languages, also known as code- mixing, is a norm in multilingual societies. Multilingual people, who are non-native English speakers, tend to code-mix using English-based phonetic typing and the insertion of anglicisms in their main language. In addition to mixing languages at the sentence level, it is fairly common to find the code-mixing behavior at the word level. This linguistic phenomenon poses a great challenge to conventional NLP systems, which currently rely on monolingual resources to handle the combination of multiple languages. I am going to perform sentiment analysis task in code-mixed social media text and investigate the performance using the BERT (Bidirectional Encoder Representations from Transformers) which uses the concepts from Attention and Transformers algorithms.

Dataset

The dataset of the Task is taken from the “SemEval 2020 Task 9 : SentiMix”.

This dataset is from the World-wide competition of CODALAB platform.

It includes the text from the Code-Mixed HINGLISH language as well as the SPANGLISH language, which are used in various tweets of Social Media Twitter.

The link for the Trial Dataset is given below:

Hinglish Trial data:
https://ritual-uh.github.io/sentimix2020/data/hinglish_trial.txt

Spanglish Trial data:
https://ritual-uh.github.io/sentimix2020/data/spanglish_trial.txt

Literature Survey

S.No.	Title	Objective	Methods	Challenges Dealt
1	BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding	Pre-train deep bidirectional representations from unlabeled text.	Jointly conditioning on both left and right context in all layers.	The pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks.
2	Attention Is All You Need	Proposal of a new simple network architecture, the Transformer.	Attention mechanisms, dispensing with recurrence and convolutions.	Superior in quality while being more parallelizable and requiring significantly less time to train.
3	Evolution of transfer learning in natural language processing	Presentation of cutting-edge methods and architectures such as BERT, GPT, ELMo, etc	Providing better performance on target task using models via transfer learning.	Traditional models do not generalize well, hence The transfer learning based model dealt with this problem.
4	SentiBERT: A Transferable Transformer-Based Architecture for Compositional Sentiment Semantics	Proposal of SentiBERT, a variant of BERT that effectively captures compositional sentiment semantics	Incorporating contextualized representation with binary constituency parse tree to capture semantic composition.	Improvement in capturing negation and the contrastive relation and model the compositional sentiment semantics.

Work Plan

Week 1 and Week 2: Learning the concepts and algorithms for Transformers and Self-Attention to implement BERT model using PyTorch and Tensorflow frameworks and Python 3.0 programming language.

Week 3: To implement the BERT model and related modules using the appropriate parameters which may/may-not be taken using Transfer Learning for the Sentiment Analysis task.

Week 4: To fine-tune the hyperparameters and other related parameters and investigate the performance of the model with various parameters on the Dataset.

Week 5-6: Complete the implementation of all the modules of the model and do Result analysis and Preparing the presentation.

Week 7-8: Further modifications to the model for improvements of the performance of the software model.

Methodology

Explanation of Code-Mixed text

Mixing languages, also known as code-mixing, is a norm in multilingual societies. Multilingual people, who are non-native English speakers, tend to code-mix using English-based phonetic typing and the insertion of anglicisms in their main language. In addition to mixing languages at the sentence level, it is fairly common to find the code-mixing behavior at the word level. This linguistic phenomenon poses a great challenge to conventional NLP systems, which currently rely on monolingual resources to handle the combination of multiple languages. Specifically, we focus on the combination of English with Hindi (Hinglish), which is the 4th most spoken languages in the world.

Statistics show that half of the messages on Twitter are in a language other than English. This evidence suggests that other languages, including multilingual and code-mixing, need to be considered by the NLP community. That is why, I have chosen to perform Sentiment Analysis on such Code-Mixed Texts with the dataset of Twitter tweets in Hinglish language.

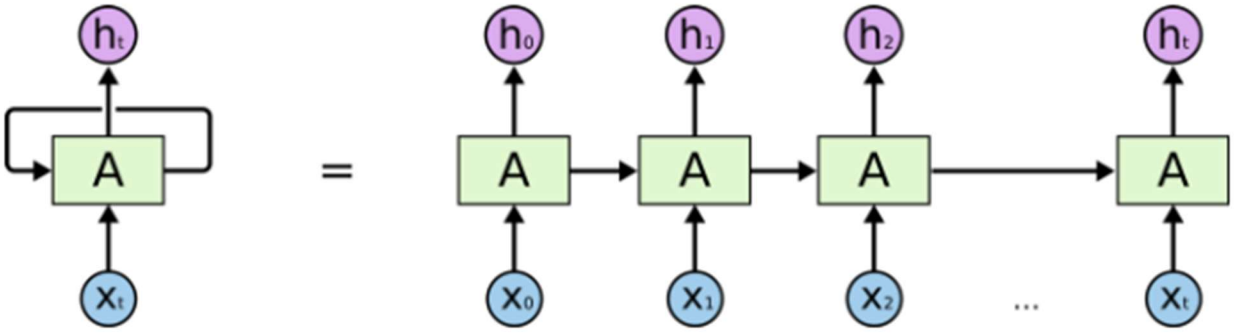
Traditional LSTM and associated Short-comings

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network capable of learning order dependence in sequence prediction problems.

This is a behavior required in complex problem domains like machine translation, speech recognition, and more.

LSTMs are a complex area of deep learning. It can be hard to get your hands around what LSTMs are, and how terms like bidirectional and sequence-to-sequence relate to the field.

LSTM and derivatives use mainly sequential processing over time. See the horizontal arrow in the diagram below:



This arrow means that long-term information has to sequentially travel through all cells before getting to the present processing cell. That means LSTMs are computationally slow and they do not take advantage of the Multiprocessing and Parallel computations which GPUs allow. Here comes the BERT and Transformers at rescue.

Transformer

The paper 'Attention Is All You Need' introduces a novel architecture called Transformer. As the title indicates, it uses the attention-mechanism we saw earlier. Like LSTM, Transformer is an architecture for transforming one sequence into another one with the help of two parts (Encoder and Decoder), but it differs from the previously described/existing sequence-to-sequence models because it does not imply any Recurrent Networks (GRU, LSTM, etc.).

Recurrent Networks were, until now, one of the best ways to capture the timely dependencies in sequences. However, the team presenting the paper proved that an architecture with only attention-mechanisms without any RNN (Recurrent Neural Networks) can improve on the results in translation task and other tasks! One improvement on Natural Language Tasks is presented by a team introducing BERT: [BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding], which is described in the later section.

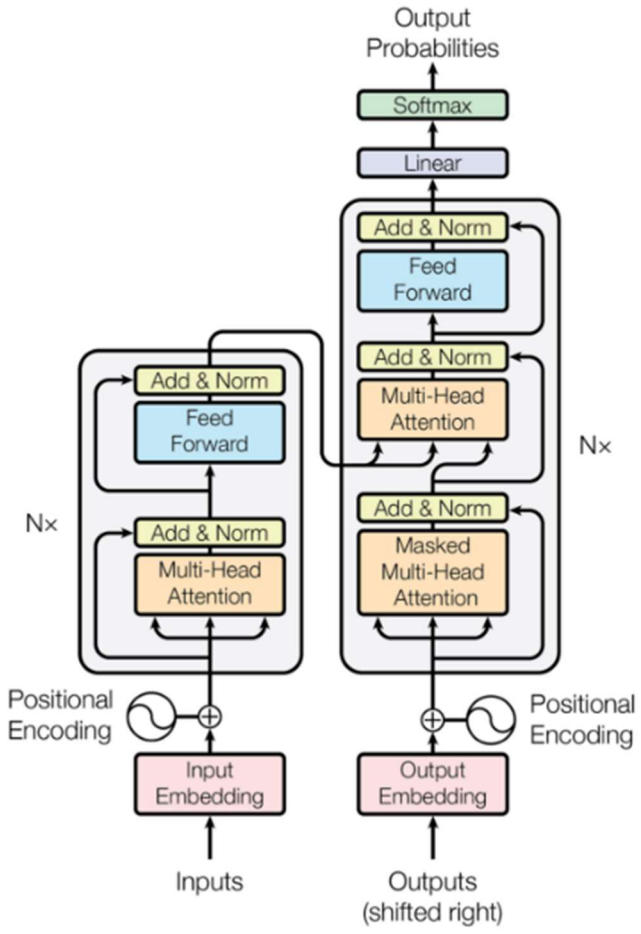


Figure 1. The Transformer Architecture

The Encoder is on the left and the Decoder is on the right. Both Encoder and Decoder are composed of modules that can be stacked on top of each other multiple times, which is described by N_x in the figure. We see that the modules consist mainly of Multi-Head Attention and Feed Forward layers. The inputs and outputs (target sentences) are first embedded into an n-dimensional space since we cannot use strings directly.

One slight but important part of the model is the positional encoding of the different words. Since we have no recurrent networks that can remember how sequences are fed into a model, we need to somehow give every word/part in our sequence a relative position since a sequence depends on the order of its elements. These positions are added to the embedded representation (n-dimensional vector) of each word.

Let's have a closer look at these Multi-Head Attention bricks in the model:

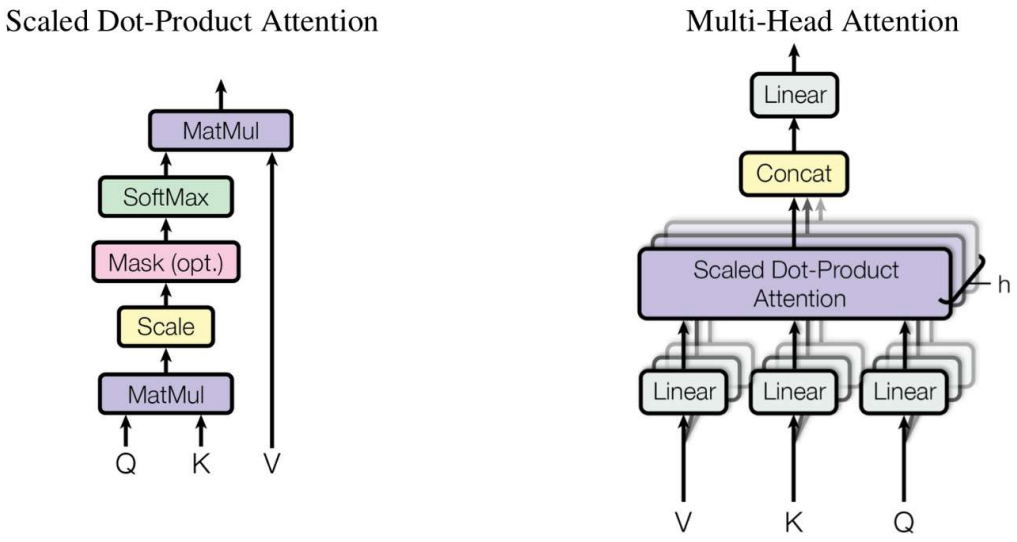


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Let's start with the left description of the attention-mechanism. It's not very complicated and can be described by the following equation:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Q is a matrix that contains the query (vector representation of one word in the sequence), K are all the keys (vector representations of all the words in the sequence) and V are the values, which are again the vector representations of all the words in the sequence. For the encoder and decoder, multi-head attention modules, V consists of the same word sequence than Q. However, for the attention module that is taking into account the encoder and the decoder sequences, V is different from the sequence represented by Q.

To simplify this a little bit, we could say that the values in V are multiplied and summed with some attention-weights *a,* where our weights are defined by:

$$a = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

This means that the weights *a* are defined by how each word of the sequence (represented by Q) is influenced by all the other words in the sequence (represented by K). Additionally, the SoftMax function is applied to the weights *a* to have a distribution between 0 and 1. Those weights are then applied to all the words in the sequence that are introduced in V (same vectors than Q for encoder and decoder but different for the module that has encoder and decoder inputs).

The righthand picture describes how this attention-mechanism can be parallelized into multiple mechanisms that can be used side by side. The attention mechanism is repeated multiple times with linear projections of Q, K and V. This allows the system to learn from different representations of Q, K and V, which is beneficial to the model. These linear representations are done by multiplying Q, K and V by weight matrices W that are learned during the training.

Those matrices Q, K and V are different for each position of the attention modules in the structure depending on whether they are in the encoder, decoder or in-between encoder and decoder. The reason is that we want to attend on either the whole encoder input sequence or a part of the decoder input sequence. The multi-head attention module that connects the encoder and decoder will make sure that the encoder input-sequence is taken into account together with the decoder input-sequence up to a given position.

After the multi-attention heads in both the encoder and decoder, we have a pointwise feed-forward layer. This little feed-forward network has identical parameters for each position, which can be described as a separate, identical linear transformation of each element from the given sequence.

BERT

BERT (Bidirectional Encoder Representations from Transformers) is a recent paper published by researchers at Google AI Language. It has caused a stir in the Machine Learning community by presenting state-of-the-art results in a wide variety of NLP tasks, including Question Answering (SQuAD v1.1), Natural Language Inference (MNLI), and others.

BERT's key technical innovation is applying the bidirectional training of Transformer, a popular attention model, to language modelling.

BERT Working

BERT makes use of Transformer, an attention mechanism that learns contextual relations between words (or sub-words) in a text. In its vanilla form, Transformer includes two separate mechanisms — an encoder that reads the text input and a decoder that produces a prediction for the task. Since BERT's goal is to generate a language model, only the encoder mechanism is necessary. The detailed workings of Transformer are described in a paper by Google.

As opposed to directional models, which read the text input sequentially (left-to-right or right-to-left), the Transformer encoder reads the entire sequence of words at once. Therefore, it is considered bidirectional, though it would be more accurate to say that it's non-directional. This characteristic allows the model to learn the context of a word based on all of its surroundings (left and right of the word).

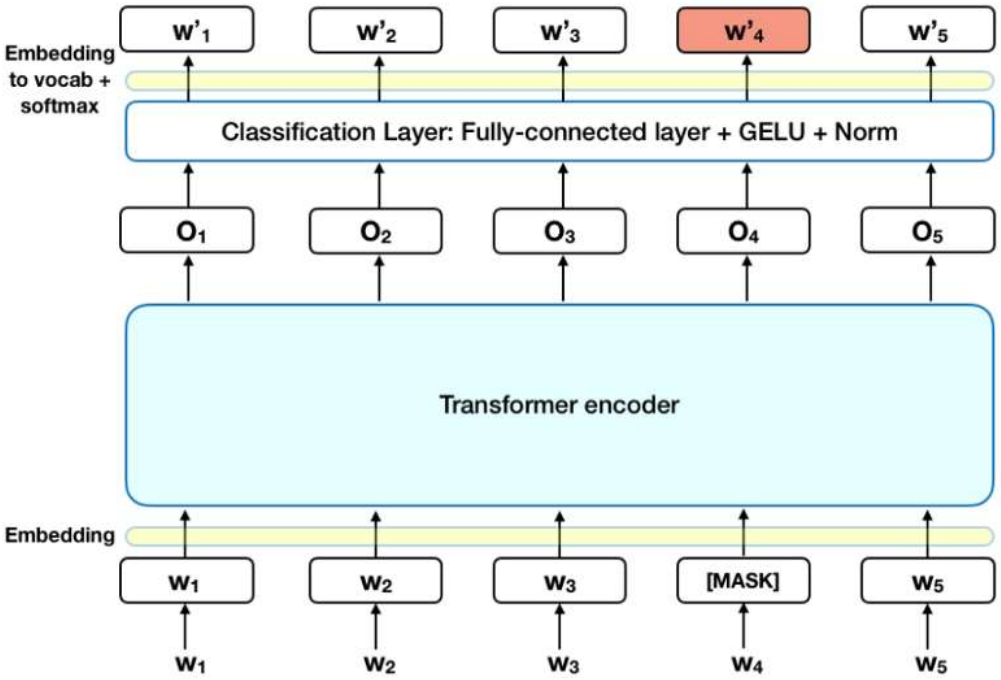
The chart below is a high-level description of the Transformer encoder. The input is a sequence of tokens, which are first embedded into vectors and then processed in the neural network. The output is a sequence of vectors of size H, in which each vector corresponds to an input token with the same index.

When training language models, there is a challenge of defining a prediction goal. Many models predict the next word in a sequence (e.g. "The child came home from ___"), a directional approach which inherently limits context learning. To overcome this challenge, BERT uses two training strategies:

Masked LM (MLM)

Before feeding word sequences into BERT, 15% of the words in each sequence are replaced with a [MASK] token. The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence. In technical terms, the prediction of the output words requires:

1. Adding a classification layer on top of the encoder output.
2. Multiplying the output vectors by the embedding matrix, transforming them into the vocabulary dimension.
3. Calculating the probability of each word in the vocabulary with softmax.



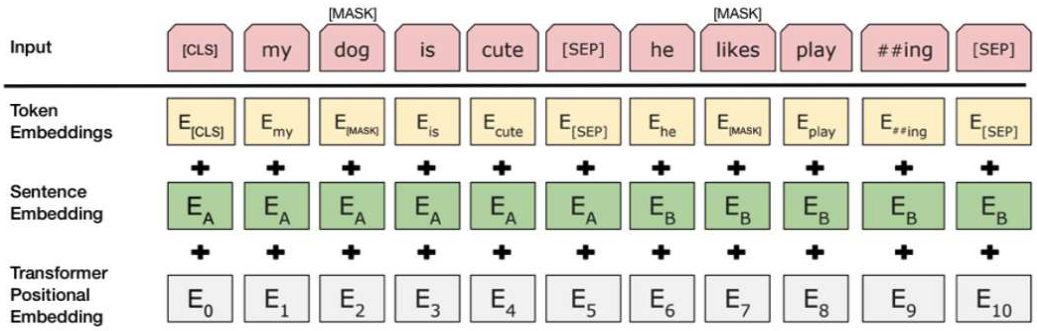
The BERT loss function takes into consideration only the prediction of the masked values and ignores the prediction of the non-masked words. As a consequence, the model converges slower than directional models, a characteristic which is offset by its increased context awareness.

Next Sentence Prediction (NSP)

In the BERT training process, the model receives pairs of sentences as input and learns to predict if the second sentence in the pair is the subsequent sentence in the original document. During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the other 50% a random sentence from the corpus is chosen as the second sentence. The assumption is that the random sentence will be disconnected from the first sentence.

To help the model distinguish between the two sentences in training, the input is processed in the following way before entering the model:

1. A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
2. A sentence embedding indicating Sentence A or Sentence B is added to each token. Sentence embeddings are similar in concept to token embeddings with a vocabulary of 2.
3. A positional embedding is added to each token to indicate its position in the sequence. The concept and implementation of positional embedding are presented in the Transformer paper.



To predict if the second sentence is indeed connected to the first, the following steps are performed:

1. The entire input sequence goes through the Transformer model.
2. The output of the [CLS] token is transformed into a 2×1 shaped vector, using a simple classification layer (learned matrices of weights and biases).
3. Calculating the probability of IsNextSequence with softmax.

When training the BERT model, Masked LM and Next Sentence Prediction are trained together, with the goal of minimizing the combined loss function of the two strategies.

How to use BERT (Fine-tuning)

Using BERT for a specific task is relatively straightforward:

BERT can be used for a wide variety of language tasks, while only adding a small layer to the core model:

1. Classification tasks such as sentiment analysis are done similarly to Next Sentence classification, by adding a classification layer on top of the Transformer output for the [CLS] token.
2. In Question Answering tasks (e.g. SQuAD v1.1), the software receives a question regarding a text sequence and is required to mark the answer in the sequence. Using BERT, a Q&A model can be trained by learning two extra vectors that mark the beginning and the end of the answer.
3. In Named Entity Recognition (NER), the software receives a text sequence and is required to mark the various types of entities (Person, Organization, Date, etc) that appear in the text. Using BERT, a NER model can be trained by feeding the output vector of each token into a classification layer that predicts the NER label.

In the fine-tuning training, most hyper-parameters stay the same as in BERT training, and the paper gives specific guidance (Section 3.5) on the hyper-parameters that require tuning. The BERT team has used this technique to achieve state-of-the-art results on a wide variety of challenging natural language tasks, detailed in Section 4 of the paper.

In my project, I have fine-tuned BERT for the classification task. I have used the HuggingFace library of transformers, details of which are provided in the later section.

Approach and Pipeline:

1. Preprocessing and Exploratory Data Analysis

In this, I explored the Dataset and made the following inferences:

Data info

- The data is in CONLL format. It looks like:
- meta uid sentiment
- token lang_id
- meta uid sentiment
- token lang_id
- token lang_id
- **Uid** is a unique id for each tweet.
- **lang_id** is 'HIN' if the token is in Hindi, 'ENG' if the token is in English, and 'O' if the token is in neither of the languages.
- The **sentiment** is either positive, negative or neutral.
- Total 17000 tweets are available. The trial data has 1869 tweets. The train data has 15131 additional tweets. (does not include trial data). The train data after split has 14k tweets. The validation data has 3k tweets.

Additional

- The Tweets contains emojis which affect the performance of the model and they have to be preprocessed accordingly.
- The dataset contains almost the equivalent number of tweets of different sentiment categories. So, there is no need of Stratifying the dataset for no skewed results.
- A new label field is added to the dataset to represent the sentiment value numerically in the form of label.

```
tweet_id=[]
tweets=[]
temp=""
sentiment=[]
emojis=""
flag=False
file = open('/content/data/train_14k_split_conll.txt','r')
for line in file:
  if len(line.split())==3:
    if(flag):
      if (len(emojis)>1):
        temp=temp+" Emotions:"+' '.join(emojis.split('_'))
      tweets.append(temp)
      temp=""
      emojis=""
    flag=True
    tweet_id.append(int(line.split()[1]))
    sentiment.append(line.split()[2])
  elif len(line.split())==2:
    if (is_emoji(line.split()[0][0])):
      for i in range(len(line.split()[0])):
        emojis=emojis+" "+emoji_to_text(line.split()[0][i])
    else:
      temp=temp+" "+line.split()[0]
if (len(emojis)>1):
  temp=temp+" Emotions:"+' '.join(emojis.split('_'))
tweets.append(temp)
file.close()
```

2. Identifying the Training, Validation and Test data

Fortunately, the training, validation and test data were given as the part of Problem, in separate files, so there was no need to split any file into the Training and Validation sets. Moreover, the trial data was merged with the Training data to increase the performance with as much data as possible.

3. Tokenization and Encoding

The tweets were tokenized using the BertTokenizer of the HuggingFace transformers library.

And the tokens are encoded with the word piece embedding method to obtain the token embeddings which represents the words (tokens) in numerical form as Feature vectors.

The encoded data is converted to the TensorDataset which can be used in Pytorch environment.

```
from transformers import BertTokenizer
from torch.utils.data import TensorDataset

tokenizer = BertTokenizer.from_pretrained(
    'bert-base-uncased',
    do_lower_case=True
)
```

```
encoded_data_train = tokenizer.batch_encode_plus(
    df.tweets.values,
    add_special_tokens=True,
    return_attention_mask=True,
    pad_to_max_length=True,
    max_length=256,
    return_tensors='pt'
)
```

```
input_ids_train = encoded_data_train['input_ids']
attention_masks_train = encoded_data_train['attention_mask']
labels_train = torch.tensor(df.label.values)
```

4. Setting up DataLoaders

The dataset of the training and testing data are packed in the data loaders which can be used in the Pytorch environment and will be useful by the model to iterate over the dataset as required. This is the necessary step as it is a requirement of the transformers model of HuggingFace library so that the model can be trained.

```
from torch.utils.data import DataLoader, RandomSampler, SequentialSampler

batch_size = 32

dataloader_train = DataLoader(dataset_train,
                              sampler=RandomSampler(dataset_train),
                              batch_size=batch_size)
```

5. Setting up BERT Pretrained Model

I have used the 'bert_base_uncased' model of BertForSequenceClassification of transformers library provided by HuggingFace.

This model has been pretrained on 12-layer, 768-hidden, 12-heads, 110M parameters on lower-cased English text.

```
from transformers import BertForSequenceClassification

model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased",
    num_labels=len(sentiment_dict),
    output_attentions=False,
    output_hidden_states=False
)
```

6. Optimiser and Scheduler

The AdamW optimizer is used with the scheduler to handle the Over-training scenario. The hyperparameters such as the learning rate are used as stated by the original authors of the paper.

7. Performance Matrix

The performance matrix used is the F1 score to measure the effective-ness of the model after the fine-tuning process.

```
import numpy as np
from sklearn.metrics import f1_score

def f1_score_func(preds, labels):
    preds_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
    return f1_score(labels_flat, preds_flat, average='weighted')


def accuracy_per_class(preds, labels):
    total_numerator=0
    total_denominator=0
    label_dict_inverse = {v: k for k, v in sentiment_dict.items()}

    preds_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()

    for label in np.unique(labels_flat):
        y_preds = preds_flat[labels_flat==label]
        y_true = labels_flat[labels_flat==label]
        print(f'Class: {label_dict_inverse[label]}')
        print(f'Accuracy: {len(y_preds[y_preds==label])}/{len(y_true)}\n')
        total_numerator=total_numerator+len(y_preds[y_preds==label])
        total_denominator=total_denominator+len(y_true)

    print(f'Total Accuracy: {(total_numerator/total_denominator)*100}\n')
```

8. Fine-tuning (Training)

The model is fine-tuned with the training data on the GPU processor with appropriate epochs and batch size. The results of each epoch is displayed using the tqdm library of python.

```
for epoch in tqdm(range(1, epochs+1)):

    model.train()

    loss_train_total = 0

    progress_bar = tqdm(dataloader_train, desc='Epoch {:1d}'.format(epoch), leave=False, disable=False)
    for batch in progress_bar:

        model.zero_grad()

        batch = tuple(b.to(device) for b in batch)

        inputs = {'input_ids':      batch[0],
                  'attention_mask': batch[1],
                  'labels':         batch[2],
                 }

        outputs = model(**inputs)

        loss = outputs[0]
        loss_train_total += loss.item()
        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        optimizer.step()
        scheduler.step()

        progress_bar.set_postfix({'Training_loss': '{:.3f}'.format(loss.item()/len(batch))})
```

9. Evaluation

The fine-tuned model is evaluated on the validation set and the test set and the Performance is measured as the F1 Score.

```
_, predictions, true_vals = evaluate(dataloader_val)
val_f1 = f1_score_func(predictions, true_vals)
tqdm.write(f'F1 Score (Weighted): {val_f1}')
accuracy_per_class(predictions, true_vals)
```

10. Prediction

The prediction is made on the new tweets using the trained model which is stored on the google drive.

Software Used

Hardware:

- GPU runtime type (Google Colab environment)
- CUDA support

Software:

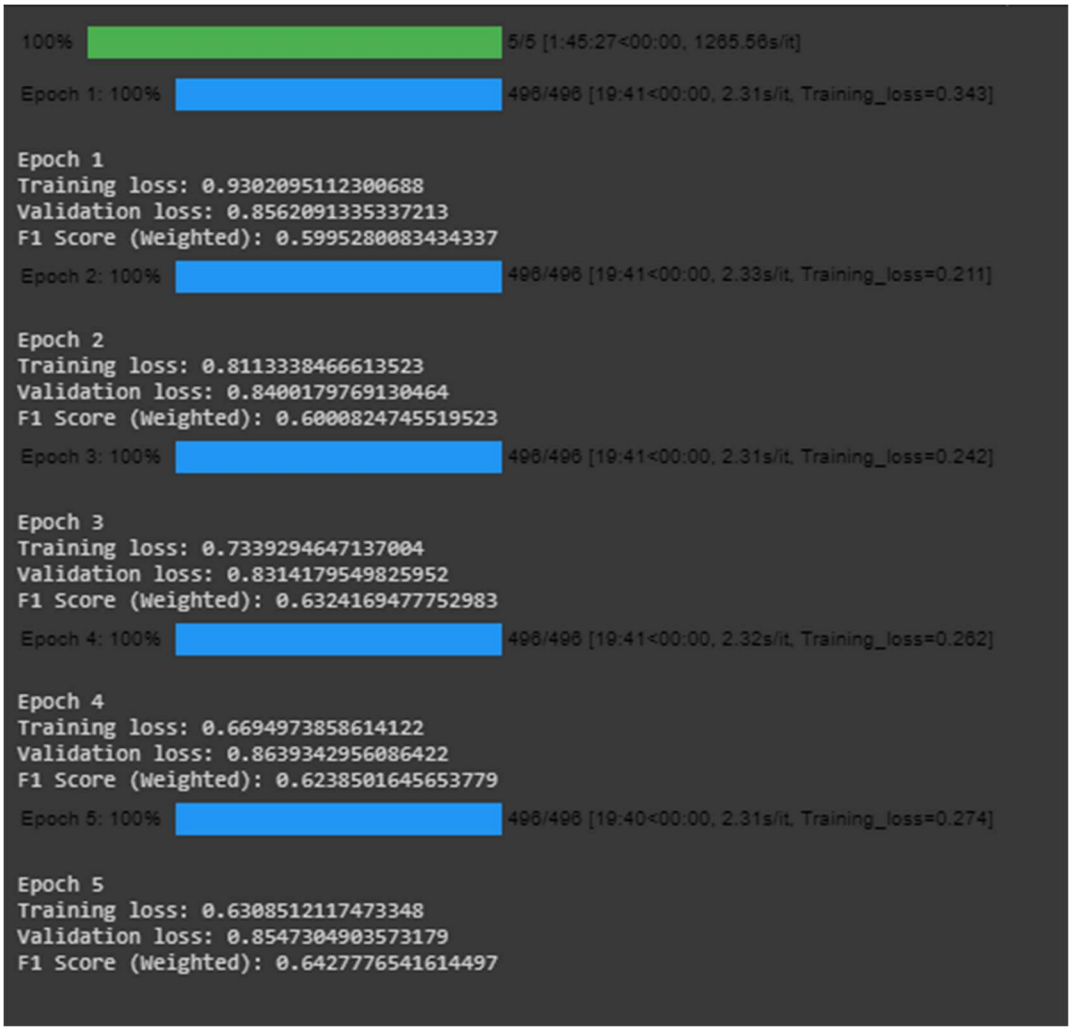
- **Language used:** Python 3.0
- **Tools used:** Google Colab
- **Framework used:** PyTorch
- **Libraries used:** transformers (huggingface.co), numpy, pandas, sklearn, tqdm, emoji
- **Dataset used:** SemEval 2020 Task 9 : SentiMix Hinglish dataset

Results and Discussion

The metric I used for evaluating the Performance of the Model is F1 Score averaged across the positives, negatives and the neutrals.

After applying the BERT (Bi-directional Encoder Representation from Transformer) model, I got the following results. I tried to optimize the solution and increase the performance of the model by handling emojis. I separated out emojis and appended all the emojis at the end of the tweets. Below mentioned screenshots shows the accuracy and F1 Score of my model over the "SemEval 2020 Task 9 : SentiMix" Competition's Hinglish dataset.

Training statistics:



We can observe that with a comparison of Training loss and Validation loss, we can deduce the model to be working best after Epoch 3. Hence, I saved the model after Epoch 3 and later used it for Testing on test set.

Results from Validation Set

```
[ ]  _, predictions, true_vals = evaluate(dataloader_val)
     val_f1 = f1_score_func(predictions, true_vals)
     tqdm.write(f'F1 Score (Weighted): {val_f1}')
     accuracy_per_class(predictions, true_vals)
```

```
F1 Score (Weighted): 0.6427776541614497
Class: neutral
Accuracy: 649/1128

Class: negative
Accuracy: 610/890

Class: positive
Accuracy: 669/982

Total Accuracy: 64.26666666666667
```

Results from Test Set.

```
[ ]  _, predictions, true_vals = evaluate(dataloader_test)
     val_f1 = f1_score_func(predictions, true_vals)
     tqdm.write(f'F1 Score (Weighted): {val_f1}')
     accuracy_per_class(predictions, true_vals)
```

```
F1 Score (Weighted): 0.6913257453953022
Class: neutral
Accuracy: 736/1100

Class: negative
Accuracy: 606/900

Class: positive
Accuracy: 723/1000

Total Accuracy: 68.83333333333333
```

We can see that the model gives a F1 Score of 0.6913257 or 69.13257%

Important Links

Dataset Links

https://drive.google.com/drive/folders/1vl5BWYLL5Vz1VJLQa6LADJc_jWSSiSyB

Code Link:

https://drive.google.com/drive/folders/13G7Y2hrnDiTvY3piXBkl-r52md6McQRf

Download and open using Google Colab.

Future Scope of Project

The model gives an accuracy/F1 Score of 69.13257% on the SemEval 2020 Task 9 : SentiMix Hinglish dataset. It can further be applied on other Datasets with multiple labels to achieve even better results after optimizations on emojis and Cases of the words in sentences.

References

1. Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" https://arxiv.org/abs/1810.04805
2. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin "Attention Is All You Need" https://arxiv.org/abs/1706.03762
3. Google AI Blog regarding research of "Transformer" https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html
4. Aditya Malte, Pratik Ratadiya "Evolution of transfer learning in natural language processing" https://arxiv.org/abs/1910.07370
5. Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, Xuanjing Huang "Pre-trained Models for Natural Language Processing: A Survey" https://arxiv.org/abs/2003.08271
6. Da Yin, Tao Meng, Kai-Wei Chang "SentiBERT: A Transferable Transformer-Based Architecture for Compositional Sentiment Semantics" https://arxiv.org/abs/2005.04114
7. Xin Li, Lidong Bing, Wenxuan Zhang, Wai Lam "Exploiting BERT for End-to-End Aspect-based Sentiment Analysis." https://arxiv.org/abs/1910.00883