

Applied Artificial Intelligence

Certified Journal

**Submitted in partial fulfilment of the
Requirements for the award of the Degree of**

**MASTER OF SCIENCE
(INFORMATION TECHNOLOGY)**

By

Anjali Rameshwar Nimje



DEPARTMENT OF INFORMATION TECHNOLOGY

KERALEEYA SAMAJAM (REGD.) DOMBIVLI'S

MODEL COLLEGE (AUTONOMOUS)

Re-Accredited 'A' Grade by NAAC

(Affiliated to University of Mumbai)

FOR THE YEAR

(2023-24)



Keraleeya Samajam(Regd.) Dombivli's

MODEL COLLEGE

Re-Accredited Grade "A" by NAAC

Kanchan Goan Village, Khambalpada, Thakurli East – 421201
Contact No – 7045682157, 7045682158. www.model-college.edu.in

DEPARTMENT OF INFORMATION TECHNOLOGY AND COMPUTER SCIENCE

CERTIFICATE

This is to certify that Mr. /Miss _____

Studying in Class _____ *Seat No.* _____

Has completed the prescribed practicals in the subject _____

During the academic year _____

Date : _____

External Examiner

Internal Examiner
M.Sc. Information Technology

INDEX

SR NO	TITLE	DATE	PAGE NO	SIGNATURE
1	Design an Expert System using AIML	16/10/23	04	
2	Design a Bot using AIML	16/10/23	06	
3	Implement Bayes Theorem using Python	22/10/23	10	
4	Implement Conditional Probability using Python	22/10/23	12	
4A	Conditional Probability	22/10/23	13	
4B	Joint Probability	22/10/23	14	
5	Implement Rule-Based System	13/10/23	15	
6	Design a Fuzzy Based Application using Python	13/11/23	17	
7	Write an application to stimulate supervised learning	13/11/23	20	
8	Write an application to implement Clustering	26/11/23	23	
9	Write an application to Implement Support Vector Machine Algorithm	27/11/23	25	

PRACTICAL 1 :DESIGN AN EXPERT SYSTEM USING AIML

Theory:

An expert system is a computer-based artificial intelligence (AI) system that emulates the decision-making abilities of a human expert in a particular domain or field of knowledge. It is designed to provide expert-level advice and solutions to problems within its specific area of expertise. Expert systems are a subset of AI known as knowledge-based systems. Key components and characteristics of expert systems include:

1. **Knowledge Base:** An expert system contains a knowledge base, which is a repository of information, facts, rules, and heuristics relevant to the domain it specializes in. This knowledge is typically acquired from human experts and is represented in a structured and formal manner.
2. **Inference Engine:** The inference engine is the core of the expert system. It uses the knowledge stored in the knowledge base to make decisions, solve problems, and provide recommendations. It employs reasoning mechanisms, such as rule-based reasoning or fuzzy logic, to draw conclusions based on the available information.
3. **User Interface:** An expert system usually has a user-friendly interface that allows users to interact with it, input queries or problems, and receive responses or solutions in a human-readable format. The interface can be text-based, graphical, or even voice-controlled.
4. **Explanation Facility:** Many expert systems include an explanation facility that can explain the reasoning behind the system's conclusions or recommendations. This is important for users to understand why a particular decision was made.
5. **Knowledge Acquisition:** Expert systems require a process for acquiring, updating, and refining their knowledge base. This involves extracting expertise from human experts, codifying it into a format suitable for the system, and continually improving the knowledge base over time.
6. **Domain Specificity:** Expert systems are typically designed for specific domains or industries, such as medical diagnosis, financial planning, or troubleshooting technical issues. They excel in narrow, well-defined areas where expert knowledge is crucial.
7. **Limited Decision-Making:** Expert systems are not general-purpose AI systems and are limited to the specific domain for which they are designed. They cannot perform tasks outside their designated expertise.
8. **Examples:** Some well-known expert systems include MYCIN (used for medical diagnosis), Dendral (used for organic chemistry), and expert systems used in various industries for quality control, maintenance, and decision support. Expert systems have been applied in various fields to assist professionals in decision-making, problem-solving, and knowledge dissemination. While they are not as flexible as general purpose AI systems like deep learning neural networks, they excel in situations where a structured, rule-based approach is suitable and where human expertise is crucial.

Source code

```
import aiml
def main():
    kernel = aiml.Kernel()
    kernel.learn("flu.aiml")
    print("Welcome to the flu diag center")
    while True:
        user_input = input("You: ").strip().lower()
        if user_input == 'exit':
            print("Goodbye!")
            break
        response = kernel.respond(user_input)
        print("Expert System: " + response)
if __name__ == "__main__":
    main()
```

'Flu.aiml

```
<aiml>
<category>
<pattern>WELCOME</pattern>
<template>Welcome to the flu diag center. Please enter your symptoms.</template>
</category>
<category>
<pattern>FEVER</pattern>
<template>No flu.</template>
</category>
<category>
<pattern>COUGH</pattern>
<template>No flu.</template>
</category>
<category>
<pattern>FEVER COUGH</pattern>
<template>You may have the flu.</template>
</category>
</aiml>
```

Output

```
Loading flu.aiml...done (0.00 seconds)
Welcome to the flu diag center
You: cough
Expert System: No flu.
You: fever
Expert System: No flu.
You: cough fever
WARNING: No match found for input: cough fever
Expert System:
You: fever cough
Expert System: You may have the flu.
You: exit
Goodbye!
```

PRACTICAL 2: TO DESIGN A BOT USING AIML

Theory:

Designing a bot using AIML (Artificial Intelligence Markup Language) involves creating a conversational agent that can understand and respond to user inputs in a natural and context-aware manner. AIML is a markup language specifically designed for building chatbots and virtual assistants, and it forms the basis for many AI chatbot implementations. Here are the key steps to design a bot using

AIML:

1. Define the Purpose and Scope:

- Begin by clearly defining the purpose and scope of your chatbot. Determine what tasks or conversations the bot will handle and what its main objectives are. This could range from customer support to providing information or entertainment.

2. Create an AIML Knowledge Base:

- AIML bots rely on a knowledge base of patterns and responses. Develop a set of AIML files that contain these patterns and responses. AIML files consist of categories, which are the building blocks of conversations, and each category typically includes a pattern and a template for the response.

3. Understand AIML Syntax:

- Familiarize yourself with AIML syntax, which includes elements like `<category>`, `<pattern>`, `<template>`, and various wildcard and meta tags. These elements are used to structure the knowledge base and define how the bot should respond to user inputs.

4. Build the Knowledge Base:

- Populate your AIML knowledge base with a variety of patterns and responses. Consider common user queries, variations, and possible inputs. Use AIML's wildcard and substitution mechanisms to make your bot's responses dynamic and context-aware.

5. Test and Refine:

- Test your bot's responses with various user inputs to ensure that it can handle different scenarios effectively. Continuously refine and expand your AIML knowledge base based on user interactions and feedback.

6. Implement Memory and Context:

- AIML bots can have memory and context capabilities. Implement these features to make your bot more conversational and capable of maintaining context over multiple turns in a conversation. You can use AIML sets and get elements to manage variables and user context.

7. Handle Special Cases:

- Consider how your bot will handle special cases, such as handling errors gracefully, providing fallback responses when it doesn't understand a user query, and managing sensitive or inappropriate content.

8. Integration:

- Integrate your AIML-based bot with the platform or channels where it will be deployed. This might involve integrating with messaging apps, websites, or other communication channels.

9. Natural Language Processing (Optional):

- While AIML is a rule-based approach, you can enhance your bot's capabilities by integrating natural language processing (NLP) techniques to better understand and generate responses. AIML can work alongside NLP components to improve conversational flow.

10. Monitor and Maintain:

- After deployment, continuously monitor your bot's performance and gather user feedback. Make updates and improvements to the AIML knowledge base to keep the bot relevant and effective.

11. Scale and Evolve:

- As your bot gains popularity and handles more tasks, consider expanding its capabilities and knowledge base to meet evolving user needs and expectations. Designing a bot using AIML can be a rewarding process, allowing you to create a conversational agent that can interact with users in a human-like way. However, keep in mind that AIML has limitations, and for more advanced AI capabilities, you may need to integrate additional technologies and machine learning approaches. We create a bot which interacts with the users asking them some information about themselves such as name, age etc. The following is the Python code for the given bot.

Source code


```
import aiml
kernel = aiml.Kernel()
kernel.learn("bot1.aiml")
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        print("Bot: Goodbye!")
        break
    bot_response = kernel.respond(user_input)
    print("Bot:", bot_response)
    bot_response = kernel.respond(user_input)
    print("Chatbot:", bot_response)
```

Bot1.aiml

```
<aiml version="1.0.1" encoding="UTF-8">
<category>
<pattern>HELLO</pattern>
<template>
Well, hello!
</template>
</category>
<category>
```

<pattern>WHAT ARE YOU</pattern>
<template>
I'm a bot!!
</template>
</category>
<category>
<pattern>SUNDAY</pattern>
<template>
the day of the week before monday and following saturday.
</template>
</category>
<category>
<pattern>MONDAY</pattern>
<template>
the day of the week before tuesday and following sunday.
</template>
</category>
<category>
<pattern>TUESDAY</pattern>
<template>
the day of the week before wednesday and following monday.
</template>
</category>
<category>
<pattern>WEDNESDAY</pattern>
<template>
the day of the week before thursday and following tuesday.
</template>
</category>
<category>
<pattern>THURSDAY</pattern>
<template>
the day of the week before friday and following wednesday.
</template>
</category>
<category>
<pattern>FRIDAY</pattern>
<template>
the day of the week before saturday and following thursday.
</template>
</category>
<category>
<pattern>SATURDAY</pattern>
<template>
the day of the week before sunday and following friday.
</template>
</category>
</aiml>

OUTPUT

```
***  Loading bot1.ai1...done (0.00 seconds)
You: hello
Bot: Well, hello!
Chatbot: Well, hello!
You: monday
Bot: the day of the week before tuesday and following sunday.
Chatbot: the day of the week before tuesday and following sunday.
You: tuesday
Bot: the day of the week before wednesday and following monday.
Chatbot: the day of the week before wednesday and following monday.
You: wednesday
Bot: the day of the week before thursday and following tuesday.
Chatbot: the day of the week before thursday and following tuesday.
You: thursday
Bot: the day of the week before friday and following wednesday.
Chatbot: the day of the week before friday and following wednesday.
You: friday
Bot: the day of the week before saturday and following thursday.
Chatbot: the day of the week before saturday and following thursday.
You: saturday
Bot: the day of the week before sunday and following friday.
Chatbot: the day of the week before sunday and following friday.
You: exit
Bot: Goodbye!
```

PRACTICAL 3 :IMPLEMENT BAYES THEOREM USING PYTHON

Theory:

Bayes' Theorem provides a way that we can calculate the probability of a piece of data belonging to a given class, given our prior knowledge. Bayes' Theorem is stated as:

$$P(\text{class}|\text{data}) = (P(\text{data}|\text{class}) * P(\text{class})) / P(\text{data})$$

Where $P(\text{class}|\text{data})$ is the probability of class given the provided data. Naive Bayes is a classification algorithm for binary (two-class) and multiclass classification problems. It is called Naive Bayes or idiot Bayes because the calculations of the probabilities for each class are simplified to make their calculations tractable.

CODE

```
def drug_user(prob_th=0.8,
              sensitivity=0.79,
              specificity=0.79,
              prevelence=0.02,
              verbose=True):
    p_user=prevelence
    p_non_user=1-prevelence
    p_pos_user=sensitivity
    p_neg_user=specificity
    p_pos_non_user=1-specificity

    num=p_pos_user*p_user
    den=p_pos_non_user*p_user+p_pos_non_user

    prob=num/den
    if verbose:
        if prob>prob_th:
            print("The test-taker could be an user")
        else:
            print("The test-taker may not be an user")
    return prob

print("Anjali Nimje")
p=drug_user(prob_th=0.5,sensitivity=0.97,specificity=0.95,prevelence=0.005)
print("probability of test-taker being a drug user is:",round(p,3))
```

OUTPUT

```
== RESTART: C:/Users/Student011/Desktop/ANJALI AAI/bayes theorem prac 3.py ==
Anjali Nimje
The test-taker may not be an user
probability of test-taker being a drug user is: 0.097
>>> |
```

CODE

```
def bayes_theorem(prior_prob,likelihood,evidence):
    posterior_prob=(likelihood*prior_prob)/evidence
    return posterior_prob

if __name__=="__main__":
    prior_prob=0.01
    likelihood_cancer=0.95
    likelihood_no_cancer=0.10
    evidence=(likelihood_cancer*prior_prob)+(likelihood_no_cancer*(1-prior_prob))
    posterior_prob=bayes_theorem(prior_prob,likelihood_cancer,evidence)
    print("Prior Probability of Cancer:",likelihood_cancer)
    print("Likelihood of Positive Test Given No Cancer:",round(posterior_prob,2))
```

Output

```
>>>
RESTART: C:/Users/Student011/Desktop/ANJALI AAI/bayes theorem prac 3 part 2.py
Prior Probability of Cancer: 0.95
Likelihood of Positive Test Given No Cancer: 0.09
>>>
```

PRACTICAL 4: IMPLEMENT CONDITIONAL PROBABILITY USING PYTHON

Theory:

What is Conditional Probability?

The probability of one event given the occurrence of another event is called the conditional probability. The conditional probability of one to one or more random variables is referred to as the conditional probability distribution.

For example, the conditional probability of event A given event B is written formally as:

- $P(A \text{ given } B)$

The “given” is denoted using the pipe “|” operator; for example:

- $P(A | B)$

The conditional probability for events A given event B is calculated as follows:

- $P(A \text{ given } B) = P(A \text{ and } B) / P(B)$

Code:

```
def conditiona():
    pass_stats=0.15
    pass_codingWstats=0.60
    pass_codingWOstats=0.40
    prob_both=pass_stats*pass_codingWstats
    print("The probability that applicant passes both is",round(prob_both,3))
    prob_coding=(prob_both)+((1-pass_stats)*pass_codingWOstats)
    print("Probability that he/she passes only coding is",round(prob_coding,3))
    status_given_coding=prob_both/prob_coding
    print("Conditional Probability is",round(status_given_coding,3))
print("ANJALI NIMJE")
conditiona()
```

Output

```
>>>
RESTART: C:/Users/Student011/Desktop/ANJALI AAI/conditional probability prac 4.py
ANJALI NIMJE
The probability that applicant passes both is 0.09
Probability that he/she passes only coding is 0.43
Conditional Probability is 0.209
>>> |
```

PRACTICAL 4.A: CONDITIONAL PROBABILITY

Code:

```
def get_valid_probability_input(prompt):
    while True:
        try:
            probability=float(input(prompt))
            if 0<= probability <=1:
                return probability
            else:
                print("probability must be between 0 and 1. please try")
        except ValueError:
            print("Invalid input. please enter a valid probability between 0 and 1.")
P_B= get_valid_probability_input("Enter the probability of event B(0 to 1):")
P_A_and_B=get_valid_probability_input("Enter the probability of events A and B (0 to 1):")
P_A_given_B= P_A_and_B/P_B
if P_A_given_B >1:
    print("Inconsistent result.Please check the inputs again.")
else:
    formatted_result = "{:.2f}".format(P_A_given_B)
    print("P(A|B)=", formatted_result)
```

Output

```
>>>
----- RESTART: D:/ANJALI AI/p4aai.py -----
Enter the probability of event B(0 to 1):1.2
probability must be between 0 and 1. please try
Enter the probability of event B(0 to 1):0.7
Enter the probability of events A and B (0 to 1):0.4
P(A|B)= 0.57
>>> |
```

PRACTICAL 4B: JOINT PROBABILITY

Code:

```
while True:
    try:
        P_A=float(input("Enter the probability of event A(0 to 1):"))
        if 0<= P_A <=1:
            break
        else:
            print("probability must be between 0 and 1. please try again")
    except ValueError:
        print("Invalid input. please enter a valid probability between 0 and 1.")
while True:
    try:
        P_B=float(input("Enter the probability of event B(0 to 1):"))
        if 0<= P_B <=1:
            break
        else:
            print("probability must be between 0 and 1. please try again")
    except ValueError:
        print("Invalid input. please enter a valid probability between 0 and 1.")
P_A_and_B=P_A*P_B
formatted_result = "{:.2f}".format(P_A_and_B)
print("P(A and B)",formatted_result)
```

Output

```
>>>
===== RESTART: D:/ANJALI AI/4b joint.py =====
Enter the probability of event A(0 to 1):0.5
Enter the probability of event B(0 to 1):0.5
P(A and B) 0.25
>>> |
```

PRACTICAL 5: IMPLEMENT A RULE BASED SYSTEM

Theory:

A rule-based system, also known as a rule-based expert system or a knowledge-based system, is a type of computer program or artificial intelligence (AI) system that uses a set of explicitly defined rules to make decisions or solve problems. These rules are typically represented as "if-then" statements, where specific conditions trigger certain actions or conclusions. Rule-based systems are widely used in various domains, including expert systems, decision support systems, and business logic applications.

Here's a basic structure of how a rule-based system works:

1. **Knowledge Base (KB):** The knowledge base is a repository of rules and facts. It contains a collection of rules that encode the expertise or knowledge in a particular domain. These rules are typically written by domain experts and are used to represent knowledge in a formal and structured way.

2. **Inference Engine:** The inference engine is the core of the rule-based system. It processes input data, matches it against the rules in the knowledge base, and makes inferences or decisions based on the conditions specified in the rules. It follows a systematic process to determine which rules are applicable and how they should be applied.

3. **Working Memory:** Working memory is a temporary storage area where the system keeps track of relevant facts and information during the reasoning process. As rules are applied, the working memory is updated with new facts and conclusions.

4. **User Interface:** Rule-based systems often have a user interface that allows users or operators to interact with the system, provide input, and receive output or recommendations based on the rules.

Rule 1: If Person A is the mother of Person B, then there is a "Mother" relation between Person A and Person B.

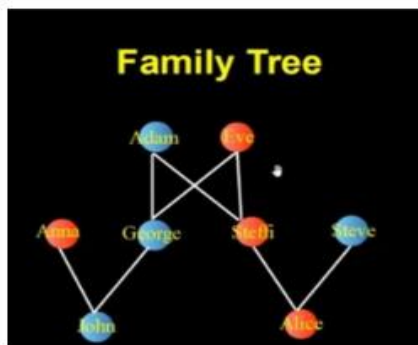
Rule 2: If Person A is the father of Person B, then there is a "Father" relation between Person A and Person B. Similar rules can be used to define the other relationship

In this family tree context, these rules can be used to represent relationships within a family, allowing the inference engine to determine family connections based on the defined rules.

In this example, the rules represent medical knowledge, and the inference engine processes patient data to make diagnoses and recommendations based on these rules.

Rule-based systems are known for their transparency and explainability, as it's easy to trace the reasoning process by examining the rules that were applied. However, they may not handle uncertainty or complex reasoning as effectively as some other AI techniques, such as probabilistic models or machine learning algorithms.

We consider the following family tree and express it using Prolog.



Code:

```
male(adam).
male(george).
male(steve).
male(john).
female(alice).
female(anna).
female(eve).
female(steffi).
parent(adam,george).
parent(eve,george).
parent(adam,steffi).
parent(eve,steffi).
parent(george,john).
parent(anna,john).
parent(steffi,alice).
parent(steve,alice).
mother(X,Y) :- parent(X,Y), female(X).
father(X,Y) :- parent(X,Y), male(X).
sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X),X\==Y.
brother(X,Y) :- parent(Z,X), parent(Z,Y), male(X),X\==Y.
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
grandmother(X,Z) :- mother(X,Y), parent(Y,Z).
siblings(X,Y) :- (brother(X,Y);sister(X,Y)),X\==Y.
uncle(X,Y) :- parent(Z,Y),brother(X,Z).
aunty(X,Y) :- parent(Z,Y),sister(X,Z).
cousin(X,Y) :- parent(A,X),parent(B,Y),siblings(A,B).
```

Output

```
% d:/anjali ml/familytree.pl compiled 0.00 sec, 26 clauses
?-
|      grandfather(X,Y) .
X = adam ,
Y = john ,
?- grandmother(X,Y) .
X = eve ,
Y = john ,
?- father(X,Y) .
X = adam ,
Y = george ,
?- mother(X,Y) .
X = eve ,
Y = george ,
?- cousin(X,Y) .
X = john ,
Y = alice ,
?- father(adam,_).
true .
?- father(adam,X) .
X = george ,
?- ■
```


PRACTICAL 6: DESIGN A FUZZY BASED APPLICATION USING PYTHON

Theory:

A Designing a fuzzy-based application, such as solving the classic "Tipping Problem," involves creating a system that uses fuzzy logic to make decisions or provide recommendations based on imprecise or vague inputs. The Tipping Problem is a common example used to demonstrate fuzzy logic in action, where you need to determine the appropriate tip amount in a restaurant based on the quality of service and the food. The following steps show how to design a fuzzy-based application for the Tipping Problem:

1. Define the Problem:

Clearly define the problem you want to solve. In this case, the problem is to calculate an appropriate tip amount based on the quality of service and food at a restaurant.

2. Identify Input Variables:

Determine the input variables that influence the tipping amount. In the Tipping Problem, common input variables are "Service Quality" and "Food Quality." These variables are often expressed as linguistic terms (e.g., "poor," "good," "excellent").

3. Define Membership Functions:

Create membership functions for each input variable to represent their fuzzy sets.

Membership functions define how each linguistic term relates to a numerical value. For example, "good service" might be represented by a triangular membership function with values ranging from 0 to 10.

4. Determine Output Variable:

Identify the output variable, which is the "Tipping Amount" in this case. Define the linguistic terms associated with it, such as "low," "medium," and "high."

5. Define Output Membership Functions:

Create membership functions for the output variable to describe how the tipping amount relates to linguistic terms like "low," "medium," and "high."

6. Establish Fuzzy Rules:

Define a set of fuzzy rules that connect the input variables to the output variable. For example, a rule might be "If Service Quality is good and Food Quality is excellent, then Tipping Amount is high."

7. Implement the Inference Engine:

Build the inference engine to evaluate the fuzzy rules using the input values. This involves fuzzification (converting crisp input values into fuzzy values), rule evaluation, and aggregation of rule outputs.

8. Perform Defuzzification:

Convert the fuzzy output values into a crisp value using a defuzzification method. Common methods include the centroid method, max membership method, or weighted average.

9. Develop a User Interface:

Create a user-friendly interface that allows users to input the quality of service and food, and then displays the recommended tipping amount.

10. Testing and Validation:

Test the system with various input combinations to ensure it provides reasonable and consistent results. Adjust membership functions and rules if necessary.

11. Fine-Tuning:

Fine-tune the system based on feedback and real-world data, if available, to improve its performance and accuracy.

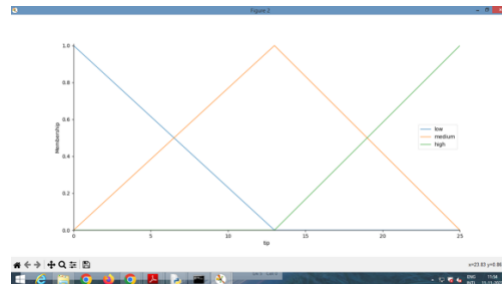
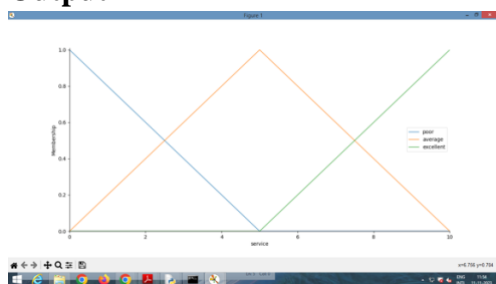
12. Deployment:

Once satisfied with the system's performance, deploy it for practical use. Ensure it can handle real-time inputs and maintain its accuracy. In designing a fuzzy-based application like the Tipping Problem, it's essential to have a good understanding of fuzzy logic principles and to carefully design the membership functions and rules to accurately capture the problem's inherent fuzziness. This approach can be applied to various other decision-making scenarios with imprecise data.

Code :

```
#import the library pip install numpy
#import the library pip install scikit-fuzzy
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
# Create fuzzy variables and their ranges
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')
# Define membership functions for quality and service
quality['poor'] = fuzz.trimf(quality.universe, [0, 0, 5])
quality['average'] = fuzz.trimf(quality.universe, [0, 5, 10])
quality['excellent'] = fuzz.trimf(quality.universe, [5, 10, 10])
service['poor'] = fuzz.trimf(service.universe, [0, 0, 5])
service['average'] = fuzz.trimf(service.universe, [0, 5, 10])
service['excellent'] = fuzz.trimf(service.universe, [5, 10, 10])
# Define membership functions for the tip
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])
# Define fuzzy rules
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(quality['excellent'] | service['excellent'], tip['high'])
# Create the fuzzy control system
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
# Create a simulation
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)
# Input values for quality and service
tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8
# Compute the tipping result
tipping.compute()
# Print the output tip value
print("Recommended tip:", tipping.output['tip'])
# Visualize the membership functions and the output
quality.view()
service.view()
tip.view()
```

Output



PRACTICAL 7: WRITE AN APPLICATION TO SIMULATE SUPERVISED LEARNING MODEL.

Theory

Supervised learning is a type of machine learning where the algorithm is trained on a labeled dataset, meaning that each input data point is associated with a corresponding target or output label. The goal is for the algorithm to learn a mapping from inputs to outputs, enabling it to make predictions or classifications on new, unseen data. Here are key concepts and theories related to supervised learning:

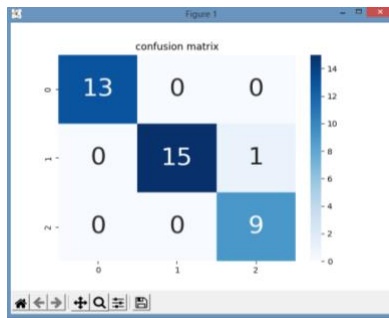
- **Training Data and Labels:**
 - **Theory:** In supervised learning, a model is trained on a dataset that includes both input features and their corresponding output labels. The model learns to map inputs to outputs based on this labeled training data.
 - **Practical Application:** Examples include image classification, where images are labeled with specific categories, and regression tasks, where the goal is to predict a continuous output.
- **Loss Functions:**
 - **Theory:** Loss functions quantify the difference between the predicted outputs and the actual labels. The goal during training is to minimize this loss, adjusting the model's parameters to improve its predictions.
 - **Practical Application:** Common loss functions include mean squared error for regression tasks and cross-entropy loss for classification tasks.
- **Model Architecture:**
 - **Theory:** The architecture of a supervised learning model defines its structure, including the number and type of layers, as well as the connections between them. Different architectures are suitable for different types of tasks.
 - **Practical Application:** Convolutional Neural Networks (CNNs) are commonly used for image classification, while Recurrent Neural Networks (RNNs) are effective for sequence data, such as natural language processing tasks.
- **Overfitting and Regularization:**
 - **Theory:** Overfitting occurs when a model learns the training data too well, including its noise, and performs poorly on new, unseen data. Regularization techniques are employed to prevent overfitting.
 - **Practical Application:** Common regularization methods include dropout, which randomly removes connections during training, and L1/L2 regularization, which adds penalties to the model's parameters.
- **Hyperparameter Tuning:**
 - **Theory:** Hyperparameters are settings that are not learned during training but are set before training begins. Tuning these hyperparameters is crucial for achieving optimal model performance.
 - **Practical Application:** Grid search or randomized search can be employed to find the best combination of hyperparameters for a given model.
- **Cross-Validation:**
 - **Theory:** Cross-validation is a technique used to assess how well a model will generalize to new, unseen data. It involves splitting the dataset into multiple subsets for training and testing.

- Practical Application: k-fold cross-validation is commonly used, where the dataset is divided into k subsets, and the model is trained and tested k times, with each subset serving as the test set once.
- Ensemble Learning:
 - Theory: Ensemble learning involves combining the predictions of multiple models to improve overall performance. Popular ensemble methods include bagging (e.g., Random Forests) and boosting (e.g., Gradient Boosting).
 - Practical Application: Ensemble methods are often used to enhance the robustness and accuracy of models, especially in scenarios where a single model may not perform well.
- Transfer Learning:
 - Theory: Transfer learning involves using a pre-trained model on a related task to improve performance on a new, target task. This leverages knowledge gained from one task to enhance learning on another task.
 - Practical Application: Transfer learning is commonly used in computer vision and natural language processing, where pre-trained models on large datasets (e.g., ImageNet for images, BERT for language) are fine-tuned for specific tasks with limited labeled data.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
dataset = datasets.load_iris()
x = dataset.data[:,[0,1,2,3]]
y = dataset.target
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, random_state=0)
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.fit_transform(x_test)
classifier = LogisticRegression(random_state=0, solver='lbfgs', multi_class='auto')
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
probs_y = classifier.predict_proba(x_test)
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)
import seaborn as sns
import pandas as pd
ax = plt.axes()
df_cm = cm
sns.heatmap(df_cm, annot=True, annot_kws={"size":30}, fmt='d', cmap='Blues', ax=ax)
ax.set_title('confusion matrix')
plt.show()
```

OUTPUT



```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 19:29:22) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:/anjali ml/supervised7A.py =====
[[13  0  0]
 [ 0 15  1]
 [ 0  0  9]]
>>>
```

PRACTICAL 8: WRITE AN APPLICATION IMPLEMENT CLUSTERING ALGORITHM.

Theory :

Clustering algorithms are a fundamental part of unsupervised machine learning used to group data points into clusters based on similarities. There are various theories and approaches to clustering algorithms, each with its strengths and applications. Here are a few notable theories:

- **Distance-Based Clustering:**
 - **K-means:** This is one of the most popular centroid-based clustering algorithms. It aims to partition data points into 'k' clusters by minimizing the within-cluster variance. It iteratively assigns points to the nearest centroid and updates the centroids' positions until convergence.
 - **Hierarchical Clustering:** This method builds a hierarchy of clusters. It can be agglomerative (bottom-up) or divisive (top-down) and creates clusters by either merging or splitting them based on the distance between data points.
- **Density-Based Clustering:**
 - **DBSCAN:** Density-Based Spatial Clustering of Applications with Noise identifies clusters based on areas of high density separated by areas of low density. It can find arbitrarily shaped clusters and handle noise well.
- **Probabilistic-Based Clustering:**
 - **Expectation-Maximization (EM) Algorithm:** Often used in Gaussian Mixture Models (GMM), this algorithm assumes data points are generated from a mixture of several Gaussian distributions. It iteratively estimates the parameters of these distributions to assign data points to clusters.
- **Spectral Clustering:**
 - Utilizes the eigenvectors of a similarity matrix derived from the data to reduce the dimensionality and then performs clustering in the reduced space. It's effective for non-linearly separable clusters.
- **Fuzzy Clustering:**
 - Algorithms like Fuzzy C-means allow data points to belong to multiple clusters with varying degrees of membership. It assigns probabilities rather than hard assignments.

The theories underlying these algorithms involve concepts like distance metrics (Euclidean, Manhattan, etc.), cluster representation (centroids, medoids), and optimization techniques (minimizing within-cluster variance, maximizing inter-cluster distances, etc.).

Choosing the right clustering algorithm often depends on the nature of the data, the desired number of clusters, computational efficiency, and the shape and distribution of the clusters in the data.

Each algorithm has its assumptions and parameters, which can impact its performance on different datasets. Understanding these theories helps in selecting the appropriate algorithm for specific clustering tasks.

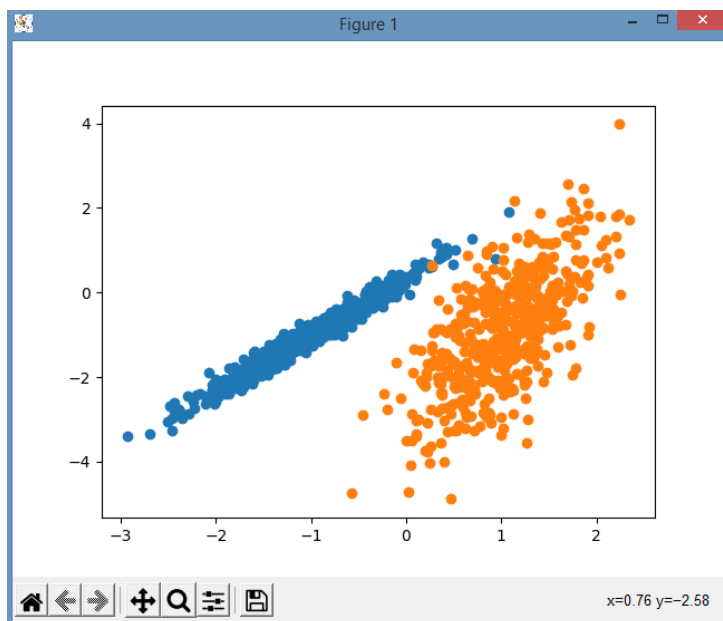
Code:

```
from numpy import where
from sklearn.datasets import make_classification
from matplotlib import pyplot

x, y = make_classification(n_samples=1000, n_features=2,
n_informative=2, n_redundant=0, n_clusters_per_class=1,
random_state=4)

for class_value in range(2):
    row_ix = where(y == class_value)
    pyplot.scatter(x[row_ix, 0], x[row_ix, 1])
pyplot.show()
```

Output



PRACTICAL:9: WRITE AN APPLICATION TO IMPLEMENT SUPPORT VECTOR MACHINE ALGORITHM.

Theory

Support Vector Machines (SVM) is a powerful supervised learning algorithm used for classification and regression tasks. The main idea behind SVM is to find a hyperplane that best separates the data into different classes. Here's the theory behind SVM and how it can be applied in practice:

SVM Theory:

- **Linear Separation:**
 - **Theory:** SVM aims to find a hyperplane in the feature space that best separates the data into different classes. This hyperplane is chosen to maximize the margin, which is the distance between the hyperplane and the nearest data points from each class.
 - **Practical Application:** SVM is effective for binary classification tasks where the goal is to separate data points of two classes with a clear margin.
- **Support Vectors:**
 - **Theory:** Support vectors are the data points that lie closest to the decision boundary (hyperplane). These are the critical points that determine the position and orientation of the hyperplane.
 - **Practical Application:** SVM focuses on the support vectors during training, and the decision boundary is primarily influenced by these key data points.
- **Kernel Trick:**
 - **Theory:** SVM can be extended to handle non-linear decision boundaries by using the kernel trick. Kernels transform the original feature space into a higher-dimensional space, making it possible to find a hyperplane in this transformed space.
 - **Practical Application:** Common kernels include the linear kernel, polynomial kernel, and radial basis function (RBF) kernel. The choice of kernel depends on the nature of the data and the desired decision boundary.
- **Margin and Regularization:**
 - **Theory:** SVM introduces the concept of a margin, which is the distance between the hyperplane and the nearest data point. Maximizing this margin leads to better generalization on unseen data. Regularization parameters control the trade-off between achieving a wide margin and minimizing classification errors.
 - **Practical Application:** The regularization parameter (often denoted as C) is a tuning parameter that influences the balance between achieving high accuracy on the training data and preventing overfitting.
- **Soft Margin SVM:**
 - **Theory:** In situations where the data is not perfectly separable, a soft margin SVM allows for some misclassifications by introducing a penalty for errors. This helps achieve a balance between maximizing the margin and allowing for some flexibility in the decision boundary.
 - **Practical Application:** Soft margin SVM is useful when dealing with noisy or overlapping data, as it provides a more robust solution.
- **Multi-Class Classification:**

- Theory: SVM is inherently a binary classifier, but it can be extended to handle multi-class classification using strategies such as one-vs-one or one-vs-all. In one-vs-one, a separate binary classifier is trained for each pair of classes, and the final decision is based on a voting scheme. In one-vs-all, each class is treated as a separate binary classification problem.
- Practical Application: SVM can be applied to tasks with more than two classes by employing these strategies.
- Grid Search and Cross-Validation:
 - Theory: When working with SVM, it's common to perform grid search and cross-validation to find the optimal combination of hyperparameters, including the choice of kernel and regularization parameter.
 - Practical Application: Grid search involves trying different hyperparameter values and selecting the combination that yields the best performance on a validation set. Cross-validation helps assess how well the model generalizes to new data.

Practical Tips for SVM:

- Feature Scaling:
 - Theory: SVM is sensitive to the scale of input features. It's important to scale features to ensure that no single feature dominates the others during the training process.
 - Practical Application: Standardization or normalization of features is often performed before training an SVM model.
- Data Preprocessing:
 - Theory: SVM performance can be affected by outliers and noise in the data. Data preprocessing steps, such as handling missing values and removing outliers, can contribute to improved model robustness.
 - Practical Application: Clean and preprocess data before training an SVM model to enhance its performance.
- Grid Search for Hyperparameter Tuning:
 - Theory: Grid search involves trying different combinations of hyperparameter values to find the optimal set. This is crucial for achieving the best performance of the SVM model.
 - Practical Application: Use grid search along with cross-validation to find the optimal hyperparameter values for the specific problem at hand.
- Model Interpretability:
 - Theory: SVM provides not only accurate predictions but also a clear decision boundary. This can be useful for interpreting and understanding the model's behavior.
 - Practical Application: Visualizing the decision boundary and support vectors can provide insights into how the model is making decisions.
- Handling Imbalanced Data:
 - Theory: When dealing with imbalanced classes, SVM might be biased towards the majority class. Techniques like adjusting class weights or using different evaluation metrics can address this issue.
 - Practical Application: Consider strategies such as assigning different weights to classes or using precision, recall, or F1-score as evaluation metrics in imbalanced classification problems.

By understanding the theoretical foundations of SVM and applying practical considerations, you can effectively use SVM for various supervised learning tasks, especially in scenarios where clear class separation is important.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
iris = datasets.load_iris()
x = iris.data[:, :2]
y = iris.target
C= 1.0
svc = svm.SVC(kernel='linear', C = 1, gamma='scale').fit(x, y)
x_min, x_max = x[:, 0].min() - 1, x[:, 0].max() + 1
y_min, y_max = x[:, 1].min() - 1, x[:, 1].max() + 1
h= (x_max / x_min)/100
xx,yy = np.meshgrid(np.arange(x_min,x_max,h),np.arange(y_min,y_max,h))
plt.subplot(1,1,1)
z= svc.predict(np.c_[xx.ravel(),yy.ravel()])
z=z.reshape(xx.shape)
plt.contour(xx, yy,z, cmap=plt.cm.Paired, alpha=0.8)
plt.scatter(x[:,0], x[:,1], c=y, cmap=plt.cm.Paired)
plt.xlabel('sepal length')
plt.ylabel('sepal width')
plt.xlim(xx.min(), xx.max())
plt.title('SVC with linear kernel')
plt.show()
```

Output

