SENTIMENT ANALYSIS

Table of Content

|  |  |
| --- | --- |
| Sl.No | Topics |
| 1 | Introduction |
| 1.1 | Purpose of the Project |
| 1.2 | Goals and Objectives |
| 2 | Data Exploration |
| 2.1 | Dataset Overview |
| 2.2 | Data Cleaning |
| 3 | Data Preprocessing |
| 3.1 | Text Preprocessing |
| 3.2 | Example Preprocessing |
| 4 | Exploratory Data Analysis (EDA) |
| 5 | Text Vectorization |
| 6 | Model Selection |
| 6.1 | Naive Bayes |
| 6.2 | Support Vector Machines (SVM) |
| 6.3 | LSTM (Long Short-Term Memory) |
| 7 | Hyperparameter Tuning |
| 8 | Cross-Validation |
| 9 | Model Interpretability |
| 10 | Evaluation Metrics |

1. Introduction

Sentiment analysis involves determining the sentiment expressed in a piece of text, making it a valuable tool for understanding public opinion. This documentation outlines the process of sentiment analysis using various techniques, including Natural Language Processing (NLP) and machine learning algorithms.

1.1 Purpose Of The Project

The project aims to analyze sentiments in textual data, categorizing them as positive, neutral, or negative. This can provide insights into public opinion, customer feedback, and social media sentiment.

1.2 Goals and Objectives

* Conduct thorough data exploration to understand the dataset.
* Preprocess the text data to prepare it for analysis.
* Explore sentiment distribution through exploratory data analysis (EDA).
* Implement text vectorization techniques such as TF-IDF and Word2Vec.
* Evaluate different machine learning models, including Naive Bayes, Support Vector Machines (SVM), and LSTM.
* Optimize model hyperparameters for improved performance.
* Perform cross-validation to assess model generalization.
* Utilize model interpretability techniques, such as LimeTextExplainer.
* Evaluate models using various metrics, including accuracy, precision, recall, and F1 score.

2. Data Exploration

2.1 Dataset Overview

The dataset, sourced from 'data.csv' is explored in terms of size, columns, and general characteristics.

# Sample code for data exploration

import pandas as pd

df = pd.read\_csv('data.csv', encoding=encoding)

print(df.shape)

print(df.columns)

print(df.head())

print(df.describe())

print(df.info())

print(df.isnull().sum())

2.2 Data Cleaning

Handle missing values and prepare the dataset for analysis.

# Sample code for data cleaning

df.dropna(inplace=True)

3. Data Preprocessing

3.1 Text Preprocessing

Prepare the text data for analysis through lowercase conversion, removal of non-alphabetic characters, tokenization, removal of stopwords, and lemmatization.

# Sample code for text preprocessing

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

from nltk.stem import WordNetLemmatizer

nltk.download('punkt')

nltk.download('stopwords')

nltk.download('wordnet')

stop\_words = set(stopwords.words('english'))

lemmatizer = WordNetLemmatizer()

def preprocess\_text(text):

text = text.lower()

text = ''.join([char for char in text if char.isalpha() or char.isspace()])

tokens = word\_tokenize(text)

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

tokens = [lemmatizer.lemmatize(token) for token in tokens]

return ' '.join(tokens)

df['preprocessed\_text'] = df['text'].apply(preprocess\_text)

3.2 Example Preprocessing

Show an example of text preprocessing on a sample text.

# Example text preprocessing

text = "Hi how are you doing ??? \*just joined twitter..."

preprocessed\_text = preprocess\_text(text)

print(preprocessed\_text)

The preprocessing step aims to make the text more suitable for analysis by reducing noise and standardizing the format.

4. Exploratory Data Analysis (EDA)

Explore sentiment distribution through visualizations.

# Sample code for sentiment distribution visualization

import matplotlib.pyplot as plt

import seaborn as sns

sentiment\_counts = df['sentiment'].value\_counts()

plt.figure(figsize=(8, 6))

sns.barplot(x=sentiment\_counts.index, y=sentiment\_counts.values, palette="viridis")

plt.title('Sentiment Distribution')

plt.xlabel('Sentiment')

plt.ylabel('Count')

plt.show()

plt.figure(figsize=(8, 8))

plt.pie(sentiment\_counts, labels=sentiment\_counts.index, autopct='%1.1f%%', startangle=90, colors=sns.color\_palette("viridis", len(sentiment\_counts)))

plt.title('Sentiment Distribution')

plt.show()

Interpret visualizations to gain insights into the sentiment distribution.

5. Text Vectorization

Implement TF-IDF and Word2Vec for text vectorization.

# Sample code for TF-IDF vectorization

from sklearn.feature\_extraction.text import TfidfVectorizer

tfidf\_vectorizer = TfidfVectorizer(max\_features=5000)

tfidf\_matrix = tfidf\_vectorizer.fit\_transform(df['preprocessed\_text'])

tfidf\_df = pd.DataFrame(tfidf\_matrix.toarray(), columns=tfidf\_vectorizer.get\_feature\_names\_out())

# Sample code for Word2Vec embedding

from gensim.models import Word2Vec

tokenized\_text = df['preprocessed\_text'].apply(word\_tokenize)

word2vec\_model = Word2Vec(sentences=tokenized\_text, vector\_size=100, window=5, min\_count=1, workers=4)

word2vec\_vectors = tokenized\_text.apply(lambda x: np.mean([word2vec\_model.wv[word] for word in x], axis=0))

word2vec\_df = pd.DataFrame(word2vec\_vectors.tolist(), columns=[f'w2v\_{i}' for i in range(100)])

6. Model Selection

6.1 Naive Bayes

Train and evaluate a Naive Bayes classifier.

# Sample code for Naive Bayes

from sklearn.model\_selection import train\_test\_split

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.naive\_bayes import MultinomialNB

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['preprocessed\_text'], df['sentiment'], test\_size=0.2, random\_state=42)

tfidf\_vectorizer = TfidfVectorizer(max\_features=5000)

X\_train\_tfidf = tfidf\_vectorizer.fit\_transform(X\_train)

X\_test\_tfidf = tfidf\_vectorizer.transform(X\_test)

nb\_classifier = MultinomialNB()

nb\_classifier.fit(X\_train\_tfidf, y\_train)

nb\_predictions = nb\_classifier.predict(X\_test\_tfidf)

# Evaluation metrics

print("Naive Bayes Metrics:")

print("Accuracy:", accuracy\_score(y\_test, nb\_predictions))

print("Precision:", precision\_score(y\_test, nb\_predictions, average='weighted'))

print("Recall:", recall\_score(y\_test, nb\_predictions, average='weighted'))

print("F1 Score:", f1\_score(y\_test, nb\_predictions, average='weighted'))

print("\nClassification Report:\n", classification\_report(y\_test, nb\_predictions))

6.2 Support Vector Machines (SVM)

Train and evaluate an SVM classifier.

# Sample code for SVM

from sklearn.svm import SVC

svm\_classifier = SVC()

svm\_classifier.fit(X\_train\_tfidf, y\_train)

svm\_predictions = svm\_classifier.predict(X\_test\_tfidf)

# Evaluation metrics

print("\nSupport Vector Machines Metrics:")

print("Accuracy:", accuracy\_score(y\_test, svm\_predictions))

print("Precision:", precision\_score(y\_test, svm\_predictions, average='weighted'))

print("Recall:", recall\_score(y\_test, svm\_predictions, average='weighted'))

print("F1 Score:", f1\_score(y\_test, svm\_predictions, average='weighted'))

print("\nClassification Report:\n", classification\_report(y\_test, svm\_predictions))

6.3 LSTM (Long Short-Term Memory)

Train and evaluate an LSTM model using TensorFlow/Keras.

# Sample code for LSTM

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, LSTM, Dense

tokenizer = Tokenizer()

tokenizer.fit\_on\_texts(df['preprocessed\_text'])

X\_seq = tokenizer.texts\_to\_sequences(df['preprocessed\_text'])

X\_pad = pad\_sequences(X\_seq, maxlen=100)

label\_mapping = {'positive': 0, 'neutral': 1, 'negative': 2}

y\_labels = df['sentiment'].map(label\_mapping)

X\_train\_lstm, X\_test\_lstm, y\_train\_lstm, y\_test\_lstm = train\_test\_split(X\_pad, y\_labels, test\_size=0.2, random\_state=42)

lstm\_model = Sequential()

lstm\_model.add(Embedding(input\_dim=len(tokenizer.word\_index) + 1, output\_dim=100, input\_length=100))

lstm\_model.add(LSTM(100))

lstm\_model.add(Dense(3, activation='softmax'))

lstm\_model.compile(loss='sparse\_categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

lstm\_model.fit(X\_train\_lstm, y\_train\_lstm, epochs=5, batch\_size=32, validation\_data=(X\_test\_lstm, y\_test\_lstm))

# Ensure the model is Sequential

if not isinstance(lstm\_model, Sequential):

raise TypeError("lstm\_model is not a Sequential model.")

lstm\_predictions = np.argmax(lstm\_model.predict(X\_test\_lstm), axis=1)

# Evaluation metrics

print("\nLSTM Metrics:")

print("Accuracy:", accuracy\_score(y\_test\_lstm, lstm\_predictions))

print("Precision:", precision\_score(y\_test\_lstm, lstm\_predictions, average='weighted'))

print("Recall:", recall\_score(y\_test\_lstm, lstm\_predictions, average='weighted'))

print("F1 Score:", f1\_score(y\_test\_lstm, lstm\_predictions, average='weighted'))

print("\nClassification Report:\n", classification\_report(y\_test\_lstm, lstm\_predictions))

7. Hyperparameter Tuning

Perform hyperparameter tuning using GridSearchCV.

# Sample code for hyperparameter tuning

from sklearn.model\_selection import GridSearchCV

from sklearn.svm import SVC

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.pipeline import Pipeline

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['preprocessed\_text'], df['sentiment'], test\_size=0.2, random\_state=42)

pipeline = Pipeline([

('tfidf', TfidfVectorizer()),

('svm', SVC())

])

param\_grid = {

'tfidf\_\_max\_features': [5000, 10000],

'svm\_\_C': [1, 10, 100],

'svm\_\_kernel': ['linear', 'rbf'],

}

grid\_search = GridSearchCV(pipeline, param\_grid, cv=5, scoring='accuracy', n\_jobs=-1)

grid\_search.fit(X\_train, y\_train)

best\_params = grid\_search.best\_params\_

best\_model = grid\_search.best\_estimator\_

predictions = best\_model.predict(X\_test)

# Display best hyperparameters and evaluation metrics

print("Best Hyperparameters:", best\_params)

print("\nOptimized SVM Metrics:")

print("Accuracy:", accuracy\_score(y\_test, predictions))

print("Precision:", precision\_score(y\_test, predictions, average='weighted'))

print("Recall:", recall\_score(y\_test, predictions, average='weighted'))

print("F1 Score:", f1\_score(y\_test, predictions, average='weighted'))

print("\nClassification Report:\n", classification\_report(y\_test, predictions))

8. Cross-Validation

Perform cross-validation using StratifiedKFold and assess model performance.

# Sample code for cross-validation

from sklearn.model\_selection import cross\_val\_score, StratifiedKFold

from sklearn.svm import SVC

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.pipeline import Pipeline

from sklearn.metrics import make\_scorer, accuracy\_score, precision\_score, recall\_score, f1\_score

pipeline = Pipeline([

('tfidf', TfidfVectorizer()),

('svm', SVC())

])

cv\_strategy = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

scoring\_metrics = {

'accuracy': make\_scorer(accuracy\_score),

'precision': make\_scorer(precision\_score, average='weighted'),

'recall': make\_scorer(recall\_score, average='weighted'),

'f1\_score': make\_scorer(f1\_score, average='weighted')

}

cv\_results = cross\_validate(pipeline, df['preprocessed\_text'], df['sentiment'], cv=cv\_strategy, scoring=scoring\_metrics, n\_jobs=-1)

# Print the cross-validation results

print(cv\_results)

print("Cross-Validation Results:")

print("Accuracy:", cv\_results['test\_accuracy'].mean())

print("Precision:", cv\_results['test\_precision'].mean())

print("Recall:", cv\_results['test\_recall'].mean())

print("F1 Score:", cv\_results['test\_f1\_score'].mean())

9. Model Interpretability

Utilize Lime for model interpretability, focusing on an SVM classifier.

# Sample code for LimeTextExplainer

!pip install lime

from lime import lime\_text

from lime.lime\_text import LimeTextExplainer

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['preprocessed\_text'], df['sentiment'], test\_size=0.2, random\_state=42)

tfidf\_vectorizer = TfidfVectorizer(max\_features=5000)

X\_train\_tfidf = tfidf\_vectorizer.fit\_transform(X\_train)

svm\_classifier = SVC(probability=True)

svm\_classifier.fit(X\_train\_tfidf, y\_train)

explainer = LimeTextExplainer(class\_names=df['sentiment'].unique())

instance\_index = 0

text\_instance = X\_test.iloc[instance\_index]

true\_label = y\_test.iloc[instance\_index]

# Display information about the instance

print(text\_instance)

print(type(text\_instance))

# Explain the prediction

exp = explainer.explain\_instance(text\_instance, svm\_classifier.predict\_proba, num\_features=10)

print(exp.as\_list())

10. Evaluation Metrics

Evaluate the models using confusion matrices, precision, and recall.

# Sample code for evaluation metrics

from sklearn.metrics import confusion\_matrix

import seaborn as sns

import matplotlib.pyplot as plt

y\_true = [0, 1, 1, 0, 1, 0, 1, 0, 1, 1] # Example, replace with your actual labels

y\_scores = [0.2, 0.8, 0.6, 0.3, 0.9, 0.1, 0.7, 0.4, 0.95, 0.75] # Example, replace with your predicted scores

# Display confusion matrix

conf\_matrix = confusion\_matrix(y\_true, (np.array(y\_scores) > 0.5).astype(int))

sns.heatmap(conf\_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=['Negative', 'Positive'], yticklabels=['Negative', 'Positive'])

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix for Binary Sentiment Classification')

plt.show()

# Sample code for classification metrics

from sklearn.feature\_extraction.text import CountVectorizer

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['preprocessed\_text'], df['sentiment'], test\_size=0.2, random\_state=42)

vectorizer = CountVectorizer()

X\_train\_vectorized = vectorizer.fit\_transform(X\_train)

X\_test\_vectorized = vectorizer.transform(X\_test)

classifier = MultinomialNB()

classifier.fit(X\_train\_vectorized, y\_train)

y\_pred = classifier.predict(X\_test\_vectorized)

# Display precision and recall

print(precision\_score(y\_test, y\_pred))

print(recall\_score(y\_test, y\_pred))

Conclusion And Takeaways

In conclusion, this sentiment analysis project provided valuable insights into the sentiment distribution within textual data. By leveraging a combination of traditional machine learning models and deep learning techniques, we successfully navigated through data exploration, preprocessing, model selection, and evaluation. The project not only demonstrated the practical implementation of sentiment analysis but also highlighted areas for improvement and future exploration.

As we continue to evolve in the field of natural language processing, sentiment analysis remains a pivotal tool for understanding and responding to the sentiments expressed in diverse textual data sources.

Key Findings

Data Exploration: We began by exploring the dataset, understanding its structure, and cleaning it to ensure high-quality analysis. The dataset, sourced from 'data.csv,' underwent preprocessing to handle missing values and prepare it for further analysis.

Text Preprocessing: Text preprocessing was a crucial step in preparing the textual data for analysis. Techniques such as lowercase conversion, removal of non-alphabetic characters, tokenization, removal of stopwords, and lemmatization were applied to enhance the quality of the text.

Exploratory Data Analysis (EDA): We visually examined the distribution of sentiments within the dataset using bar plots and pie charts. These visualizations provided valuable insights into the overall sentiment distribution, aiding in a better understanding of the data.

Text Vectorization: Two text vectorization techniques, TF-IDF and Word2Vec, were implemented to convert the preprocessed text into numerical forms suitable for machine learning models.

Model Selection: We evaluated the performance of three different models: Naive Bayes, Support Vector Machines (SVM), and Long Short-Term Memory (LSTM). Each model was trained, validated, and assessed using various metrics such as accuracy, precision, recall, and F1 score.

Hyperparameter Tuning: To optimize model performance, hyperparameter tuning was performed using GridSearchCV. The best hyperparameters were identified to enhance the models' predictive capabilities.

Cross-Validation: The models' generalization capabilities were assessed through cross-validation, providing a more robust evaluation of their performance.

Model Interpretability: LimeTextExplainer was utilized to interpret the predictions of the Support Vector Machines (SVM) model, shedding light on the features influencing the model's decisions.

Evaluation Metrics: We employed confusion matrices, precision, and recall to evaluate the models' performance. Additionally, we explored the use of CountVectorizer for binary sentiment classification.

Lessons Learned and Future Work

Imbalanced Classes: Future iterations of the project may consider addressing imbalanced class issues to enhance the models' ability to predict minority classes accurately.

Advanced Model Architectures: Exploring more sophisticated model architectures, such as deep learning models with attention mechanisms, could further improve sentiment analysis accuracy.

Enhanced Feature Engineering: Continuous refinement of text preprocessing and feature engineering techniques can lead to better model performance.

Real-time Data Analysis: Implementing real-time sentiment analysis capabilities could be considered for applications where timely insights are critical.