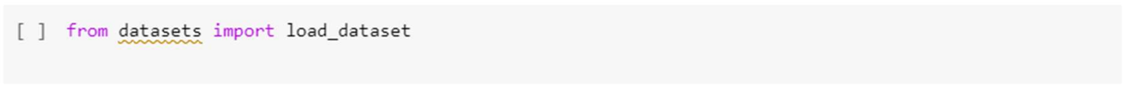
Report

Assignment – Multi-Label Classification

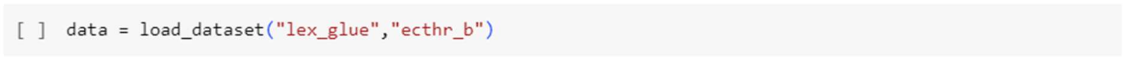
Dataset:

Ecthr_b - The European Court of Human Rights (ECtHR) hears allegations that a state has breached human rights provisions of the European Convention of Human Rights (ECHR). For each case, the dataset provides a list of factual paragraphs (facts) from the case description. Each case is mapped to articles of ECHR that were allegedly violated (considered by the court).

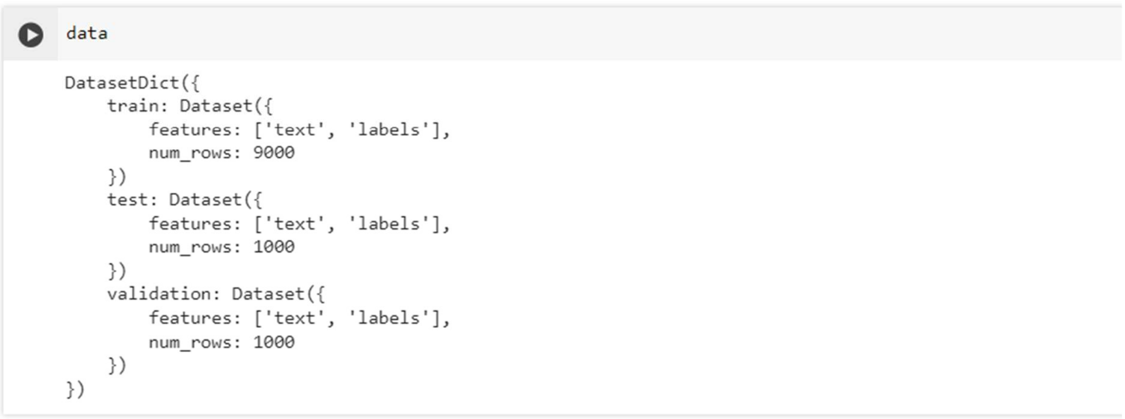
The train part contains 9k rows, validation contains 1k and the test contains 1k rows. There are total 10 classes (0 to 9).



```
[ ] from datasets import load_dataset
```

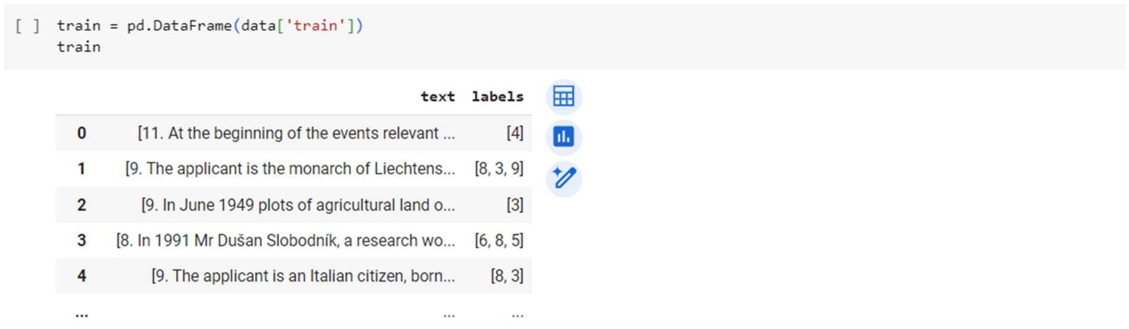


```
[ ] data = load_dataset("lex_glue","ecthr_b")
```



```
data

DatasetDict({
    train: Dataset({
        features: ['text', 'labels'],
        num_rows: 9000
    })
    test: Dataset({
        features: ['text', 'labels'],
        num_rows: 1000
    })
    validation: Dataset({
        features: ['text', 'labels'],
        num_rows: 1000
    })
})
```

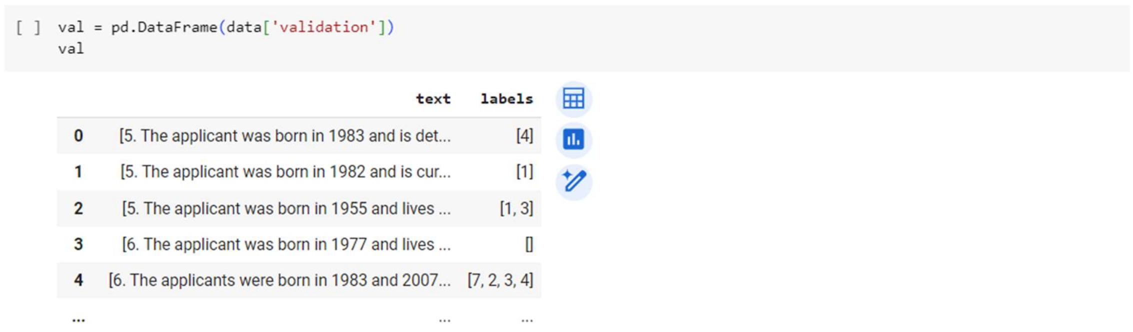
Train:



```
[ ] train = pd.DataFrame(data['train'])
    train
```

	text	labels
0	[11. At the beginning of the events relevant ...	[4]
1	[9. The applicant is the monarch of Liechtens...	[8, 3, 9]
2	[9. In June 1949 plots of agricultural land o...	[3]
3	[8. In 1991 Mr Dušan Slobodník, a research wo...	[6, 8, 5]
4	[9. The applicant is an Italian citizen, born...	[8, 3]
...	...	...

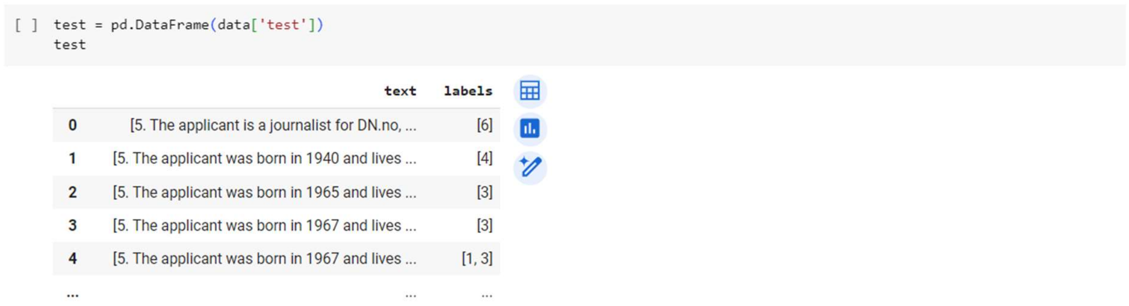
Validation:



```
val = pd.DataFrame(data['validation'])
val
```

	text	labels
0	[5. The applicant was born in 1983 and is det...	[4]
1	[5. The applicant was born in 1982 and is cur...	[1]
2	[5. The applicant was born in 1955 and lives ...	[1, 3]
3	[6. The applicant was born in 1977 and lives ...	[]
4	[6. The applicants were born in 1983 and 2007...	[7, 2, 3, 4]
...	...	...

Test:

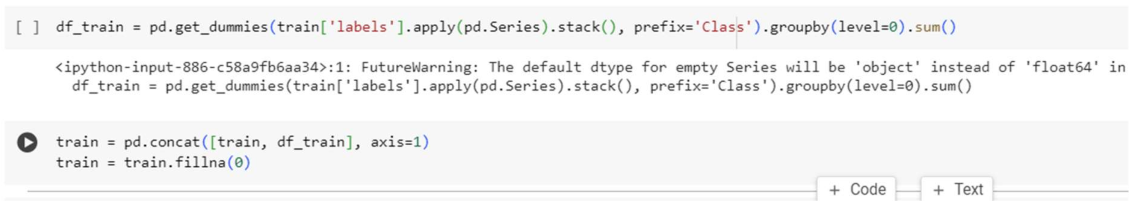


```
test = pd.DataFrame(data['test'])
test
```

	text	labels
0	[5. The applicant is a journalist for DN.no, ...	[6]
1	[5. The applicant was born in 1940 and lives ...	[4]
2	[5. The applicant was born in 1965 and lives ...	[3]
3	[5. The applicant was born in 1967 and lives ...	[3]
4	[5. The applicant was born in 1967 and lives ...	[1, 3]
...	...	...

Data Preprocessing:

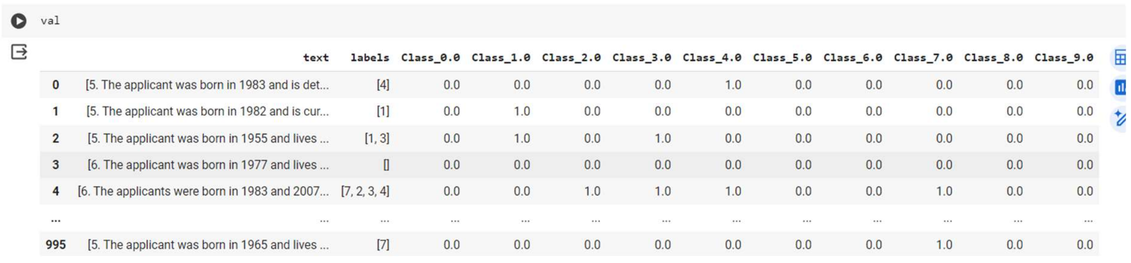
Making the one-hot encoding for the label values for all train, validation and test part



```
df_train = pd.get_dummies(train['labels'].apply(pd.Series).stack(), prefix='Class').groupby(level=0).sum()
```

```
<ipython-input-886-c58a9fb6aa34>:1: FutureWarning: The default dtype for empty Series will be 'object' instead of 'float64' in
  df_train = pd.get_dummies(train['labels'].apply(pd.Series).stack(), prefix='Class').groupby(level=0).sum()
```

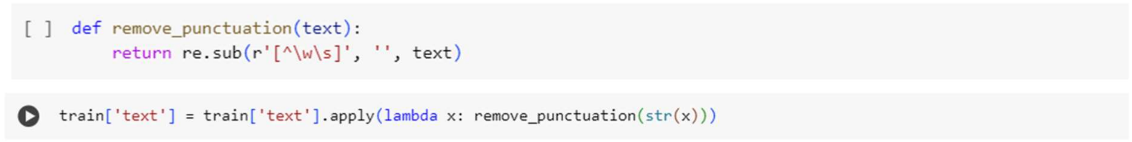
```
train = pd.concat([train, df_train], axis=1)
train = train.fillna(0)
```



```
val
```

	text	labels	Class_0.0	Class_1.0	Class_2.0	Class_3.0	Class_4.0	Class_5.0	Class_6.0	Class_7.0	Class_8.0	Class_9.0
0	[5. The applicant was born in 1983 and is det...	[4]	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
1	[5. The applicant was born in 1982 and is cur...	[1]	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	[5. The applicant was born in 1955 and lives ...	[1, 3]	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
3	[6. The applicant was born in 1977 and lives ...	[]	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	[6. The applicants were born in 1983 and 2007...	[7, 2, 3, 4]	0.0	0.0	1.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...
995	[5. The applicant was born in 1965 and lives ...	[7]	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0

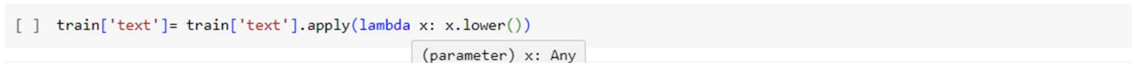
1. Removing the punctuation



```
def remove_punctuation(text):
    return re.sub(r'[^\w\s]', '', text)
```

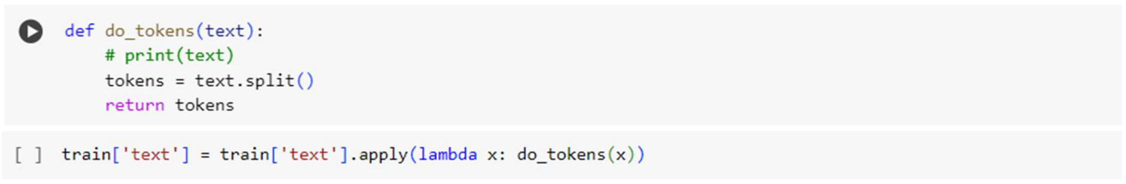
```
train['text'] = train['text'].apply(lambda x: remove_punctuation(str(x)))
```

2. Lowering the text



```
[ ] train['text']= train['text'].apply(lambda x: x.lower())
                                              (parameter) x: Any
```

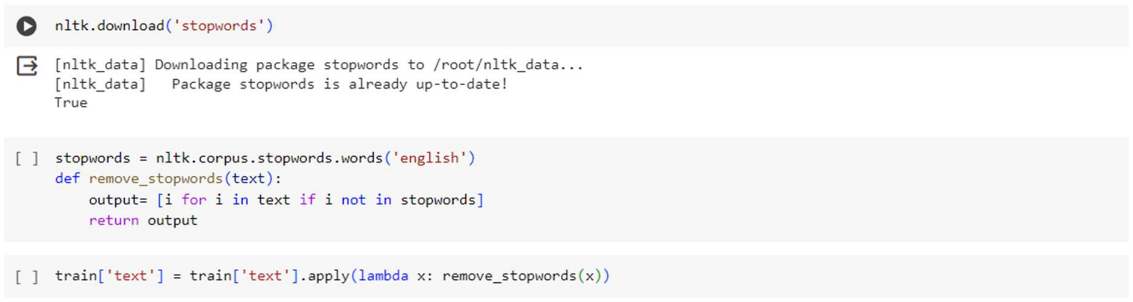
3. Splitting the text into tokens



```
def do_tokens(text):
    # print(text)
    tokens = text.split()
    return tokens
```

```
[ ] train['text'] = train['text'].apply(lambda x: do_tokens(x))
```

4. Removing the stopwords



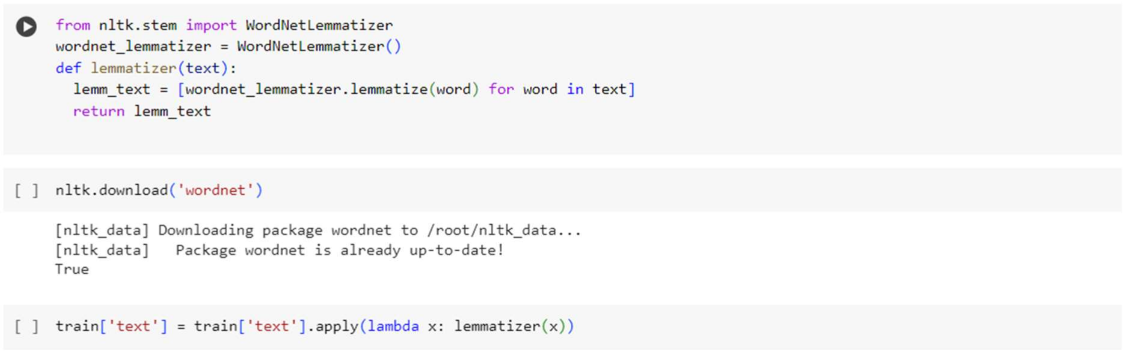
```
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
True
```

```
[ ] stopwords = nltk.corpus.stopwords.words('english')
    def remove_stopwords(text):
        output= [i for i in text if i not in stopwords]
        return output
```

```
[ ] train['text'] = train['text'].apply(lambda x: remove_stopwords(x))
```

5. Lemmatizing the tokens



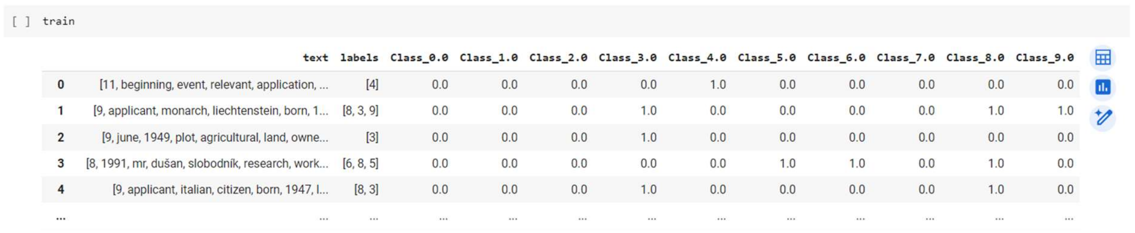
```
from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()
def lemmatizer(text):
  lemm_text = [wordnet_lemmatizer.lemmatize(word) for word in text]
  return lemm_text
```

```
[ ] nltk.download('wordnet')
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
True
```

```
[ ] train['text'] = train['text'].apply(lambda x: lemmatizer(x))
```

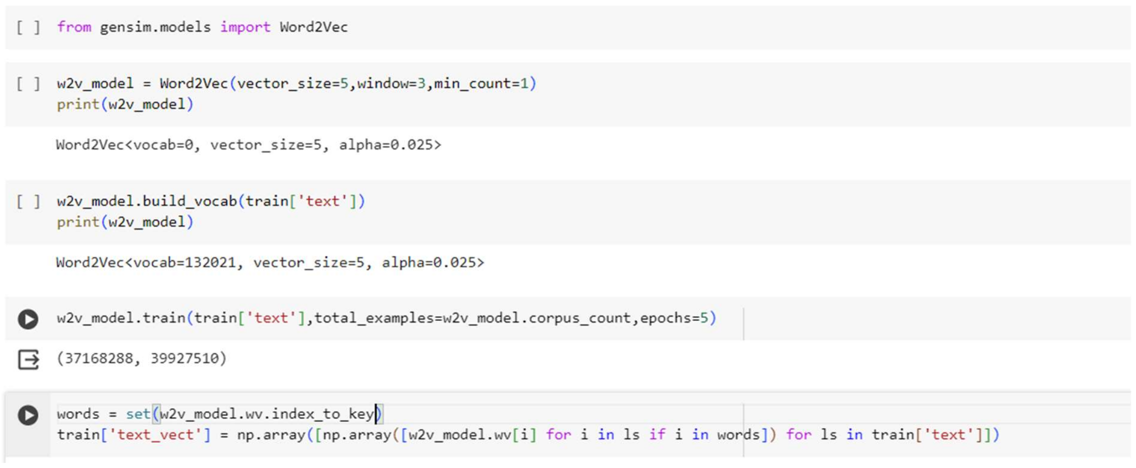
Final pre-processed data:



```
[ ] train
```

	text	labels	Class_0.0	Class_1.0	Class_2.0	Class_3.0	Class_4.0	Class_5.0	Class_6.0	Class_7.0	Class_8.0	Class_9.0
0	[11, beginning, event, relevant, application, ...	[4]	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
1	[9, applicant, monarch, liechtenstein, born, 1...	[8, 3, 9]	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0
2	[9, june, 1949, plot, agricultural, land, owne...	[3]	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
3	[8, 1991, mr, dušan, slobodník, research, work...	[6, 8, 5]	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	0.0
4	[9, applicant, italian, citizen, born, 1947, l...	[8, 3]	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...

Now using, the genism Word2Vec for producing the embeddings for the pre-processed text.



```
[ ] from gensim.models import Word2Vec
```

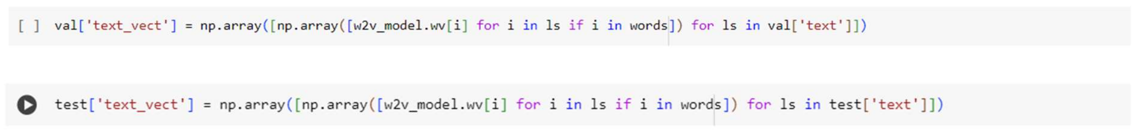
```
[ ] w2v_model = Word2Vec(vector_size=5,window=3,min_count=1)
    print(w2v_model)

    Word2Vec<vocab=0, vector_size=5, alpha=0.025>
```

```
[ ] w2v_model.build_vocab(train['text'])
    print(w2v_model)

    Word2Vec<vocab=132021, vector_size=5, alpha=0.025>
```

```
w2v_model.train(train['text'],total_examples=w2v_model.corpus_count,epochs=5)

(37168288, 39927510)
```

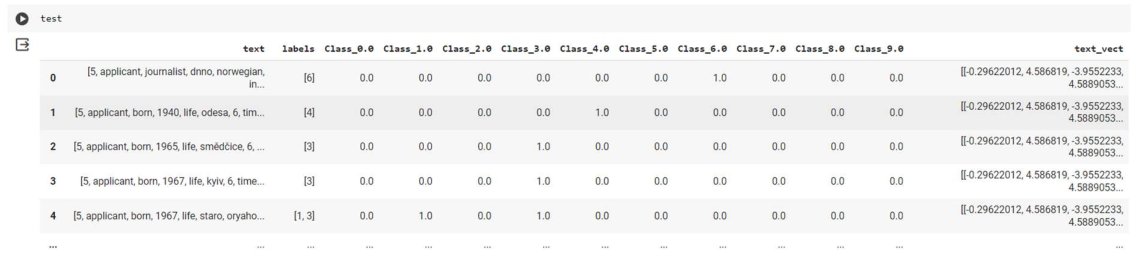
```
words = set(w2v_model.wv.index_to_key)
train['text_vect'] = np.array([np.array([w2v_model.wv[i] for i in ls if i in words]) for ls in train['text']])
```

Now, training the train text to produce the vocab that can be further used to change the val and test data to numeric data.



```
[ ] val['text_vect'] = np.array([np.array([w2v_model.wv[i] for i in ls if i in words]) for ls in val['text']])
```

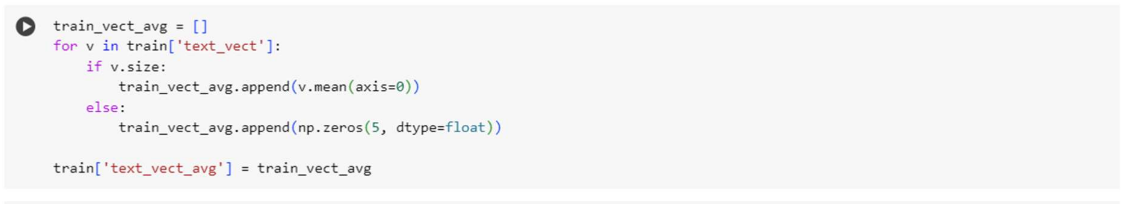
```
test['text_vect'] = np.array([np.array([w2v_model.wv[i] for i in ls if i in words]) for ls in test['text']])
```



```
test
```

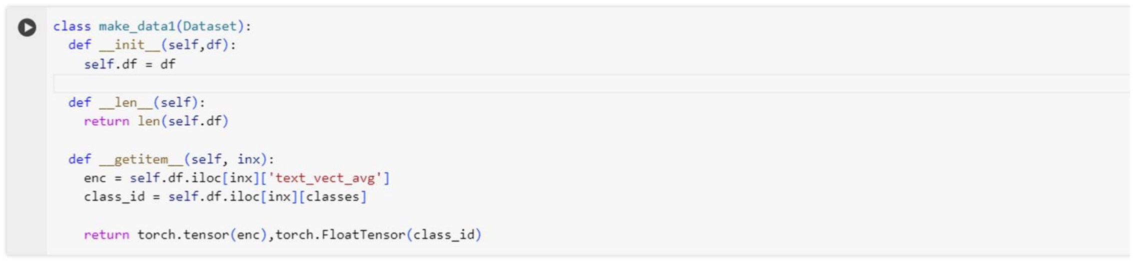
	text	labels	Class_0.0	Class_1.0	Class_2.0	Class_3.0	Class_4.0	Class_5.0	Class_6.0	Class_7.0	Class_8.0	Class_9.0	text_vect
0	[5, applicant, journalist, dnno, norwegian, in...	[6]	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	[[-0.29622012, 4.586819, -3.9552233, 4.5889053...
1	[5, applicant, born, 1940, life, odesa, 6, tim...	[4]	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	[[-0.29622012, 4.586819, -3.9552233, 4.5889053...
2	[5, applicant, born, 1965, life, smědčice, 6, ...	[3]	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	[[-0.29622012, 4.586819, -3.9552233, 4.5889053...
3	[5, applicant, born, 1967, life, kyiv, 6, time...	[3]	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	[[-0.29622012, 4.586819, -3.9552233, 4.5889053...
4	[5, applicant, born, 1967, life, staro, oryaho...	[1, 3]	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	[[-0.29622012, 4.586819, -3.9552233, 4.5889053...
...	...	...	...	...	...	...	...	...	...	...	...	...	...

Now, to make the embeddings of the same length taking the mean across the axis=0

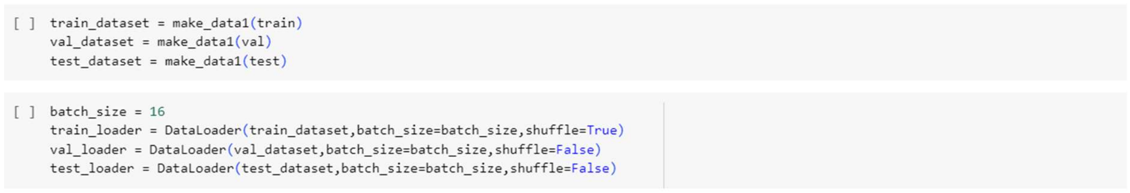


```
train_vect_avg = []
for v in train['text_vect']:
    if v.size:
        train_vect_avg.append(v.mean(axis=0))
    else:
        train_vect_avg.append(np.zeros(5, dtype=float))

train['text_vect_avg'] = train_vect_avg
```

Loading the data and making the dataloader with batch size of 16



```
class make_data1(Dataset):
  def __init__(self,df):
    self.df = df

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

  def __getitem__(self, inx):
    enc = self.df.iloc[inx]['text_vect_avg']
    class_id = self.df.iloc[inx][classes]

    return torch.tensor(enc),torch.FloatTensor(class_id)
```

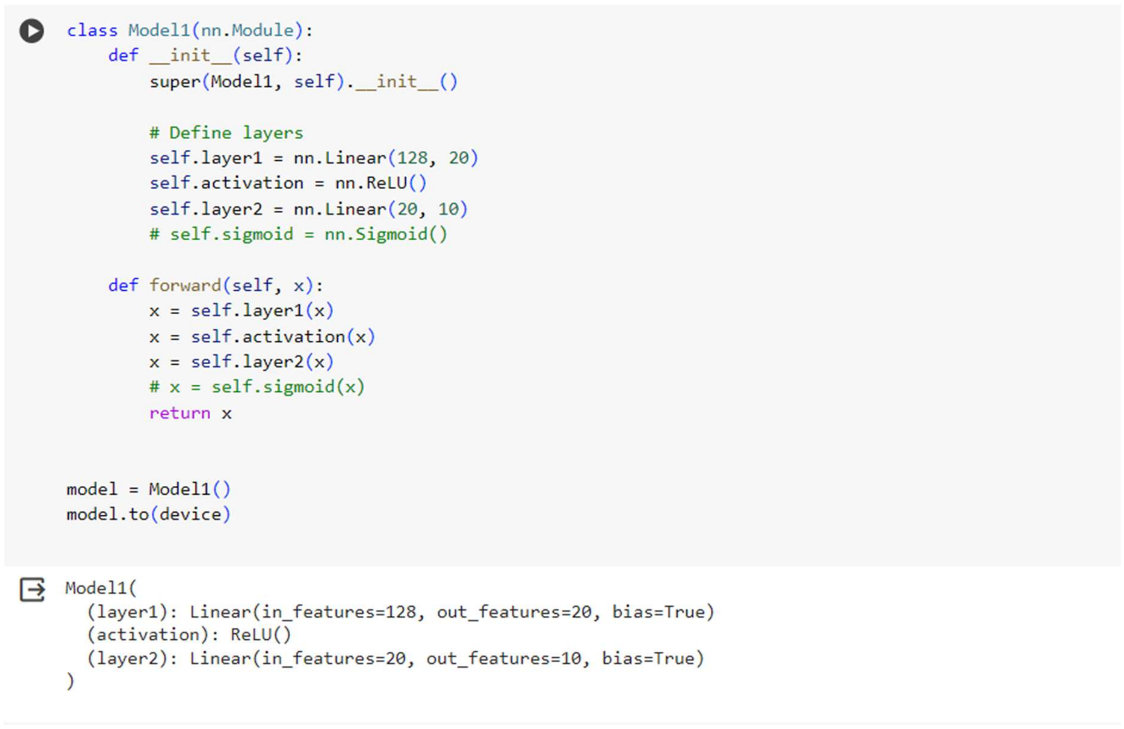


```
[ ]  train_dataset = make_data1(train)
     val_dataset = make_data1(val)
     test_dataset = make_data1(test)
```

```
[ ]  batch_size = 16
     train_loader = DataLoader(train_dataset,batch_size=batch_size,shuffle=True)
     val_loader = DataLoader(val_dataset,batch_size=batch_size,shuffle=False)
     test_loader = DataLoader(test_dataset,batch_size=batch_size,shuffle=False)
```

Model1:

MLP model: Approach: a simple model to be used for classification



```
class Model1(nn.Module):
    def __init__(self):
        super(Model1, self).__init__()

        # Define layers
        self.layer1 = nn.Linear(128, 20)
        self.activation = nn.ReLU()
        self.layer2 = nn.Linear(20, 10)
        # self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.layer1(x)
        x = self.activation(x)
        x = self.layer2(x)
        # x = self.sigmoid(x)
        return x


model = Model1()
model.to(device)
```

```
Model1(
  (layer1): Linear(in_features=128, out_features=20, bias=True)
  (activation): ReLU()
  (layer2): Linear(in_features=20, out_features=10, bias=True)
)
```

The model is a simple feedforward neural network (also known as a Multi-Layer Perceptron or MLP) with two linear layers and a ReLU activation function between them. Here's how you can adapt it for multi-label text classification:

Input size: 128 (embeddings generated above)
Output size: 10 (number of classes)
Activation function: ReLU (introduces non-linearity between the layers)

Loss function: **BCEWithLogitsLoss** which combines the sigmoid activation and binary cross-entropy loss in a numerically stable way. This loss function is commonly used for multi-label classification problems.

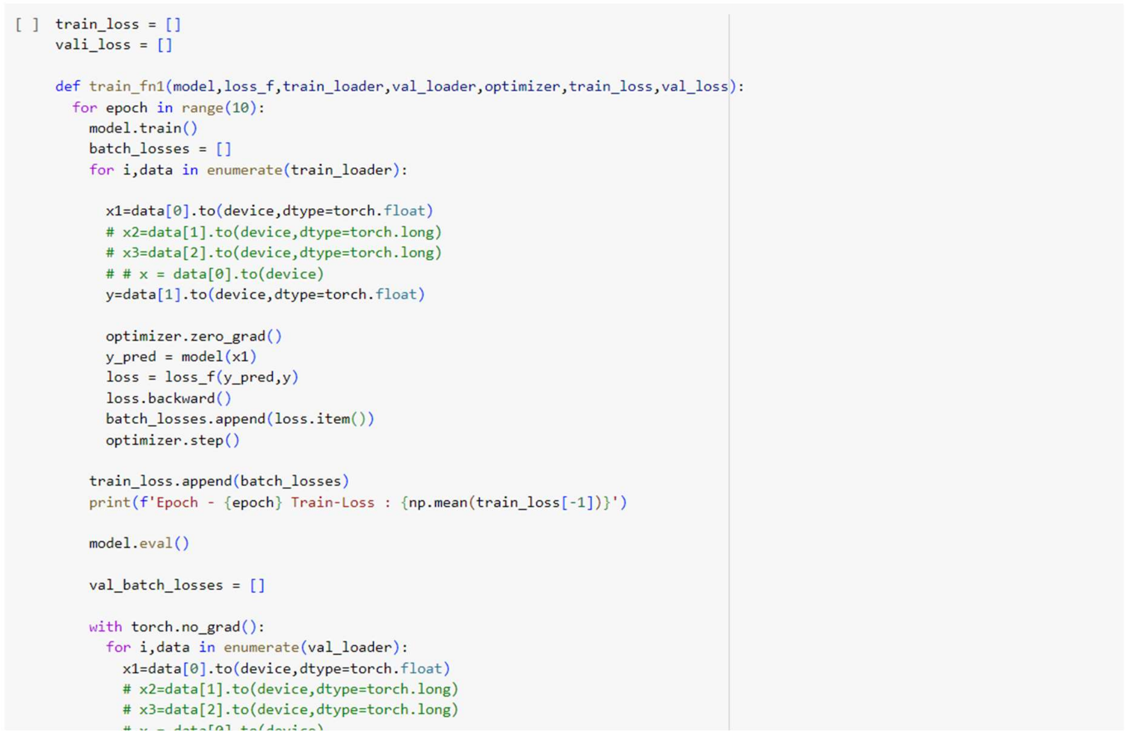
Why it is used?

- It can be applied to complex non-linear problems.
- It works well with large input data.
- The same accuracy can be achieved even with smaller data.

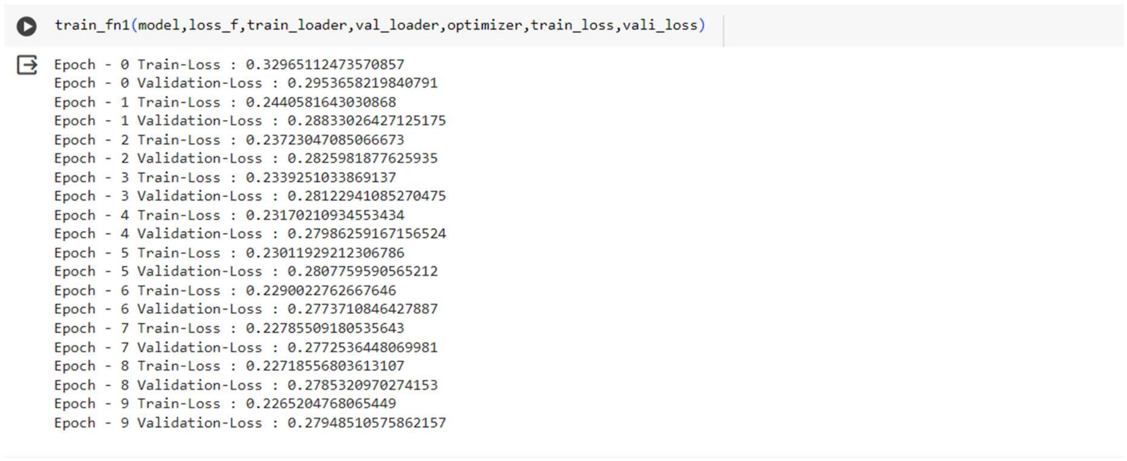
Cons of using this model:

- May struggle with capturing complex relationships in sequential data.

Training:

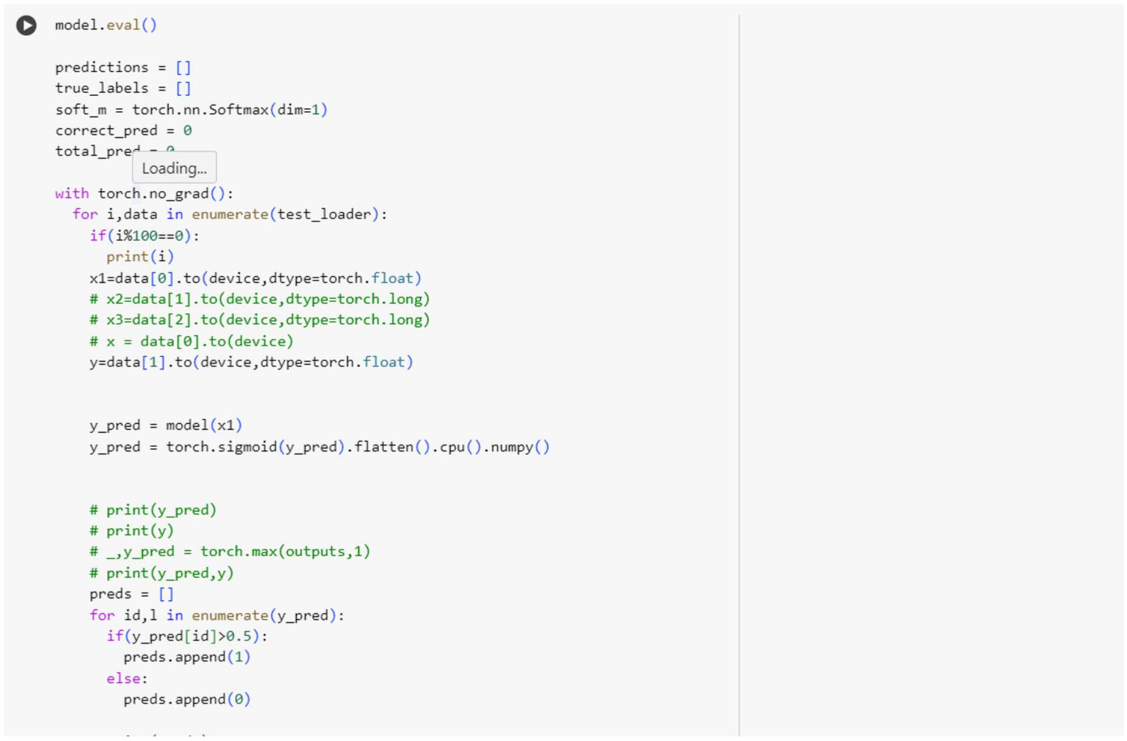


```
[ ]  train_loss = []
     vali_loss = []

     def train_fn1(model,loss_f,train_loader,val_loader,optimizer,train_loss,val_loss):
       for epoch in range(10):
         model.train()
         batch_losses = []
         for i,data in enumerate(train_loader):

           x1=data[0].to(device,dtype=torch.float)
           # x2=data[1].to(device,dtype=torch.long)
           # x3=data[2].to(device,dtype=torch.long)
           # # x = data[0].to(device)
           y=data[1].to(device,dtype=torch.float)

           optimizer.zero_grad()
           y_pred = model(x1)
           loss = loss_f(y_pred,y)
           loss.backward()
           batch_losses.append(loss.item())
           optimizer.step()

         train_loss.append(batch_losses)
         print(f'Epoch - {epoch} Train-Loss : {np.mean(train_loss[-1])}')

         model.eval()

         val_batch_losses = []

         with torch.no_grad():
           for i,data in enumerate(val_loader):
             x1=data[0].to(device,dtype=torch.float)
             # x2=data[1].to(device,dtype=torch.long)
             # x3=data[2].to(device,dtype=torch.long)
```



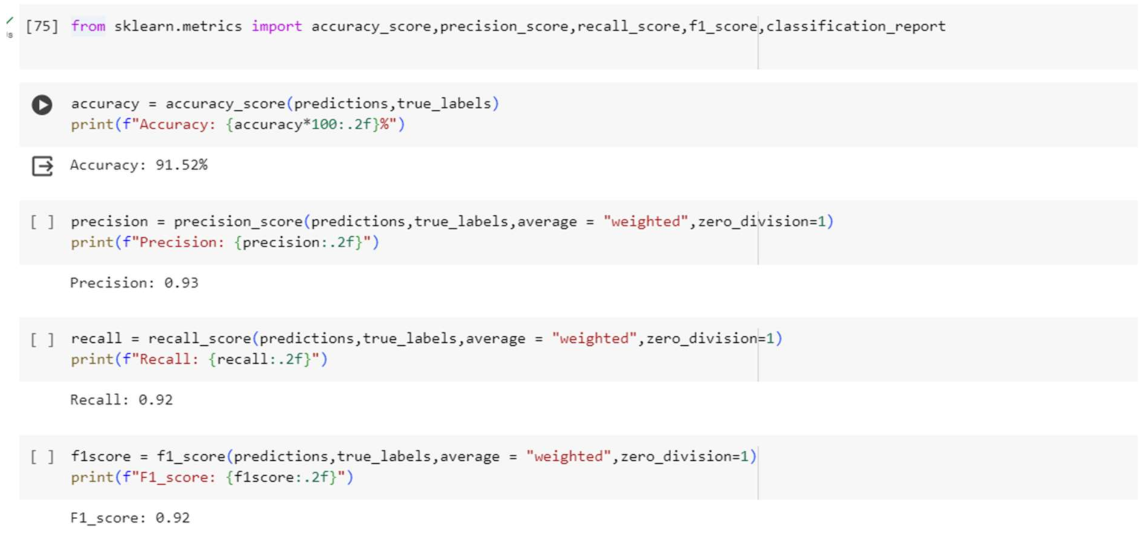
```
train_fn1(model,loss_f,train_loader,val_loader,optimizer,train_loss,vali_loss)

Epoch - 0 Train-Loss : 0.32965112473570857
Epoch - 0 Validation-Loss : 0.2953658219840791
Epoch - 1 Train-Loss : 0.2440581643030868
Epoch - 1 Validation-Loss : 0.28833026427125175
Epoch - 2 Train-Loss : 0.23723047085066673
Epoch - 2 Validation-Loss : 0.2825981877625935
Epoch - 3 Train-Loss : 0.2339251033869137
Epoch - 3 Validation-Loss : 0.28122941085270475
Epoch - 4 Train-Loss : 0.23170210934553434
Epoch - 4 Validation-Loss : 0.27986259167156524
Epoch - 5 Train-Loss : 0.23011929212306786
Epoch - 5 Validation-Loss : 0.2807759590565212
Epoch - 6 Train-Loss : 0.2290022762667646
Epoch - 6 Validation-Loss : 0.2773710846427887
Epoch - 7 Train-Loss : 0.22785509180535643
Epoch - 7 Validation-Loss : 0.2772536448069981
Epoch - 8 Train-Loss : 0.22718556803613107
Epoch - 8 Validation-Loss : 0.2785320970274153
Epoch - 9 Train-Loss : 0.2265204768065449
Epoch - 9 Validation-Loss : 0.27948510575862157
```

Testing:



```
model.eval()

predictions = []
true_labels = []
soft_m = torch.nn.Softmax(dim=1)
correct_pred = 0
total_pred = 0

with torch.no_grad():
  for i,data in enumerate(test_loader):
    if(i%100==0):
      print(i)
    x1=data[0].to(device,dtype=torch.float)
    # x2=data[1].to(device,dtype=torch.long)
    # x3=data[2].to(device,dtype=torch.long)
    # x = data[0].to(device)
    y=data[1].to(device,dtype=torch.float)


    y_pred = model(x1)
    y_pred = torch.sigmoid(y_pred).flatten().cpu().numpy()


    # print(y_pred)
    # print(y)
    # _,y_pred = torch.max(outputs,1)
    # print(y_pred,y)
    preds = []
    for id,l in enumerate(y_pred):
      if(y_pred[id]>0.5):
        preds.append(1)
      else:
        preds.append(0)
```

Evaluation:



```
[75] from sklearn.metrics import accuracy_score,precision_score,recall_score,f1_score,classification_report
```

```
accuracy = accuracy_score(predictions,true_labels)
print(f"Accuracy: {accuracy*100:.2f}%")
```

```
Accuracy: 91.52%
```

```
[ ] precision = precision_score(predictions,true_labels,average = "weighted",zero_division=1)
    print(f"Precision: {precision:.2f}")
```

```
Precision: 0.93
```

```
[ ] recall = recall_score(predictions,true_labels,average = "weighted",zero_division=1)
    print(f"Recall: {recall:.2f}")
```

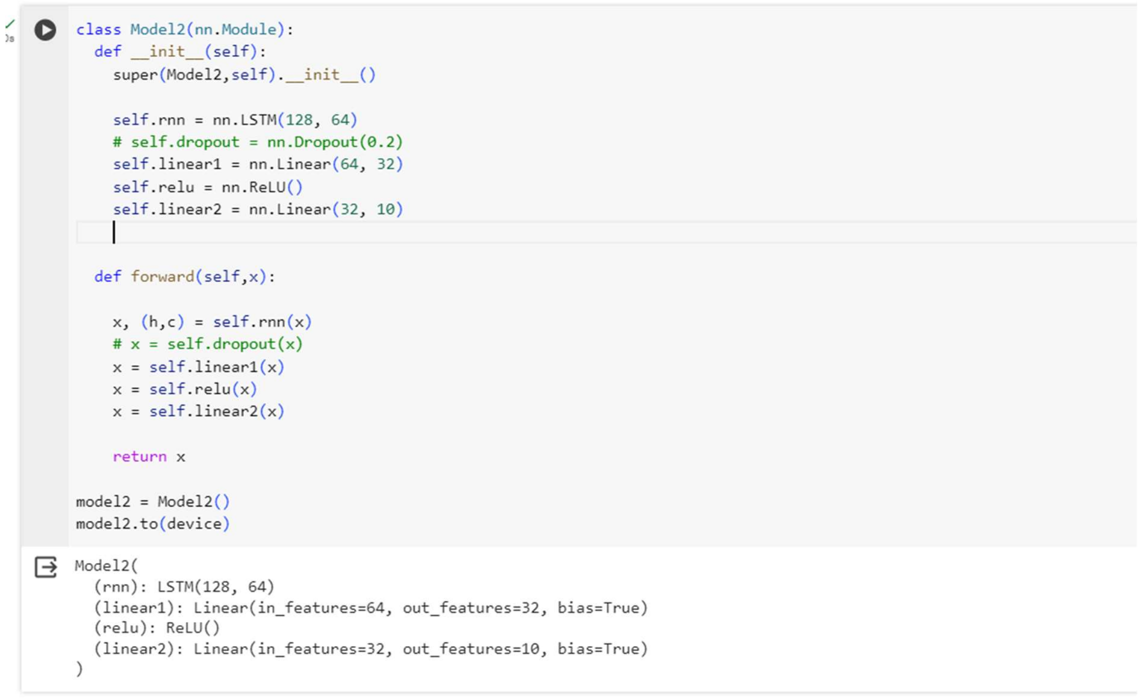
```
Recall: 0.92
```

```
[ ] f1score = f1_score(predictions,true_labels,average = "weighted",zero_division=1)
    print(f"F1_score: {f1score:.2f}")
```

```
F1_score: 0.92
```

Model2:

Using LSTM model: Approach: a model that uses lstm layer for classification



```
class Model2(nn.Module):
  def __init__(self):
    super(Model2,self).__init__()

    self.rnn = nn.LSTM(128, 64)
    # self.dropout = nn.Dropout(0.2)
    self.linear1 = nn.Linear(64, 32)
    self.relu = nn.ReLU()
    self.linear2 = nn.Linear(32, 10)


  def forward(self,x):

    x, (h,c) = self.rnn(x)
    # x = self.dropout(x)
    x = self.linear1(x)
    x = self.relu(x)
    x = self.linear2(x)

    return x

model2 = Model2()
model2.to(device)
```

```
Model2(
  (rnn): LSTM(128, 64)
  (linear1): Linear(in_features=64, out_features=32, bias=True)
  (relu): ReLU()
  (linear2): Linear(in_features=32, out_features=10, bias=True)
)
```

The model architecture involves an LSTM layer followed by linear layers with ReLU activations, which can be suitable for processing sequential data like text and making predictions for multiple labels.

Input size: 128 (embeddings generated above)
Hidden layer = (input_size – 64 , output_size – 32)
Output size: 10 (number of classes)
Activation function: ReLU (introduces non-linearity between the layers)

Loss function: **BCEWithLogitsLoss** which combines the sigmoid activation and binary cross-entropy loss in a numerically stable way. This loss function is commonly used for multi-label classification problems.

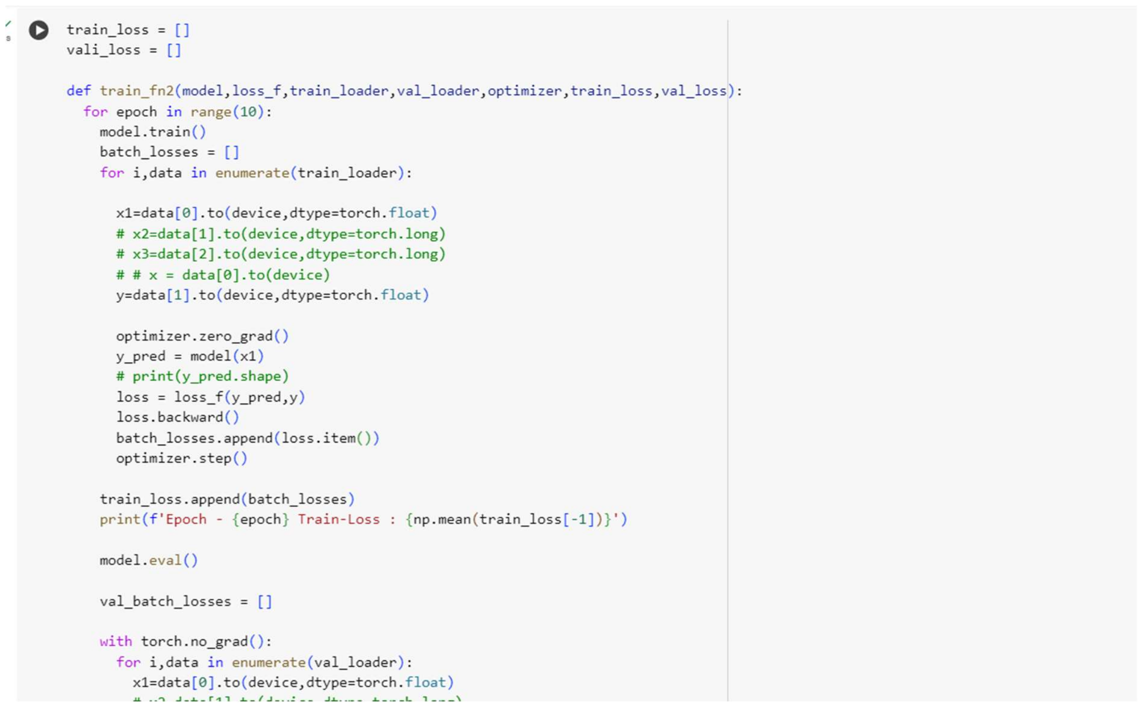
Why it is used?

- It can be applied to create an additional module in a neural network that learns when to remember and when to forget pertinent information.
- It is capable of capturing the patterns of both long-term seasonalities such as a yearly pattern and short-term seasonalities such as weekly patterns.

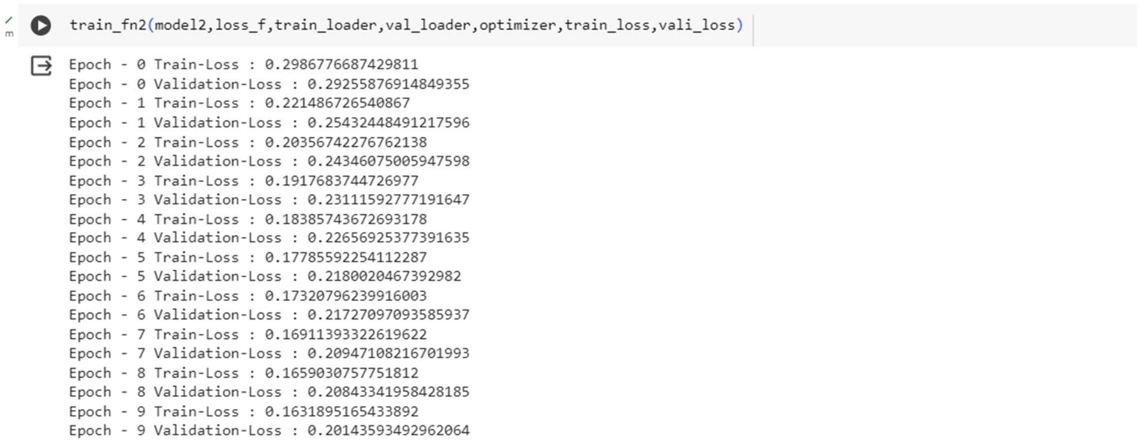
Cons of using this model:

- Can be computationally expensive, especially with large datasets.
- Requires careful tuning of hyperparameters.

Training:



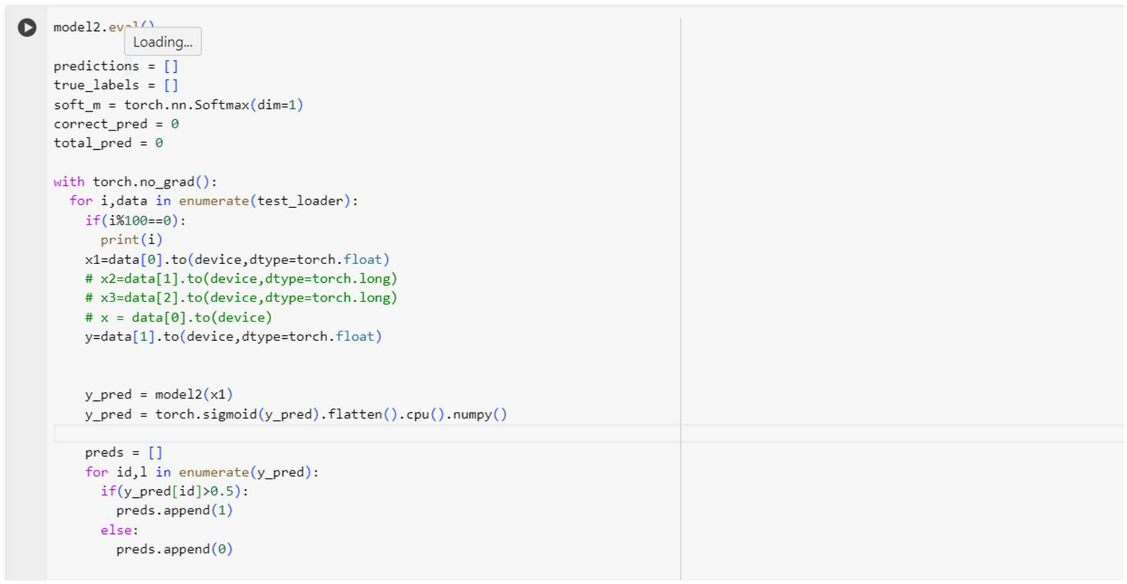
```python
train_loss = []
vali_loss = []

def train_fn2(model,loss_f,train_loader,val_loader,optimizer,train_loss,val_loss):
  for epoch in range(10):
    model.train()
    batch_losses = []
    for i,data in enumerate(train_loader):

      x1=data[0].to(device,dtype=torch.float)
      # x2=data[1].to(device,dtype=torch.long)
      # x3=data[2].to(device,dtype=torch.long)
      # # x = data[0].to(device)
      y=data[1].to(device,dtype=torch.float)

      optimizer.zero_grad()
      y_pred = model(x1)
      # print(y_pred.shape)
      loss = loss_f(y_pred,y)
      loss.backward()
      batch_losses.append(loss.item())
      optimizer.step()

    train_loss.append(batch_losses)
    print(f'Epoch - {epoch} Train-Loss : {np.mean(train_loss[-1])}')

    model.eval()

    val_batch_losses = []

    with torch.no_grad():
      for i,data in enumerate(val_loader):
        x1=data[0].to(device,dtype=torch.float)
```



```python
train_fn2(model2,loss_f,train_loader,val_loader,optimizer,train_loss,vali_loss)
```

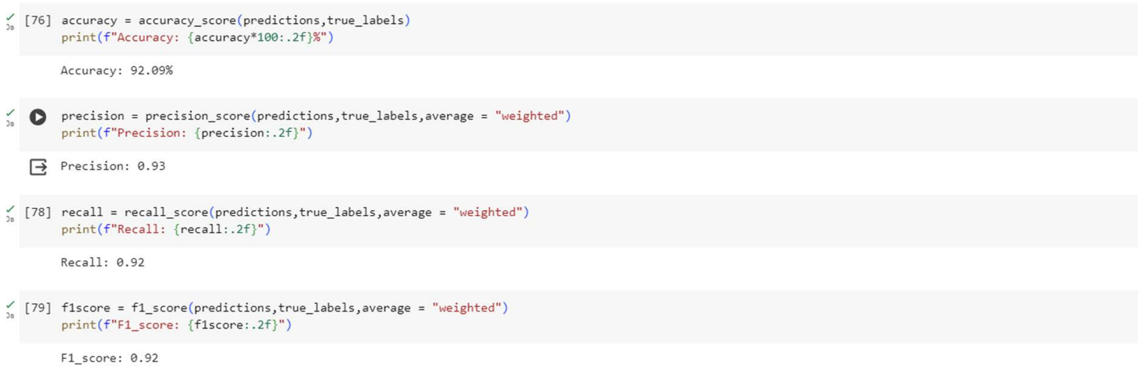
```
Epoch - 0 Train-Loss : 0.2986776687429811
Epoch - 0 Validation-Loss : 0.29255876914849355
Epoch - 1 Train-Loss : 0.221486726540867
Epoch - 1 Validation-Loss : 0.25432448491217596
Epoch - 2 Train-Loss : 0.20356742276762138
Epoch - 2 Validation-Loss : 0.24346075005947598
Epoch - 3 Train-Loss : 0.1917683744726977
Epoch - 3 Validation-Loss : 0.23111592777191647
Epoch - 4 Train-Loss : 0.18385743672693178
Epoch - 4 Validation-Loss : 0.22656925377391635
Epoch - 5 Train-Loss : 0.17785592254112287
Epoch - 5 Validation-Loss : 0.2180020467392982
Epoch - 6 Train-Loss : 0.17320796239916003
Epoch - 6 Validation-Loss : 0.21727097093585937
Epoch - 7 Train-Loss : 0.16911393322619622
Epoch - 7 Validation-Loss : 0.20947108216701993
Epoch - 8 Train-Loss : 0.1659030757751812
Epoch - 8 Validation-Loss : 0.20843341958428185
Epoch - 9 Train-Loss : 0.1631895165433892
Epoch - 9 Validation-Loss : 0.20143593492962064
```

Testing:



```
model2.eval()

predictions = []
true_labels = []
soft_m = torch.nn.Softmax(dim=1)
correct_pred = 0
total_pred = 0

with torch.no_grad():
  for i,data in enumerate(test_loader):
    if(i%100==0):
      print(i)
    x1=data[0].to(device,dtype=torch.float)
    # x2=data[1].to(device,dtype=torch.long)
    # x3=data[2].to(device,dtype=torch.long)
    # x = data[0].to(device)
    y=data[1].to(device,dtype=torch.float)


    y_pred = model2(x1)
    y_pred = torch.sigmoid(y_pred).flatten().cpu().numpy()

    preds = []
    for id,l in enumerate(y_pred):
      if(y_pred[id]>0.5):
        preds.append(1)
      else:
        preds.append(0)
```

Evaluation:

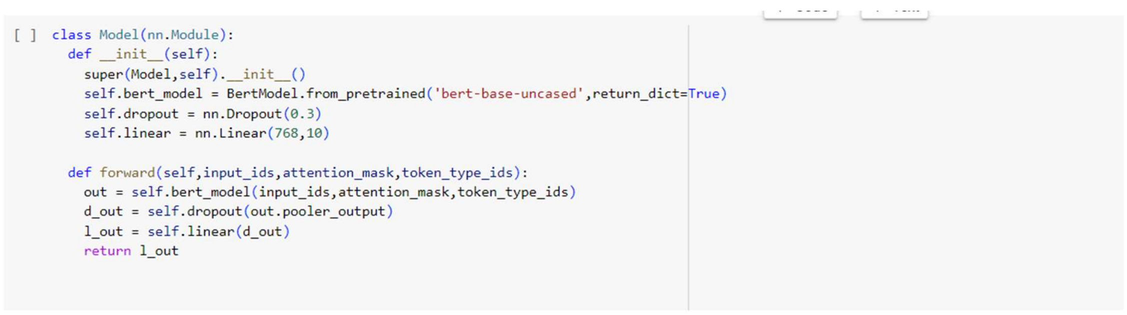


```
[76] accuracy = accuracy_score(predictions,true_labels)
     print(f"Accuracy: {accuracy*100:.2f}%")

     Accuracy: 92.09%

     precision = precision_score(predictions,true_labels,average = "weighted")
     print(f"Precision: {precision:.2f}")

     Precision: 0.93

[78] recall = recall_score(predictions,true_labels,average = "weighted")
     print(f"Recall: {recall:.2f}")

     Recall: 0.92

[79] f1score = f1_score(predictions,true_labels,average = "weighted")
     print(f"F1_score: {f1score:.2f}")

     F1_score: 0.92
```

Model3:

BERT model: Approach: Transformer Based approach due to:

1. Pre-trained Representations: BERT is pre-trained on large amounts of diverse text data, allowing it to capture rich contextual information and semantic relationships between words.
2. Contextual Understanding: BERT-based models inherently have a better understanding of the context in which words appear in sentences. This can be crucial for tasks that require grasping nuanced and contextual meanings.
3. Less Sensitivity to Sequence Length: BERT can handle variable-length sequences more effectively than traditional RNNs or LSTMs, making it more robust to different lengths of input text.



```
[ ] class Model(nn.Module):
      def __init__(self):
        super(Model,self).__init__()
        self.bert_model = BertModel.from_pretrained('bert-base-uncased',return_dict=True)
        self.dropout = nn.Dropout(0.3)
        self.linear = nn.Linear(768,10)

      def forward(self,input_ids,attention_mask,token_type_ids):
        out = self.bert_model(input_ids,attention_mask,token_type_ids)
        d_out = self.dropout(out.pooler_output)
        l_out = self.linear(d_out)
        return l_out
```

Embedding generates of dimension = 768
Output size: 10 (number of classes)
Loss function: **BCEWithLogitsLoss** which combines the sigmoid activation and binary cross-entropy loss in a numerically stable way. This loss function is commonly used for multi-label classification problems.

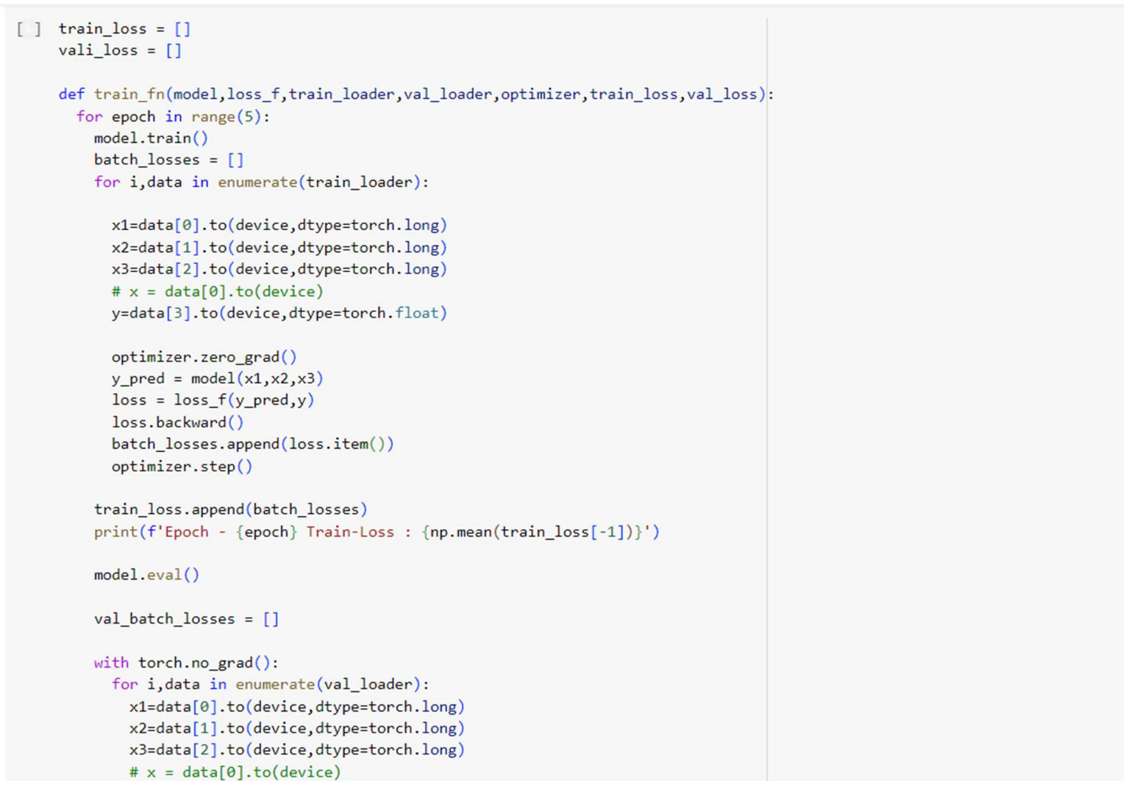
Why it is used?

- It can be applied to train machine learning models on textual data.
- BERT is specifically designed for NLP tasks that require a deep understanding of the context of words.

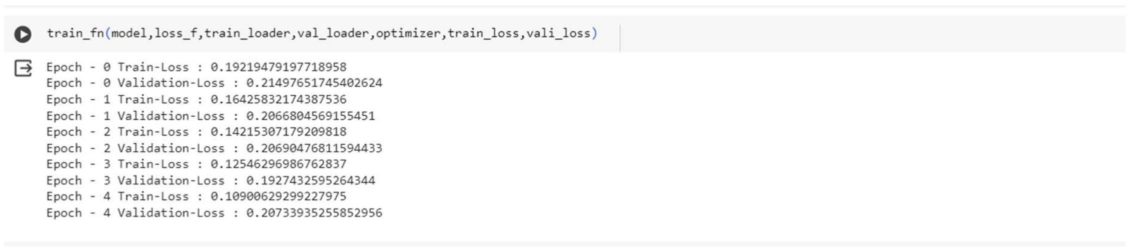
Cons of using this model:

- Can be computationally expensive and memory-intensive.
- Requires a large amount of training data for fine-tuning.

Training:



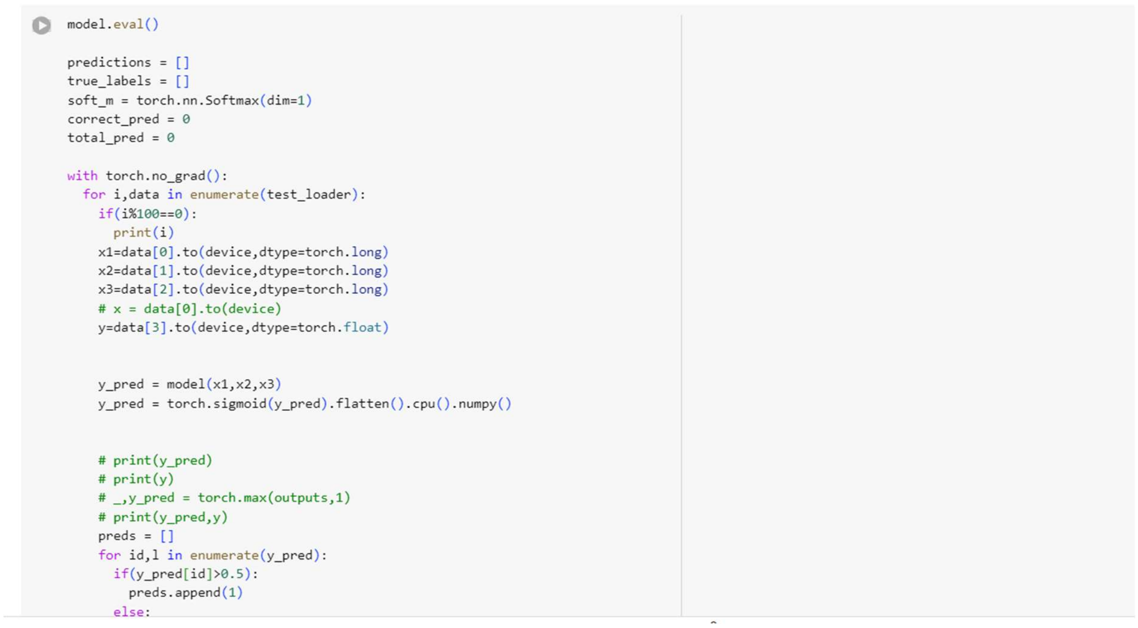
```
[ ] train_loss = []
    vali_loss = []

    def train_fn(model,loss_f,train_loader,val_loader,optimizer,train_loss,val_loss):
      for epoch in range(5):
        model.train()
        batch_losses = []
        for i,data in enumerate(train_loader):

          x1=data[0].to(device,dtype=torch.long)
          x2=data[1].to(device,dtype=torch.long)
          x3=data[2].to(device,dtype=torch.long)
          # x = data[0].to(device)
          y=data[3].to(device,dtype=torch.float)

          optimizer.zero_grad()
          y_pred = model(x1,x2,x3)
          loss = loss_f(y_pred,y)
          loss.backward()
          batch_losses.append(loss.item())
          optimizer.step()

        train_loss.append(batch_losses)
        print(f'Epoch - {epoch} Train-Loss : {np.mean(train_loss[-1])}')

        model.eval()

        val_batch_losses = []

        with torch.no_grad():
          for i,data in enumerate(val_loader):
            x1=data[0].to(device,dtype=torch.long)
            x2=data[1].to(device,dtype=torch.long)
            x3=data[2].to(device,dtype=torch.long)
            # x = data[0].to(device)
```



```
train_fn(model,loss_f,train_loader,val_loader,optimizer,train_loss,vali_loss)
```

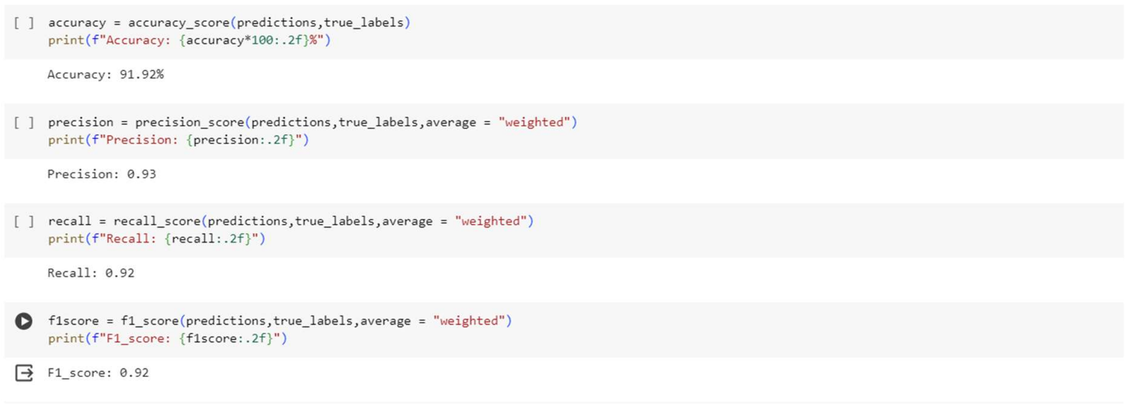
```
Epoch - 0 Train-Loss : 0.19219479197718958
Epoch - 0 Validation-Loss : 0.21497651745402624
Epoch - 1 Train-Loss : 0.16425832174387536
Epoch - 1 Validation-Loss : 0.2066804569155451
Epoch - 2 Train-Loss : 0.14215307179209818
Epoch - 2 Validation-Loss : 0.20690476811594433
Epoch - 3 Train-Loss : 0.12546296986762837
Epoch - 3 Validation-Loss : 0.1927432595264344
Epoch - 4 Train-Loss : 0.10900629299227975
Epoch - 4 Validation-Loss : 0.20733935255852956
```

Testing:



```
model.eval()

predictions = []
true_labels = []
soft_m = torch.nn.Softmax(dim=1)
correct_pred = 0
total_pred = 0

with torch.no_grad():
  for i,data in enumerate(test_loader):
    if(i%100==0):
      print(i)
    x1=data[0].to(device,dtype=torch.long)
    x2=data[1].to(device,dtype=torch.long)
    x3=data[2].to(device,dtype=torch.long)
    # x = data[0].to(device)
    y=data[3].to(device,dtype=torch.float)


    y_pred = model(x1,x2,x3)
    y_pred = torch.sigmoid(y_pred).flatten().cpu().numpy()


    # print(y_pred)
    # print(y)
    # _,y_pred = torch.max(outputs,1)
    # print(y_pred,y)
    preds = []
    for id,l in enumerate(y_pred):
      if(y_pred[id]>0.5):
        preds.append(1)
      else:
```

Evaluation:



```
accuracy = accuracy_score(predictions,true_labels)
print(f"Accuracy: {accuracy*100:.2f}%")
```

```
Accuracy: 91.92%
```

```
precision = precision_score(predictions,true_labels,average = "weighted")
print(f"Precision: {precision:.2f}")
```

```
Precision: 0.93
```

```
recall = recall_score(predictions,true_labels,average = "weighted")
print(f"Recall: {recall:.2f}")
```

```
Recall: 0.92
```

```
f1score = f1_score(predictions,true_labels,average = "weighted")
print(f"F1_score: {f1score:.2f}")
```

```
F1_score: 0.92
```

As conclusion all the models have performed well. The choice of the best model for multilabel text classification depends on various factors such as the size of your dataset, the complexity of your text data, and the computational resources available.