Department of Computer Science and Engineering (Data Science)

NAME: Tanisha Chavan SAPID: 60009220086 RollNo: D113

Experiment No 3

Aim: Implement stemming using Porter Stemmer Algorithm

Theory:

Stemming is the process of producing morphological variants of a root/base word. Stemming programs are commonly referred to as stemming algorithms or stemmers. A stemming algorithm reduces the words “chocolates”, “chocolatey”, “choco” to the root word, “chocolate” and “retrieval”, “retrieved”, “retrieves” reduce to the stem “retrieve”. **Stemming** is an important part of the pipelining process in Natural language processing. The input to the stemmer is tokenized words. How do we get these tokenized words? Well, tokenization involves breaking down the document into different words.
Stemming is a natural language processing technique used to reduce words to their base form, also known as the root form. The process of stemming is used to normalize text and make it easier to process. It is an important step in text pre-processing, and it is commonly used in information retrieval and text mining applications.

There are several different algorithms for stemming, including the Porter stemmer, Snowball stemmer, and the Lancaster stemmer. The Porter stemmer is the most widely used algorithm, and it is based on a set of heuristics that are used to remove common suffixes from words. The Snowball stemmer is a more advanced algorithm that is based on the Porter stemmer, but it also supports several other languages in addition to English. The Lancaster stemmer is a more aggressive stemmer, and it is less accurate than the Porter stemmer and Snowball stemmer.

Stemming can be useful for several natural language processing tasks such as text classification, information retrieval, and text summarization. However, stemming can also have negative effects such as reducing the text's readability, and it may not always produce the correct root form of a word.

It is important to note that stemming is different from Lemmatization. Lemmatization is the process of reducing a word to its base form, but unlike stemming, it considers the context of the word and produces a valid word, unlike stemming which may produce a non-word as the root form.

Some more example of stemming for root word "like" include:

->"likes"

->"liked"

->"likely"

->"liking"

Errors in Stemming:
There are mainly two errors in stemming –

- over-stemming
- under-stemming

Over-stemming occurs when two words stem from the same root of different stems. Over-stemming can also be regarded as false positives. Over-stemming is a problem that can occur when using stemming algorithms in natural language processing. It refers to the situation where a stemmer produces a root form that is not a valid word or is not the correct root form of a word. This can happen when the stemmer is too aggressive in removing suffixes or when it does not consider the context of the word.
Over-stemming can lead to a loss of meaning and make the text less readable. For example, the word "arguing" may be stemmed to "argu," which is not a valid word and does not convey the same meaning as the original word. Similarly, the word "running" may be stemmed to "run," which is the base form of the word, but it does not convey the meaning of the original word.

To avoid over-stemming, it is important to use a stemmer that is appropriate for the task and language. It is also important to test the stemmer on a sample of text to ensure that it is producing valid root forms. In some cases, using a lemmatize instead of a stemmer may be a better solution as it considers the context of the word, making it less prone to errors.

Another approach to this problem is to use techniques like semantic role labeling, sentiment analysis, context-based information, etc. that help to understand the context of the text and make the stemming process more precise.

Under-stemming occurs when two words stem from the same root not of different stems. Under-stemming can be interpreted as false negatives. Under-stemming is a problem that can occur when using stemming algorithms in natural language processing. It refers to the situation where a stemmer does not produce the correct root form of a word or does not reduce a word to its base form. This can happen when the stemmer is not aggressive enough in removing suffixes or when it is not designed for the specific task or language.
Under-stemming can lead to a loss of information and make it more difficult to analyze text. For example, the words "arguing" and "argument" may be stemmed to "argu," which does not convey the meaning of the original words. Similarly, the words "running" and "runner" may be stemmed to "run," which is the base form of the word, but it does not convey the meaning of the original words.

To avoid under-stemming, it is important to use a stemmer that is appropriate for the task and language. It is also important to test the stemmer on a sample of text to ensure that it is producing the correct root forms. In some cases, using a lemmatizer instead of a stemmer may be a better solution as it takes into account the context of the word, making it less prone to errors.

Another approach to this problem is to use techniques like semantic role labeling, sentiment analysis, context-based information, etc. that help to understand the context of the text and make the stemming process more precise.

Applications of stemming:

1. Stemming is used in information retrieval systems like search engines.
2. It is used to determine domain vocabularies in domain analysis.
3. To display search results by indexing while documents are evolving into numbers and to map documents to common subjects by stemming.
4. Sentiment Analysis, which examines reviews and comments made by different users about anything, is frequently used for product analysis, such as for online retail stores. Before it is interpreted, stemming is accepted in the form of the text-preparation mean.
5. A method of group analysis used on textual materials is called document clustering (also known as text clustering). Important uses of it include subject extraction, automatic document structuring, and quick information retrieval.

Fun Fact: Google search adopted a word stemming in 2003. Previously a search for "fish" would not have returned "fishing" or "fishes".

Some Stemming algorithms are:

Porter's Stemmer algorithm

It is one of the most popular stemming methods proposed in 1980. It is based on the idea that the suffixes in the English language are made up of a combination of smaller and simpler suffixes. This stemmer is known for its speed and simplicity. The main applications of Porter Stemmer include data mining and Information retrieval. However, its applications are only limited to English words. Also, the group of stems is mapped on to the same stem and the output stem is not necessarily a meaningful word.
The algorithms are lengthy in nature and are known to be the oldest stemmer.
Example: EED -> EE means "if the word has at least one vowel and consonant plus EED ending, change the ending to EE" as 'agreed' becomes 'agree'.

Advantage: It produces the best output as compared to other stemmers and it has a lower error rate.
Limitation: Morphological variants produced are not always real words.

Steps of Porter Stemmer Algorithm:

Algorithm

To present the suffix stripping algorithm in its entirety we will need a few definitions.

A consonant in a word is a letter other than A, E, I, O or U, and other than Y preceded by a consonant. (The fact that the term **consonant** is defined to some extent in terms of itself does not

make it ambiguous.) So in TOY the consonants are T and Y, and in SYZYGY they are S, Z and G. If a letter is not a consonant it is a vowel.
A consonant will be denoted by c, a vowel by v. A list ccc... of length greater than 0 will be denoted by C, and a list vvv... of length greater than 0 will be denoted by V. Any word, or part of a word, therefore, has one of the four forms:

- CVCV ... C
- CVCV ... V
- VCVC ... C
- VCVC ... V

These may all be represented by the single form

[C]VCVC ... [V]

where the square brackets denote arbitrary presence of their contents. Using (VC^m) to denote VC repeated m times, this may again be written as

[C]($ VC^m $)[V]

m will be called the measure of any word or word part when represented in this form. The case m = 0 covers the null word. Here are some examples:

- m=0 TR, EE, TREE, Y, BY.
- m=1 TROUBLE, OATS, TREES, IVY.
- m=2 TROUBLES, PRIVATE, OATEN, ORRERY.

The rules for removing a suffix will be given in the form

(condition) S1 -> S2

This means that if a word ends with the suffix S1, and the stem before S1 satisfies the given condition, S1 is replaced by S2. The condition is usually given in terms of m, e.g.

(m > 1) EMENT ->

Here S1 is 'EMENT' and S2 is null. This would map REPLACEMENT to REPLAC, since REPLAC is a word part for which m = 2.

The 'condition' part may also contain the following:

- *S - the stem ends with S (and similarly for the other letters).
- *v* - the stem contains a vowel.
- m=2 TROUBLES, PRIVATE, OATEN, ORRERY.
- *d - the stem ends with a double consonant (e.g. -TT, -SS).
- *o - the stem ends cvc, where the second c is not W, X or Y (e.g. -WIL, -HOP).

And the condition part may also contain expressions with and, or and not, so that:

(m>1 and (*S or *T)) : tests for a stem with m>1 ending in S or T, while
(*d and not (*L or *S or *Z)) : tests for a stem ending witha double consonant other than L, S or Z. Elaborate conditions like this are required only rarely.
In a set of rules written beneath each other, only one is obeyed, and this will be the one with the longest matching S1 for the given word. For example, with

- SSES -> SS
- IES -> I
- SS -> SS
- S ->

(here the conditions are all null) CARESSES maps to CARESS since SSES is the longest match for S1. Equally CARESS maps to CARESS (S1=SS) and CARES to CARE (S1=S).
In the rules below, examples of their application, successful or otherwise, are given on the right in lower case. The algorithm now follows:

Step 1a :

- SSES -> SS (Example : caresses -> caress)
- IES -> I (Example : ponies -> poni ; ties -> ti)
- SS -> SS (Example : caress -> caress)
- S -> (Example : cats -> cat)

Step 1b :

- (m>0) EED -> EE (Example : feed -> feed ; agreed -> agree)
- (*v*) ED -> (Example : plastered -> plaster ; bled -> bled)

- (*v*) ING -> (Example : motoring -> motor ; sing -> sing)
- S -> (Example : cats -> cat)

If the second or third of the rules in Step 1b is successful, the following is done:

- AT -> ATE (Example : conflat(ed) -> conflate)
- BL -> BLE (Example : troubl(ed) -> trouble)
- IZ -> IZE (Example : siz(ed) -> size)
- S -> (Example : cats -> cat)
- (*d and not (*L or *S or *Z)) -> single letter (Example : hopp(ing) -> hop ; tann(ed) -> tan ; fall(ing) -> fall ; hiss(ing) -> hiss ; fizz(ed) -> fizz)
- (m=1 and *o) -> E (Example : fail(ing) -> fail ; fil(ing) -> file)

The rule to map to a single letter causes the removal of one of the double letter pair. The -E is put back on -AT, -BL and -IZ, so that the suffixes -ATE, -BLE and -IZE can be recognised later. This E may be removed in step 4.

Step 1c :
(*v*) Y -> I (Example : happy -> happi ; sky -> sky)
Step 1 deals with plurals and past participles. The subsequent steps are much more straightforward.
Step 2 :

- (m>0) ATIONAL -> ATE (Example : relational -> relate
- (m>0) TIONAL -> TION (Example : conditional -> condition ; rational -> rational)
- (m>0) ENCI -> ENCE (Example : valenci -> valence)
- (m>0) ANCI -> ANCE (Example : hesitanci -> hesitance)
- (m>0) IZER -> IZE (Example : digitizer -> digitize)
- (m>0) ABLI -> ABLE (Example : conformabli -> conformable)
- (m>0) ALLI -> AL (Example : radicalli -> radical)
- (m>0) ENTLI -> ENT (differentli -> different)
- (m>0) ELI -> E (vileli -> vile)

- (m>0) OUSLI -> OUS (analogousli -> analogous)
- (m>0) IZATION -> IZE (vietnamization -> vietnamize)
- (m>0) ATION -> ATE (predication -> predicate)
- (m>0) ATOR -> ATE (operator -> operate)
- (m>0) ALISM -> AL (feudalism -> feudal)
- (m>0) IVENESS -> IVE (decisiveness -> decisive)
- (m>0) FULNESS -> FUL (hopefulness -> hopeful)
- (m>0) OUSNESS -> OUS (callousness -> callous)
- (m>0) ALITI -> AL (formaliti -> formal)
- (m>0) IVITI -> IVE (sensitiviti -> sensitive)
- (m>0) BILITI -> BLE (sensibiliti -> sensible)

The test for the string S1 can be made fast by doing a program switch on the penultimate letter of the word being tested. This gives a fairly even breakdown of the possible values of the string S1. It will be seen in fact that the S1-strings in step 2 are presented here in the alphabetical order of their penultimate letter. Similar techniques may be applied in the other steps.

Step 3 :

- (m>0) ICATE -> IC (Example : triplicate -> triplic)
- (m>0) ATIVE -> (Example : formative -> form)
- (m>0) ALIZE -> AL (Example : formalize -> formal)
- (m>0) ICITI -> IC (Example : electriciti -> electric))
- (m>0) ICAL -> IC (Example : electrical -> electric)
- (m>0) FUL -> (Example : hopeful -> hope)
- (m>0) NESS -> (Example : goodness -> good)

Step 4 :

- (m>1) AL -> (Example : revival -> reviv)
- (m>1) ANCE -> (Example : allowance -> allow)

- (m>1) ENCE -> (Example : inference -> infer)
- (m>1) ER -> (Example : airliner -> airlin)
- (m>1) IC -> (Example : gyroscopic -> gyroscop)
- (m>1) ABLE -> (Example : adjustable -> adjust)
- (m>1) IBLE -> (Example : defensible -> defens)
- (m>1) ANT -> (Example : irritant -> irrit)
- (m>1) EMENT -> (Example : replacement -> replac)
- (m>1) MENT -> (Example : adjustment -> adjust)
- (m>1) ENT -> (Example : dependent -> depend)
- (m>1 and (*S or *T)) ION -> (Example : adoption -> adopt)
- (m>1) OU -> (Example : homologou -> homolog)
- (m>1) ISM -> (Example : communism -> commun)
- (m>1) ATE -> (Example : activate -> activ)
- (m>1) ITI -> (Example : angulariti -> angular)
- (m>1) OUS -> (Example : homologous -> homolog)
- (m>1) IVE -> (Example : effective -> effect)
- (m>1) IZE -> (Example : bowdlerize -> bowdler)

The suffixes are now removed. All that remains is a little tidying up.

Step 5a :

- (m>1) E -> (Example : probate -> probat ; rate -> rate)
- (m=1 and not *o) E -> (Example : cease -> ceas)

Step 5b
(m > 1 and *d and *L) -> single letter
(Example : controll -> control ; roll -> roll)

Lab assignment to be performed in this session:

Department of Computer Science and Engineering (Data Science)

1. **Implement stemming using Porter Stemmer algorithm in Python.**
2. **Implement stemming using library functions in NLTK for stemming.**

TANISHA CHAVAN- 60009220086

STEMMMING USING PORTER STEMMER ALGORITHM

```
import re

vowels = "aeiou"
double_consonants = ["bb", "dd", "ff", "gg", "mm", "nn", "pp", "rr", "tt"]
li_ending = "cdeghkmnrt"
```

```
def is_consonant(word, i):
    if word[i] in vowels:
        return False
    if word[i] == "y":
        return i == 0 or not is_consonant(word, i - 1)
    return True

def measure(word):
    form = "".join(["C" if is_consonant(word, i) else "V" for i in range(len
    (word))])
    return form.count("VC")
```

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

def step1b(word):
    if word.endswith("eed"):
        if measure(word[:-1]) > 0:
            return word[:-1]
        else:
            return word

    if word.endswith("ed") and re.search("[aeiou]", word[:-2]):
        return word[:-2]

    if word.endswith("ing") and re.search("[aeiou]", word[:-3]):
        return word[:-3]

    return word
```

```
def stem(word):
    word = word.lower()
    word = step1a(word)
    word = step1b(word)
    return word
```

```
words = ["running", "flies", "caresses", "happiness"]
stems = [stem(w) for w in words]
print(stems)
```

```
['runn', 'fli', 'caress', 'happiness']
```

STEMMING USING LIB FUNCTIONS IN NLTK

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
```

```
stemmer = PorterStemmer()
words = ["running", "flies", "caresses", "happiness"]
stems = [stemmer.stem(word) for word in words]
print(stems)

```

```
['run', 'fli', 'caress', 'happi']
```

CONCLUSION:

The experiment successfully demonstrated two distinct methods for implementing the Porter Stemmer algorithm. The manual approach provided insight into the underlying logic and structure of the algorithm, while the NLTK implementation offered a streamlined, ready-to-use solution that efficiently processes text data. Both methods effectively reduced words to their root forms, emphasizing the importance of stemming in natural language processing tasks.