| **Course Title** | **Natural Language Processing Lab** | | | | **Course Type** | | **Theory** | |
|---|---|---|---|---|---|---|---|---|
| **Course Code** | B22EA0604 | **Credits** | **2** | | **Class** | | **VI semester** | |
| **Course Structure** | TLP | Credits | Contact Hours | Work Load | Total Number of Classes Per Semester | | Assessment in Weightage | |
| | Theory | - | - | - | | | | |
| | Practice | 1 | 2 | 2 | Theory | Practical | CIE | SEE |
| | Tutorial | - | - | - | | | | |
| | **Total** | **1** | **2** | **2** | | **28** | **50** | **50** |

**COURSE OVERVIEW:**
This course examines fundamental Natural Language Processor and related pre-processing techniques. In particular, the important phases of language recognition will be reviewed, emphasizing the significance of each phase of NLP different. The course will also include concepts such as test word level analysis and syntactic analysis.

**COURSE OBJECTIVE(S):**

- Analyze the natural language text.
- Define the importance of natural language.
- Understand the concepts Text mining.
- Illustrate information retrieval techniques

COURSE CONTENT PRACTICE:

**:**

| **No** | **Title of the Experiment** | **Tools and Techniques** | **Expected Skill /Ability** |
|---|---|---|---|
| **1** | Write a Python program using NLTK to perform basic text preprocessing: tokenization and sentence segmentation. | Python, NLTK, spaCy, Regular Expressions | Ability to clean and prepare text data for NLP tasks; Understanding of basic text preprocessing concepts |
| **2** | Implement word frequency analysis and create word clouds from a given text corpus using Python. | Python, NLTK, WordCloud, matplotlib | Proficiency in text analysis and visualization; Skills in generating meaningful visual representations of text data |
| **3** | Develop a program to perform Parts-of-Speech (POS) tagging on given text using NLTK's built-in tagger | NLTK POS Tagger, Python | Ability to identify grammatical components of text; Understanding of linguistic structure |

| | | | |
|---|---|---|---|
| **4** | Write a program to implement Porter Stemmer and compare its output with WordNet Lemmatizer. | Porter Stemmer, WordNet Lemmatizer, NLTK | Understanding of morphological analysis; Ability to reduce words to their base forms |
| **5** | Create unigram and bigram language models from a small text corpus and calculate probabilities. | Python, NLTK, Probability Calculations | Skills in probabilistic language modeling; Understanding of n-gram models |
| **6** | Implement basic Word Sense Disambiguation using NLTK's Lesk algorithm for given sentences. | NLTK, WordNet, Lesk Algorithm | Ability to determine correct word meanings in context; Understanding of lexical semantics |
| **7** | Write a program to demonstrate Hidden Markov Model basics with a simple POS tagging example. | Python, NumPy, NLTK | Understanding of probabilistic sequence models; Implementation skills for HMM |
| **8** | Develop a rule-based Named Entity Recognition program to identify person names and locations. | NLTK, Regular Expressions, Rule-based Systems | Ability to identify and classify named entities; Skills in pattern matching |
| **9** | Create a basic program to identify subject, verb, and object in simple English sentences. | Python, NLTK Parser, Grammar Rules | Understanding of basic sentence structure; Ability to identify syntactic components |
| **10** | Write a program to implement word-level alignment between English and Hindi sentences. | Python, NumPy, NLTK | Skills in basic machine translation concepts; Understanding of cross-language alignment |
| **11** | Implement a simple sentiment analyzer using NLTK's VADER for movie reviews. | NLTK VADER, Python, Text Classification | Ability to perform sentiment classification; Understanding of opinion mining |
| **12** | Develop a pattern-based question answering system for "what" and "who" questions. | Python, NLTK, Regular Expressions | Skills in information extraction; Ability to implement basic QA systems |

**What does f stand for in a print statement? Can you omit it?**

The f in a **formatted string literal (f-string)** stands for **"formatted"**, introduced in **Python 3.6**. It allows you to **embed variables inside strings** using {}.

**No**, if you remove f, the {} placeholders will not be replaced.

**4. Write a program to implement Porter Stemmer and compare its output with WordNet Lemmatizer.**

```
import nltk

from nltk.stem import PorterStemmer, WordNetLemmatizer

from nltk.corpus import wordnet

from nltk import pos_tag


nltk.download('wordnet')

nltk.download('omw-1.4')

nltk.download('averaged_perceptron_tagger')


def get_wordnet_pos(word):

   tag = pos_tag([word])[0][1][0].upper()

   return {'J': wordnet.ADJ, 'N': wordnet.NOUN, 'V': wordnet.VERB, 'R': wordnet.ADV}.get(tag, 
wordnet.NOUN)


stemmer = PorterStemmer()

lemmatizer = WordNetLemmatizer()

words = ["running", "flies", "better", "studies", "ponies", "children", "leaves", "playing"]


print(f"{'Word':<10} {'Stemmed':<10} {'Lemmatized':<10}")

print("-" * 30)


for word in words:

   print(f"{word:<10} {stemmer.stem(word):<10} {lemmatizer.lemmatize(word, 
get_wordnet_pos(word)):<10}")
```

**Explanation:**

**i) Importing Required Libraries**

```
import nltk

from nltk.stem import PorterStemmer, WordNetLemmatizer

from nltk.corpus import wordnet

from nltk import pos_tag
```

 **PorterStemmer**: Implements the **Porter Stemming Algorithm**.

 **WordNetLemmatizer**: Uses WordNet for **lemmatization**.

 **wordnet**: Provides linguistic data for lemmatization.

 **pos_tag**: Assigns **part-of-speech (POS) tags** to words.

ii) Downloading Required NLTK Resources

```
nltk.download('wordnet')

nltk.download('omw-1.4')

nltk.download('averaged_perceptron_tagger')
```

Ensures that **WordNet** and **POS tagging models** are available.

iii) Function to Get POS Tags for Lemmatization

```
def get_wordnet_pos(word):

   tag = pos_tag([word])[0][1][0].upper()  # Get first letter of POS tag

   return {'J': wordnet.ADJ, 'N': wordnet.NOUN, 'V': wordnet.VERB, 'R': wordnet.ADV}.get(tag, wordnet.NOUN)
```

 Uses **POS tagging** to determine if a word is an **adjective (J), noun (N), verb (V), or adverb (R)**.

 Returns the corresponding **WordNet POS tag**.

 Defaults to **noun** if no specific POS is found.

iv) Initializing Stemmer and Lemmatizer

```
stemmer = PorterStemmer()

lemmatizer = WordNetLemmatizer()
```

 **Stemmer**: Reduces words to their **root form**.

 **Lemmatizer**: Converts words to their **dictionary base form**, considering their POS.

v) List of Words to Process

```
words = ["running", "flies", "better", "studies", "ponies", "children", "leaves", "playing"]
```

- A sample list containing words with different forms (verbs, plurals, comparatives, etc.).

vi) Printing Header

```
print(f"{'Word':<10} {'Stemmed':<10} {'Lemmatized':<10}")
print("-" * 30)
```

Prints a **table header** for readability.

vii) Processing Each Word

```
for word in words:
    print(f"{word:<10} {stemmer.stem(word):<10} {lemmatizer.lemmatize(word,
get_wordnet_pos(word)):<10}")
```

 Iterates over each **word** in the list.

 **Applies Stemming** using stemmer.stem(word).

 **Applies Lemmatization** using lemmatizer.lemmatize(word, get_wordnet_pos(word)) (with correct POS).

 Prints the **original word, its stemmed version, and its lemmatized version**.

**Key Differences Between Stemming and Lemmatization**

| **Word** | **Stemmed** | **Lemmatized** |
|---|---|---|
| running | run | run |
| flies | fli | fly |
| better | better | good |
| studies | studi | study |

| **Word** | **Stemmed** | **Lemmatized** |
|---|---|---|
| ponies | poni | pony |
| children | children | child |
| leaves | leav | leaf |
| playing | play | play |

**Summary of Behavior**

1. **Porter Stemmer** applies **rule-based suffix stripping**, leading to **non-standard words** (e.g., *studies* → *studi*).
2. **WordNet Lemmatizer** uses **dictionary lookups** with **POS tagging**, producing **correct dictionary words** (e.g., *better* → *good*).

5. Create unigram and bigram language models from a small text corpus and calculate probabilities.

```
import nltk
nltk.download('punkt_tab')


import nltk
from collections import Counter
from nltk.util import bigrams
from nltk.tokenize import word_tokenize


# Download necessary resources
nltk.download('punkt')


# Sample corpus
corpus = "I love natural language processing. I love machine learning."


# Tokenize and lowercase the text
tokens = word_tokenize(corpus.lower())


# **Unigram Model**
```

```
unigram_counts = Counter(tokens)
total_unigrams = sum(unigram_counts.values())

# Compute Unigram Probabilities: P(word) = count(word) / total_words
unigram_probs = {word: count / total_unigrams for word, count in unigram_counts.items()}

# **Bigram Model**
bigram_counts = Counter(bigrams(tokens))

# Compute Bigram Probabilities: P(word2 | word1) = count(word1, word2) / count(word1)
bigram_probs = {pair: count / unigram_counts[pair[0]] for pair, count in bigram_counts.items()}

# **Output Probabilities**
print("\nUnigram Probabilities:")
for word, prob in unigram_probs.items():
    print(f"P({word}) = {prob:.4f}")

print("\nBigram Probabilities:")
for (w1, w2), prob in bigram_probs.items():
    print(f"P({w2} | {w1}) = {prob:.4f}")
```

**Output:**

**Unigram Probabilities:**

```
P(i) = 0.1818
P(love) = 0.1818
P(natural) = 0.0909
P(language) = 0.0909
P(processing) = 0.0909
P(.) = 0.1818
P(machine) = 0.0909
P(learning) = 0.0909
```

**Bigram Probabilities:**

```
P(love | i) = 1.0000

P(natural | love) = 0.5000

P(language | natural) = 1.0000

P(processing | language) = 1.0000

P(. | processing) = 1.0000

P(i | .) = 0.5000

P(machine | love) = 0.5000

P(learning | machine) = 1.0000

P(. | learning) = 1.0000
```

**Explanation:**

 Tokenization

- The text is converted to lowercase for consistency.
- word_tokenize() splits the text into words.

 Unigram Model

- Counts occurrences of each word.
- Computes P(word) = count(word) / total_words.

 Bigram Model

- Extracts (word1, word2) pairs using bigrams().
- Counts occurrences of each bigram.
- Computes P(word2 | word1) = count(word1, word2) / count(word1).

 Output Probabilities

- Prints unigram probabilities (individual word probabilities).
- Prints bigram probabilities (conditional probabilities of word pairs).

**i) Import Required Libraries**

```
import nltk

from collections import Counter

from nltk.util import bigrams
```

from nltk.tokenize import word_tokenize

- **nltk**: Used for natural language processing.
- **Counter**: Counts occurrences of words (unigrams) and word pairs (bigrams).
- **bigrams**: Generates bigram pairs from a sequence of words.
- **word_tokenize**: Splits text into individual words.

### ii) Download Required Resources

**nltk.download('punkt')**

**Downloads Punkt tokenizer, which helps split sentences into words.**

### iii) Define Sample Corpus

**corpus = "I love natural language processing. I love machine learning."**

### iv) Tokenization

**tokens = word_tokenize(corpus.lower())**

- **Converts text to lowercase (ensures consistency).**
- **Splits the sentence into individual words**

### v) Create the Unigram Model

**unigram_counts = Counter(tokens)**

**total_unigrams = sum(unigram_counts.values())**

**Counts occurrences of each word (unigram).**

**Finds the total number of words in the corpus.**

### vi) Calculate Unigram Probabilities

**unigram_probs = {word: count / total_unigrams for word, count in unigram_counts.items()}**

**Formula:**

- **P(word)=total number of wordscount(word)**

### vii) Create the Bigram Model

**bigram_counts = Counter(bigrams(tokens))**

- **Uses bigrams(tokens) to generate word pairs**
- **Counts occurrences of each bigram (word1, word2).**

**viii) Calculate Bigram Probabilities**

**bigram_probs = {pair: count / unigram_counts[pair[0]] for pair, count in bigram_counts.items()}**

- **P(word2|word1)=count(word1)count(word1, word2)**

**ix) Print the Probabilities**

```
print("\nUnigram Probabilities:")
for word, prob in unigram_probs.items():
  print(f"P({word}) = {prob:.4f}")
```

- **Prints unigram probabilities in a readable format.**

```
print("\nBigram Probabilities:")
for (w1, w2), prob in bigram_probs.items():
  print(f"P({w2} | {w1}) = {prob:.4f}")
```

- **Prints bigram probabilities, showing the probability of word2 given word1.**

**� Unigram Model: Estimates the probability of single words.**

**� Bigram Model: Estimates the probability of a word given the previous word.**

**� Limitation: This model does not handle unseen bigrams (zero probabilities for unseen word pairs)**

**6)** Implement basic Word Sense Disambiguation using NLTK's Lesk algorithm for given sentences.

**What is Word Sense Disambiguation (WSD)?**

**Word Sense Disambiguation (WSD)** is the process of identifying the correct meaning (sense) of a word in a given context when the word has multiple meanings.

For example, consider the word **"bank"**:

- **"I deposited money in the bank." → Bank (financial institution)**
- **"He sat on the river bank and relaxed." → Bank (side of a river)**

**Dictionary-based (Lesk Algorithm)** – Uses word definitions (glosses) from a dictionary like WordNet.

**CODE:**

```
import nltk
nltk.download('punkt_tab')

import nltk
from nltk.wsd import lesk
from nltk.tokenize import word_tokenize
from nltk.corpus import wordnet

# Download required NLTK resources
nltk.download('punkt')
nltk.download('wordnet')

# Sample sentences for disambiguation
sentences = [
    "The bank will not approve my loan.",
    "He sat on the river bank and enjoyed the view."
]

# Target word to disambiguate
target_word = "bank"

# Perform Word Sense Disambiguation
for sentence in sentences:
    best_sense = lesk(word_tokenize(sentence), target_word)
    print(f"Sentence: {sentence}")
    print(f"Best Sense: {best_sense.name()} - {best_sense.definition()}\n")
```

**Code Explanation**

1. **Import Necessary Libraries**

   ```
   import nltk
   from nltk.wsd import lesk
   ```

**from nltk.tokenize import word_tokenize**

**from nltk.corpus import wordnet**

- **nltk.wsd.lesk: Implements the Lesk algorithm for WSD.**
- **word_tokenize(): Splits the sentence into words.**
- **wordnet: Provides definitions (synsets) for words.**

## 2. Download Required Resources

**nltk.download('punkt')**

**nltk.download('wordnet')**

- **punkt: Needed for tokenization.**
- **wordnet: Provides word meanings.**

**3. Define Sample Sentences**

```
sentences = [
  "The bank will not approve my loan.",
  "He sat on the river bank and enjoyed the view."
]
```

**The word "bank" has different meanings in both sentences.**

- **Financial institution (sentence 1).**
- **Side of a river (sentence 2).**

**4. Apply the Lesk Algorithm**

```
best_sense = lesk(word_tokenize(sentence), target_word)
```

- **word_tokenize(sentence): Tokenizes the sentence.**
- **lesk(): Finds the most relevant meaning based on the sentence context.**

**5. Print the Best Sense and Its Definition**

```
print(f"Best Sense: {best_sense.name()} - {best_sense.definition()}\n")
```

- **best_sense.name(): Displays the sense name.**
- **best_sense.definition(): Shows the meaning.**

**Output:**

**Sentence:** The bank will not approve my loan.

**Best Sense:** bank.n.07 - a financial institution that accepts deposits and channels the money into lending activities

**Sentence:** He sat on the river bank and enjoyed the view.
**Best Sense:** bank.n.01 - sloping land (especially the slope beside a body of water)

**Correctly disambiguates "bank"** based on the sentence context.

**Summary**

✅ **Lesk algorithm** finds the most relevant meaning using **WordNet definitions**.

✅ Works **without training data** (dictionary-based).