MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

CSE_3258
NATURAL LANGUAGE PROCESSING LAB MANUAL

THIRD YEAR

(2022-CURRICULUM)

VI SEMESTER

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING-ARTIFICIL ENGINEERING

MANIPAL INSTITUTE OF TECHNOLOGY MAHE, BENGALURU

KARNATAKA - 560064

Contents

	2. Format the dataset to include: User queries, Intent labels (e.g., "Order Status", "Product Inquiry", "Refund Request"), Entities (e.g., "Order ID", "Product Name") Deliverable: A labeled dataset ready for training. 3. Build a model to classify user intent. Train models like: Logistic Regression, SVM Transformer-based models (e.g., BERT or DistilBERT) (Optional) 4. Evaluate accuracy, precision, recall, and F1-score for intent classification. 5. Extract entities from user queries. A trained Named Entity Recognition (NER) model capable of extracting entities like "Order ID", "Date", or "Product Name".
14	Case Study 4: News Article Classification and Summarization A news agency wants to automatically classify and summarize articles for better organization and quick reader consumption. The system should classify articles into predefined categories (e.g., "Politics," "Sports," "Technology") and generate concise summaries. 1. Collect and preprocess news article data: Use publicly available datasets (e.g., BBC News Classification Dataset, AG News, or Reuters-21578). 2. Preprocess the text: Remove special characters, stopwords, numbers, and URLs. Tokenize and lemmatize the text. 3. Analyze: Distribution of articles across categories. Average word count and sentence length of articles. Visualize word frequency using bar charts or word clouds for each category. Deliverable: Insights and visualizations highlighting dataset characteristics. 4. Text Classification: Classify articles into predefined categories. Experiment with multiple models. Use Traditional: Naïve Bayes, SVM and Deep Learning: LSTM, CNNs 5. Evaluate: Split the dataset into training and testing sets, and evaluate using precision, recall, and F1-score. 6. Text Summarization: Generate concise summaries for news articles. Implement extractive summarization and implement abstractive summarization. Compare the quality of summaries generated by extractive and abstract approaches.

Course objectives:

This laboratory course enables students to

• Apply fundamental algorithms and techniques in the area of NLP

• Implement, design and test Language modelling

• Apply the concepts of NLP techniques to solve real-world problems

• Implement basic algorithms in classification applied to text

Course outcomes:

At the end of this course, students will gain the

1 Ability to understand and explore natural language processing (NLP) libraries in Python

2 Learn various techniques for implementing NLP including parsing & text processing.

3 Understand how to use NLP for text feature engineering

4 Ability to comprehend and understand the language models.

5 Ability to apply, implement and text classification models and interpret the results.

Evaluation plan

• **Internal Assessment Marks: 60%**

✓ Students must work out the python programs using NLTK.

✓ Students shall submit the lab code along with the results obtained under every program as per the schedule. Eg: At the end of lab 2, both lab 1 & lab 2 programs should be submitted.

• **End semester assessment of 2-hour duration: 40%**

✓ A complete Python program with NLTK will be asked. The given program must be worked out in code and compiled. The **Python code** should be submitted. The name of file should be like **regno.c Eg: 20100345.**

Sample Lab Observation Note Preparation

Title: SIMPLE C PROGRAMS

Sample input and output: Screen shot of result – must be included.

Python NLTK Tutorial 1

Lab Overview

– Introduction to Natural Language Toolkit (NLTK)

– Python quick overview;

– Lexical analysis: Word and text tokenizer;

– n-gram and collocations;

– NLTK corpora;

– HMM Model

What is NLTK?

Natural Language Toolkit (NLTK) is a popular platform for building Python programs to work with human language data; i.e., for Natural Language Processing. It is accompanied by a book that explains the underlying concepts behind the language processing tasks supported by the toolkit. NLTK is intended to support research and teaching in NLP or closely related areas, including empirical linguistics, cognitive science, artificial intelligence, information retrieval, and machine learning.

We will start with a quick Python introduction, but if you would like to learn more about Python, there are many resources on the Web and books. For a simple beginner Python tutorial take a look at:

http://www.tutorialspoint.com/python/python_tutorial.pdf.

Python quick overview;

Identifiers

Python identifier is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter A to Z or a to z, or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9). Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, Variable and variable are two different identifiers in Python.

Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced. The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

Quotation

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

Examples:

word = 'word'

sentence = "This is a sentence."

paragraph = """This is a paragraph. It is made up of multiple lines and sentences."""

Data types, assigning and deleting values

Python has five standard data types:

• numbers;

• strings;

• lists;

• tuples;

• dictionaries.

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables. The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

For example:

```
counter = 100 # An integer assignment

miles = 1000.0 # A floating point

name = "John" # A string
```

Lists

```
print(len([1, 2, 3]))              # 3 - length

print([1, 2, 3] + [4, 5, 6])       # [1, 2, 3, 4, 5, 6] - concatenation

print(['Hi!'] * 4)                 # ['Hi!', 'Hi!', 'Hi!, 'Hi!'] - repetition

print(3 in [1, 2, 3])              # True - checks membership for x in [1, 2, 3]:

print(x)                           # 1 2 3 - iteration
```

Some of the useful built-in functions useful in work with lists are max, min, cmp, len, list (converts tuple to list), etc. Some of the list-specific functions are list.append, list.extend, list.count, etc.

Tuples

```
tup1 = ('physics', 'chemistry', 1997, 2000)

tup2 = (1, 2, 3, 4, 5, 6, 7)

print(tup1[0]) # prints: physics print(tup2[1:5]) # prints: [2, 3, 4, 5]
```

Basic tuple operations are same as with lists: length, concatenation, repetition, membership and iteration.

Dictionaries

```
dict = {'Name':'Zara', 'Age':7, 'Class':'First'}
```

```
dict['Age'] = 8 # update existing entry

dict['School'] = "DPS School" # Add new entry

del dict['School'] # Delete existing entry
```

List comprehension

Comprehensions are constructs that allow sequences to be built from other sequences. Python 2.0 introduced list comprehensions and Python 3.0 comes with dictionary and set comprehensions. The following is the example:

```
a_list = [1, 2, 9, 3, 0, 4]

squared_ints = [e**2 for e in a_list]

print(squared_ints) # [ 1, 4, 81, 9, 0, 16 ]
```

This is same as:

```
a_list = [1, 2, 9, 3, 0, 4]

squared_ints = []

for e in a_list:

squared_ints.append(e**2)

print(squared_ints) # [ 1, 4, 81, 9, 0, 16 ]
```

Now, let’s see the example with if statement. The example shows how to filter out non integer types from mixed list and apply operations.

```
a_list = [1, '4', 9, 'a', 0, 4]

squared_ints = [ e**2 for e in a_list if type(e) is int ]

print(squared_ints) # [ 1, 81, 0, 16 ]
```

However, if you want to include if else statement, the arrangement looks a bit different.

```
a_list = [1, ’4’, 9, ’a’, 0, 4]

squared_ints = [ e**2 if type(e) is int else 'x' for e in a_list]

print(squared_ints) # [1, 'x', 81, 'x', 0, 16]
```

You can also generate dictionary using list comprehension:

```
a_list = ["I", "am", "a", "data", "scientist"]

science_list = { e:i for i, e in enumerate(a_list) }

print(science_list) # {'I': 0, 'am': 1, 'a': 2, 'data': 3, 'scientist': 4}
```

... or list of tuples:

```
a_list = ["I", "am", "a", "data", "scientist"]
```

science_list = [(e,i) for i, e in enumerate(a_list)]

print(science_list) # [('I', 0), ('am', 1), ('a', 2), ('data', 3), ('scientist’, 4)]

String handling

Examples with string operations:

str = 'Hello World!'

print(str) # Prints complete string

print(str[0]) # Prints first character of the string

print(str[2:5]) # Prints characters starting from 3rd to 5th

print(str[2:]) # Prints string starting from 3rd character

print(str*2) # Prints string two times

print(str + "TEST") # Prints concatenated string

Other useful functions include join, split, count, capitalize, strip, upper, lower, etc.

Example of string formatting:

print("My name is %s and age is %d!" % ('Zara',21))

IO handling

Python has two built-in functions for reading from standard input: raw_input and input.

str = raw_input("Enter your input: ")

print("Received input is : ", str)

File opening

To handle files in Python, you can use function open. Syntax:

file object = open(file_name [, access_mode][, buffering])

One of the useful packages for handling tsv and csv files is csv library.

Functions

An example how to define a function in Python:

def functionname(parameters):

"function_docstring"

function_suite return [expression]

Experiment 1

WRITE A PROGRAM GIVEN A PIECE OF TEXT, WE WANT TO SPLIT THE TEXT AT ALL SPACES (INCLUDING NEW LINE CHARACTERS AND CARRIAGE RETURNS) AND PUNCTUATION MARKS.

Explanation: we should use of regular expressions. Regular expressions (regex) are a powerful tool in Natural Language Processing (NLP) for handling and processing text data. They allow for efficient and flexible manipulation of strings, which is essential in many NLP tasks. Regular expressions are a foundational tool in NLP for text processing and pattern matching. They are highly efficient for structured and semi-structured data but should be complemented with advanced NLP techniques (like machine learning or deep learning) for more complex tasks.

1. **Regex Pattern r"[^\w]+"**:
   - Matches one or more non-word characters (anything that is not a letter, digit, or underscore).
   - This effectively splits the text at spaces, punctuation, and other special characters.
2. **Filtering Empty Strings**:
   - After splitting, some elements may be empty strings ("") because of consecutive punctuation or leading/trailing delimiters.
   - We filter these out using a list comprehension.

Input

"This is a sample text,with some punctuation marks! Also, newlines and carriage returns.Let's split it."

Output

```
['This', 'is', 'a', 'sample', 'text', 'with', 'some', 'punctuation', 'marks', 'Also', 'newlines', 'and', 'carriage', 'returns', 'Let', 's', 'split', 'it']
```

Additional Exercise

1. Write a regex pattern to extract all dates in the format DD/MM/YYYY from a paragraph of text.
2. How would you use regular expressions to identify all phone numbers in a text document, regardless of their format (e.g., 123-456-7890, (123) 456-7890, etc.)?

Experiment 2:

WRITE A PROGRAM TO REMOVE THE FIRST AND LAST CHARACTERS IF THEY ARE NOT LETTERS OR NUMBERS FROM A GIVEN SENTENCE.

This program effectively removes unwanted characters from the edges of a given sentence using Python's regular expression module. It simplifies text preprocessing tasks in natural language processing and data cleaning applications.

1. Regular Expression Logic:

- **^[^a-zA-Z0-9]+**: Matches one or more non-alphanumeric characters ([^a-zA-Z0-9]) at the **start** of the string (^).
- **[^a-zA-Z0-9]+$**: Matches one or more non-alphanumeric characters at the **end** of the string ($).
- |: Ensures that both the start and end patterns are checked.

2. re.sub() Function:

- This function replaces all matches of the given pattern with an empty string (''), effectively removing non-alphanumeric characters at the edges.

3. Workflow:

1. Input a sentence with possible punctuation or special characters at the edges.
2. Apply the regex to remove unwanted characters from the start and end.
3. Return the cleaned sentence.

Input: "!Hello, World?!"
Output : Hello, World

Additional Exercise

1. Write a program to count the number of characters that are not letters or numbers in a given sentence.
2. Write a program to replace all characters that are not letters or numbers in a sentence with a specified character.

Experiment 3:

WRITE A PROGRAM TO SPLIT A WORD INTO PAIR'S AT ALL POSSIBLE POSITIONS. FOR EXAMPLE, CARRIED WILL BE SPLIT INTO {C, ARRIED, CA ,RRIED, CAR, RIED, CARR, IED, CARRI, ED, CARRI, D}.

This program efficiently splits a word into all possible pairs using Python string slicing. The output can be used in text processing, language analysis, or any application requiring string segmentation.

Explanation:

1. Function split_word_into_pairs:

- **Purpose:** Generate all possible pairs by splitting the word at every position.
- **Parameters:**
  - word (str): The word to split.
- **Logic:**
  - Use slicing to split the word at index i.
  - Append the split parts as a tuple to a list.

2. Loop through Word:

```
for i in range(1, len(word)):
   pairs.append((word[:i], word[i:]))
```

- **word[:i]**: Takes the substring from the start of the word to index i (excluding i).
- **word[i:]**: Takes the substring from index i to the end of the word.
- **Example:** For the word CARRIED:
  - When i = 1: word[:1] = C, word[1:] = ARRIED.
  - When i = 2: word[:2] = CA, word[2:] = RRIED.

3. Return Result:

- The program returns a list of tuples containing all the pairs.

Input :

CARRIED

Output:

```
Original Word: CARRIED
Pairs:
('C', 'ARRIED')
('CA', 'RRIED')
('CAR', 'RIED')
('CARR', 'IED')
('CARRI', 'ED')
('CARRIE', 'D')
```

Additional Exercise

1. Write a program to generate all possible prefixes and suffixes of a given word.

2. Write a program to split a word into two parts at random positions and print all splits.

Experiment 4:

WRITE A PROGRAM TO FIND OUT THE FREQUENCIES OF DISTINCT WORDS, GIVEN A SENTENCE USING N-GRAMS.

Definition:

An **N-gram** is a contiguous sequence of N items from a given text or speech. These "items" can be words, characters, or even syllables, but in natural language processing (NLP), N-grams are typically formed from **words** or **characters**. The term "N" refers to the number of items in the sequence. N-grams are a fundamental concept in text analysis and natural language processing. They are used to represent text sequences and model word dependencies, making them useful for a variety of applications like language modeling, text classification, and information retrieval. Despite their simplicity and effectiveness, N-grams are limited by data sparsity and their inability to capture long-range dependencies between words.

For example:

- A **unigram** is an N-gram where N=1 (a single word).
- A **bigram** is an N-gram where N=2 (a pair of consecutive words).
- A **trigram** is an N-gram where N=3 (a sequence of three consecutive words), and so on.

Types of N-grams:

1. **Unigrams (1-grams):**
   - **Definition**: A unigram is a single word or token from the sentence.
   - **Example**:
     Sentence: "I love Python"
     Unigrams: ['I', 'love', 'Python']
2. **Bigrams (2-grams):**
   - **Definition**: A bigram is a pair of consecutive words in the sentence.
   - **Example**:
     Sentence: "I love Python"
     Bigrams: [('I', 'love'), ('love', 'Python')]
3. **Trigrams (3-grams):**
   - **Definition**: A trigram is a sequence of three consecutive words.
   - **Example**:
     Sentence: "I love Python"
     Trigrams: [('I', 'love', 'Python')]
4. **Higher-order N-grams (4-grams, 5-grams, etc.):**
   - These are sequences of 4, 5, or more consecutive words.
   - **Example (4-gram)**:
     Sentence: "I love programming in Python"
     4-grams: [('I', 'love', 'programming', 'in'), ('love', 'programming', 'in', 'Python')]

Definition:

A **bigram** is a type of N-gram where N=2. It refers to a sequence of two consecutive items (usually words) in a text or sentence. Bigrams are widely used in text processing, language modeling, and natural language processing tasks.

Formation of Bigrams:
To create bigrams, you pair each word in a sentence with the word that immediately follows it.
Example:
For the sentence:
"This is a sample sentence"
The bigrams are:

- ('This', 'is')
- ('is', 'a')
- ('a', 'sample')
- ('sample', 'sentence')

Use Cases of Bigrams:

1. **Language Modeling:**
   - Bigrams are used to calculate the probability of a word occurring given the previous word.
   - Example: Predicting the next word based on bigram probabilities.
2. **Text Analysis:**
   - Analyze word pairs to find commonly occurring phrases or relationships between words.
3. **Text Generation:**
   - Used in applications like autocomplete or text generation to predict the next word.
4. **Spam Detection:**
   - Analyze frequent bigrams in spam emails or messages to identify patterns.

Mathematics of Bigrams:

The **probability** of a sentence using bigrams is calculated as:

$P(W)=P(w1)\cdot P(w2|w1)\cdot P(w3|w2)\cdot\ldots$

Where:

- $P(w2|w1)$ is the probability of word w2 occurring given that w1 has already occurred.

Example:

Sentence: "I love Python"
Bigram probabilities:

- $P(love|I)$: Probability of "love" given "I".
- $P(Python|love)$: Probability of "Python" given "love".

Output:

```
('This', 'is'): 1
('is', 'a'): 1
('a', 'sample'): 1
('sample', 'sentence'): 1
('sentence', 'and'): 1
('and', 'this'): 1
('this', 'is'): 1
('is', 'another'): 1
('another', 'sample'): 1
```

Advantages of Bigrams:

1. **Contextual Understanding:**
   - They provide limited context by considering two consecutive words, improving upon unigrams.
2. **Simplicity:**
   - Easier to implement and compute compared to higher-order N-grams (e.g., trigrams or 4-grams).

Limitations of Bigrams:

1. **Limited Context:**
   - They only capture the relationship between two consecutive words, ignoring the broader context of the sentence.
2. **Data Sparsity:**
   - Some bigrams may rarely occur in the data, making them hard to generalize.

Application in Your Program:

In the provided program, bigrams are generated by setting N=2. This produces all possible pairs of consecutive words in the given sentence and calculates their frequencies.

Example Output for Bigram Analysis:

```
Sentence: "This is a sample sentence"
Bigrams and frequencies:
```

- ('This', 'is'): 1
- ('is', 'a'): 1
- ('a', 'sample'): 1
- ('sample', 'sentence'): 1

Additional Exercise:

1. Write a program to calculate the probabilities of each n-gram in a sentence or text.
2. Write a program to generate n-grams in reverse order (e.g., starting from the end of the sentence).

Experiment 5:
WRITE A PROGRAM TO REMOVE DIGITS FROM A GIVEN SENTENCE USING GREEDY TOKENIZER.

A **greedy tokenizer** is a text processing tool or algorithm used in natural language processing (NLP) that aims to break down text into tokens (such as words, phrases, or subword units) in a way that maximizes the number of tokens, often by consuming as much of the input text as possible at each step. The "greedy" part refers to the approach of always selecting the largest possible token or match at each step, without considering longer-term consequences.

How it works:

- A greedy tokenizer typically works by searching through the text and matching the longest possible string or pattern that can form a valid token, then moving to the next unmatched portion of the text.
- For example, if the tokenizer is designed to recognize words and punctuation, it will greedily consume the longest match (which could be a word, a punctuation mark, or even a combination of characters) and treat it as a token before moving on to the next part of the text.

Example:

Consider the text:

"I love NLP!"

A greedy tokenizer might tokenize it as:

- "I"
- "love"
- "NLP"
- "!"

Here, the tokenizer greedily chooses "NLP" as a single token (even though it could technically be broken down into subword units like "N" and "LP" if the tokenizer were less greedy), since it's the longest valid match.

Explanation:

1. **Tokenization:**
    - The sentence is split into individual tokens (words) using the split() function, which uses spaces as the delimiter. This is a greedy approach since it greedily splits based on the longest valid token (a word in this case).
2. **Greedy Removal of Digits:**
    - The program filters out tokens that contain digits using a regular expression r'\d'. The re.search(r'\d', token) checks if a token contains any digits.
    - If a token contains digits, it is excluded from the list of tokens.
3. **Reconstruction:**
    - The filtered tokens (words without digits) are then joined back into a sentence using ' '.join(tokens).

Example:

Input:

"Hello, I have 2 apples and 3 bananas."

Output:
"Hello, I have apples and bananas."

Key Characteristics:

1. **Efficiency:** The greedy tokenizer is typically fast and simple, making it a popular choice in situations where processing speed is critical.
2. **Simplicity:** The logic is straightforward — always try to make the largest valid token match at each step.
3. **Potential for Overfitting:** In some cases, the greedy approach might break down when dealing with more complex patterns, as it can miss context or nuances in the text.

Use Cases:

- **Word Tokenization:** For general text processing where we want to split text into words, especially when we don't need to handle complex or ambiguous tokenization rules.
- **Subword Tokenization:** In some machine learning models, greedy tokenization is used for breaking down text into subword units or byte pair encodings (BPE), where the algorithm greedily combines frequent character pairs to form new tokens.

Limitations:

- **Overly Simplistic:** Greedy tokenization doesn't always handle ambiguities or complex word boundaries well. For instance, it might not handle compound words or contractions as well as other, more sophisticated tokenization methods.
- **May Ignore Context:** Because the tokenizer makes decisions based solely on the longest match at each step, it might miss tokens that could be meaningful in the context of the text as a whole.

Input:
"Hello, I have 2 apples and 3 bananas."
Output:
"Hello, I have apples and bananas."

Additional Exercise

1. Write a program to count how many digits are present in a given sentence.
2. Write a program to extract and print all digits from a sentence using a greedy tokenizer.
3. Write a program that greedily tokenizes a sentence but prioritizes specific patterns, such as dates (\d{1,2}/\d{1,2}/\d{2,4}) and email addresses ([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}), over general tokens.

   Input Example:
   "Contact me at abc.mahe@manipal.edu before 02/28/2025."

   Expected Output:
["Contact me at ", " abc.mahe@manipal.edu ", " before ", "02/28/2025", "."]

Experiment 6:
DEMONSTRATE NOISE REMOVAL FOR ANY TEXTUAL DATA AND REMOVE REGULAR EXPRESSION PATTERN SUCH AS HASHTAG FROM TEXTUAL DATA.

Noisy Channel Model:
The **noisy channel model** is a theoretical framework used primarily in the fields of natural language processing (NLP) and information theory to explain how a message can be corrupted or distorted during transmission. The concept comes from communication theory, where a "noisy channel" refers to a system that introduces errors or noise into the data as it passes from the sender to the receiver.
In the context of NLP, the noisy channel model is often used for tasks like **machine translation**, **speech recognition**, and **spell checking**, among others.
Mathematical Formulation:
The noisy channel model is based on **Bayes' Theorem**, which states that:
$P(X|Y)= P(Y)P(Y|X)\cdot P(X)$
Where:

- X is the original, intended message (the correct sequence of words or letters we are trying to recover).
- Yis the noisy, observed message (the sequence that was actually received, which may contain errors or distortions).
- P(X|Y)is the probability of X given Y, i.e., the probability that X was the original message given the noisy Y we observed.
- P(Y|X) is the likelihood, i.e., the probability of receiving Y given that X was the original message.
- P(X) is the prior probability of X, i.e., the likelihood of X occurring in the first place.
- P(Y) is the evidence, i.e., the total probability of receiving Y(which we can ignore if we're only concerned with maximizing P(X|Y).

In practice, we are interested in maximizing P(X|Y), or finding the most likely original message X given the noisy message Y.
Simplifying the Model:
To simplify, we use the following approximation:
$P(X|Y) \approx argXmaxP(Y|X)\cdot P(X)$
This means that we aim to find the X (the intended message) that maximizes the product of:

- The **likelihood** of observing Y given X (how probable the observed output is given the candidate input).
- The **prior** probability of X (how likely this message is in general).

How It Works in NLP Tasks:

1. **Machine Translation:**
    - In machine translation, the noisy channel model works by modeling the translation as a noisy channel. The goal is to **recover the original sentence in the source language** (i.e., the most probable translation) from the noisy observed translation in the target language.

    - For example, if the sentence in English is "I am happy," and the noisy translation is "Estoy feliz" in Spanish, the noisy channel model attempts to figure out the most likely original sentence (in English) given the noisy input in Spanish.
2. **Speech Recognition:**
    - In speech recognition, the model treats the **spoken audio** as the noisy version of the text. The system's job is to recover the original transcription (the most likely sequence of words) from the audio, which may contain noise or distortions.
    - For example, if the input speech contains background noise, the noisy channel model helps to decipher the most probable intended words by considering the context.
3. **Spell Checking:**
    - In spell checking, the noisy channel model is used to correct spelling errors in a given word. The observed word (which may be misspelled) is treated as the noisy version of the correct word.
    - The task is to find the most likely correct word by looking at the noisy (misspelled) word and comparing it to a dictionary of valid words.

Advantages of the Noisy Channel Model:

1. **Flexibility:** The model is very flexible and can be applied to various NLP tasks such as translation, speech recognition, and text correction.
2. **Modularity:** The model treats translation or recognition as a probabilistic problem, making it easier to incorporate different components (e.g., language models, acoustic models) into the system.
3. **Statistical Foundation:** It is grounded in probability theory, which makes it suitable for handling uncertainties and ambiguities in language.

Limitations of the Noisy Channel Model:

1. **Data Sparsity:** It can suffer from data sparsity, particularly for less frequent words or phrases, leading to poor performance in low-resource scenarios.
2. **Complexity:** The model can be computationally expensive, especially when dealing with large datasets or complex tasks.
3. **Dependence on Accurate Models:** The performance of the noisy channel model heavily depends on the accuracy of the prior probability (language model) and the likelihood function (error model). If these models are inaccurate, the overall performance may degrade.

Input :

"This is a #sample text! Remove the #hashtag and noise! Visit @example."

Output :

"This is a sample text Remove the hashtag and noise Visit example"

Additional Exercise

1. Write a program to detect and remove emoticons or emojis from textual data.
2. Write a program to normalize text by removing extra whitespace and converting all text to lowercase.

3. Write a python program to extract all dates in various formats (DD/MM/YYYY, MM-DD-YYYY, Month Day, Year) from mixed text.
4. Write a python program to extract phone numbers from the text in various formats and standardize them to a common format.

Lab 7: Q. PERFORM LEMMATIZATION AND STEMMING USING PYTHON LIBRARY NLTK

Lemmatization and Stemming

Stemming is a process where the word is reduced to its root form by removing prefixes or suffixes. The root may not necessarily be a valid word, but it helps in text processing by reducing variations of the same word.

Lemmatization is a more sophisticated process compared to stemming, where the word is reduced to its base form (lemma) based on its meaning. It ensures that the root word is a valid word in the language.

Solution : import nltk

```
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.corpus import stopwords

# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('stopwords')
# Download the 'punkt_tab' resource
nltk.download('punkt_tab')

# Initialize the stemmer and lemmatizer
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()


# Output the results
print("Original Tokens:", tokens)
print("Filtered Tokens (without stopwords):", filtered_tokens)
print("Stemmed Tokens:", stemmed_tokens)
print("Lemmatized Tokens:", lemmatized_tokens)
```

Additional Exercise:

1. Write a python program to design a custom tokenizer that can handle multiple types of noise (hashtags, mentions, URLs, punctuation, etc.) and then apply stemming and lemmatization.
2. Write a python program to create a Named Entity Recognition (NER) system to identify entities (like people, organizations, dates) and perform text normalization (remove or standardize text such as dates, monetary values, and numbers).

Lab 8: DEMONSTRATE OBJECT STANDARDIZATION SUCH AS REPLACE SOCIAL MEDIA SLANGS FROM A TEXT.

Object Standardization: In this context, we are standardizing informal objects (social media slangs) in a text into more formal versions. This is useful for natural language processing tasks to maintain consistent meaning across texts.

Regular Expressions: The re.sub() function is used to find and replace slang terms with their standardized equivalents. This method works efficiently, even with large texts.

Additional exercise:

1. Create a dictionary for common slangs and another dictionary for emojis. Implement a function that replaces slangs, abbreviations, and emojis with their formal counterparts. Normalize any irregular spaces or punctuation marks (e.g., "hello!!!" -> "hello!"). Test the function on a variety of text inputs containing slangs, emojis, and multiple punctuation.
2. Given a sentence with mixed case, slang words, and irregular punctuation. Write a python program that: Replaces slang with full words, Standardizes the punctuation, Converts the text to lowercase.

Lab 9: Write a python program to perform part of speech tagging on any textual data.

Part of Speech (POS) Tagging is the process of assigning a part of speech (such as noun, verb, adjective, etc.) to each word in a sentence. In Python, the Natural Language Toolkit (NLTK) provides an easy way to perform POS tagging.
Steps:

1. Tokenize the text into words.
2. Use NLTK's POS tagger to tag each word with its corresponding part of speech.
3. Display the results.

Noun Phrase (NP) extraction can be improved by using chunking techniques in NLTK. Chunking groups words into phrases based on their POS tags

Additional Exercise:

1. Write a python program to implement a system that performs the following on a given text:

For each word in a sentence, identify its part of speech (noun, verb, adjective, etc.).
Analyze the syntactic structure of the sentence by identifying how words depend on each other (e.g., subject-verb-object relationships).
Identify and extract all noun phrases (groups of words containing a noun).
Named Entity Recognition (NER): Extract named entities like people, organizations, and locations.
Visualization: Visualize the dependency tree to better understand the sentence structure.

2. Extend the system including this feature

Improve the noun phrase extraction by using chunking techniques or regular expressions with POS tags.

3. Extend the system to handle multiple languages using spaCy's multilingual models.

Lab 10: IMPLEMENT TEXT CLASSIFICATION USING HMM MODEL.

The Hidden Markov Model (HMM) can be used for text classification by modelling the sequence of words or features in text data as observations and the class labels as hidden states. To implement text classification using a Hidden Markov Model (HMM), the process typically involves modeling the sequence of words (or other tokens) in text as a sequence of observations generated by hidden states. Each class corresponds to a distinct HMM.
Steps for Implementation:

1. Data Preprocessing:
   - Tokenize text into words or subwords.
   - Label each document with its corresponding class.
2. Feature Extraction:
   - Use word frequencies, bigrams, or other linguistic features for the observations.
   - Encode these features numerically.
3. Training HMM for Each Class:
   - Estimate initial probabilities, transition probabilities, and emission probabilities from labeled training data for each class.
4. Classification:
   - Given a new document, calculate the likelihood of the sequence under each class's HMM.
   - Assign the document to the class with the highest likelihood.
5. Evaluation:
   - Evaluate the model using metrics like accuracy, precision, recall, and F1-score on a test set.

Additional Exercise:

1. Cross-Domain Classification: Test the HMM's ability to classify texts from different domains.
   - Train HMMs on a dataset from one domain (e.g., technology).
   - Test them on a dataset from a different domain (e.g., healthcare).
   - Analyze the limitations and propose methods to improve cross-domain performance.

Use domain-specific datasets like 20 Newsgroups or custom corpora.

2. Hybrid HMM-Naïve Bayes : Combine HMM with Naïve Bayes for text classification.
   - Use HMMs to extract sequence-based features from text.
   - Feed these features into a Naïve Bayes classifier for final classification.
   - Compare the performance of the hybrid model with the standalone HMM.

Use posterior probabilities from the HMM as input features for Naïve Bayes.

Lab-11
Case Study 1:
Chatbot for Customer Support
A company wants to build a customer support chatbot to assist users with common queries related to their services. The chatbot should understand user intent, extract relevant entities, and provide appropriate responses. This case study focuses on developing and implementing an NLP-based conversational agent.

1. **Collect a dataset of customer queries.** Use a pre-existing dataset (e.g., Chatbot Corpus, Intent Recognition Dataset, or create synthetic data).
2. **Format the dataset to include**: User queries, Intent labels (e.g., "Order Status", "Product Inquiry", "Refund Request"), Entities (e.g., "Order ID", "Product Name")

Deliverable: A labeled dataset ready for training.

3. **Build a model to classify user intent.** Train models like: Logistic Regression, SVM Transformer-based models (e.g., BERT or DistilBERT) (Optional)
4. **Evaluate** accuracy, precision, recall, and F1-score for intent classification.
5. **Extract entities from user queries.** A trained Named Entity Recognition (NER) model capable of extracting entities like "Order ID", "Date", or "Product Name".

Lab 12:
Case Study 2:
News Article Classification and Summarization
A news agency wants to automatically classify and summarize articles for better organization and quick reader consumption. The system should classify articles into predefined categories (e.g., "Politics," "Sports," "Technology") and generate concise summaries.

1. **Collect and preprocess news article data:** Use publicly available datasets (e.g., BBC News Classification Dataset, AG News, or Reuters-21578).
2. **Preprocess the text:** Remove special characters, stopwords, numbers, and URLs. Tokenize and lemmatize the text.
3. **Analyze**: Distribution of articles across categories. Average word count and sentence length of articles. Visualize word frequency using bar charts or word clouds for each category.

Deliverable: Insights and visualizations highlighting dataset characteristics.

4. **Text Classification:** Classify articles into predefined categories. Experiment with multiple models. Use Traditional: Naïve Bayes, SVM and Deep Learning: LSTM, CNNs
5. **Evaluate:** Split the dataset into training and testing sets, and evaluate using precision, recall, and F1-score.
6. Text Summarization: Generate concise summaries for news articles. Implement extractive summarization and implement abstractive summarization. Compare the quality of summaries generated by extractive and abstract approaches.

Lab 13
Case Study 3:
Chatbot for Customer Support

A company wants to build a customer support chatbot to assist users with common queries related to their services. The chatbot should understand user intent, extract relevant entities, and provide appropriate responses. This case study focuses on developing and implementing an NLP-based conversational agent.

6. Collect a dataset of customer queries. Use a pre-existing dataset (e.g., Chatbot Corpus, Intent Recognition Dataset, or create synthetic data).
7. Format the dataset to include: User queries, Intent labels (e.g., "Order Status", "Product Inquiry", "Refund Request"), Entities (e.g., "Order ID", "Product Name")
   Deliverable: A labeled dataset ready for training.
8. Build a model to classify user intent. Train models like: Logistic Regression, SVM Transformer-based models (e.g., BERT or DistilBERT) (Optional)
9. Evaluate accuracy, precision, recall, and F1-score for intent classification.
10. Extract entities from user queries. A trained Named Entity Recognition (NER) model capable of extracting entities like "Order ID", "Date", or "Product Name".

Lab 14:

Case Study 4:

News Article Classification and Summarization

A news agency wants to automatically classify and summarize articles for better organization and quick reader consumption. The system should classify articles into predefined categories (e.g., "Politics," "Sports," "Technology") and generate concise summaries.

1. Collect and preprocess news article data: Use publicly available datasets (e.g., BBC News Classification Dataset, AG News, or Reuters-21578).
2. Preprocess the text: Remove special characters, stopwords, numbers, and URLs. Tokenize and lemmatize the text.
3. Analyze: Distribution of articles across categories. Average word count and sentence length of articles. Visualize word frequency using bar charts or word clouds for each category.
   Deliverable: Insights and visualizations highlighting dataset characteristics.
4. Text Classification: Classify articles into predefined categories. Experiment with multiple models. Use Traditional: Naïve Bayes, SVM and Deep Learning: LSTM, CNNs
5. Evaluate: Split the dataset into training and testing sets, and evaluate using precision, recall, and F1-score.
6. Text Summarization: Generate concise summaries for news articles. Implement extractive summarization and implement abstractive summarization. Compare the quality of summaries generated by extractive and abstract approaches.