Automated Essay Grading using Transformer Models

September 7, 2024

1 Import Libraries

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

[ ]:
```
MODEL_PATH = "drive/MyDrive/model//"
XLNET_MODEL_PATH = "drive/MyDrive/model/xlnet//"
ALBERT_MODEL_PATH = "drive/MyDrive/model/albert//"
```

[ ]:
```
DATA_PATH = "drive/MyDrive//training_set_rel3.tsv"
```

[ ]:
```
 pip install torch transformers scikit-learn pandas openai
```

[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import re
from tqdm import tqdm
from transformers import BertTokenizer
from torch.utils.data import TensorDataset, DataLoader, RandomSampler,␣
 ↪SequentialSampler
from transformers import (
    BertTokenizer, BertForSequenceClassification,
    RobertaTokenizer, RobertaForSequenceClassification,
    XLNetTokenizer, XLNetForSequenceClassification,
    ElectraTokenizer, ElectraForSequenceClassification,
    AlbertTokenizer, AlbertForSequenceClassification,
    PreTrainedTokenizer, PreTrainedModel
)
from sklearn.metrics import cohen_kappa_score
from transformers import AdamW, get_linear_schedule_with_warmup
import torch
from sklearn.linear_model import LinearRegression
import openai
from sklearn.model_selection import train_test_split
```

2 Explore data

Some very good visualizations that help us understand the dataset come from this notebook : https://github.com/Turanga1/Automated-Essay-Scoring/blob/master/0_EDA_and_Topic_Modeling_with_LDA.ipynb

[ ]:
```
df = pd.read_csv(DATA_PATH, sep='\t', encoding='ISO-8859-1')
df.rename(columns={'essay': 'essay','domain1_score' : 'grade'}, inplace=True)
df = df[["essay_id","essay_set","essay","grade"]]

df.head(3)
```

2.1 Analyze the train_df_df_dfing set size per essay set

[ ]:
```
df.groupby('essay_set').agg('count').plot.bar(y='essay', legend=False)
plt.title('Number of essays per set')
plt.ylabel('Count')
plt.show()
```

2.2 Analyze the distribution of words per essay set

[ ]:
```
df['word_count'] = df['essay'].str.strip().str.split().str.len()
df.hist(column='word_count', by='essay_set', bins=25, sharey=True, sharex=True,␣
 ↪layout=(2, 4), figsize=(7,4), rot=0)
plt.suptitle('Word count by essay set')
plt.xlabel('Number of words')
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

[ ]:
```
topic_number = 0
fig, ax = plt.subplots(4,2, figsize=(8,10))
for i in range(4):
    for j in range(2):
        topic_number += 1
        sns.violinplot(x='grade', y='word_count', data=df[df['essay_set'] ==␣
 ↪topic_number], ax=ax[i,j])
        ax[i,j].set_title('Topic %i' % topic_number)
ax[3,0].locator_params(nbins=10)
ax[3,1].locator_params(nbins=10)
plt.suptitle('Word count by score')
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

It appears that longer essays tend to score better for all essay sets.

[ ]:
```
essay_set_number = 0
fig, ax = plt.subplots(4,2, figsize=(9,9), sharey=False)
```

```
for i in range(4):
    for j in range(2):
        essay_set_number+= 1
        df[df['essay_set'] == essay_set_number]\
            .groupby('grade')['essay_id']\
            .agg('count')\
            .plot.bar(ax=ax[i, j], rot=0)
        ax[i,j].set_title('Essay set %i' % essay_set_number)
ax[3,0].locator_params(nbins=10)
ax[3,1].locator_params(nbins=10)
plt.suptitle('Histograms of essay scores')
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

3 Finetune : BERT, RoBERTa, XLNet, ELECTRA, ALBERT

[ ]:
```
import torch
from transformers import BertTokenizer, BertForSequenceClassification, AdamW,␣
↪get_linear_schedule_with_warmup
from torch.utils.data import TensorDataset, DataLoader, RandomSampler,␣
↪SequentialSampler
import torch.nn as nn

class EssayGrader:
    def __init__(self, model_name: str, use_cuda: bool = True):
        self.tokenizer = BertTokenizer.from_pretrained(model_name)
        self.model = BertForSequenceClassification.from_pretrained(model_name,␣
↪num_labels=1)
        self.device = torch.device('cuda' if torch.cuda.is_available() and␣
↪use_cuda else 'cpu')
        self.model = self.model.to(self.device)
        self.model_name = model_name

    def save_model(self, path):
        self.model.save_pretrained(path)
        self.tokenizer.save_pretrained(path)

    def encode_data(self, df):
        input_ids = []
        attention_masks = []

        for essay in df['essay']:
            encoded_dict = self.tokenizer.encode_plus(
                essay,
                add_special_tokens=True,
                max_length=512,
```

```
                padding='max_length', # change pad_to_max_length to padding
                return_attention_mask=True,
                return_tensors='pt',
                truncation=True # handle sequences longer than model max input␣
↪length
            )

            input_ids.append(encoded_dict['input_ids'])
            attention_masks.append(encoded_dict['attention_mask'])

        input_ids = torch.cat(input_ids, dim=0)
        attention_masks = torch.cat(attention_masks, dim=0)
        labels = torch.tensor(df['grade'].values, dtype=torch.float32)

        return input_ids, attention_masks, labels

    def prepare_dataloader(self, train_df, test_df, batch_size=8):
        train_input_ids, train_attention_masks, train_labels = self.
↪encode_data(train_df)
        test_input_ids, test_attention_masks, test_labels = self.
↪encode_data(test_df)

        train_data = TensorDataset(train_input_ids, train_attention_masks,␣
↪train_labels)
        test_data = TensorDataset(test_input_ids, test_attention_masks,␣
↪test_labels)

        self.train_dataloader = DataLoader(
            train_data,
            sampler=RandomSampler(train_data),
            batch_size=batch_size
        )

        self.validation_dataloader = DataLoader(
            test_data,
            sampler=SequentialSampler(test_data),
            batch_size=batch_size
        )

    def train_model(self, epochs=1):
      optimizer = AdamW(self.model.parameters(), lr=2e-5, eps=1e-8)
      total_steps = len(self.train_dataloader) * epochs
      scheduler = get_linear_schedule_with_warmup(optimizer,␣
↪num_warmup_steps=0, num_training_steps=total_steps)
      loss_fn = nn.MSELoss()
```

```
        for epoch in range(epochs):
            self.model.train()
            total_train_loss = 0.0

            for step, batch in enumerate(self.train_dataloader):
                batch = tuple(t.to(self.device) for t in batch)
                input_ids, attention_masks, labels = batch

                self.model.zero_grad()
                outputs = self.model(
                    input_ids=input_ids,
                    attention_mask=attention_masks,
                    labels=labels
                )
                logits = outputs.logits
                loss = loss_fn(logits.squeeze(), labels)
                total_train_loss += loss.item()

                loss.backward()
                torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)

                optimizer.step()
                scheduler.step()

            avg_train_loss = total_train_loss / len(self.train_dataloader)
            print(f"Epoch {epoch + 1}/{epochs} - Average training loss: 
↪{avg_train_loss:.4f}")


    def generate_predictions(self, test_df):
        self.model.eval()
        input_ids, attention_masks, _ = self.encode_data(test_df)
        test_data = TensorDataset(input_ids, attention_masks)
        test_dataloader = DataLoader(
            test_data,
            sampler=SequentialSampler(test_data),
            batch_size=8
        )

        predictions = []

        for batch in test_dataloader:
            batch = tuple(t.to(self.device) for t in batch)
            input_ids, attention_masks = batch

            with torch.no_grad():
                outputs = self.model(
```

```
                    input_ids=input_ids,
                    token_type_ids=None,
                    attention_mask=attention_masks
                )

            logits = outputs.logits
            predicted_grades = logits.squeeze().cpu().numpy().tolist()
            predictions.extend(predicted_grades)

        df_predictions = test_df.copy()
        df_predictions['Predicted Grade'] = predictions

        return df_predictions
```

4 Run models

5 Split Dataset

[ ]:
```
# Convert grade to float32
df['grade'] = df['grade'].astype('float32')

# First, we split our dataset (80%/20%)
test_size = 0.2

train_df, test_df = train_test_split(df, test_size=test_size, random_state=42)
```

[ ]:
```
# train_df = train_df[["essay","grade"]]
```

5.1 BERT

[ ]:
```
# Initialize model
model_name = 'bert-base-uncased'
grader_bert = EssayGrader(model_name)

# Prepare dataloader
grader_bert.prepare_dataloader(train_df, test_df, batch_size=8)

# Train model
grader_bert.train_model(epochs=5)
```

[ ]:
```
# Save the model so that it doesn't have to run again

grader_bert.save_model(MODEL_PATH)
```

[ ]:
```python
# Generate predictions
predictions_df_bert = grader_bert.generate_predictions(test_df)
```

[ ]:
```python
from sklearn.metrics import cohen_kappa_score

# 'grade' and 'Predicted Grade' are column names containing actual and␣
 ↪predicted grades
y_true = test_df['grade']
y_pred_bert = predictions_df_bert['Predicted Grade'].round()  # round the␣
 ↪predictions to make them discrete

qwk_bert = cohen_kappa_score(y_true, y_pred_bert, weights='quadratic')

print(f"Quadratic Weighted Kappa (QWK) is: {qwk_bert}")
```

[ ]:
```python
# Make sure the training and testing data do not overlap
assert len(set(train_df.index) & set(test_df.index)) == 0
```

[ ]:
```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model_name = MODEL_PATH
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertForSequenceClassification.from_pretrained(model_name)
model = model.to(device)
```

[ ]:
```python
def generate_predictions_new(model, dataloader):
    model.eval()
    predictions = []

    for batch in dataloader:
        batch = tuple(t.to(device) for t in batch)
        input_ids, attention_masks = batch

        with torch.no_grad():
            outputs = model(
                input_ids=input_ids,
                token_type_ids=None,
                attention_mask=attention_masks
            )

        logits = outputs.logits
        predicted_grades = logits.squeeze().cpu().numpy().tolist()
        predictions.extend(predicted_grades)

    return predictions

# Prepare your DataLoader for the test data
```

```python
# You can use the encode_data and DataLoader initialization logic from the `EssayGrader` class here
test_input_ids, test_attention_masks, _ = grader_bert.encode_data(test_df.head())
test_data = TensorDataset(test_input_ids, test_attention_masks)
test_dataloader = DataLoader(test_data, sampler=SequentialSampler(test_data), batch_size=8)

# Generate predictions
predictions = generate_predictions_new(model, test_dataloader)

# Add the predictions to the DataFrame
temp = test_df.head().copy()
temp['Predicted'] = predictions
```

[ ]:

```python

```

[ ]:

```python
temp
```

5.2 XLNet

[ ]:

```python
!pip install sentencepiece
```

[ ]:

```python
import torch
from transformers import XLNetTokenizer, XLNetForSequenceClassification, AdamW, get_linear_schedule_with_warmup
from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler
import torch.nn as nn
import sentencepiece
from sklearn.model_selection import train_test_split
import pandas as pd

class EssayGrader:
    def __init__(self, model_name: str, use_cuda: bool = True):
        self.tokenizer = XLNetTokenizer.from_pretrained(model_name)
        self.model = XLNetForSequenceClassification.from_pretrained(model_name, num_labels=1)
        self.device = torch.device('cuda' if torch.cuda.is_available() and use_cuda else 'cpu')
        self.model = self.model.to(self.device)
        self.model_name = model_name

    def save_model(self, path):
        self.model.save_pretrained(path)
        self.tokenizer.save_pretrained(path)
```

```python
    def encode_data(self, df):
        input_ids = []
        attention_masks = []

        for essay in df['essay']:
            encoded_dict = self.tokenizer.encode_plus(
                essay,
                add_special_tokens=True,
                max_length=512,
                padding='max_length', # change pad_to_max_length to padding
                return_attention_mask=True,
                return_tensors='pt',
                truncation=True # handle sequences longer than model max␣
↪input length
            )

            input_ids.append(encoded_dict['input_ids'])
            attention_masks.append(encoded_dict['attention_mask'])

        input_ids = torch.cat(input_ids, dim=0)
        attention_masks = torch.cat(attention_masks, dim=0)
        labels = torch.tensor(df['grade'].values, dtype=torch.float32)

        return input_ids, attention_masks, labels

    def prepare_dataloader(self, train_df, test_df, batch_size=8):
        train_input_ids, train_attention_masks, train_labels = self.
↪encode_data(train_df)
        test_input_ids, test_attention_masks, test_labels = self.
↪encode_data(test_df)

        train_data = TensorDataset(train_input_ids, train_attention_masks,␣
↪train_labels)
        test_data = TensorDataset(test_input_ids, test_attention_masks,␣
↪test_labels)

        self.train_dataloader = DataLoader(
            train_data,
            sampler=RandomSampler(train_data),
            batch_size=batch_size
        )

        self.validation_dataloader = DataLoader(
            test_data,
            sampler=SequentialSampler(test_data),
            batch_size=batch_size
```

```
        )

    def train_model(self, epochs=1):
        optimizer = AdamW(self.model.parameters(), lr=2e-5, eps=1e-8)
        total_steps = len(self.train_dataloader) * epochs
        scheduler = get_linear_schedule_with_warmup(optimizer,␣
↪num_warmup_steps=0, num_training_steps=total_steps)
        loss_fn = nn.MSELoss()

        for epoch in range(epochs):
            self.model.train()
            total_train_loss = 0.0

            for step, batch in enumerate(self.train_dataloader):
                batch = tuple(t.to(self.device) for t in batch)
                input_ids, attention_masks, labels = batch

                self.model.zero_grad()
                outputs = self.model(
                    input_ids=input_ids,
                    attention_mask=attention_masks,
                    labels=labels
                )
                logits = outputs.logits
                loss = loss_fn(logits.squeeze(), labels)
                total_train_loss += loss.item()

                loss.backward()
                torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)

                optimizer.step()
                scheduler.step()

            avg_train_loss = total_train_loss / len(self.train_dataloader)
            print(f"Epoch {epoch + 1}/{epochs} - Average training loss:␣
↪{avg_train_loss:.4f}")


    def generate_predictions(self, test_df):
        self.model.eval()
        input_ids, attention_masks, _ = self.encode_data(test_df)
        test_data = TensorDataset(input_ids, attention_masks)
        test_dataloader = DataLoader(
            test_data,
            sampler=SequentialSampler(test_data),
            batch_size=8
        )
```

```
        predictions = []

        for batch in test_dataloader:
            batch = tuple(t.to(self.device) for t in batch)
            input_ids, attention_masks = batch

            with torch.no_grad():
                outputs = self.model(
                    input_ids=input_ids,
                    token_type_ids=None,
                    attention_mask=attention_masks
                )

            logits = outputs.logits
            predicted_grades = logits.squeeze().cpu().numpy().tolist()
            predictions.extend(predicted_grades)

        df_predictions = test_df.copy()
        df_predictions['Predicted Grade'] = predictions

        return df_predictions
```

[ ]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
DATA_PATH = "drive/MyDrive//training_set_rel3.tsv"
df = pd.read_csv(DATA_PATH, sep='\t', encoding='ISO-8859-1')
df.rename(columns={'essay': 'essay','domain1_score' : 'grade'}, inplace=True)
df = df[["essay_id","essay_set","essay","grade"]]

# Convert grade to float32
df['grade'] = df['grade'].astype('float32')

# First, we split our dataset (80%/20%)
test_size = 0.2

train_df, test_df = train_test_split(df, test_size=test_size, random_state=42)
```

[ ]:
```
# Usage

model_name = 'xlnet-base-cased'
grader_xlnet = EssayGrader(model_name)
```

[ ]:
```
grader_xlnet.prepare_dataloader(train_df, test_df, batch_size=8)
grader_xlnet.train_model(epochs=5)
predictions_df_xlnet = grader_xlnet.generate_predictions(test_df)
predictions_df_xlnet
```

```
[ ]: from sklearn.metrics import cohen_kappa_score

     # 'grade' and 'Predicted Grade' are column names containing actual and␣
      ↪predicted grades
     y_true = test_df['grade']
     y_pred_xlnet = predictions_df_xlnet['Predicted Grade'].round()  # round the␣
      ↪predictions to make them discrete

     qwk_xlnet = cohen_kappa_score(y_true, y_pred_xlnet, weights='quadratic')

     print(f"Quadratic Weighted Kappa (QWK) is: {qwk_xlnet}")
```

```
[ ]: grader_xlnet.save_model(XLNET_MODEL_PATH)
```

5.3 ELECTRA

```
[ ]: import torch
     from transformers import ElectraTokenizer, ElectraForSequenceClassification,␣
      ↪AdamW, get_linear_schedule_with_warmup
     from torch.utils.data import TensorDataset, DataLoader, RandomSampler,␣
      ↪SequentialSampler
     import torch.nn as nn

     class EssayGrader:
         def __init__(self, model_name: str, use_cuda: bool = True):
             self.tokenizer = ElectraTokenizer.from_pretrained(model_name)
             self.model = ElectraForSequenceClassification.
      ↪from_pretrained(model_name, num_labels=1)
             self.device = torch.device('cuda' if torch.cuda.is_available() and␣
      ↪use_cuda else 'cpu')
             self.model = self.model.to(self.device)
             self.model_name = model_name

         def save_model(self, path):
             self.model.save_pretrained(path)
             self.tokenizer.save_pretrained(path)

         def encode_data(self, df):
               input_ids = []
               attention_masks = []

               for essay in df['essay']:
                   encoded_dict = self.tokenizer.encode_plus(
                       essay,
                       add_special_tokens=True,
                       max_length=512,
```

```
                padding='max_length', # change pad_to_max_length to padding
                return_attention_mask=True,
                return_tensors='pt',
                truncation=True # handle sequences longer than model max␣
↪input length
            )

            input_ids.append(encoded_dict['input_ids'])
            attention_masks.append(encoded_dict['attention_mask'])

        input_ids = torch.cat(input_ids, dim=0)
        attention_masks = torch.cat(attention_masks, dim=0)
        labels = torch.tensor(df['grade'].values, dtype=torch.float32)

        return input_ids, attention_masks, labels

    def prepare_dataloader(self, train_df, test_df, batch_size=8):
        train_input_ids, train_attention_masks, train_labels = self.
↪encode_data(train_df)
        test_input_ids, test_attention_masks, test_labels = self.
↪encode_data(test_df)

        train_data = TensorDataset(train_input_ids, train_attention_masks,␣
↪train_labels)
        test_data = TensorDataset(test_input_ids, test_attention_masks,␣
↪test_labels)

        self.train_dataloader = DataLoader(
            train_data,
            sampler=RandomSampler(train_data),
            batch_size=batch_size
        )

        self.validation_dataloader = DataLoader(
            test_data,
            sampler=SequentialSampler(test_data),
            batch_size=batch_size
        )

    def train_model(self, epochs=1):
        optimizer = AdamW(self.model.parameters(), lr=2e-5, eps=1e-8)
        total_steps = len(self.train_dataloader) * epochs
        scheduler = get_linear_schedule_with_warmup(optimizer,␣
↪num_warmup_steps=0, num_training_steps=total_steps)
        loss_fn = nn.MSELoss()
```

```
        for epoch in range(epochs):
            self.model.train()
            total_train_loss = 0.0

            for step, batch in enumerate(self.train_dataloader):
                batch = tuple(t.to(self.device) for t in batch)
                input_ids, attention_masks, labels = batch

                self.model.zero_grad()
                outputs = self.model(
                    input_ids=input_ids,
                    attention_mask=attention_masks,
                    labels=labels
                )
                logits = outputs.logits
                loss = loss_fn(logits.squeeze(), labels)
                total_train_loss += loss.item()

                loss.backward()
                torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)

                optimizer.step()
                scheduler.step()

            avg_train_loss = total_train_loss / len(self.train_dataloader)
            print(f"Epoch {epoch + 1}/{epochs} - Average training loss: 
↪{avg_train_loss:.4f}")


    def generate_predictions(self, test_df):
        self.model.eval()
        input_ids, attention_masks, _ = self.encode_data(test_df)
        test_data = TensorDataset(input_ids, attention_masks)
        test_dataloader = DataLoader(
            test_data,
            sampler=SequentialSampler(test_data),
            batch_size=8
        )

        predictions = []

        for batch in test_dataloader:
            batch = tuple(t.to(self.device) for t in batch)
            input_ids, attention_masks = batch

            with torch.no_grad():
                outputs = self.model(
```

```
                    input_ids=input_ids,
                    token_type_ids=None,
                    attention_mask=attention_masks
                )

            logits = outputs.logits
            predicted_grades = logits.squeeze().cpu().numpy().tolist()
            predictions.extend(predicted_grades)

        df_predictions = test_df.copy()
        df_predictions['Predicted Grade'] = predictions

        return df_predictions
```

[ ]:
```
model_name = 'google/electra-small-discriminator'
grader_electra = EssayGrader(model_name)
grader_electra.prepare_dataloader(train_df, test_df, batch_size=8)
grader_electra.train_model(epochs=5)
predictions_df_electra = grader_electra.generate_predictions(test_df)
predictions_df_electra
```

[ ]:
```
# 'grade' and 'Predicted Grade' are column names containing actual and␣
 ↪predicted grades
y_true = test_df['grade']
y_pred_electra = predictions_df_electra['Predicted Grade'].round()  # round the␣
 ↪predictions to make them discrete

qwk_electra = cohen_kappa_score(y_true, y_pred_electra, weights='quadratic')

print(f"Quadratic Weighted Kappa (QWK) is: {qwk_electra}")
```

5.4 ALBERT

[ ]:
```
!pip install sentencepiece
```

[ ]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
DATA_PATH = "drive/MyDrive//training_set_rel3.tsv"
df = pd.read_csv(DATA_PATH, sep='\t', encoding='ISO-8859-1')
df.rename(columns={'essay': 'essay','domain1_score' : 'grade'}, inplace=True)
df = df[["essay_id","essay_set","essay","grade"]]

# Convert grade to float32
df['grade'] = df['grade'].astype('float32')

# First, we split our dataset (80%/20%)
```

```
test_size = 0.2

train_df, test_df = train_test_split(df, test_size=test_size, random_state=42)
```

[ ]:
```
import torch
from transformers import AlbertTokenizer, AlbertForSequenceClassification,␣
↪AdamW, get_linear_schedule_with_warmup
from torch.utils.data import TensorDataset, DataLoader, RandomSampler,␣
↪SequentialSampler
import torch.nn as nn

class EssayGrader:
    def __init__(self, model_name: str, use_cuda: bool = True):
        self.tokenizer = AlbertTokenizer.from_pretrained(model_name)
        self.model = AlbertForSequenceClassification.
↪from_pretrained(model_name, num_labels=1)
        self.device = torch.device('cuda' if torch.cuda.is_available() and␣
↪use_cuda else 'cpu')
        self.model = self.model.to(self.device)
        self.model_name = model_name

    def save_model(self, path):
        self.model.save_pretrained(path)
        self.tokenizer.save_pretrained(path)

    def encode_data(self, df):
          input_ids = []
          attention_masks = []

          for essay in df['essay']:
              encoded_dict = self.tokenizer.encode_plus(
                  essay,
                  add_special_tokens=True,
                  max_length=512,
                  padding='max_length', # change pad_to_max_length to padding
                  return_attention_mask=True,
                  return_tensors='pt',
                  truncation=True # handle sequences longer than model max␣
↪input length
              )

              input_ids.append(encoded_dict['input_ids'])
              attention_masks.append(encoded_dict['attention_mask'])

          input_ids = torch.cat(input_ids, dim=0)
          attention_masks = torch.cat(attention_masks, dim=0)
          labels = torch.tensor(df['grade'].values, dtype=torch.float32)
```

```
        return input_ids, attention_masks, labels

    def prepare_dataloader(self, train_df, test_df, batch_size=8):
        train_input_ids, train_attention_masks, train_labels = self.
↪encode_data(train_df)
        test_input_ids, test_attention_masks, test_labels = self.
↪encode_data(test_df)

        train_data = TensorDataset(train_input_ids, train_attention_masks,␣
↪train_labels)
        test_data = TensorDataset(test_input_ids, test_attention_masks,␣
↪test_labels)

        self.train_dataloader = DataLoader(
            train_data,
            sampler=RandomSampler(train_data),
            batch_size=batch_size
        )

        self.validation_dataloader = DataLoader(
            test_data,
            sampler=SequentialSampler(test_data),
            batch_size=batch_size
        )

    def train_model(self, epochs=1):
        optimizer = AdamW(self.model.parameters(), lr=2e-5, eps=1e-8)
        total_steps = len(self.train_dataloader) * epochs
        scheduler = get_linear_schedule_with_warmup(optimizer,␣
↪num_warmup_steps=0, num_training_steps=total_steps)
        loss_fn = nn.MSELoss()

        for epoch in range(epochs):
            self.model.train()
            total_train_loss = 0.0

            for step, batch in enumerate(self.train_dataloader):
                batch = tuple(t.to(self.device) for t in batch)
                input_ids, attention_masks, labels = batch

                self.model.zero_grad()
                outputs = self.model(
                    input_ids=input_ids,
                    attention_mask=attention_masks,
                    labels=labels
                )
```

```
                logits = outputs.logits
                loss = loss_fn(logits.squeeze(), labels)
                total_train_loss += loss.item()

                loss.backward()
                torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)

                optimizer.step()
                scheduler.step()

            avg_train_loss = total_train_loss / len(self.train_dataloader)
            print(f"Epoch {epoch + 1}/{epochs} - Average training loss: {avg_train_loss:.4f}")


    def generate_predictions(self, test_df):
        self.model.eval()
        input_ids, attention_masks, _ = self.encode_data(test_df)
        test_data = TensorDataset(input_ids, attention_masks)
        test_dataloader = DataLoader(
            test_data,
            sampler=SequentialSampler(test_data),
            batch_size=8
        )

        predictions = []

        for batch in test_dataloader:
            batch = tuple(t.to(self.device) for t in batch)
            input_ids, attention_masks = batch

            with torch.no_grad():
                outputs = self.model(
                    input_ids=input_ids,
                    token_type_ids=None,
                    attention_mask=attention_masks
                )

            logits = outputs.logits
            predicted_grades = logits.squeeze().cpu().numpy().tolist()
            predictions.extend(predicted_grades)

        df_predictions = test_df.copy()
        df_predictions['Predicted Grade'] = predictions

        return df_predictions
```

```
[ ]: model_name = 'albert-base-v2'
grader_albert = EssayGrader(model_name)
grader_albert.prepare_dataloader(train_df, test_df, batch_size=8)
grader_albert.train_model(epochs=10)
predictions_df_albert = grader_albert.generate_predictions(test_df)
predictions_df_albert
```

```
[ ]: # 'grade' and 'Predicted Grade' are column names containing actual and␣
   ↪predicted grades
y_true = test_df['grade']
y_pred_albert = predictions_df_albert['Predicted Grade'].round()  # round the␣
   ↪predictions to make them discrete

qwk_albert = cohen_kappa_score(y_true, y_pred_albert, weights='quadratic')

print(f"Quadratic Weighted Kappa (QWK) is: {qwk_albert}")
```

```
[ ]: from google.colab import drive
drive.mount('/content/gdrive')
# Saving BERT model
model_path = "/content/gdrive/MyDrive/bert_model.pth"
torch.save(grader_bert.model.state_dict(), model_path)

# Saving XLNet model
model_path = "/content/gdrive/MyDrive/xlnet_model.pth"
torch.save(grader_xlnet.model.state_dict(), model_path)

# Saving Electra model
model_path = "/content/gdrive/MyDrive/electra_model.pth"
torch.save(grader_electra.model.state_dict(), model_path)

# Saving Albert model
model_path = "/content/gdrive/MyDrive/albert_model.pth"
torch.save(grader_albert.model.state_dict(), model_path)
```

```
[ ]: # Loading BERT model
model_path = "/content/gdrive/MyDrive/bert_model.pth"
bert_model = BertForSequenceClassification.from_pretrained(model_name,␣
   ↪num_labels=1)
bert_model.load_state_dict(torch.load(model_path))
bert_model.eval()

# Loading XLNet model
model_path = "/content/gdrive/MyDrive/xlnet_model.pth"
xlnet_model = XLNetForSequenceClassification.from_pretrained(model_name,␣
   ↪num_labels=1)
xlnet_model.load_state_dict(torch.load(model_path))
```

```
xlnet_model.eval()

# Loading Electra model
model_path = "/content/gdrive/MyDrive/electra_model.pth"
electra_model = ElectraForSequenceClassification.from_pretrained(model_name,␣
 ↪num_labels=1)
electra_model.load_state_dict(torch.load(model_path))
electra_model.eval()

# Loading Albert model
model_path = "/content/gdrive/MyDrive/albert_model.pth"
albert_model = AlbertForSequenceClassification.from_pretrained(model_name,␣
 ↪num_labels=1)
albert_model.load_state_dict(torch.load(model_path))
albert_model.eval()
```

[ ]:

```
```

6 GPT-3 few-shot learning

[ ]:

```
openai.api_key = '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'

def grade_essay_one_gpt3(essay, essay_set):
    prompt = f"""
    Context: 'You are an AI model with extensive knowledge in language and␣
 ↪writing styles. You're provided with a pool of essays, each corresponding to␣
 ↪a specific essay set with a unique grading rubric.
    The grading rubrics:
    Essay Set 1 Rubric:
          Rubric range: 1-6
          Score Point 1: An undeveloped response that may take a position but␣
 ↪offers no more than very minimal support. Typical elements:
          Contains few or vague details, ss awkward and fragmented, may be␣
 ↪difficult to read and understand, may show no awareness of audience.
          Score Point 2: An under-developed response that may or may not take a␣
 ↪position. Typical elements:
          Contains only general reasons with unelaborated and/or list-like␣
 ↪details, shows little or no evidence of organization, may be awkward and␣
 ↪confused or simplistic, may show little awareness of audience.
          Score Point 3: A minimally-developed response that may take a␣
 ↪position, but with inadequate support and details. Typical elements:
          Has reasons with minimal elaboration and more general than specific␣
 ↪details, shows some organization, may be awkward in parts with few␣
 ↪transitions, shows some awareness of audience.
```

```
        Score Point 4: A somewhat-developed response that takes a position␣
↪and provides adequate support. Typical elements:
        Has adequately elaborated reasons with a mix of general and specific␣
↪details, shows satisfactory organization, may be somewhat fluent with some␣
↪transitional language, shows adequate awareness of audience.
        Score Point 5: A developed response that takes a clear position and␣
↪provides reasonably persuasive support. Typical elements:
        Has moderately well elaborated reasons with mostly specific details,␣
↪exhibits generally strong organization, may be moderately fluent with␣
↪transitional language throughout, may show a consistent awareness of␣
↪audience.
        Score Point 6: A well-developed response that takes a clear and␣
↪thoughtful position and provides persuasive support. Typical elements:
        Has fully elaborated reasons with specific details, exhibits strong␣
↪organization, is fluent and uses sophisticated transitional language, may␣
↪show a heightened awareness of audience.


    Instruction:
    Given the provided essay and its essay set number, apply the grading rubric␣
↪associated with that essay set and provide a grade as an integer. Important:␣
↪You should produce a number without any text!

    YOUR ANSWER SHOULD ONLY BE A NUMBER. JUST A NUMBER, NO OTHER WORDS, NOTHING␣
↪ELSE.

    Essay set: "{essay_set}"

    {essay}

    """

    response = openai.Completion.create(
        engine="text-davinci-002",
        prompt=prompt,
        temperature=0.5,
        max_tokens=10
    )

    output = response.choices[0].text.strip()
    return output # Convert the grade from string to integer
```

[ ]:
```
def grade_essay_two_gpt3(essay, essay_set):
    prompt = f"""
```

```
    Context: 'You are an AI model with extensive knowledge in language and writing styles. You're provided with a pool of essays, each corresponding to a specific essay set with a unique grading rubric.
    The grading rubrics:
Essay Set 2 Rubric:
          We add the scores from the following two domains:
          Rubric range domain 1: 1-6
          Score Point 6: A Score Point 6 paper is rare. It fully accomplishes the task in a thorough and insightful manner and has a distinctive quality that sets it apart as an outstanding performance.
          Score Point 5: A Score Point 5 paper represents a solid performance. It fully accomplishes the task, but lacks the overall level of sophistication and consistency of a Score Point 6 paper.
          Score Point 4: A Score Point 4 paper represents a good performance. It accomplishes the task, but generally needs to exhibit more development, better organization, or a more sophisticated writing style to receive a higher score.
          Score Point 3: A Score Point 3 paper represents a performance that minimally accomplishes the task. Some elements of development, organization, and writing style are weak.
          Score Point 2: A Score Point 2 paper represents a performance that only partially accomplishes the task. Some responses may exhibit difficulty maintaining a focus. Others may be too brief to provide sufficient development of the topic or evidence of adequate organizational or writing style.
          Score Point 1: A Score Point 1 paper represents a performance that fails to accomplish the task. It exhibits considerable difficulty in areas of development, organization, and writing style. The writing is generally either very brief or rambling and repetitive, sometimes resulting in a response that may be difficult to read or comprehend.
          Rubric range domain 2: 1-4
          Score 4: Does the writing sample exhibit a superior command of language skills?
          A Score Point 4 paper exhibits a superior command of written English language conventions. The paper provides evidence that the student has a thorough control of the concepts outlined in the Indiana Academic Standards associated with the student’s grade level. In a Score Point 4 paper, there are no errors that impair the flow of communication. Errors are generally of the first-draft variety or occur when the student attempts sophisticated sentence construction.
          Score 3: Does the writing sample exhibit a good control of language skills?
          In a Score Point 3 paper, errors are occasional and are often of the first-draft variety; they have a minor impact on the flow of communication.
          Score 2: Does the writing sample exhibit a fair control of language skills?
```

```
        In a Score Point 2 paper, errors are typically frequent and may␣
↪occasionally impede the flow of communication.
        Score 1: Does the writing sample exhibit a minimal or less than␣
↪minimal control of language skills?
        In a Score Point 1 paper, errors are serious and numerous. The reader␣
↪may need to stop and reread part of the sample and may struggle to discern␣
↪the writer’s meaning.
    Essay Set 3 and 4 Rubric:

    Instruction:
    Given the provided essay and its essay set number, apply the grading rubric␣
↪associated with that essay set and provide a grade as an integer. Important:␣
↪You should produce a number without any text!

    YOUR ANSWER SHOULD ONLY BE A NUMBER. JUST A NUMBER, NO OTHER WORDS, NOTHING␣
↪ELSE.

    Essay set: "{essay_set}"

    {essay}

    """

    response = openai.Completion.create(
        engine="text-davinci-002",
        prompt=prompt,
        temperature=0.5,
        max_tokens=10
    )

    output = response.choices[0].text.strip()
    return output # Convert the grade from string to integer
```

[ ]:
```
def grade_essay_three_four_gpt3(essay, essay_set):
    prompt = f"""
    Context: 'You are an AI model with extensive knowledge in language and␣
↪writing styles. You're provided with a pool of essays, each corresponding to␣
↪a specific essay set with a unique grading rubric.
    The grading rubrics:

    Essay Set 3 and 4 Rubric:
    Rubric range: 0-3
        Score 0: The response is completely irrelevant or incorrect, or there␣
↪is no response.
        Score 1: The response shows evidence of a minimal understanding of␣
↪the text. Typical elements:
```

```
        May show evidence that some meaning has been derived from the text,␣
↪may indicate a misreading of the text or the question, may lack information␣
↪or explanation to support an understanding of the text in relation to the␣
↪question
        Score 2: The response demonstrates a partial or literal understanding␣
↪of the text. Typical elements:
        Addresses the demands of the question, although may not develop all␣
↪parts equally, uses some expressed or implied information from the text to␣
↪demonstrate understanding, may not fully connect the support to a conclusion␣
↪or assertion made about the text(s)
        Score 3: The response demonstrates an understanding of the␣
↪complexities of the text. Typical elements:
        Addresses the demands of the question, uses expressed and implied␣
↪information from the text, clarifies and extends understanding beyond the␣
↪literal

    Instruction:
    Given the provided essay and its essay set number, apply the grading rubric␣
↪associated with that essay set and provide a grade as an integer. Important:␣
↪You should produce a number without any text!

    YOUR ANSWER SHOULD ONLY BE A NUMBER. JUST A NUMBER, NO OTHER WORDS, NOTHING␣
↪ELSE.

    Essay set: "{essay_set}"

    {essay}

    """

    response = openai.Completion.create(
        engine="text-davinci-002",
        prompt=prompt,
        temperature=0.5,
        max_tokens=10
    )

    output = response.choices[0].text.strip()
    return output # Convert the grade from string to integer
```

[ ]:
```
def grade_essay_five_six_gpt3(essay, essay_set):
    prompt = f"""
    Context: 'You are an AI model with extensive knowledge in language and␣
↪writing styles. You're provided with a pool of essays, each corresponding to␣
↪a specific essay set with a unique grading rubric.
    The grading rubrics:
```

```
    Essay Set 5 and 6 Rubric:
          Rubric range: 0-4
          Score Point 0: The response is incorrect or irrelevant or contains␣
↪insufficient information to demonstrate comprehension.
          Score Point 1: The response is a minimal description of the mood␣
↪created by the author. The response includes little or no information from␣
↪the memoir and may include misinterpretations or the response relates␣
↪minimally to the task.
          Score Point 2: The response is a partial description of the mood␣
↪created by the author. The response includes limited information from the␣
↪memoir and may include misinterpretations.
          Score Point 3: The response is a mostly clear, complete, and accurate␣
↪description of the mood created by the author. The response includes␣
↪relevant but often general information from the memoir.
          Score Point 4: The response is a clear, complete, and accurate␣
↪description of the mood created by the author. The response includes␣
↪relevant and specific information from the memoir.

    Instruction:
    Given the provided essay and its essay set number, apply the grading rubric␣
↪associated with that essay set and provide a grade as an integer. Important:␣
↪You should produce a number without any text!

    YOUR ANSWER SHOULD ONLY BE A NUMBER. JUST A NUMBER, NO OTHER WORDS, NOTHING␣
↪ELSE.

    Essay set: "{essay_set}"

    {essay}

    """

    response = openai.Completion.create(
        engine="text-davinci-002",
        prompt=prompt,
        temperature=0.5,
        max_tokens=10
    )

    output = response.choices[0].text.strip()
    return output # Convert the grade from string to integer
```

[ ]:
```
def grade_essay_seven_gpt3(essay, essay_set):
    prompt = f"""
```

```
   Context: 'You are an AI model with extensive knowledge in language and␣
↪writing styles. You're provided with a pool of essays, each corresponding to␣
↪a specific essay set with a unique grading rubric.
   The grading rubrics:


   Essay Set 7 Rubric:
         Rubric range: 0-15
         We add the scores from each of following four traits:
         Ideas
         Score 6: Tells a story with ideas that are clearly focused on the␣
↪topic and are thoroughly developed with specific, relevant details.
         Score 4: Tells a story with ideas that are somewhat focused on the␣
↪topic and are developed with a mix of specific and/or general details.
         Score 2: Tells a story with ideas that are minimally focused on the␣
↪topic and developed with limited and/or general details.
         Score 0: Ideas are not focused on the task and/or are undeveloped.
         Organization
         Score 3: Organization and connections between ideas and/or events are␣
↪clear and logically sequenced.
         Score 2: Organization and connections between ideas and/or events are␣
↪logically sequenced.
         Score 1: Organization and connections between ideas and/or events are␣
↪weak.
         Score 0: No organization evident.
         Style
         Score 3: Command of language, including effective and compelling word␣
↪choice and varied sentence structure, clearly supports the writer's purpose␣
↪and audience.
         Score 2: Adequate command of language, including effective word␣
↪choice and clear sentences, supports the writer's purpose and audience.
         Score 1: Limited use of language, including lack of variety in word␣
↪choice and sentences, may hinder support for the writer's purpose and␣
↪audience.
         Score 0: Ineffective use of language for the writer's purpose and␣
↪audience.
         Conventions
         Score 3: Consistent, appropriate use of conventions of Standard␣
↪English for grammar, usage, spelling, capitalization, and punctuation for␣
↪the grade level.
         Score 2: Adequate use of conventions of Standard English for grammar,␣
↪usage, spelling, capitalization, and punctuation for the grade level.
         Score 1: Limited use of conventions of Standard English for grammar,␣
↪usage, spelling, capitalization, and punctuation for the grade level.
         Score 0: Ineffective use of conventions of Standard English for␣
↪grammar, usage, spelling, capitalization, and punctuation.
```

```
    Instruction:
    Given the provided essay and its essay set number, apply the grading rubric␣
↪associated with that essay set and provide a grade as an integer. Important:␣
↪You should produce a number without any text!

    YOUR ANSWER SHOULD ONLY BE A NUMBER. JUST A NUMBER, NO OTHER WORDS, NOTHING␣
↪ELSE.

    Essay set: "{essay_set}"

    {essay}

    """

    response = openai.Completion.create(
        engine="text-davinci-002",
        prompt=prompt,
        temperature=0.5,
        max_tokens=10
    )

    output = response.choices[0].text.strip()
    return output # Convert the grade from string to integer
```

[ ]:
```
def grade_essay_eight_gpt3(essay, essay_set):
    prompt = f"""
    Context: 'You are an AI model with extensive knowledge in language and␣
↪writing styles. You're provided with a pool of essays, each corresponding to␣
↪a specific essay set with a unique grading rubric.
    The grading rubrics:


    Essay Set 8 Rubric:
          Rubric Guidelines
          A rating of 1-6 on the following six traits:
          Ideas and Content
          Score 12: The writing is exceptionally clear, focused, and␣
↪interesting. It holds the reader's attention throughout. Main ideas stand␣
↪out and are developed by strong support and rich details suitable to␣
↪audience and purpose. The writing is characterized by
          Score 10: The writing is clear, focused and interesting. It holds the␣
↪reader's attention. Main ideas stand out and are developed by supporting␣
↪details suitable to audience and purpose. The writing is characterized by
          Score 8: The writing is clear and focused. The reader can easily␣
↪understand the main ideas. Support is present, although it may be limited or␣
↪rather general. The writing is characterized by
```

```
        Score 6: The reader can understand the main ideas, although they may be overly broad or simplistic, and the results may not be effective. Supporting detail is often limited, insubstantial, overly general, or occasionally slightly off-topic. The writing is characterized by
        Score 4: Main ideas and purpose are somewhat unclear or development is attempted but minimal. The writing is characterized by
        Score 2: The writing lacks a central idea or purpose.
        Organization
        Score 12: The organization enhances the central idea(s) and its development. The order and structure are compelling and move the reader through the text easily.
        Score 10: The organization enhances the central idea(s) and its development. The order and structure are strong and move the reader through the text.
        Score 8: Organization is clear and coherent. Order and structure are present, but may seem formulaic. The writing is characterized by
        Score 6: An attempt has been made to organize the writing; however, the overall structure is inconsistent or skeletal.
        Score 4: The writing lacks a clear organizational structure. An occasional organizational device is discernible; however, the writing is either difficult to follow and the reader has to reread substantial portions, or the piece is simply too short to demonstrate organizational skills.
        Score 2: The writing lacks coherence; organization seems haphazard and disjointed. Even after rereading, the reader remains confused.
        Sentence Fluency
        Score 12: The writing has an effective flow and rhythm. Sentences show a high degree of craftsmanship, with consistently strong and varied structure that makes expressive oral reading easy and enjoyable.
        Score 10: The writing has an easy flow and rhythm. Sentences are carefully crafted, with strong and varied structure that makes expressive oral reading easy and enjoyable.
        Score 8: The writing flows; however, connections between phrases or sentences may be less than fluid. Sentence patterns are somewhat varied, contributing to ease in oral reading.
        Score 6: The writing tends to be mechanical rather than fluid. Occasional awkward constructions may force the reader to slow down or reread.
        Score 4: The writing tends to be either choppy or rambling. Awkward constructions often force the reader to slow down or reread.
        Score 2: The writing is difficult to follow or to read aloud. Sentences tend to be incomplete, rambling, or very awkward.
        Conventions
```

```
        Score 24: The writing demonstrates exceptionally strong control of␣
↪standard writing conventions (e.g., punctuation, spelling, capitalization,␣
↪grammar and usage) and uses them effectively to enhance communication.␣
↪Errors are so few and so minor that the reader can easily skim right over␣
↪them unless specifically searching for them.
        Score 20: The writing demonstrates strong control of standard writing␣
↪conventions (e.g., punctuation, spelling, capitalization, grammar and usage)␣
↪and uses them effectively to enhance communication. Errors are few and minor.
↪ Conventions support readability.
        Score 16: The writing demonstrates control of standard writing␣
↪conventions (e.g., punctuation, spelling, capitalization, grammar and usage).
↪ Significant errors do not occur frequently. Minor errors, while perhaps␣
↪noticeable, do not impede readability.
        Score 12: The writing demonstrates limited control of standard␣
↪writing conventions (e.g., punctuation, spelling, capitalization, grammar␣
↪and usage). Errors begin to impede readability.
        Score 8: The writing demonstrates little control of standard writing␣
↪conventions. Frequent, significant errors impede readability.
        Score 4: Numerous errors in usage, spelling, capitalization, and␣
↪punctuation repeatedly distract the reader and make the text difficult to␣
↪read. In fact, the severity and frequency of errors are so overwhelming that␣
↪the reader finds it difficult to focus on the message and must reread for␣
↪meaning.

  Instruction:
  Given the provided essay and its essay set number, apply the grading rubric␣
↪associated with that essay set and provide a grade as an integer. Important:␣
↪You should produce a number without any text!

  YOUR ANSWER SHOULD ONLY BE A NUMBER. JUST A NUMBER, NO OTHER WORDS, NOTHING␣
↪ELSE.

  Essay set: "{essay_set}"

  {essay}

  """

  response = openai.Completion.create(
      engine="text-davinci-002",
      prompt=prompt,
      temperature=0.5,
      max_tokens=10
  )

  output = response.choices[0].text.strip()
```

```python
    return output # Convert the grade from string to integer
```

[ ]:
```python
from tqdm import tqdm

def generate_gpt3_predictions(df):
    gpt3_predictions = []
    for _, row in tqdm(df.iterrows(), total=df.shape[0]):
        essay = row['essay']
        essay_set = row['essay_set']
        if essay_set == 1:
          grade = grade_essay_one_gpt3(essay, essay_set)
        if essay_set == 2:
          grade = grade_essay_two_gpt3(essay, essay_set)
        if essay_set == 3 or essay_set == 4 :
          grade = grade_essay_three_four_gpt3(essay, essay_set)
        if essay_set == 5 or essay_set == 6:
          grade = grade_essay_five_six_gpt3(essay, essay_set)
        if essay_set == 7:
          grade = grade_essay_seven_gpt3(essay, essay_set)
        if essay_set == 8:
          grade = grade_essay_eight_gpt3(essay, essay_set)

        gpt3_predictions.append(grade)
    return gpt3_predictions
```

[ ]:
```python
gpt3_predictions = generate_gpt3_predictions(smaller_dataset)
```

[ ]:
```python
smaller_dataset
```

[ ]:
```python
smaller_dataset.to_csv('smaller_dataset.csv', index=False)
```

[ ]:
```python
with open("list.txt", "w") as file:
    for item in gpt3_predictions:
        file.write(str(item) + "\n")
```

[ ]:
```python
import pandas as pd
from sklearn.model_selection import train_test_split

# Specify the column for which you want to maintain the distribution
target_column = "essay_set"

# Set the desired size of the smaller dataset
desired_dataset_size = 1000

# Perform stratified sampling to create the smaller dataset
smaller_dataset = train_df.groupby(target_column, group_keys=False).apply(
    lambda x: x.sample(int(desired_dataset_size / len(train_df) * len(x)))
```

```
)

# Reset the index of the smaller dataset
smaller_dataset.reset_index(drop=True, inplace=True)

# Optional: Save the smaller dataset to a new CSV file
smaller_dataset.to_csv("smaller_dataset.csv", index=False)
```

[ ]:
```
len(gpt3_predictions)
```

[ ]:
```
gpt3_predictions[:5]
```

[ ]:
```
```

[ ]:
```
y_true.head()
```

[ ]:
```
# 'grade' and 'Predicted Grade' are column names containing actual and␣
↪predicted grades
y_true = test_df['grade']
y_pred_gpt3 = gpt3_predictions  # round the predictions to make them discrete

qwk_albert = cohen_kappa_score(y_true, y_pred_gpt3, weights='quadratic')

print(f"Quadratic Weighted Kappa (QWK) is: {qwk_albert}")
```

6.1 Ensemble

[ ]:
```
from sklearn.linear_model import LinearRegression

# Assuming the grades from your models are contained in a list of lists where
# each list contains the grades from a single model
grades = [[...], [...], [...], [...], [...]]  # BERT, ALBERT, ELECTRA, XLNet,␣
↪GPT-3

# Convert grades to features and targets
X = list(zip(*grades))  # features are the grades from all models
y = [...]  # target is the actual grades

# Initialize Linear Regression model
regressor = LinearRegression()

# Fit the model
regressor.fit(X, y)
```

[ ]:
```
# Step 1: Load the tokenizer
```

```
tokenizer = BertTokenizer.from_pretrained("path/to/spiece_model", "path/to/
↪special_tokens_map.json")

# Step 2: Load the model architecture
config = BertConfig.from_json_file("path/to/config.json")
model = BertModel(config)

# Step 3: Load the model weights
model.load_state_dict(torch.load("path/to/pytorch_model.bin"))
```

```
[ ]: from transformers import ElectraTokenizer, ElectraConfig, ElectraModel

# Step 1: Load the tokenizer
tokenizer = ElectraTokenizer.from_pretrained("/content/drive/MyDrive/model/
↪electra_/vocab.txt", "/content/drive/MyDrive/model/electra_/
↪special_tokens_map.json")

# Step 2: Load the model architecture
config = ElectraConfig.from_json_file("/content/drive/MyDrive/model/electra_/
↪config.json")
model_electra = ElectraModel(config)

# Step 3: Load the model weights
model_electra.load_state_dict(torch.load("/content/drive/MyDrive/model/electra_/
↪pytorch_model.bin"))
```

```
[ ]: from transformers import AlbertTokenizer, AlbertConfig, XLNetModel

# Step 1: Load the tokenizer
tokenizer = AlbertTokenizer.from_pretrained("/content/drive/MyDrive/model/
↪albert/spiece.model", "/content/drive/MyDrive/model/albert/
↪special_tokens_map.json")
# Step 2: Load the model architecture
config = AlbertConfig.from_json_file("/content/drive/MyDrive/model/albert/
↪config.json")
model_albert = AlbertModel(config)

# Step 3: Load the model weights
model_albert.load_state_dict(torch.load("/content/drive/MyDrive/model/albert/
↪pytorch_model.bin"))
```