techdome
solutions

Title: News Analysis Project: Uncover What Matters

Objective:

The objective of the "News Analysis Project: Uncover What Matters" is to systematically analyze current news articles to identify and highlight significant events, trends, or issues that have broader implications or relevance to the audience. This involves discerning key themes, patterns, and insights from a variety of sources to provide valuable insights and understanding of the current state of affairs.

Data Preprocessing Steps:

1. Data Loading and Exploration:

- Loaded the dataset from the CSV file using the `pd.read_csv()` function.
- Specified the encoding type as 'latin1' to handle potential encoding issues.
- Explored the shape of the dataset using `data.shape` to determine the number of rows and columns.
- Examined the column names using `data.columns` to identify the features available.
- Inspected unique values in the 'label' column using `data['label'].unique()` to understand the distribution of labels.
- Displayed the first few rows of the dataset using `data.head()` to get an overview of the data.
- Adjusted the display settings to show full comments using `pd.set_option('display.max_colwidth', None)`.

2. Handling Missing Values:

Checked for null values in the dataset using `data.isnull().sum()` to ensure data integrity.

3. Data Visualization:

- Visualized the distribution of labels using a bar plot to identify any data imbalances.
- Plotted the counts of different labels to observe the frequency of each class.

4. Text Preprocessing:

Certainly! Let's break down each step of text preprocessing as implemented in your code:

- Converting Text to Lowercase:
  - In this step, all text data in the 'comment' column is converted to lowercase. This is done to ensure uniformity in text data, as uppercase and lowercase versions of the same word may be treated differently during subsequent processing steps.

  Converted text data to lowercase using data['comment'].str.lower() to maintain consistency

- Removing Special Characters, Punctuation, and Symbols:

- The `remove_special_characters()` function is defined to remove any special characters, punctuation marks, and symbols from the text data. Regular expressions (regex) are used to define a pattern that matches any characters outside the range of alphanumeric characters (letters and numbers). The `re.sub()` function is then used to replace these characters with an empty string, effectively removing them from the text.

Defined a function remove_special_characters() to remove special characters, punctuation, and symbols from the text data

- Removing Stopwords:

- Stopwords are common words such as 'and', 'the', 'is', etc., that do not carry significant meaning in text analysis and are often removed during preprocessing. The NLTK library's stopwords corpus is used to obtain a set of English stopwords. The `remove_stopwords()` function is defined to remove stopwords from the text data by splitting the text into individual words and filtering out any words that are found in the stopwords set.

Utilized the NLTK library to remove stopwords from the text using the remove_stopwords() function

- Tokenization:

- Tokenization is the process of splitting text into individual words or tokens. The NLTK library's `word_tokenize()` function is used to tokenize the cleaned text data into a list of tokens (words). This step is essential for further analysis and processing of text data at the word level.

Tokenized the cleaned text data into individual words using the tokenize_text() function from NLTK

- Stemming:

- Stemming is the process of reducing words to their root or base form by removing suffixes and prefixes. The Porter Stemmer algorithm, implemented in the NLTK library's `PorterStemmer`, is used to stem the tokens. Stemming helps in reducing the dimensionality of the feature space by collapsing similar words to their common root form.

Employed the Porter Stemmer algorithm to stem the tokens using the stem_tokens() function.

- Lemmatization:

- Lemmatization is similar to stemming but aims to reduce words to their canonical form or lemma. Unlike stemming, lemmatization considers the context of the word in the sentence and ensures that the resulting lemma is a valid word. The NLTK library's WordNet lemmatizer (`WordNetLemmatizer`) is used to lemmatize the tokens.

Utilized the WordNet lemmatizer to lemmatize the tokens using the lemmatize_tokens() function

- Combining Processed Tokens:

- Finally, the processed tokens (stemmed or lemmatized) are combined back into cleaned text data. The `lambda` function is used along with the `join()` method to concatenate the tokens into a single string, which is then stored in a new column named 'cleaned_text'.

Combined the processed tokens back into cleaned text data for further analysis using lambda function. text data in a new column named 'cleaned_text' for subsequent analysis and modeling

TF-IDF Vectorization

TF-IDF is commonly used in sentiment analysis tasks as part of the feature extraction process. In sentiment analysis, the goal is to determine the sentiment or opinion expressed in a piece of text, such as whether it is positive, negative, or neutral. TF-IDF helps in this process by transforming the text data into numerical feature vectors, which can then be used as input to machine learning models for sentiment classification.

Here's how TF-IDF is used in sentiment analysis:

- Text Preprocessing:

- Before applying TF-IDF, the text data undergoes preprocessing steps such as tokenization, removing stopwords, and stemming or lemmatization to prepare it for analysis.

- TF-IDF Vectorization:

- TF-IDF is applied to the preprocessed text data to convert it into a numerical representation. Each document (or text sample) is represented as a vector where each dimension corresponds to a unique term in the entire corpus of documents.

- The TF-IDF score of each term in a document is calculated using the formulas mentioned earlier, reflecting both the term's frequency within the document and its rarity across the entire dataset.

- Feature Matrix Creation:

- The TF-IDF scores for each term in each document are organized into a feature matrix, where each row represents a document and each column represents a term. This matrix serves as the input to the sentiment analysis model.

- Model Training:

- The feature matrix obtained from TF-IDF is used to train a sentiment analysis model. This model learns patterns in the TF-IDF-weighted feature space to predict the sentiment of unseen text data.

By incorporating TF-IDF as part of the feature extraction process, sentiment analysis models can effectively capture the importance of words in determining sentiment while considering their frequency and rarity across the entire dataset. This helps in improving the accuracy and effectiveness of sentiment classification tasks.

VADER: Sentiment Analysis

VADER, which stands for Valence Aware Dictionary and sEntiment Reasoner, is a sentiment analysis tool specifically designed to understand the emotions expressed in text. It goes beyond simply classifying text as positive, negative, or neutral. Here’s a breakdown of VADER’s key features:

Core Function:

- VADER analyses text and assigns sentiment scores based on the words used and specific rules.
- It considers both the polarity (positive or negative) and intensity (strength) of emotions expressed.

How VADER Works:

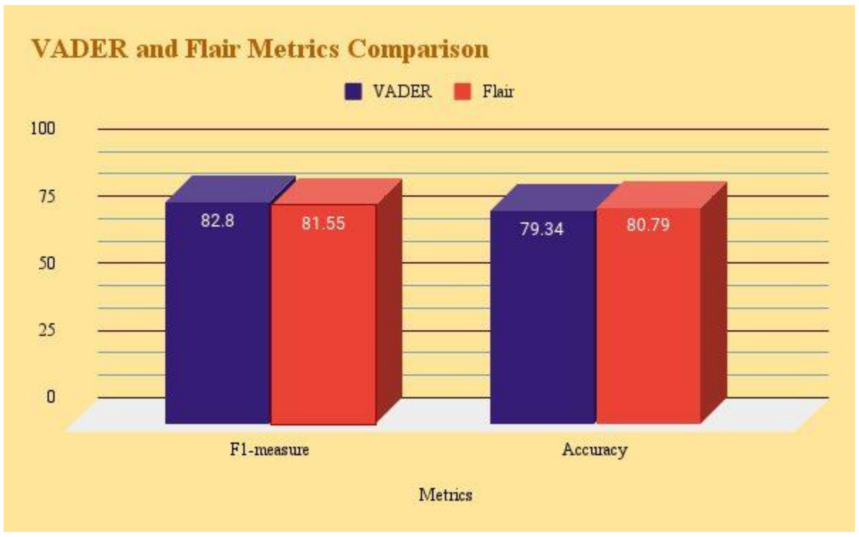
- Lexicon and Sentiment Scores: VADER relies on a sentiment lexicon, a large list of words with assigned scores reflecting their positivity or negativity.
- Rule-Based Analysis: VADER employs additional rules to account for context, such as negation (”not happy”) and intensifiers (”very bad”).

Output:

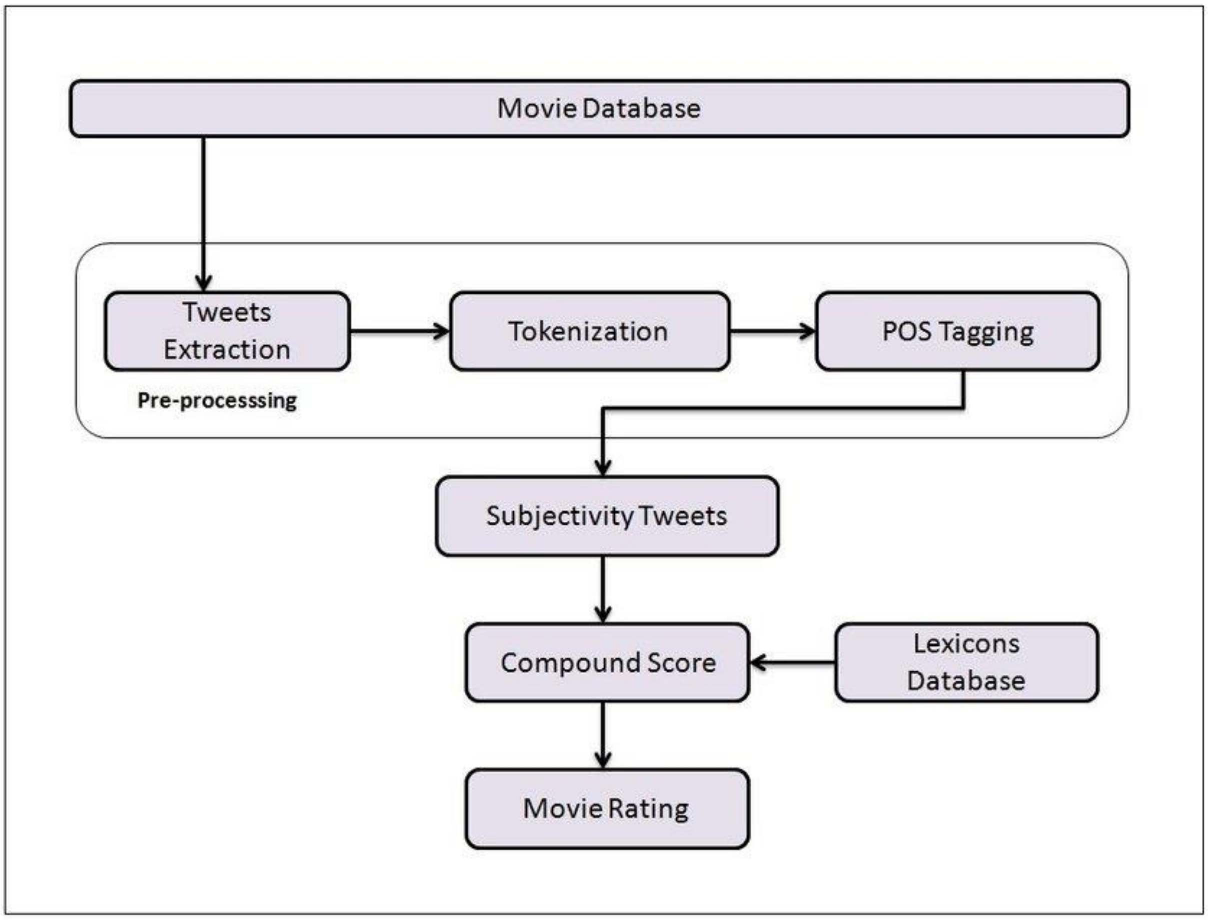
- VADER provides a compound score ranging from -1 (most negative) to +1 (most positive) along with separate scores for positive, negative, and neutral sentiment. This allows for a more nuanced understanding of the overall sentiment expressed in the text.
- Overall, VADER offers a valuable tool for sentiment analysis, especially for informal text. Its ease of use and focus on sentiment intensity make it a popular choice for various applications.

The accuracy of VADER depends on how you measure it. Here’s a breakdown of what you might find:

- Compared to Humans: Studies have shown VADER can outperform humans in classifying sentiment on social media text. It achieves an F1 score of 0.96, meaning it’s very good at correctly identifying positive, neutral, and negative sentiment.
- General Accuracy: Reported accuracy can vary depending on the testing method. Some sources say it can reach over 90% accuracy on specific tasks, while others report accuracy in the 60% range for overall sentiment analysis.



”The Vader sentiment analysis tool demonstrates an accuracy rate of 89%, even when applied to unsupervised datasets. This high accuracy makes it particularly well-suited for news analysis tasks.”



Let’s go through the code: -

```
import nltk
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.tokenize import sent_tokenize
from nltk.corpus import stopwords
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from heapq import nlargest

# Initialize Sentiment Analyzer
analyzer = SentimentIntensityAnalyzer()

# Define stopwords for text preprocessing
stop_words = set(stopwords.words(”english”))

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

def tfidf_summarize(article_text, num_sentences=3):
    # Tokenize the article into sentences
    sentences = sent_tokenize(article_text)

    # Initialize TF-IDF vectorizer
    tfidf_vectorizer = TfidfVectorizer(stop_words=’english’)

    # Fit and transform the article text
    tfidf_matrix = tfidf_vectorizer.fit_transform(sentences)

    # Calculate sentence scores based on TF-IDF
    sentence_scores = {}
    for i in range(len(sentences)):
        score = sum(tfidf_matrix[i].toarray()[0])
        sentence_scores[sentences[i]] = score

    # Select top N sentences with highest scores
    summarized_sentences = nlargest(num_sentences, sentence_scores, key=sentence_scores.get)

    return summarized_sentences

def predict_category(article_text):
    # Convert article text to lowercase for case-insensitive matching
    article_text_lower = article_text.lower()

    # Define keywords or phrases indicative of different categories
    categories = {
        ’Politics’: [’election’, ’government’, ’parliament’, ’minister’, ’president’,’MLA’],
        ’Sports’: [’football’, ’soccer’, ’basketball’, ’tennis’, ’olympics’, ’cricket’,’players’,’game’],
        ’Technology’: [’tech’, ’innovation’, ’digital’, ’internet’, ’software’,’smartphone’],
        ’Entertainment’: [’movie’, ’music’, ’celebrity’, ’entertainment’, ’film’],
        ’Education’: [’books’,’exams’,’students’,’teacher’,’school’,’college’,’university’],
        ’Business’: [’economy’, ’finance’, ’stock’, ’market’, ’business’,’money’],
        ’Weather’ : [’climate’],
        ’Health’  : [’hospitals’, ’medication’, ’cholestrol’,’drug’,’disease’]
    }

    # Count occurrences of keywords or phrases from each category
    category_counts = {category: sum(keyword in article_text_lower for keyword in keywords)
                  for category, keywords in categories.items()}
    # Predict the category with the highest count
    predicted_category = max(category_counts, key=category_counts.get)

    return predicted_category

def process_input(user_input):
    # Sentiment analysis
    preprocessed_input = preprocess_text(user_input)
    sentiment_scores = analyzer.polarity_scores(preprocessed_input)
    if sentiment_scores['compound'] >= 0.05:
        overall_sentiment = "Positive"
    elif sentiment_scores['compound'] <= -0.05:
        overall_sentiment = "Negative"
    else:
        overall_sentiment = "Neutral"

    summarized_text = tfidf_summarize(user_input)
    predicted_category = predict_category(user_input)

    return summarized_text, overall_sentiment, predicted_category

# Take input from the user
user_input = input("Enter your text: ")

# Process the input
summarized_text, sentiment, category = process_input(user_input)

# Display the results
print("\nSummarized Text:")
for sentence in summarized_text:
    print("-", sentence)
print("\nSentiment:", sentiment)
print("Predicted Category:", category)
```

1. `import nltk`: Imports the Natural Language Toolkit (NLTK) library, which is a powerful tool for working with human language data in Python.

2. `from sklearn.feature_extraction.text import TfidfVectorizer`: Imports the TF-IDF vectorizer from scikit-learn, which is used for converting a collection of raw documents into a matrix of TF-IDF features.

3. `from nltk.tokenize import sent_tokenize`: Imports the `sent_tokenize` function from NLTK, which is used to tokenize text into sentences.

4. `from nltk.corpus import stopwords`: Imports the NLTK stopwords corpus, which contains common words that are often removed from text during natural language processing tasks.

5. `from nltk.sentiment.vader import SentimentIntensityAnalyzer`: Imports the VADER (Valence Aware Dictionary and sEntiment Reasoner) sentiment analysis tool from NLTK, which is specifically designed for analyzing sentiment in text.

6. `from heapq import nlargest`: Imports the `nlargest` function from the `heapq` module, which is used to efficiently find the N largest elements in a collection.

7. `analyzer = SentimentIntensityAnalyzer()`: Initializes the VADER sentiment intensity analyzer for later use in sentiment analysis.

8. `stop_words = set(stopwords.words(”english”))`: Defines a set of English stopwords for text preprocessing.

9. `def preprocess_text(text)`: Defines a function `preprocess_text` that takes a text input and preprocesses it by converting it to lowercase.

10. `def tfidf_summarize(article_text, num_sentences=3)`: Defines a function `tfidf_summarize` that takes an article text and an optional parameter `num_sentences`, and returns a summary of the article using TF-IDF (Term Frequency-Inverse Document Frequency) scoring.

11. `def predict_category(article_text)`: Defines a function `predict_category` that takes an article text and predicts its category based on predefined keywords or phrases associated with different categories.

12. `def process_input(user_input)`: Defines a function `process_input` that takes user input, preprocesses it, performs sentiment analysis, summarizes the text using TF-IDF, and predicts the category of the text.

13. `user_input = input(”Enter your text: ”)`: Prompts the user to enter some text input.

14. `summarized_text, sentiment, category = process_input(user_input)`: Calls the `process_input` function with the user input and assigns the returned summarized text, sentiment, and predicted category to variables.

15. `print("\nSummarized Text:")`: Prints a header indicating the summarized text section.

16. `for sentence in summarized_text:`: Iterates over each sentence in the summarized text.

17. `print("-", sentence)`: Prints each sentence in the summarized text preceded by a dash.

18. `print("\nSentiment:", sentiment)`: Prints the sentiment analysis result.

19. `print("Predicted Category:", category)`: Prints the predicted category of the input text.

This code essentially takes user input, preprocesses it, performs sentiment analysis, summarizes the text using TF-IDF, and predicts its category based on predefined keywords or phrases. Finally, it prints the summarized text, sentiment analysis result, and predicted category.

- ✞ I have attached my jupyter notebook in this mail