Report

Preprocessing Explanation

In this segment, we focus on the **data preprocessing steps** needed to prepare the **MBTI dataset** for use with the **BERT model**. Below is an explanation of each step carried out in the code:

1. Loading the Dataset

- The MBTI dataset is loaded using pandas.read_csv() function. The dataset consists of posts (text data) and their corresponding personality types based on the **Myers-Briggs Type Indicator (MBTI)** framework. The text data is expected to be in the posts column, and the personality type is in the type column.

```
mbti_df = pd.read_csv('C:/Users/Vanshika/Downloads/mbtidataset/mbti_1.csv')
```

2. Basic Text Preprocessing

- To ensure uniformity and to remove noise from the data, the text is:
    - Converted to lowercase: This eliminates case sensitivity.
    - Special characters are removed: This step ensures that any punctuation, digits, or symbols are eliminated, which are generally irrelevant for the task

```
# Converting text into lowercase
mbti_df['posts'] = mbti_df['posts'].str.lower()

# Removing special characters
mbti_df['posts'] = mbti_df['posts'].str.replace(r'[^a-zA-Z\s]', '', regex=True)

# Step 2: Tokenization of text
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

3. Personal Pronouns Count

- As an additional feature, we count the occurrence of **personal pronouns** (like "I", "me", "my", etc.) in the posts. The hypothesis is that personality types may influence how often a person uses personal pronouns

```
personal_pronouns = ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves']
def count_pronouns(text):
    return sum(1 for word in text.split() if word in personal_pronouns)

mbti_df['pronoun_count'] = mbti_df['posts'].apply(count_pronouns)
```

This function checks each word in a given post and counts it if it is a personal pronoun.

4. Label Encoding

- The **MBTI type** is a categorical variable, so we use **Label Encoding** to convert the text labels into numeric format. This is necessary as most machine learning models (including BERT) require

```
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(mbti_df['type'])
```

5 .Tokenization with BERT's Tokenizer-

- To prepare the text for BERT, we use the BertTokenizer. The tokenizer takes the raw text, splits it into tokens (subwords), and converts these tokens into the corresponding token IDs. The sequence is padded to a fixed length of 512 tokens to be compatible with BERT's input size.

```
def encode_text(text):
    return tokenizer(text, padding='max_length', truncation=True, max_length=512, return_tensors='pt')

# Apply Tokenization to the posts
mbti_df['encoded_posts'] = mbti_df['posts'].apply(encode_text)
```

6. Train-Test Split

- Finally, we split the data into training and testing sets using an **80-20 split**. This ensures that the model will be trained on one subset of data and evaluated on another, helping to prevent overfitting.

```
X_train, X_test, y_train, y_test = train_test_split(mbti_df['encoded_posts'], y_encoded, test_size=0.2, random_state=42)
```

MODEL:

7. Custom Dataset Class for PyTorch

- In this segment, we define a custom **PyTorch dataset** class (MBTIDataset) to handle the tokenized text data. This class is necessary because we need to provide the input data to BERT in the correct format (tokens and attention masks).
- The __getitem__ method retrieves the input tokens, attention masks, and the corresponding label for each sample. It returns them as a dictionary, which is used by the DataLoader.

```
class MBTIDataset(Dataset):
    def __init__(self, texts, labels):
        self.texts = texts
        self.labels = labels

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

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = torch.tensor(self.labels[idx], dtype=torch.long)
        encoded_text = text['input_ids'].squeeze(0)  # Get token ids
        attention_mask = text['attention_mask'].squeeze(0)  # Get attention mask
        return {'input_ids': encoded_text, 'attention_mask': attention_mask, 'labels': label}
```

8. BERT Model for Sequence Classification

- Next, we load the pre-trained **BERT model** for sequence classification. We specify the number of labels (in this case, the number of MBTI types) and use the **BertForSequenceClassification** class, which includes the classification head on top of

```
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=len(label_encoder.classes_))
```

the transformer encoder.

9. Training Arguments

- The training arguments are configured using TrainingArguments from the transformers library. This step allows us to define crucial parameters such as:
    - Learning rate
    - Batch size
    - Number of epochs
    - Weight decay
    - Logging strategy

```
training_args = TrainingArguments(
    output_dir='./results',          # output directory
    evaluation_strategy='epoch',     # evaluation strategy to adopt during training
    learning_rate=2e-5,              # learning rate for optimization
    per_device_train_batch_size=16,  # batch size for training
    per_device_eval_batch_size=16,   # batch size for evaluation
    num_train_epochs=3,              # number of epochs
    weight_decay=0.01,               # strength of weight decay
    logging_dir='./logs',            # directory for storing logs
)
```

10. Trainer Setup

- The Trainer class from the transformers library is used to simplify the training and evaluation process. We pass the model, training arguments, and datasets into the Trainer object.

```
trainer = Trainer(
    model=model,                         # the model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=train_dataset,         # training dataset
    eval_dataset=test_dataset            # evaluation dataset
)
```

11. Model Training

- The train() method of the Trainer class is called to start the training process. The model learns the patterns in the training data for the specified number of epochs.

```
trainer.train()
```

12. Model Evaluation

- After training, the model is evaluated on the test dataset using the evaluate() method. This provides important metrics like accuracy, loss, and other evaluation results that are printed at the end of the process.

```
results = trainer.evaluate()
print(results)
```

Key Concepts -

1. **Loss Functions**: The loss function used in BERT is typically **cross-entropy loss** for classification tasks. The BERT model internally computes this loss during training based on the difference between the predicted and actual labels.
2. **Optimizers**: BERT uses the **Adam optimizer** (adaptive moment estimation), which combines the advantages of both **RMSProp** and **SGD with momentum**. The optimizer adjusts the learning rate based on the gradients of the loss function with respect to the parameters.
3. **Training Strategy**: The model uses a **batch size of 16** to process 16 samples at once, which fits into the GPU memory. The learning rate of **2e-5** is commonly used for fine-tuning BERT.
4. **Evaluation**: After training, the model's performance is evaluated on a test set, allowing us to assess its generalization ability.

Conclusion

In this project, we have created a **model to predict MBTI personality types** based on textual input using **BERT** for **sequence classification**. The overall approach can be divided into several steps:

1. **Data Preprocessing**:
    - The text data from the MBTI dataset was cleaned and tokenized using BERT's tokenizer.
    - We also enriched the dataset by adding features like the count of personal pronouns, which could potentially give additional insights into the personality types.
2. **Model Design**:

- We fine-tuned a pre-trained BERT model, specifically designed for **sequence classification** tasks. The model was set up to classify text into one of the 16 MBTI personality types.

3. **Training**:
    - The model was trained on the dataset, and several important training parameters (such as learning rate, batch size, and number of epochs) were set based on the specific requirements of fine-tuning BERT.
4. **Evaluation**:
    - After training, the model's performance was evaluated using the test set, where the evaluation metrics provided insights into its accuracy and effectiveness at predicting personality types.
5. **Prediction with User Input**:
    - A function was created to accept user input, preprocess the text, and predict the MBTI personality type using the trained model.

This project demonstrates the power of **transformers** and **pre-trained models** for text classification tasks and showcases how natural language can be leveraged to predict personality traits from text. Given its ability to understand deep contextual information, BERT provides an excellent framework for fine-tuning and achieving high accuracy in such classification tasks.