

ST.ANNE' S COLLEGE OF ENGINEERING AND TECHNOLOGY

Approved by AICTE, New Delhi. Affiliated to Anna University, Chennai Accredited by NAAC,

ANGUCHETTPALAYAM, PANRUTI – 607 106.



LAB MANUAL

CD3281 NATURAL LANGUAGE PROCESSING LABORATORY

REGULATION 2021

YEAR SEMESTER: II V

PREPARED BY

Ms.R.MANICKAVASAGAN,AP/AIML

1 .WORD ANALYSIS

AIM:

To write a python program for word analysis.

ALGORITHM:

1. Input the text as a string.
2. Remove punctuation by checking each character and keeping only non-punctuation characters.
3. Convert the text to lowercase for uniformity.
4. Split the cleaned text into a list of words (tokens) using spaces.
5. Count total words using len() on the token list.
6. Count unique words using len(set(tokens)).
7. Count frequency of each word using a dictionary or collections.Counter.
8. Sort and get the most common words by frequency.
9. Display the report with total, unique, and top frequent words.
10. End the program.

PROGRAM:

```
import string
from collections import Counter
text = "The quick brown fox jumps over the lazy dog. The dog barks."
s = text.lower()
s = "".join(ch for ch in s if ch not in string.punctuation)
tokens = s.split()
```

```
total_words = len(tokens)
unique_words = len(set(tokens))
word_freq = Counter(tokens)
print("— Word Analysis Report (Pure Python) —")
print("Total words: {}".format(total_words))
print("Unique words: {}".format(unique_words))
print()
print("Most common words:")
for word, freq in word_freq.most_common(5):
    print(" {}: {}".format(word, freq))
```

OUTPUT:

```
— Word Analysis Report (Pure Python) —
Total words: 12
Unique words: 9
Most common words:
the: 3
dog: 2
quick: 1
brown: 1
fox: 1
```

RESULT :

Thus the above word analysis python program was executed successfully.

2.WORD GENERATION

AIM:

To write a python program for word analysis.

ALGORITHM:

1. Create three lists: one each for prefixes, roots, and suffixes.
2. Define a function to generate one word at a time.
3. Randomly choose one prefix from the prefix list.
4. Randomly choose one root from the root list.
5. Randomly choose one suffix from the suffix list.
6. Concatenate prefix, root, and suffix to form a new word.
7. Return the generated word from the function.
8. Repeat the process multiple times (e.g., 5 times) using a loop.
9. Print each generated word with its count label
10. End.

PROGRAM:

```
import random

prefixes = ["un", "re", "pre", "mis"]
roots = ["act", "read", "view", "port"]
suffixes = ["ing", "ed", "able", "ful"]

def generate_word():
    prefix = random.choice(prefixes)
    root = random.choice(roots)
    suffix = random.choice(suffixes)
    return prefix + root + suffix
```

```
print("— Word Generation (Pure Python) —")  
for i in range(5):  
    print(f"Generated word {i+1}: {generate_word()}")
```

OUTPUT:

```
— Word Generation (Pure Python) —  
Generated word 1: misviewed  
Generated word 2: unactable  
Generated word 3: misreaded  
Generated word 4: previewed  
Generated word 5: misviewed
```

RESULT:

Thus the above word generation python program was implemented and executed successfully.

3.MORPHOLOGY

AIM:

To write a python program for morphology

ALGORITHM:

1. Input a word to be stemmed.
2. Convert the word to lowercase to standardize processing.
3. Check if the word ends with "ing" and has more than 3 characters, then remove "ing".
4. If the word ends with "ed", remove the "ed" suffix.
5. If the word ends with "s", remove the "s" to handle simple plurals.
6. If none of the conditions match, return the word as-is.
7. Repeat for all words in the list using a loop.
8. Print both original and stemmed words side by side.
- 9.End.

PROGRAM:

```
def simple_stemmer(word):  
    word = word.lower()  
    if word.endswith('ing'):  
        # Check for a possible 'e' before removing 'ing'  
        if word.endswith('ing') and len(word) > 3:  
            return word[:-3]  
    else:  
        return word
```

```
if word.endswith('ed'):
    return word[:-2]
if word.endswith('s'):
    # Handle simple plural `s`
    return word[:-1]
return word
words_to_analyze = ["running", "ran", "runs", "better", "geese", "corpora"]
print("— Morphology (Pure Python Simple Stemmer) —")
for word in words_to_analyze:
    stem = simple_stemmer(word)
    print(f"Original: {word.ljust(10)}, Stem: {stem}")
```

OUTPUT:

— Morphology (Pure Python Simple Stemmer) —

Original: running , Stem: runn

Original: ran , Stem: ran

Original: runs , Stem: run

Original: better , Stem: better

Original: geese , Stem: geese

Original: corpora , Stem: corpora

RESULT:

Thus the above morphology python program was implemented and executed successfully.

4.N GRAMS

AIM:

To write a python program for N GRAMS.

ALGORITHM:

1. Input a text string that you want to analyze.
2. Convert the text to lowercase for consistency.
3. Split the text into tokens (words) using `.split()`.
4. Define a function that takes the list of tokens and a number `n`.
5. Loop from index 0 to `len(tokens) - n + 1` to cover all possible N-grams.
6. Extract N tokens at each step using slicing `tokens[i:i+n]`.
7. Convert the slice to a tuple to represent one N-gram.
8. Append each N-gram to the list of N-grams.
9. Return the final list of N-gram tuples.
10. Call the function with `n=2` and `n=3` to get bigrams and trigrams.
11. Print the generated N-grams clearly.
12. End.

PROGRAM:

```
def generate_ngrams(tokens, n):  
    """Generates N-grams from a list of tokens."""  
    ngrams_list = []  
    for i in range(len(tokens) - n + 1):  
        ngram = tuple(tokens[i:i+n])  
        ngrams_list.append(ngram)  
    return ngrams_list
```

```
text = "N-grams are contiguous sequences of N items."  
tokens = text.lower().split()
```

```
print("— N-Grams (Pure Python) —")  
bigrams = generate_ngrams(tokens, 2)  
print("Bigrams:", bigrams)
```

```
trigrams = generate_ngrams(tokens, 3)  
print("\nTrigrams:", trigrams)
```

OUTPUT:

— N-Grams (Pure Python) —

Bigrams: [('n-grams', 'are'), ('are', 'contiguous'), ('contiguous', 'sequences'), ('sequences', 'of'), ('of', 'n'), ('n', 'items.')]]

Trigrams: [('n-grams', 'are', 'contiguous'), ('are', 'contiguous', 'sequences'), ('contiguous', 'sequences', 'of'), ('sequences', 'of', 'n'), ('of', 'n', 'items.')]]

RESULT:

Thus the above n grams python program was implemented and executed successfully.

5.N GRAMS SMOOTHING

AIM:

To write a python program for N GRAMS SMOOTHING.

ALGORITHM:

1. Input a text corpus and the value of n (bigram = 2, trigram = 3, etc.).
2. Convert the text to lowercase and split it into tokens (words).
3. Add <START> and <END> tokens to represent the beginning and end of sentences.
4. Create a vocabulary set from the list of tokens and compute its size.
5. Initialize count dictionaries for N-Grams and (N-1)-Gram contexts.
6. Slide through the token list and extract each N-Gram and its context.
7. Count how many times each context and each next word occurs.
8. Initialize a dictionary for smoothed probabilities.
9. For each context, calculate the smoothed probability of each possible next word using:
10. Add probabilities for unseen words in each context to make the model complete.
11. Return the full probability model as a nested dictionary.
12. For selected context-word pairs, lookup and print their smoothed probabilities.
13. Repeat the same process for trigram model (n=3) and test other context predictions.
14. End.

PROGRAM:

```
import collections  
  
def laplace_smoothing_alternative(text, n=2):
```

```

words = text.lower().split()
words = ['<START>'] * (n - 1) + words + ['<END>'] * (n - 1)
vocabulary = set(words)
vocab_size = len(vocabulary)
ngram_counts = collections.defaultdict(lambda: collections.defaultdict(int))
context_counts = collections.defaultdict(int)
for i in range(len(words) - n + 1):
    ngram = tuple(words[i:i+n])
    context = ngram[:-1]
    next_word = ngram[-1]
    ngram_counts[context][next_word] += 1
    context_counts[context] += 1
smoothed_probabilities = collections.defaultdict(dict)
for context, next_word_counts in ngram_counts.items():
    denominator = context_counts[context] + vocab_size
    for next_word in vocabulary:
        numerator = next_word_counts.get(next_word, 0) + 1
        prob = numerator / denominator
        smoothed_probabilities[context][next_word] = prob
for context in context_counts.keys():
    denominator = context_counts[context] + vocab_size
    for word in vocabulary:
        if word not in smoothed_probabilities[context]:
            numerator = 1 # The count is 0, so add one
            prob = numerator / denominator
            smoothed_probabilities[context][word] = pro

```

```
    return smoothed_probabilities

text_corpus = """
the cat sat on the mat.
the dog chased the cat.
a cat is an animal.
"""

smoothed_model = laplace_smoothing_alternative(text_corpus, n=2)
print("Smoothed Probabilities:")
context = ('the',)
next_word = 'cat'
if context in smoothed_model and next_word in smoothed_model[context]:
    prob = smoothed_model[context][next_word]
    print(f"P('{next_word}' | '{context[0]}') = {prob:.4f}")
else:
    print(f"P('{next_word}' | '{context[0]}') not found.")
next_word = 'dog'
if context in smoothed_model and next_word in smoothed_model[context]:
    prob = smoothed_model[context][next_word]
    print(f"P('{next_word}' | '{context[0]}') = {prob:.4f}")
else:
    print(f"P('{next_word}' | '{context[0]}') not found.")
context = ('is',)
next_word = 'cat'
if context in smoothed_model and next_word in smoothed_model[context]:
    prob = smoothed_model[context][next_word]
    print(f"P('{next_word}' | '{context[0]}') = {prob:.4f}")
```

else:

```
print(f"P('{next_word}' | '{context[0]}') not found.")
```

```
smoothed_trigram_model = laplace_smoothing_alternative(text_corpus, n=3)
```

```
context = ('the', 'cat')
```

```
next_word = 'mat'
```

```
if context in smoothed_trigram_model and next_word in  
smoothed_trigram_model[context]:
```

```
    prob = smoothed_trigram_model[context][next_word]
```

```
    print(f"\nP('{next_word}' | '{context[0]} {context[1]}') = {prob:.4f}")
```

else:

```
    print(f"\nP('{next_word}' | '{context[0]} {context[1]}') not found.")
```

OUTPUT:

Smoothed Probabilities:

$P(\text{'cat'} \mid \text{'the'}) = 0.1111$

$P(\text{'dog'} \mid \text{'the'}) = 0.1111$

$P(\text{'cat'} \mid \text{'is'}) = 0.0667$

$P(\text{'mat'} \mid \text{'the cat'})$ not found.

RESULT:

Thus the above n grams smoothing python program was implemented and executed successfully.

6.POS TAGGING: Hidden Markov model

AIM:

To write a python program for POS TAGGING: Hidden Markov model.

ALGORITHM:

1. Input training sentences with (word, tag) pairs.
2. Count transitions between tags: $P(\text{tag}_i | \text{tag}_{i-1})$.
3. Count emissions: $P(\text{word} | \text{tag})$.
4. Count total occurrences for each tag.
5. For a new sentence, use the Viterbi method to find the tag sequence with highest probability.
6. At each step, multiply previous max path with transition and emission probability.
7. Track back best paths using dynamic programming.
8. Return the most probable tag sequence.
9. End.

PROGRAM:

```
states = ['NOUN', 'VERB']
observations = ['dog', 'barks']
start_probability = {'NOUN': 0.6, 'VERB': 0.4}
transition_probability = {'NOUN': {'NOUN': 0.1, 'VERB': 0.9},
                          'VERB': {'NOUN': 0.8, 'VERB': 0.2}}
emission_probability = {'NOUN': {'dog': 0.9, 'barks': 0.1},
                        'VERB': {'dog': 0.1, 'barks': 0.9}}
def viterbi(obs, states, start_p, trans_p, emit_p):
    V = [{}]
```

```

path = {}
for state in states:
    V[0][state] = start_p[state] * emit_p[state].get(obs[0], 0)
    path[state] = [state]
for t in range(1, len(obs)):
    V.append({})
    new_path = {}
    for curr_state in states:
        (prob, prev_state) = max(
            (V[t - 1][prev_state] * trans_p[prev_state][curr_state] *
             emit_p[curr_state].get(obs[t], 0),
             prev_state)
            for prev_state in states
        )
        V[t][curr_state] = prob
        new_path[curr_state] = path[prev_state] + [curr_state]
    path = new_path
    (prob, state) = max((V[-1][state], state) for state in states)
    return path[state]

sentence = ['dog', 'barks']

tags = viterbi(sentence, states, start_probability, transition_probability,
               emission_probability)

print("Sentence:", sentence)
print("Predicted POS tags:", tags)

```

OUTPUT:

Sentence: ['dog', 'barks']

Predicted POS tags: ['NOUN', 'VERB']

RESULT:

Thus the above POS tagging python program was implemented and executed successfully.

7. POS Tagging: Viterbi Decoding

AIM:

To write a python program for POS TAGGING Viterbi Decoding

ALGORITHM:

1. Define the states (POS tags) and observations (words).
2. Initialize the probabilities: start, transition, and emission.
3. Create a Viterbi table and a backpointer table.
4. For the first word, compute initial probabilities.
5. For each word ($t > 0$), for each tag:
 - a. Compute the max probability over previous tags.
 - b. Store the backpointer.
6. Continue until the last word.
7. Backtrack from the last word using backpointers.
8. Return the most probable POS tag sequence.

PROGRAM:

```
states = ['Noun', 'Verb']
observations = ['fish', 'swim']
start_p = {'Noun': 0.6, 'Verb': 0.4}
trans_p = {'Noun': {'Noun': 0.3, 'Verb': 0.7},
           'Verb': {'Noun': 0.8, 'Verb': 0.2}}
emit_p = {'Noun': {'fish': 0.5, 'swim': 0.1},
          'Verb': {'fish': 0.4, 'swim': 0.6}}
V = [{}]
```

path = {}

for y in states:

```
    V[0][y] = start_p[y] * emit_p[y].get(observations[0], 0)
    path[y] = [y]
```

```
for t in range(1, len(observations)):
    V.append({})
    new_path = {}
    for y in states:
        (prob, state) = max((V[t-1][y0]* trans_p[y0][y] * emit_p[y].get(observations[t],
0), y0) for y0 in states)
        V[t][y] = prob
        new_path[y] = path[state] + [y]
    path = new_path

(n, final_state) = max((V[-1][y], y) for y in states)
print("Best path:", path[final_state])
```

OUTPUT:

Best path: ['Noun', 'Verb']

RESULT:

Thus the above POS tagging : Viterbi Decoding python program was implemented and executed successfully.

8. Building POS Tagger

AIM:

To write a python program for Building POS Tagger

Algorithm :

1. Collect a labeled corpus (word/tag pairs).
2. Train a simple model: e.g., most frequent tag for each word.
3. Apply rule-based or statistical backoff for unknown words.
4. Tag a sentence by applying the trained model word by word.

PROGRAM:

```
# Training data (word, tag)
```

```
training_data = [
```

```
    ('the', 'DT'), ('dog', 'NN'), ('barks', 'VBZ'),
```

```
    ('a', 'DT'), ('cat', 'NN'), ('meows', 'VBZ'),
```

```
    ('the', 'DT'), ('dog', 'NN'), ('runs', 'VBZ')
```

```
]
```

```
tag_freq = {}
```

```
for word, tag in training_data:
```

```
    if word not in tag_freq:
```

```
        tag_freq[word] = {}
```

```
        tag_freq[word][tag] = tag_freq[word].get(tag, 0) + 1
```

```
# Step 2: Assign the most frequent tag to each word
```

```
most_freq_tag = {}
```

```
for word, tags in tag_freq.items():
```

```
    most_common = max(tags, key=tags.get)
```

```
most_freq_tag[word] = most_common  
# Step 3: Define the POS tagger  
def pos_tag(sentence):  
    return [(word, most_freq_tag.get(word, 'NN')) for word in sentence]  
# Step 4: Test the tagger  
test_sentence = ['a', 'dog', 'meows']  
tagged = pos_tag(test_sentence)  
print("Tagged Sentence:", tagged)
```

OUTPUT:

Tagged Sentence: [('a', 'DT'), ('dog', 'NN'), ('meows', 'VBZ')]

RESULT:

Thus the above POS tagging : Viterbi Decoding python program was implemented and executed successfully.

9.CHUNKING

AIM:

To write a python program for CHUNKING

ALGORITHM:

1. Use a POS-tagged sentence.
2. Define chunk grammar (e.g., noun phrases).
3. Apply regex patterns to POS sequences.
4. Extract chunks from the sentence.

PROGRAM:

```
import re

pos_tagged = [('The', 'DT'), ('quick', 'JJ'), ('brown', 'JJ'),
              ('fox', 'NN'), ('jumps', 'VBZ'), ('over', 'IN'),
              ('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')]

tag_sequence = ' '.join(tag for _, tag in pos_tagged)

pattern = re.compile(r'\b(DT(?: JJ)* NN)\b')

def find_chunks(pos_tagged, pattern):

    tag_seq = ' '.join(tag for _, tag in pos_tagged)
    word_seq = [word for word, _ in pos_tagged]
    tag_list = tag_seq.split()
    chunks = []
    tags = [tag for _, tag in pos_tagged]
    i = 0
    while i < len(tags) - 2:
        if tags[i] == 'DT':
```

```
j = i + 1
while j < len(tags) and tags[j] == 'JJ':
    j += 1
if j < len(tags) and tags[j] == 'NN':
    chunk = pos_tagged[i:j+1]
    chunks.append(chunk)
    i = j + 1
    continue
i += 1

return chunks

# Run the chunker
chunks = find_chunks(pos_tagged, pattern)
for chunk in chunks:
    print("Chunk:", chunk)
```

OUTPUT:

Chunk: [('The', 'DT'), ('quick', 'JJ'), ('brown', 'JJ'), ('fox', 'NN')]

Chunk: [('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')]

RESULT:

Thus the above Chunking python program was implemented and executed successfully.

10. Building Chunker

AIM:

To write a python program for . Building Chunker

ALGORITHM:

1. Define chunk grammar (e.g., $NP \rightarrow DT\ JJ^*\ NN$).
2. Tokenize and POS tag the sentence.
3. Scan for pattern matches.
4. Extract matched phrases.
5. Store chunked structures.
6. Handle overlapping/non-matching tokens.
7. Optimize chunk grammar.
8. Apply chunking to multiple sentences.
9. Evaluate chunk accuracy.
10. Fine-tune grammar or use ML for chunking.

PROGRAM:

```
import re

sentence = [('The', 'DT'), ('lazy', 'JJ'), ('dog', 'NN'),
            ('barked', 'VBD'), ('at', 'IN'), ('the', 'DT'),
            ('strange', 'JJ'), ('cat', 'NN')]

tags = ''.join(tag for _, tag in sentence)

pattern = re.compile(r'DT(?: JJ)* NN')

chunks = []

i = 0

while i < len(sentence) - 2:
    if sentence[i][1] == 'DT':
        j = i + 1
        while j < len(sentence) and sentence[j][1] == 'JJ':
```

```
        j += 1
    if j < len(sentence) and sentence[j][1] == 'NN':
        chunks.append(sentence[i:j+1])
        i = j + 1
        continue
    i += 1
for chunk in chunks:
    print("Chunk:", chunk)
```

OUTPUT:

Chunk: [('The', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')]

Chunk: [('the', 'DT'), ('strange', 'JJ'), ('cat', 'NN')]

RESULT:

Thus the above Building Chunker python program was implemented and executed successfully.

