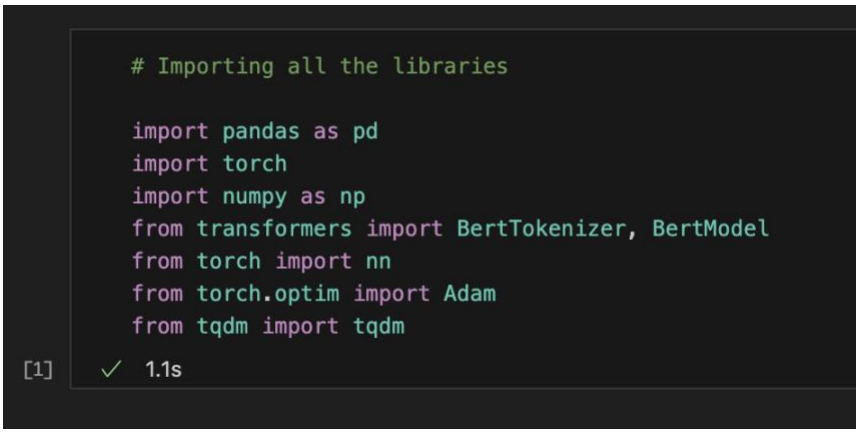
CEO product-related classifier using BERT

I developed a text-classifier model to categorize the CEO's announcements, using the BERT language model implemented using PyTorch.

Imports used for the code:



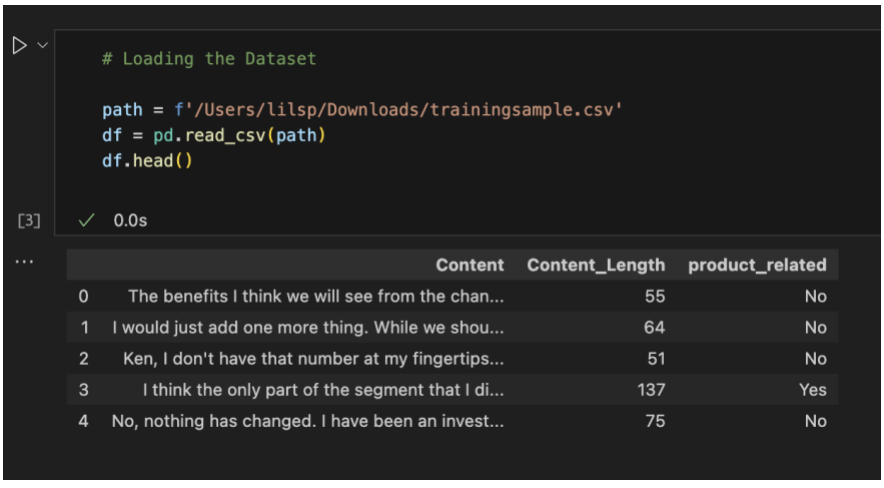
```
# Importing all the libraries

import pandas as pd
import torch
import numpy as np
from transformers import BertTokenizer, BertModel
from torch import nn
from torch.optim import Adam
from tqdm import tqdm
```

[1] ✓ 1.1s

The BertTokenizer is the tokenizer that oversees preparing the inputs for the model and making them more suitable for the model to process.

I have also implemented the Adam algorithm in Pytorch, which is a method of stochastic optimization.

Managing and preprocessing the dataset:



```
# Loading the Dataset

path = f'/Users/lilsp/Downloads/trainingsample.csv'
df = pd.read_csv(path)
df.head()
```

[3] ✓ 0.0s

	Content	Content_Length	product_related
0	The benefits I think we will see from the chan...	55	No
1	I would just add one more thing. While we shou...	64	No
2	Ken, I don't have that number at my fingertips...	51	No
3	I think the only part of the segment that I di...	137	Yes
4	No, nothing has changed. I have been an invest...	75	No

I have used Pandas to load the dataset given and display the first few entries.

```python
# Preprocessing the data

tokenizer = BertTokenizer.from_pretrained('bert-base-cased')

labels = {'No':0,
          'Yes':1,
          }

class Dataset(torch.utils.data.Dataset):

    def __init__(self, df):

        self.labels = [labels[label] for label in df['product_related']]
        self.texts = [tokenizer(text,
                                padding='max_length', max_length = 512, truncation=True,
                                 return_tensors="pt") for text in df['Content']]

    def classes(self):
        return self.labels

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

    def get_batch_labels(self, idx):
        # Fetch a batch of labels
        return np.array(self.labels[idx])

    def get_batch_texts(self, idx):
        # Fetch a batch of inputs
        return self.texts[idx]

    def __getitem__(self, idx):

        batch_texts = self.get_batch_texts(idx)
        batch_y = self.get_batch_labels(idx)

        return batch_texts, batch_y
```

[4] ✓ 0.3s

The above code consists of declaring the tokenizer variable, the labels (“yes” and “no” for the categorization). I have also created a Dataset class, which is responsible for the data preprocessing and tokenizing the labels and the inputs.

After some basic preprocessing of the data, I also split the data into the training, validation, and test sets.

Building, training, and evaluating the model:

I implemented the BERT model by creating a BertClassifier class that takes care of the various layers that are part of the BERT learning model.

I did some fine-tuning and found relatively suitable values for the batch size, epochs, and learning rate that should end up giving the ideal prediction accuracy, without underfitting or overfitting the model.

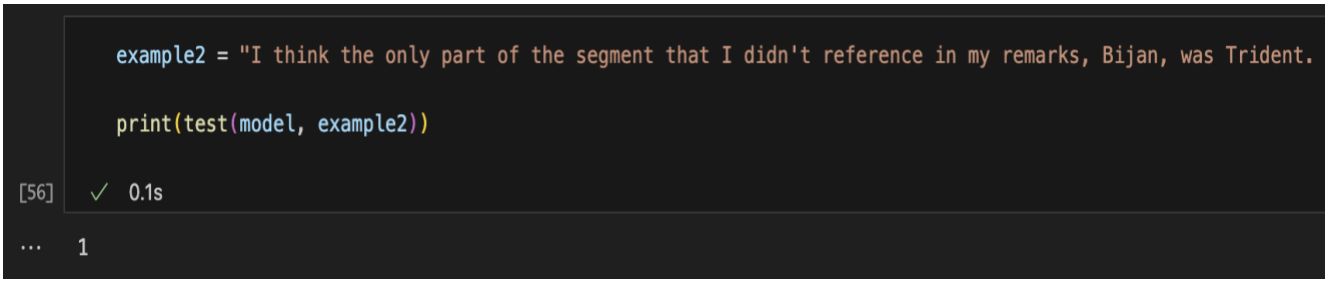
I decided to go with a batch size of 2, since it provided me with a faster running model and was relatively more accurate. Since, the batch size and learning rate can be theoretically taken as having a linear relationship, I chose a suitable learning rate of 1e-6.

```
100%|          | 120/120 [00:56<00:00,  2.11it/s]
Epochs: 1 | Train Loss:  0.352 | Train Accuracy:  0.562 | Val Loss:  0.352 | Val Accuracy:  0.600
100%|          | 120/120 [00:56<00:00,  2.12it/s]
Epochs: 2 | Train Loss:  0.354 | Train Accuracy:  0.558 | Val Loss:  0.319 | Val Accuracy:  0.700
100%|          | 120/120 [24:36<00:00, 12.30s/it]
Epochs: 3 | Train Loss:  0.343 | Train Accuracy:  0.554 | Val Loss:  0.350 | Val Accuracy:  0.400
100%|          | 120/120 [00:54<00:00,  2.21it/s]
Epochs: 4 | Train Loss:  0.303 | Train Accuracy:  0.688 | Val Loss:  0.316 | Val Accuracy:  0.600
100%|          | 120/120 [00:52<00:00,  2.28it/s]
Epochs: 5 | Train Loss:  0.279 | Train Accuracy:  0.700 | Val Loss:  0.329 | Val Accuracy:  0.600
100%|          | 120/120 [00:52<00:00,  2.28it/s]
Epochs: 6 | Train Loss:  0.270 | Train Accuracy:  0.767 | Val Loss:  0.323 | Val Accuracy:  0.567
100%|          | 120/120 [00:52<00:00,  2.28it/s]
Epochs: 7 | Train Loss:  0.235 | Train Accuracy:  0.854 | Val Loss:  0.285 | Val Accuracy:  0.700
100%|          | 120/120 [00:52<00:00,  2.28it/s]
Epochs: 8 | Train Loss:  0.212 | Train Accuracy:  0.883 | Val Loss:  0.284 | Val Accuracy:  0.667
100%|          | 120/120 [00:52<00:00,  2.28it/s]
Epochs: 9 | Train Loss:  0.206 | Train Accuracy:  0.875 | Val Loss:  0.297 | Val Accuracy:  0.633
100%|          | 120/120 [00:52<00:00,  2.28it/s]
Epochs: 10 | Train Loss:  0.172 | Train Accuracy:  0.938 | Val Loss:  0.298 | Val Accuracy:  0.600
```

Next, the evaluate() method uses the trained model to make predictions on the test set. I got an accuracy of 93.4%, but this varies due to the randomness of the training process and the splitting of the datasets.

```
    evaluate(model, df_test)
[72] ✓ 2.3s

...  Test Accuracy: 0.934
```

Finally, I created the test() function that implements all the above methods and classes to provide the user with a Boolean output (0 or 1), on whether or not the inputted sentence is product-related or not.



NOTES:

1. Since, I am using a MAC, I had to implement the MPS framework for accelerated Pytorch learning.
2. Though I am happy with the current accuracy of the predictions, I believe that I can fine-tune the parameters and develop a more accurate model. However, due to the time constraint, I am happy to settle with what I have right now.