Name : Yash Sarang	Class/Roll No. : D16AD / 47	Grade :

Title of Experiment :
Comparative Analysis of Porter's Stemmer Algorithm and NLTK Stemming Library for Text Stemming

Problem Statement :
The experiment aims to compare the performance and effectiveness of Porter's Stemmer algorithm and NLTK stemming library in stemming text data. Specifically, we intend to analyze the differences in stemming output and computational efficiency between these two approaches.

Description / Theory :
Stemming is a fundamental preprocessing step in natural language processing (NLP) and information retrieval. It involves reducing words to their base or root forms (stems), which helps in simplifying the analysis of textual data. Porter's Stemmer is one of the most widely used stemming algorithms, designed to produce effective stems for English words. On the other hand, NLTK (Natural Language Toolkit) provides a robust library for stemming in various languages, including English.

Flowchart :
1. Input text data
2. Preprocess text (tokenization, lowercasing, etc.)
3. Apply Porter's Stemmer algorithm to obtain stems
4. Apply NLTK stemming library to obtain stems
5. Compare the stemming outputs from Porter's Stemmer and NLTK
6. Analyze the computational efficiency of both approaches
7. Present the results and discussions

Program:

```
import re
import nltk
from nltk.stem import PorterStemmer

nltk.download('punkt')

porter = PorterStemmer()
[redacted]
def porter_stemmer(word):
    def apply_rule(word, suffix, replacement, condition):
        if condition(word):
            stem = re.sub(suffix + "$", replacement, word)
            return stem
        return word

    def step1a(word):
        if word.endswith("sses"):
            return word[:-2]
        elif word.endswith("ies"):
            return word[:-2]
        elif word.endswith("ss"):
            return word
        elif word.endswith("s"):
            return word[:-1]
        return word

    def step1b(word):
        if word.endswith("eed"):
            if re.search(r"eed$", word) and len(word) > 4:
                return word[:-1]
        elif word.endswith("ed"):
            if re.search(r"ed$", word):
                                return apply_rule(word[:-2], "ed", "", lambda w:
re.search(r"[aeiouy]", w))
        elif word.endswith("ing"):
            if re.search(r"ing$", word):
```

```
                return apply_rule(word[:-3], "ing", "", lambda w: re.search(r"[aeiouy]", w))
        return word

    def step1c(word):
        if word.endswith("y"):
            if re.search(r"y$", word):
                return apply_rule(word[:-1], "y", "i", lambda w: re.search(r"[aeiouy]", w))
        return word

    def step2(word):
        if re.search(r"(ational|tional|enci|anci|izer|bli|alli|entli|eli|ousli|ization|ation|ator|alism|iveness|fulness|ousness|aliti|iviti|biliti)$", word):
            return apply_rule(word, r"(ational|tional|enci|anci|izer|bli|alli|entli|eli|ousli|ization|ation|ator|alism|iveness|fulness|ousness|aliti|iviti|biliti)$", "", lambda w: re.search(r"[aeiouy]", w))
        return word

    def step3(word):
        if re.search(r"(icate|ative|alize|iciti|ical|ful|ness)$", word):
            return apply_rule(word, r"(icate|ative|alize|iciti|ical|ful|ness)$", "", lambda w: re.search(r"[aeiouy]", w))
        return word

    def step4(word):
        if re.search(r"(al|ance|ence|er|ic|able|ible|ant|ement|ment|ent|ism|ate|iti|ous|ive|ize)$", word):
            return apply_rule(word, r"(al|ance|ence|er|ic|able|ible|ant|ement|ment|ent|ism|ate|iti|ous|ive|ize)$", "", lambda w: len(w) > 1)
        elif word.endswith("sion") or word.endswith("tion"):
            return apply_rule(word, r"(sion|tion)$", "", lambda w: re.search(r"[aeiouy]", w))
```

```
        return word

    def step5a(word):
        if re.search(r"e$", word):
            return apply_rule(word[:-1], "e", "", lambda w: m(w) > 1 or (m(w) == 1 and not cvc(w[:-1])))
        return word

    def step5b(word):
        if m(word) > 1 and re.search(r"ll$", word):
            return word[:-1]
        return word

    def m(word):
        return len(re.findall(r"[aeiouy]+", word))

    def cvc(word):
        return re.search(r"[aeiouy][^aeiouy][aeiouy]$", word) and not re.search(r"[wxy]$", word)

    word = word.lower()
    word = step1a(word)
    word = step1b(word)
    word = step1c(word)
    word = step2(word)
    word = step3(word)
    word = step4(word)
    word = step5a(word)
    word = step5b(word)

    return word
# Example usage
words = ["ruined", "filler", "studies", "happier", "agreement", "saying", "strange"]
for word in words:
    print(f"{word}: {porter_stemmer(word)}")
# Apply Porter stemmer
for word in words:
```

```
    stemmed_word = porter.stem(word)
    print(f"{word}: {stemmed_word}")
```

Output:

Our_Stemmer

```
ruined: ruin
filler: fill
studies: studi
happier: happi
agreement: agr
saying: sa
strange: strang
```

Porter_Stemmer

```
ruined: ruin
filler: filler
studies: studi
happier: happier
agreement: agreement
saying: say
strange: strang
```

Results and Discussions :

The experiment results will include a comparison of stems generated by Porter's Stemmer and NLTK stemming library for a given set of input texts. Additionally, computational efficiency, such as processing time and memory usage, will be compared for both methods. The discussion will focus on the differences observed in stemming outputs and the implications for downstream NLP tasks.

Conclusion:

The experiment concludes by summarizing the comparison between Porter's Stemmer algorithm and NLTK stemming library in terms of their performance, stemming accuracy, and computational efficiency. The findings will help in understanding which method is more suitable for specific NLP applications and data types.

******