Sentiment Analysis Project Documentation

Chapter 1: Custom Training the Model on Kaggle

1.1 Setting Up Kaggle Jupyter Notebook

To get started with training the model, we needed a cloud-based development environment. Kaggle provided a free and powerful notebook interface with GPU support and pre-installed libraries, which made it ideal for this project.

1.1.1 Verifying Account

Before running code on Kaggle, we verified our account to unlock internet access. This was essential for downloading pretrained models and external libraries like Hugging Face Transformers and WordCloud.

> 💡🔐 ***Navigate to your Kaggle account settings and complete the phone number and email verification process.***

1.1.2 Enabling Internet in Kaggle

To enable internet in your notebook:

1. Click on the Settings tab (⚙️ gear icon) on the right-hand sidebar of your notebook.

2. Toggle the Internet option to ON.

3. Restart the kernel to ensure internet access is applied properly.

> 💡 ***Tip: Restarting the kernel ensures that internet-enabled resources are reinitialized correctly.***

1.1.3 Setting Up GPU and Runtime Configuration

Kaggle provides access to both CPU and GPU environments. To accelerate model training with transformers like BERT, enabling GPU significantly reduces training time and improves resource utilization.

Steps to Enable GPU in Kaggle Notebook:

1. 1. Click the **Settings (⚙)** icon on the top-right of the notebook interface.

2. Under **Accelerator**, select **GPU**.

3. Restart the kernel to apply changes.

Once enabled, you can verify GPU access with:

```
import torch
print(torch.cuda.is_available())  # Output should be True
```

To move your model and data to the GPU:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

All tensors used in training and inference must also be moved to the same device:

```
input_ids = batch['input_ids'].to(device)
attention_mask = batch['attention_mask'].to(device)
labels = batch['label'].to(device)
```

> 💡⚡ ***Training time was reduced from hours to minutes using GPU acceleration. This step was essential for handling large datasets and transformer models efficiently.***

1.2 Code Walkthrough

This section outlines the full pipeline of preparing and training a custom BERT model for sentiment analysis on YouTube comments. The primary objective is to transform raw textual data into a form suitable for deep learning, fine-tune a pretrained transformer model, and prepare the outputs for seamless deployment. Every step—from importing libraries and preprocessing data to tokenization, training, and model saving—ensures that the model is robust, reproducible, and ready for real-world inference.

1.2.1 Importing Libraries

Key libraries used

```
import torch
import pandas as pd
from transformers import BertTokenizer, BertForSequenceClassification
from torch.utils.data import Dataset, DataLoader
```

1.2.2 Loading and Cleaning Data

```
df = pd.read_csv('youtube_comments.csv')
df.dropna(inplace=True)
```

1.2.3 Balancing the Dataset

```
df_positive = df[df['label'] == 1]
df_negative = df[df['label'] == 0]
df_balanced = pd.concat([df_positive.sample(3000), df_negative.sample(3000
```

df_positive = df[df['label'] == 1]
df_negative = df[df['label'] == 0]
df_balanced = pd.concat([df_positive.sample(3000), df_negative.sample(3000)])

1.2.4 Visualization

```
from wordcloud import WordCloud
wc = WordCloud().generate(" ".join(df['text']))
plt.imshow(wc, interpolation='bilinear')
plt.title("Top words in YouTube Comments")
plt.axis('off')
plt.show()
```

1.2.5 Text Preprocessing

Removed emojis, punctuation, and links. Added lowercasing and optional stopword removal to standardize input.

Example preprocessing function:

```
import re
stopwords = set(['the', 'is', 'and', 'a', 'an', 'in', 'on', 'at'])

def preprocess_text(text):
    text = text.lower()
    text = re.sub(r"http\S+", "", text)  # Remove links
    text = re.sub(r"[^a-zA-Z\s]", "", text)  # Remove punctuation
    tokens = text.split()
    tokens = [word for word in tokens if word not in stopwords]
    return " ".join(tokens)
```

💡 This function ensures input is clean and consistent before tokenization.

1.2.6 Tokenization

Tokenizer splits text into subwords compatible with BERT vocabulary:

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

> 💡☑ ***We chose `bert-base-uncased` for its balance between performance and speed. Models like `bert-large` offer more accuracy but are slower and harder to deploy on limited resources.***

1.2.7 Dataset & DataLoader

Define a custom dataset class:

```python
class CommentDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        encoding = self.tokenizer(
            self.texts[idx],
            add_special_tokens=True,
            max_length=self.max_len,
            padding='max_length',
            return_attention_mask=True,
            truncation=True,
            return_tensors='pt')
        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'label': torch.tensor(self.labels[idx], dtype=torch.long)
        }
```

1.2.8 Model Setup

Instantiated the model with a classification head:

```
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)
```

> 🧠 ***This sets up the model to output logits for binary classification (positive/negative sentiment).***

We chose `bert-base-uncased` due to its practical balance between performance, accuracy, and resource requirements.

- **Size**: With around 110 million parameters, it is significantly smaller than bert-large, making it feasible for training on Kaggle and deployment on free-tier platforms like Railway or Hugging Face Spaces.
- **Speed**: The reduced size leads to faster inference times and lower cold start overhead—crucial for web-based sentiment applications.

- **Accuracy**: While not as accurate as bert-large, it still performs very well on binary classification tasks, especially when fine-tuned.
- **Community Support**: It is one of the most tested and stable base models in the Hugging Face ecosystem.

Trade-offs:

- bert-base-uncased may slightly underperform compared to larger models on nuanced classification problems.
- However, the speed, deployment flexibility, and resource efficiency outweighed those concerns for our use case.

🔍 For projects requiring higher accuracy and where resources allow, exploring bert-large, RoBERTa, or DistilBERT variants may be beneficial.

1.2.9 Training Loop

Used AdamW optimizer and a learning rate scheduler:

```
optimizer = AdamW(model.parameters(), lr=2e-5)
for epoch in range(epochs):
        model.train()
        for batch in train_loader:
                optimizer.zero_grad()
                outputs = model(input_ids=batch['input_ids'],
                attention_mask=batch['attention_mask'], labels=batch['label'])
                loss = outputs.loss
                loss.backward()
                optimizer.step()
```

1.2.10 Saving Model and Tokenizer

After training:

```
model.save_pretrained('model')
tokenizer.save_pretrained('model')
```

- This saves the model and tokenizer in a Hugging Face-compatible format, producing several key files:

- config.json: Defines the model architecture (e.g., number of hidden layers, number of attention heads, label mappings). This file is essential for reconstructing the model class during inference.
- pytorch_model.bin or model.safetensors: Contains the trained weights for all model parameters. During inference, this is loaded into the model instance to reproduce the trained behavior.
- vocab.txt: Holds the vocabulary used by the tokenizer. This ensures consistent tokenization between training and inference stages.
- tokenizer_config.json and special_tokens_map.json: Metadata files that inform how the tokenizer should behave (e.g., which token is used for padding, separation, classification).

> *Saving both the model and tokenizer together ensures full reproducibility and smooth deployment on Hugging Face Model Hub or in any inference environment using Transformers.*

1.2.11 Evaluation

Assessed model with classification metrics:

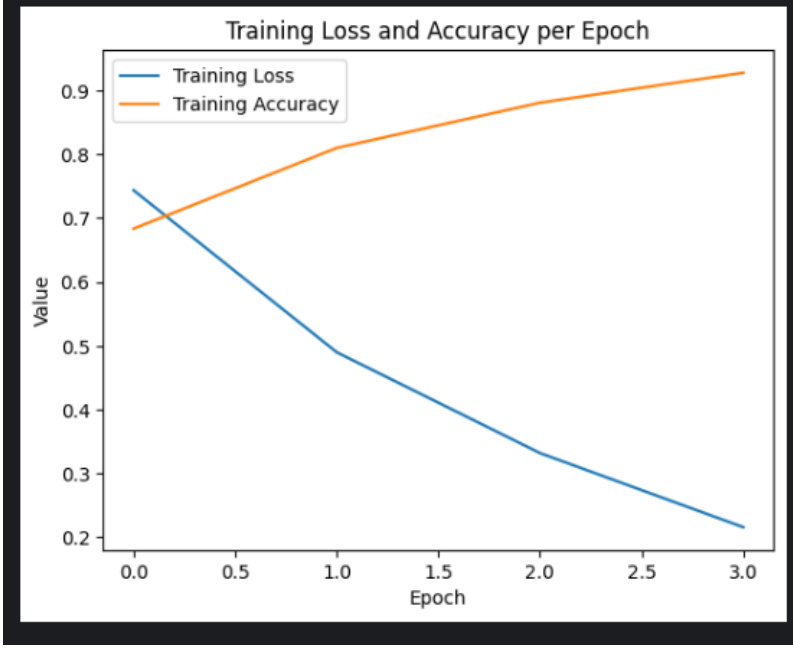
```
from sklearn.metrics import classification_report
preds = [0, 1, 1, 0]
labels = [0, 1, 0, 0]
print(classification_report(labels, preds))
```

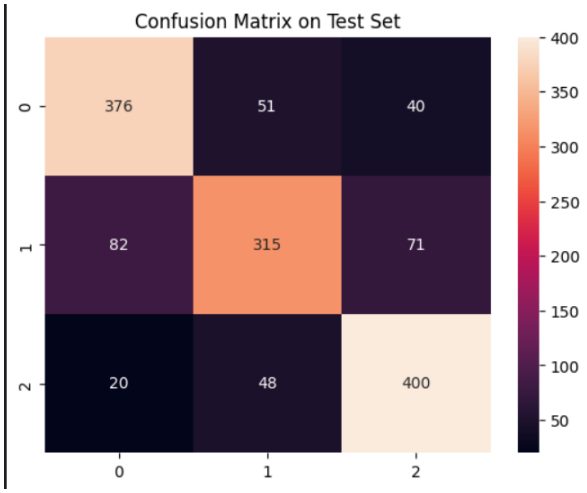
Sample output:

```
plt.plot(train_losses, label='Training Loss')
plt.plot(train_accuracies, label='Training Accuracy')
plt.xlabel("Epoch")
plt.ylabel("Value")
plt.title("Training Loss and Accuracy per Epoch")
plt.legend()
plt.show()
```

```
Epoch 1 Progress: 100% ████████████████████ 176/176 [08:24<00:00,  2.29s/it]

Training loss: 0.7435 | Accuracy: 0.6835

Epoch 2/4

Epoch 2 Progress: 100% ████████████████████ 176/176 [08:30<00:00,  2.26s/it]

Training loss: 0.4898 | Accuracy: 0.8099

Epoch 3/4

Epoch 3 Progress: 100% ████████████████████ 176/176 [08:30<00:00,  2.27s/it]

Training loss: 0.3320 | Accuracy: 0.8807

Epoch 4/4

Epoch 4 Progress: 100% ████████████████████ 176/176 [08:29<00:00,  2.27s/it]

Training loss: 0.2159 | Accuracy: 0.9276
```

```
Training loss: 0.2159 | Accuracy: 0.9276
```





Optionally, a confusion matrix was generated to visually analyze misclassifications.