Project Report

The Pretrained Model And Dataset:

The model we used is “xlm-r-large-arabic-toxic”, which is a model used to classify Arabic sentences to toxic or non-toxic, the model was trained on huge dataset of tweets, we trained the model on dataset of company reviews that has two classes: 0 (positive) and 1 (negative), the company reviews have three features: “review_description”, “rating” and “company”

Data Preprocessing And Cleaning:

```
import pandas as pd

# Load the CSV file into a Pandas DataFrame
df = pd.read_csv('CompanyReviews.csv')
df = df.dropna()
df.head()

print(df['rating'].unique())
print(df['rating'].value_counts()[0])
# Map the class labels to integers
label_map = {1: 0, -1: 1}
df['rating'] = df['rating'].map(label_map)
df.head()
```

We first import the dataset, and get rid of the null values, and also remove the neutral class, so we can have only two classes (positive and negative), because the pretrained model deals with two target classes.

```
from transformers import InputExample, InputFeatures

input_texts = df['review_description'].astype(str).values.tolist()

# Tokenize the input sequences using the tokenizer you loaded
encoded_data = tokenizer.batch_encode_plus(
    input_texts,
    padding='max_length',
    truncation=True,
    max_length=128,
    return_tensors='pt')
```

the `'review_description'` is going to be the input text, and converted to a Python list using the `astype(str).values.tolist()` method.

The `tokenizer.batch_encode_plus()` method is then called to tokenize the input sequences. This method tokenizes multiple sequences simultaneously, which can be more efficient than tokenizing them one by one. It takes the following parameters:

- `input_texts`: The list of input sequences to be tokenized.
- `padding='max_length'`: This parameter specifies that the sequences should be padded to the maximum length of 128 tokens. If a sequence is shorter than 128 tokens, it will be padded with special tokens to reach the maximum length.
- `truncation=True`: If a sequence is longer than 128 tokens, it will be truncated to fit the maximum length.
- `max_length=128`: The maximum length of the tokenized sequences.
- `return_tensors='pt'`: This parameter specifies that the output should be returned as PyTorch tensors.

The encoded_data variable will store the output of the `tokenizer.batch_encode_plus()` method. It will be a dictionary-like object that contains the tokenized input sequences as tensors.

```
# Split the data into training, validation, and test sets
train_size = int(0.9 * len(df))
val_size = int(0.05 * len(df))
test_size = len(df) - train_size - val_size
```

Then we define the split ratios, the `test_size` is the difference between `train_size` and `val_size`, then we use the tokenized test and the aforementioned ratios to create `train_data`, `val_data`, and `test_data` are lists of dictionaries, where each dictionary contains the input sequence's tokenized representation and its corresponding label:

```
train_data = [{'input': {'input_ids': input_ids, 'attention_mask':
attention_mask}, 'label': labels}
              for input_ids, attention_mask, labels in
zip(encoded_data['input_ids'][:train_size],

encoded_data['attention_mask'][:train_size],

df['rating'][:train_size])]
val_data = [{'input': {'input_ids': input_ids, 'attention_mask':
attention_mask}, 'label': label}            for input_ids, attention_mask,
label in zip(encoded_data['input_ids'][train_size:train_size+val_size],

encoded_data['attention_mask'][train_size:train_size+val_size],
```

```
df['rating'][train_size:train_size+val_size])]
test_data = [{'input': {'input_ids': input_ids, 'attention_mask':
attention_mask}, 'label': label}             for input_ids, attention_mask,
label in zip(encoded_data['input_ids'][-test_size:],

encoded_data['attention_mask'][-test_size:],

df['rating'][-test_size:])]
```

Model Preperation And Training:

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification

tokenizer =
AutoTokenizer.from_pretrained("akhooli/xlm-r-large-arabic-toxic")

model =
AutoModelForSequenceClassification.from_pretrained("akhooli/xlm-r-large-ara
bic-toxic")
```

Firstly, we load the pre-trained model and the pre-trained tokenizer
The tokenizer object represents the loaded tokenizer. It provides methods for tokenizing and encoding text, as well as handling special tokens and padding.

The model object represents the loaded model for sequence classification. It contains the pre-trained weights and architecture for the model and provides methods for performing inference on input sequences.

```
# Define your model architecture
class MyModel(nn.Module):
    def __init__(self, num_labels):
        super().__init__()
        self.model =
AutoModelForSequenceClassification.from_pretrained("akhooli/xlm-r-large-ara
bic-toxic", num_labels=num_labels)

    def forward(self, input_ids, attention_mask):
        outputs = self.model(input_ids=input_ids,
attention_mask=attention_mask)
        logits = outputs.logits
        return logits

num_labels = 2  # Specify the number of labels
model = MyModel(num_labels)
```

Here we define the model architecture, by loading a pre-trained model for sequence classification using `AutoModelForSequenceClassification.from_pretrained()`, The forward method is also defined within the `MyModel` class. This method specifies the forward pass of the model, which describes how input data flows through the model layers. In this case, the forward method takes `input_ids` and `attention_mask` as inputs, which represent the tokenized input sequence and its attention mask, respectively. Finally, the logits (output scores before applying a softmax activation) are extracted from the outputs using `outputs.logits` and returned from the forward method.

Then we define that our model only deals with two classification tasks.

```
# Define the device
device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
model.to(device)

# Define the optimizer and scheduler
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-7)
num_epochs = 1
batch_size = 8
num_train_steps = len(train_data) * num_epochs // batch_size
num_warmup_steps = num_train_steps // 10
scheduler = get_linear_schedule_with_warmup(optimizer,
num_warmup_steps, num_train_steps)
num_batches = len(train_data) // batch_size
print("Number of batches:", num_batches)
```

Here we check if CUDA-enabled GPU is available, if it's not available, we resort to using CPU, then we define the optimizer (AdamW in this case), and the hyperparameters:

- Learning rate.
- Number of epochs.
- Batch size.

Then we define a learning rate scheduler is a technique used in machine learning to adjust the learning rate during the training process, it starts with a warm-up phase, gradually increasing the learning rate during the warm-up steps, and then decaying it linearly over the remaining steps.

```
# Define the evaluation function
def eval_step(model, dataloader):
    model.eval()
    correct = 0
```

```
    total = 0
    with torch.no_grad():
        for batch in dataloader:
            input_ids = batch['input']['input_ids'].to(device)
            attention_mask = batch['input']['attention_mask'].to(device)
            labels = batch['label'].to(device).long()

            outputs = model(input_ids=input_ids,
attention_mask=attention_mask)
            predictions = torch.argmax(outputs, dim=1)
            correct += (predictions == labels).sum().item()
            total += labels.size(0)

    accuracy = correct / total
    return accuracy
```

Here we define an evaluation function that iterates over the batches (but firstly we disable gradient computation to save more resources), and every iteration obtains the output logits, and then computes the predicted label with the highest probability, then it computes the number of correct predictions, by comparing the true labels with the predicted ones.

```
# Training loop
for epoch in range(num_epochs):
    train_dataloader = DataLoader(train_data, batch_size=batch_size,
shuffle=True)
    val_dataloader = DataLoader(val_data, batch_size=batch_size,
shuffle=False)

    model.train()
    for i, batch in enumerate(train_dataloader):
        input_ids = batch['input']['input_ids'].to(device)
        attention_mask = batch['input']['attention_mask'].to(device)
        labels = batch['label'].to(device).long()

        optimizer.zero_grad()
        outputs = model(input_ids=input_ids,
attention_mask=attention_mask)
        loss = nn.CrossEntropyLoss()(outputs, labels)
        loss.backward()
        optimizer.step()
        scheduler.step()

        if i % 100 == 0:
            train_acc = eval_step(model, train_dataloader)
            val_acc = eval_step(model, val_dataloader)
```

```
            print(f'Epoch: {epoch+1}, Batch:
{i+1}/{len(train_dataloader)}, Train Acc: {train_acc:.3f}, Val Acc:
{val_acc:.3f}')

    # Evaluate model on validation data after each epoch
    val_acc = eval_step(model, val_dataloader)
    print(f'Epoch: {epoch+1}, Val Acc: {val_acc:.3f}')
```

Then we define the training loop, in which we load the training and validation sets, each iteration (epoch) goes through the batches, to calculate the output (forward pass), and the loss (backward pass) and then updates the model parameters using the computed gradients and the chosen optimization algorithm (AdamW in this case), `scheduler.step()` updates the learning rate of the optimizer according to the specified learning rate schedule. This helps to adjust the learning rate during training.

Then it calculates the training and validation accuracy by calling the `eval_step` function defined earlier.

```
# Evaluate model on the test set
test_dataloader = DataLoader(test_data, batch_size=batch_size,
shuffle=False)
test_acc = eval_step(model, test_dataloader)
print(f'Test Acc: {test_acc:.3f}')
```

Finally, we evaluate the model on the test set, and calculate the test accuracy.

Results

Before fine-tuning, the test accuracy was %62.9, however after manipulating the hyperparameters and split ratios, the test accuracy improved considerably in most cases:

Hyperparameters	Split Ratios	Test Accuracy
learning rate =5e-4 num_epochs = 3 batch_size = 100	train_size = 80% val_size = 10% test_size = 10%	49%
learning rate =1e-4 num_epochs = 3 batch_size = 64	train_size = 80% val_size = 10% test_size = 10%	49%

learning rate =1e-5 num_epochs = 3 batch_size = 64	train_size = 80% val_size = 10% test_size = 10%	%86.5
learning rate =0.01 num_epochs = 3 batch_size = 100	train_size = 80% val_size = 10% test_size = 10%	%49
learning rate =1e-5 num_epochs = 3 batch_size = 64	train_size = 85% val_size = 5% test_size = 5%	%87.3
learning rate =1e-5 num_epochs = 6 batch_size = 64	train_size = 80% val_size = 10% test_size = 10%	%85.9
learning rate =1e-5 num_epochs = 6 batch_size = 32	train_size = 80% val_size = 10% test_size = 10%	%85.9
earning rate =1e-5 num_epochs = 10 batch_size = 64	train_size = 90% val_size = 5% test_size = 5%	%87.1
earning rate =1e-5 num_epochs = 3 batch_size = 64	**train_size = 90% val_size = 5% test_size = 5%**	**%88.5 (best result)**

Conclusion:

A small learning rate like 1e-5 can lead to better generalization of the data, but slower convergence, as for the batch size, we chose 64 because it is the highest batch size that can fall under the rule of 2^n that we can use given the hardware capabilities, we resorted to change the split ratios by increasing the training size, which allowed the model to learn from a substantial amount of data, which can be beneficial for achieving good performance.

After we got those results, we wanted to implement dropout regularization, but we ran out of compute units, and resources, we subscribed four times to get those results, and we can not afford to subscribe more, so implementing dropout regularization can be considered a future work, however, the modifications needed are as follows:

```
class MyModel(nn.Module):
    def __init__(self, num_labels, dropout_rate=0.1):
        super().__init__()
        self.model = model  # Use the pre-trained model from transformers
```

```
        self.dropout = nn.Dropout(dropout_rate)
        self.classifier = nn.Linear(self.model.config.hidden_size,
num_labels)

    def forward(self, input_ids, attention_mask):
        outputs = self.model(input_ids=input_ids,
attention_mask=attention_mask)
        pooled_output = outputs.pooler_output
        pooled_output = self.dropout(pooled_output)  # Apply dropout
        logits = self.classifier(pooled_output)
        return logits

num_labels = 2  # Specify the number of labels
dropout_rate = 0.1  # Set the dropout rate (adjust as needed)
model = MyModel(num_labels, dropout_rate)
```

And then we just manipulate the dropout rate as a hyperparameter.