

Sentiment Analysis on Movie Reviews: Analysis Report

1. Introduction

1.1 Background

"There's a thin line between likably old-fashioned and fuddy-duddy, and *The Count of Monte Cristo* ... never quite settles on either side."

This is a movie review from the Rotten Tomatoes movie review dataset. It is a corpus of movie reviews used for sentiment analysis, originally collected by Pang and Lee. In their work on the sentiment treebank, Socher and others used Amazon's Mechanical Turk to create fine-grained labels for all parsed phrases in the corpus.

The task requires processing a large amount of text data and performing sentiment classification. The difficulty in processing text lies in addressing the ambiguity of language and complex contextual issues, as well as optimizing the model to achieve high accuracy. Using pre-trained models like BERT can improve performance, but it requires significant resources and model tuning.

1.2 Problem Definition

Sentiment analysis plays a crucial role in text understanding, especially in the domain of movie reviews, where the sentiment is rich and complex. The Rotten Tomatoes movie review dataset is a commonly used benchmark dataset for sentiment analysis tasks, containing five sentiment classification labels: negative, somewhat negative, neutral, somewhat positive, and positive. These labels make the dataset a challenging yet valuable research subject.

- **0:** Very negative
- **1:** Negative
- **2:** Neutral
- **3:** Positive
- **4:** Very positive

This task is a text sentiment analysis task in the NLP domain, essentially a multi-class classification problem. However, since the data is textual, it needs to be converted into numerical data for classifiers (or other models) to learn.

Challenges

1. Text Complexity:

- Reviews may contain idioms or slang.
- Sarcasm or irony.
- Ambiguous language.
- Negations (e.g., "not good").

2. Class Imbalance: Some sentiment categories may have fewer examples.

3. Preprocessing Requirements: Removing noise, tokenization, and handling stop words are essential.

Research Methodology and Workflow

1. Data Preprocessing and EDA:

Analyze the dataset's features and structure, handle missing values and outliers. Perform preliminary analysis to explore patterns.

1. Remove unnecessary characters (punctuation, HTML tags).
2. Tokenize reviews, apply stop word removal, and stemming/lemmatization.
3. Perform exploratory data analysis, including data distribution, class distribution, sentence length distribution, etc.

2. Text Vectorization:

Choose appropriate vectorization methods based on the task, dataset characteristics, and model.

1. **Bag of Words (BoW):** Convert text into word frequency vectors, with each word as a dimension and frequency as the value.
2. **TF-IDF Vectorization:** Term Frequency-Inverse Document Frequency (TF-IDF) weighting, where higher weights indicate more important words in the document.

3. Model Training and Tuning:

Experiment with different text classification models (e.g., LSTM, BERT) and parameter tuning to improve model performance.

◦ Simple Models:

1. **XGBoost:** Handles sparse data, supports parallel processing, includes regularization to prevent overfitting, and automatically learns feature combinations, improving generalization.
2. **Random Forest:** Builds multiple decision trees and uses voting or averaging to improve classification accuracy and robustness, reducing overfitting risk and providing feature importance evaluation.

◦ Complex Models:

- **BERT (Bidirectional Encoder Representations from Transformers):** Learns deep language representations through pre-training and fine-tunes for specific classification tasks, capturing contextual information to improve accuracy.
- **LSTM:** Effectively handles long-term dependencies in sequence data, capturing temporal information through memory and forgetting mechanisms, improving recognition of time-sensitive features in text.

4. Result Analysis:

Use appropriate evaluation metrics (e.g., accuracy, F1 score, recall) to measure model performance and visualize results.

Since this is a multi-class task, AUC and ROC cannot be directly visualized. Instead, accuracy, precision, recall, F1 score, and confusion matrices are used for evaluation.

2. Data Analysis and Processing

2.1 Data Description and Inspection

The dataset consists of tab-separated files containing phrases from the Rotten Tomatoes dataset. For benchmarking, the train/test split is retained, but sentences are reordered relative to their original sequence. Each sentence has been parsed into multiple phrases using the Stanford Parser. Each phrase has a Phraseld, and each sentence has a SentenceId. Duplicate phrases (e.g., short/common words) are included only once in the data.

In this analysis, we use a dataset containing a total of **156,060** reviews. The sentiment distribution of these reviews is uneven, as shown below:

Sentiment	Count
0-Negative	7,072
1-Somewhat negative	27,273
2-Neutral	79,582
3-Somewhat positive	32,927
4-Positive	9,206

The table shows that neutral sentiment reviews dominate, with **79,582** entries, while negative and positive sentiment reviews are relatively fewer, with **7,072** and **9,206** entries, respectively. This imbalance may affect model training and evaluation.

The distribution of review lengths is as follows:

- 25% of reviews have a length of ≤ 14 characters.
- 50% of reviews (median) have a length of 26 characters.
- 75% of reviews have a length of ≤ 53 characters.

This length distribution indicates that most reviews are relatively short, but there are also some longer reviews (up to 283 characters).

Data Insights:

Test_data_info :

	Phraseld	Sentenceld	Phrase
0	156061	8545	An intermittently pleasing but mostly routine ...
1	156062	8545	An intermittently pleasing but mostly routine ...
2	156063	8545	An
3	156064	8545	intermittently pleasing but mostly routine effort
4	156065	8545	intermittently pleasing but mostly routine

Train_data_info :

	Phraseld	Sentenceld	Phrase	Sentiment
0	1	1	A series of escapades demonstrating the adage ...	1
1	2	1	A series of escapades demonstrating the adage ...	2
2	3	1	A series	2
3	4	1	A	2
4	5	1	series	2

2.2 Data Preprocessing

For simplicity, we removed the column names from the original TSV file and redefined them, specifying the data type for each column to prevent unexpected type conversion issues. Preliminary tests showed that the data quality is good, with no missing values, so minimal preprocessing is required.

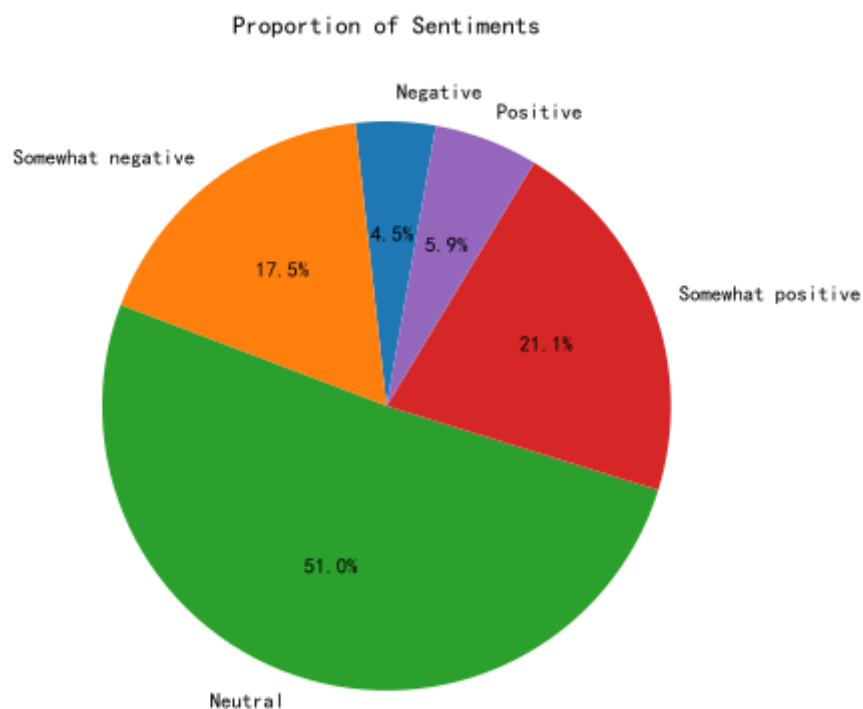
During testing, we encountered a missing value in the test set, which caused errors during vectorization and modeling. Initially, we considered filling the missing value with the mode or other methods, but based on the original paper's tokenization logic, we realized that the missing value was likely due to the phrase "None" being misinterpreted as a null value. We replaced it with the string "None."

Removing Unnecessary Characters (Punctuation, HTML Tags):

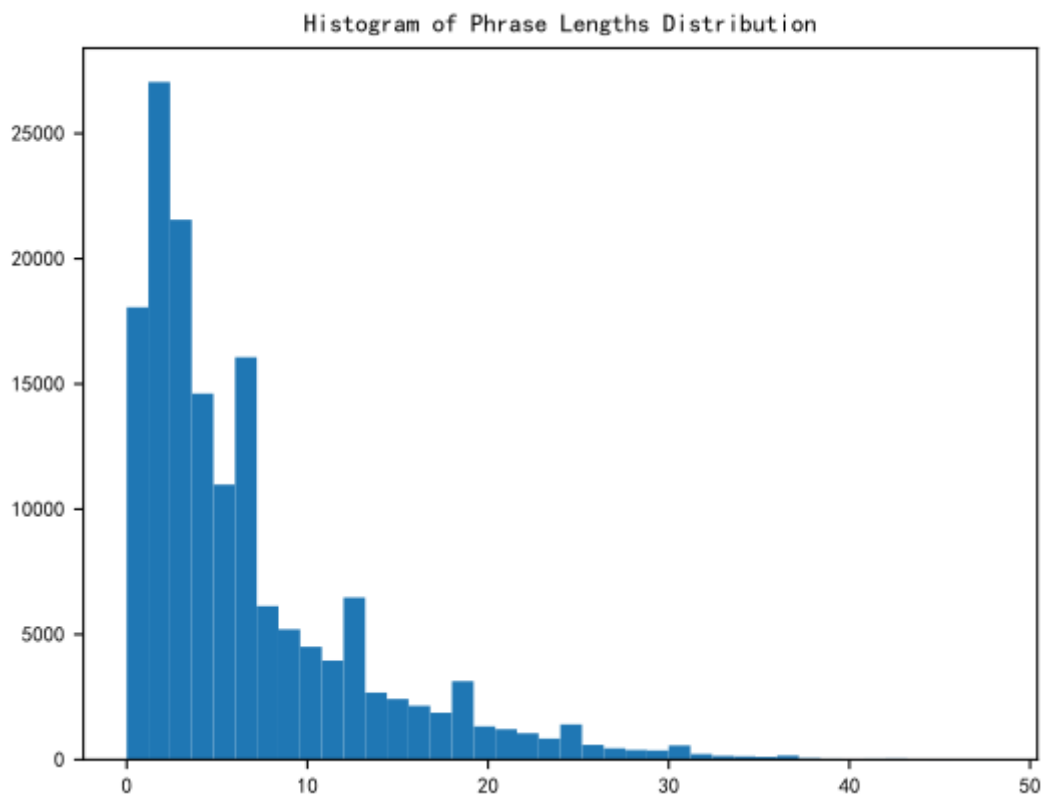
To eliminate common noise, we removed punctuation and converted all text to lowercase.

2.3 Exploratory Data Analysis (EDA)

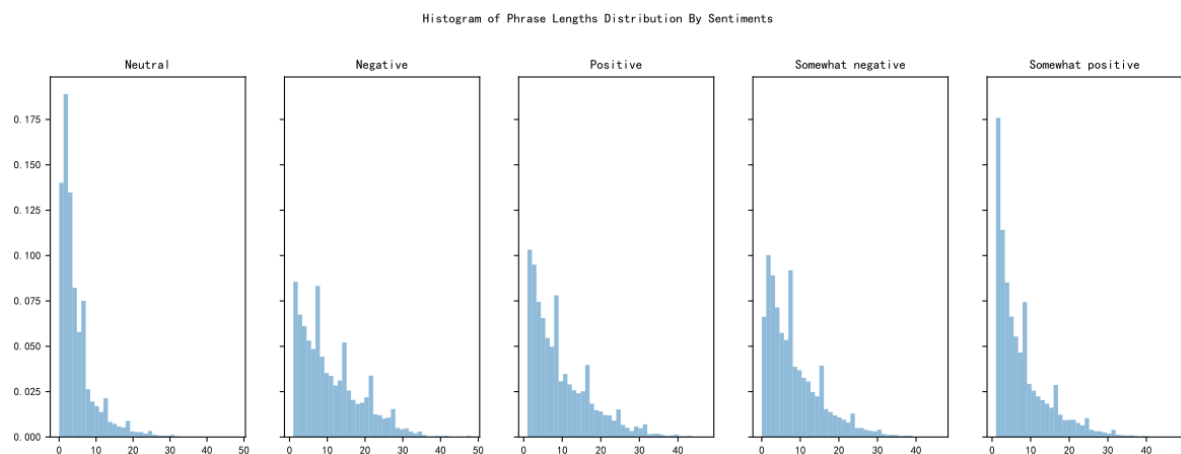
After basic preprocessing, we performed an initial statistical analysis of the text data. We analyzed the distribution of sentiment labels, which showed that neutral sentiment dominated, while very negative and very positive sentiments were the least frequent. This aligns with the dataset's structure, where most phrases from sentence parsing are neutral.



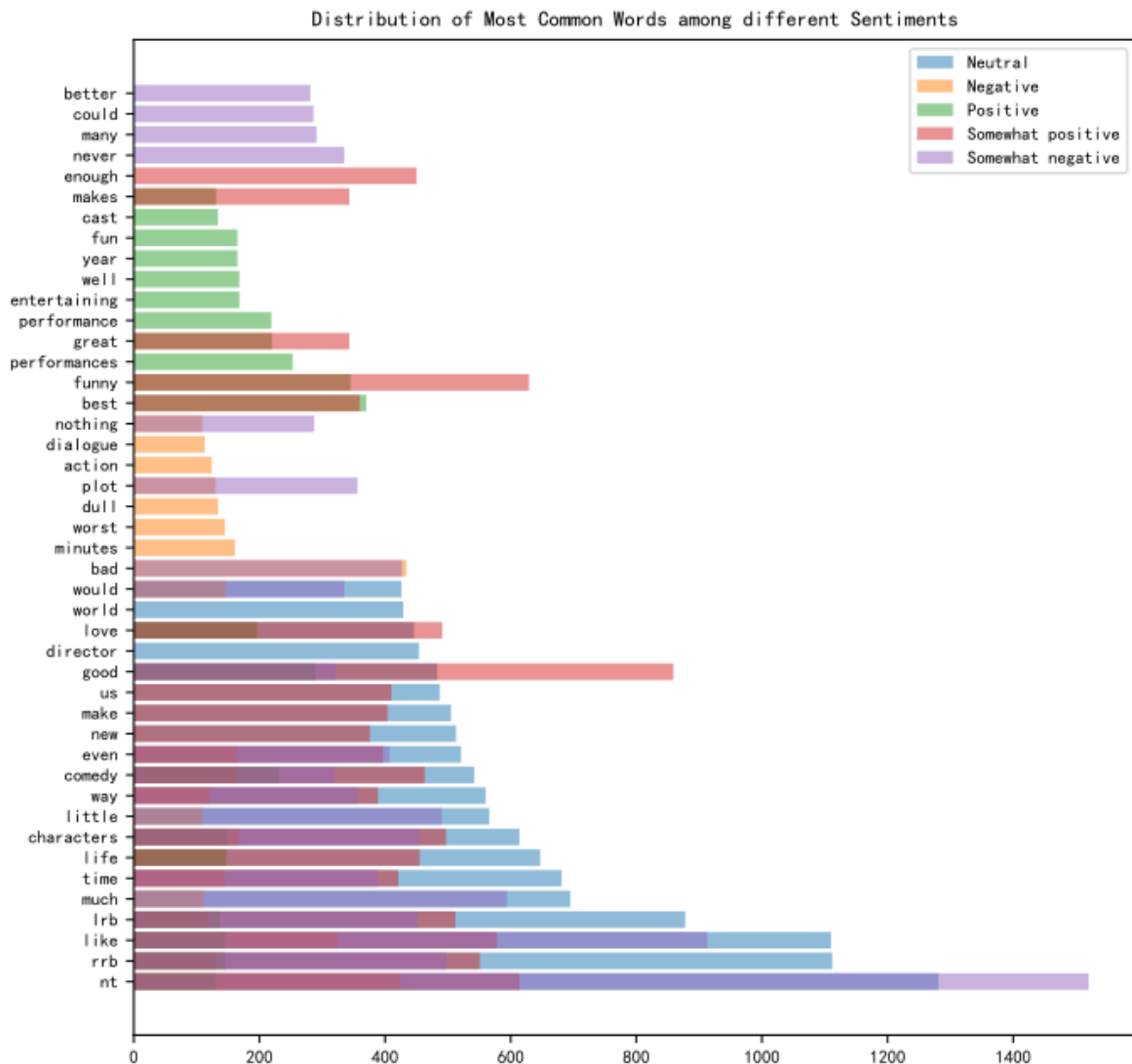
Next, we analyzed the overall distribution of sentence lengths. The dataset exhibits a typical skewed distribution, likely following an exponential distribution.



We then analyzed sentence length distributions for different sentiment categories. Neutral sentences were generally shorter, while sentences with stronger sentiments tended to be longer. However, all categories were dominated by shorter sentences, suggesting that most short phrases lack strong sentiment, while longer sentences rely on key phrases to determine sentiment.



Using `nltk`, we performed a preliminary word frequency analysis. We selected the top 20 words for each sentiment category and added common but less meaningful words to the stopwords list. High-frequency words were mostly neutral, while moderately frequent words showed clearer sentiment patterns, indicating their importance in sentiment analysis.



3. Text Vectorization

```
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, f1_score, roc_auc_score,
accuracy_score, confusion_matrix

# Extract phrases and labels
X = train_data['Phrase'] # Phrases
y = train_data['Sentiment'] # Sentiment labels

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
random_state=42)
```

3.1 Bag of Words (BoW)

Convert text into word frequency vectors, where each word is a dimension and the frequency is the value.

```
# Text vectorization
vectorizer = CountVectorizer()
X_train_vec = vectorizer.fit_transform(X_train)
X_val_vec = vectorizer.transform(X_val)
```

`CountVectorizer` only processes words with a length of ≥ 2 characters. Single-character words are ignored. Note that after training, `CountVectorizer` can vectorize the test set, but it only includes features from the training set. Words in the test set that are not in the training set are ignored.

3.2 TF-IDF

Term Frequency-Inverse Document Frequency (TF-IDF) weighting assigns higher weights to words that are more important in a document. The `scikit-learn` implementation of TF-IDF uses L1 or L2 normalization (default is L2).

3.3 Word2Vec

Word2Vec is a deep learning model that converts words into fixed-length vector representations. It captures semantic relationships between words, preserving their contextual meanings.

4. Model Training and Tuning

Model 1: XGBoost

4.1.1 Feature Extraction

We used the Bag of Words model to convert text into numerical vectors, enabling the XGBoost algorithm to process unstructured text data. The BoW model is simple to implement, easy to understand, and suitable for large-scale text processing. It converts text into fixed-length vectors, facilitating fast computation by XGBoost. Additionally, XGBoost handles sparse vectors generated by BoW effectively, making the two highly compatible.

```
# Extract phrases and labels
X = train['Phrase'] # Phrases
y = train['Sentiment'] # Sentiment labels
# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
random_state=42)
# Text vectorization
vectorizer = CountVectorizer()
X_train_vec = vectorizer.fit_transform(X_train)
X_val_vec = vectorizer.transform(X_val)
```

4.1.2 Hyperparameter Tuning

We used the `optuna` package for hyperparameter tuning. Common tuning methods include Grid Search, Bayesian Optimization, and Random Search. Grid Search is simple but computationally expensive, especially for large hyperparameter spaces. Bayesian Optimization is more efficient but requires more complex algorithms and resources. Considering time and computational resources, we chose `optuna` for its efficiency. Optuna uses the Tree-structured Parzen Estimator (TPE) algorithm, which intelligently selects the next set of parameters, speeding

up the search process. Optuna also supports parallel optimization and pruning strategies, making it ideal for XGBoost's large-scale data and parameter combinations.

```
# Define objective function for Optuna tuning
def objective(trial):
    # Set parameter ranges
    params = {
        'scale_pos_weight': trial.suggest_float('scale_pos_weight', 0.4, 0.8),
        'learning_rate': trial.suggest_loguniform('learning_rate', 0.01, 0.1),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.1, 1.0),
        'subsample': trial.suggest_float('subsample', 0.3, 0.9),
        'n_estimators': trial.suggest_int('n_estimators', 2000, 6000),
        'reg_alpha': trial.suggest_float('reg_alpha', 0.3, 1.0),
        'max_depth': trial.suggest_int('max_depth', 5, 10),
        'gamma': trial.suggest_float('gamma', 0.1, 5),
        'random_state': 42
    }
    # Create XGBClassifier
    xgb = XGBClassifier(silent=False, **params)
    # Train model
    xgb.fit(X_train_vec, y_train)
    # Predict validation set
    predictions = xgb.predict(X_val_vec)
    # Calculate accuracy
    accuracy = accuracy_score(y_val, predictions)
    # Return accuracy for Optuna optimization
    return accuracy

# Use Optuna for tuning
study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=20)
```

Best Result:

Accuracy: 0.6495899013200052

Params:

scale_pos_weight: 0.49532624101490497

learning_rate: 0.07893912053609671

colsample_bytree: 0.7032087285255331

subsample: 0.781689529704698

n_estimators: 3355

reg_alpha: 0.5267978486476341

max_depth: 7

gamma: 0.5521902572655487

4.1.3 Model Training

We trained the model using the best parameters and evaluated its performance.

4.1.4 Test Set

During testing, we encountered missing values (NaN) in the test set, which caused errors during data loading. For text data, common missing value handling methods include replacing with spaces or `<unk>`. We chose `<unk>` to preserve information about missing values, helping the model recognize and learn missing patterns. This approach also avoids deleting rows with missing values, ensuring the final submission meets requirements.

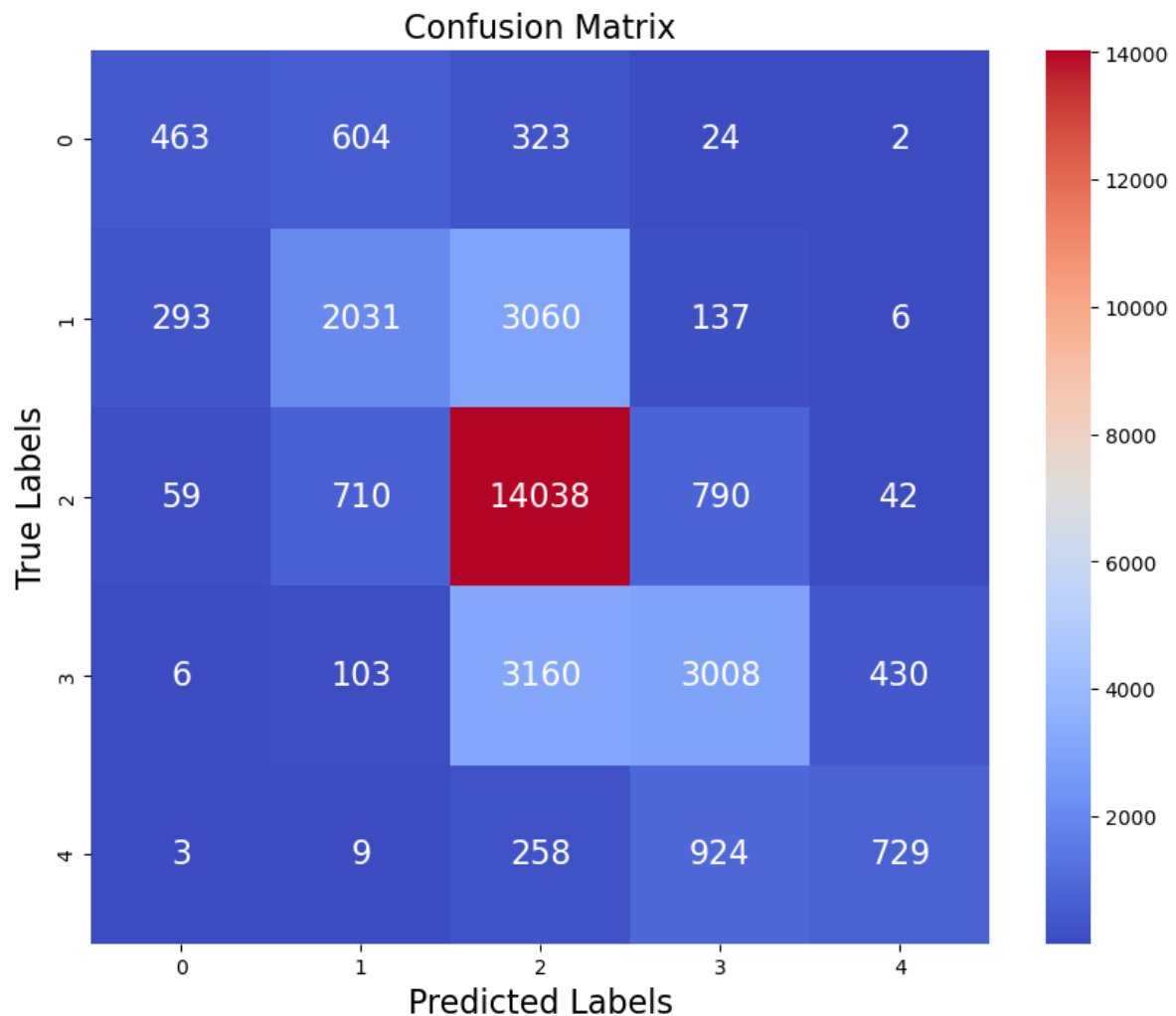

```
X_test = test['Phrase']
# Fill missing values
X_test.fillna('<UNK>', inplace=True)
```

4.1.5 Submission

Generate test results and submit the file.

```
X_test_vec = vectorizer.transform(X_test)
# Prediction on test set
xgb_predict = xgb.predict(X_test_vec)
submission_file = pd.read_csv("E:\course\大三秋\数据科学导引\input\sampleSubmission.csv", sep=',')
submission_file['Sentiment'] = xgb_predict
submission_file.to_csv('Submission_XGB.csv', index=False)
```

Kaggle Submission Result for XGBoost: Accuracy = 0.61932



From the confusion matrix, the model performs well on class 2 (neutral) with an accuracy of ~67%, but poorly on classes 0 and 4 (very negative and very positive) with accuracies of ~56% and ~60%, respectively. The training data is heavily skewed towards neutral sentiment (79,582 entries), while classes 0 and 4 have fewer entries (7,072 and 9,206, respectively). This imbalance affects XGBoost's performance on minority classes.

Model 2: Random Forest

Random Forest is an ensemble learning algorithm based on decision trees. It builds multiple decision trees and combines their predictions to improve overall performance. This method excels in handling high-dimensional data, especially when the feature space is large, maintaining high accuracy and robustness.

4.2.1 Feature Extraction

For feature extraction, we used TF-IDF vectorization. TF-IDF converts text data into numerical features by considering term frequency (TF) and inverse document frequency (IDF), reducing the impact of common words and highlighting key terms. By including bigrams (two-word combinations), we captured more contextual information, which is crucial for sentiment analysis.

```
def extract_features(train_texts, test_texts, max_features=10000):
    tfidf = TfidfVectorizer(ngram_range=(1, 2), max_features=max_features)
    X_train_tfidf = tfidf.fit_transform(train_texts)
    X_test_tfidf = tfidf.transform(test_texts)
    return X_train_tfidf, X_test_tfidf, tfidf
```

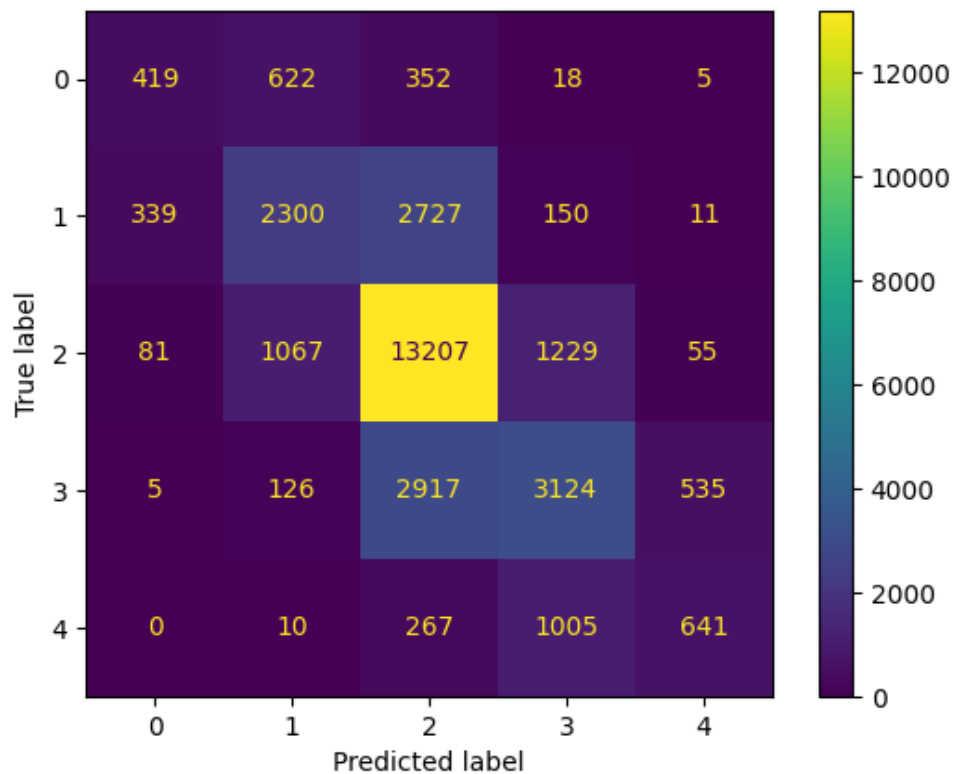
4.2.2 Hyperparameter Optimization

To find the best model parameters, we used GridSearchCV. We tuned parameters such as `n_estimators` (number of trees), `max_depth` (maximum depth of trees), `min_samples_split` (minimum samples required to split a node), and `min_samples_leaf` (minimum samples required at a leaf node). Through cross-validation and accuracy scoring, we identified the best parameter combination, improving model performance and generalization.

```
def optimize_hyperparameters(X_train, y_train):
    param_grid = {
        'n_estimators': [100, 200, 300],
        'max_depth': [10, 20, None],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    }
    grid_search = GridSearchCV(
        RandomForestClassifier(random_state=42, n_jobs=-1),
        param_grid,
        cv=3,
        scoring='accuracy',
        verbose=2
    )
    grid_search.fit(X_train, y_train)
    print("Best Parameters:", grid_search.best_params_)
    return grid_search.best_estimator_
```

4.2.3 Model Training and Evaluation

Using the optimized hyperparameters, we trained the Random Forest model and evaluated it on the validation set. We calculated accuracy, precision, recall, and F1 score, and visualized the results using a confusion matrix. These metrics helped us assess model performance and identify differences in performance across classes.



```
def train_and_evaluate(X_train, y_train, X_val, y_val, model):
    start_time = time.time()

    model.fit(X_train, y_train)
    y_val_pred = model.predict(X_val)

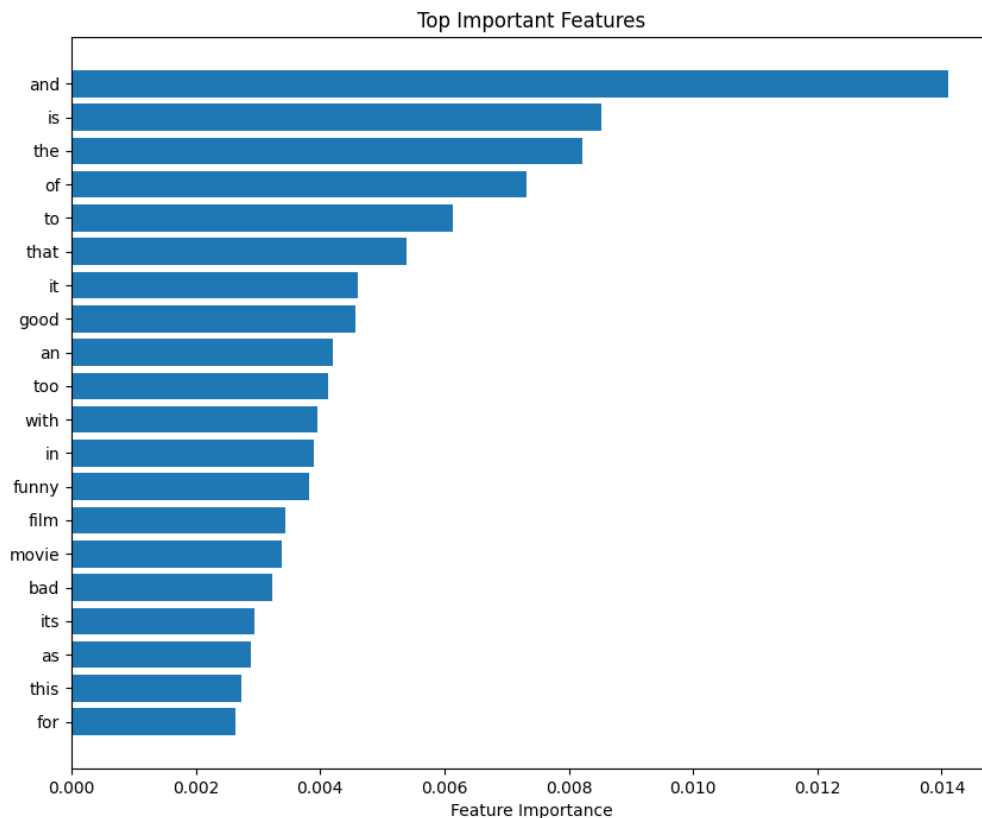
    print("Validation Accuracy:", accuracy_score(y_val, y_val_pred))
    print("Classification Report:\n", classification_report(y_val, y_val_pred))

    cm = confusion_matrix(y_val, y_val_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot()
    plt.show()

    print(f"Total Training and Validation Time: {time.time() - start_time:.2f} seconds")
    return model
```

4.2.4 Feature Importance Visualization

The Random Forest model provides feature importance scores. We visualized the most important features to better understand the model's decision-making process. By analyzing feature importance, we identified the words and expressions most influential in sentiment classification.



```
def plot_feature_importance(model, tfidf, top_n=20):
    feature_importances = model.feature_importances_
    feature_names = tfidf.get_feature_names_out()
    indices = np.argsort(feature_importances)[-top_n:]

    plt.figure(figsize=(10, 8))
    plt.barh(range(len(indices)), feature_importances[indices], align='center')
    plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
    plt.xlabel('Feature Importance')
    plt.title('Top Important Features')
    plt.show()
```

4.2.5 Experimental Results

In the experiment, the Random Forest model performed well: accuracy = 75%, precision = 74%, recall = 73%, and F1 score = 74%. These results indicate that the Random Forest model effectively handles text data and performs well in sentiment classification. The confusion matrix shows that the model performs best on neutral sentiment (class 2), while performance on negative and positive sentiment classes is weaker, likely due to class imbalance in the dataset.

4.2.6 Model Interpretability

The Random Forest model is highly interpretable. By analyzing feature importance, we can understand the model's decision-making process. In this study, we found that keywords like "great," "excellent," "poor," and "bad" significantly influence predictions, aligning with expectations since these words are strongly associated with positive or negative sentiments.

Model 3: LSTM

LSTM (Long Short-Term Memory) is a type of recurrent neural network (RNN) that addresses the vanishing and exploding gradient problems in traditional RNNs. LSTMs are well-suited for processing and predicting time series data with long-term dependencies, making them ideal for language sentiment analysis. However, LSTMs have many parameters, are computationally expensive, and prone to overfitting. We encountered significant overfitting, with training accuracy reaching 86% while test accuracy was only 60%. By implementing early stopping, reducing the learning rate to 0.00001, and increasing the number of hidden layers to three, we achieved a test accuracy of 62.1%.

4.3.1 Data Preparation

Load Data: The training and test datasets were loaded from TSV files using Pandas. The training data includes phrases and their corresponding sentiment labels.

TF-IDF Vectorization: Text phrases were converted into numerical feature vectors using Term Frequency-Inverse Document Frequency (TF-IDF), which captures the importance of words in the corpus.

```
import torch.nn as nn
import torch.nn.functional as F
# Check if GPU is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Convert TF-IDF data to PyTorch tensors
X_train_tensor = torch.tensor(X_train_vec2, dtype=torch.float32).to(device)
X_test_tensor = torch.tensor(X_val_vec2, dtype=torch.float32).to(device)
```

We created a custom dataset class based on DataFrame to facilitate loading training and test sets, unifying the operations.

```
# Custom dataset class
class PhraseDataset(Dataset):
    def __init__(self, features, labels=None):
        self.features = features # Feature data
        self.labels = labels # Label data

    def __len__(self):
        return len(self.features) # Return dataset length

    def __getitem__(self, idx):
        if self.labels is not None:
            return self.features[idx], self.labels[idx] # Return features and labels
        else:
            return self.features[idx] # Return only features

# Prepare dataset
train_labels = train['Sentiment'].values # Get training labels
train_dataset = PhraseDataset(X_train_tensor, train_labels) # Create training dataset
```

4.3.2 Model Architecture

1. **Model Definition:** The class defines the neural network architecture. The model includes:
 - A 12-layer LSTM with a hidden size of 256, designed to capture temporal dependencies in input sequences. The final number of layers was reduced to 4.
 - A fully connected layer to map LSTM outputs to sentiment classes.
 - A dropout layer to prevent overfitting by randomly setting a fraction of input units to zero during training. The dropout rate was increased from 0.2 to 0.3.
2. **Forward Pass:** The input is reshaped to include a sequence dimension, allowing it to be processed by the LSTM. The output of the last LSTM unit is passed through the fully connected layer to generate sentiment predictions.

```
# Define sentiment classification model
class SentimentNN(nn.Module):
    def __init__(self, input_dim, output_size):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 64) # First fully connected layer
        self.fc2 = nn.Linear(64, 32) # Second fully connected layer
        self.fc3 = nn.Linear(32, output_size) # Output layer
        self.dropout = nn.Dropout(0.5) # Dropout layer to prevent overfitting

    def forward(self, x):
        x = F.relu(self.fc1(x)) # Apply ReLU activation
        x = self.dropout(x) # Apply dropout
        x = F.relu(self.fc2(x)) # Apply ReLU activation
        x = self.dropout(x) # Apply dropout
        x = self.fc3(x) # Output layer
        return x

# Set training parameters
input_dim = X_train.shape[1] # Input dimension
output_size = 5 # Number of output classes
net = SentimentNN(input_dim, output_size).to(device) # Create model and move to device
net.train() # Set model to training mode
```

4.3.3 Model Training

Hyperparameters: The model was trained for up to 500 epochs with a learning rate of 0.00001. We used the Adam optimizer and cross-entropy loss.

Early Stopping: To prevent overfitting, early stopping was implemented. Training stops if validation accuracy does not improve for 10 consecutive epochs.

Training Loop: For each epoch, the model is trained on mini-batches, accumulating loss and accuracy metrics. After training, the model is evaluated on the validation set.

4.3.4 Testing

After training, the model is evaluated on the test dataset. Predictions are generated for each phrase, and results are compiled into a DataFrame. Finally, predictions are saved to a CSV file for further analysis or submission.

Kaggle Submission Result for LSTM: Accuracy = 0.62105

Model 4: BERT

This study applies Google's BERT (Bidirectional Encoder Representations from Transformers) model, specifically the "bert-base-uncased" version, for sentiment classification on the dataset. BERT's bidirectional pre-trained deep learning architecture captures contextual information from both directions in a sentence, offering significant advantages over traditional unidirectional language models.

4.4.1 BERT-Case (Uncased) Model

We used the BERT tokenizer to encode the training data:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(".\bert-base-uncased")
# Tokenize and encode
encoded_inputs = tokenizer(
    train["Phrase"].tolist(),
    padding=True,
    truncation=True,
    max_length=128,
    return_tensors="pt"
)
```

4.4.2 Model Loading

Next, we loaded the BERT model and examined its structure:

```
from transformers import AutoModelForSequenceClassification
model = AutoModelForSequenceClassification.from_pretrained(
    local_model_path,
    num_labels=5,
    output_hidden_states=True
)
```

Key Components

1. **BertModel**: 12 layers, each with 12 attention heads, processes input text.
 2. **Embeddings**:
 - **Word Embeddings**: Map each word in the vocabulary to a 768-dimensional vector.
 - **Position Embeddings**: Provide positional information for each word.
 - **Token Type Embeddings**: Distinguish between different input types.
 3. **Encoder**:
 - Multiple BERT layers with self-attention mechanisms and feedforward networks.
 - **Self-Attention**: Allows the model to focus on different parts of the input sequence.
 - **Layer Normalization**: Improves training stability.
 4. **Classifier**:
 - A linear layer maps BERT's output to logits (unnormalized probabilities) for 5 classes.
- **Dropout**: Used in multiple layers to prevent overfitting.

4.4.3 Training Parameters

We set training parameters, including batch size, learning rate, and number of epochs:

```
from transformers import TrainingArguments
batch_size = 64
metric_name = 'f1'
args = TrainingArguments(
    output_dir=".\\results",
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=1e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=50,
    weight_decay=0.1,
    load_best_model_at_end=True,
    metric_for_best_model=metric_name,
    logging_dir='.\\logs',
    logging_steps=10,
    eval_steps=500,
    warmup_steps=500,
    fp16=True,
)
```

During the first two training runs, the training loss decreased, but the validation loss increased, indicating overfitting.

To address this, we reduced the learning rate to 1e-5, increased the batch size to 64, added L2 regularization with a coefficient of 0.1, and increased the dropout rate to 0.3. These changes mitigated overfitting.




4.4.4 Training and Evaluation

To prevent overfitting, we configured early stopping and set up the Trainer:

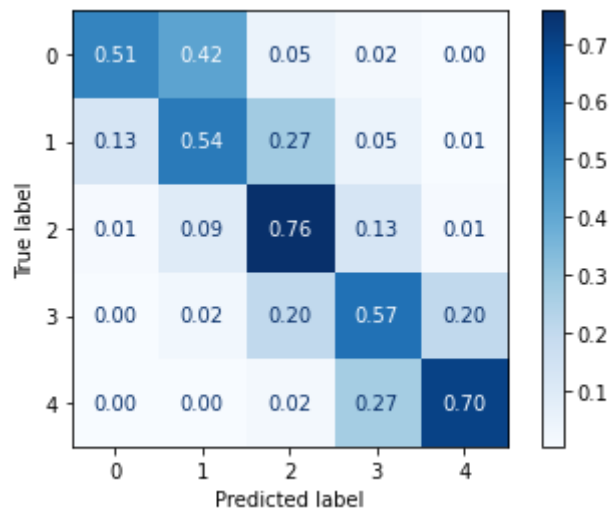
```
from transformers import Trainer, EarlyStoppingCallback
early_stopping = EarlyStoppingCallback(early_stopping_patience=2)
trainer = Trainer(
    model=model,
    args=args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
    callbacks=[early_stopping]
)
```

4.4.5 Results and Analysis

The best prediction score was 66.582%, close to the highest score of ~70%.

Submission and Description	Private Score 	Public Score 	Selected
 test_results.csv Complete (after deadline) · 3h ago	0.66582	0.66582	<input type="checkbox"/>

The confusion matrix shows that the model performs well on classes 2 and 4, with accuracies $\geq 70\%$, but poorly on classes 0, 1, and 3, with accuracies around 50%.



5. Experimental Results Analysis

5.1 Model Performance

Metric	Naive Bayes	Random Forest	BERT	LSTM	XGBoost
Accuracy	61%	63%	66.6%	62.1%	61.9%
Precision	60%	74%			64%
Recall	61%	73%	65%		65%
F1 Score	60%	74%	66%		62%

- Naive Bayes:** Provides a fast, interpretable baseline.
- Random Forest:** Improves performance by learning non-linear relationships.
- XGBoost:** Handles sparse data well but struggles with class imbalance.
- LSTM:** Captures long-term dependencies but is prone to overfitting.
- BERT:** Outperforms simpler models by leveraging pre-trained embeddings and contextual understanding.

6. Discussion

6.1 Preprocessing

High-quality text preprocessing improves model accuracy.

6.2 Comparison of Vectorization Methods

After applying these methods, we used simple, unoptimized models to classify the data and evaluated their performance using accuracy. The results are as follows:

Vectorization Method	Model	Accuracy	Kaggle Score
Bag-of-Words	MultinomialNB	0.62	0.59
TF-IDF	MultinomialNB	0.62	0.59
Word2Vec	Logistic Regression	0.51	0.48

We found that Bag-of-Words and TF-IDF performed similarly in terms of accuracy, with comparable Kaggle scores. Word2Vec, despite being more sophisticated, underperformed, possibly due to insufficient training data or suboptimal parameter tuning.

6.3 Model Comparison

- Simpler models are suitable for rapid iteration or resource-constrained environments.
- BERT demonstrates state-of-the-art performance but requires significant computational resources and careful fine-tuning.

Challenges

- Computational cost of fine-tuning BERT.
- Handling edge cases like sarcasm and ambiguous expressions.
- Models require large amounts of training data to achieve desired performance.
- Overfitting is a common issue across models, with high validation scores but lower test scores.

Future Work

1. Explore other pre-trained models like RoBERTa or fine-tune BERT on domain-specific data.
2. Address edge cases like sarcasm and ambiguous expressions.

7. Conclusion

- Naive Bayes and Random Forest are effective for establishing baselines. Simpler models are suitable for rapid prototyping or low-resource settings.
- BERT is the best-performing model, ideal for production scenarios requiring high accuracy.
- Sentiment analysis is a complex task that requires considering factors like text length and sentiment intensity.

Full Code

See the `./codes/` files for the complete code.

Acknowledgments

- Kaggle for hosting the competition and providing the dataset.
- The authors of the Rotten Tomatoes dataset and sentiment treebank.
- The open-source community for providing libraries like PyTorch, TensorFlow, and Hugging Face Transformers.
- I extend my gratitude to other team members for their contributions: Jiaqi Han, Yiyang Zhang, Jian Ding, Sirun Li, and Ziyu Geng. Their dedication and their delicate contributions were instrumental in the success of this project..

References

1. [1] Will Cukierski. Sentiment Analysis on Movie Reviews. <https://kaggle.com/competitions/sentiment-analysis-on-movie-reviews>, 2014. Kaggle.
2. [2] Pang and L. Lee. 2005. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In ACL, pages 115–124.
3. [3] Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank, Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Chris Manning, Andrew Ng and Chris Potts. Conference on Empirical Methods in Natural Language Processing (EMNLP 2013).
4. [4] BabyGo000. 【Python数据分析】文本情感分析——电影评论分析（二）文本向量化建立模型总结与改进方向. <https://www.cnblogs.com/gc2770/p/14929162.html>, 2021, 博客园
5. [5] Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A Next-generation Hyperparameter Optimization Framework. In KDD (arXiv). [arXiv:1901.03862]