Inter IIT Dev Rev

Akarshan Kapoor B21003

Name and Type of Model

The model is based on the Simple Transformers library by Hugging Faces. This library provides a pre-built infrastructure that can be used for different Natural Language Processing requirements. For this problem statement we were required to build a Question Answering model that could identify whether a given question could be answered from a given set of Passages.

Preprocessing Into The Required Format:

- The training file provided to us was a simple csv that contained different contextual paragraphs and possible questions which could be made.
- 60 percent of the data was utilized for faster training of the model. The data extracted was then divided into a 70:30 train test ratio to ensure proper working of the model.
- This however had to be processed into a list of dictionaries and had to be made in accordance to this format. The given link required that the train, test and the predicted formats be in accordance.
- The following code blocks implements the conversion of the CSV into the required format::

```
#Finding unique Paragraphs across the dataset:
train_inst=train["Paragraph"].unique()

training_data=[]
id=1
#According to each paragraph context the questions were assigned unique IDs and were
tokenized into "is_impossible" and "answer_text" format as required.
for inst in train_inst:
   dictionary={}
   current=train.loc[train['Paragraph'] == inst]
   current=current.reset_index(drop=True)
   dictionary["context"]=inst
   dictionary["qas"]=[]
   for values in range(len(current)):
       qas_dict={}
```

```
  qas_dict["id"]=str(id)
       qas_dict["is_impossible"]=~current["Answer_possible"][values]
       qas_dict["question"]=current["Question"][values]
       qas_dict["answers"]=[]
       if(qas_dict["is_impossible"]==False):
           answer_dict={}
           x=len(current["Answer_text"][values])
           y=len(current["Answer_start"][values])
           answer_dict["text"]=current["Answer_text"][values][2:x-2]
           answer_dict["answer_start"]=int(current["Answer_start"][values][1:y-1])
```

```
            qas_dict["answers"].append(answer_dict)
        id+=1
        dictionary["qas"].append(qas_dict)
    training_data.append(dictionary)
```

Configuration of the model and suggested improvements:

- The model was trained on *distilbert* and the type used was *distilbert-base-case*. The reason why this was chosen is that it is lightweight and uses almost 40 percent less parameters for achieving the same output as bert-base-case and is also 60 percent faster.
- The no. of epochs had to be kept low since the training data file was huge. The learning was kept slower than usual to make up for lesser no.of epochs. The batch size was also kept small for the gradient curve to be updated quickly enough and to minimize the learning loss.
- Also the size output for the no. of answers was limited to 5 for further optimization.

```
# Configuring the model... the model parameters were specified as follows:
model_devrev= QuestionAnsweringArgs()
model_devrev.train_batch_size= 8
model_devrev.evaluate_during_training= False
model_devrev.num_train_epochs=1
model_devrev.reprocess_input_data= True
model_devrev.n_best_size=5
model_devrev.learning_rate=3e-5

finmod=
QuestionAnsweringModel("distilbert","distilbert-base-cased",args=model_devrev,use_cud
a=False)
```

- The model could further be improved by using tokenizers which could make the model training more custom and dynamic. These tokenizers have separate data access methods which can increase output results and decrease training time.

Testing and Simple Predictions:

- The testing was implemented on the last 30 percent of the data as follows:

```
#The model was locally saved into the outputs directory and and was loaded for the
testing part.
finmod=QuestionAnsweringModel("distilbert","outputs/",use_cuda=False)

#The remaining 30 percent of the data was tested upon to check if the model had been
trained properly.
result,text=finmod.eval_model(testing_data)
```

- To make sure that our model was working the following prediction was run:

```
to_predict = [
   {
       "context": "Peter Parker is Spiderman",
       "qas": [
           {
               "question": "Who is Spiderman?",
               "id": "0",
           }
       ],
   }
]
```

- The following output was achieved:

```
[{'id': '0', 'answer': ['Peter Parker', '', 'Parker', 'Peter']}]
[{'id': '0', 'probability': [0.960883934404576, 0.032658683934185036,
0.003435587910957817, 0.0017487183959488306]}]
```

Predictions on the Data Set provided:

- To actually judge the model's abilities it had to be compared against the outputs of the data set provided. Again the provided data set had to be modeled into the required format for proper compiling:

```
test_inst=final_testfile["context"].unique()
finaltesting_data=[]
id=1
for inst in test_inst:
   dictionary={}
   current=final_testfile.loc[final_testfile['context'] == inst]
   current=current.reset_index(drop=True)
   dictionary["context"]=inst
   dictionary["qas"]=[]
   for values in range(len(current)):
       qas_dict={}
       qas_dict["id"]=str(id)
       qas_dict["question"]=current["question"][values]
       id+=1
       dictionary["qas"].append(qas_dict)
   finaltesting_data.append(dictionary)
```

- Also the answers returned by the model were in a dictionary format so they had to converted accordingly :

```
result=[]
result_dict={}
for vals in range(len(answer)):
   if(answer[vals]["answer"][0]!=""):
      result_dict["answer_possible"]=True
      result_dict["answer"]=answer[vals]["answer"][0]
   else:
       result_dict["answer_possible"]=False
       result_dict["answer"]="[]"
   result.append(result_dict)
   result_dict={}
```

- The final accuracy score was calculated using sklearn.metrics.accuracy_score which turned out to be 63 percent. The accuracy achieved was significantly higher than expectation due to the no. of epoch being less and also use of a lesser amount of training data.

Implementation of A Mix of Paragraph matching and Question Answering:

- The more challenging part of the problem statement required that the question had to be answered from a given set of paragraphs, i.e we had to check whether the question could be answered from any of the given set of paragraphs.
- A very basic way to do this was as follows:

```
prediction=[]
for question in questions["question"]:
   final_question=""
   final_context=""
   final_answer=""
   is_possible=""
   for context in paragraphs["context"]:
       to_predict[0]["context"]=context
       to_predict[0]["qas"][0]["question"]=question
       answers, probabilities = finmod.predict(to_predict)
       if(answers[0]["answer"][0]!="" and probabilities[0]["probability"][0] > 0.5):
           final_question=question
           final_context=context
           final_answer=answers[0]["answer"][0]
           is_possible="True"
           break
       else:
           final_question=question
           final_context="[]"
           final_answer="[]"
           is_possible="False"
   prediction.append([final_question,final_context,final_answer,is_possible])
   print("Question done")
   print(final_question)
```

```
print(final_answer)
print(final_context)
print(is_possible)
```

- The above method crudely checks for the output of a question on every instance of the paragraph provided. Then it checks if the probability of prediction is greater than 50 percent to append the answer else it marks the question unanswerable.The above model could not be completely run due to less time.
- A Suggested improvement on the above would be to use Named Entity Recognition, another NLP module provided for picking up the key term from given paragraphs. These key terms or themes can then be compared to a question to rule out if a question does belong to the contextual theme or not thus improving both accuracy and computation time.