Model Specific Tasks

Mahendar Byra
20MCME25

December 28, 2023

Contents

1 Tasks **2**

2 About the Model **2**
2.1 Logits 2
2.2 Activation Function : Softmax 3
2.3 Rounding the Probability with argmax 3

3 Feature Extraction and Train-test-split **3**

4 Model Training **4**
4.1 Load the Tokenizer and Pre-trained model 4
4.2 Encodings and CustomDataSets 4
4.3 Training Arguments and Trainer 4
4.4 Train and Save the model 5

5 Model Testing **5**
5.1 Load the Saved model and tokenizer 5
5.2 Predict the logits 6
5.3 Apply softmax and argmax 6

6 Evaluation : Evaluation Metrics **6**

7 Conclusion **7**

8 GPU utilization **7**

9 Github repository and References **8**

1 Tasks

I have been allotted, the **Model creation and Training** part of our project. Primarily I have been tasked with the following:

1. Feature Extraction and Train-test-split
2. Model Training
3. Model Testing.
4. Model Evaluation : Evaluation Metrics of the Model

2 About the Model

I have used the **BertForSequenceClassification** Model . It is a pre-trained BERT model designed specifically for sequence classification tasks. It extends the BERT architecture to handle tasks where the goal is to classify sequences of tokens into predefined categories. This type of model is well-suited for tasks like sentiment analysis, text categorization, and, in your case, log sequence classification.

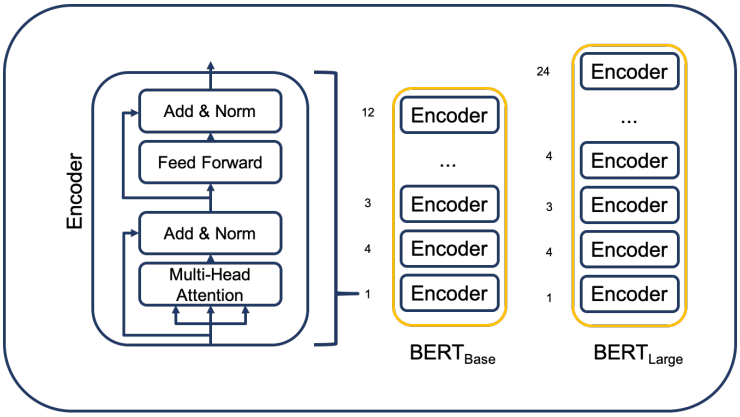

Figure 1: Bert Model.

2.1 Logits

In the BertForSequenceClassification model, the final layer generates a set of logits for each class. Logits can be interpreted as raw scores representing the model's confidence in assigning a given input sequence to each class. The class with the highest logit is predicted as the output class for the sequence.

2.2 Activation Function : Softmax

To convert logits to probabilities, the softmax activation function is applied, transforming raw logits into a probability distribution.

2.3 Rounding the Probability with argmax

The predicted class is determined by selecting the class with the highest probability after applying softmax, a process known as argmax:

Predicted Class=argmax(softmax(logits))

The argmax operation selects the class index with the maximum probability, making it the predicted class for the input sequence.

3 Feature Extraction and Train-test-split

The dataset has two columns
1.Event sequences.
2.Label.
The data is divided into input(X) and output(y) using dataframes . Using train-test-split() the data is divided into training data and testing data,Train data is 80% and test data is 20% . Seed value is used to produce the same split in each run . The code segment:

```
import pandas
# Load the Data
df = pd.read_csv('HDFS_sequence.csv', sep=',', quotechar='"', names
    =["text", "label"])

# Feature extraction
X = list(df['text'])
y = list(df['label'])

# Get dummies(mapping)
y = pd.get_dummies(y, drop_first=True)['Normal']
y = y.astype(int)

for x in range(len(y)):
  if y[x] == 0:
    y[x] = 1
  else:
    y[x] = 0

"""## Train-test split ##"""

X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,
    random_state=0)
```

4 Model Training

4.1 Load the Tokenizer and Pre-trained model

In this step, I have loaded the pre-trained **BertForSequenceClassification** model and corresponding tokenizer from the transformers library. Later this will be used for training. Here I have used "bert-base-uncased" model which has 12 encoders.

```
from transformers import BertTokenizer,
    BertForSequenceClassification

model_name = "bert-base-uncased"
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertForSequenceClassification.from_pretrained(model_name)
```

4.2 Encodings and CustomDataSets

Using the tokenizer, get the encodings for the test input and train input, Then pass the encodings along with labels through CustomDataset class to get Datasets which will be used in the Training step.

```
    train_encodings = tokenizer(X_train, truncation=True, padding=
    True)
    test_encodings = tokenizer(X_test, truncation=True, padding=
    True)

    # Create datasets
class CustomDataset(Dataset):
    def _init_(self, inputs, labels):
        self.inputs = inputs
        self.labels = labels

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

    def _getitem_(self, idx):
        input_ids = self.inputs['input_ids'][idx]
        attention_mask = self.inputs['attention_mask'][idx]
        label = torch.tensor(self.labels.iloc[idx])

        return {'input_ids': input_ids, 'attention_mask':
    attention_mask, 'label': label}

# Create an instance of the custom dataset
dataset_train = CustomDataset(train_encodings, y_train)
dataset_test = CustomDataset(test_encodings, y_test)
```

4.3 Training Arguments and Trainer

The training arguments is defined, specifying parameters such as batch size(2), evaluation strategy(steps), and the number of training epochs(10). These parameters impact how the model is trained.It also specifies where the logs should

be saved.

The Trainer class is used train the BERT model on the dataset based on the defined training arguments.

```
# Training Arguments
training_args = TrainingArguments(
    output_dir="./bert_base_model",
    evaluation_strategy="steps",
    eval_steps=100,
    per_device_train_batch_size=2,
    per_device_eval_batch_size=2,
    save_steps=1000,
    save_total_limit=2,
    num_train_epochs=10,
    logging_dir="./logs",
)

# Trainer
trainer = Trainer(
    model=model,                         # the instantiated
    Transformers model to be trained
    args=training_args,                  # training arguments,
    defined above
    train_dataset=dataset_train,         # training dataset
    eval_dataset=dataset_test             # evaluation dataset
)
```

4.4 Train and Save the model

The trainer will train the model using the train() function ,later this model and tokenizer will be saved using save_pretrained() function. The files regardinding the model will be saved in the respecive path.

```
 #Training
trainer.train()

# Save the Trained Model
model.save_pretrained("./fine_tuned_bert_model_for_HDFS")
tokenizer.save_pretrained("./fine_tuned_bert_model_for_HDFS")
```

Now the Model is ready to use.

5 Model Testing

5.1 Load the Saved model and tokenizer

In the Training step ,we have saved our trained model in a specific path. Now load the saved model and tokenizer using that path.After loading , the model is ready for the prediction for the new input.
The Model path should be same as where we have saved it.

```
model_path = "./fine_tuned_bert_model_for_HDFS"
model_saved = BertForSequenceClassification.from_pretrained(
    model_path)
tokenizer_saved = BertTokenizer.from_pretrained(model_path)
```

5.2 Predict the logits

We already have the test_encodings from the training process or we can get them by passing the test data to saved tokenizer. Now pass the test_encodings to the model then call logits, It will produce the respective logits for the test data.

```
test_encodings = tokenizer_saved(X_test, truncation=True, padding=
    True, return_tensors="pt")
predictions = model_saved(**test_encodings)
logits = predictions.logits
```

5.3 Apply softmax and argmax

Once we got the logits ,as discussed earlier we pass them to softmax which produces probabilities then pass these probabilities to argmax which will produce the classification . In our case (0 or 1). 0 means **Normal** log and 1 means **Anomalous** log.

```
# Apply softmax to get probabilities
probabilities = F.softmax(logits, dim=1)

# Get the predicted label (0 or 1)
predicted_labels = torch.argmax(probabilities, dim=1)
predicted_labels = predicted_labels.tolist()
# Print the predicted labels
print(predicted_labels)
```

6 Evaluation : Evaluation Metrics

We can evaluate our model based on the following metrics

- Accuracy
- Precision
- Recall
- F1 score

To get these values we need to calculate the confusion matrix.
The following code produces the metrics.
The actual metrics for all the datasets is calculated by sushanth.
I have repeated the same Training ,Testing and Evaluation for all the datasets.

```
"""## Evaluation ##"""
y_test_labels = y_test.tolist()
# Create confusion matrix
cm = confusion_matrix(y_test_labels, predicted_labels)

# Calculate metrics
accuracy = accuracy_score(y_test_labels, predicted_labels)
precision = precision_score(y_test_labels, predicted_labels)
recall = recall_score(y_test_labels, predicted_labels)
f1 = f1_score(y_test_labels, predicted_labels)

print("Confusion Matrix:")
print(cm)
print("\nAccuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

7 Conclusion

In conclusion, the training phase involved the utilization of a pre-trained BERT model and tokenizer to learn patterns from the provided log data. Testing showcased the model's ability to make predictions on new sequences, while evaluation metrics such as accuracy, precision, recall, and F1 score showed its performance. Through training and testing, the fine-tuned BERT model demonstrated effectiveness in log sequence classification. All the above-mentioned code has been shared with Sathwik, he performs model training and aggregates the models within the federated environment.

8 GPU utilization

Here we have discussed the training and prediction without GPU involvement. If your system has good GPU ,then the code which uses GPU is also provided in the Github repository. To access GPU we need to transfer the model and data to the GPU. The training and testing will be done quickly with GPU by parellel processing.

Step 1 : Access GPU

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
    ")
```

Step 2 : Transfer the Model to GPU

```
model_name = "bert-base-uncased"
tokenizer = BertTokenizer.from_pretrained(model_name)
```

```
model = BertForSequenceClassification.from_pretrained(model_name).
    to(device)
```

Step 3 : Convert the data into tensors then transfer it to GPU

```
# Get the Encodings
train_encodings = tokenizer(X_train, truncation=True, padding=True,
     return_tensors="pt")
test_encodings = tokenizer(X_test, truncation=True, padding=True,
    return_tensors="pt")

# Move data to GPU
train_encodings = {key: value.to(device) for key, value in
    train_encodings.items()}
test_encodings = {key: value.to(device) for key, value in
    test_encodings.items()}
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long).to(
    device)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.long).to(
    device)
# Create an instance of the custom dataset
dataset_train = CustomDataset(train_encodings, y_train_tensor)
dataset_test = CustomDataset(test_encodings, y_test_tensor)
```

In the provided code, train_encodings and test_encodings are instances of the BatchEncoding class, which is part of the transformers library. These objects are implicitly handled by PyTorch and don't need explicit conversion to tensors.

On the other hand, y_train and y_test are pandas Series, and you need to explicitly convert them to PyTorch tensors using torch.tensor()
Now train the model with dataset_train and dataset_test. In the loading of saved model ,prediction also use the .to(device) code to use GPU.

9 Github repository and References

Github link : https://github.com/sushanthk-262/Log_anomaly_detection
References : BertForSequenceClassification from Hug-

gingface transformers