SENTIMENT ANALYSIS FOR YELP REVIEWS

ELETI RAKSHITH REDDY

NATURAL LANGUAGE PROCESSING(NLP)

DEPARTMENT OF DATA SCIENCE

GEORGE WASHINGTON UNIVERSITY

INDIVIDUAL REPORT

1. INTRODUCTION

The purpose of this project is to predict the Yelp ratings polarity of a local business review, which is either 0 or 1 depending on the review language. We employed natural language processing (NLP) models with a focus on sentiment analysis. Yelp is a review-based website that allows consumers to find and share information about local businesses. When customers make a quick search for a company. When consumers perform a quick search for a business, they are more likely to judge quality only based on the star rating than reading the review text. As a result, we're searching for any links between the phrasing of reviews and the associated rating. We'll look at pre-trained models like DistilBert and Electra, as well as RNN models like LSTM. On the testing dataset, the pipeline involves data cleaning, padding and packing, vectorizing, training, and determining model accuracy. **My role in the project is to run different models and select the best one.**

1.1 DATASET DESCRIPTION

- We used Kaggle to obtain our data.
- The polarity dataset for Yelp reviews is built by treating stars 1 and 2 as negative (label: 0) and 3 and 4 as positive (label: 1).
- There are 559,922 training sample points.
- 37,997 sample points were used in the testing.
- Both labels have an equal number of train and test sample points.
- Review content is the focus of the feature column.
- Label 0 and 1 in the target column.

1.2 ALGORITHM DEVELOPMENT

WHY PRE-TRAINED MODELS?

The current breakthrough in Natural Learning Processing (NLP) is mostly due to transfer learning approaches. It may provide cutting-edge solutions by utilizing pre-trained models, which saves us the time and effort necessary to build huge models.

ELECTRA

Pre-training Text Encoders as Discriminators Rather Than Generators is a project called ELECTRA (Efficiently Learning an Encoder that Classifies Token Replacements Accurately). ELECTRA is a new pretraining method that involves training two transformer models: the generator and discriminator. The generator is trained as a masked language model because its job is to substitute tokens in a sequence. The discriminator, which is the model we're interested in, attempts to determine which tokens in the sequence were substituted by the generator. The generator and discriminator are both present in the ELECTRA checkpoints saved using Google Research's implementation. The user must specify the model to export into the relevant architecture in the conversion script. This implies that the discriminator may be loaded in the ElectraForMaskedLM model, while the generator can be loaded in the ElectraForPreTraining model (the classification head will be arbitrarily set in the generator because it doesn't exist in the discriminator).

DISTILBERT

Operating huge models in on-the-edge and/or under tight computational training or inference budgets remains hard as Transfer Learning from large-scale pre-trained models becomes increasingly prominent in Natural Language Processing (NLP). DistilBERT, a method for pre-training a smaller general-purpose language representation model that can subsequently be fine-tuned to perform well on a variety of tasks, like its bigger equivalents. While most previous research focused on using knowledge distillation to build task-specific models, by using knowledge distillation during the pre-training phase, we can reduce the size of a BERT model by 40% while retaining 97% of its language understanding capabilities and being 60% faster. Also, offers a triple loss that combines language modeling, distillation, and cosine-distance losses to utilize the inductive biases learnt by bigger models during pre-training.

CONVBERT

Language models that have been pre-trained, such as BERT and its derivatives, have lately demonstrated outstanding performance in a variety of natural language interpretation tasks. BERT, on the other hand, is significantly reliant on the global self-attention block, resulting in a huge memory footprint and computation cost. Although all of its attention heads query the whole input sequence to generate the attention map from a global perspective, we discover

that certain heads only need to learn local dependencies, implying that computation redundancy exists. To directly describe local dependencies, a novel span-based dynamic convolution was developed to replace these self-attention heads. Together with the rest of the self-attention heads, the unique convolution heads create a new mixed attention block that is more efficient at both global and local context learning. With this divided focus, BERT was able to create and build a ConvBERT model. ConvBERT beats BERT and its derivatives in a variety of downstream tasks, with lower training costs and fewer model parameters, according to experiments. Surprisingly, the ConvBERTbase model gets a GLUE score of 86.4, 0.7 greater than ELECTRAbase, while requiring just a quarter of the training time.

1.3 WORKING

Since, our aim was to run various pre-trained models on the dataset and then choose the best model to use. DistilBert, ConvBert, and Electra are three pre-trained models on which I worked. To begin, I imported all the necessary libraries, including Numpy, Pandas, Pytorch, Transformers, and Scikit-Learn. The number of epochs I used to run the model was three. The max length that was considered was 200. Libraries used for tokenization and modeling are:
ConvBertTokenizer: It is the same as BertTokenizer in that it performs complete tokenization, including punctuation separation and wordpiece. Use the superclass BertTokenizer for parameter use examples and documentation.
ElectraTokenizer: ElectraTokenizerFast is the same as BertTokenizerFast in that it does complete tokenization, including punctuation splitting and wordpiece.
DistilBertTokenizer: DistilBertTokenizer is the same as BertTokenizer in that it does end-to-end tokenization, including punctuation splitting and wordpiece.
ConvBERT: The basic ConvBERT Model transformer, with no special head on top, outputs raw hidden states. This model is a subclass of PyTorch torch.nn.Module.
ElectraModel: The Electra Model transformer, with no special head on top, outputs raw hidden-states. If the hidden and embedding sizes are different, it utilizes an extra linear layer between the embedding layer and the encoder, like the BERT model. This model allows you to load both the generator and discriminator checkpoints.
DistilBertForSequenceClassification: For GLUE tasks, for example, a DistilBert Model transformer with a sequence classification/regression head on top (a linear layer on top of the pooled output).
Here is the code that was used to get the desired outcomes.

```
from tqdm.auto import tqdm
import pandas as pd
from sklearn.model_selection import train_test_split
from transformers import ElectraTokenizer, DataCollatorWithPadding,
get_scheduler
#from transformers import DistilBertTokenizer, DataCollatorWithPadding,
get_scheduler
#from transformers import ConvBertTokenizer,  DataCollatorWithPadding,
get_scheduler
import torch
```

```
from torch.utils.data import DataLoader
from transformers import ElectraForSequenceClassification, AdamW
#from transformers import DistilBertForSequenceClassification, AdamW
#from transformers import ConvBertForSequenceClassification, AdamW
from sklearn.metrics import accuracy_score
import numpy as np

num_epochs = 3
#checkpoint = 'distilbert-base-uncased'
#checkpoint = 'YituTech/conv-bert-base'
checkpoint = 'google/electra-small-discriminator'
max_len = 200

train_df = pd.read_csv('train.csv')[:20000]
train_df.columns = ['label','text']
train_df['label'] = train_df['label'] - 1
train_texts = train_df['text'].values.tolist()
train_labels = train_df['label'].values.tolist()

test_df = pd.read_csv('test.csv')[:1000]
test_df.columns = ['label','text']
test_df['label'] = test_df['label'] - 1
test_texts = test_df['text'].values.tolist()
test_labels = test_df['label'].values.tolist()

train_texts, val_texts, train_labels, val_labels =
train_test_split(train_texts, train_labels, test_size=.2)
tokenizer = ElectraTokenizer.from_pretrained(checkpoint)
#tokenizer = DistilBertTokenizer.from_pretrained(checkpoint)
#tokenizer = ConvBertTokenizer.from_pretrained(checkpoint)
train_encodings = tokenizer(train_texts, truncation=True, padding=True,
max_length=max_len)
val_encodings = tokenizer(val_texts, truncation=True, padding=True,
max_length=max_len)
test_encodings = tokenizer(test_texts, truncation=True, padding=True,
max_length=max_len)

class YelpDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in
self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = YelpDataset(train_encodings, train_labels)
val_dataset = YelpDataset(val_encodings, val_labels)
test_dataset = YelpDataset(test_encodings, test_labels)


device = torch.device('cuda') if torch.cuda.is_available() else
```

```
torch.device('cpu')

model = ElectraForSequenceClassification.from_pretrained(checkpoint)
#model = DistilBertForSequenceClassification.from_pretrained(checkpoint)
#model = ElectraForSequenceClassification.from_pretrained(checkpoint)
model.to(device)


data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

train_dataloader = DataLoader(train_dataset,
                              shuffle=True, batch_size=8,
collate_fn=data_collator)
eval_dataloader = DataLoader(val_dataset,
                             batch_size=8, collate_fn=data_collator)

optim = AdamW(model.parameters(), lr=5e-5)
num_training_steps = num_epochs * len(train_dataloader)
lr_scheduler = get_scheduler( "linear", optimizer=optim,
                              num_warmup_steps=0,
num_training_steps=num_training_steps)
print(num_training_steps)

progress_bar = tqdm(range(num_training_steps))
model.train()
for epoch in range(num_epochs):
    for batch in train_dataloader:
        optim.zero_grad()
        input_ids = batch['input_ids'].to(device)
        # attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids, labels=labels)
        loss = outputs[0]
        loss.backward()
        optim.step()
        lr_scheduler.step()
        progress_bar.update(1)


PRED = []
Y = []
def update(pred,y):
    PRED.append(pred.detach().cpu().numpy())
    Y.append(y.detach().cpu().numpy())


num_eval_steps = num_epochs * len(eval_dataloader)
progress_bar = tqdm(range(num_eval_steps))

model.eval()
for batch in eval_dataloader:
    with torch.no_grad():
        input_ids = batch['input_ids'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids,labels=labels)

    logits = outputs.logits
```

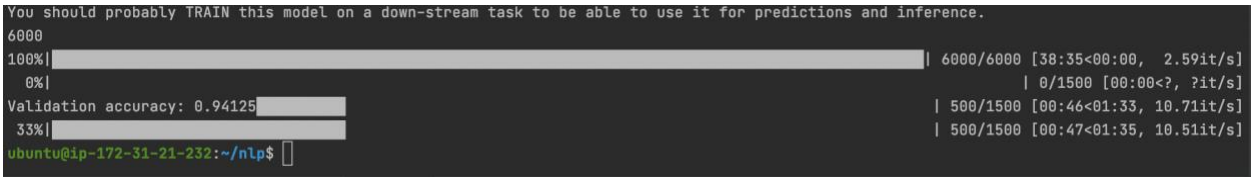
```
    predictions = torch.argmax(logits, dim=-1)
    update(predictions,labels)
    progress_bar.update(1)


Y = np.array(Y).reshape(-1,1)
PRED = np.array(PRED).reshape(-1,1)
res = accuracy_score(Y, PRED)
print("\nValidation accuracy:",res)
```

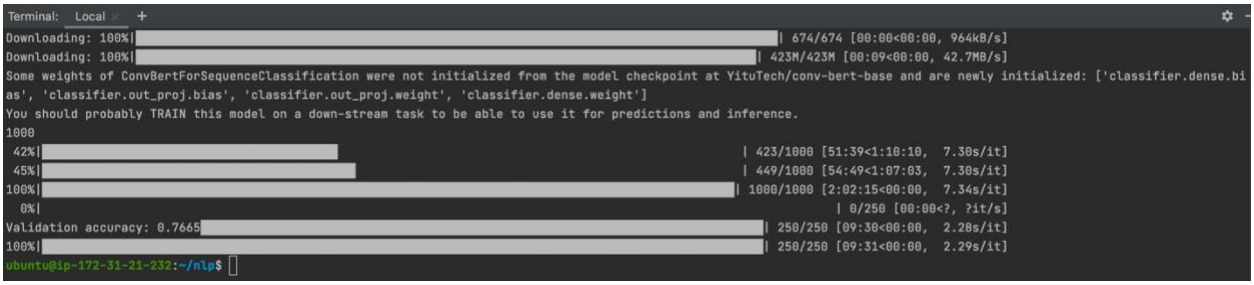
I found the accuracies for each individual model, which will be shown in the results section.

1.4 RESULTS

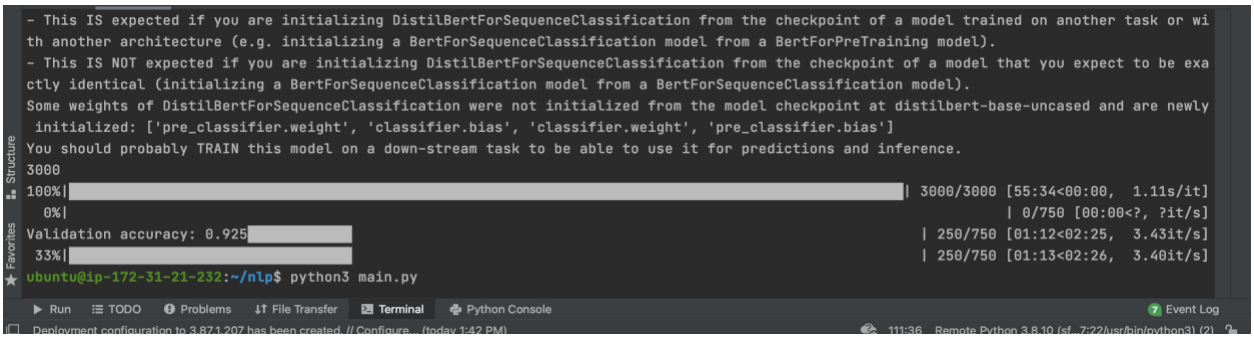
I used the pre-trained model ***"ELECTRA"*** with an epoch size of 3 and achieved a validation accuracy of 94.12%.



I used the ***"ConvBert"*** pre-trained model with epoch size 3 and achieved a validation accuracy of 76.6 %.



I used the ***"DistilBert"*** pre-trained model with epoch size 3 and achieved a validation accuracy of 92.5 %.



1.5 SUMMARY AND CONCLUSION

Finally, I worked on the implementation of three models: electra, distilbert, and convbert, and obtained accuracies of 76 percent, 92 percent, and 94 percent, respectively, which were slightly lower than the other models we were working on for this project. When it came to the end of the project, pre-trained models had far greater accuracy than LSTM, which was clear. For the pre-trained models, we utilized the raw data without any cleaning since they use the structure of the phrase from both sides to connect every output element to every input element.

1.6 PERCENTAGE

Lines of code from internet: 40
Modified: 20
Added: 22
Percentage = (40-20 / 40+22) x 100 = (20/62) x 100 = **32%**

1.7 REFERENCES

1. https://www.kdnuggets.com/2019/09/bert-roberta-distilbert-xlnet-one-use.html
2. http://ankit-ai.blogspot.com/2021/02/understanding-state-of-art-language.html
3. https://huggingface.co/docs/transformers/index