CSE508_Winter2024_A4_MT23063_Report

KANTILAL PRAVINBHAI PATEL
MT23063
Kantilal23063@iiitd.ac.in

Question 1: Clean and preprocess the dataset's 'Text' and 'Summary' columns.

In this Question, we want to clean and preprocess the 'text' and 'Summary' columns from the dataset so we write a Python script that preprocesses textual data from a CSV file containing reviews by removing HTML tags, converting text to lowercase, removing punctuation, eliminating stopwords, and consolidating extra spaces. It utilizes libraries like BeautifulSoup, pandas, and re for text processing tasks. The cleaned data is then saved into a new CSV file for further analysis or applications. This preprocessing step is crucial in natural language processing tasks as it helps standardize and optimize the text data for downstream tasks such as sentiment analysis or classification.

```
# Load stopwords
stop_words = set(stopwords.words('english'))

# Function to clean text
def clean_text(text):
    if pd.isnull(text):
        return ""
    # Remove HTML tags
    text = BeautifulSoup(text, "html.parser").get_text()
    # Lowercase the text
    text = text.lower()
    # Remove punctuation
    text = re.sub(r'[^\w\s]', '', text)
    # Remove stopwords and extra spaces
    text = ' '.join(word for word in text.split() if word not in stop_words)
    return text

# Read the CSV file
df = pd.read_csv('/kaggle/input/cse508-winter2024-a4-data/Reviews.csv')

# Clean 'Text' column
df['Text'] = df['Text'].apply(clean_text)

# Clean 'Summary' column
df['Summary'] = df['Summary'].apply(clean_text)

# Generate new CSV file with cleaned data
df.to_csv('/kaggle/working/cleaned_data.csv', index=False)
```

Question 2: Model Training.

For This, we first Divide the dataset into Training (75%) and Testing (25%).

Divide the dataset into training and testing (75:25)

```
df = pd.read_csv('/kaggle/working/cleaned_data.csv').head(10000)

select_column = df[['Score','Text', 'Summary']]

# Split the dataset into training and testing sets (75:25 ratio)
training_df, testing_df = train_test_split(select_column, test_size=0.25, random_state=42)
```

After That, we define the Custom Dataset Class. This Python class, named CustomData, is designed to prepare textual data for training a neural network model, likely for tasks such as text summarization or sentiment analysis. It is implemented as a custom dataset using PyTorch's Dataset class. The class takes as input a DataFrame containing review data, a tokenizer, and a maximum sequence length. During initialization, it sets up attributes such as the DataFrame, tokenizer, and maximum sequence length. The class defines methods for determining the length of the dataset and retrieving individual samples. For each sample, it concatenates the review text and summary text, tokenizes the combined text using the provided tokenizer, and prepares the input tensors for the model by encoding, padding, and truncating the tokenized sequences. Additionally, it converts the score associated with each review into a PyTorch tensor, which could be useful for supervised learning tasks. This class facilitates efficient text data processing for training neural network models.

```
class CustomData(Dataset):
    def __init__(self, dataframe, tokenizer, max_length):
        super().__init__()
        self.df = dataframe
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.tokenizer.pad_token = self.tokenizer.eos_token
    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        reviewtext = str(self.df.iloc[idx]['Text'])
        summarytext = str(self.df.iloc[idx]['Summary'])

        # Combine review text and summary text
        text = f"Review Text: {reviewtext}\nSummary: {summarytext}"

        # Tokenize the combined text
        inputs = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_length,
            padding='max_length',
            truncation=True,
            return_tensors='pt'
        )
        input_ids = inputs['input_ids'].squeeze(0)
        attention_mask = inputs['attention_mask'].squeeze(0)
```

After that, we write a Python script for training a Gpt model. Using PyTorch, this Python script demonstrates the fine-tuning process of the GPT-2 language model for a specific task, likely text generation or completion. It begins by instantiating the GPT-2 tokenizer and model from the 'gpt2' pre-trained checkpoint. The training data is prepared using a custom dataset and data loader. Key hyperparameters such as learning rate, epochs, and warmup steps are defined, and an AdamW optimizer with a linear scheduler is employed for training. The fine-tuning loop iterates over the specified number of epochs, within which batches of data are processed. During each iteration, input tensors are fed into the model, and the loss is calculated based on model predictions compared to the ground truth labels. Backpropagation is then performed to update the model parameters, followed by optimization and scheduler step updates. After training is completed, the fine-tuned model is saved to a specified directory. This script provides a clear implementation of fine-tuning a pre-trained GPT-2 model for custom downstream tasks, enhancing its capability to generate or comprehend specific types of text data.

Training Loop

```
# Instantiate GPT-2 tokenizer and model
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
model = GPT2LMHeadModel.from_pretrained('gpt2')

# Define dataset and dataloader
training_dataset = CustomData(training_df, tokenizer, max_length=128)
train_loader = DataLoader(training_dataset, batch_size=10, shuffle=True)

# Define optimizer and scheduler
learning_rate = 1e-5
epochs = 3
warmup_steps = int(0.1 * len(train_loader) * epochs)
optimizer = AdamW(model.parameters(), lr=learning_rate)
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=warmup_steps, num_training_steps=len(train_loa

# Fine-tuning loop
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
model.train()

for epoch in range(epochs):
    for batch in train_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
```

This Python function, `generate_summary`, aims to produce a summary based on a given review text using a fine-tuned GPT-2 language model. The review text is tokenized using the GPT-2 tokenizer and converted into input tensors, ensuring they are placed on the same device as the model (either GPU or CPU). Subsequently, the fine-tuned GPT-2 model, previously saved in the specified directory, is loaded onto the device. The summary is generated using the model's `generate` method, with parameters specifying the maximum length of the summary, the number of beams for beam search, and early stopping criteria. Finally, the generated summary tokens are decoded into human-readable text format, excluding special tokens, and returned as the output. This function encapsulates the process of utilizing a fine-tuned GPT-2 model to generate concise summaries from review texts, facilitating automated summarization tasks within natural language processing applications.

Generating Summaries

```python
def generate_summary(review_text):
    # Tokenize the review text
    inputs = tokenizer.encode_plus(
        review_text,
        return_tensors="pt",
        max_length=1024,
        truncation=True
    )

    # Move input tensors to the same device as the model
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    inputs = {key: tensor.to(device) for key, tensor in inputs.items()}

    # Generate summary using the model
    model = GPT2LMHeadModel.from_pretrained('/kaggle/working/fine_tuned_gpt2_Model').to(device)
    summary_ids = model.generate(inputs['input_ids'], max_length=1024, num_beams=4, early_stopping=True)

    # Decode the generated summary tokens
    generated_summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)

    return generated_summary
```

This code segment aims to evaluate the performance of the summarization model on a given dataset by calculating ROUGE scores for generated summaries compared to actual summaries. It begins by opening the specified CSV file and iterating over its rows, with a limit set by `num_rows`. For each row, it retrieves the review text and the actual summary. If the actual summary is empty, the row is skipped and marked accordingly. Otherwise, it proceeds to generate a summary using the `generate_summary` function previously defined. The generated summary is then split from the original review text, and ROUGE scores are calculated by comparing the generated summary to the actual summary using a function named `rougescore`. Finally, the results, including the review text, generated summary, and ROUGE scores for various metrics (ROUGE-1, ROUGE-2, ROUGE-L) are written to an output CSV file. Upon completion, it notifies the user of the successful calculation and saving of ROUGE scores to the specified output file. This process enables the quantitative evaluation of the summarization model's performance against ground truth summaries, providing insights into its effectiveness in capturing the essence of the original reviews.

```python
# Open output file in write mode
with open(output_file, mode='w', newline='', encoding='utf-8') as output_csv:
    csv_writer = csv.writer(output_csv)
    csv_writer.writerow(['Text', 'Generated Summary', 'ROUGE-1 Precision', 'ROUGE-1 Recall', 'ROUGE-1 F1',
                         'ROUGE-2 Precision', 'ROUGE-2 Recall', 'ROUGE-2 F1',
                         'ROUGE-L Precision', 'ROUGE-L Recall', 'ROUGE-L F1'])
    with open(csv_file, mode='r', newline='', encoding='utf-8') as file:
        csv_reader = csv.DictReader(file)
        for idx, row in enumerate(csv_reader):
            if idx >= num_rows:
                break

            review_text = row['Text']
            actual_summary = row['Summary']  # Adjust column name

            # Skip rows with empty actual summary
            if not actual_summary:
                print(f"Skipping row {idx + 1} due to empty Summary.")
                continue

            # Generate summary
            gen_summary = generate_summary(review_text)
            splited_summary = gen_summary.split(review_text)
            generated_summary = splited_summary[1].strip()

            # Calculate ROUGE scores
            rouge_scores = rougescore(generated_summary, actual_summary)
```