A Detail Report on Capstone Project:Intensity Analysis using NLP and Python

Step 1: Importing Necessary Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import re
import seaborn as sns
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import warnings
warnings.filterwarnings('ignore')
```

- pandas: For data manipulation and analysis, especially for handling datasets in DataFrame format.
- numpy: For numerical operations and handling arrays.
- matplotlib.pyplot: For plotting graphs and visualizations.
- re: For regular expressions to clean text data.
- seaborn: For enhanced data visualizations based on Matplotlib.
- nltk: The Natural Language Toolkit for processing text data.
- stopwords: To filter out common words that add little meaning (like "and", "the").
- word_tokenize: To split sentences into words.
- warnings: To suppress warnings that may clutter output.

Step 2: Loading the Dataset

Assuming your CSV files are named file1.csv, file2.csv, and file3.csv

```
file1 = pd.read_csv('angriness.csv')
file2 = pd.read_csv('happiness.csv')
file3 = pd.read_csv('sadness.csv')
```

Concatenate the dataframes vertically (row-wise)

intensity_df = pd.concat([file1, file2, file3], ignore_index=True)

Now `combined_df contains data from all three CSV files

print(intensity_df.head())

Printing shape of the dataset

print(intensity_df.shape)

printing columns and rows information

print(intensity_df.info())

- Loading CSV file: Reads three separate CSV files (for different intensity levels) into pandas DataFrames.

Concatenating DataFrames: Combines these DataFrames into a single DataFrame (`intensity_df`).

head(): Displays the first few rows of the combined DataFrame for a quick look.

shape: Prints the number of rows and columns in the DataFrame.

`info(): Provides detailed information about the DataFrame, including data types and null values.

Step 3: Exploratory Data Analysis

Count of unique values in intensity column

intensity_df['intensity'].value_counts()

Count of unique values in content column

intensity_df['content'].value_counts()

- value_counts(): Calculates how many times each unique value appears in the 'intensity' and 'content' columns. Useful for understanding class distribution.

Step 4: Text Preprocessing

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word_tokenize

from nltk.stem import PorterStemmer, WordNetLemmatizer

```
import re
```

Downloading required nltk data

```
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')  # Needed for lemmatization
nltk.download('omw-1.4')  # Additional resources for lemmatization
```

`PorterStemmer` and `WordNetLemmatizer`: For reducing words to their base or root forms.

Downloading NLTK Data: Downloads necessary resources for stopwords, tokenization, and lemmatization.

Cleaning and Tokenizing Content

```
Stop words set
stop_words = set(stopwords.words('english'))
```

Initialize stemmer and lemmatizer

```
ps = PorterStemmer()
lemmatizer = WordNetLemmatizer()
```

Function to clean, tokenize, stem, and lemmatize content

```
def clean_tokenize_stem_lemmatize(content):
```

Removing html brackets and other square brackets from the string using regex

```
    content = re.sub(r'<.*?>', '', content)
```

Removing special characters like @, #, $, etc

```
    content = re.sub(r'[^a-zA-Z0-9\s]', '', content)
```

Removing numbers

```
    content = re.sub(r'\d+', '', content)
```

Converting text to lower case

```
content = content.lower()
```

Tokenization of words

```
tokens = word_tokenize(content)
```

Stop words removal

```
tokens = [word for word in tokens if word not in stop_words]
```

Apply stemming and lemmatization

```
stemmed = [ps.stem(word) for word in tokens]
lemmatized = [lemmatizer.lemmatize(word) for word in stemmed]

return lemmatized  # Returning a list of lemmatized tokens
```

Applying the function on the dataset

```
intensity_df['tokenized_content'] = intensity_df['content'].apply(clean_tokenize_stem_lemmatize)
```

Verify changes (original content and tokenized version)

```
print(intensity_df[['content', 'tokenized_content']].head())
```

Text Cleaning: The `clean_tokenize_stem_lemmatize` function removes HTML tags, special characters, and numbers, converts text to lowercase, tokenizes the text, removes stop words, and applies stemming and lemmatization.

Applying Cleaning: The function is applied to the 'content' column, resulting in a new 'tokenized_content' column.

`print()`: Displays original and cleaned content for verification.

Step 5: Converting Labels (y_train)

```
from sklearn.preprocessing import LabelEncoder
```

Initialize LabelEncoder

label_encoder = LabelEncoder()

Assuming 'intensity' column contains labels like 'happy', 'sad', 'angry'

y = intensity_df['intensity']

Fit and transform labels to numeric form

y_encoded = label_encoder.fit_transform(y)

Verify the encoded labels

print(y_encoded[:10]) # Check first 10 labels

Label Encoding: Converts categorical labels ('intensity') into numeric format for model training using `LabelEncoder`.

`fit_transform()`: Fits the encoder and transforms the labels in one step.

Step 6: Feature Engineering

Textual Features: TF-IDF and N-gram Features

from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer

from scipy.sparse import hstack

Step 1: Convert the tokenized content back to a string for TF-IDF and N-grams

intensity_df['joined_content'] = intensity_df['tokenized_content'].apply(lambda x: ' '.join(x))

Step 2: Initialize TF-IDF Vectorizer

tfidf_vectorizer = TfidfVectorizer(max_features=5000) # Limit to 5000 most common words

tfidf_features = tfidf_vectorizer.fit_transform(intensity_df['joined_content'])

Step 3: Initialize CountVectorizer for n-grams (e.g., unigrams, bigrams, and trigrams)

ngram_vectorizer = CountVectorizer(ngram_range=(1, 3), max_features=5000) # Adjust n-grams as needed

ngram_features = ngram_vectorizer.fit_transform(intensity_df['joined_content'])

Step 4: Combine TF-IDF and N-gram features

combined_features = hstack((tfidf_features, ngram_features))

Optional: Convert to a dense matrix (more memory-intensive)

combined_dense = combined_features.toarray()

Print the shape of the combined features

print("Combined features shape:", combined_dense.shape)

Optional: Check some feature names for verification

feature_names = tfidf_vectorizer.get_feature_names_out()

ngram_names = ngram_vectorizer.get_feature_names_out()

Print some of the feature names from TF-IDF and n-grams

print("Sample TF-IDF feature names:", feature_names[:10]) # Adjust index for more/less

print("Sample n-gram feature names:", ngram_names[:10]) # Adjust index for more/less

- Joining Tokens: Converts tokenized words back into strings for vectorization.
- TF-IDF Vectorization: Initializes a TF-IDF vectorizer to transform text into TF-IDF features.
- N-gram Features: Initializes a CountVectorizer for n-grams (1 to 3 words).
- Combining Features: Combines TF-IDF and n-gram matrices into one sparse matrix.
- Dense Matrix: Converts the combined sparse matrix to a dense matrix if needed.
- Feature Shapes: Prints the shape of the combined feature matrix and some sample feature names.

Part-of-Speech (POS) Tags

Function to get POS tags for a list of tokens

def get_pos_tags(tokens):

```
    return nltk.pos_tag(tokens)  # Returns a list of tuples (word, POS tag)
```

Apply the function to the tokenized content

```
intensity_df['pos_tags'] = intensity_df['tokenized_content'].apply(get_pos_tags)
```

Create a function to count POS tags

```
def pos_tag_counts(pos_tags):
    pos_counts = {}
    for word, pos in pos_tags:
        if pos in pos_counts:
            pos_counts[pos] += 1
        else:
            pos_counts[pos] = 1
    return pos_counts
```

Apply the function to create a new DataFrame for POS tag counts

```
pos_counts_df = intensity_df['pos_tags'].apply(pos_tag_counts).apply(pd.Series).fillna(0)
```

Combine the POS counts with the original DataFrame

```
intensity_df = pd


.concat([intensity_df, pos_counts_df], axis=1)
```

Display the POS counts for verification

```
print(intensity_df[['content', 'pos_tags']].head())
```

POS Tagging: The `get_pos_tags` function tags words with their part of speech (e.g., noun, verb).

Counting POS Tags: The `pos_tag_counts` function counts occurrences of each POS tag.

Creating POS Counts DataFrame: The counts are aggregated into a new DataFrame and concatenated with the original DataFrame.

Final Step: Model Preparation and Validation

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
```

Split the data into training and testing sets

```
X_train, X_test, y_train, y_test = train_test_split(combined_features, y_encoded, test_size=0.2, random_state=42)
```

Initialize the Logistic Regression model

```
model = LogisticRegression(max_iter=1000)  # Increase max_iter for convergence
```

Fit the model on the training data

```
model.fit(X_train, y_train)
```

Predict on the test set

```
y_pred = model.predict(X_test)
```

Evaluate the model's performance

```
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

- Train-Test Split: Splits the feature set and labels into training and testing datasets (80% training, 20% testing).
- Logistic Regression Model: Initializes and fits a logistic regression model on the training data.
- Predictions: Generates predictions on the test set.
- Evaluation Metrics: Evaluates the model's accuracy, generates a classification report, and outputs a confusion matrix for performance analysis.

Overview of `pickle`

`pickle` is a Python module used for serializing and deserializing Python objects. Serialization refers to the process of converting a Python object into a byte stream (saving it), and deserialization is the reverse process (loading it back into memory). This is particularly useful in machine learning for saving trained models and any associated preprocessing steps.

1. Saving Objects

- **Serialization refers to the process of converting a Python object into a byte stream (saving it)**

```
import pickle
```

Save the best model

```
with open('logistic_model.pkl', 'wb') as model_file:
    pickle.dump(best_model, model_file)
```

Save the TF-IDF vectorizer

```
with open('tfidf_vectorizer.pkl', 'wb') as vectorizer_file:
    pickle.dump(tfidf_vectorizer, vectorizer_file)
```

Save the CountVectorizer for n-grams

```
with open('ngram_vectorizer.pkl', 'wb') as ngram_file:
    pickle.dump(ngram_vectorizer, ngram_file)
```

Save the LabelEncoder

```
with open('label_encoder.pkl', 'wb') as le_file:
    pickle.dump(label_encoder, le_file)
```

Save the combined features if needed later

```
with open('combined_features.pkl', 'wb') as features_file:
    pickle.dump(combined_dense, features_file)
```

Explanation:

Opening a File for Writing (`'wb'`): The `open` function is used with the mode `'wb'` (write binary) to create a new file or overwrite an existing file where the object will be saved.

`pickle.dump`: This function takes the object you want to save (like a trained model or a vectorizer) and the file object, and it writes the serialized representation of the object to the file.

Objects Being Saved:

- `best_model`: The trained logistic regression model used for predictions.
- `tfidf_vectorizer`: The TF-IDF vectorizer used for converting text to numerical features.
- `ngram_vectorizer`: The CountVectorizer for n-grams, which captures sequences of words for features.
- `label_encoder`: This encodes the target labels (e.g., intensity labels) into numerical values.
- `combined_dense`: If you used any combined feature set for training, you can save that for later use.

2. Loading Objects

Load the best model

```
with open('logistic_model.pkl', 'rb') as model_file:
    loaded_model = pickle.load(model_file)
```

Load the TF-IDF vectorizer

```
with open('tfidf_vectorizer.pkl', 'rb') as vectorizer_file:
    loaded_tfidf_vectorizer = pickle.load(vectorizer_file)
```

Load the CountVectorizer for n-grams

```
with open('ngram_vectorizer.pkl', 'rb') as ngram_file:
    loaded_ngram_vectorizer = pickle.load(ngram_file)
```

Load the LabelEncoder

```
with open('label_encoder.pkl', 'rb') as le_file:
    loaded_label_encoder = pickle.load(le_file)
```

Load the combined features if needed

```
with open('combined_features.pkl', 'rb') as features_file:
    loaded_combined_features = pickle.load(features_file)
```

Explanation:

- Opening a File for Reading (`'rb'`): Similar to saving, the files are opened with the mode `'rb'` (read binary) for loading the objects.

- `pickle.load`: This function reads the file and reconstructs the original object from the byte stream.

- Loaded Objects: The same objects saved earlier are loaded back into the program for use in predictions.

3. Making Predictions

Once the necessary components are loaded, the code shows how to use them to make predictions on new text input:

```
from scipy.sparse import hstack  # Ensure this import is included
```

Example text input for prediction

```
input_text = ['My phone screen is brighter than my future']
```

Preprocess the input text (vectorization)

```
input_tfidf = loaded_tfidf_vectorizer.transform(input_text)
input_ngram = loaded_ngram_vectorizer.transform(input_text)
```

Combine the TF-IDF and n-gram features

input_combined = hstack((input_tfidf, input_ngram))

Make predictions using the loaded model

predictions = loaded_model.predict(input_combined)

Decode the predictions back to original labels if using LabelEncoder

decoded_predictions = loaded_label_encoder.inverse_transform(predictions)

print(f"Predicted class: {decoded_predictions}")

Output:Predicted class: ['angriness']

Explanation:

- Input Text: The example text is provided as input for prediction. This should be preprocessed in the same way as the training data.

- Vectorization: The text input is transformed using the loaded vectorizers (`loaded_tfidf_vectorizer` and `loaded_ngram_vectorizer`). Each vectorizer converts the text into its respective feature format.

- Combining Features: The `hstack` function from `scipy.sparse` combines the two feature sets (TF-IDF and n-gram) into a single sparse matrix suitable for prediction.

Making Predictions: The loaded model (`loaded_model`) predicts the class labels based on the combined feature set.

Decoding Predictions: The numeric predictions are transformed back into their original labels using the `LabelEncoder`.

Summary

This documentation outlines the entire process of saving and loading machine learning models and their associated preprocessing tools using `pickle`. It ensures that you can recreate the environment needed for making predictions at any later time, thus enabling a smooth workflow for real-time predictions. By using `pickle`, you ensure that all your trained models and preprocessing steps can be reused without the need to retrain or reconfigure them. If you have any further questions or need additional information, feel free to ask!

Conclusion

The code effectively prepares text data for NLP classification tasks, conducts thorough preprocessing, and evaluates model performance using logistic regression. This pipeline should enable to predict emotional intensity in text reviews with decent accuracy.