AI 511 Machine Learning 2023

Assignment 2 Report

Team Name: Alchemists

Problem Statement

Work with a dataset of book reviews, and build a model that can predict the rating of a review, ranging from 0-5.

Data Cleaning and Pre-processing

Text pre-processing is a critical step in the preparation of textual data for machine learning models, particularly in natural language processing (NLP) tasks. Raw text data often contains noise, inconsistencies, and irrelevant information that can hinder the performance of machine learning algorithms. By employing text pre-processing techniques, such as lowercasing, punctuation removal, and stop word elimination, the data is refined to a more structured and uniform format. Tokenization and stemming further enhance the efficiency of the model by reducing words to their root forms and breaking down sentences into manageable units. This pre-processing not only cleanses the data but also standardizes it, ensuring that the model focuses on meaningful patterns rather than superficial variations. Additionally, the removal of stop words eliminates common language elements that carry little semantic value. In the context of predictive modelling, effective text pre-processing enhances model accuracy, reduces computational load, and contributes to the overall robustness of the machine learning system, allowing it to derive more meaningful insights from textual input.

Steps of Pre-processing performed

Selection of Relevant Columns for Predictive Modeling

n the initial phase of data preparation, a strategic decision was made to streamline the dataset for predictive modelling purposes. The specific columns selected for inclusion in the refined dataset were meticulously chosen to serve the primary objective of predicting ratings based on textual content. The curated subset of columns includes 'review_id', 'review_text', and 'rating.' The 'review_text' column holds the textual data essential for deriving insights and patterns, while 'rating' stands as the target variable to be predicted. By judiciously limiting the dataset to these pertinent columns, the focus remains sharply aligned with the task at hand—constructing a robust machine learning model for accurate and meaningful rating predictions. This deliberate column selection ensures that the model is trained and evaluated with the utmost relevance, contributing to the precision and efficiency of the predictive analytics process.

```
# Select specific columns
selected_columns = ['review_id', 'review_text', 'rating']
cleaned_df = df[selected_columns]
print(cleaned_df.head())
```

Lowercasing

We initiate the preprocessing by converting the entire 'review_text' to lowercase. This essential step establishes a standardized foundation, mitigating potential case sensitivity issues that might arise during subsequent analysis. Consistent casing is pivotal for uniform feature representation and ensures a cohesive and reliable dataset.

Removal of Punctuation and Numbers

Following lowercasing, we systematically remove punctuation and numerical characters from the text. This strategic measure serves to streamline the feature space, reducing noise and maintaining a focused dataset. By concentrating solely on alphabetic content, we enhance the clarity and quality of the text data.

Tokenization

Tokenization marks the next phase, where we break down the text into individual words or tokens. This foundational step is critical for subsequent analysis, laying the groundwork for identifying distinct linguistic elements within the text.

Removal of Stopwords

In the subsequent step, common stopwords—words with limited semantic value—are systematically removed from the text. This deliberate curation aims to accentuate more meaningful words, contributing to improved model performance and interpretability.

Stemming

Our final preprocessing step involves stemming, wherein words are reduced to their root or base form. This process standardizes the vocabulary, capturing variations of words and consolidating them to a common representation. The resulting processed text is optimized for feature extraction and model training in various natural language processing (NLP) applications.

```
# Preprocess the 'review_text' column
def preprocess_text(text):
    # Convert to lowercase
    text = text.lower()
    # Remove punctuation and numbers
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    # Tokenization
    tokens = word_tokenize(text)
    # Remove stopwords
    stop_words = set(stopwords.words('english'))
```

```
    tokens = [word for word in tokens if word not in stop_words]
    # Stemming (you can use lemmatization instead if you prefer)
    stemmer = PorterStemmer()
    tokens = [stemmer.stem(word) for word in tokens]
    # Join the tokens back into a string
    processed_text = ' '.join(tokens)
    return processed_text

# Apply the preprocessing function to the 'review_text' column
cleaned_df['review_text'] =
cleaned_df['review_text'].apply(preprocess_text)
```

This systematic and comprehensive preprocessing approach ensures that the 'review_text' data is meticulously refined for subsequent machine learning tasks. Each preprocessing step is carefully selected to address specific challenges related to case sensitivity, noise, and semantic relevance, ultimately enhancing the text data's suitability for analysis and modeling.

Handling Missing Values

The presence of missing values can introduce uncertainties into subsequent analyses and modeling endeavours. In our pursuit of a comprehensive and resilient dataset, we employ a strategic approach to handle these potential gaps.

Our methodology involves the judicious use of the **fillna** method, allowing us to replace missing values with a standardized empty string (''). This pragmatic choice not only safeguards the structural integrity of our dataset but also serves as a neutral placeholder for absent text data. The adoption of a uniform empty string representation aligns with our commitment to data consistency and aids in the seamless integration of the dataset into subsequent stages of our analytical pipeline.

```
# Handle missing values in 'review_text' for both training and testing sets
df_subset['review_text'].fillna('', inplace=True)
```

Logistic Regression Model Training

Logistic regression, a fundamental algorithm in the realm of machine learning, serves as a cornerstone for binary classification tasks. This essay delves into the intricacies of training a logistic regression model, unravelling the step-by-step process that transforms raw data into a predictive tool.

Data Preprocessing and Feature Engineering: The journey begins with meticulous data preprocessing, where the dataset is dissected into features (X) and a target variable (y). In our context, the 'review_text' column assumes the role of features, while the 'rating' column, having undergone transformation into a binary representation, becomes the target variable. This initial stage sets the foundation for subsequent model development, laying bare the essential elements required for effective learning.

Stratification: To ensure robust model evaluation, the dataset undergoes a strategic division into training and testing sets. This process, known as stratification, safeguards against biases by preserving the distribution of the target variable in both subsets. Incorporating a random seed adds a layer of reproducibility, a crucial aspect of scientific rigor, ensuring that the same results can be obtained across different executions.

Handling Missing Values: An often overlooked yet critical aspect of data preparation involves addressing missing values. Focused on the 'review_text' column, the meticulous use of the **fillna** method ensures a seamless flow of information. This preemptive measure safeguards against potential disruptions during model training, enhancing the reliability and resilience of the predictive model.

Model Training and Evaluation: With a preprocessed and stratified dataset at our disposal, the logistic regression model steps into the spotlight. The training process involves a delicate interplay between the features and target variable, gradually fine-tuning the model's parameters to discern patterns and relationships within the data. The evaluation of the model's performance on a distinct testing set provides valuable insights into its ability to generalize to unseen data.

Iterative Refinement: The journey of logistic regression model training is rarely a linear path. It involves an iterative refinement process, where model parameters may be adjusted, features reconsidered, and hyperparameters fine-tuned to enhance predictive accuracy. This iterative nature is a testament to the dynamic and adaptive quality of the model-building process.

Making test and train split

First we undertake a pivotal task of partitioning our dataset into distinct features (X) and the corresponding target variable (y). This division forms the bedrock of our analytical framework, enabling us to systematically delineate the predictive features from the outcome variable. Specifically, we isolate the 'review_text' column as our feature set (X) and the 'rating' column as our target variable (y), assuming 'rating' to be an integer within the range of 0 to 5.

In the interest of data consistency and compatibility with modeling algorithms, we judiciously convert the 'rating' column to integers using the **astype** method. This step ensures a uniform numeric representation for our target variable, laying the groundwork for a seamless integration into subsequent modeling endeavors.

Further refining our dataset preparation, we embark on the crucial step of splitting our data into distinct training and testing sets. Leveraging the **train_test_split** function, we allocate a specified portion (in this case, 20%) of our dataset to serve as the testing set, facilitating robust model evaluation. The integration of a random seed (random_state=42) enhances reproducibility, providing a consistent basis for model assessment.

Additionally, we conscientiously address potential missing values in the 'review_text' column for both the training and testing sets. The utilization of the **fillna** method ensures that any missing values are replaced with a standardized empty string (''), preserving the structural integrity of our text data.

```
# Handle missing values in 'review_text' for both training and testing sets
df_subset['review_text'].fillna('', inplace=True)

# Split the data into features (X) and target (y)
X = df_subset['review_text']
y = df_subset['rating']  # Assuming 'rating' is an integer between 0 and 5

# Convert the rating to integers (if it's not already)
y = y.astype(int)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Explicitly handle missing values in 'review_text' for both training and
testing sets
X_train = X_train.fillna('')
X_test = X_test.fillna('')
```

Text Vectorization Using TF-IDF

In this segment, we employ the TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer to convert the textual data into a numerical format suitable for machine learning models. Utilizing the TfidfVectorizer from scikit-learn, we limit the maximum number of features to 5000 to manage computational complexity and memory usage.

- **TF-IDF Vectorizer Initialization:** We create an instance of the TfidfVectorizer with a specified maximum feature limit.
- **Training Set Vectorization:** The fit_transform method is applied to the training set (X_train), transforming the raw text into TF-IDF-weighted features. This process calculates the importance of each term in relation to the entire corpus.
- **Testing Set Vectorization:** Using the transform method, we apply the previously fitted vectorizer to the testing set (X_test). This ensures consistency in the representation of text across both training and testing datasets.

This transformation prepares our textual data for integration into machine learning models, allowing for meaningful analysis and prediction.

```
vectorizer = TfidfVectorizer(max_features=5000)
X_train_vectorized = vectorizer.fit_transform(X_train)
X_test_vectorized = vectorizer.transform(X_test)
```

Logistic Regression Model Training

In this phase, we construct and train a logistic regression model using the processed and vectorized textual data. Logistic regression is a widely used algorithm for binary and multiclass classification tasks.

- **Model Initialization:** We instantiate a logistic regression model using the LogisticRegression class from scikit-learn. The hyperparameter 'C' controls the inverse of the regularization strength, 'max_iter' determines the maximum number of iterations for convergence, and 'n_jobs' enables parallel processing for faster computation.
- **Model Training:** The fit method is applied to the model, utilizing the TF-IDF vectorized training set (X_train_vectorized) and the corresponding target variable (y_train). This step involves the optimization of the model's parameters to learn the underlying patterns in the data.

```
lr = LogisticRegression(C= 2, max_iter = 1000, n_jobs=-1)
lr.fit(X_train_vectorized, y_train)
```

Model Evaluation and Prediction

Following the training of the logistic regression model, the next crucial step involves making predictions on the previously unseen test set and evaluating the model's performance.

- **Prediction:** Utilizing the predict method, the model generates predictions (y_pred) based on the TF-IDF vectorized test set (X_test_vectorized).
- **Model Evaluation Metrics:** The accuracy_score function is employed to quantify the accuracy of the model by comparing its predictions (y_pred) with the actual target values (y_test). The accuracy score provides a straightforward measure of the model's overall correctness in its predictions.
- **Classification Report:** The classification_report function offers a comprehensive breakdown of the model's performance across multiple metrics such as precision, recall, and F1-score for each class. This report provides valuable insights into the model's behavior for different classes and aids in understanding its strengths and weaknesses.

Users are encouraged to interpret these metrics collectively to gauge the logistic regression model's effectiveness in predicting the ratings based on the processed textual data.

```
# Make predictions on the test set
y_pred = lr.predict(X_test_vectorized)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Print classification report
print("Classification Report:\n", classification_report(y_test, y_pred))
```

Classification Report

```
Accuracy: 0.5233968253968254
Classification Report:
              precision    recall  f1-score   support

           0       0.51      0.28      0.36      4327
           1       0.46      0.29      0.36      4035
           2       0.42      0.28      0.34     10187
           3       0.46      0.41      0.44     26590
           4       0.49      0.61      0.54     43754
           5       0.63      0.63      0.63     37107
```

```
    accuracy                           0.52    126000
   macro avg       0.50      0.42      0.44    126000
weighted avg       0.52      0.52      0.52    126000
```

Our model is giving an accuracy of 0.5233968253968254.

Hyperparameter Tuning for Logistic Regression

In the pursuit of refining our logistic regression model, we turn to the realm of hyperparameter tuning using the GridSearchCV technique. This method allows us to systematically explore a predefined hyperparameter space and identify the optimal configuration for enhancing model performance.

- **Hyperparameter Space:** The grid defined by 'param_grid' spans potential values for the regularization parameter 'C' and the maximum number of iterations 'max_iter.' The provided values are mere suggestions, and users can tailor them to align with their preferences and the specific characteristics of their dataset.
- **Logistic Regression Model:** We instantiate a logistic regression model, denoted as 'lr,' setting the stage for the subsequent hyperparameter tuning process.
- **GridSearchCV:** Leveraging the GridSearchCV module from scikit-learn, we initiate the hyperparameter search. The cross-validation parameter 'cv' is set to 5, indicating a five-fold cross-validation strategy. The 'scoring' metric is specified as 'accuracy,' although users can adapt this metric based on their evaluation criteria.
- **Best Hyperparameters:** Following the completion of the grid search, the best hyperparameters are extracted using the 'best_params_' attribute. These values represent the configuration that yielded optimal performance during the tuning process.
- **Evaluation on Test Set:** To assess the model's generalization capabilities, we evaluate the logistic regression model using the best hyperparameters on the designated test set. The achieved accuracy is reported as a performance metric.

Note: This hyperparameter tuning process is conducted during the model development phase to identify the most effective parameter configuration. The final model training employs the best parameters obtained from this tuning phase for optimal predictive performance.

```
#Use of hyper parameter tuning
from sklearn.model_selection import GridSearchCV
# Build and train a Logistic Regression model with hyperparameter tuning
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Adjust the range based on your
preference
    'max_iter': [50, 100, 200, 500]  # Adjust the range based on your
preference
}

# Create a logistic regression model
lr = LogisticRegression(n_jobs=-1)

# Use GridSearchCV for hyperparameter tuning
grid_search = GridSearchCV(lr, param_grid, cv=5, scoring='accuracy')  #
Adjust the scoring metric as needed
grid_search.fit(X_train_scaled, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)

# Evaluate the model on the test set using the best hyperparameters
best_lr = grid_search.best_estimator_
accuracy = best_lr.score(X_test_scaled, y_test)
print("Accuracy on Test Set:", accuracy)
```