Sentimental Analysis

1. Thông tin liên quan:

Phân tích cảm xúc là một kỹ thuật được sử dụng để xác định cảm xúc hoặc tâm trạng được thể hiện trong một đoạn văn bản. Quá trình này liên quan đến phân tích và phân loại thông tin chủ quan trong văn bản để xác định xem cảm xúc được thể hiện có tính tích cực, tiêu cực hay trung lập.

Mô hình sử dụng:

Tên mô hình: BERT (Bidirectional Encoder Representations from Transformers)

Kiến trúc: BERT là một kiến trúc mô hình dựa trên transformer được giới thiệu bởi Google vào năm 2018. Nó bao gồm nhiều lớp transformer xếp chồng lên nhau, cho phép nắm bắt thông tin ngữ cảnh từ cả hai phía trái và phải của một từ hoặc một đơn vị mã thông tin (token) cho trước.

Phiên bản mô hình: bert-base-uncased

"base" chỉ ra phiên bản cơ bản của kiến trúc mô hình BERT, thường bao gồm 12 lớp transformer.

"uncased" cho biết mô hình được huấn luyện trên văn bản viết thường (chữ thường) và không phân biệt chữ hoa và chữ thường.

Nhiệm vụ của mô hình: Phân loại chuỗi (Sequence Classification)

Phiên bản cụ thể được sử dụng được gọi là "BertForSequenceClassification", đây là một mô hình BERT đã được điều chỉnh để giải quyết các nhiệm vụ phân loại chuỗi.

Mô hình này được thiết kế để phân loại các chuỗi văn bản vào các lớp hoặc các nhãn khác nhau.

Cấu hình của mô hình:

Mô hình được khởi tạo với trọng số được huấn luyện trước của bert-base-uncased, đã được huấn luyện trên một tập lớn văn bản tiếng Anh từ internet.

Mô hình được thích ứng cho việc phân loại chuỗi bằng cách thêm một lớp phân loại lên trên kiến trúc BERT cơ bản.

Số lượng nhãn cho phân loại được đặt thành 4 trong đoạn mã đã cung cấp.

Các thành phần của mô hình:

Tokenizer: Tokenizer được sử dụng là BertTokenizer từ thư viện transformers của Hugging Face. Nó có nhiệm vụ chuyển đổi đầu vào văn bản thành các mã thông tin số học mà mô hình có thể hiểu.

Label Encoder: LabelEncoder từ thư viện scikit-learn được sử dụng để mã hóa nhãn cảm xúc thành giá trị số học để sử dụng trong quá trình huấn luyện và đánh giá.

Huấn luyện:

Mô hình được huấn luyện bằng bộ tối ưu hóa Adam với tỷ lệ học tập là 2e-5.

Hàm mất mát được sử dụng là CrossEntropyLoss, phù hợp cho các nhiệm vụ phân loại đa lớp.

Huấn luyện được thực hiện trong một số lượng epoch cụ thể (5 trong đoạn mã đã cung cấp).

Quá trình huấn luyện được hiển thị, bao gồm mất mát trung bình cho mỗi epoch và độ chính xác trên tập validation.

Dự đoán:

Sau khi huấn luyện, mô hình có thể được sử dụng để dự đoán trên các câu văn mới.

Câu văn đầu vào được mã thông tin hóa bằng tokenizer, và các mã thông tin kết quả được truyền cho mô hình để dự đoán.

Kết quả đầu ra của mô hình cung cấp nhãn cảm xúc dự đoán cho câu văn đầu vào.

BERT là một mô hình dựa trên transformer, một kiến trúc mạng nơ-ron trên máy tính dựa trên cơ sở xử lý song song.

Cụ thể, mô hình sử dụng BertForSequenceClassification từ thư viện transformers của Hugging Face. Đây là một biến thể của BERT đã được điều chỉnh để giải quyết nhiệm vụ phân loại chuỗi. Mô hình BERT sử dụng các lớp transformer xếp chồng lên nhau để nắm bắt thông tin ngữ cảnh từ cả hai phía trái và phải của một từ hoặc một đơn vị mã thông tin.

Thuật toán huấn luyện sử dụng bộ tối ưu hóa Adam để điều chỉnh các trọng số của mô hình. Hàm mất mát được sử dụng là CrossEntropyLoss, phù hợp cho các nhiệm vụ phân loại đa lớp.

1. Mô hình Sử Dụng:

import torch  
from torch.utils.data import DataLoader  
from transformers import BertTokenizer, BertForSequenceClassification  
import pandas as pd  
from sklearn.preprocessing import LabelEncoder  
from tqdm import tqdm  
import networkx as nx  
import matplotlib.pyplot as plt  
#%%  
model\_name = 'bert-base-uncased'  
tokenizer = BertTokenizer.from\_pretrained(model\_name)  
model = BertForSequenceClassification.from\_pretrained(model\_name, num\_labels = 4)  
label\_encoder = LabelEncoder()  
  
#%%  
train\_file = 'twitter\_training.csv'  
val\_file = 'twitter\_validation.csv'  
#%%  
print(torch.cuda.is\_available())  
#%%  
class SentimentDataset(torch.utils.data.Dataset):  
 def \_\_init\_\_(self, data, text\_column, sentiment\_column, graph, max\_num\_neighbors):  
 self.data = data  
 self.text\_column = text\_column  
 self.sentiment\_column = sentiment\_column  
 self.graph = graph  
 self.max\_num\_neighbors = max\_num\_neighbors  
  
 def \_\_len\_\_(self):  
 return len(self.data)  
  
 def \_\_getitem\_\_(self, idx):  
 text = self.data.iloc[idx, self.text\_column]  
 sentiment = self.data.iloc[idx, self.sentiment\_column]  
 if pd.isnull(text):  
 text = None # Replace missing text with an empty string  
  
 if pd.isnull(sentiment):  
 sentiment = 0 # Replace missing sentiment with a default value (0 in this example)  
  
 encoding = tokenizer.encode\_plus(  
 str(text),  
 add\_special\_tokens=True,  
 max\_length=128,  
 padding='max\_length',  
 truncation=True,  
 return\_tensors='pt'  
 )  
 input\_ids = encoding['input\_ids'].squeeze()  
 attention\_mask = encoding['attention\_mask'].squeeze()  
 neighbors = list(self.graph.neighbors(idx))  
  
 if len(neighbors) < self.max\_num\_neighbors:  
 neighbors += [0] \* (self.max\_num\_neighbors - len(neighbors))  
 else:  
 neighbors = neighbors[:self.max\_num\_neighbors]  
  
 additional\_features = torch.tensor(neighbors, dtype=torch.long).unsqueeze(0)  
  
 return {  
 'input\_ids': input\_ids,  
 'attention\_mask': attention\_mask,  
 'sentiment': sentiment,  
 'additional\_features': additional\_features  
 }  
  
#%%  
  
train\_data = pd.read\_csv(train\_file)  
train\_data = train\_data.iloc[:len(train\_data)//15, :]  
graph = nx.Graph()  
# Add nodes to the graph  
graph.add\_nodes\_from(range(len(train\_data)))  
  
# Add edges to the graph (example: connect nodes 0 and 1)  
graph.add\_edge(0, 1)  
max\_num\_neighbors = max(graph.degree, key=lambda x: x[1])[1]  
train\_dataset = SentimentDataset(train\_data, text\_column=3, sentiment\_column=2, graph=graph, max\_num\_neighbors=max\_num\_neighbors )  
val\_data = pd.read\_csv(val\_file)  
val\_dataset = SentimentDataset(val\_data, text\_column=3, sentiment\_column=2, graph=graph, max\_num\_neighbors=max\_num\_neighbors)  
  
label\_encoder.fit(train\_data.iloc[: ,2])  
print(train\_dataset.\_\_len\_\_())  
  
print(label\_encoder.classes\_)  
#%%  
batch\_size = 8  
train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)  
val\_loader = DataLoader(val\_dataset, batch\_size=batch\_size, shuffle=False)  
device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  
model.to(device)  
num\_epochs = 5  
optimizer = torch.optim.Adam(model.parameters(), lr=2e-5)  
loss\_fn = torch.nn.CrossEntropyLoss()  
checkpoint\_path = "model\_checkpoint.pt"  
  
for epoch in range(num\_epochs):  
 model.train()  
 total\_loss = 0  
 progress\_bar = tqdm(train\_loader, desc=f"Epoch {epoch+1}/{num\_epochs}", leave=False)  
  
 for batch in progress\_bar:  
 input\_ids = batch['input\_ids'].to(device)  
 attention\_mask = batch['attention\_mask'].to(device)  
 labels = label\_encoder.transform(batch['sentiment'])  
 labels = torch.LongTensor(labels).to(device)  
  
 optimizer.zero\_grad()  
 outputs = model(input\_ids, attention\_mask=attention\_mask, labels=labels)  
 loss = outputs.loss  
 total\_loss += loss.item()  
 loss.backward()  
 optimizer.step()  
  
 progress\_bar.set\_postfix({'Loss': loss.item(), 'GPU Memory': torch.cuda.memory\_allocated(device=device)})  
  
 average\_loss = total\_loss / len(train\_loader)  
 print(f"Epoch {epoch+1}/{num\_epochs} - Average Loss: {average\_loss}")  
  
 # Evaluation on validation data  
 model.eval()  
 total\_correct = 0  
 total\_samples = 0  
  
 with torch.no\_grad():  
 for batch in val\_loader:  
 input\_ids = batch['input\_ids'].to(device)  
 attention\_mask = batch['attention\_mask'].to(device)  
 labels = label\_encoder.transform(batch['sentiment'])  
 labels = torch.LongTensor(labels).to(device)  
  
 outputs = model(input\_ids, attention\_mask=attention\_mask)  
 \_, predicted\_labels = torch.max(outputs.logits, dim=1)  
  
 total\_correct += (predicted\_labels == labels).sum().item()  
 total\_samples += labels.size(0)  
  
 checkpoint = {  
 'epoch': epoch,  
 'model\_state\_dict': model.state\_dict(),  
 'optimizer\_state\_dict': optimizer.state\_dict(),  
 'loss': loss,  
 }  
 torch.save(checkpoint, checkpoint\_path)  
 accuracy = total\_correct / total\_samples  
 print(f"Validation Accuracy: {accuracy}")

Kết quả huấn luyện:

Epoch 1/5 - Average Loss: 0.5942340768323569

Validation Accuracy: 0.5685685685685685

Epoch 2/5 - Average Loss: 0.15854580796893272

Validation Accuracy: 0.5115115115115115

Epoch 3/5 - Average Loss: 0.1006263473428439

Validation Accuracy: 0.5725725725725725

Epoch 4/5 - Average Loss: 0.07697831928493817

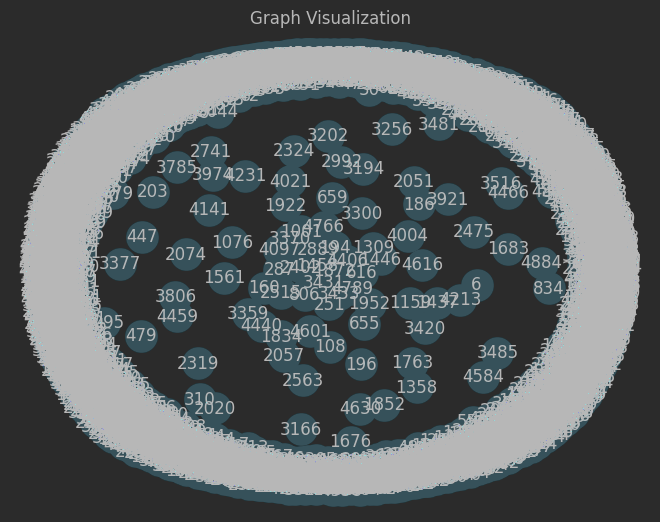
Validation Accuracy: 0.5815815815815816

Epoch 5/5 - Average Loss: 0.0832948047806207

Validation Accuracy: 0.5245245245245245

Đồ thị:

Đồ thị đại diện cho một mạng lưới các đỉnh và các cạnh. Mỗi đỉnh đại diện cho một điểm dữ liệu hoặc một ví dụ trong tập dữ liệu, và các cạnh đại diện cho các kết nối hoặc mối quan hệ giữa các đỉnh. Trong code này, đồ thị được sử dụng để lưu trữ các mối quan hệ giữa các điểm dữ liệu, và các hàng xóm của mỗi đỉnh.



Đồ thị đại diện cho sự phân phối của “nhãn” Cảm Xúc.

