Sentiment Analysis for Movie Reviews

T3 Members:

Name	ID	Section
علي سامح سعد عبدالرحمن	20201700503	Sec.5
لوناري محمد صبري إمام	20201700619	Sec.6
يوسف احمد عمر محمد	20201701156	Sec.9
السيد مصطفى ابراهيم	20201700138	Sec.2
مصطفى محمد بيومي محمد	20201700837	Sec.7
الحسين جمعه عبدالفتاح	20201701166	Sec 2

Introduction

This project focuses on sentiment analysis specifically tailored for movie reviews. By analyzing the sentiments expressed in these reviews, we aim to provide insights into the overall reception of movies, identify trends in audience preferences, and aid users in making informed decisions about which movies to watch.

Data Preprocessing

Importing Libraries

```
import re
import spacy
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
```

- **re**: Provides regular expression matching operations, used for text cleaning.

- **spacy**: A popular NLP library used for tokenization, lemmatization, and more. Here, it is used for lemmatization.
- **nltk.tokenize.word_tokenize**: Tokenizes text into words.
- **nltk.corpus.stopwords**: Provides a set of standard stop words in English.

Loading spaCy Model

```
nlp = spacy.load('en_core_web_sm')
```

Loads the spaCy small English model, which includes functionalities like tokenization, part-of-speech tagging, and lemmatization.

Defining Custom Stop Words

```
stop_words = set(stopwords.words('english')) - {'no', 'no
t'}
```

Defines a custom set of stop words by excluding 'no' and 'not' from the standard English stop words list. This ensures that negative sentiments are preserved during preprocessing.

Defining Contraction Mapping

```
contraction_mapping = {
    "ain't": "is not",
    "aren't": "are not",
    ...
    "you're": "you are",
    "you've": "you have"
}
```

Provides a dictionary for expanding contractions in the text. This helps in standardizing the text and improving the accuracy of text analysis.

Text Preprocessing Function

```
def preprocess_text(text):
    # Convert text to lowercase
    text = text.lower()
```

```
    # Expand contractions
    text = ' '.join([contraction_mapping.get(word, word) fo
r word in text.split()])

    # Remove URLs
    text = re.sub(r'http\S+', '', text)

    # Remove non-alphabetic characters
    text = re.sub(r'[^a-zA-Z\s]', '', text)

    # Tokenize text into words
    tokens = word_tokenize(text)

    # Lemmatize tokens
    tokens = [token.lemma_ for token in nlp(" ".join(token
s))]

    # Remove stopwords
    tokens = [token for token in tokens if token not in sto
p_words]

    return ' '.join(tokens)
```

Step-by-Step Explanation

1. **Convert to Lowercase**

   ```
   text = text.lower()
   ```

   Converts all characters in the text to lowercase to ensure uniformity.

2. **Expand Contractions**

   ```
   text = ' '.join([contraction_mapping.get(word, word) for
   word in text.split()])
   ```

   Expands contractions using the defined mapping, replacing words like "can't" with "cannot".

3. **Remove URLs**

```
pythonCopy code
text = re.sub(r'http\S+', '', text)
```

   Removes any URLs from the text, which are usually irrelevant for sentiment analysis.

4. **Remove Non-Alphabetic Characters**

```
text = re.sub(r'[^a-zA-Z\s]', '', text)
```

   Removes any characters that are not alphabetic or whitespace, cleaning the text further.

5. **Tokenize Text**

```
tokens = word_tokenize(text)
```

   Splits the text into individual words (tokens).

6. **Lemmatize Tokens**

```
tokens = [token.lemma_ for token in nlp(" ".join(token
s))]
```

   Converts each token to its base form (lemma), reducing different forms of a word to a common base.

7. **Remove Stopwords**

```
tokens = [token for token in tokens if token not in stop
_words]
```

   Removes commonly used words (stopwords) that are unlikely to contribute to the sentiment of the text, while retaining 'no' and 'not'.

8. **Return Preprocessed Text**

```
return ' '.join(tokens)
```

   Joins the tokens back into a single string, providing the cleaned and standardized text.

Conclusion

The `preprocess_text` function prepares raw movie review texts for sentiment analysis by converting text to lowercase, expanding contractions, removing URLs and non-alphabetic characters, tokenizing, lemmatizing, and removing stopwords. These steps ensure that the text is in a clean and consistent format for subsequent sentiment analysis.

Loading Data

Introduction

This section outlines the process of loading and preparing the movie review dataset for sentiment analysis. The dataset consists of movie reviews categorized as positive or negative, and the objective is to preprocess the text data and organize it along with corresponding target labels for subsequent analysis.

Code Explanation

Setting Dataset Path

```
# Path to the movie review dataset
review_dataset_path = "/kaggle/input/movie-review/txt_sentoken"
```

Specifies the directory path where the movie review dataset is located.

Listing Directories in Dataset

```
print(os.listdir(review_dataset_path))
```

Lists the directories present within the dataset directory, providing an overview of the available data.

Setting Paths for Positive and Negative Reviews

```
# Path to positive and negative review folders
pos_review_folder_path = os.path.join(review_dataset_path, "pos")
```

```
neg_review_folder_path = os.path.join(review_dataset_path,
"neg")
```

Constructs the paths for the directories containing positive and negative movie reviews within the dataset.

Listing Files in Positive and Negative Review Folders

```
pos_review_file_names = os.listdir(pos_review_folder_path)
neg_review_file_names = os.listdir(neg_review_folder_path)
```

Retrieves the list of filenames present in the positive and negative review directories, facilitating access to individual review files.

Function to Load Text from Text File

```
def load_text_from_textfile(path):
    with open(path, "r") as file:
        return file.read()
```

Defines a function to read and return the contents of a text file given its path.

Function to Load Review from Text File

```
def load_review_from_textfile(path):
    return load_text_from_textfile(path)
```

Wrapper function utilizing `load_text_from_textfile` to load and return the text content of a review file.

Function to Get Data and Target Labels

```
def get_data_target(folder_path, file_names, review_type):
    data = [preprocess_text(load_review_from_textfile(os.pa
th.join(folder_path, file_name))) for file_name in file_nam
es]
    target = [review_type] * len(data)
    return data, target
```

Combines the loading and preprocessing steps to retrieve preprocessed review data along with corresponding target labels. The function takes as input the folder path, list of filenames, and the type of review (positive or negative).

Data Preparation

```
# Getting preprocessed positive and negative data along wit
h target labels
pos_data, pos_target = get_data_target(pos_review_folder_pa
th, pos_review_file_names, "positive")
neg_data, neg_target = get_data_target(neg_review_folder_pa
th, neg_review_file_names, "negative")

# Concatenating positive and negative data and targets
data = pos_data + neg_data
target_ = pos_target + neg_target

# Shuffling data and targets
combined = list(zip(data, target_))
random.shuffle(combined)
data[:], target_[:] = zip(*combined)
```

Prepares the dataset by obtaining preprocessed positive and negative review data along with their corresponding target labels. The positive and negative data are concatenated, and the labels are shuffled to ensure randomness in the dataset.

Conclusion

This section illustrates the process of loading and preparing the movie review dataset for sentiment analysis. By organizing the data into preprocessed text and target labels, we establish the foundation for training and evaluating sentiment analysis models on the movie review data. The subsequent steps will involve feature engineering, model training, and evaluation to derive insights from the sentiment analysis of movie reviews.

Text Vectorization

Introduction

Text vectorization is a crucial step in natural language processing (NLP) tasks, including sentiment analysis. It involves converting textual data into numerical representations that machine learning algorithms can process. In this section, we employ the TF-IDF (Term Frequency-Inverse Document Frequency) vectorization technique to transform the preprocessed movie review text into TF-IDF features.

Code Explanation

Importing Libraries and Splitting Data

```
from sklearn.model_selection import train_test_split
```

Imports the necessary library for splitting the dataset into training and testing sets.

```
# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data, t
arget_, test_size=0.2, random_state=42)
```

Splits the preprocessed review data (`data`) and corresponding target labels (`target_`) into training and testing sets. The training set will be used to train the sentiment analysis model, while the testing set will be used for evaluation.

Initializing TF-IDF Vectorizer

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

Imports the TF-IDF vectorizer from scikit-learn, which will be used to transform the text data into TF-IDF features.

```
# Initializing TF-IDF vectorizer
tfidf_vectorizer = TfidfVectorizer(max_features=5000)
```

Initializes the TF-IDF vectorizer with a maximum of 5000 features. This parameter controls the maximum number of unique words (features) to consider during vectorization.

Transforming Text Data into TF-IDF Features

```
# Transforming text data into TF-IDF features
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)
```

```
X_test_tfidf = tfidf_vectorizer.transform(X_test)
```

Transforms the preprocessed text data into TF-IDF features using the initialized TF-IDF vectorizer. The `fit_transform()` method is used on the training set (`X_train`), which both learns the vocabulary and transforms the text data. The `transform()` method is then applied to the testing set (`X_test`) to transform it into TF-IDF features using the vocabulary learned from the training set.

Conclusion

Text vectorization, particularly using TF-IDF, plays a crucial role in preparing textual data for sentiment analysis. By converting movie review text into numerical representations, we enable machine learning algorithms to effectively analyze and classify sentiments. The TF-IDF vectorization technique allows us to capture the importance of words in each review while considering their frequency across the entire corpus. The transformed TF-IDF features will serve as input to train and evaluate sentiment analysis models in subsequent stages of the project.

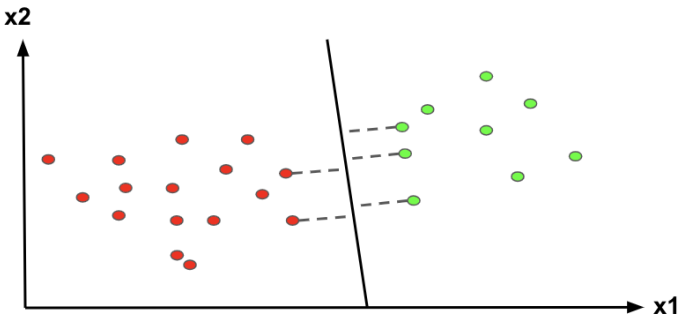
Model Training and Evaluation

Support Vector Machine (SVM) Model

Model Training

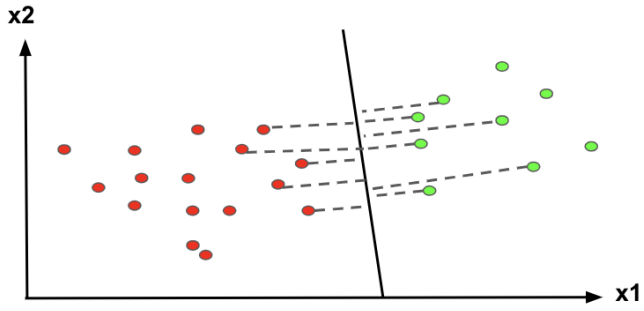
The Support Vector Machine (SVM) model is trained using various combinations of hyperparameters:

- **C**: The regularization parameter. A smaller C encourages a larger margin separating the classes, while a larger C aims to classify all training examples correctly, potentially leading to overfitting.
- **gamma**: The gamma parameter in SVMs is **a hyperparameter that controls the shape of the decision boundary.**



High Gamma

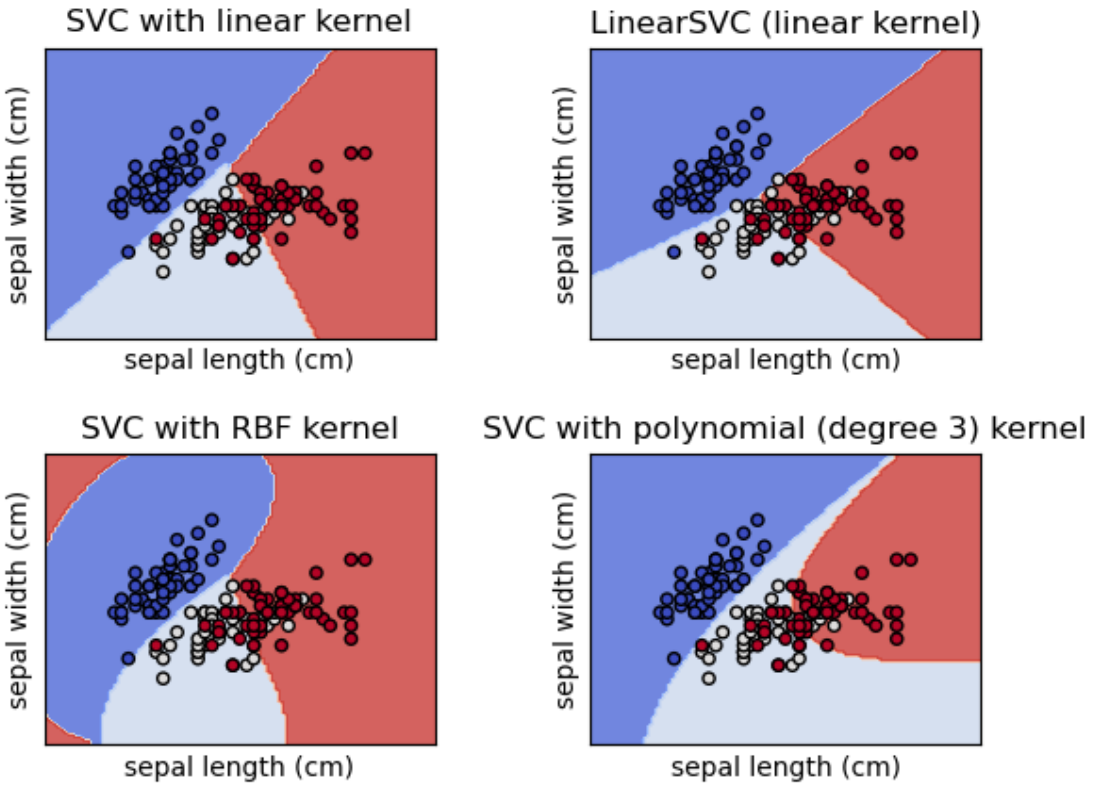
- only near points are considered.



Low Gamma

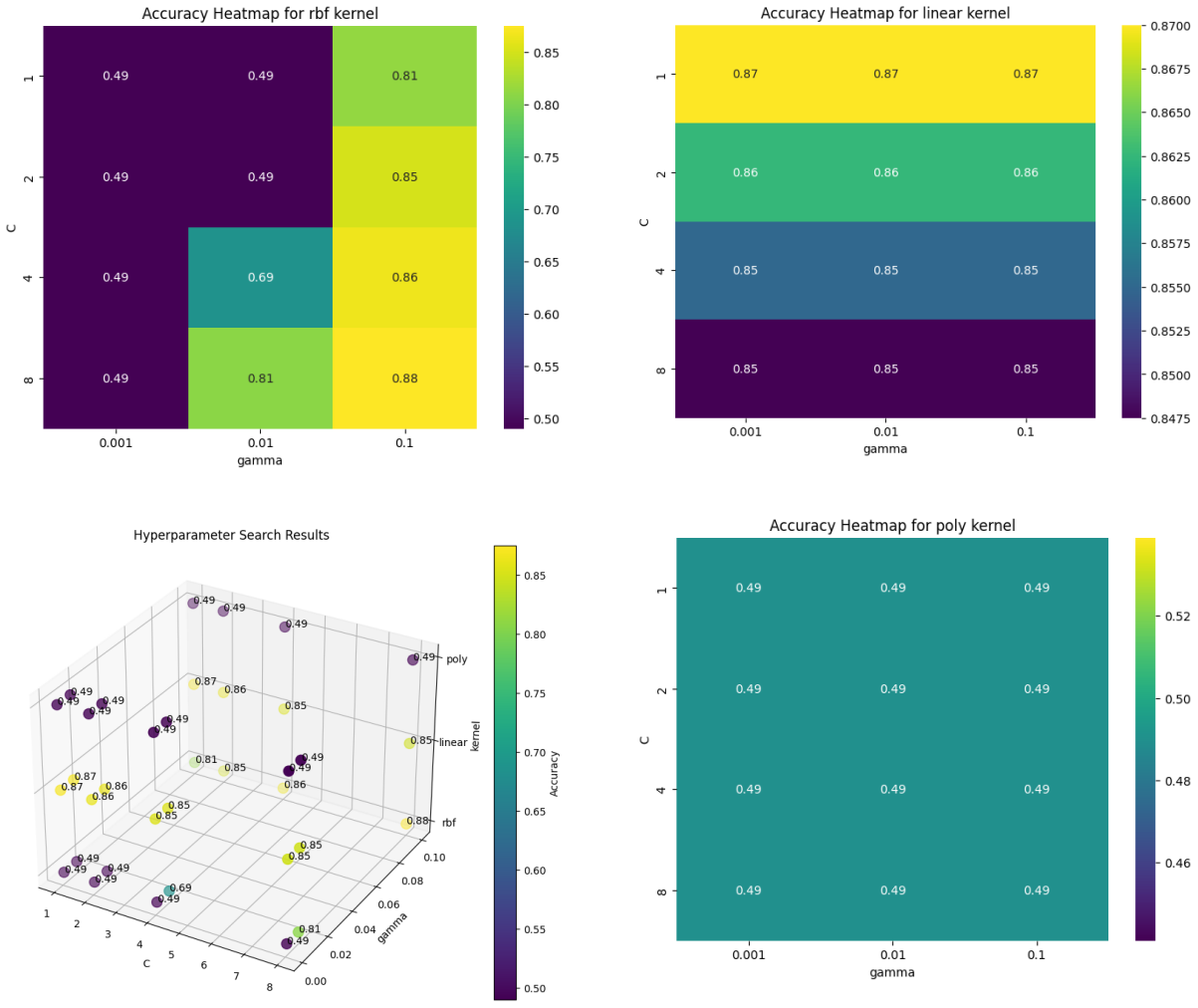
- far away points are also considered

- **kernel**: Specifies the kernel type to be used in the algorithm. The options include 'linear', 'poly', 'rbf', and 'sigmoid'. Each kernel function maps the input data into a higher-dimensional space.



Model Evaluation

The SVM model is trained on the TF-IDF features of the preprocessed movie review text. For each combination of hyperparameters, the model's accuracy is evaluated on the testing set. The best accuracy achieved was with C=8, gamma=0.1, kernel='rbf', resulting in a test accuracy of 0.88. The best model is saved for future use.



Random Forest Model

Model Training

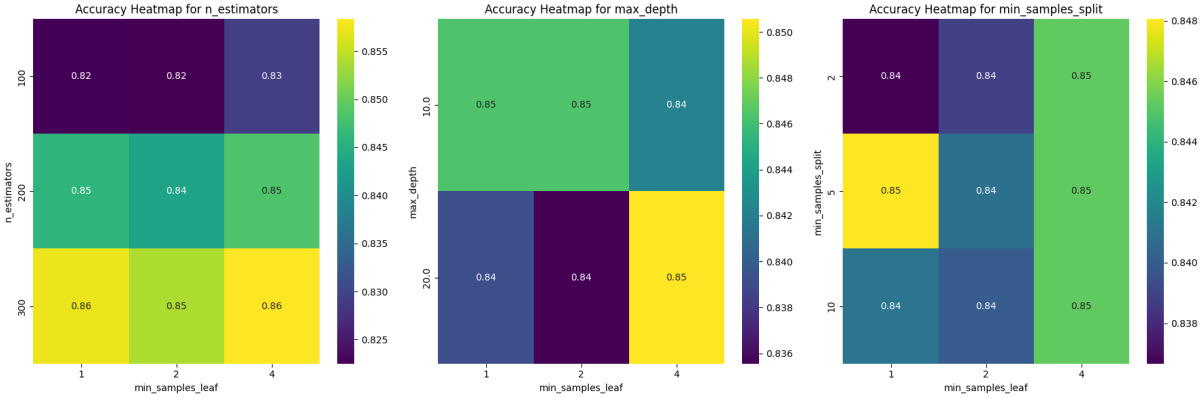
The Random Forest model is trained using different combinations of hyperparameters:

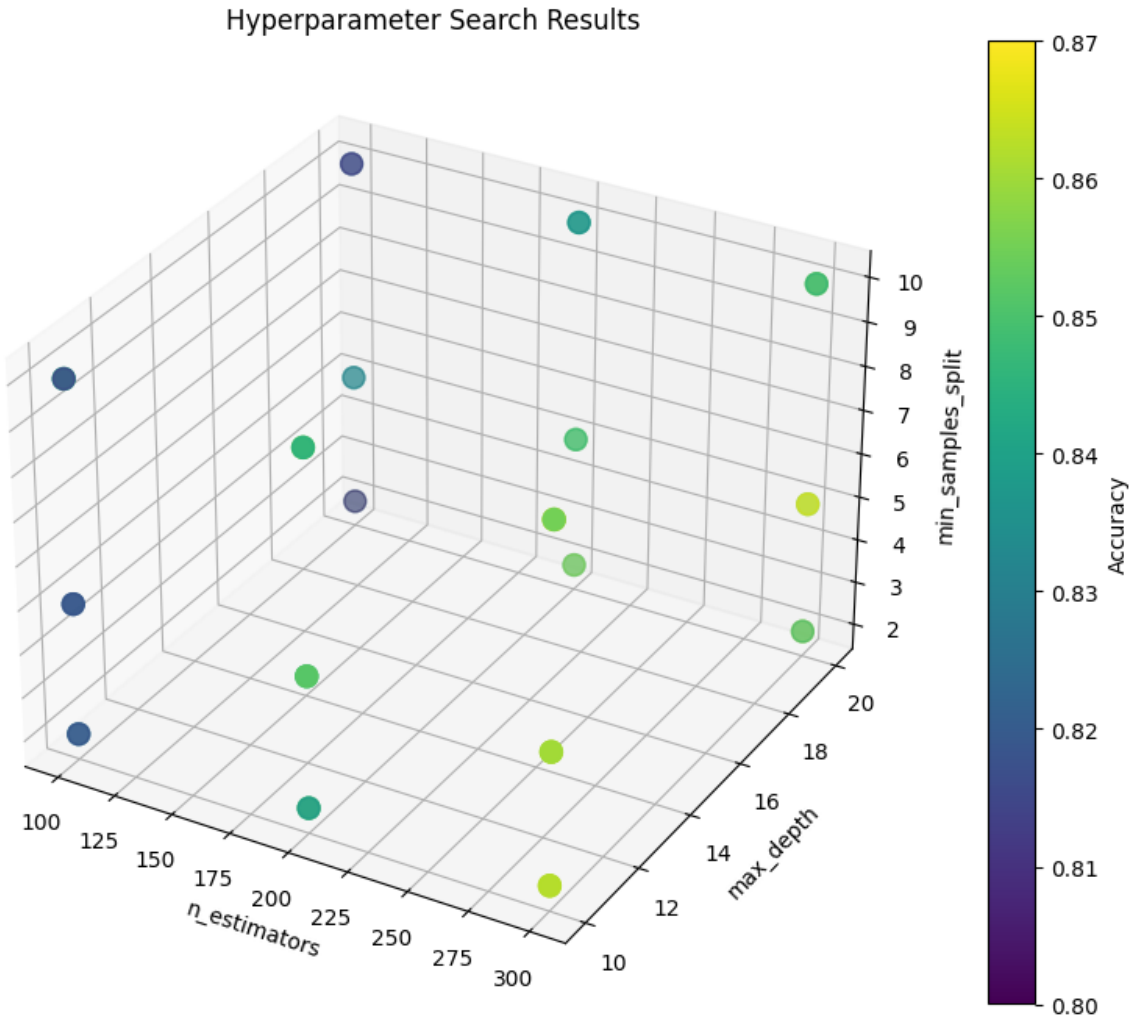
- **n_estimators**: The number of trees in the forest. More trees usually lead to better performance but also increase computational cost.

- **max_depth**: The maximum depth of each tree. Limiting the depth can prevent overfitting by controlling the complexity of the model.
- **min_samples_split**: The minimum number of samples required to split an internal node. Higher values can prevent the model from learning overly specific patterns.
- **min_samples_leaf**: The minimum number of samples required to be at a leaf node. This parameter prevents the creation of nodes with few samples, helping to avoid overfitting.

Model Evaluation

The model's accuracy is evaluated on the testing set for each hyperparameter combination. The best accuracy achieved was with n_estimators=300, max_depth=20, min_samples_split=5, min_samples_leaf=4, resulting in a test accuracy of 0.87. The best model is saved for future use.





Logistic Regression Model

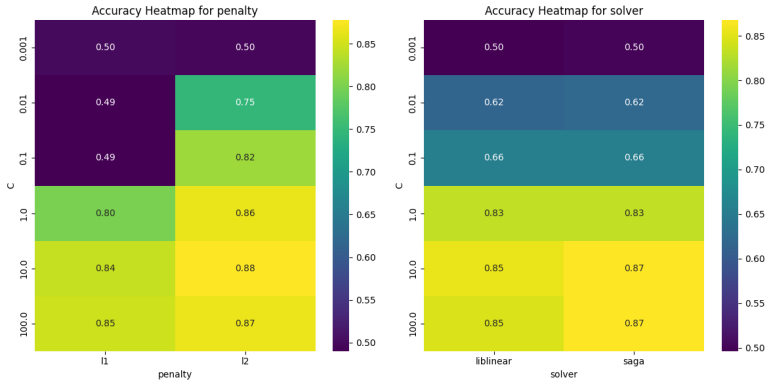
Model Training

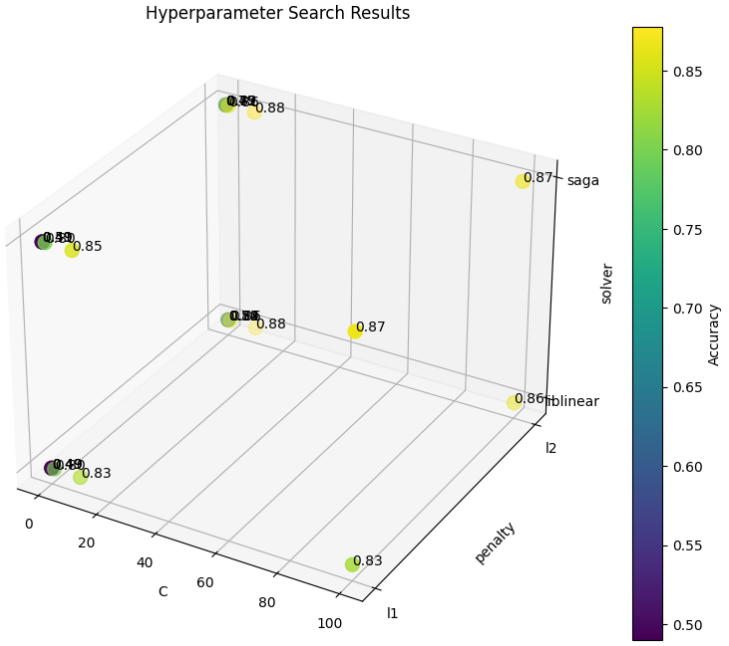
The Logistic Regression model is trained using a grid of hyperparameters:

- **C**: The inverse of regularization strength. Smaller values specify stronger regularization.
- **penalty**: The norm used in the penalization ('l1' or 'l2'). L1 regularization can lead to sparse models, while L2 tends to distribute error evenly across all terms.
- **solver**: The algorithm to use in the optimization problem. 'liblinear' is good for small datasets, while 'saga' is better for large datasets or with L1 regularization.

Model Evaluation

The accuracy of the Logistic Regression model is evaluated on the testing set for each hyperparameter combination. The best accuracy achieved was with C=10, penalty='l2', solver='liblinear', resulting in a test accuracy of 0.88. The best model is saved for future use.





XGBoost Model

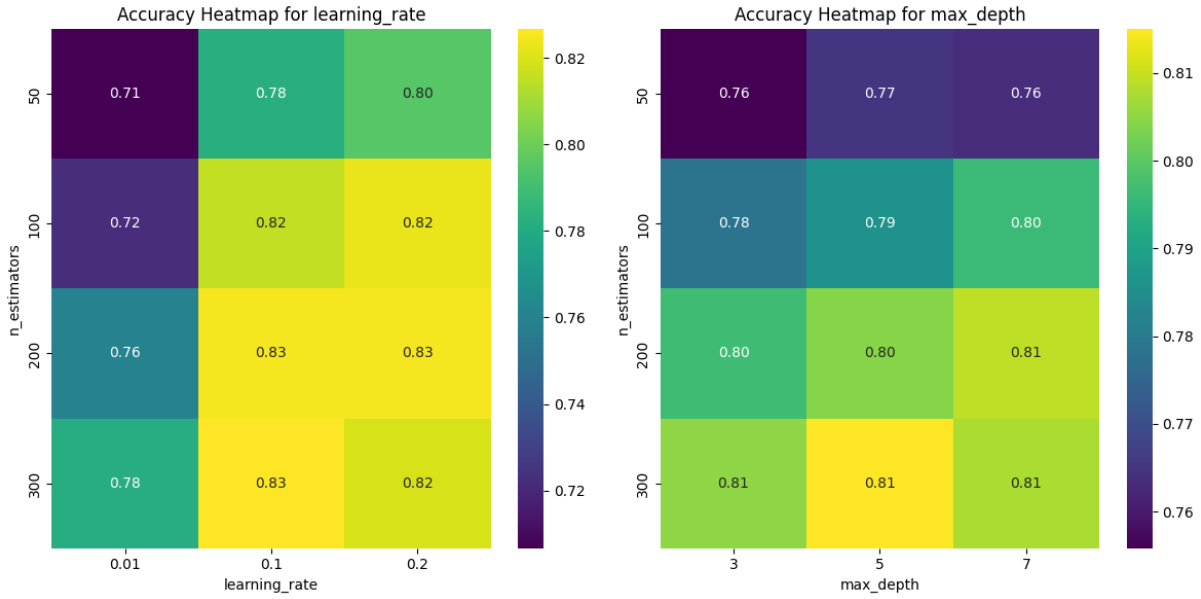
Model Training

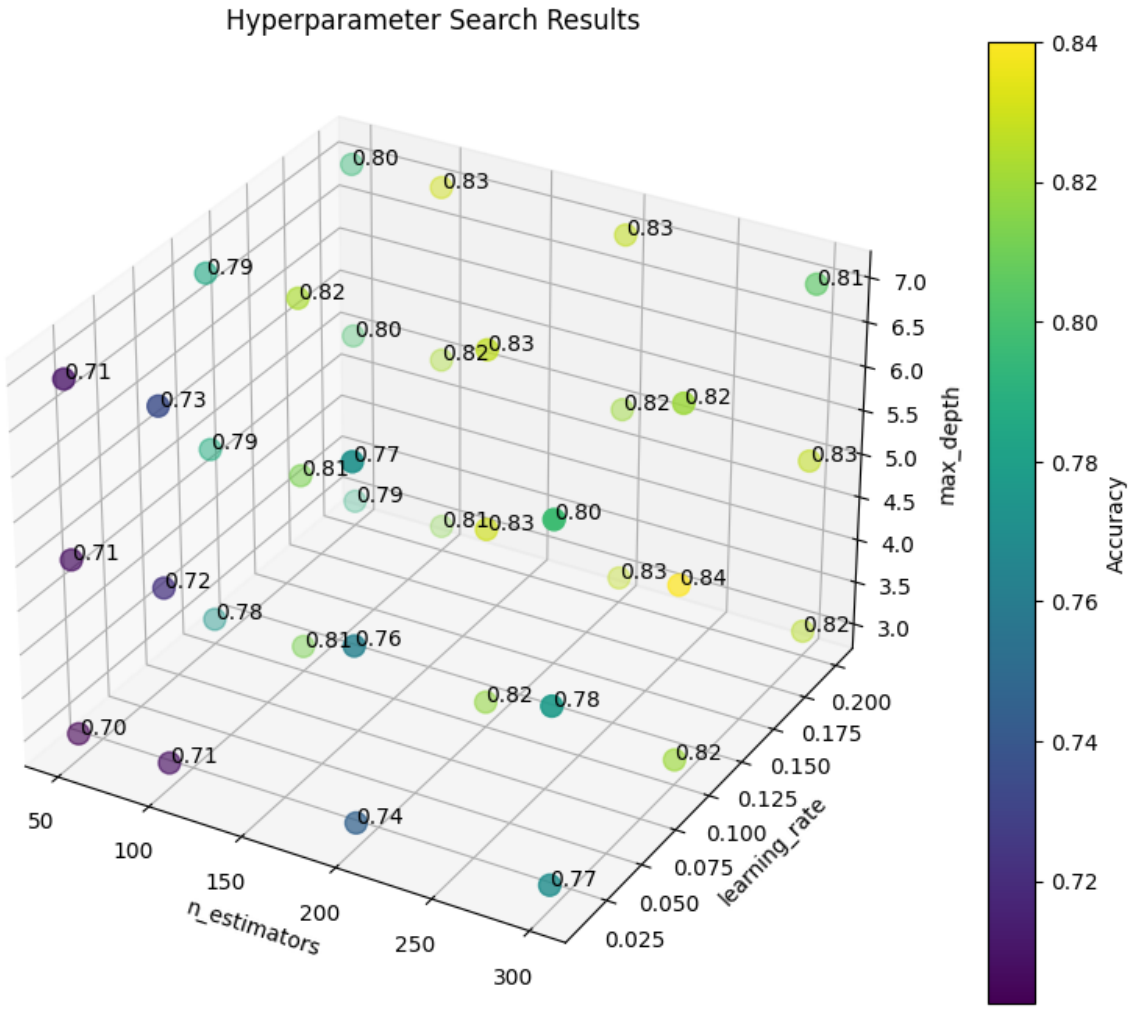
The XGBoost model is trained using different combinations of hyperparameters:

- **n_estimators**: The number of trees to fit. More trees can improve performance but increase the risk of overfitting.
- **learning_rate**: The step size shrinkage used to prevent overfitting. Smaller values make the model more robust to overfitting but require more trees.
- **max_depth**: The maximum depth of a tree. Controls the complexity of the model; deeper trees can capture more information but might overfit.

Model Evaluation

The XGBoost model's accuracy is evaluated on the testing set for each hyperparameter combination. The best accuracy achieved was with n_estimators=300, learning_rate=0.1, max_depth=5, resulting in a test accuracy of 0.84. The best model is saved for future use.





Visualization

- **Heatmaps**: Heatmaps are created to visualize the accuracy scores of the models corresponding to different hyperparameter combinations. They help identify which hyperparameters lead to the best performance.
- **3D Scatter Plots**: 3D scatter plots are generated to visualize the relationship between hyperparameters and accuracy scores. Annotations are added to each point representing the accuracy score. These visualizations provide insights into how different hyperparameters affect model performance.

Conclusion

This documentation outlines the training and evaluation process for four different machine learning models used for sentiment analysis of movie reviews. By experimenting with various hyperparameter combinations, the models' performance was optimized, achieving the highest accuracy rates for each:

- **SVM**: Best accuracy of 0.88 with C=8, gamma=0.1, kernel='rbf'.
- **Random Forest**: Best accuracy of 0.87 with n_estimators=300, max_depth=20, min_samples_split=5, min_samples_leaf=4.
- **Logistic Regression**: Best accuracy of 0.88 with C=10, penalty='l2', solver='liblinear'.
- **XGBoost**: Best accuracy of 0.84 with n_estimators=300, learning_rate=0.1, max_depth=5.

These results highlight the importance of hyperparameter tuning in improving model performance. The best models are saved for deployment and future use in sentiment analysis tasks.