Report on Text Preprocessing Code

Introduction

This report details the steps taken to preprocess a text dataset for Natural Language Processing (NLP) analysis on therapy sessions, using a custom-built code to clean, tokenize, remove stop words, and stem the text. The preprocessing code, implemented in Python, is crucial for simplifying and standardizing the text data while preserving semantic meaning, preparing it for further analytical tasks such as sentiment analysis, text classification, or clustering.

1. Loading and Initial Cleaning

- **Data Loading:** We begin by reading a CSV file containing text data into a pandas DataFrame. The specific column of interest, post, contains text data that we aim to analyze.
- **Handling Missing Values:** Any NaN values in the post column are replaced with an empty string to ensure that missing data does not interfere with further processing.
- **Text Lowercasing:** A new column, lowercase, is created by converting all text to lowercase. This step standardizes the text, removing case sensitivity that could create inconsistency in subsequent analysis.
- **Removing Special Characters:** In the no_special_chars column, all non-alphabet characters (like punctuation, numbers, etc.) are removed. This helps simplify the text for further processing, ensuring that only relevant words remain.

2. Stop Words Removal

- **Setup of Stop Words:** Using the NLTK library, we define a set of English stop words—common words like "and," "the," and "is," which usually add little meaning to text analysis.
- **Stop Words Removal Function:** We create a function, remove_stopwords, that tokenizes each text entry and removes any stop words. This function is applied to no_special_chars, resulting in a new column, no_stopwords, with these common words filtered out.

3. Tokenization

- **Tokenizing the Cleaned Text:** We further process no_stopwords by tokenizing the text (splitting it into individual words). This creates a tokens column where each row contains a list of words, providing a structured format for word-level analysis.

4. Stemming Using Custom Porter Stemmer

- **Custom Porter Stemmer Implementation:** We implement a custom version of the Porter Stemmer algorithm within a porter class. Stemming reduces words to their root form, which is helpful in text analysis by reducing different forms of a word to a common base.
- **Class Methods and Stemming Rules:**
    - **Counting Consonant-Vowel Pairs (**m_count**)** to determine word structure.
    - **Checking for Vowels and Consonants** to apply specific stemming rules based on word patterns.

- **Applying Porter Stemming Rules:** The porter class defines various steps (e.g., step_1a, step_1b, etc.), each implementing specific transformations on suffixes like -sses, -ies, -ed, and -ing to reduce words to their base forms. This is essential to reduce noise in the data by standardizing variations of a word.

- **Applying the Stemming Process:** A function, stem_words, applies each step of the Porter Stemmer to every token. This function is then applied to the tokens column, creating a stemmed column that contains lists of stemmed words.

5. Observing Results

- **Displaying Results:** After processing, we print the first few rows of the key columns (lowercase, no_special_chars, no_stopwords, tokens, and stemmed) to verify the effectiveness of each preprocessing step.

Conclusion

This preprocessing pipeline/code performs essential text transformations that prepare the data for further analysis. By standardizing case, removing irrelevant characters, filtering out stop words, and stemming each word, this code helps reduce data complexity while preserving the core meaning of each text entry.