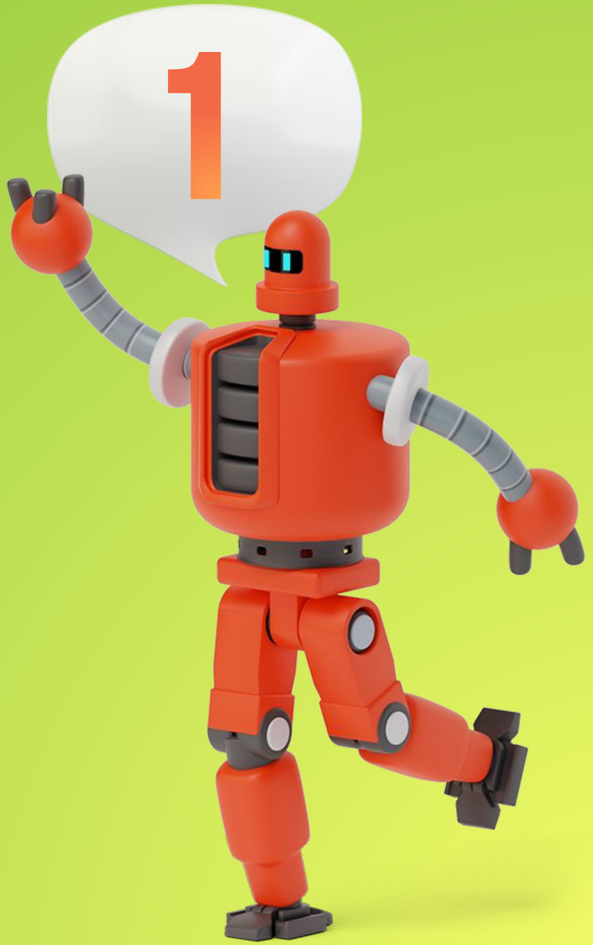
TOXIC COMMENT CLASSIFICATION

CS 482 FINAL PROJECT
VAISNAVI NEMALA

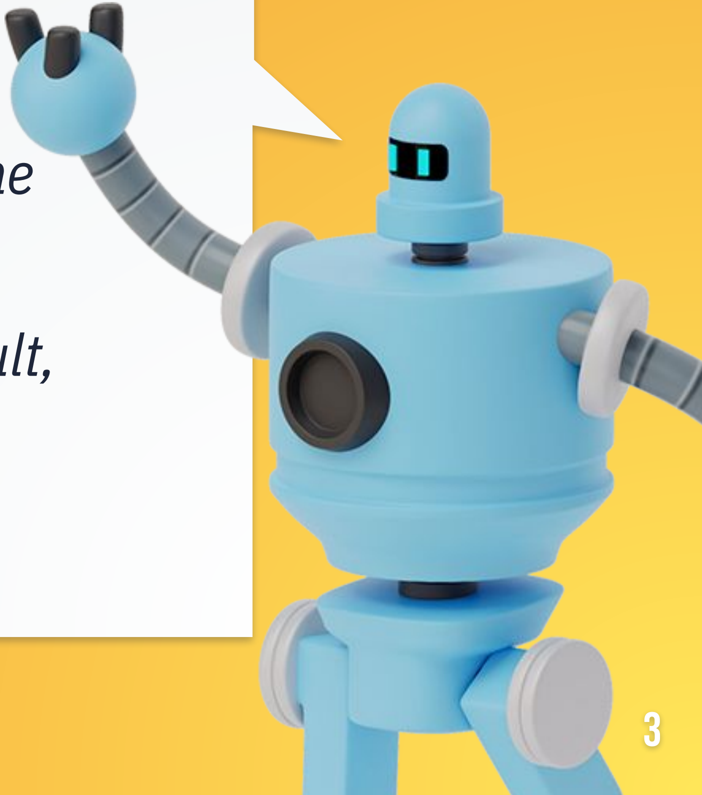
INTRODUCTION

Discussion of Challenge



The goal of the Toxic Comment Classification project is to identify and classify online comments according to the following classes of toxicity:

[toxic, severe_toxic, obscene, threat, insult, identity_hate]

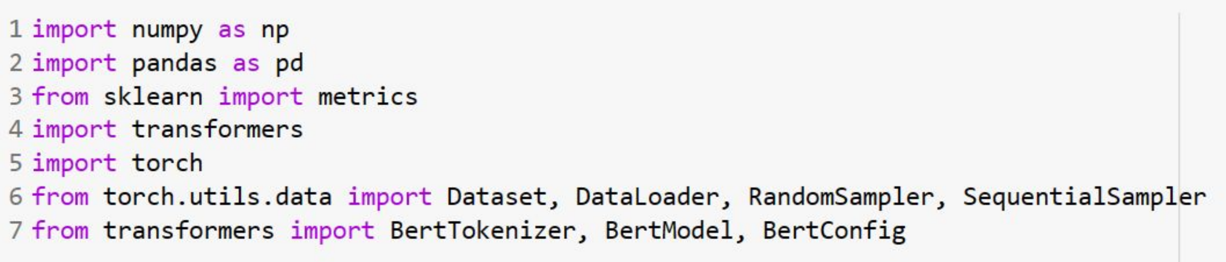


CODE WALKTHROUGH

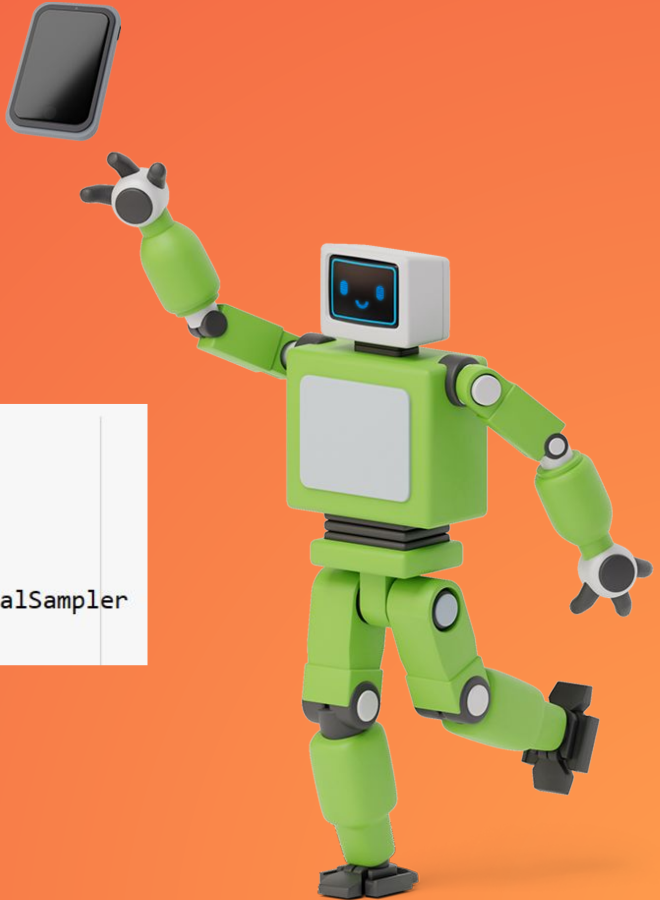
Code & App Documentation

1. PREPARING THE ENVIRONMENT

- Install necessary libraries
- Import libraries

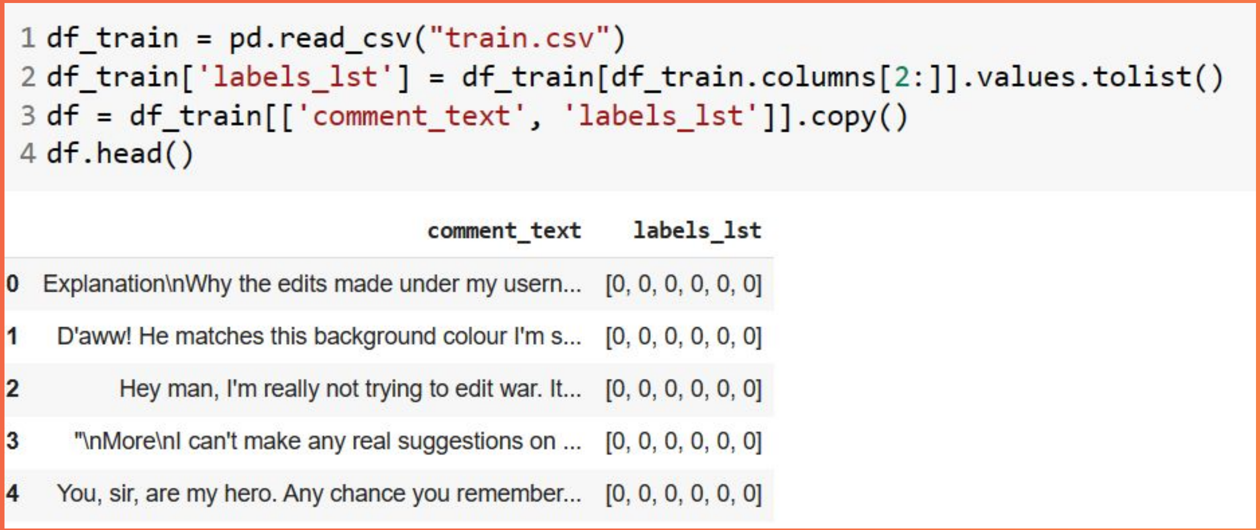


```python
import numpy as np
import pandas as pd
from sklearn import metrics
import transformers
import torch
from torch.utils.data import Dataset, DataLoader, RandomSampler, SequentialSampler
from transformers import BertTokenizer, BertModel, BertConfig
```



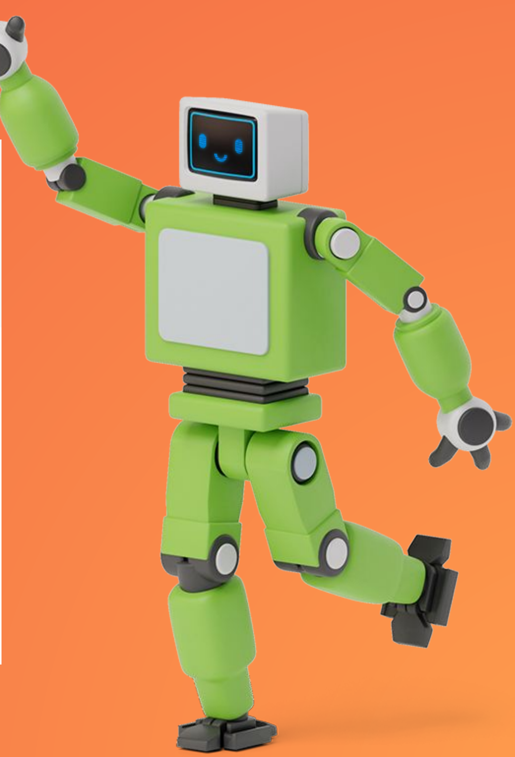
2. DATA LOADING & PREPROCESSING

- Load in csv files



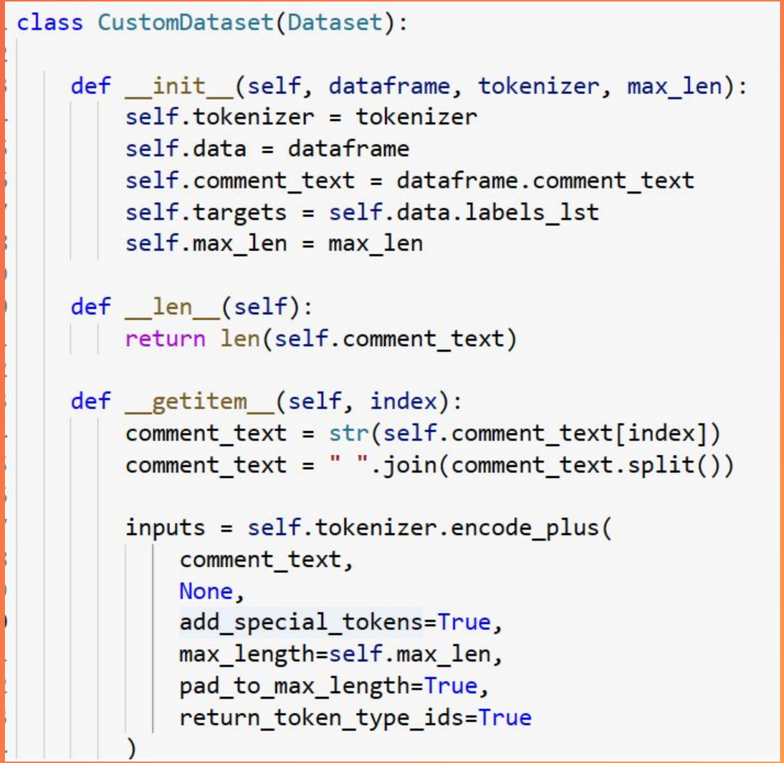
```python
df_train = pd.read_csv("train.csv")
df_train['labels_lst'] = df_train[df_train.columns[2:]].values.tolist()
df = df_train[['comment_text', 'labels_lst']].copy()
df.head()
```

	comment_text	labels_lst
0	Explanation\nWhy the edits made under my usern...	[0, 0, 0, 0, 0, 0]
1	D'aww! He matches this background colour I'm s...	[0, 0, 0, 0, 0, 0]
2	Hey man, I'm really not trying to edit war. It...	[0, 0, 0, 0, 0, 0]
3	"\nMore\nI can't make any real suggestions on ...	[0, 0, 0, 0, 0, 0]
4	You, sir, are my hero. Any chance you remember...	[0, 0, 0, 0, 0, 0]



2. DATA LOADING & PREPROCESSING

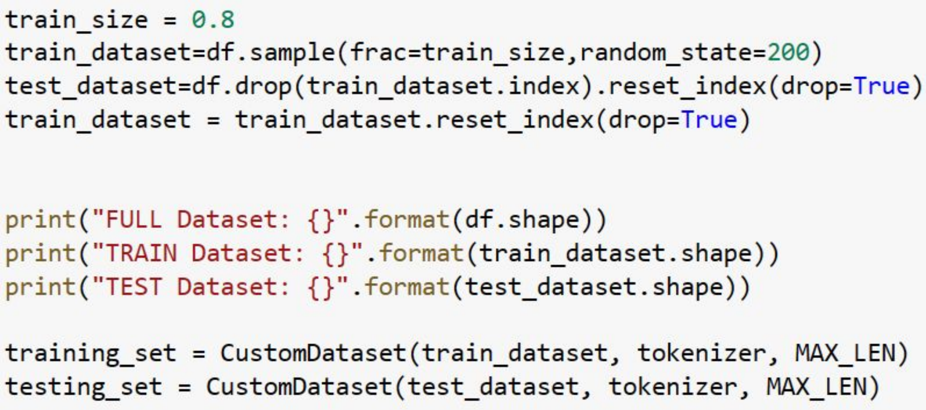
- Create custom dataset class



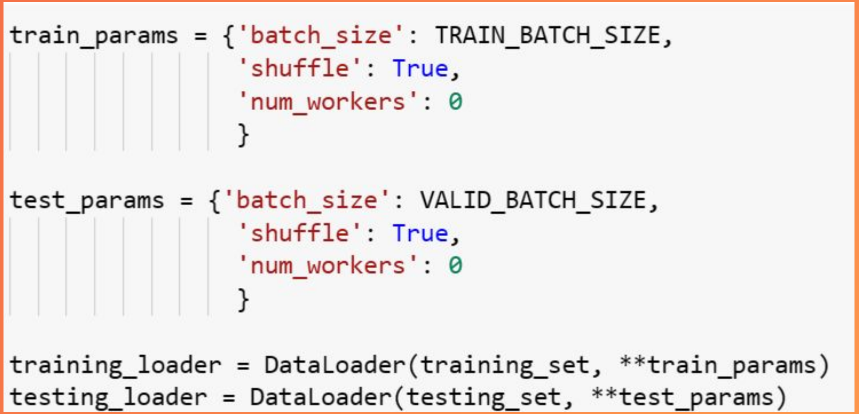
```
class CustomDataset(Dataset):

    def __init__(self, dataframe, tokenizer, max_len):
        self.tokenizer = tokenizer
        self.data = dataframe
        self.comment_text = dataframe.comment_text
        self.targets = self.data.labels_lst
        self.max_len = max_len

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

    def __getitem__(self, index):
        comment_text = str(self.comment_text[index])
        comment_text = " ".join(comment_text.split())

        inputs = self.tokenizer.encode_plus(
            comment_text,
            None,
            add_special_tokens=True,
            max_length=self.max_len,
            pad_to_max_length=True,
            return_token_type_ids=True
        )
```

2. DATA LOADING & PREPROCESSING

- Create datasets for dataloaders



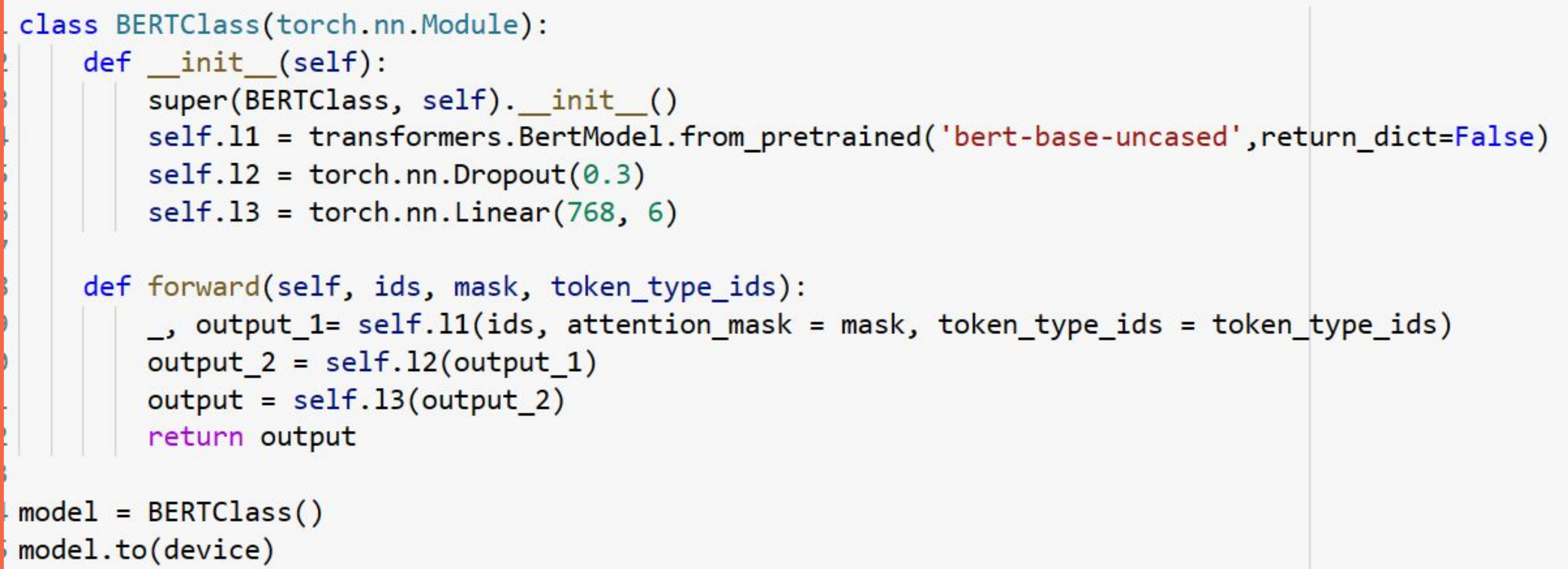
```
train_size = 0.8
train_dataset=df.sample(frac=train_size,random_state=200)
test_dataset=df.drop(train_dataset.index).reset_index(drop=True)
train_dataset = train_dataset.reset_index(drop=True)


print("FULL Dataset: {}".format(df.shape))
print("TRAIN Dataset: {}".format(train_dataset.shape))
print("TEST Dataset: {}".format(test_dataset.shape))

training_set = CustomDataset(train_dataset, tokenizer, MAX_LEN)
testing_set = CustomDataset(test_dataset, tokenizer, MAX_LEN)
```



```
train_params = {'batch_size': TRAIN_BATCH_SIZE,
                'shuffle': True,
                'num_workers': 0
                }

test_params = {'batch_size': VALID_BATCH_SIZE,
                'shuffle': True,
                'num_workers': 0
                }

training_loader = DataLoader(training_set, **train_params)
testing_loader = DataLoader(testing_set, **test_params)
```

3. CREATE MODEL

- Create custom BERT class from ‘bert-base-uncased’ pretrained model



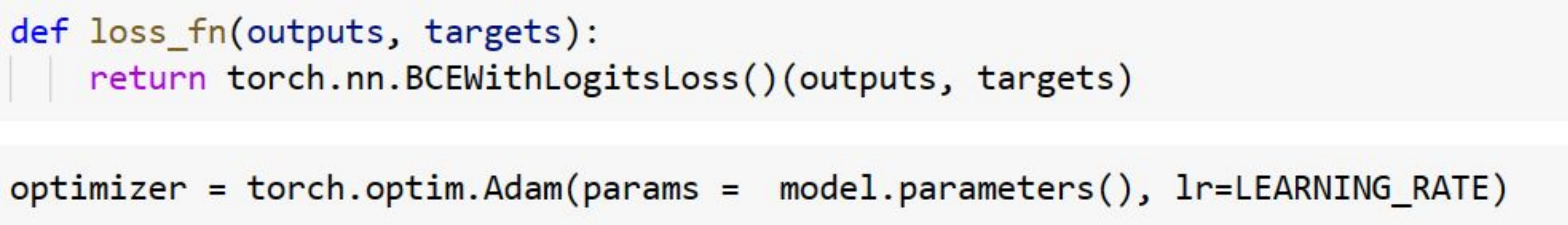
```python
class BERTClass(torch.nn.Module):
    def __init__(self):
        super(BERTClass, self).__init__()
        self.l1 = transformers.BertModel.from_pretrained('bert-base-uncased',return_dict=False)
        self.l2 = torch.nn.Dropout(0.3)
        self.l3 = torch.nn.Linear(768, 6)

    def forward(self, ids, mask, token_type_ids):
        _, output_1= self.l1(ids, attention_mask = mask, token_type_ids = token_type_ids)
        output_2 = self.l2(output_1)
        output = self.l3(output_2)
        return output

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

3. CREATE MODEL

- Model is composed of:
  - (1) pretrained ‘bert-base-uncased’ model
  - (2) dropout layer (0.3)
  - (3) output (6 neurons - class probabilities)

3. CREATE MODEL

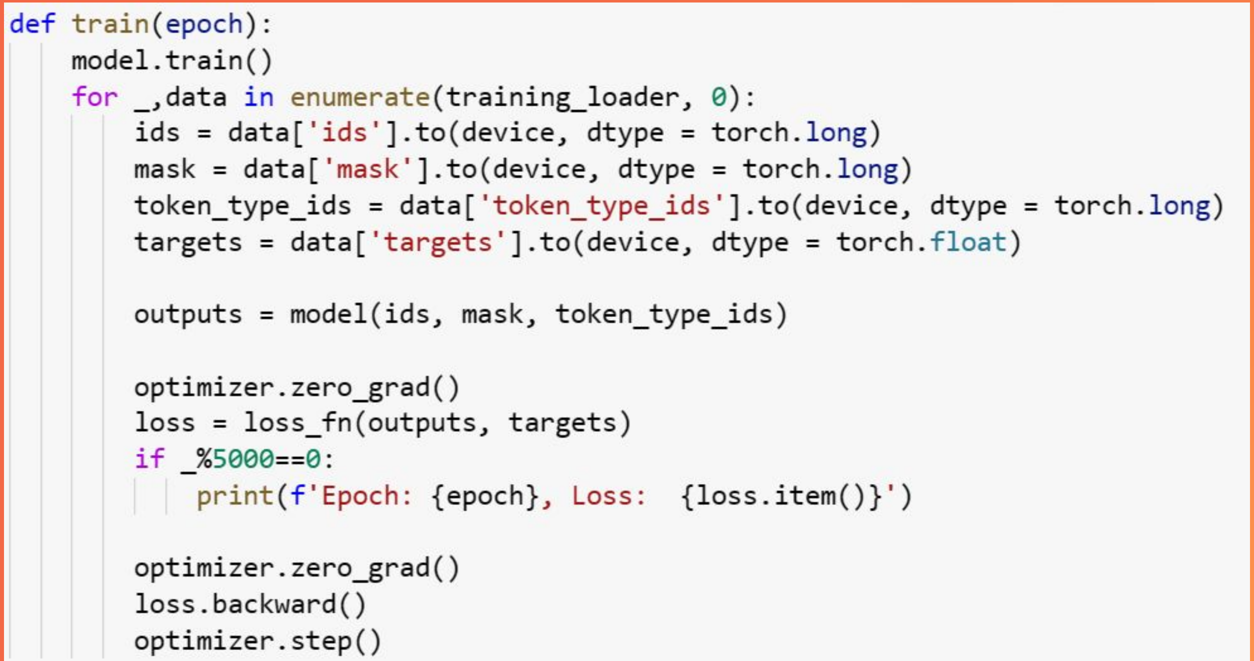
- Define loss and optimization function



```
def loss_fn(outputs, targets):
    return torch.nn.BCEWithLogitsLoss()(outputs, targets)

optimizer = torch.optim.Adam(params =  model.parameters(), lr=LEARNING_RATE)
```

4. TRAINING THE MODEL

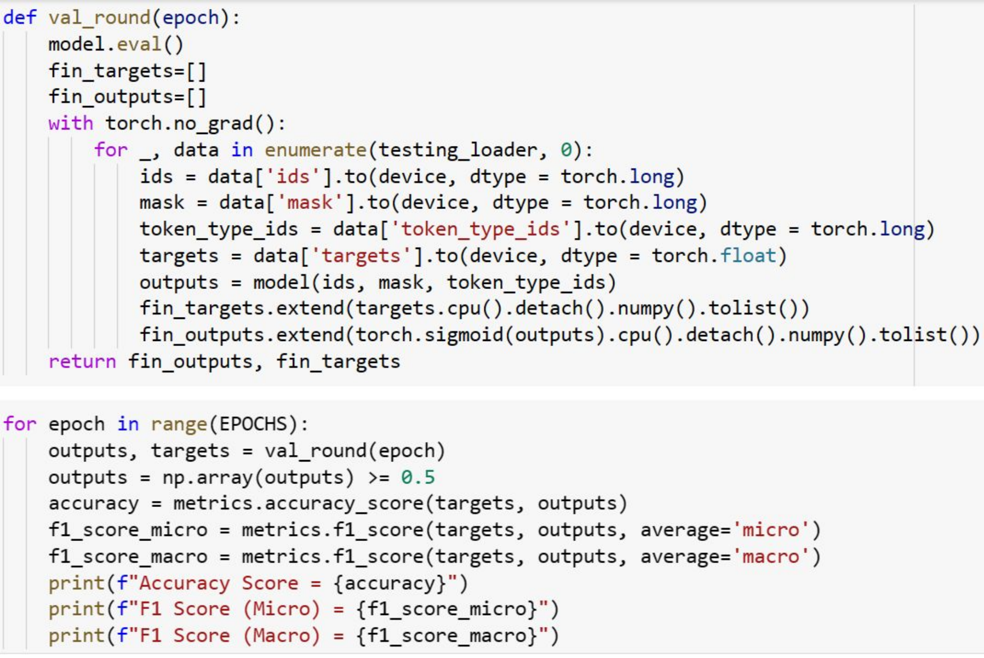
- Train the model on the training dataset



```python
def train(epoch):
    model.train()
    for _,data in enumerate(training_loader, 0):
        ids = data['ids'].to(device, dtype = torch.long)
        mask = data['mask'].to(device, dtype = torch.long)
        token_type_ids = data['token_type_ids'].to(device, dtype = torch.long)
        targets = data['targets'].to(device, dtype = torch.float)

        outputs = model(ids, mask, token_type_ids)

        optimizer.zero_grad()
        loss = loss_fn(outputs, targets)
        if _%5000==0:
            print(f'Epoch: {epoch}, Loss:  {loss.item()}')

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

5. EVALUATING THE MODEL

- Determine the accuracy of the model with a validation dataset that the model has never seen before.

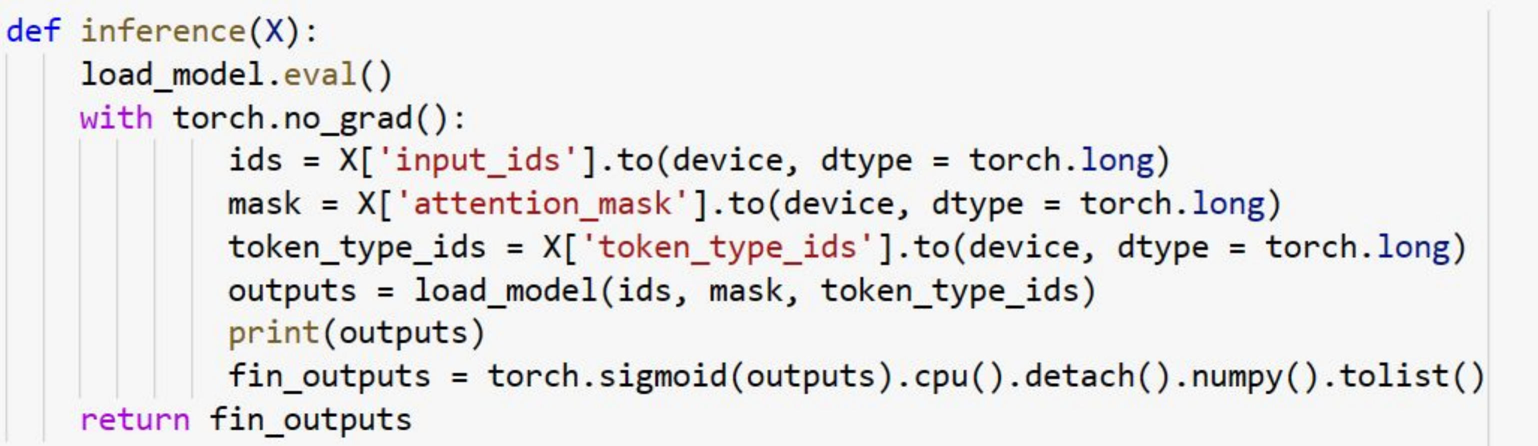


```python
def val_round(epoch):
    model.eval()
    fin_targets=[]
    fin_outputs=[]
    with torch.no_grad():
        for _, data in enumerate(testing_loader, 0):
            ids = data['ids'].to(device, dtype = torch.long)
            mask = data['mask'].to(device, dtype = torch.long)
            token_type_ids = data['token_type_ids'].to(device, dtype = torch.long)
            targets = data['targets'].to(device, dtype = torch.float)
            outputs = model(ids, mask, token_type_ids)
            fin_targets.extend(targets.cpu().detach().numpy().tolist())
            fin_outputs.extend(torch.sigmoid(outputs).cpu().detach().numpy().tolist())
    return fin_outputs, fin_targets
```

```python
for epoch in range(EPOCHS):
    outputs, targets = val_round(epoch)
    outputs = np.array(outputs) >= 0.5
    accuracy = metrics.accuracy_score(targets, outputs)
    f1_score_micro = metrics.f1_score(targets, outputs, average='micro')
    f1_score_macro = metrics.f1_score(targets, outputs, average='macro')
    print(f"Accuracy Score = {accuracy}")
    print(f"F1 Score (Micro) = {f1_score_micro}")
    print(f"F1 Score (Macro) = {f1_score_macro}")
```

6. PREDICTING VALUES WITH THE MODEL

- Create a function to use the model to predict toxicity classes based on an input text



```python
def inference(X):
    load_model.eval()
    with torch.no_grad():
            ids = X['input_ids'].to(device, dtype = torch.long)
            mask = X['attention_mask'].to(device, dtype = torch.long)
            token_type_ids = X['token_type_ids'].to(device, dtype = torch.long)
            outputs = load_model(ids, mask, token_type_ids)
            print(outputs)
            fin_outputs = torch.sigmoid(outputs).cpu().detach().numpy().tolist()
    return fin_outputs
```

EXAMPLE

```
text = "Thank you for
understanding. I think very
highly of you and would not
revert without discussion."
```

Output Class Probabilities:

```
[[0.07172095030546188,
0.0038206269964575768,
0.08545476943254471,
0.00023505152785219252
, 0.06668499857187271,
0.0045667183585464954]]
```

No toxicity detected.