

## **EXPERIMENT-9**

### **AIM:**

Write a program to implement Hangman game

### **THEORY:**

Implementing Hangman in Prolog requires defining predicates to select a word for guessing, displaying the initial game state, accepting player guesses, updating the game state, and determining win or loss conditions. The game continues in a loop until the player correctly guesses the word or runs out of attempts.

### **ALGORITHM:**

Select Word:

Define a predicate to select a word randomly from a predefined list.

Initialize Game State:

Define predicates to initialize the game state:

Hide the selected word with underscores to represent unrevealed letters. Set the number of allowed incorrect guesses.

Display Game State:

Define a predicate to display the current game state:

Display the hidden word with revealed letters.

Display the number of incorrect guesses remaining.

Display the letters guessed so far.

Accept Guess:

Define a predicate to accept a guess from the player:

Prompt the player to guess a letter.

Ensure the input is a valid single letter.

Update Game State:

Define a predicate to update the game state based on the player's guess:

Replace underscores with the guessed letter if correct.

Decrement the number of incorrect guesses remaining if incorrect.

Track the letters guessed so far.

Check Win or Loss:

Define predicates to check for win or loss conditions:

Check if all letters in the word have been guessed correctly.

Check if the number of incorrect guesses has reached the limit.

Play Game:

Define a predicate to play the game:

Initialize the game state.

Display the initial game state.

Accept guesses from the player until win or loss conditions are met.

Display win or loss message accordingly.

Interactivity:

Ensure predicates are interactive, allowing players to input guesses and see the updated game state.

## **SOURCE CODE:**

```
% Define the list of words for the game
```

```
word(apple).
```

```
word(banana).
```

```
word(orange).
```

```
word(grape).
```

```
word(strawberry).
```

```
word(watermelon).
```

```
% Predicate to select a random word from the list
```

```
random_word(Word) :-
```

```
findall(W, word(W), Words),
```

```

length(Words, Length),
random(0, Length, Index),
nth0(Index, Words, Word).
% Predicate to initialize the game state

initialize_game :-
    random_word(Word),
    string_chars(Word, Chars),
    length(Chars, Length),
    empty_string(Length, Underscores),
    display_game(Underscores, []).

% Predicate to create a string of underscores
empty_string(0, []).
empty_string(N, [_ | T]) :-
    N > 0,
    N1 is N - 1,
    empty_string(N1, T).

% Predicate to display the game state
display_game(Word, Guessed) :-
    writeln(Word),
    writeln("Guessed letters:"),
    writeln(Guessed).

% Predicate to accept a guess from the player
get_guess(Guess) :-
    write("Guess a letter: "),

```

```
read_line_to_string(user_input, Guess).
```

```
% Predicate to update the game state based on the guess
```

```
update_game(Word, Guess, NewWord, Guessed) :-
```

```
member(Guess, Word),
```

```
!,
```

```
update_word(Word, Guess, NewWord),
```

```
update_guessed(Guess, Guessed, NewGuessed),
```

```
display_game(NewWord, NewGuessed).
```

```
update_game(Word, Guess, Word, Guessed) :- \+
```

```
member(Guess, Word),
```

```
!,
```

```
update_guessed(Guess, Guessed, NewGuessed),
```

```
display_game(Word, NewGuessed).
```

```
% Predicate to update the word with the guessed letter
```

```
update_word([], _, []).
```

```
update_word([H|T], Guess, [Guess|T]) :-
```

```
H == Guess,
```

```
!.
```

```
update_word([H|T], Guess, [H|NewT]) :-
```

```
update_word(T, Guess, NewT).
```

```
% Predicate to update the list of guessed letters
```

```
update_guessed(Guess, Guessed, [Guess|Guessed]) :- \+
```

```
member(Guess, Guessed).
```

```
update_guessed(_, Guessed, Guessed).
```

```
% Predicate to play the game
```

```
play :-
```

```
writeln("Welcome to Hangman!"),
```

```
initialize_game,
```

```
play_game([]).
```

```
% Predicate to play the game loop
```

```
play_game(Guessed) :-
```

```
display_game(Word, Guessed),
```

```
get_guess(Guess),
```

```
update_game(Word, Guess, NewWord, NewGuessed),
```

```
check_win(NewWord, NewGuessed),
```

```
!,
```

```
play_game(NewGuessed).
```

```
% Predicate to check for win condition
```

```
check_win(Word, Guessed) :-
```

```
string_chars(Word, Chars),
```


```
subset(Chars, Guessed),
```

```
writeln("Congratulations! You win!").
```

```
% Start the game
```

```
:- initialization(play).
```

## OUTPUT:

 ?- Your query goes here ...  

Welcome to Hangman!  
[ , \_ , \_ , \_ , \_ , \_ , \_ , \_ ]  
Guessed letters:  
[]  
\_26858  
Guessed letters:  
[]  
Guess a letter:

line> Send Abort

## **EXPERIMENT-10**

### **AIM:**

Write a program to implement Tic-Tac-Toe game

### **THEORY:**

Implementing Tic-Tac-Toe in Prolog involves defining predicates to display the game board, check for winning conditions, handle player moves, and determine game outcomes. By representing the game state as a list, and recursively alternating player turns until a win, draw, or loss condition is met, a complete game logic is achieved.

### **ALGORITHM:**

Display Board:

Define a predicate to display the Tic-Tac-Toe board.

Check for Winning Conditions:

Define predicates to check for winning conditions:

Check if a player has three in a row horizontally, vertically, or diagonally.

Check for Draw:

Define a predicate to check if the board is full without a winner, indicating a draw. Get

Player Move:

Define a predicate to get the next player's move:

Prompt the player for their move.

Ensure the move is valid and the cell is empty.

Update the board with the player's move.

Play Game:

Define a predicate to play the game:

Display the board.

Get the current player's move.

Check for win or draw conditions.

Repeat until the game is over.

Start Game:

Define a predicate to start the game:

Initialize the board.

Start playing with the first player.

End Game:

Display the winner or a draw message when the game ends.

Interactivity: Ensure predicates are interactive, allowing players to input moves and see the updated board.

## SOURCE CODE:

% A Tic-Tac-Toe program in Prolog. S. Tanimoto, May 11, 2003.

% To play a game with the computer, type

% playo.

% To watch the computer play a game with itself, type

% selfgame.

% Predicates that define the winning conditions:

win(Board, Player) :- rowwin(Board, Player).

win(Board, Player) :- colwin(Board, Player).

win(Board, Player) :- diagwin(Board, Player).

rowwin(Board, Player) :- Board = [Player,Player,Player,\_,\_,\_,\_,\_].

rowwin(Board, Player) :- Board = [\_,\_,Player,Player,Player,\_,\_,\_].

rowwin(Board, Player) :- Board = [\_,\_,\_,\_,Player,Player,Player].

colwin(Board, Player) :- Board = [Player,\_,\_,Player,\_,\_,Player,\_,\_].

colwin(Board, Player) :- Board = [\_,Player,\_,\_,Player,\_,\_,Player,\_,\_].

colwin(Board, Player) :- Board = [\_,\_,Player,\_,\_,Player,\_,\_,Player].



```
diagwin(Board, Player) :- Board = [Player,_,_,Player,_,_,Player].
```

```
diagwin(Board, Player) :- Board = [_,_Player,_,Player,_,Player,_,_].
```

```
% Helping predicate for alternating play in a "self" game:
```

```
other(x,o).
```

```
other(o,x).
```

```
game(Board, Player) :- win(Board, Player), !, write([player, Player, wins]).
```

```
game(Board, Player) :-
```

```
    other(Player,Otherplayer),
```

```
    move(Board,Player,Newboard),
```

```
    !,
```

```
    display(Newboard),
```

```
    game(Newboard,Otherplayer).
```

```
move([b,B,C,D,E,F,G,H,I], Player, [Player,B,C,D,E,F,G,H,I]).
```

```
move([A,b,C,D,E,F,G,H,I], Player, [A,Player,C,D,E,F,G,H,I]).
```

```
move([A,B,b,D,E,F,G,H,I], Player, [A,B,Player,D,E,F,G,H,I]).
```

```
move([A,B,C,b,E,F,G,H,I], Player, [A,B,C,Player,E,F,G,H,I]).
```

```
move([A,B,C,D,b,F,G,H,I], Player, [A,B,C,D,Player,F,G,H,I]).
```

```
move([A,B,C,D,E,b,G,H,I], Player, [A,B,C,D,E,Player,G,H,I]).
```

```
move([A,B,C,D,E,F,b,H,I], Player, [A,B,C,D,E,F,Player,H,I]).
```

```
move([A,B,C,D,E,F,G,b,I], Player, [A,B,C,D,E,F,G,Player,I]).
```

```
move([A,B,C,D,E,F,G,H,b], Player, [A,B,C,D,E,F,G,H,Player]).
```

```
display([A,B,C,D,E,F,G,H,I]) :- write([A,B,C]),nl,write([D,E,F]),nl,
```

```
write([G,H,I]),nl,nl.
```

```
selfgame :- game([b,b,b,b,b,b,b,b],x).  
% Predicates to support playing a game with the user:
```

```
x_can_win_in_one(Board) :- move(Board, x, Newboard), win(Newboard, x).
```

```
% The predicate orespond generates the computer's (playing o) reponse %  
from the current Board.
```

```
orespond(Board,Newboard) :-
```

```
    move(Board, o, Newboard),
```

```
    win(Newboard, o),
```

```
    !.
```

```
orespond(Board,Newboard) :-
```

```
    move(Board, o, Newboard),
```

```
    not(x_can_win_in_one(Newboard)).
```

```
orespond(Board,Newboard) :-
```

```
    move(Board, o, Newboard).
```

```
orespond(Board,Newboard) :-
```

```
    not(member(b,Board)),
```

```
    !,
```

```
    write('Cats game!'), nl,
```

```
    Newboard = Board.
```

```
% The following translates from an integer description
```

```
% of x's move to a board transformation.
```

```
xmove([b,B,C,D,E,F,G,H,I], 1, [x,B,C,D,E,F,G,H,I]).
```

```

xmove([A,b,C,D,E,F,G,H,I], 2, [A,x,C,D,E,F,G,H,I]).
xmove([A,B,b,D,E,F,G,H,I], 3, [A,B,x,D,E,F,G,H,I]).
xmove([A,B,C,b,E,F,G,H,I], 4, [A,B,C,x,E,F,G,H,I]).
xmove([A,B,C,D,b,F,G,H,I], 5, [A,B,C,D,x,F,G,H,I]).
xmove([A,B,C,D,E,b,G,H,I], 6, [A,B,C,D,E,x,G,H,I]).
xmove([A,B,C,D,E,F,b,H,I], 7, [A,B,C,D,E,F,x,H,I]).
xmove([A,B,C,D,E,F,G,b,I], 8, [A,B,C,D,E,F,G,x,I]).
xmove([A,B,C,D,E,F,G,H,b], 9, [A,B,C,D,E,F,G,H,x]).
xmove(Board, _, Board) :- write('Illegal move.'), nl.

```

% The 0-place predicate playo starts a game with the user.

```

playo :- explain, playfrom([b,b,b,b,b,b,b,b]).

```

```

explain :-

```

```

    write('You play X by entering integer positions followed by a period.'), nl,
    display([1,2,3,4,5,6,7,8,9]).

```

```

playfrom(Board) :- win(Board, x), write('You win!').

```

```

playfrom(Board) :- win(Board, o), write('I win!').

```

```

playfrom(Board) :- read(N),

```

```

    xmove(Board, N, Newboard),

```

```

    display(Newboard),

```

```

    orespond(Newboard, Newnewboard),

```

```

    display(Newnewboard),

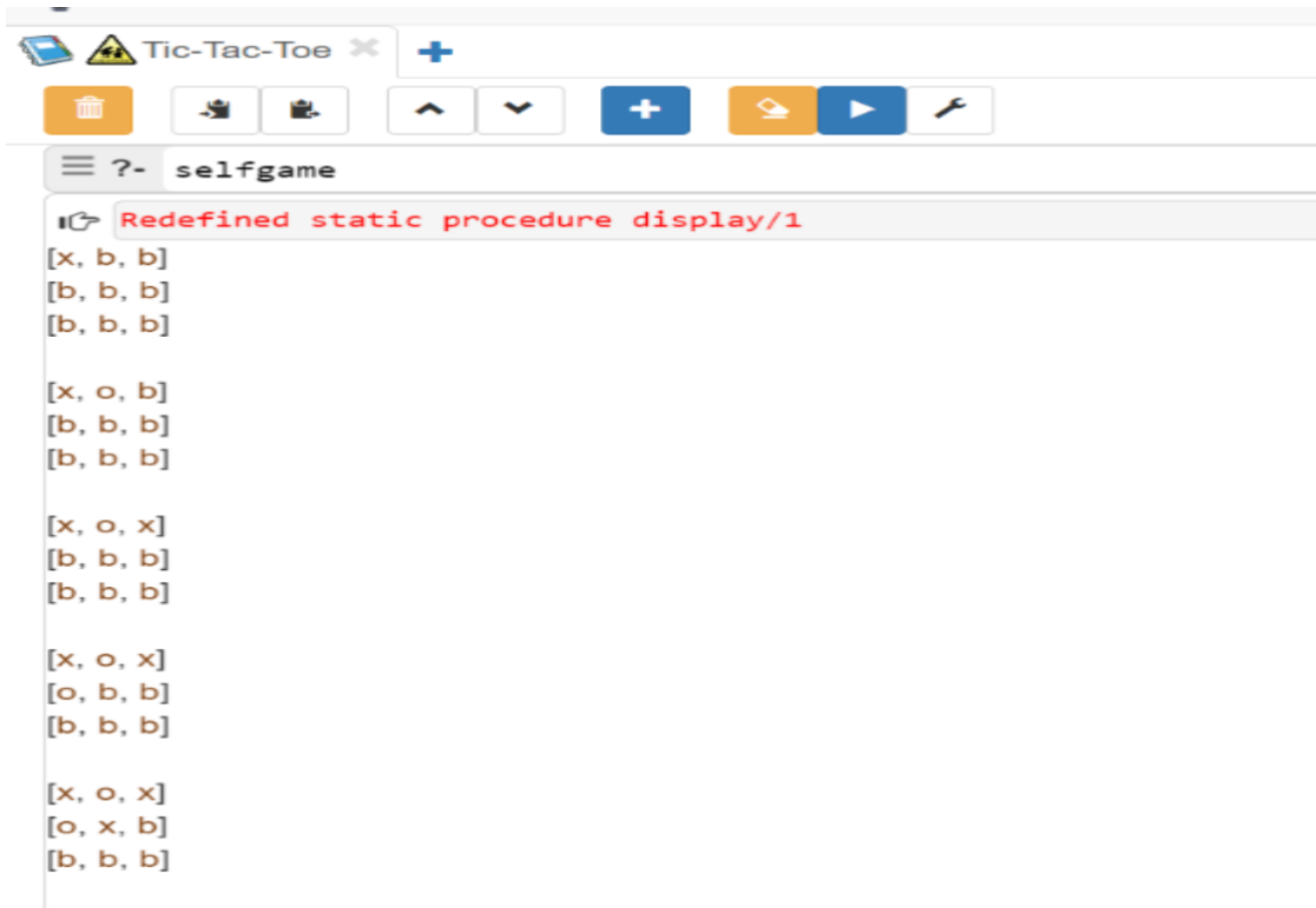
```

```

    playfrom(Newnewboard).

```

## OUTPUT:



The screenshot shows a Prolog IDE window titled "Tic-Tac-Toe". The interface includes a toolbar with icons for file operations, navigation, and execution. The main area displays the output of a query, showing a sequence of board states for a Tic-Tac-Toe game. The output is formatted as a list of lists, where each list represents a board state. The first three states are all empty boards. The subsequent states show the progression of the game, with 'x' and 'o' being placed in various positions. The final state shows a board where 'x' has won by having three 'x's in a row.

```
selfgame  
Redefined static procedure display/1  
[x, b, b]  
[b, b, b]  
[b, b, b]  
  
[x, o, b]  
[b, b, b]  
[b, b, b]  
  
[x, o, x]  
[b, b, b]  
[b, b, b]  
  
[x, o, x]  
[o, b, b]  
[b, b, b]  
  
[x, o, x]  
[o, x, b]  
[b, b, b]
```



Tic-Tac-Toe



[x, o, x]

[o, x, o]

[b, b, b]

[x, o, x]

[o, x, o]

[x, b, b]

[x, o, x]

[o, x, o]

[x, o, b]

[player, x, wins]

true

## **EXPERIMENT-11**

### **AIM:**

Write a program that removes stopwords from a passage in a text file using Python and NLTK

### **THEORY:**

- Stopwords are common words like "the," "a," "an," etc., which might not carry much meaning for analysis. Removing them can help focus on the more content-rich words.
- The decision of whether or not to remove stopwords depends on your specific NLP task. In some cases, stopwords might be relevant (e.g., sentiment analysis).
- NLTK provides stopwords for various languages. You can specify the language argument (e.g., `stopwords.words('spanish')`) to use a different list.

### **ALGORITHM:**

#### **1.Import Necessary Libraries:**

We'll use libraries from Natural Language Toolkit (NLTK) for text processing.

#### **2.Download Stopwords (if not already done):**

NLTK stopwords aren't downloaded by default. You might need to download the list for your desired language using a separate command in your terminal.

**3.Read the Text File:**Open the text file containing the passage you want to analyze.Read the entire content of the file into a variable.

#### **4.Preprocessing (Optional):**

This step might be necessary depending on your analysis goals.Consider converting all characters to lowercase for case-insensitive stopword removal.You might want to remove punctuation marks like commas, periods, etc. that might not be relevant.Split the text into individual words or "tokens" for easier processing.

**5.Stopword Removal:**Retrieve the list of stopwords for your chosen language (e.g., English) from NLTK.Iterate through the tokens (individual words).If a token is not present in the stopword list, keep it.This effectively filters out stopwords from the text.

#### **6.Post-processing (Optional):**

If needed, join the filtered words back into a string for further analysis.

## SOURCE CODE-

```
[1]: import nltk
import ssl
import certifi

[2]: ssl._create_default_https_context = ssl._create_unverified_context
Jupyter

[3]: nltk.download('stopwords')
nltk.download('punkt')

[nltk_data] Downloading package stopwords to
[nltk_data] /Users/anushkaagrani/nltk_data...
[nltk_data] Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package punkt to
[nltk_data] /Users/anushkaagrani/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.

[3]: True

[4]: from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

[5]: with open('/Users/anushkaagrani/Desktop/an.txt', 'r') as file:
text = file.read()

[6]: tokens = word_tokenize(text.lower())

[7]: english_stopwords = set(stopwords.words('english'))

[8]: filtered_tokens = [t for t in tokens if t not in english_stopwords]

[9]: cleaned_passage = ' '.join(filtered_tokens)
```

## OUTPUT-

```
[10]: print("Cleaned Passage:")
print(cleaned_passage)

Cleaned Passage:
cat sat mat . cat furry animal . cat likes play yarn . cat good pet .
```

## INPUT FILE-



```
an.txt
The cat sat on the mat. The cat is a furry animal. The cat likes to play with yarn. The cat is a good pet.
```

## **EXPERIMENT-12**

### **AIM:**

Write a program to implement stemming for a given sentence using NLTK.

### **THEORY:**

Stemming is the process of reducing a word to its base or root form. It aims to group words with similar meanings together, regardless of their inflections (e.g., "running," "runs," "ran" would all be stemmed to "run")

**Here are some important points to consider regarding stemming:**

- Stemming is a simpler and faster approach compared to lemmatization (which aims for dictionary forms).
- Stemming might not always produce accurate base forms (e.g., "stem" could be stemmed to "stem" or "step").
- Choose the appropriate stemmer for your language and task.

### **ALGORITHM:**

#### **1.Import Necessary Libraries:**

We'll use libraries from Natural Language Toolkit (NLTK) for text processing.

#### **2.Choose and Create a Stemmer Object:**

NLTK provides various stemmer algorithms (e.g., PorterStemmer). Select the appropriate one for your language and task.

#### **3.Create an instance of the chosen stemmer class.**

#### **4.Prepare the Sentence:**

Obtain the sentence you want to process.

#### **5.Tokenization (Optional):**

Depending on your needs, you might want to split the sentence into individual words for easier processing.

#### **6.Stemming:**

Iterate through each word in the sentence (or tokenized list).Apply the stemmer object to each word, effectively reducing it to its base form.



### 7.Post-processing (Optional):

If you tokenized the sentence in step 4, join the stemmed words back into a sentence for further analysis.

### SOURCE CODE-

```
[11]: from nltk.stem import PorterStemmer

[12]: stemmer = PorterStemmer()

[13]: sentence = "Programmers program with programming languages"

[14]: words = word_tokenize(sentence)

[15]: stemmed_sentence = ' '.join(stemmer.stem(w) for w in words)
```

### OUTPUT-

```
[16]: print("Stemmed Sentence:")
      print(stemmed_sentence)
```

```
Stemmed Sentence:
programm program with program languag
```

## EXPERIMENT-13

### AIM:

Write a program to POS(part of speech) tagging for the given sentence using NLTK.

### THEORY:

POS tagging involves assigning a grammatical label (tag) to each word in a sentence. These tags indicate the word's function within the sentence structure (e.g., noun, verb, adjective, adverb, etc.)

- POS tagging helps understand the grammatical structure and relationships between words in a sentence.
- It has applications in tasks like sentiment analysis, machine translation, and text summarization.
- The accuracy of POS tagging can vary depending on the chosen tagger and the complexity of the text.

### ALGORITHM:

1. Import libraries (nltk, pos\_tag).
2. Download resources (if needed).
3. Prepare the sentence.
4. Tokenize the sentence (optional).
5. Apply pos\_tag to the sentence (or tokens).
6. Analyze the resulting list of tuples (word, POS tag).

### SOURCE CODE:

```
[17]: nltk.download('averaged_perceptron_tagger')  
  
[nltk_data] Downloading package averaged_perceptron_tagger to  
[nltk_data] /Users/anushkaagrani/nltk_data...  
[nltk_data] Unzipping taggers/averaged_perceptron_tagger.zip.  
  
[17]: True  
  
[18]: sentence = "I am learning NLP in Python"  
  
[19]: tokens = nltk.word_tokenize(sentence)  
  
[20]: pos_tags = nltk.pos_tag(tokens)
```

### OUTPUT:

```
[21]: print("POS Tagged Tokens:")  
      print(pos_tags)
```

POS Tagged Tokens:

```
[('I', 'PRP'), ('am', 'VBP'), ('learning', 'VBG'), ('NLP', 'NNP'), ('in', 'IN'), ('Python', 'NNP')]
```

## EXPERIMENT-14

### AIM:

Write a program to implement Lemmatization using NLTK.

### THEORY:

Lemmatization aims to reduce words to their dictionary forms (lemmas), considering their context. Unlike stemming, which might produce non-existent words, lemmatization focuses on identifying the base or canonical form of a word.

**Here are some important factors considered important while lemmatization:**

- Lemmatization provides a more accurate way to reduce words to their base forms compared to stemming.
- It's beneficial for tasks like information retrieval, text analysis, and sentiment analysis where understanding the core meaning of words is crucial.
- The accuracy of lemmatization can depend on the quality of the underlying lexical database (WordNet in this case).

### ALGORITHM:

1. Import libraries (nltk, WordNetLemmatizer).
2. Download resources (if needed).
3. Prepare the sentence.
4. Tokenize the sentence (optional).
5. Perform POS tagging (optional, but recommended).
6. Create a WordNetLemmatizer object.
7. Iterate through words (or tokens) and their POS tags (if available).
8. Use `lemmatizer.lemmatize(word, pos_tag)` to get the lemma.
9. Join the lemmas back into a sentence (optional).

### SOURCE CODE:

```
[22]: from nltk.stem import WordNetLemmatizer

[23]: lemmatizer = WordNetLemmatizer()

[25]: nltk.download('wordnet')

[nltk_data] Downloading package wordnet to
[nltk_data] /Users/anushkaagrani/nltk_data...

[25]: True
```

## OUTPUT:

```
[26]: print("rocks:", lemmatizer.lemmatize("rocks"))  
      print("corpora:", lemmatizer.lemmatize("corpora"))  
      print("better (adjective):", lemmatizer.lemmatize("better", pos="a"))  
  
      rocks: rock  
      corpora: corpus  
      better (adjective): good
```

## EXPERIMENT-15

### AIM:

Write a program for Text Classification for the given sentence using NLTK.

### THEORY:

Text classification is the task of assigning a category label to a piece of text. NLTK provides tools for building and evaluating text classifiers.

- Naive Bayes is a simple yet effective supervised learning algorithm for text classification.
- Feature engineering plays a crucial role in achieving good classification accuracy.
- NLTK provides other classification algorithms (e.g., Support Vector Machines) that you might explore for more complex tasks.

### ALGORITHM:

1. Import libraries (nltk, NaiveBayesClassifier, FreqDist, classify).
2. Prepare labeled text data (sentences with categories).
3. Preprocess text (tokenize, optional stopwords removal/lemmatization).
4. Extract features (word presence, frequency, TF-IDF, etc.).
5. Train the Naive Bayes classifier on the labeled dataset.
6. Classify a new sentence using the trained classifier.

### SOURCE CODE:

```
[33]: import nltk
      from nltk.classify import NaiveBayesClassifier
      from nltk.classify.util import accuracy as nltk_accuracy
      from nltk.corpus import movie_reviews
      from nltk.tokenize import word_tokenize
      import random
```

```
[34]: nltk.download('movie_reviews')
      nltk.download('punkt')
```

```
[nltk_data] Downloading package movie_reviews to
[nltk_data] /Users/anushkaagrani/nltk_data...
[nltk_data] Unzipping corpora/movie_reviews.zip.
[nltk_data] Downloading package punkt to
[nltk_data] /Users/anushkaagrani/nltk_data...
[nltk_data] Package punkt is already up-to-date!
```

```
[34]: True
```

```
[35]: def extract_features(words):
        return {word.lower(): True for word in words}

        # Load movie reviews from NLTK
        documents = [(list(movie_reviews.words(fileid)), category)
                      for category in movie_reviews.categories()
                      for fileid in movie_reviews.fileids(category)]
        random.shuffle(documents)

        # Feature extraction
        feature_sets = [(extract_features(doc), category) for (doc, category) in documents]

[36]: train_set, test_set = feature_sets[:int(len(feature_sets) * 0.8)], feature_sets[int(len(feature_sets) * 0.8):]

[37]: classifier = NaiveBayesClassifier.train(train_set)
```

```
[38]: def classify_sentence(sentence):
        words = word_tokenize(sentence)
        features = extract_features(words)
        return classifier.classify(features)
```

## OUTPUT:

```
input_sentence = "This movie is amazing, with an excellent plot and great characters!"
print("Classification:", classify_sentence(input_sentence))
```

Classification: pos

```
[39]: print("Accuracy:", nltk_accuracy(classifier, test_set))
```

Accuracy: 0.68