गृह मंत्रालय
MINISTRY OF
HOME AFFAIRS

राष्ट्रीय न्यायालयिक विज्ञान विश्वविद्यालय
(राष्ट्रीय महत्त्व का संस्थान, गृह मंत्रालय, भारत सरकार)
National Forensic Sciences University
(An Institution of National Importance under Ministry of Home Affairs, Government of India)

NATURAL LANGUAGE PROCESSING

Lab Manual- v1

Dr Ranjit Kolkar
ranjit.kolkar@gmail.com

Contents

Lab 1: Analysis of Tokenization and N-grams in Natural Language Processing

Input: A sample document containing text.

Output:

1. Total number of tokens in the document.
2. Number of unique tokens in the document.
3. Token frequency distribution.
4. N-gram frequency distribution (bi-grams by default).

Analysis: Tokenization: Tokenization is the process of breaking down a text into smaller units, typically words or punctuation marks. In this analysis, the document is tokenized using the word_tokenize function from the NLTK library. The total number of tokens and the number of unique tokens are calculated to understand the complexity and diversity of the text.

N-grams: N-grams are contiguous sequences of n items from a given text. In this analysis, bi-grams are generated from the tokenized text using the ngrams function from the NLTK library. The frequency distribution of bi-grams is calculated to identify common sequences of two words in the text.

Impact: Tokenization and N-grams are essential techniques in natural language processing (NLP) for various tasks such as text analysis, information retrieval, and machine translation. By tokenizing text, we can analyze the structure of the text and extract meaningful information. N-grams help in capturing context and understanding relationships between words in a text.

Advantages:

1. Tokenization helps in breaking down text into meaningful units for analysis.
2. N-grams capture contextual information and can be used in language modeling and predictive text applications.
3. Token frequency distribution provides insights into the most common words in the text.
4. N-gram frequency distribution helps in identifying common phrases or sequences of words in the text.

Disadvantages:

1. Tokenization may not always accurately capture the semantics of the text, especially in languages with complex grammar.
2. N-grams can become computationally expensive for large values of n, leading to increased processing time and memory usage.
3. N-grams may produce sparse representations of text, especially for higher values of n, which can impact the performance of some NLP tasks.

Tokenization and N-grams are valuable techniques in NLP for analyzing and processing text data, providing insights into the structure and content of the text

Code

```
import nltk
nltk.download('punkt')
from nltk.tokenize import word_tokenize
from nltk.util import ngrams
from nltk.probability import FreqDist

# Sample document
document = "Natural language processing (NLP) is a subfield of artificial intelligence (AI) that focuses on the interaction between computers and humans using natural language."

# Tokenization
tokens = word_tokenize(document.lower())

# Total number of tokens
total_tokens = len(tokens)

# Number of unique tokens
unique_tokens = set(tokens)
num_unique_tokens = len(unique_tokens)

# Token frequency distribution
token_freq = FreqDist(tokens)

# Generate N-grams
n = 2  # Change to desired n-gram size
n_grams = list(ngrams(tokens, n))

# N-gram frequency distribution
n_gram_freq = FreqDist(n_grams)

# Report
```

```
print("Total number of tokens:", total_tokens)
print("Number of unique tokens:", num_unique_tokens)
print("\nToken frequency distribution:")
for token, freq in token_freq.items():
    print(f"{token}: {freq}")
print("\nN-gram frequency distribution:")
for n_gram, freq in n_gram_freq.items():
    print(f"{' '.join(n_gram)}: {freq}")
```

Output:

Total number of tokens: 28
Number of unique tokens: 24

Token frequency distribution:
natural: 2
language: 2
processing: 1
(: 2
nlp: 1
): 2
is: 1
a: 1
subfield: 1
of: 1
artificial: 1
intelligence: 1
ai: 1
that: 1
focuses: 1
on: 1
the: 1
interaction: 1
between: 1
computers: 1
and: 1
humans: 1
using: 1
.: 1

N-gram frequency distribution:
natural language: 2
language processing: 1
processing (: 1
(nlp: 1
nlp): 1
) is: 1
is a: 1

a subfield: 1
subfield of: 1
of artificial: 1
artificial intelligence: 1
intelligence (: 1
(ai: 1
ai): 1
) that: 1
that focuses: 1
focuses on: 1
on the: 1
the interaction: 1
interaction between: 1
between computers: 1
computers and: 1
and humans: 1
humans using: 1
using natural: 1
language .: 1

Lab 2: Analysis of Stop Words, Stemming, and Lemmatization in Natural Language Processing

Input: A sample document containing text.

Output:

1. Document after stop words removal.
2. Document after stemming.
3. Document after lemmatization.

Analysis: Stop Words Removal: Stop words are common words (e.g., "the", "is", "and") that are often filtered out from text data as they do not contribute significantly to the meaning of the text. In this analysis, stop words are removed from the document using a predefined list of stop words.

Stemming: Stemming is the process of reducing words to their base or root form. It helps in normalizing words and reducing them to their common base form. In this analysis, stemming is applied to the document using the Porter stemmer algorithm from the NLTK library.

Lemmatization: Lemmatization is similar to stemming but takes into account the context of the word to determine its lemma (base or dictionary form). Lemmatization provides more accurate results compared to stemming but can be computationally expensive. In this analysis, lemmatization is applied to the document using the WordNetLemmatizer from the NLTK library.

Impact: Stop Words Removal: Removing stop words can help in reducing the noise in text data and improving the performance of text analysis tasks such as sentiment analysis and text classification.

Stemming: Stemming can help in reducing words to their base form, which can improve the efficiency of text retrieval and search engines by grouping together words with the same root.

Lemmatization: Lemmatization provides a more accurate representation of words compared to stemming, as it considers the context of the word. It can improve the performance of NLP tasks that require understanding the meaning of words, such as machine translation and information extraction.

Advantages:

1. Stop words removal helps in reducing noise and improving the efficiency of text analysis.
2. Stemming and lemmatization help in normalizing words and reducing them to their base form, improving text processing tasks.
3. Lemmatization provides a more accurate representation of words compared to stemming.

Disadvantages:

1. Stop words removal may remove important context from the text, leading to potential loss of information.
2. Stemming may produce incorrect or non-words as it only focuses on removing prefixes or suffixes to reduce words to their base form.
3. Lemmatization can be computationally expensive compared to stemming, especially for large text datasets.

Stop words removal, stemming, and lemmatization are important preprocessing steps in NLP that help in improving the quality of text data and enhancing the performance of NLP tasks.

```
import nltk

nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer, WordNetLemmatizer

# Sample document
document = "The quick brown fox jumps over the lazy dog. This is a sample
sentence for testing the NLTK preprocessing techniques like stop words
removal, stemming, and lemmatization. These techniques help in cleaning
and normalizing text data for natural language processing tasks."
# Tokenization
tokens = word_tokenize(document.lower())

# Stop words removal
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word not in stop_words]

# Stemming
stemmer = PorterStemmer()
stemmed_tokens = [stemmer.stem(word) for word in filtered_tokens]

# Lemmatization
lemmatizer = WordNetLemmatizer()
lemmatized_tokens = [lemmatizer.lemmatize(word) for word in
filtered_tokens]

# Report
print("Document after stop words removal:")
print(' '.join(filtered_tokens))
print("\nDocument after stemming:")
print(' '.join(stemmed_tokens))
print("\nDocument after lemmatization:")
print(' '.join(lemmatized_tokens))

# Token count and count of removed tokens
print("\nToken count before preprocessing:", len(tokens))
print("Token count after preprocessing:", len(filtered_tokens))
print("Count of removed tokens:", len(tokens) - len(filtered_tokens))
```

```
Document after stop words removal:
quick brown fox jumps lazy dog . sample sentence testing nltk preprocessing
techniques like stop words removal , stemming , lemmatization . techniques
help cleaning normalizing text data natural language processing tasks .

Document after stemming:
quick brown fox jump lazi dog . sampl sentenc test nltk preprocess techniqu
like stop word remov , stem , lemmat . techniqu help clean normal text data
natur languag process task .

Document after lemmatization:
quick brown fox jump lazy dog . sample sentence testing nltk preprocessing
technique like stop word removal , stemming , lemmatization . technique help
cleaning normalizing text data natural language processing task .
Token count before preprocessing: 46
Token count after preprocessing: 33
Count of removed tokens: 13
```

Lab 3: Part-of-Speech (POS) Tagging in Natural Language Processing

Input: A sample document containing text.

Output:

1. POS tags for each word in the document.
2. Distribution of POS tags in the document.

Analysis: Part-of-Speech (POS) Tagging: POS tagging is the process of assigning a grammatical category (such as noun, verb, adjective, etc.) to each word in a text. It helps in understanding the syntactic structure of the text and is used in various NLP tasks such as text analysis, information extraction, and machine translation.

In this analysis, the NLTK library is used to perform POS tagging on the input document. The `pos_tag` function from the NLTK library is used to assign POS tags to each word in the document.

Impact: POS tagging provides valuable information about the syntactic structure of a text, which can be used in various NLP tasks. It helps in understanding the relationships between words in a sentence and can improve the accuracy of text analysis tasks.

Advantages:

1. POS tagging helps in understanding the grammatical structure of a text.
2. It provides useful information for tasks such as information extraction, text summarization, and machine translation.
3. POS tags can be used as features in machine learning models for NLP tasks.

Disadvantages:

1. POS tagging accuracy can vary depending on the language and the complexity of the text.
2. It can be computationally expensive for large texts or texts with complex grammatical structures.
3. POS tagging may not always accurately capture the nuances of natural language, especially in ambiguous cases.

POS tagging is a valuable tool in NLP for understanding the grammatical structure of text and can be used in a wide range of applications to improve the analysis of natural language text.

```
import nltk

from nltk.tokenize import word_tokenize
from collections import Counter

nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

# Sample document
document = """
Natural language processing (NLP) is a subfield of linguistics, computer
science, information engineering, and artificial intelligence concerned
with the interactions between computers and human (natural) languages, in
particular how to program computers to process and analyze large amounts
of natural language data.
"""

# Tokenize the document
tokens = word_tokenize(document)

# Perform POS tagging
pos_tags = nltk.pos_tag(tokens)

# POS tags for each word
print("POS tags for each word in the document:")
for word, pos_tag in pos_tags:
    print(f"{word}: {pos_tag}")

# Distribution of POS tags
pos_tag_distribution = Counter(tag for word, tag in pos_tags)
print("\nDistribution of POS tags in the document:")
for tag, count in pos_tag_distribution.items():
    print(f"{tag}: {count}")
```

Output

```
POS tags for each word in the document:
Natural: JJ
language: NN
processing: NN
(: (
NLP: NNP
): )
is: VBZ
a: DT
subfield: NN
```

```
of: IN
linguistics: NNS
,: ,
computer: NN
science: NN
,: ,
information: NN
engineering: NN
,: ,
and: CC
artificial: JJ
intelligence: NN
concerned: VBN
with: IN
the: DT
interactions: NNS
between: IN
computers: NNS
and: CC
human: JJ
(: (
natural: JJ
): )
languages: NNS
,: ,
in: IN
particular: JJ
how: WRB
to: TO
program: NN
computers: NNS
to: TO
process: VB
and: CC
analyze: VB
large: JJ
amounts: NNS
of: IN
natural: JJ
language: NN
data: NNS
.: .

Distribution of POS tags in the document:
JJ: 7
NN: 10
(: 2
NNP: 1
): 2
VBZ: 1
DT: 2
IN: 5
NNS: 7
,: 4
```

```
CC: 3
VBN: 1
WRB: 1
TO: 2
VB: 2
.: 1
```

Lab 4: Analysis of Minimum Distance Between Two Words in a Text

Input: A sample text containing words.

Output: Minimum distance between two specified words in the text.

Analysis: The task involves finding the minimum distance between two specified words in a given text. In this analysis, we implement a Python function to calculate this distance. The function takes three inputs: the text, and two target words for which the minimum distance needs to be calculated.

The function first splits the input text into words and then iterates through the words to find the indices of the two target words. It keeps track of the minimum distance found so far and updates it whenever a new minimum distance is calculated. Finally, it returns the minimum distance between the two target words.

Impact: Calculating the minimum distance between two words can be useful in various text analysis tasks. For example, in information retrieval systems, it can help in identifying the proximity of keywords in a document. In natural language processing tasks, it can assist in analyzing the syntactic or semantic relationships between words.

Advantages:

1. Provides a simple and efficient way to calculate the minimum distance between two words in a text.
2. Can be easily integrated into text processing pipelines for various NLP tasks.
3. Helps in extracting useful information about the relationship between words in a text.

Disadvantages:

1. The function assumes that the input text is preprocessed and tokenized into words. It may not work correctly with unstructured text.
2. It calculates the minimum distance based on word indices, which may not always reflect the actual semantic distance between words in the text.

Overall, the analysis of the minimum distance between two words in a text provides valuable insights into the spatial relationships between words and can be a useful tool in various text analysis tasks.

```
def min_distance_between_words(text, word1, word2):
    words = text.split()
    indices_word1 = [i for i, word in enumerate(words) if word.lower() ==
word1.lower()]
```

```
    indices_word2 = [i for i, word in enumerate(words) if word.lower() ==
word2.lower()]

    if not indices_word1 or not indices_word2:
        return "One or both words not found in the text."

    min_distance = float('inf')
    for i in indices_word1:
        for j in indices_word2:
            distance = abs(i - j)
            if distance < min_distance:
                min_distance = distance

    return min_distance

# Sample text and words
sample_text = "This is a sample text containing sample words."
word1 = "sample"
word2 = "words"

# Calculate and print the minimum distance
min_dist = min_distance_between_words(sample_text, word1, word2)
print(f"Minimum distance between '{word1}' and '{word2}': {min_dist}")
```

Output:

```
Minimum distance between 'sample' and 'words': One or both words not found in
the text.
```

Lab 5: Implementing the Viterbi Algorithm for Hidden Markov Models in Python

Input:

- A sequence of observations (obs) representing the observed data.
- A set of hidden states (states) representing the possible states of the system.
- Initial probabilities of each state (start_prob).
- Transition probabilities between states (trans_prob).
- Emission probabilities of each state emitting each observation (emit_prob).

Output:

- The most likely sequence of hidden states corresponding to the observed data.

Analysis: The Viterbi algorithm is a dynamic programming algorithm used to find the most likely sequence of hidden states in a hidden Markov model (HMM) given a sequence of observations. It is widely used in various fields such as speech recognition, bioinformatics, and natural language processing.

The algorithm involves three main steps: initialization, recursion, and backtracking.

- Initialization: Initialize the delta and phi matrices to store the probabilities and backpointers for each state at each time step.
- Recursion: Iterate through the observations and update the delta and phi matrices based on the probabilities of transitioning between states and emitting observations.
- Backtracking: Once all observations are processed, backtrack through the phi matrix to find the most likely sequence of hidden states.

Impact:

1. Speech Recognition: In speech recognition systems, the Viterbi algorithm is used to find the most likely sequence of phonemes based on the observed speech signal.
2. Part-of-Speech Tagging: In natural language processing, the algorithm is used to assign the most likely part-of-speech tags to words in a sentence.
3. Bioinformatics: In bioinformatics, the algorithm is used to predict the most likely sequence of hidden states (e.g., gene structures) based on observed data (e.g., DNA sequences).
4. Error Correction: The algorithm can also be used in error correction systems to find the most likely correct sequence of data given an observed sequence with errors.

```python
import numpy as np

# Define the states (HOT, COLD)
states = ['HOT', 'COLD']

# Define the observations (1, 2, 3)
observations = [1, 2, 3]

# Define the initial probabilities
start_prob = {'HOT': 0.8, 'COLD': 0.2}

# Define the transition probabilities
trans_prob = {
    'HOT': {'HOT': 0.6, 'COLD': 0.4},
    'COLD': {'HOT': 0.3, 'COLD': 0.7}
}

# Define the emission probabilities
emit_prob = {
    'HOT': {1: 0.4, 2: 0.4, 3: 0.2},
    'COLD': {1: 0.3, 2: 0.2, 3: 0.5}
}

def viterbi(obs, states, start_prob, trans_prob, emit_prob):
    T = len(obs)
    N = len(states)
    path = np.zeros(T)
    delta = np.zeros((T, N))
    phi = np.zeros((T, N))

    # Initialization
    for s in range(N):
        delta[0][s] = start_prob[states[s]] * emit_prob[states[s]][obs[0]]
        phi[0][s] = 0

    # Recursion
    for t in range(1, T):
        for s in range(N):
```

```
            delta[t][s] = np.max(delta[t - 1] *
np.array([trans_prob[states[k]][states[s]] for k in range(N)])) *
emit_prob[states[s]][obs[t]]
            phi[t][s] = np.argmax(delta[t - 1] *
np.array([trans_prob[states[k]][states[s]] for k in range(N)]))

    # Backtrack
    path[T - 1] = np.argmax(delta[T - 1])
    for t in range(T - 2, -1, -1):
        path[t] = phi[t + 1][int(path[t + 1])]

    return [states[int(state)] for state in path]

# Example usage
observed_sequence = [3, 1, 3]
hidden_states_sequence = viterbi(observed_sequence, states, start_prob,
trans_prob, emit_prob)
print("Most likely sequence of weather states given the observed
sequence:", hidden_states_sequence)
```

Mini Project

1. Select the Mini project from the NLP.

2. The project should be complete and deployable. Select the project that is applicable or useful to the college/student/faculty/admin

3. Evaluation Criteria

    1. Presentation/ Demo
    2. Implementation/ Design / Algorithm
    3. Application, or deployable
    4. Report