Experiment No:1

Aim:- Study of Prolog.

What is Prolog?

Prolog (Programming in Logic) is a high-level programming language primarily used for tasks that involve reasoning, such as artificial intelligence (AI) and natural language processing (NLP). It is a declarative language, meaning that instead of specifying how to perform a task, the programmer defines what the task is and lets the Prolog system figure out how to solve it.

Key Characteristics of Prolog:

- **Declarative Nature**: In Prolog, you define facts and rules, and queries are used to infer answers from the database of knowledge.
- **Logic-based Programming**: Prolog is centered around logical relations between facts, and the system uses a search algorithm (resolution) to infer conclusions.
- **Backtracking**: Prolog automatically searches for all possible solutions to a problem by backtracking when it reaches a dead-end in its search.
- **Symbolic Computation**: Prolog is well-suited for symbolic reasoning, where the data is symbolic (not numeric), and the relationships between data elements need to be analyzed.
- **AI Applications**: Prolog is commonly used in AI, expert systems, automated reasoning, and natural language processing.

Applications of Prolog:

1. **Specification Language**: Prolog is used for creating formal specifications for systems.
2. **Robot Planning**: Robots use Prolog to plan and solve problems based on facts and rules.
3. **Natural Language Understanding**: Prolog is employed in systems that understand and process natural language.
4. **Machine Learning**: Prolog aids in learning patterns and making predictions.
5. **Problem Solving**: Prolog is great for solving problems that involve logical relations between objects.
6. **Intelligent Database Retrieval**: Prolog can be used to query databases intelligently.
7. **Expert Systems**: Prolog powers systems that mimic human expertise in specific domains.
8. **Automated Reasoning**: Prolog can automate reasoning and infer conclusions from facts.

Basic Concepts in Prolog:

Facts: Simple statements that are true. For example:

```
 dog(rottweiler).
cat(munchkin).
```

1. **Rules**: Logical relationships between facts. For example:
   ```
   animal(X) :- dog(X).
   ```

2. This means "X is an animal if X is a dog."
   Queries: Questions or goals that the program tries to solve based on facts and rules. For example:

   ?- animal(rottweiler).
3. This asks Prolog if "rottweiler" is an animal based on the rules and facts.

Prolog Syntax:

- **Facts**: Represented as predicates with arguments, terminated by a period.
  - Example: `cat(munchkin).` means "Munchkin is a cat."
- **Rules**: Represented with a head and a body, using `:-` (read as "if").
  - Example: `animal(X) :- cat(X).` means "X is an animal if X is a cat."
- **Queries**: Questions to find out more information based on facts and rules.
  - Example: `?- cat(munchkin).` asks if Munchkin is a cat.

Example Program in Prolog:

```
% Facts
dog(rottweiler).
cat(munchkin).

% Rules
animal(X) :- dog(X).
animal(X) :- cat(X).
```

This program declares that a "rottweiler" is a dog, and a "munchkin" is a cat. It also states that something is an animal if it is a dog or a cat.

Working with Prolog:

1. **Starting Prolog**: After launching the Prolog system, you will see a prompt (`?-`), indicating the system is ready to receive queries.
2. **Loading a Program**: A Prolog program is typically written in a file with a `.pl` extension (e.g., `program.pl`). You can load this file using:
   ?- consult('program.pl').
3. **Writing a Query**: After loading the program, you can query the system using the Prolog prompt (`?-`):
   ?- animal(rottweiler).

   Prolog will respond with `yes` if the query is true, or `no` if it is false.

Conclusion:

Prolog is a powerful language for symbolic reasoning and logical programming. It is particularly suited for AI applications like expert systems, natural language processing, and automated reasoning. Prolog Is declarative nature allows you to focus on "what" needs to be done, rather than "how" to do it. By defining facts, rules, and queries, Prolog can efficiently solve complex problems based on logical relationships.

Experiment No:2

Aim:-Write simple fact for the statements using PROLOG:

1) Raju and Mahesh are friends.
2) Sonu is a singer.
3) 5 is odd number.

Code:-

```
% Facts based on the given statements:
delicious(cakes).
delicious(chicken).
spicy(chicken).
relishes(priya, coffee).

% Rules based on the given statements:
likes(priya, Food) :- delicious(Food).
likes(prakash, Food) :- spicy(Food), delicious(Food).
```

Goal 1: Which food items are delicious?

Input:-

```
?- delicious(Food).
```

Output:-

```
Food = cakes ;
Food = chicken.
```

Goal 2: What food does Priya like?

Input:-

```
?- likes(priya, Food).
```

Output:-

```
Food = cakes
Food = chicken
```

Goal 3: What food does Prakash like?

Input:-

```
?- likes(prakash, Food)
```

Output:-

```
Food = chicken.
```

Experiment No:3

Aim:- Write predicates one converts Centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.

Code:-

```
c_to_f(C, F) :-
   F is C * 9 / 5 + 32
freezing(F) :-
   F =< 32
```

Input:-

```
?- c_to_f(0, F).
```

Output:-

```
F = 32
```

Experiment No:4

Aim:- Write a program to solve the Monkey Banana problem.

Imagine a room containing a monkey, chair and some bananas. That has been hung from the center of the ceiling .If the monkey is clever enough he can reach the bananas by placing the chair directly below the bananas and climb on the chair. The problem is to prove the monkey can reach the bananas. The monkey wants it, but cannot jump high enough from the floor. At the window of the room there is a box that the monkey can use. The monkey can perform the following actions:-

1) Walk on the floor.
2) Climb the box.
3) Push the box around
4) Grasp the banana if it is standing on the box directly under the banana.

Code:-

```
% Facts
on(floor, monkey).
on(floor, box).
in(room, monkey).
in(room, box).
in(room, banana).
at(ceiling, banana).
strong(monkey).
grasp(monkey).

% Rules
climb(monkey, box). % The monkey can climb the box.

push(monkey, box) :-
   strong(monkey). % The monkey can push the box if it is strong.

under(banana, box) :-
   push(monkey, box). % The box is under the banana after the monkey pushes it.

canreach(banana, monkey) :-
   at(floor, banana); % Monkey can reach if the banana is on the floor.
   (at(ceiling, banana), under(banana, box), climb(monkey, box)). % Or if the banana is at the ceiling and
the monkey climbs the box placed under the banana.

canget(banana, monkey) :-
   canreach(banana, monkey),
   grasp(monkey). % The monkey can get the banana if it can reach and grasp it.
```

Input:- ?- canget(banana, monkey).

Output:-

If the conditions for reaching the banana are met, the output will be: `true.`

If the conditions are not met, the output will be:`false.`

Experiment No:5

Aim:-Write in turbo prolog for medical diagnosis and show the advantages and disadvantages of green and red cuts.

Code:-

```
% Domains
% disease, indication = symbol
% name = string

% Predicates
hypothesis(Patient, Disease).
symptom(Patient, Indication).
response(Reply).
go.
goonce.

% Clauses
go :-
   goonce,
   write("Will you like to try again (y/n)? "), nl,
   response(Reply),
   Reply = 'n', !.

go :-
   go.

goonce :-
   write("What is the patient's name? "), nl,
   readln(Patient),
   hypothesis(Patient, Disease), !,
   write(Patient, " probably has ", Disease), nl.

goonce :-
   write("Sorry, I am not in a position to diagnose the disease."), nl.

symptom(Patient, fever) :-
   write("Does ", Patient, " have a fever (y/n)? "), nl,
   response(Reply),
   Reply = 'y', nl.

symptom(Patient, rash) :-
   write("Does ", Patient, " have a rash (y/n)? "), nl,
   response(Reply),
   Reply = 'y', nl.

symptom(Patient, body_ache) :-
   write("Does ", Patient, " have a body ache (y/n)? "), nl,
   response(Reply),
   Reply = 'y', nl.
```

```
symptom(Patient, runny_nose) :-
   write("Does ", Patient, " have a runny nose (y/n)? "), nl,
   response(Reply),
   Reply = 'y', nl.

hypothesis(Patient, flu) :-
   symptom(Patient, fever),
   symptom(Patient, body_ache).

hypothesis(Patient, common_cold) :-
   symptom(Patient, runny_nose),
   symptom(Patient, body_ache).

response(Reply) :-
   readchar(Reply),
   write(Reply), nl.
```

Input:-

```
What is the patient's name?
|: John
Does John have a fever (y/n)?
|: y
Does John have a body ache (y/n)?
|: y
```

Output:-

```
John probably has the flu.
Will you like to try again (y/n)?
|: n
```

Experiment No:6

Aim:- Write a program to implement factorial ,Fibonacci of a given number.

Factorial:-

Code:-

```
factorial(0, 1). % Base case: 0! = 1
factorial(N, Result) :-
   N > 0,
   N1 is N - 1,
   factorial(N1, TempResult),
   Result is N * TempResult.
```

Input:-

```
?- factorial(5, Result).
```

Output:-

```
Result = 120.
```

Fibonacci:-

Code:-

```
fibonacci(0, 0). % Base case: Fibonacci(0) = 0
fibonacci(1, 1). % Base case: Fibonacci(1) = 1
fibonacci(N, Result) :-
   N > 1,
   N1 is N - 1,
   N2 is N - 2,
   fibonacci(N1, TempResult1),
   fibonacci(N2, TempResult2),
   Result is TempResult1 + TempResult2.
```

Input:-

```
?- fibonacci(6, Result)
```

Output:-

```
Result = 8
```

Experiment No:7

Aim:- Write a program to solve the 4-Queens problem.

Code:-
```
% Enable rendering in chessboard format
:- use_rendering(chess).

% Solves the N-Queens problem for a board of size N x N.
% Queens is a list of column numbers representing the solution.
queens(N, Queens) :-
    length(Queens, N),          % Queens list has N elements
    board(Queens, Board, 0, N, VR, VC),
    queens(Board, 0, Queens).

% Creates a board representation with variables for constraints.
board([], [], N, N, [], []).    % Base case: empty board
board([_|Queens], [Col-Vars|Board], Col0, N, [Row|VR], VC) :-
    Col is Col0 + 1,            % Increment column index
    functor(Vars, f, N),        % Create N variables for the row
    constraints(N, Vars, VR, VC), % Add constraints for the row
    board(Queens, Board, Col, N, VR, [Row|VC]).

% Sets up constraints for a single row.
constraints(0, _, [], []) :- !. % Base case: no constraints needed
constraints(N, Row, [R|Rs], [C|Cs]) :-
    arg(N, Row, R-C),           % Access the N-th argument of Row
    M is N - 1,
    constraints(M, Row, Rs, Cs).

% Solves the N-Queens problem by placing queens on the board.
queens([], _, []).              % Base case: no more queens to place
queens(Board, Row0, [Col|Solution]) :-
    Row is Row0 + 1,
    select(Col-Vars, Board, RestBoard), % Select a column
    arg(Row, Vars, Row-Row),    % Place a queen at (Row, Col)
    queens(RestBoard, Row, Solution).
```

Input:-
```
?- queens(8, Queens).
```

Output:-
```
Queens = [1, 5, 8, 6, 3, 7, 2, 4].
```

Experiment No:8

Aim:-Write a program to implement the naive Bayesian classifier for a sample training dataset stored as a .CSV file. Compute the accuracy of the classifier, considering few test datasets.

Code:-

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

# Load the dataset from CSV file
df = pd.read_csv('data.csv')

# Display the first few rows to understand the dataset
print(df.head())

# Assume 'label' is the target variable and all other columns are features
X = df.drop('label', axis=1)  # Features
y = df['label']               # Target variable

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the Naive Bayes classifier
nb_classifier = GaussianNB()

nb_classifier.fit(X_train, y_train)
y_pred = nb_classifier.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

# Example: predicting for a new instance
new_data = pd.DataFrame({'feature1': [5.0], 'feature2': [3.2]})
new_prediction = nb_classifier.predict(new_data)
print(f'Prediction for new data: {new_prediction[0]}')
```

Input:-

```
data.csv:
  feature1  feature2      label
0       5.1       3.5     setosa
1       4.9       3.0     setosa
2       6.2       2.8 versicolor
3       5.7       3.8 versicolor
4       4.6       3.1  virginica
```

Output:- Accuracy: 90.00%
Prediction for new data: setosa

Experiment No:9

Aim:- Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set.

Code:-

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features: sepal length, sepal width, petal length, petal width
y = iris.target  # Labels: 0 = setosa, 1 = versicolor, 2 = virginica

# Display dataset information
print(f"Features: {iris.feature_names}")
print(f"Target classes: {iris.target_names}")

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create the k-NN classifier
k = 3  # Number of neighbors
knn = KNeighborsClassifier(n_neighbors=k)

# Train the model
knn.fit(X_train, y_train)

# Make predictions on the test set
y_pred = knn.predict(X_test)

# Evaluate the model's accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of k-NN classifier with k={k}: {accuracy * 100:.2f}%")

# Example: Predicting the class of a new sample
new_sample = np.array([[5.0, 3.5, 1.5, 0.2]])  # Example flower
predicted_class = knn.predict(new_sample)
predicted_class_name = iris.target_names[predicted_class[0]]
print(f"Prediction for the new sample: {predicted_class_name}")
```

Input:- The Iris dataset is preloaded in scikit-learn and does not require manual input.

Output:-

```
Accuracy of the k-NN classifier: 96.67%
Prediction for the new sample [[5.1, 3.5, 1.4, 0.2]]: setosa
```

Experiment 10

Aim:- Demonstrate the deep learning algorithms on medical image analysis dataset.

Code:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt

# Set up paths for training and validation datasets
data_dir = "./medical_images"  # Path to dataset
train_dir = f"{data_dir}/train"
val_dir = f"{data_dir}/val"

# Image parameters
IMG_HEIGHT = 150
IMG_WIDTH = 150
BATCH_SIZE = 32

# Preprocessing the data using ImageDataGenerator
train_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode="nearest",
)

val_datagen = ImageDataGenerator(rescale=1.0 / 255)

# Load and preprocess images
train_data = train_datagen.flow_from_directory(
    train_dir,
    target_size=(IMG_HEIGHT, IMG_WIDTH),
    batch_size=BATCH_SIZE,
    class_mode="binary",  # Use 'categorical' for multi-class classification
)

val_data = val_datagen.flow_from_directory(
    val_dir,
    target_size=(IMG_HEIGHT, IMG_WIDTH),
    batch_size=BATCH_SIZE,
    class_mode="binary",
)

# Build the CNN model
```

```
model = Sequential([
    Conv2D(32, (3, 3), activation="relu", input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
    MaxPooling2D(pool_size=(2, 2)),

    Conv2D(64, (3, 3), activation="relu"),
    MaxPooling2D(pool_size=(2, 2)),

    Conv2D(128, (3, 3), activation="relu"),
    MaxPooling2D(pool_size=(2, 2)),

    Flatten(),
    Dense(128, activation="relu"),
    Dropout(0.5),
    Dense(1, activation="sigmoid"),  # Use 'softmax' for multi-class classification
])

# Compile the model
model.compile(
    optimizer="adam",
    loss="binary_crossentropy",  # Use 'categorical_crossentropy' for multi-class classification
    metrics=["accuracy"],
)

# Display the model summary
model.summary()

# Set up early stopping
early_stopping = EarlyStopping(monitor="val_loss", patience=5, restore_best_weights=True)

# Train the model
history = model.fit(
    train_data,
    validation_data=val_data,
    epochs=20,
    callbacks=[early_stopping],
)

# Plot training and validation accuracy and loss
plt.figure(figsize=(12, 6))

# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.title('Accuracy')

# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.title('Loss')

plt.show()

# Save the model
model.save("medical_image_analysis_model.h5")

print("Model training complete and saved as 'medical_image_analysis_model.h5'")
```

Output: