List of Experiments (NLP)

1. Word Analysis

2. Word Generation

3. Morphology

4. N-Grams

5. N-Grams Smoothing

LAB PROG1 :

AIM: simple Python program that performs basic word analysis in NLP

THEORY:

This program uses the word_tokenize function from the nltk.tokenize module to split the text into words. It then uses the nltk.FreqDist class to perform frequency analysis on the tokens and finds the 10 most frequent words. The frequency of each word is then printed.

EXP1:WORD ANALYSIS

Word analysis in NLP (Natural Language Processing) refers to the process of studying individual words within a larger text corpus in order to understand their meaning, context, and relationships with other words. This can include tasks such as word tokenization, stemming and lemmatization, part-of-speech tagging, and Named Entity Recognition (NER).

Word analysis is a crucial component of NLP and is often used as a preprocessing step before more advanced NLP techniques, such as sentiment analysis, text classification, and machine translation. The goal of word analysis is to extract meaningful information from text and to represent that information in a way that can be processed by computational algorithms.

Word analysis in NLP involves several tasks to extract meaningful information from text and represent it in a computationally manageable format. Here's a simple example that demonstrates some common word analysis techniques:

Suppose we have the following sentence: "The quick brown fox jumps over the lazy dog."

1. Tokenization: This involves breaking down the sentence into individual words, or tokens. In this example, the sentence would be tokenized into the following list of words: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog."]
2. Stemming and Lemmatization: These are techniques that reduce words to their root form, so that words with the same meaning are represented in the same way. For example, the word "jumps" could be stemmed to "jump" and lemmatized to "jump."

3. Part-of-Speech (POS) Tagging: This involves marking each word in a sentence with its corresponding part of speech, such as noun, verb, adjective, etc. For example, the word "jumps" in this sentence would be tagged as a verb.
4. Named Entity Recognition (NER): This involves identifying and categorizing named entities, such as people, organizations, and locations, within a sentence. For example, the word "dog" in this sentence could be tagged as a named entity of type "animal."

These word analysis techniques allow us to extract useful information from text, such as the meaning of words, relationships between words, and context in which words are used. This information can then be used in various NLP applications, such as text classification, sentiment analysis, and machine translation.

PROGRAM:

```
import nltk

from nltk.tokenize import word_tokenize

nltk.download('punkt')

  # Sample text

text = "Natural language processing (NLP) is the ability of a computer
program to understand human language as it is spoken."

# Tokenize the text into words

tokens = word_tokenize(text)

print(tokens)

# Perform frequency analysis on the tokens

word_freq = nltk.FreqDist(tokens)

# Print the 10 most frequent words

print("The 10 most frequent words are:")

for word, freq in word_freq.most_common(10):

    print(f"{word}: {freq}")
```

RESULTS:

```
['Natural', 'language', 'processing',
'(', 'NLP', ')', 'is', 'the', 'ability',
'of', 'a', 'computer', 'program', 'to',
'understand', 'human', 'language', 'as',
'it', 'is', 'spoken', '.']

The 10 most frequent words are:

language: 2

is: 2

Natural: 1
```

```
processing: 1
(: 1
NLP: 1
): 1
the: 1
ability: 1
of: 1
```

LAB PROG2 :

AIM:

simple Python program for word generation in NLP

THEORY:

EXP2:WORD GENERATION

Word generation in NLP (Natural Language Processing) refers to the task of generating new words or sentences based on a given text corpus or set of inputs. This can be accomplished through various methods, such as statistical language modeling, neural network-based language generation, and rule-based text generation.

In statistical language modeling, a model is trained on a large corpus of text and then used to generate new words or sentences by predicting the likelihood of certain sequences of words.

In neural network-based language generation, deep learning models, such as Recurrent Neural Networks (RNNs) or Transformer networks, are trained on a large corpus of text to generate new words or sentences based on patterns and relationships learned from the training data.

In rule-based text generation, a set of rules or templates is used to generate new words or sentences based on specific patterns or relationships between words.

Word generation is a valuable tool in NLP for tasks such as data augmentation, text summarization, and language translation, among others. However, it can also be used to generate misleading or fake text, so it's important to be aware of its potential limitations and ethical considerations.

Word generation in NLP refers to the task of generating new words or sentences based on a given text corpus or set of inputs. Here's a simple example of word generation using a neural network-based language model:

Suppose we have a corpus of text that includes the sentence "The quick brown fox jumps over the lazy dog." A neural network-based language model can be trained on this text to predict the next word in a sentence, given a set of previous words as input.

For example, if we input the sequence "The quick brown fox" into the model, it may generate the next word "jumps." We can then feed that output back into the model to generate the next word, and so on. The final output could be a sentence like: "The quick brown fox jumps over the green fence."

In this example, the neural network-based language model has learned the relationships between words and the patterns of language used in the training corpus, and has generated a new sentence based on that information.

Word generation can be a useful tool in NLP for tasks such as text summarization, data augmentation, and language translation. However, it's important to note that the quality of the generated words or sentences can vary depending on the quality of the training data and the methods used for word generation.

PROGRAM:

```
# Importing the necessary libraries

import nltk

import random

from nltk.corpus import words

nltk.download('words')

 # Generating a random word from the corpus

random_word = random.choice(words.words())

# Printing the generated random word

print("The randomly generated word is:", random_word)
```

RESULTS:

```
The randomly generated word is: meritoriousness
```

LAB PROG3 :

AIM:

simple Python program for Morphology in NLP

THEORY:

Morphology is the study of word structure and formation, including inflection and derivation. Here is a simple Python program that performs basic morphological operations, such as stemming and lemmatization, using the Natural Language Toolkit (NLTK) library in this program, we first tokenize the input text into words, and then perform stemming and lemmatization on each word. The PorterStemmer and WordNetLemmatizer classes from the NLTK library are used for this purpose. The resulting stemmed and lemmatized words are then returned and printed.

EXP3:MORPHOLOGY

Morphology in NLP (Natural Language Processing) is the study of the internal structure of words and how they can be modified to create new words. It deals with inflection, derivation, and compounding.

For example, the word "unhappy" is formed by adding the prefix "un-" to the base word "happy." The prefix "un-" changes the meaning of the word to the opposite, making "unhappy" mean "not happy." This process is an example of morphology in NLP, where the structure of words is analyzed to understand how they are formed and how their meanings are affected by various modifications.

PROGRAM:

```
import nltk

from nltk.stem import PorterStemmer

from nltk.tokenize import word_tokenize

from nltk.stem import WordNetLemmatizer

nltk.download('punkt')

nltk.download('wordnet')

nltk.download('omw-1.4')

def perform_stemming(text):

    stemmer = PorterStemmer()

    stemmed_words = []

    words = word_tokenize(text)

    for word in words:

        stemmed_words.append(stemmer.stem(word))

    return stemmed_words

def perform_lemmatization(text):

    lemmatizer = WordNetLemmatizer()

    lemmatized_words = []

    words = word_tokenize(text)

    for word in words:

        lemmatized_words.append(lemmatizer.lemmatize(word))
```

```python
    return lemmatized_words

text = "This is an example sentence showing off the stemming and lemmatization in NLP"

stemmed_words = perform_stemming(text)

print("Stemmed words:", stemmed_words)

lemmatized_words = perform_lemmatization(text)

print("Lemmatized words:", lemmatized_words)
```

RESULTS:

```
Stemmed words: ['thi', 'is', 'an', 'exampl', 'sentenc', 'show', 'off', 'the', 'stem', 'and', 'lemmat', 'in', 'nlp']

Lemmatized words: ['This', 'is', 'an', 'example', 'sentence', 'showing', 'off', 'the', 'stemming', 'and', 'lemmatization', 'in', 'NLP']
```

LAB PROG4 :

AIM:

a simple Python program that generates N-grams in NLP

THEORY:

EXP4: N-Grams

N-grams in NLP (Natural Language Processing) refer to a contiguous sequence of N items from a given text, where N can be any positive integer. The items can be words, letters, or other units, depending on the context.

For example, let's say we have the sentence: "The quick brown fox jumps over the lazy dog."

- A 1-gram (or unigram) would be a sequence of one item: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
- A 2-gram (or bigram) would be a sequence of two items: ["The quick", "quick brown", "brown fox", "fox jumps", "jumps over", "over the", "the lazy", "lazy dog"]
- A 3-gram (or trigram) would be a sequence of three items: ["The quick brown", "quick brown fox", "brown fox jumps", "fox jumps over", "jumps over the", "over the lazy", "the lazy dog"]

N-grams are often used in NLP for tasks such as language modeling, text classification, and machine translation, as they can provide insights into the structure and meaning of a text.

PROGRAM:

```
import nltk

from nltk.util import ngrams

nltk.download('punkt')

  # Sample text

text = "Natural language processing (NLP) is the ability of a computer
program to understand human language as it is spoken."

# Tokenize the text into words

tokens = nltk.word_tokenize(text)

# Generate N-grams from the tokens

N = 3

ngrams_list = list(ngrams(tokens, N))

# Print the N-grams

print("The {}-grams are:".format(N))

for ngram in ngrams_list:

    print(ngram)
```

RESULTS:

```
The 3-grams are:

('Natural', 'language', 'processing')

('language', 'processing', '(')

('processing', '(', 'NLP')

('(', 'NLP', ')')

('NLP', ')', 'is')

(')', 'is', 'the')

('is', 'the', 'ability')

('the', 'ability', 'of')

('ability', 'of', 'a')

('of', 'a', 'computer')

('a', 'computer', 'program')

('computer', 'program', 'to')

('program', 'to', 'understand')

('to', 'understand', 'human')

('understand', 'human', 'language')
```

```
('human', 'language', 'as')

('language', 'as', 'it')

('as', 'it', 'is')

('it', 'is', 'spoken')

('is', 'spoken', '.')
```

LAB PROG5 :

AIM:

a simple Python program that implements N-gram smoothing in NLP

THEORY:

This program uses the word_tokenize function from the nltk module to split the text into words. It then generates bigrams (pairs of words) from the tokens using the ngrams function. A Kneser-Ney Interpolated (KN-Interpolated) language model is trained on the bigrams using the KneserNeyInterpolated class from the nltk.lm module. The model is then used to evaluate the probabilities of the bigrams in the text.

Kneser-Ney smoothing is a type of N-gram smoothing that adjusts the probabilities of N-grams based on the frequency of lower-order N-grams in the training data. This helps to avoid the issue of zero probability N-grams that can arise with simple maximum likelihood estimation.

EXP5: N-Grams Smoothing

N-gram smoothing is a technique used in NLP (Natural Language Processing) to estimate the probability of an N-gram that has not been seen in the training data. This technique helps to solve the problem of zero probability, which occurs when an N-gram that appears in the test data has not been seen in the training data.

There are several methods of N-gram smoothing, but one of the most commonly used methods is called add-k smoothing. In add-k smoothing, a small value k is added to the count of each N-gram, before computing the probabilities. This helps to avoid zero probabilities and ensures that the probabilities of all N-grams sum up to 1.

For example, let's say we have a corpus of text containing the following bigrams: "the quick", "quick brown", "brown fox", and "fox jumps". We want to estimate the probability of the bigram "jumps over", which is not present in the training data.

Using add-k smoothing with k = 1, we can compute the probabilities as follows:

- P("jumps over") = (0 + 1) / (4 + 4) = 1/8

Here, we added 1 to the count of each bigram to avoid zero probabilities, and divided the result by the total count of all bigrams plus 4 (which is the number of possible bigrams that can be formed from the given vocabulary).

N-gram smoothing is a useful technique in NLP for improving the accuracy of language models and other NLP applications.

PROGRAM:

```
import nltk

from nltk.util import ngrams

from nltk.lm import KneserNeyInterpolated

nltk.download('punkt')

# Sample text

text = "Natural language processing (NLP) is the ability of a computer program to understand human language as it is spoken."

# Tokenize the text into words

tokens = nltk.word_tokenize(text)

# Generate bigrams from the tokens

bigrams = ngrams(tokens, 2)

# Train a Kneser-Ney Interpolated (KN-Interpolated) language model on the bigrams

model = KneserNeyInterpolated(2)

model.fit(bigrams,bigrams)

# Evaluate the model's probabilities of bigrams

print("Probabilities of bigrams:")

for bigram in bigrams:

    prob = model.score(bigram)

    print("{}: {:.3f}".format(bigram, prob))
```

RESULTS:

Probabilities of bigrams: