Implementation Part Presentation of chosen theses for CSE 596 course.
Part 2(Model analysis)

(UCC Dataset)

Theses- Using PLM (Pretrained Language Models) for Toxic Comment Classification

Group 5

Members-

Md. Sayadul Hoque (2235106650)

Moinul Hasan Sifat (2235015650)

Bristy Cathrin (2235366650)

Muntaka Sayef (2235082650)

Import Part:

```
1 !nvidia-smi
```

- The "!nvidia-smi" command is used in a Jupyter Notebook or command line to check the status of the NVIDIA GPU devices installed on the system. It stands for "NVIDIA System Management Interface" and provides information on the GPU usage, memory usage, and temperature of the GPU devices.

```
1 %%capture
2 !pip install transformers
3 !pip install pytorch-lightning
```

- The "%%capture" command is a Jupyter Notebook magic command that allows you to capture the output of a cell in a notebook.
- "!pip install transformers", installs the "transformers" package, which is a popular library for Natural Language Processing (NLP) tasks such as text classification, sentiment analysis, and language translation. The library is built on top of PyTorch and provides pre-trained models for various NLP tasks.
- "!pip install pytorch-lightning", installs the "PyTorch Lightning" package, which is a lightweight PyTorch wrapper for high-performance deep learning research. PyTorch Lightning provides a higher-level interface for organizing and structuring PyTorch code, making it easier to write complex models and experiments. It also provides useful features like automatic checkpointing, distributed training, and mixed-precision training.

```
1 import os
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 from google.colab import drive
```

- The "os" library provides a way to interact with the operating system, such as accessing files and directories.
- The "numpy" library is a popular library for numerical computing in Python, providing support for arrays, matrices, and various mathematical functions.

- The "pandas" library is a library for data manipulation and analysis. It provides data structures like DataFrame and Series, which are widely used in data analysis tasks.
- The "matplotlib" library is a plotting library in Python, which allows you to create various types of visualizations, such as line plots, scatter plots, and histograms.
- The "google.colab" library is a library specific to Google Colaboratory, which provides access to the Google Drive API for accessing files and folders on Google Drive.

```
1 train_data = pd.read_csv(train_path)

1 train_data.head(5)
```

- First line is a Python code snippet that reads a CSV file into a pandas DataFrame object.
- The "pd" module is an alias for the pandas library, which provides functions and data structures for data analysis and manipulation.
- The "read_csv()" function is a pandas function that reads a CSV (Comma Separated Value) file and converts it into a DataFrame object. The function takes the file path as input and returns a DataFrame object that contains the data from the CSV file.
- In this code snippet, the "train_path" variable is assumed to contain the file path to a CSV file containing training data, and the "train_data" variable is assigned the resulting DataFrame object. This allows you to manipulate and analyze the training data using the pandas library's built-in functions and methods.
- Second line is a Python code snippet that displays the first five rows of a pandas DataFrame object.
- The "head()" method is a pandas DataFrame method that returns the first n rows of the DataFrame, where n is the number specified in the method's argument. By default, if no argument is specified, the method returns the first five rows of the DataFrame.

Inspect data Part:

```
1 train_data['unhealthy'] = np.where(train_data['healthy'] == 1, 0, 1)
2
3 attributes = ['antagonize', 'condescending','dismissive','generalisation',
4               'generalisation_unfair','hostile','sarcastic','unhealthy']
5
6 train_data[attributes].sum().plot.bar()
```

- First line is a Python code snippet that creates a new column in a pandas DataFrame object based on the values in an existing column.
- The output of the "where()" function is assigned to a new column in the "train_data" DataFrame called "unhealthy". This means that the "unhealthy" column will have a value of 0 for rows where the "healthy" column is equal to 1, and a value of 1 for rows where the "healthy" column is not equal to 1.
- Second line is a Python code snippet that creates a list of strings called "attributes". Each string in the list represents a column name in a pandas DataFrame object.
- Third line is a Python code snippet that generates a bar chart using the matplotlib library.
- The ".sum()" method is used on the "train_data[attributes]" DataFrame object to calculate the sum of each column specified in the "attributes" list.
- The ".plot.bar()" method is then called on the resulting Series object to generate a bar chart of the sums. This creates a bar chart where each bar represents the sum of a column in the DataFrame. The x-axis of the chart shows the column names specified in the "attributes" list, and the y-axis shows the sum of each column.
- Overall, this code snippet is useful for quickly visualizing the distribution of the data across the different columns specified in the "attributes" list. It can help to identify any columns that may have an unusually high or low sum, which could be an indication that further investigation is needed.

Dataset Part:

```
1 from torch.utils.data import Dataset
```

- This is a Python code snippet that imports the "Dataset" class from the "torch.utils.data" module.
- The "torch.utils.data" module is a part of PyTorch's data loading utilities, which is used to load and preprocess the data for deep learning models.
- The "Dataset" class is a PyTorch class that represents a dataset. In particular, it is an abstract class that defines the minimum requirements for any PyTorch dataset. It provides an interface for accessing the samples in a dataset, and allows users to customize how data is loaded and processed.
- When building a custom dataset for a PyTorch model, it is common to subclass the "Dataset" class and implement the necessary methods for loading and processing the data. The resulting subclass can then be used with PyTorch's data loading utilities to load and preprocess the data.

```
1 class UCC_Dataset(Dataset):
2
3   def __init__(self, data_path, tokenizer, attributes, max_token_len: int = 128, sample = 5000):
4     self.data_path = data_path
5     self.tokenizer = tokenizer
6     self.attributes = attributes
7     self.max_token_len = max_token_len
8     self.sample = sample
9     self._prepare_data()
10
```

- This is a Python code snippet that defines a custom dataset class called "UCC_Dataset". The "UCC" acronym likely stands for some domain-specific name or application.
- The "Dataset" class was imported from the "torch.utils.data" module in the previous code snippet.
- This custom dataset class is defined by subclassing the "Dataset" class and implementing the necessary methods for loading and processing the data.
- The "init()" method is the constructor method for a class, and it is called when an object of the class is created. In this case, the "init()" method takes several arguments:

    - "data_path": the file path to the dataset.
    - "tokenizer": an object that will be used to tokenize the text data.
    - "attributes": a list of strings representing the attribute names that will be used for classification.
    - "max_token_len": an integer representing the maximum length of the tokenized input.
    - "sample": an integer representing the number of samples to include in the dataset.
    - The method initializes these attributes as instance variables, and then calls the "_prepare_data()" method to load and preprocess the data.
- Overall, this "init()" method is used to configure the dataset and load the data from the specified file path using the given tokenizer. It also sets the maximum length of the tokenized input and the number of samples to include in the dataset.

```
def _prepare_data(self):
  data = pd.read_csv(self.data_path)
  data['unhealthy'] = np.where(data['healthy'] == 1, 0, 1)
  if self.sample is not None:
    unhealthy = data.loc[data[attributes].sum(axis=1) > 0]
    clean = data.loc[data[attributes].sum(axis=1) == 0]
    self.data = pd.concat([unhealthy, clean.sample(self.sample, random_state=7)])
  else:
    self.data = data

def __len__(self):
  return len(self.data)
```

- The "_prepare_data()" method is a helper method that is called by the "init()" method to load and preprocess the data.
- In this case, the "_prepare_data()" method first loads the data from the specified file path using the Pandas "read_csv()" method. It then creates a new binary column "unhealthy" which is set to 1 if any of the attributes are true (i.e., if the comment is classified as unhealthy).
- If the "sample" attribute is not None, the method then splits the data into two parts: unhealthy and clean comments, based on whether they contain any unhealthy attributes. It then samples a specified number of clean

comments (self.sample) and concatenates them with the unhealthy comments to create a balanced dataset. The random_state parameter is set to ensure reproducibility of results.

- If the "sample" attribute is None, then the method simply sets the "data" attribute to the loaded and preprocessed data.
- Overall, this "_prepare_data()" method is used to load and preprocess the data in a way that is customized to the needs of the "UCC_Dataset" class. It creates a binary column to label each comment as unhealthy or not, and it samples the data to ensure a balanced dataset if specified.
- The __len__() method is defined for a custom class, and it returns the length of the data attribute of the class instance. When the built-in len() function is called on an instance of this class, it will return the length of the data attribute.

```
def __getitem__(self, index):
  item = self.data.iloc[index]
  comment = str(item.comment)
  attributes = torch.FloatTensor(item[self.attributes])
  tokens = self.tokenizer.encode_plus(comment,
                                      add_special_tokens=True,
                                      return_tensors='pt',
                                      truncation=True,
                                      padding='max_length',
                                      max_length=self.max_token_len,
                                      return_attention_mask = True)
  return {'input_ids': tokens.input_ids.flatten(), 'attention_mask': tokens.attention_mask.flatten(), 'labels': attributes}
```

- "getitem" method of the "UCC_Dataset" class is a special method that allows you to access individual items in the dataset by index.
- When called with an index, this method retrieves the corresponding row of data from the "self.data" DataFrame using the ".iloc[]" method.
- Then, the comment text is extracted from the "comment" column of the row as a string. The "attributes" variable is created by selecting the columns of the row corresponding to the attributes in the "attributes" list, and converting them to a PyTorch FloatTensor using the "torch.FloatTensor()" method.

- Next, the comment text is tokenized using the "self.tokenizer" object. The "encode_plus()" method is used to tokenize the comment and convert it to a PyTorch tensor that can be passed to the model. The resulting tensor contains the input IDs, attention mask, and token type IDs of the tokens.
- The input IDs are flattened to a one-dimensional tensor using the "flatten()" method, which is required by some PyTorch models. The attention mask tensor is also flattened, and the labels tensor is returned as is.
- Finally, the method returns a Python dictionary containing the input IDs, attention mask, and labels tensors as values, and the keys "input_ids", "attention_mask", and "labels", respectively. This dictionary represents a single data point in the dataset.

```
1 import torch
2 from transformers import AutoTokenizer
3 model_name = 'bert-base-uncased'
4 tokenizer = AutoTokenizer.from_pretrained(model_name)
5 ucc_ds = UCC_Dataset(train_path, tokenizer, attributes=attributes)
6 ucc_ds_val = UCC_Dataset(val_path, tokenizer, attributes=attributes, sample=None)
```

- “import torch”: Imports the PyTorch library.
- “from transformers import AutoTokenizer”: Imports the AutoTokenizer class from the transformers library, which is used to tokenize input text for NLP models.
- “model_name = 'bert-base-uncased'”: Sets the name of the pre-trained BERT model to be used as bert-base-uncased.
- “tokenizer = AutoTokenizer.from_pretrained(model_name)”: Initializes the AutoTokenizer class with the specified pre-trained BERT model, which is loaded from the Hugging Face model hub.
- “ucc_ds = UCC_Dataset(train_path, tokenizer, attributes=attributes)”: Creates a training dataset object using the UCC_Dataset class, passing the file path to the training data, the initialized tokenizer object, and the list of attribute names as arguments.
- “ucc_ds_val = UCC_Dataset(val_path, tokenizer, attributes=attributes, sample=None)”: Creates a validation dataset object using the UCC_Dataset class, passing the file path to the validation data, the initialized tokenizer

object, the list of attribute names, and the sample parameter set to None as arguments.

```
1 import pytorch_lightning as pl
2 from torch.utils.data import DataLoader
```

These lines of code import two important modules in PyTorch:

- pytorch_lightning: a lightweight PyTorch wrapper for high-performance training, which provides several utilities for training and helps in writing cleaner and more concise code.
- DataLoader class from torch.utils.data module, which provides an efficient way to load and iterate over data in batches during model training.

These modules are commonly used for building deep learning models using PyTorch, especially when working with large datasets.

Data Module Part:

```
1 class UCC_Data_Module(pl.LightningDataModule):
```

“UCC_Data_Module” is a class that inherits from “pl.LightningDataModule” in the PyTorch Lightning library, and it defines the data processing pipelines used for training, validation, and testing.

```
def __init__(self, train_path, val_path, attributes, batch_size: int = 16, max_token_length: int = 128,  model_name='bert-base-uncased'):
  super().__init__()
  self.train_path = train_path
  self.val_path = val_path
  self.attributes = attributes
  self.batch_size = batch_size
  self.max_token_length = max_token_length
  self.model_name = model_name
  self.tokenizer = AutoTokenizer.from_pretrained(model_name)
```

The constructor of this class takes several arguments:

- train_path: the file path to the training data
- val_path: the file path to the validation data

- attributes: a list of attribute names
- batch_size: the batch size to use for training and validation data loaders
- max_token_length: the maximum length of a token sequence after tokenization
- model_name: the name of the pre-trained transformer model to use for tokenization

The constructor initializes these arguments as instance variables, and also initializes a tokenizer instance using the specified pre-trained model.

```
def setup(self, stage = None):
  if stage in (None, "fit"):
    self.train_dataset = UCC_Dataset(self.train_path, attributes=self.attributes, tokenizer=self.tokenizer)
    self.val_dataset = UCC_Dataset(self.val_path, attributes=self.attributes, tokenizer=self.tokenizer, sample=None)
  if stage == 'predict':
    self.val_dataset = UCC_Dataset(self.val_path, attributes=self.attributes, tokenizer=self.tokenizer, sample=None)
```

- The setup() method is defined for the UCC_Data_Module class. It has one argument called stage, which is set to None by default.
- If stage is None or "fit", the method instantiates train_dataset and val_dataset objects as instances of the UCC_Dataset class. The train_dataset object is created using the train_path parameter, while the val_dataset object is created using the val_path parameter. Both objects are passed the attributes, tokenizer, and sample parameters. The sample parameter is set to None for the val_dataset object.
- If stage is "predict", only the val_dataset object is instantiated. The sample parameter is set to None, and the other parameters are the same as for the train_dataset and val_dataset objects created in the previous if statement.

```
def setup(self, stage = None):
  if stage in (None, "fit"):
    self.train_dataset = UCC_Dataset(self.train_path, attributes=self.attributes, tokenizer=self.tokenizer)
    self.val_dataset = UCC_Dataset(self.val_path, attributes=self.attributes, tokenizer=self.tokenizer, sample=None)
  if stage == 'predict':
    self.val_dataset = UCC_Dataset(self.val_path, attributes=self.attributes, tokenizer=self.tokenizer, sample=None)
```

- The train_dataloader method returns a data loader for the training dataset.
- The val_dataloader method returns a data loader for the validation dataset.
- The predict_dataloader method returns a data loader for the validation dataset for prediction.

Model Part:

```
1 from transformers import AutoModel, AdamW, get_cosine_schedule_with_warmup
2 import torch.nn as nn
3 import math
4 from torchmetrics.functional.classification import auroc
5 import torch.nn.functional as F
```

- “from transformers import AutoModel, AdamW, get_cosine_schedule_with_warmup” imports three different modules from the Transformers library.
    - “AutoModel” is a module that can be used to automatically load any of the pre-trained transformer models available in the library. It simplifies the process of using pre-trained models by automatically downloading and loading the required tokenizer and model weights.
    - “AdamW” is a variant of the Adam optimizer that is commonly used for training deep learning models. It stands for "Adaptive Moment Estimation with Weight Decay". AdamW differs from

regular Adam by adding weight decay to the update rule for the model weights.

- “get_cosine_schedule_with_warmup” is a function that returns a learning rate schedule. The learning rate is increased linearly during a warmup period, and then decreases according to a cosine function over the remaining epochs of training. This learning rate schedule is commonly used in training transformer models.

- import torch.nn as nn: Importing PyTorch's nn module, which provides support for creating neural network components.
- import math: Importing the math module which provides mathematical functions.
- from torchmetrics.functional.classification import auroc: Importing the auroc metric from PyTorch metrics to measure the performance of the model.
- import torch.nn.functional as F: Importing PyTorch's functional module, which provides functions to apply non-linear operations on activations.

```
1 class UCC_Comment_Classifier(pl.LightningModule):
```

- This is a Python class definition that extends the pl.LightningModule class from the PyTorch Lightning framework. It defines a neural network model that can be trained to classify text comments into multiple categories.
- The UCC_Comment_Classifier class is intended to be used as a base class for creating specific comment classification models. It contains methods that define the structure of the model, specify the training and validation steps, and provide hooks for monitoring the training process.

```
def __init__(self, config: dict):
  super().__init__()
  self.config = config
  self.pretrained_model = AutoModel.from_pretrained(config['model_name'], return_dict = True)
  self.hidden = torch.nn.Linear(self.pretrained_model.config.hidden_size, self.pretrained_model.config.hidden_size)
  self.classifier = torch.nn.Linear(self.pretrained_model.config.hidden_size, self.config['n_labels'])
  torch.nn.init.xavier_uniform_(self.classifier.weight)
  self.loss_func = nn.BCEWithLogitsLoss(reduction='mean')
  self.dropout = nn.Dropout()
```

- This code defines a class constructor __init__ for a neural network model. The class inherits from the torch.nn.Module class. The constructor takes a dictionary config as an input. The AutoModel.from_pretrained() function from the transformers library is used to load a pre-trained model specified in the model_name key of the config dictionary. The return_dict argument is set to True which specifies that the forward method of the model will return a dictionary containing model outputs.
- A linear layer is defined with torch.nn.Linear which will take the output of the pre-trained model as an input and transform it to a hidden size specified in the config dictionary. Another linear layer is defined which will take the output of the first linear layer and transform it to the output size specified in the n_labels key of the config dictionary.
- The weight of the classifier linear layer is initialized using the xavier_uniform_ initialization method.
- A binary cross-entropy loss function with logits is defined using nn.BCEWithLogitsLoss with the reduction method set to mean.
- A dropout layer is defined using nn.Dropout.

```
def forward(self, input_ids, attention_mask, labels=None):
  # roberta layer
  output = self.pretrained_model(input_ids=input_ids, attention_mask=attention_mask)
  pooled_output = torch.mean(output.last_hidden_state, 1)
  # final logits
  pooled_output = self.dropout(pooled_output)
  pooled_output = self.hidden(pooled_output)
  pooled_output = F.relu(pooled_output)
  pooled_output = self.dropout(pooled_output)
  logits = self.classifier(pooled_output)
  # calculate loss
  loss = 0
  if labels is not None:
    loss = self.loss_func(logits.view(-1, self.config['n_labels']), labels.view(-1, self.config['n_labels']))
  return loss, logits
```

- This forward() method is defining the forward pass of the model. It takes input_ids and attention_mask as input and calculates the output logits using the following steps:
    - Passes the inputs through the pretrained_model layer to get the output from the model's last hidden state.
    - Calculates the mean across each feature dimension of the last hidden state.
    - Applies dropout to the pooled output to reduce overfitting.
    - Passes the output of dropout through a hidden linear layer with ReLU activation.
    - Applies dropout again to the output of the hidden layer.
    - Passes the output of the second dropout through a linear layer to get the logits.
    - The loss function used is binary cross-entropy with logits (BCEWithLogitsLoss) which is commonly used for binary classification problems.
- If labels are not provided (i.e. during inference/prediction), the method returns only logits. However, if labels are provided (i.e. during training), the method computes the binary cross-entropy loss between the predicted logits and the true labels. The loss is then returned along with the logits.

```
def training_step(self, batch, batch_index):
  loss, outputs = self(**batch)
  self.log("train loss ", loss, prog_bar = True, logger=True)
  return {"loss":loss, "predictions":outputs, "labels": batch["labels"]}
```

This method defines a training step for the model during the training loop. It takes in a batch of data and the batch index, calculates the loss and predictions by calling the forward method defined earlier with the batch as input, logs the training loss using self.log method with progress bar and logger set to True, and returns a dictionary containing the loss, predictions and labels in the batch.

```
def validation_step(self, batch, batch_index):
  loss, outputs = self(**batch)
  self.log("validation loss ", loss, prog_bar = True, logger=True)
  return {"val_loss": loss, "predictions":outputs, "labels": batch["labels"]}
```

- This function is defining the validation step in a PyTorch Lightning training loop for a UCC (Unified Code Count) Comment Classifier model.
- The function takes in two parameters:
  - batch: a batch of data containing input text sequences and their corresponding labels for the validation set
  - batch_index: the index of the batch in the validation set
- The function calculates the loss and outputs of the model on the given batch using the self(**batch) call, where self refers to the current instance of the UCC_Comment_Classifier class.
- Then, it logs the validation loss using self.log(), which writes the loss value to the logger for later access. The prog_bar argument specifies that the loss value should be displayed in a progress bar during training, while the logger argument specifies that the value should be written to the logger.
- Finally, the function returns a dictionary containing the validation loss, model predictions, and the corresponding labels for the given batch. This information can be used to calculate metrics like accuracy and F1 score during evaluation.

```
def predict_step(self, batch, batch_index):
  loss, outputs = self(**batch)
  return outputs
```

- This function is defining the prediction step in a PyTorch Lightning training loop for a UCC (Unified Code Count) Comment Classifier model.
- The function takes in two parameters:
  - batch: a batch of data containing input text sequences for which the model is to make predictions
  - batch_index: the index of the batch in the dataset
- The function calculates the loss and outputs of the model on the given batch using the self(**batch) call, where self refers to the current instance of the UCC_Comment_Classifier class.

- Then, it returns the model's predictions for the given batch, which can be used to evaluate the model's performance on a test set or to make predictions on new, unseen data.

```
def configure_optimizers(self):
  optimizer = AdamW(self.parameters(), lr=self.config['lr'], weight_decay=self.config['weight_decay'])
  total_steps = self.config['train_size']/self.config['batch_size']
  warmup_steps = math.floor(total_steps * self.config['warmup'])
  warmup_steps = math.floor(total_steps * self.config['warmup'])
  scheduler = get_cosine_schedule_with_warmup(optimizer, warmup_steps, total_steps)
  return [optimizer],[scheduler]
```

This code defines the configure_optimizers() method for the LightningModule subclass. This method is used to configure the optimizer and learning rate scheduler for the model. Here is what each line of the method does:

- optimizer = AdamW(self.parameters(), lr=self.config['lr'], weight_decay=self.config['weight_decay']): This line initializes an AdamW optimizer with the given learning rate and weight decay. self.parameters() is a method that returns an iterator over all the trainable parameters of the model.
- total_steps = self.config['train_size']/self.config['batch_size']: This line calculates the total number of steps that the optimizer will take during the training process, based on the size of the training data and the batch size.
- warmup_steps = math.floor(total_steps * self.config['warmup']): This line calculates the number of warm-up steps for the learning rate scheduler. The warmup parameter is a fraction of the total number of steps, and is specified in the model configuration.
- warmup_steps = math.floor(total_steps * self.config['warmup']): This line is a duplicate of the previous line and can be removed.
- scheduler = get_cosine_schedule_with_warmup(optimizer, warmup_steps, total_steps): This line creates a cosine learning rate scheduler with warm-up. The scheduler will linearly increase the learning rate for the first warmup_steps steps, and then decrease it using a cosine function for the

remaining steps. The scheduler object is returned along with the optimizer object in a list.

- return [optimizer],[scheduler]: This line returns a list of two elements: a list containing the optimizer object, and a list containing the scheduler object.

```
idx=0
input_ids = ucc_ds.__getitem__(idx)['input_ids']
attention_mask = ucc_ds.__getitem__(idx)['attention_mask']
labels = ucc_ds.__getitem__(idx)['labels']
model.cpu()
loss, output = model(input_ids.unsqueeze(dim=0), attention_mask.unsqueeze(dim=0), labels.unsqueeze(dim=0))
print(labels.shape, output.shape, output)
```

- idx=0: Assigning the value 0 to the variable idx.
- input_ids = ucc_ds.__getitem__(idx)['input_ids']: Retrieving the input_ids tensor from the dataset ucc_ds for the index idx.
- attention_mask = ucc_ds.__getitem__(idx)['attention_mask']: Retrieving the attention_mask tensor from the dataset ucc_ds for the index idx.
- labels = ucc_ds.__getitem__(idx)['labels']: Retrieving the labels tensor from the dataset ucc_ds for the index idx.
- model.cpu(): Moving the model to the CPU.
- loss, output = model(input_ids.unsqueeze(dim=0), attention_mask.unsqueeze(dim=0), labels.unsqueeze(dim=0)): Running the model on the input tensors and labels. The unsqueeze method is used to add a batch dimension to the input tensors and labels as the model expects a batch of inputs.
- print(labels.shape, output.shape, output): Printing the shape of labels and output tensors, along with the values in output.

Model Part:

```
# datamodule
ucc_data_module = UCC_Data_Module(train_path, val_path, attributes=attributes, batch_size=config['batch_size'])
ucc_data_module.setup()

# model
model = UCC_Comment_Classifier(config)

# trainer and fit
trainer = pl.Trainer(max_epochs=config['n_epochs'], num_sanity_val_steps=50)
trainer.fit(model, ucc_data_module)
```

- This code sets up a PyTorch Lightning trainer, a UCC_Data_Module containing train and validation data, and a UCC_Comment_Classifier model with specified configuration.
- The trainer is initialized with a maximum number of epochs and a number of validation steps for sanity checks during training. The fit method is then called on the trainer with the model and data module as inputs, which trains the model using the training data and evaluates it on the validation data for each epoch.

“Predict with Model” Part:

```
# method to convert list of comments into predictions for each comment
def classify_raw_comments(model, dm):
  predictions = trainer.predict(model, datamodule=dm)
  flattened_predictions = np.stack([torch.sigmoid(torch.Tensor(p)) for batch in predictions for p in batch])
  return flattened_predictions
```

This function uses a trained model and a datamodule to classify raw comments. It calls the predict method of a pl.Trainer object to get the predictions for the raw comments. It then stacks the predictions obtained from different batches and

applies the sigmoid function to them. Finally, it returns the flattened predictions as a numpy array.

```
val_data = pd.read_csv(val_path)
val_data['unhealthy'] = np.where(val_data['healthy'] == 1, 0, 1)
true_labels = np.array(val_data[attributes])
```

- In these lines of code, we are reading the validation dataset using pandas from the val_path file path. Then we are creating a new column called 'unhealthy' based on the value of the 'healthy' column. If 'healthy' column has the value of 1, then 'unhealthy' column will have the value of 0, otherwise it will have the value of 1.
- Finally, we are creating a numpy array called true_labels using the values of the 'toxicity', 'severe_toxicity', 'obscene', 'threat', 'insult' and 'identity_hate' columns of the validation dataset.

```
from sklearn import metrics
plt.figure(figsize=(15, 8))

for i, attribute in enumerate(attributes):
  fpr, tpr, _ = metrics.roc_curve(
      true_labels[:,i].astype(int), predictions[:, i])
  auc = metrics.roc_auc_score(
      true_labels[:,i].astype(int), predictions[:, i])
  plt.plot(fpr, tpr, label='%s %g' % (attribute, auc))
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.title('BERT Trained on UCC Datatset - AUC ROC')
```

This code is for generating an AUC-ROC (Area Under the Receiver Operating Characteristic Curve) plot using the scikit-learn metrics library and matplotlib for visualization.

- from sklearn import metrics: Importing the metrics module from the scikit-learn library for evaluation metrics.
- plt.figure(figsize=(15, 8)): Create a new matplotlib figure with a size of 15 by 8 inches.
- for i, attribute in enumerate(attributes):: Looping through the list of attributes and using enumerate() function to get the index and the attribute name.
- fpr, tpr, _ = metrics.roc_curve(true_labels[:,i].astype(int), predictions[:, i]): Generating the false positive rate (fpr), true positive rate (tpr) and thresholds for the i-th attribute using the roc_curve() function from metrics module. The true_labels[:,i] is used as the true label and predictions[:, i] as the predicted probabilities.
- auc = metrics.roc_auc_score(true_labels[:,i].astype(int), predictions[:, i]): Calculate the area under the ROC curve using the roc_auc_score() function from metrics module.
- plt.plot(fpr, tpr, label='%s %g' % (attribute, auc)): Plot the ROC curve for the i-th attribute using fpr and tpr. The label parameter of plt.plot() function is set to display the attribute name and its AUC value on the legend.
- plt.xlabel('False Positive Rate'): Set the X-axis label to "False Positive Rate".
- plt.ylabel('True Positive Rate'): Set the Y-axis label to "True Positive Rate".
- plt.legend(loc='lower right'): Display a legend on the plot with the position set to lower right.
- plt.title('BERT Trained on UCC Datatset - AUC ROC'): Set the title of the plot to "BERT Trained on UCC Datatset - AUC ROC".
