Name: Ngô Lê Ngọc Quý . **Student ID:** 21040003

Course: Artificial Intelligence

Lecturer: Trần Thế Vũ

CHALLENGE 3 REPORT

I. Overview

The primary objective of this challenge is to create a robust system dedicated to the identification and prevention of violent, offensive, and harmful language on the internet. The project consists of four main phases: data collection and preprocessing, model building and training, prediction and deployment.

Github repository of the challenge: [https://github.com/ngocquyxs/-AI-Challenge-3.git]

II. Phases

1. Data Collection and Preprocessing

This code loads a dataset, explores its structure, visualizes the distribution of labels, and then splits it into training and testing sets for further machine learning model development.

```
# Load dataset

%cd /content/drive/MyDrive/[AI] Ngoc Quy/Bully

data = pd.read_csv('data/data.csv')

data = data.iloc[:, 1:]

print(data.info())

data.head()

# Visualize the distribution of labels

sns.countplot(x='label', data=data)

# Create training dataset

train_df = data.iloc[:int(len(data)* 0.8)]

train_df.info()

# Create testing dataset

test_df = data.iloc[len(train_df):]

test_df.info()
```

2. Model Building and Training

This code is setting up stratified k-fold cross-validation for the training dataset (train_df).

```
# "StratifiedKFold" class
skf = StratifiedKFold(n_splits=N_SPLITS)
for fold, (_, val_) in enumerate(skf.split(X=train_df, y=train_df.label)):
    train_df.loc[val_, "kfold"] = fold
```

This code is using the Hugging Face transformers library to load a tokenizer for the PhoBERT (Pre-trained models for Vietnamese) model.

```
# Load a tokenizer for the PhoBERT
tokenizer = AutoTokenizer.from_pretrained("vinai/phobert-base", use_fast=False)
```

This code defines a custom PyTorch dataset class called "SentimentDataset" for sentiment analysis.

```
class SentimentDataset(Dataset):
    # Initialization
    def __init__(self, df, tokenizer, max_len=120):
        self.df = df
        self.max_len = max_len
        self.tokenizer = tokenizer

    # Length Method
    def __len__(self):
        return len(self.df)

    # Get Item Method
    def __getitem__(self, index):
        row = self.df.iloc[index]
```

```
    text, label = self.get_input_data(row)


    encoding = self.tokenizer.encode_plus(
        text,
        truncation=True,
        add_special_tokens=True,
        max_length=self.max_len,
        padding='max_length',
        return_attention_mask=True,
        return_token_type_ids=False,
        return_tensors='pt',
    )


    return {
        'text': text,
        'input_ids': encoding['input_ids'].flatten(),
        'attention_masks': encoding['attention_mask'].flatten(),
        'targets': torch.tensor(label, dtype=torch.long),
    }

# Get Input Data Method
def get_input_data(self, row):
    # Preprocessing: {remove icon, special character, lower}
    text = row['content']
    text = ' '.join(simple_preprocess(text))
    label = row['label']
```

```
    return text, label
```

This code is generating a distribution plot of the token count in sentences to provide insights into the length distribution of the text data.

```
# Distribution of length of Sentence
all_data = train_df.content.tolist() + test_df.content.tolist()
all_data = [' '.join(simple_preprocess(text)) for text in all_data]
encoded_text = [tokenizer.encode(text, add_special_tokens=True) for text in all_data]
token_lens = [len(text) for text in encoded_text]
sns.displot(token_lens)
plt.xlim([0,max(token_lens)])
plt.xlabel('Token Count')
```

This model is designed for sentiment classification tasks, where the PhoBERT model is used as a feature extractor, and a linear layer is employed for class prediction.

```
class SentimentClassifier(nn.Module):
  # Initialization
  def __init__(self, n_classes, device):
    super(SentimentClassifier, self).__init__()
    # Model Architecture
    self.bert = AutoModel.from_pretrained("vinai/phobert-base")
    self.drop = nn.Dropout(p=0.3)
    self.fc = nn.Linear(self.bert.config.hidden_size, n_classes)
    nn.init.normal_(self.fc.weight, std=0.02)
    nn.init.normal_(self.fc.bias, 0)
```

```
    self.device = device


  # Forward Method
  def forward(self, input_ids, attention_mask):
    last_hidden_state, output = self.bert(
      input_ids=input_ids,
      attention_mask=attention_mask,
      return_dict=False
    )


    x = self.drop(output)
    x = self.fc(x)
    return x
```

This code defines two functions: train for training the model and eval for evaluating the model on either the validation or test set.

```
# "train" function
def train(model, criterion, optimizer, train_loader):
  model.train()
  losses = []
  correct = 0


  for data in train_loader:
    input_ids = data['input_ids'].to(device)
    attention_mask = data['attention_masks'].to(device)
    targets = data['targets'].to(device)
```

```
        optimizer.zero_grad()

        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask
        )

        loss = criterion(outputs, targets)
        _, pred = torch.max(outputs, dim=1)

        correct += torch.sum(pred == targets)
        losses.append(loss.item())
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        lr_scheduler.step()

    print(f'Train Accuracy: {correct.double()/len(train_loader.dataset)} Loss: {np.mean(losses)}')

# "eval" function
def eval(test_data = False):
    model.eval()
    losses = []
    correct = 0

    with torch.no_grad():
```

```
    data_loader = test_loader if test_data else valid_loader
    for data in data_loader:
        input_ids = data['input_ids'].to(device)
        attention_mask = data['attention_masks'].to(device)
        targets = data['targets'].to(device)

        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask
        )

        _, pred = torch.max(outputs, dim=1)

        loss = criterion(outputs, targets)
        correct += torch.sum(pred == targets)
        losses.append(loss.item())

if test_data:
    print(f'Test Accuracy: {correct.double()/len(test_loader.dataset)} Loss: {np.mean(losses)}')
    return correct.double()/len(test_loader.dataset)
else:
    print(f'Valid Accuracy: {correct.double()/len(valid_loader.dataset)} Loss: {np.mean(losses)}')
    return correct.double()/len(valid_loader.dataset)
```

This code defines a function prepare_loaders to create PyTorch DataLoader objects for training and validation datasets.

```
# "prepare_loaders" function
def prepare_loaders(df, fold):
    df_train = df[df.kfold != fold].reset_index(drop=True)
    df_valid = df[df.kfold == fold].reset_index(drop=True)

    train_dataset = SentimentDataset(df_train, tokenizer, max_len=120)
    valid_dataset = SentimentDataset(df_valid, tokenizer, max_len=120)

    train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True, num_workers=2)
    valid_loader = DataLoader(valid_dataset, batch_size=16, shuffle=True, num_workers=2)

    return train_loader, valid_loader
```

This code is performing training and evaluation in a cross-validation loop for a sentiment analysis model.

```
# Training and evaluation
for fold in range(skf.n_splits):
    print(f'-----------Fold: {fold+1} ------------------')
    train_loader, valid_loader = prepare_loaders(train_df, fold=fold)
    model = SentimentClassifier(n_classes=7, device = device).to(device)
    criterion = nn.CrossEntropyLoss()
    # Recommendation by BERT: lr: 5e-5, 2e-5, 3e-5
    # Batchsize: 16, 32
    optimizer = AdamW(model.parameters(), lr=2e-5)
```

```
lr_scheduler = get_linear_schedule_with_warmup(
            optimizer,
            num_warmup_steps=0,
            num_training_steps=len(train_loader)*EPOCHS
        )
best_acc = 0
for epoch in range(EPOCHS):
    print(f'Epoch {epoch+1}/{EPOCHS}')
    print('-'*30)

    train(model, criterion, optimizer, train_loader)
    val_acc = eval()

    if val_acc > best_acc:
        torch.save(model.state_dict(), f'phobert_fold{fold+1}.pth')
        best_acc = val_acc
```

This code defines a test function for evaluating the ensemble model on a test dataset.

```
# "test" function
def test(data_loader):
    models = []
    for fold in range(skf.n_splits):
        model = SentimentClassifier(n_classes=7, device = device)
        model.to(device)
        model.load_state_dict(torch.load(f'phobert_fold{fold+1}.pth'))
```

```
    model.eval()
    models.append(model)

texts = []
predicts = []
predict_probs = []
real_values = []

for data in data_loader:
    text = data['text']
    input_ids = data['input_ids'].to(device)
    attention_mask = data['attention_masks'].to(device)
    targets = data['targets'].to(device)

    total_outs = []
    for model in models:
        with torch.no_grad():
            outputs = model(
                input_ids=input_ids,
                attention_mask=attention_mask
            )
            total_outs.append(outputs)

    total_outs = torch.stack(total_outs)
    _, pred = torch.max(total_outs.mean(0), dim=1)
    texts.extend(text)
```

```
        predicts.extend(pred)
        predict_probs.extend(total_outs.mean(0))
        real_values.extend(targets)

    predicts = torch.stack(predicts).cpu()
    predict_probs = torch.stack(predict_probs).cpu()
    real_values = torch.stack(real_values).cpu()
    print(classification_report(real_values, predicts))
    return real_values, predicts
```

This code is testing sentiment analysis model on the test dataset.

```
test_dataset = SentimentDataset(test_df, tokenizer, max_len=50)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=True, num_workers=2)
real_values, predicts = test(test_loader)
```

```
              precision    recall  f1-score   support

           0       0.89      0.89      0.89      1144
           1       0.88      0.89      0.88      1068

    accuracy                           0.89      2212
   macro avg       0.89      0.89      0.89      2212
weighted avg       0.89      0.89      0.89      2212
```

This code is using Seaborn to create a heatmap of the confusion matrix for sentiment analysis model's predictions on the test dataset.

```
class_names = [0, 1]
sns.heatmap(confusion_matrix(real_values, predicts), annot=False, xticklabels =
class_names, yticklabels = class_names)
```

This code is implemented a function "check_wrong" to identify and print some examples where sentiment analysis model made incorrect predictions.

```
# "check_wrong" function
def check_wrong(real_values, predicts):
    wrong_arr = []
    wrong_label = []
    for i in range(len(predicts)):
        if predicts[i] != real_values[i]:
            wrong_arr.append(i)
            wrong_label.append(predicts[i])
    return wrong_arr, wrong_label


for i in range(15):
    print('-'*50)
    wrong_arr, wrong_label = check_wrong(real_values, predicts)
    print(test_df.iloc[wrong_arr[i]].content)
    print(f'Predicted: ({class_names[wrong_label[i]]}) --vs-- Real label: ({class_names[real_values[wrong_arr[i]]]})')
```

This code is implementing an "infer" function for making predictions on a single input text using sentiment analysis model.

```
# "infer" function
def infer(text, tokenizer, model, max_len=120):
    encoded_review = tokenizer.encode_plus(
        text,
        max_length=max_len,
        truncation=True,
```

```
        add_special_tokens=True,
        padding='max_length',
        return_attention_mask=True,
        return_token_type_ids=False,
        return_tensors='pt',
    )

    input_ids = encoded_review['input_ids'].to(device)
    attention_mask = encoded_review['attention_mask'].to(device)

    output = model(input_ids, attention_mask)
    _, y_pred = torch.max(output, dim=1)

    print(f'Text: {text}')
    print(f'Negative: {class_names[y_pred]}')
```

This code is saving model.

```
# Save model
output_path = "/content/drive/MyDrive/[AI] Ngoc Quy/Bully/bully_detection_model"
torch.save(model.state_dict(), f'{output_path}.pth')
tokenizer.save_pretrained(output_path)
```

3. Model Predicting

This code is

```
# "load_model_and_tokenizer" function
def load_model_and_tokenizer(model_class, tokenizer_class, model_path, tokenizer_path,
```

```
device):
    # Load model
    model = model_class(n_classes=7, device=device)
    model.load_state_dict(torch.load(model_path, map_location=torch.device('cpu')))
    model.eval()

    # Load tokenizer
    tokenizer = tokenizer_class.from_pretrained(tokenizer_path)

    return model, tokenizer

model_path = '/content/drive/MyDrive/[AI] Ngoc Quy/Bully/bully_detection_model.pth'
tokenizer_path = '/content/drive/MyDrive/[AI] Ngoc Quy/Bully/bully_detection_model'
# Load saved model and tokenizer
loaded_model, loaded_tokenizer = load_model_and_tokenizer(SentimentClassifier,
AutoTokenizer, model_path, tokenizer_path, device)
```

This code is making sentiment predictions on a given input sentence using a loaded sentiment analysis model and tokenizer.

```
class_names = [0, 1]
# "predict_sentiment" function
def predict_sentiment(sentence, tokenizer, model, device, max_len=120):
    # Tokenize the input sentence
    encoded_input = tokenizer.encode_plus(
        sentence,
        max_length=max_len,
```

```
        truncation=True,
        add_special_tokens=True,
        padding='max_length',
        return_attention_mask=True,
        return_token_type_ids=False,
        return_tensors='pt',
    )

    # Move input tensors to the appropriate device
    input_ids = encoded_input['input_ids'].to(device)
    attention_mask = encoded_input['attention_mask'].to(device)

    # Make the prediction
    with torch.no_grad():
        output = model(input_ids, attention_mask)
        _, predicted_label = torch.max(output, dim=1)

    return predicted_label.item()

# Example usage
sentence_to_predict = "Chào buổi sáng"
predicted_label = predict_sentiment(sentence_to_predict, loaded_tokenizer, loaded_model, device)


print(f"Predicted Sentiment Label: {class_names[predicted_label]}")
```

4. Deployment

4.1. Creating Template

This HTML template is a crucial part of web application, as it allows users to interact with sentiment analysis model by entering sentences and viewing the predictions on the web page.

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Sentiment Analysis</title>

    <style>

        body {

            font-family: 'Arial', sans-serif;

            margin: 20px;

            text-align: center;

            background-color: #f4f4f4;

        }

        h1 {

            color: #333;

        }

        form {

            max-width: 400px;

            margin: auto;

            background-color: #fff;

            padding: 20px;

            border-radius: 8px;
```

```
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}
label {
    display: block;
    margin-top: 10px;
    color: #555;
    font-weight: bold;
}
input[type="text"] {
    width: calc(100% - 22px);
    padding: 10px;
    margin-top: 5px;
    margin-bottom: 20px;
    box-sizing: border-box;
    border: 1px solid #ccc;
    border-radius: 4px;
    display: inline-block;
}
button {
    background-color: #4CAF50;
    color: white;
    padding: 10px 15px;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    font-size: 16px;
```

```
        }
        button:hover {
            background-color: #45a049;
        }
        h2 {
            margin-top: 20px;
            color: #333;
        }
        p#result {
            font-size: 18px;
            font-weight: bold;
            color: #333;
        }
    </style>
</head>
<body>
    <h1>Sentiment Analysis</h1>
    <form action="/predict" method="post">
        <label for="sentence">Enter a sentence:</label>
        <input type="text" name="sentence" id="sentence" required>
        <br>
        <button type="submit">Predict</button>
    </form>
    <h2>Result:</h2>
    <p id="result">{{ result }}</p>
</body>
```

```
</html>
```

4.2. Using Flask server

This Flask application loads a sentiment analysis model and tokenizer, provides a web interface for users to input sentences, predicts the sentiment of the input sentences, and displays the results on the web page.

```
from flask import Flask, render_template, request
import os
import torch
import speech_recognition as sr
from transformers import AutoTokenizer
from model import SentimentClassifier, load_model_and_tokenizer, predict_sentiment

app = Flask(__name__)

# Load saved model and tokenizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model_path = r'E:\ngocquy_python\AI\bully\bully_detection_model.pth'
tokenizer_path = r'E:\ngocquy_python\AI\bully\bully_detection_model'
loaded_model, loaded_tokenizer = load_model_and_tokenizer(SentimentClassifier, AutoTokenizer, model_path, tokenizer_path, device)

class_names = {0: 'Non-Violence', 1: 'Violence'}  # Update with your actual class names

@app.route('/')
def index():
    return render_template('index.html')
```

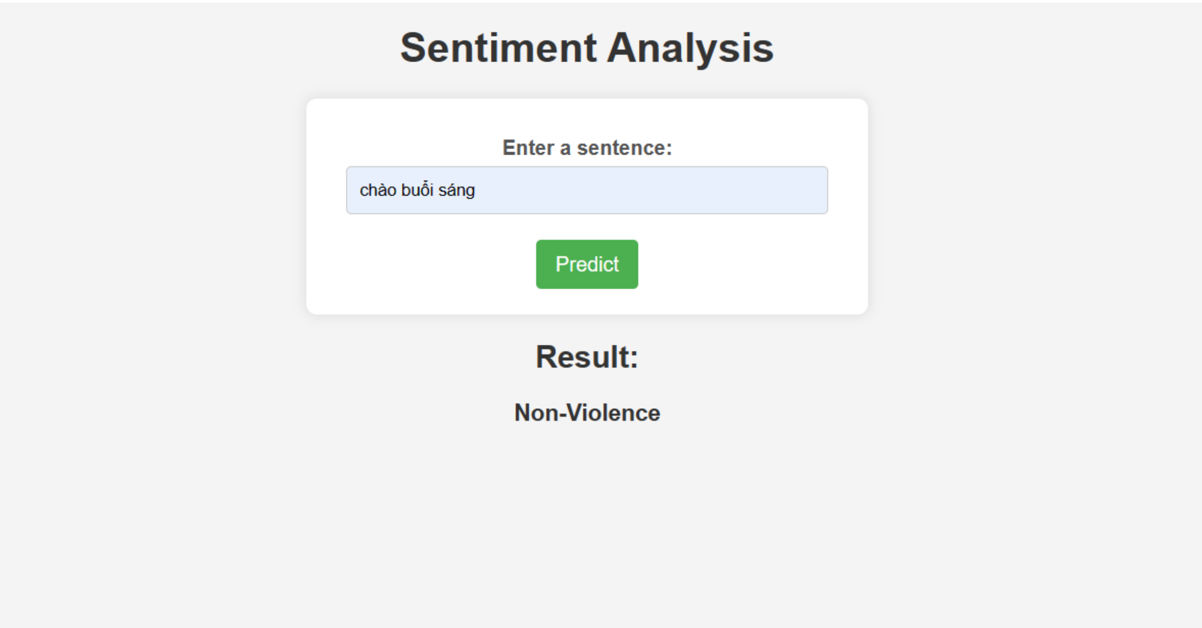
```
@app.route('/predict', methods=['POST'])
def predict():
    if request.method == 'POST':
        sentence = request.form['sentence']
        predicted_label = predict_sentiment(sentence, loaded_tokenizer, loaded_model, device)

        result = class_names[predicted_label]
        return render_template('index.html', result=result)

if __name__ == '__main__':
    app.run(debug=True)
```

III. Conclusion

This challenge has successfully implemented development of a comprehensive system for sentiment analysis and the deployment of Flask-based web application to make predictions based on a pre-trained model.



IV. Reference

The above challenge has reference from

[https://github.com/phusroyal/ViHOS?fbclid=IwAR2rAnYcXjgj4YQ4WXthjYJrb_ij_hwLbz2sF3_Y5lJierEa7VOyT3Aw5fw]

[https://www.kaggle.com/code/trnmtin/phobert-classification-for-vietnamese-text/notebook?fbclid=IwAR32To1PbBtC7eWpS2cglKWhzg78balFHpK5CcYwyBztXzfUBnCwjpFNiDk]