MACHINE LEARNING

Text Classification using BERT

TEAM :

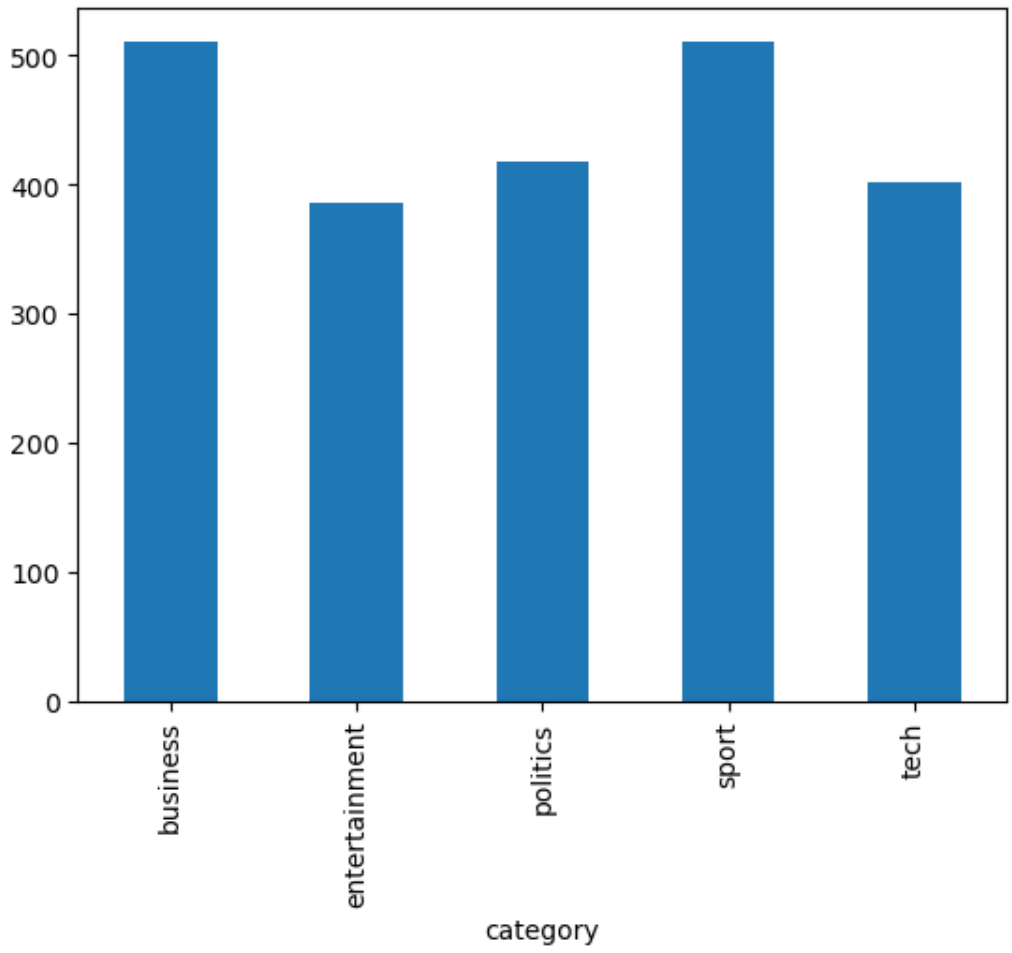
CS22B1052-G CHAITANYA REDDY
CS22B1055-S VIGNESH

1 Objective

The project aims to classify BBC news articles into predefined categories (business, entertainment, politics, sport, and technology) using machine learning. Leveraging the power of the BERT model, the project fine-tunes a pre-trained language representation model for accurate text classification. This involves tokenizing input text, training a custom classification layer, and evaluating performance metrics like accuracy and loss. The ultimate goal is to build a robust model that can generalize well to unseen text data while maintaining high classification accuracy.

Dataset: bbc-text.csv

```
df = pd.read_csv('bbc-text.csv')
df.groupby(['category']).size().plot.bar()
```



2 Methodology

2.1 Preprocessing

Tokenizer: The BERT tokenizer (`bert-base-cased`) is employed to encode text into a numerical format compatible with BERT. This involves:

- Padding sequences to a fixed length (512 tokens).
- Truncating sequences longer than the maximum length.

Label Encoding:
Maps each category to a numeric label:

- `business: 0`, `entertainment: 1`, `sport: 2`, `tech: 3`, `politics: 4`.

```
tokenizer = BertTokenizer.from_pretrained('bert-base-cased')
labels = {'business': 0, 'entertainment': 1, 'sport': 2, 'tech': 3,
          'politics': 4}
```

2.2 Custom Dataset Class

A PyTorch `Dataset` class is defined to:

- Process text samples with the tokenizer.
- Provide numeric labels for each text.
- Enable batch-wise loading during training and evaluation.

```
class Dataset(torch.utils.data.Dataset):

    def __init__(self, df):

        self.labels = [labels[label] for label in df['category']]
        self.texts = [tokenizer(text,
                                padding='max_length', max_length = 512,
                                   truncation=True, return_tensors="pt")
                                    for text in df['text']]

```

```
    def classes(self):
        return self.labels

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

    def get_batch_labels(self, idx):
        # Fetch a batch of labels
        return np.array(self.labels[idx])

    def get_batch_texts(self, idx):
        # Fetch a batch of inputs
        return self.texts[idx]

    def __getitem__(self, idx):

        batch_texts = self.get_batch_texts(idx)
        batch_y = self.get_batch_labels(idx)

        return batch_texts, batch_y
```

3 Model Architecture: BertClassifier

The `BertClassifier` class defines the architecture for fine-tuning a BERT-based model for text classification. Key components include:

BERT Backbone:

- The pre-trained BERT model (`bert-base-cased`) is used to extract contextual embeddings.
- Fine-tuning is enabled by setting `requires_grad=True` for all BERT parameters, allowing the model to adapt to the specific classification task.

Sentence Representation:

- Instead of using the [CLS] token, mean pooling is applied across all token embeddings to create a robust sentence representation.

Additional Layers:

- **Dropout Layer**: A dropout rate of **0.1** is used to reduce overfitting while retaining most of the model's representational capacity.

- **Linear Layer**: A fully connected layer projects the 768-dimensional pooled embeddings to 5 output logits, corresponding to the number of classes.
- **Batch Normalization**: Normalizes the logits to stabilize and accelerate training.

Forward Pass:

- Input token IDs and attention masks are passed through the BERT model to obtain hidden states.
- Mean pooling aggregates the token-level embeddings into a single vector for the sentence.
- The dropout layer and batch normalization are applied before the linear layer to enhance generalization and training stability.
- The final output consists of raw logits suitable for multi-class classification.

```
class BertClassifier(nn.Module):
    def __init__(self, dropout=0.1):
        super(BertClassifier, self).__init__()

        # Load pre-trained BERT model
        self.bert = BertModel.from_pretrained('bert-base-cased')

        # Enable fine-tuning of BERT layers
        for param in self.bert.parameters():
            param.requires_grad = True  # Allow gradients for BERT
               layers

        # Add dropout, linear layers, and batch normalization
        self.dropout = nn.Dropout(dropout)
        self.linear = nn.Linear(768, 5)  # Output layer for 5 classes
        self.bn = nn.BatchNorm1d(5)  # Added batch normalization

    def forward(self, input_id, mask):
        # Get the last hidden state from BERT
        outputs = self.bert(input_ids=input_id, attention_mask=mask,
           return_dict=True)
        hidden_states = outputs.last_hidden_state  # Shape: (batch_size
           , seq_len, 768)

        # Apply mean pooling to get sentence embeddings
        pooled_output = hidden_states.mean(dim=1)  # Mean pooling
           instead of CLS token
```

```
24
25          # Apply dropout, linear transformation, and batch normalization
26          dropout_output = self.dropout(pooled_output)
27          linear_output = self.linear(dropout_output)
28          normalized_output = self.bn(linear_output)  # Normalize outputs
29
30          return normalized_output  # Return raw logits (no activation
               applied)
```

4 Training Pipeline and Model Optimization

The training process involves a well-defined pipeline that ensures efficient learning and evaluation of the model. Below are the key components and methodologies employed in the training function:

1. Data Preparation

- **Training and Validation Data**: The `Dataset` class is used to format the input data into PyTorch tensors. Two dataloaders are created:
  - **Training Dataloader**: Shuffles and batches the training data for each epoch.
  - **Validation Dataloader**: Prepares validation data without shuffling.
- **Batch Size**: Set to `2` to process small subsets of data at a time.

2. Loss Function and Optimizer

- **Loss Function**: Cross-entropy loss is used for multi-class classification tasks. It measures the difference between predicted and actual label distributions.
- **Optimizer**: AdamW (a variant of Adam with weight decay) optimizes the model's parameters.
- **Learning Rate Scheduler**: A linear decay scheduler (`get_scheduler`) is implemented to gradually reduce the learning rate over the training steps.

3. Gradient Accumulation

- **Purpose**: Allows effective training with larger batch sizes on memory-limited GPUs.
- **Mechanism**: Gradients are accumulated over multiple smaller batches before updating the model's parameters, controlled by the `accumulation_steps` variable (set to `4` in this case).

4. Training Phase

The training loop iterates over epochs and batches:

1. **Forward Pass**: Inputs (tokenized text) are passed through the model to get predictions.
2. **Loss Calculation**: Loss is computed using the cross-entropy function and scaled by the accumulation steps.
3. **Backward Pass**: Gradients are computed for model parameters via backpropagation.
4. **Optimizer Step**: After the specified number of batches, the optimizer updates the parameters, and the scheduler adjusts the learning rate.

5. Validation Phase

At the end of each epoch, the model is evaluated on the validation set:

- **Metrics**: Loss and accuracy are calculated to track model performance.

6. Metrics and Logging

After each epoch, the following metrics are logged:

- **Training Loss**: The average loss over all training batches.
- **Training Accuracy**: The proportion of correctly classified samples in the training set.
- **Validation Loss**: The average loss over all validation batches.
- **Validation Accuracy**: The proportion of correctly classified samples in the validation set.

```
def train(model, train_data, val_data, learning_rate, epochs):
    # Prepare data
    train, val = Dataset(train_data), Dataset(val_data)
    train_dataloader = torch.utils.data.DataLoader(train, batch_size=2,
        shuffle=True)
    val_dataloader = torch.utils.data.DataLoader(val, batch_size=2)

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

```

```
    # Loss and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = AdamW(model.parameters(), lr=learning_rate)

    # Scheduler for learning rate decay
    num_training_steps = len(train_dataloader) * epochs
    scheduler = get_scheduler("linear", optimizer=optimizer,
       num_warmup_steps=0, num_training_steps=num_training_steps)

    accumulation_steps = 4  # Define accumulation steps

    for epoch_num in range(epochs):
        # Training phase
        model.train()
        total_acc_train = 0
        total_loss_train = 0

        optimizer.zero_grad()  # Initialize gradients before epoch

        for i, (train_input, train_label) in enumerate(tqdm(
           train_dataloader, desc=f"TrainingEpoch{epoch_num+1}")):
            train_label = train_label.to(device)
            mask = train_input['attention_mask'].to(device)
            input_id = train_input['input_ids'].squeeze(1).to(device)

            # Forward pass
            output = model(input_id, mask)
            batch_loss = criterion(output, train_label.long())
            batch_loss = batch_loss / accumulation_steps  # Scale loss
               by accumulation steps
            batch_loss.backward()  # Backpropagate

            # Track metrics
            total_loss_train += batch_loss.item() * accumulation_steps
            acc = (output.argmax(dim=1) == train_label).sum().item()
            total_acc_train += acc

            # Accumulate gradients
            if (i + 1) % accumulation_steps == 0 or (i + 1) == len(
               train_dataloader):
                optimizer.step()
                scheduler.step()  # Update learning rate
                optimizer.zero_grad()  # Clear gradients for next
```

```
                    accumulation

        # Validation phase
        model.eval()
        total_acc_val = 0
        total_loss_val = 0

        with torch.no_grad():
            for val_input, val_label in tqdm(val_dataloader, desc=
 f"ValidationEpoch{epoch_num+1}"):
                val_label = val_label.to(device)
                mask = val_input['attention_mask'].to(device)
                input_id = val_input['input_ids'].squeeze(1).to(device)

                output = model(input_id, mask)
                batch_loss = criterion(output, val_label.long())
                total_loss_val += batch_loss.item()
                acc = (output.argmax(dim=1) == val_label).sum().item()
                total_acc_val += acc

        # Print epoch summary
        print(f"Epoch{epoch_num+1}/{epochs}")
        print(f"TrainLoss:{total_loss_train/len(train_data):.3f},
           TrainAccuracy:{total_acc_train/len(train_data):.3f}")
        print(f"ValLoss:{total_loss_val/len(val_data):.3f},ValAccuracy
           :{total_acc_val/len(val_data):.3f}")
np.random.seed(112)
df_train, df_val, df_test = np.split(df.sample(frac=1, random_state=42)
   ,
                                     [int(.8*len(df)), int(.9*len(df))
                                        ])

print(len(df_train),len(df_val), len(df_test))
EPOCHS = 5
model = BertClassifier()
LR = 2e-6

train(model, df_train, df_val, LR, EPOCHS)
```

```
Training Epoch 1: 100%|          | 890/890 [02:56<00:00,  5.03it/s]
Validation Epoch 1: 100%|          | 111/111 [00:06<00:00, 16.54it/s]
Epoch 1/5
Train Loss: 0.848, Train Accuracy: 0.311
Val Loss: 0.521, Val Accuracy: 0.649
Training Epoch 2: 100%|          | 890/890 [03:01<00:00,  4.90it/s]
Validation Epoch 2: 100%|          | 111/111 [00:06<00:00, 16.46it/s]
Epoch 2/5
Train Loss: 0.704, Train Accuracy: 0.485
Val Loss: 0.361, Val Accuracy: 0.847
Training Epoch 3: 100%|          | 890/890 [03:01<00:00,  4.90it/s]
Validation Epoch 3: 100%|          | 111/111 [00:06<00:00, 16.56it/s]
Epoch 3/5
Train Loss: 0.652, Train Accuracy: 0.543
Val Loss: 0.305, Val Accuracy: 0.901
Training Epoch 4: 100%|          | 890/890 [03:01<00:00,  4.90it/s]
Validation Epoch 4: 100%|          | 111/111 [00:06<00:00, 16.55it/s]
Epoch 4/5
Train Loss: 0.629, Train Accuracy: 0.594
Val Loss: 0.319, Val Accuracy: 0.928
Training Epoch 5: 100%|          | 890/890 [03:01<00:00,  4.90it/s]
Validation Epoch 5: 100%|          | 111/111 [00:06<00:00, 16.58it/s]
Epoch 5/5
Train Loss: 0.619, Train Accuracy: 0.610
Val Loss: 0.321, Val Accuracy: 0.950
```

5 Model Evaluation

5.0.1 Evaluation Workflow

Forward Pass

For each batch in the test data:

1. **Inputs and Masks**: The input data is tokenized, and the model receives both the input IDs (`input_ids`) and attention masks (`attention_mask`). The attention mask ensures that the model only attends to non-padding tokens during processing.
2. **Output Predictions**: The model produces a set of logits (raw predictions) for each class. The logits are passed through an `argmax` operation, which identifies the class with the highest probability for each sample in the batch.
3. **Accuracy Calculation**: For each batch, the predicted class labels are compared against the true labels (`test_label`), and the number of correct predictions is counted. This count is accumulated to compute the total number of correct predictions across all batches.

Accuracy Aggregation

- **Overall Accuracy**:

After iterating over the entire test set, the total number of correct predictions is summed and divided by the total number of samples in the test dataset to compute the final test accuracy. This metric reflects how well the model performed on the unseen test data.

5.0.2 Evaluation Metrics

Test Accuracy

The **test accuracy** is printed as the primary metric, which represents the proportion of correctly classified samples in the test dataset. It is calculated as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Test Samples}} \tag{1}$$

Classification Report

The **classification report** provides a detailed breakdown of the models performance across all classes. It includes:

- **Precision**: The proportion of true positive predictions for a class relative to all predicted instances of that class.

- **Recall**: The proportion of true positive predictions for a class relative to all actual instances of that class.
- **F1-Score**: The harmonic mean of precision and recall, providing a balanced measure of the classifierâs performance.
- **Support**: The number of actual occurrences of each class in the dataset.

```
def evaluate(model, test_data):
    test = Dataset(test_data)
    test_dataloader = torch.utils.data.DataLoader(test, batch_size=2)

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    if use_cuda:
        model = model.cuda()

    total_acc_test = 0
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for test_input, test_label in test_dataloader:
            test_label = test_label.to(device)
            mask = test_input['attention_mask'].to(device)
            input_id = test_input['input_ids'].squeeze(1).to(device)
            output = model(input_id, mask)
            preds = output.argmax(dim=1)

            # Collect predictions and labels for evaluation
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(test_label.cpu().numpy())

            acc = (preds == test_label).sum().item()
            total_acc_test += acc

    accuracy = total_acc_test / len(test_data)
    print(f'TestAccuracy:{accuracy:.3f}')
    print("\nClassificationReport:")
    print(classification_report(all_labels, all_preds))
    print("\nConfusionMatrix:")
    print(confusion_matrix(all_labels, all_preds))
```

```
Test Accuracy: 0.90
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.95      0.92        57
           1       0.74      0.94      0.83        33
           2       0.98      0.94      0.96        50
           3       0.91      0.78      0.84        40
           4       0.95      0.86      0.90        43

    accuracy                           0.90       223
   macro avg       0.90      0.89      0.89       223
weighted avg       0.91      0.90      0.90       223


Confusion Matrix:
[[54  2  0  1  0]
 [ 1 31  0  0  1]
 [ 1  1 47  0  1]
 [ 1  8  0 31  0]
 [ 3  0  1  2 37]]
```

6 Comparison of Model Performance

We have also done the same text classification task using Vectorization and used multiple ML Models in that and found Random Forest is best Now we are comparing the results of the task with BERT and with Vectorization.

6.0.1 Description of Model:

Random Forest with Vectorization is a machine learning model where the text data is first transformed using a vectorizer (TF-IDF), and then the transformed numerical data is used as input to multiple models and found **Random Forest** model is best suited with high accuracy among for classification. All the values are in (%)

Model	Test Accuracy	Precision	Recall	F1-Score
BERT Model	90	91	90	90
Model 1 (Random Forest)	81	82	81	81
Model 2 (MultinomialNB)	78	79	78	78
Model 3 (LogisticRegression)	78	78	78	78

Table 1: Comparison of Test Accuracy, Precision, Recall, and F1-Score Across Models

The Results show that BERT significantly outperforms traditional models, making it a suitable choice for text classification tasks.

7 GitHub Repositories

7.0.1 Code and Resources

All the complete machine learning model codes and their corresponding outputs have been uploaded in the GitHub. The links to the repositories are provided below:

BERT Project Repository: Code Link

Model Using vectorization: Code link
These repositories include:

1. Preprocessing scripts for the dataset.
2. Jupyter Notebook files containing model implementations and outputs.
3. Scripts for evaluation metrics like confusion matrices and classification reports.

8 References

Garrido-Merchan, Eduardo C., Gozalo-Brizuela, Roberto, and Gonzalez-Carvajal, Santiago, *Comparing BERT Against Traditional Machine Learning Models in Text Classification*, Journal of Computational and Cognitive Engineering, BON VIEW PUBLISHING PTE, Volume 2, Issue 4, Pages 352 to 356, April 2023.
Available at: `http://dx.doi.org/10.47852/bonviewJCCE3202838`