Introduction to Machine Learning

Final Project – Natural Language Processing

Professor: Dr. Seyedin

Authors:

Name: Amir Azad **Student Number: 9823004**

Name: Erfan Rasti **Student Number: 9823034**

Project Link on GitHub:

[ErfanRasti/MachineLearningProjects: Here I share my codes from a university machine learning course. (github.com)]

Abstract

In this project, we want to evaluate some tweets and classify them as hate or not hate. First, we preprocess and then tokenize it with the special tokenizer of our model. when the data has been prepared, we use the main model. The pre-trained model in this project is DistilBERT. After all, we evaluate the model on the test set using different metrics like accuracy, precision, recall, f1 score, and confusion matrix.

Loading the data

The first step in learning is importing the data. This data is from hugging face and we imported it directly from this source.

Preprocessing the data

This data includes lots of sentences that are in different tweets. These sentences have many unnecessary characters that aren't needed for hate/not hate classification. So we remove these characters: punctuations, whitespaces, and stopwords.

Tokenizing the data

To face different words in the sentences, we should tokenize the words present in the text. The default structure of a tokenizer was 'AutoTokenizer'. As we are using Distil BERT model, we change the tokenizer to 'DistilBertTokenizer'. When we change to tokenizer the padding should be changed. The default value of *padding* in the base code was True. The definition of these two values of *padding* parameter is written below:

- True or 'longest': pad to the longest sequence in the batch (no padding is applied if you only provide a single sequence).
- 'max_length': pad to a length specified by the max_length argument or the maximum length accepted by the model if no max_length is provided (max_length=None). Padding will still be applied if you only provide a single sequence.[1]

According to these definitions, the True value sets the *padding* size to the longest sequence length. But the 'max_length' value sets the *padding* to the maximum sequence that the model can accept. As long as we used DistilBERT and this model cannot accept any value for the *padding* we should set it to a specific length. So we choose the padding parameter as 'max_length'.

A tokenizer also set the truncation to reduce the size of the sequence to the 'max_length' size.

Fine-tuning the model

According to the complexity of the input data and the problem, we should choose the best model. The BERT model has lots of parameters and takes a lot of time to be trained. These extra parameters will cause overfit and the result will not be what we want. One of the best models which have been built based on BERT is Distil BERT. As the name of this model says it was created based on the distilling of the BERT algorithm. DistilBERT is a small, fast, cheap, and light Transformer model trained by distilling BERT base. It has 40% fewer parameters than *bert-base-uncased* and runs 60% faster while preserving over 95% of BERT's performances as measured on the GLUE language understanding benchmark.[2]

After choosing the algorithm we should set the hyperparameters of the model. The model and its hyperparameters are set as below:

```
model = DistilBertForSequenceClassification.from_pretrained(
   'distilbert-base-cased',
   num_labels=2,
   dropout=0.3
   )
```

Code Block 1 - Model Parameters

'`num_labels` ': number of labels we should predict. In this case, we have two categories (hate / not hate).

'`dropout`': to prevent overfitting we should ignore some neuron's weights. So, we use the dropout layer to delete some neurons and generalize the model.

[1] Padding and truncation (huggingface.co)
[2] DistilBERT (huggingface.co)

Training arguments

After setting the model parameters, we should set some arguments which are related to training the model. All of the training arguments which we used are as below:

```
training_args = TrainingArguments(output_dir="test_trainer",
                          evaluation_strategy="epoch",
                          save_strategy="epoch",
                          save_total_limit=1,
                          learning_rate=1e-5,
                          num_train_epochs=3,
                          weight_decay=0.001,
                          per_device_eval_batch_size=8,
                          load_best_model_at_end=True,
                          optim="adamw_torch")
```

Code Block 2 - Training Arguments

'`output_dir`': This argument sets the path to checkpoints that should be saved. We set the folder of saved checkpoints as *test_trainer*

'`evaluation_strategy`': This argument sets the strategy for evaluating the model which we set to *epoch* and the model will be evaluated at the end of each epoch.

'`save_strategy`': The method we want to save the models is at the end of each epoch.

'`save_total_limit`': To decrease the volume of the saved model We decreased the total number of saved models to one, and only the best model will be saved.

'`learning_rate`': To prevent overfitting and change the parameters slowly decrease the learning rate parameter to 10^{-5}.

'`num_train_epochs`': After the methods which we used to prevent overfitting, we should increase the number of training epochs to give extra time to the model for fitting.

'`weight_decay`': The DistilBERT model has been trained with a huge dataset. Fine-tuning will change the representation that the model has been learned before and converge it to the training set. It can lead to overfitting. To prevent this problem, we decrease the weight decay to 0.001.

'`per_device_eval_batch_size`': The validating process is slow. To speed it up we divide the work of validation into the different parts of processing units. So, we set the number of device train batch-size to 8.

'`load_best_model_at_end`': When all epochs have been trained, we probably have overfitting. But in the middle part of epochs, we should have a good model which generalizes well. We load the best model at the end of the training process.

'`optim`': If we run the code without setting this parameter and with the default value, we get this warning:

`FutureWarning: This implementation of AdamW is deprecated and will be removed in a future version. Use the PyTorch implementation torch.optim.AdamW instead`[3]

According to this warning, the *AdamW* optimizer is deprecated; so, it's better to change it to the newer optimizers like *adam_torch*.

Monitor metrics

To monitor the training process, we should define different metrics. Maybe in the middle of the learning process, we figure out that the model has a major problem and we can stop it quickly. To monitor the learning process, we defined some metrics which are demonstrated in Code Block 3.

```
metrics_to_compute = evaluate.combine([
   evaluate.load("accuracy"),
   evaluate.load("precision", average='weighted'),
   evaluate.load("recall", average='weighted'),
   evaluate.load("f1", average='weighted')
])
```

Code Block 3 - Monitor Metrics

'`accuracy`': This metric has lots of applications and any classification problem checks it. but this metric is not enough and sometimes we cannot lean just into it.

'`precision`': Precision shows us how much we could be confident about a prediction that the model has suggested. We used weighted precision to include both labels according to their importance and the number of samples belonging to each label. The weighted average is usually used for unbalanced class labels.

'`recall`': This metric shows us how many labels from the whole number of that specific label is predicted. So, we can evaluate the generalization of the model using this parameter. In this metric, we also used the weighted form to encounter both labels.

'`f1`': f1 is a balance between precision and recall. Weighted f1, calculates the metric for both labels and averages them based on their weights.

According to the results in Table 1 as long as the validation loss has been increased and the training loss got farther from the validation loss, the model has been overfitted to the training set. We could reduce the overfitting by using higher drop-out and lower layers in the model but the model would perform purely on the test set. So, we were forced to take the number of layers high. We could solve this problem by augmenting the data and increasing the complexity of the model. The final result on the test set will be done by the best checkpoint thanks to `load_best_model_at_end`.

[3] Huggingface transformers longformer optimizer warning AdamW - Models - Hugging Face Forums

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
1	0.513000	0.533818	0.734000	0.682540	0.704918	0.693548
2	0.431200	0.573469	0.734000	0.647166	0.829040	0.726899
3	0.397300	0.586434	0.742000	0.667327	0.789227	0.723176

Table 1 - Calculated metrics at the end of each epoch

Evaluation

Finally, to evaluate the model we should use some metrics. In this code, we used accuracy, precision, recall, f1, ROC AUC, ROC curve, and confusion matrix. 4 first metrics are explained in the previous part. 3 last metrics are explained below:

'`roc_auc_score`': Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.[4]

'`roc_curve`': Compute Receiver operating characteristic (ROC).

Note: this implementation is restricted to the binary classification task.[5]

'`confusion_matrix`': Confusion matrix is defined as the number of observations known to be in group i and predict to be in group j.(i, j are the number of rows and columns of the confusion matrix element.)

The final results and the calculated metrics on the test set are as bellow:

```
Accuracy:  0.48484848484848486
Precision:  0.6220240181304175
Recall:  0.48484848484848486
F1:  0.40965059632167344
ROC AUC:  0.5445322408477054
```

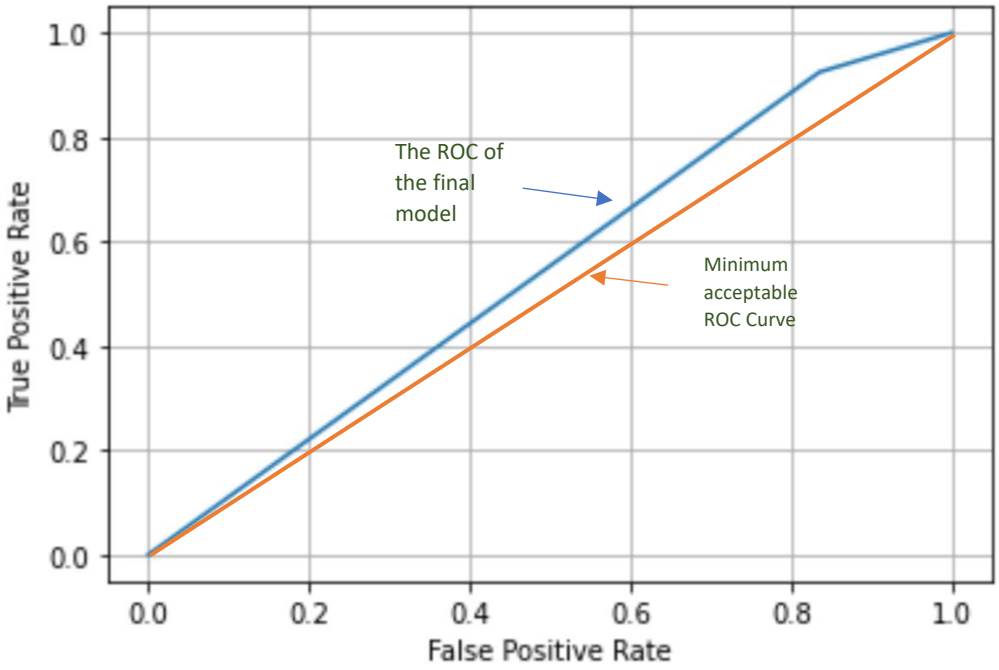
The accuracy is not very high and we could increase it using a more complex model and use better generalization parameters.

The ROC curve has been shown in Plot 1 . according to the plot we can see that the performance is a little bit over the minimum. If the curve is across the down left to top right diagonal the performance will be minimum. The curve is a little bit higher than this station.

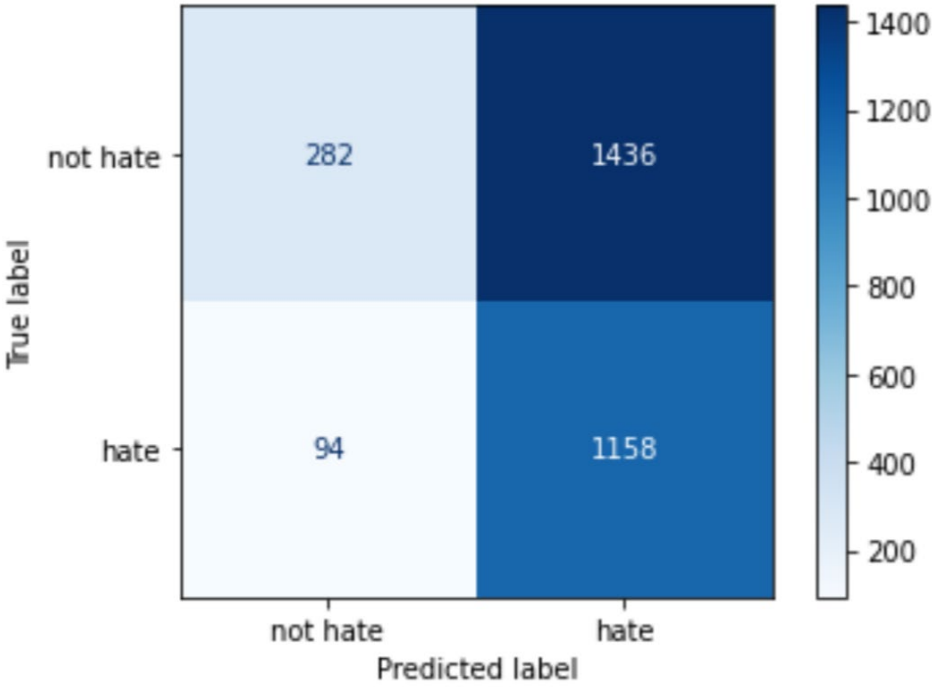
The confusion matrix of the test set has been shown in Plot 2 . according to this plot, we can understand that lots of *not-hate* labels have been confused with *hate*. This is why the performance of the model is low. The recall of the hate label is good but the recall of the not-hate label is terrible.

[4] sklearn.metrics.roc_auc_score — scikit-learn 1.2.0 documentation
[5] sklearn.metrics.roc_curve — scikit-learn 1.2.0 documentation

So overall the recall is not very good. The precision of the *not-hate* label is good but the precision of the *hate* label is not good enough and overall the weighted precision is a little good.



Plot 1 - The ROC Curve



Plot 2 - Confusion matrix of the test set prediction