**PYTHON-BASED NOTEBOOK ENVIRONMENTS**

**AIM:** Introduction to Python-based notebook environments

**DESCRIPTION:**

A Python-based notebook environment is an interactive computational environment that allows users to write and execute code, visualize data, and document their workflow in a single, cohesive interface. These environments are widely used in data science, machine learning, and scientific computing due to their versatility and ease of use.

**PYTHON-BASED NOTEBOOK ENVIRONMENTS:**

**1. JUPYTER NOTEBOOK:**

Jupyter Notebook is an open-source web application that allows you to create and share documents containing live code, equations, visualizations, and narrative text. It's widely used for data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

**Key Features:**

1. **Interactive Code Execution**: Run code in small, manageable chunks or cells, and view the results immediately.
2. **Rich Text Support**: Use Markdown and LaTeX to add formatted text, equations, and other descriptive elements.
3. **Data Visualization**: Integrates with libraries like Matplotlib, Seaborn, and Plotly to create inline visualizations.
4. **Language Support**: Primarily used for Python but supports over 40 programming languages via kernels, such as R, Julia, and Scala.
5. **Inline Output**: Display outputs, such as graphs, tables, and images, directly within the notebook cells.
6. **Reproducibility**: Combine code, output, and documentation in a single document, making it easy to reproduce and share analyses.
7. **Extensibility**: Enhance functionality with a wide range of extensions for code linting, execution timing, and more.
8. **Collaboration**: Share notebooks via email, GitHub, or Jupyter's own nbviewer, and collaborate with others easily.

**2. GOOGLE COLAB:**

Google Colab, short for Colaboratory, is a free, cloud-based Jupyter notebook environment provided by Google. It allows users to write and execute Python code through the web, offering a powerful platform for machine learning, data analysis, and education.

**Key Features:**

1. **Free Access**: Provides free access to computing resources, including CPUs, GPUs, and TPUs.
2. **Cloud-Based**: No installation required; access your notebooks from any device with an internet connection.
3. **Collaboration**: Easily share notebooks and collaborate with others in real-time, similar to Google Docs.
4. **Integration with Google Drive**: Save and access notebooks directly from Google Drive.
5. **Pre-installed Libraries**: Comes with many popular Python libraries pre-installed, such as TensorFlow, Keras, and PyTorch.
6. **Interactive Code Execution**: Execute code in cells and see results immediately, with support for rich text and Markdown.
7. **Data Visualization**: Integrates with libraries like Matplotlib and Plotly for inline visualizations.
8. **Notebook Sharing**: Share notebooks via links, making it easy to distribute and collaborate on projects.

**3. PYCHARM:**

PyCharm is a widely-used Integrated Development Environment (IDE) specifically designed for Python development. Developed by JetBrains, it offers a robust set of tools and features to streamline the coding, debugging, and deployment processes.

**Key Features:**

1. **Code Editor**: Advanced code editor with syntax highlighting, code completion, and intelligent code navigation.
2. **Debugger**: Powerful integrated debugger with breakpoints, watches, and interactive console.
3. **Version Control**: Seamless integration with version control systems like Git, SVN, and Mercurial.
4. **Refactoring Tools**: Extensive refactoring capabilities to improve code structure and maintainability.
5. **Testing**: Built-in support for testing frameworks such as pytest, unittest, and Nose.
6. **Web Development**: Support for web frameworks like Django, Flask, and Pyramid, with templates and tools specific to web development.
7. **Database Tools**: Built-in tools to manage and query databases, with support for SQL, MongoDB, and more.
8. **Code Analysis**: Static code analysis to identify potential errors and improve code quality.
9. **Plugins**: Extensive plugin ecosystem to extend functionality, including support for other languages and frameworks.

**4. PYTHON IDLE:**

IDLE (Integrated Development and Learning Environment) is a simple, lightweight IDE for Python provided with the standard Python distribution. It is designed for beginners and those who need a quick and easy way to write, test, and debug Python code.

**Key Features:**

1. **Built-In Interpreter**: Interactive shell for executing Python commands and testing code snippets in real-time.
2. **Code Editor**: Basic text editor with syntax highlighting, auto-indentation, and line numbering.
3. **Debugger**: Integrated debugger with stepping, breakpoints, and call stack visibility.
4. **Execution**: Run Python scripts directly from the editor or the interactive shell.
5. **Output Window**: Separate output window to display program results, helping to keep the workspace organized.
6. **Cross-Platform**: Available on Windows, macOS, and Linux, making it accessible for all Python users.
7. **Lightweight**: Minimal setup and resource requirements, ideal for small projects and learning environments.
8. **Ease of Use**: Simple, intuitive interface suitable for beginners and educational purposes.

**5. IBM WATSON STUDIO:**

IBM Watson Studio is a cloud-based platform designed for data scientists, application developers, and subject matter experts to collaboratively and easily work with data. It offers a comprehensive suite of tools for data analysis, machine learning, and AI development.

**Key Features**:

1. **Integrated Environment**: Combines multiple tools for data preparation, analysis, modeling, and deployment in one platform.
2. **Collaboration**: Facilitates team collaboration with project-based workspaces, version control, and sharing capabilities.
3. **Data Refinery**: Provides data wrangling tools to clean, shape, and enrich data for analysis.
4. **Machine Learning**: Supports building, training, and deploying machine learning models with automated machine learning (AutoML) and custom model creation.
5. **Jupyter Notebooks**: Integrated Jupyter Notebook support for interactive coding in Python, R, and Scala.
6. **SPSS Modeler**: Includes SPSS Modeler for advanced statistical analysis and predictive modeling.
7. **AutoAI**: Automates the end-to-end data science workflow, including data preprocessing, model selection, and hyperparameter optimization.
8. **Deployment**: Simplifies model deployment and management with tools for real-time and batch scoring.
9. **Scalability**: Leverages IBM Cloud for scalable computing resources, accommodating large datasets and complex models.
10. **Integration**: Seamlessly integrates with various data sources, including databases, cloud storage, and IBM Cloud services.
11. **Visualization**: Offers rich data visualization tools to explore and present data insights effectively.
12. **Security**: Ensures data security and compliance with enterprise-grade security features and controls.

**A\* ALGORITHM FOR 8 PUZZLE PROBLEM**

**AIM:** To implement A\* algorithm for 8 puzzle problem.

**DESCRIPTION:**

* A\* Algorithm is one of the best and popular techniques used for path finding and graph traversals.
* A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.
* It is essentially a best first search algorithm.

A\* Algorithm works as-

* It maintains a tree of paths originating at the start node.
* It extends those paths one edge at a time.
* It continues until its termination criterion is satisfied.

A\* Algorithm extends the path that minimizes the following function-

**f(n) = g(n) + h(n)**

Here,

* ‘n’ is the last node on the path
* g(n) is the cost of the path from start node to node ‘n’
* h(n) is a heuristic function that estimates cost of the cheapest path from node ‘n’ to the goal node

**PROGRAM:**

import copy

class Puzzle:

def \_\_init\_\_(self, size, board):

self.size = size

self.board = board

def \_\_str\_\_(self):

result = ""

for row in self.board:

result += " ".join(map(str, row)) + "\n"

return result

def find\_blank(self):

for i in range(self.size):

for j in range(self.size):

if self.board[i][j] == 0:

return i, j

def move(self, direction):

i, j = self.find\_blank()

new\_board = copy.deepcopy(self.board)

if direction == "up" and i > 0:

new\_board[i][j], new\_board[i - 1][j] = new\_board[i - 1][j], new\_board[i][j]

elif direction == "down" and i >self.size - 1:

new\_board[i][j], new\_board[i + 1][j] = new\_board[i + 1][j], new\_board[i][j]

elif direction == "left" and j > 0:

new\_board[i][j], new\_board[i][j - 1] = new\_board[i][j - 1], new\_board[i][j]

elif direction == "right" and j < self.size - 1:

new\_board[i][j], new\_board[i][j + 1] = new\_board[i][j + 1], new\_board[i][j]

return Puzzle(self.size, new\_board)

def main():

size = 3

initial\_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

goal\_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

puzzle = Puzzle(size, initial\_state)

print("Initial State:")

print(puzzle)

moves = ["up","down", "left", "right"]

for move in moves:

new\_puzzle = puzzle.move(move)

print(move.capitalize())

print(new\_puzzle)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**OUTPUT:**

Initial State:

1 2 3

4 5 6

7 8 0

Up

1 2 3

4 5 0

7 8 6

Down

1 2 3

4 5 6

7 8 0

Left

1 2 3

4 5 6

7 0 8

Right

4 2 3

4 5 6

7 8 0

**FIND-S ALGORITHM**

**🡪BY TAKING DATA SET DIRECTLY :**

**AIM:** To implement find-s algorithm.

**DESCRIPTION:**

The find-S algorithm is a basic concept learning algorithm in machine learning. The find-S algorithm finds the most specific hypothesis that fits all the positive examples. We have to note here that the algorithm considers only those positive training example. The find-S algorithm starts with the most specific hypothesis and generalizes this hypothesis each time it fails to classify an observed positive training data. Hence, the Find-S algorithm moves from the most specific hypothesis to the most general hypothesis.

**PROGRAM:**

def find\_s\_algorithm(training\_data):

hypothesis = [None] \* (len(training\_data[0]) - 1)

for example in training\_data:

if example[-1] == 'Yes':

for i in range(len(example) - 1):

if hypothesis[i] is None:

hypothesis[i] = example[i]

elif hypothesis[i] != example[i]:

hypothesis[i] = '?'

return hypothesis

if \_\_name\_\_ == "\_\_main\_\_":

training\_data = [

['Green', 'Big', 'Round', 'Smooth', 'No'],

['Orange', 'Big', 'Round', 'Smooth', 'No'],

['Green', 'Big', 'Irregular', 'Rough', 'Yes'],

['Orange', 'Big', 'Irregular', 'Rough', 'Yes'],

['Green', 'Small', 'Round', 'Smooth', 'No'],

['Orange', 'Small', 'Round', 'Smooth', 'No'],

['Green', 'Small', 'Irregular', 'Rough', 'Yes'],

['Orange', 'Small', 'Irregular', 'Rough', 'Yes']

]

hypothesis = find\_s\_algorithm(training\_data)

print("Learned hypothesis:",hypothesis)

**OUTPUT:**

Learned hypothesis: ['?', '?', 'Irregular', 'Rough']

🡪 **DATA SET IMPORTED FROM A P ATH:**

**AIM:** To implement find-s algorithm.

**DESCRIPTION:**

The find-S algorithm is a basic concept learning algorithm in machine learning. The find-S algorithm finds the most specific hypothesis that fits all the positive examples. We have to note here that the algorithm considers only those positive training example. The find-S algorithm starts with the most specific hypothesis and generalizes this hypothesis each time it fails to classify an observed positive training data. Hence, the Find-S algorithm moves from the most specific hypothesis to the most general hypothesis.

**PROGRAM:**

import csv

def find\_s\_algorithm(training\_data):

hypothesis = [None] \* (len(training\_data[0]) - 1)

for example in training\_data:

if example[-1] == 'Yes':

for i in range(len(example) - 1):

if hypothesis[i] is None:

hypothesis[i] = example[i]

elif hypothesis[i] != example[i]:

hypothesis[i] = '?'

return hypothesis

if \_\_name\_\_ == "\_\_main\_\_":

training\_data = []

with open('/content/drive/MyDrive/trainingdata.csv', 'r') as csvfile:

reader = csv.reader(csvfile)

for row in reader:

training\_data.append(row)

hypothesis = find\_s\_algorithm(training\_data)

print("Learned hypothesis:", hypothesis)

**OUTPUT:**

Learned hypothesis: ['Sunny', 'Warm', '?', 'Strong', '?', '?']

**DATA SET :**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **sky** | **airTemp** | **humidity** | **wind** | **water** | **forecast** | **enjoySport** |
| Sunny | Warm | Normal | Strong | Warm | Same | Yes |
| Sunny | Warm | High | Strong | Warm | Same | Yes |
| Rainy | Cold | High | Strong | Warm | Change | No |
| Sunny | Warm | High | Strong | Cool | Change | Yes |

**HILL CLIMBING ALGORITHM TO SOLVE TRAVELLING SALES PERSON PROBLEM:**

**AIM:** To Implement and test hill climbing based search algorithms to solve Travelling Salesman Problem.

**DESCRIPTION:**

The TSP stands as one of the best known problems when it comes to work with NP-hard problems, which implies that no known algorithm exists to solve it in polynomial time. The problem can be summarized as follows : "Given a set of cities and the cost of travel (or distance) between each possible pairs, the TSP, is to find the best possible way of visiting all the cities and returning to the starting point that minimize the travel cost (or travel distance)." The exact solution to this problem with n cities can only be determined through evaluating (n-1)!/2 possibilities.

**PROGRAM:**

import itertools

import math

def distance(point1, point2):

return math.sqrt((point1[0] - point2[0])\*\*2 + (point1[1] - point2[1])\*\*2)

points = [(0, 0), (1, 3), (4, 1), (3, 4), (2, 2)]

all\_routes = itertools.permutations(range(len(points)))

best\_route = min(all\_routes, key=lambda route: sum(distance(points[route[i]], points[route[i+1]]) for i in range(len(points) - 1)))

total\_distance = sum(distance(points[best\_route[i]], points[best\_route[i+1]]) for i in range(len(points) - 1))

total\_distance += distance(points[best\_route[-1]], points[best\_route[0]])

print("Best Route:", best\_route)

print("Minimum Distance:", total\_distance)

**OUTPUT:**

Best Route: (0, 4, 1, 3, 2)

Minimum Distance: 13.764091950405117

**ID3 ALGORITHM**

**AIM:** To write a program to demonstrate the working of the decision tree based ID3 algorithm by using an appropriate data set for building the decision tree and use it to classify a new sample.

**DESCRIPTION:**

The ID3 algorithm is a popular decision tree algorithm used in machine learning. It aims to build a decision tree by iteratively selecting the best attribute to split the data based on information gain. Each node represents a test on an attribute, and each branch represents a possible outcome of the test. The leaf nodes of the tree represent the final classifications. In this article, we will learn how to use the ID3 algorithm to build a decision tree to predict the output in detail.

A decision tree is a flowchart-like representation, with internal nodes representing features, branches representing rules, and leaf nodes representing algorithm results This versatile supervised machine-learning algorithm applies to both classification and regression problems, ie and power. Decision trees are valued for their interpretability, as the rules they generate are easy to understand.

**PROGRAM:**

import numpy as np

# Define the ID3 algorithm

def id3(X, y, attribute\_names):

if len(np.unique(y)) <= 1:

return np.unique(y)[0]

elif len(attribute\_names) == 0:

return np.unique(y)[np.argmax(np.unique(y, return\_counts=True)[1])]

else:

best\_attribute\_index = np.argmax([information\_gain(X, y, i) for i in range(len(attribute\_names))])

tree = {attribute\_names[best\_attribute\_index]: {}}

remaining\_attribute\_names = [i for i in attribute\_names if i != attribute\_names[best\_attribute\_index]]

for value in np.unique(X[:, best\_attribute\_index]):

value = value

sub\_X = X[X[:, best\_attribute\_index] == value]

sub\_y = y[X[:, best\_attribute\_index] == value]

subtree = id3(sub\_X, sub\_y, remaining\_attribute\_names)

tree[attribute\_names[best\_attribute\_index]][value] = subtree

return tree

# Define the information gain function

def information\_gain(X, y, split\_attribute\_index):

entropy\_before = entropy(y)

values, counts = np.unique(X[:, split\_attribute\_index], return\_counts=True)

weighted\_entropy\_after = np.sum([(counts[i] / np.sum(counts)) \* entropy(y[X[:, split\_attribute\_index] == values[i]]) for i in range(len(values))])

information\_gain = entropy\_before - weighted\_entropy\_after

return information\_gain

# Define the entropy function

def entropy(y):

unique\_values, counts = np.unique(y, return\_counts=True)

probabilities = counts / np.sum(counts)

entropy = -np.sum(probabilities \* np.log2(probabilities))

return entropy

# Test dataset

X = np.array([[1, 'Sunny'], [1, 'Sunny'], [0, 'Overcast'], [0, 'Rain'], [0, 'Rain']])

y = np.array(['No', 'No', 'Yes', 'Yes', 'Yes'])

attribute\_names = ['Temperature', 'Outlook']

# Build the decision tree

tree = id3(X, y, attribute\_names)

print(tree)

**OUTPUT:**

{'Temperature': {'0': 'Yes', '1': 'No'}}

**BACK PROPOGATION ALGORITHM**

**AIM**: To Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

**DESCRIPTION:**

A neural network is a network structure, by the presence of computing units(neurons) the neural network has gained the ability to compute the function.

In machine learning, backpropagation is an effective algorithm used to train artificial neural networks, especially in feed-forward neural networks.

Backpropagation is an iterative algorithm, that helps to minimize the cost function by determining which weights and biases should be adjusted. During every epoch, the model learns by adapting the weights and biases to minimize the loss by moving down toward the gradient of the error. Thus, it involves the two most popular optimization algorithms, such as gradient descent or stochastic gradient descent.

**PROGRAM:**

import tensorflow as tf

import numpy as np

input\_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

output\_data = np.array([[0], [1], [1], [0]])

model = tf.keras.Sequential([

tf.keras.layers.Dense(units=2, activation='sigmoid', input\_shape=(2,)),

tf.keras.layers.Dense(units=1, activation='sigmoid')])

model.compile(optimizer='sgd', loss='binary\_crossentropy', metrics=['accuracy'])

model.fit(input\_data, output\_data, epochs=5000, verbose=0)

test\_input = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

predictions = model.predict(test\_input)

for i in range(len(predictions)):

print(f"Input: {test\_input[i]}, Predicted Output: {predictions[i]}")

**OUTPUT:**

Input: [0 0], Predicted Output: [0.48396885]

Input: [0 1], Predicted Output: [0.5038496]

Input: [1 0], Predicted Output: [0.50018275]

Input: [1 1], Predicted Output: [0.5167273]

**CANDIDATE ELIMINATION ALGORITHM**

**AIM:** To implement candidate elimination algorithm to find most specific hypothesis and most general hypothesis.

**DESCRIPTION:**

The candidate elimination algorithm incrementally builds the version space given a hypothesis space H and a set E of examples. The examples are added one by one; each example possibly shrinks the version space by removing the hypotheses that are inconsistent with the example. The candidate elimination algorithm does this by updating the general and specific boundary for each new example.

* You can consider this as an extended form of the Find-S algorithm.
* Consider both positive and negative examples.
* Actually, positive examples are used here as the Find-S algorithm (Basically they are generalizing from the specification).
* While the negative example is specified in the generalizing form.

**PROGRAM:**

import csv

with open("/content/drive/MyDrive/trainingdata.csv") as f:

csv\_file=csv.reader(f)

data=list(csv\_file)

s=data[1][:-1]

g=[['?' for i in range(len(s))] for j in range(len(s))]

for i in data:

if i[-1]=="Yes":

for j in range(len(s)):

if i[j]!=s[j]:

s[j]='?'

g[j][j]='?'

elif i[-1]=="No":

for j in range(len(s)):

if i[j]!=s[j]:

g[j][j]=s[j]

else:

g[j][j]="?"

print("\nStep-",data.index(i)+1)

print(s)

print(g)

gh=[]

for i in g:

for j in i:

if j!='?':

gh.append(i)

break

print("\nFinal specific hypothesis:\n",s)

print("\nFinal general hypothesis:\n",gh)

**OUTPUT:**

Step- 1

['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same']

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Step- 2

['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same']

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Step- 3

['Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same']

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Step- 4

['Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same']

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]

Step- 5

['Sunny', 'Warm', '?', 'Strong', '?', '?']

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final specific hypothesis:

['Sunny', 'Warm', '?', 'Strong', '?', '?']

Final general hypothesis:

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]

**DATA SET:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **sky** | **airTemp** | **humidity** | **wind** | **water** | **forecast** | **enjoySport** |
| Sunny | Warm | Normal | Strong | Warm | Same | Yes |
| Sunny | Warm | High | Strong | Warm | Same | Yes |
| Rainy | Cold | High | Strong | Warm | Change | No |
| Sunny | Warm | High | Strong | Cool | Change | Yes |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

**NAVIVE BAYESIAN CLASSIFIER**

**AIM:** To write a program to implement the naive Bayesian classifier for a sample training data set stored as a .CSV file. And compute the accuracy of the classifier by considering few test data sets also calculate the accuracy, precision, and recall for your data set.

**DESCRIPTION:**

Naïve Bayes algorithm is a supervised learning algorithm, which is based on **Bayes theorem** and used for solving classification problems. It is mainly used in *text classification* that includes a high-dimensional training dataset. Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions. It is a probabilistic classifier, which means it predicts on the basis of the probability of an object. Some popular examples of Naïve Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.

**PROGRAM:**

import pandas as pd

from sklearn import tree

from sklearn.preprocessing import LabelEncoder

from sklearn.naive\_bayes import GaussianNB

data = pd.read\_csv('/content/drive/MyDrive/tennisdata.csv')

print("The first 5 values of data is :\n",data.head())

X = data.iloc[:,:-1]

print("\nThe First 5 values of train data is\n",X.head())

y = data.iloc[:,-1]

print("\nThe first 5 values of Train output is\n",y.head())

le\_outlook = LabelEncoder()

X.Outlook = le\_outlook.fit\_transform(X.Outlook)

le\_Temperature = LabelEncoder()

X.Temperature = le\_Temperature.fit\_transform(X.Temperature)

le\_Humidity = LabelEncoder()

X.Humidity = le\_Humidity.fit\_transform(X.Humidity)

le\_Windy = LabelEncoder()

X.Windy = le\_Windy.fit\_transform(X.Windy)

print("\nNow the Train data is :\n",X.head())

le\_PlayTennis = LabelEncoder()

y = le\_PlayTennis.fit\_transform(y)

print("\nNow the Train output is\n",y)

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, test\_size=0.20)

classifier = GaussianNB()

classifier.fit(X\_train,y\_train)

from sklearn.metrics import accuracy\_scorek

print("Accuracy is:",accuracy\_score(classifier.predict(X\_test),y\_test))

**OUTPUT:**

The first 5 values of data is :

Outlook Temperature Humidity Windy PlayTennis

0 Sunny Hot High False No

1 Sunny Hot High True No

2 Overcast Hot High False Yes

3 Rainy Mild High False Yes

4 Rainy Cool Normal False Yes

The First 5 values of train data is

Outlook Temperature Humidity Windy

0 Sunny Hot High False

1 Sunny Hot High True

2 Overcast Hot High False

3 Rainy Mild High False

4 Rainy Cool Normal False

The first 5 values of Train output is

0 No

1 No

2 Yes

3 Yes

4 Yes

Name: PlayTennis, dtype: object

Now the Train data is :

Outlook Temperature Humidity Windy

0 2 1 0 0

1 2 1 0 1

2 0 1 0 0

3 1 2 0 0

4 1 0 1 0

Now the Train output is

[0 0 1 1 1 0 1 0 1 1 1 1 1 0]

Accuracy is: 0.3333333333333333

**DATA SET:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Outlook** | **Temperature** | **Humidity** | **Windy** | **PlayTennis** |
| Sunny | Hot | High | FALSE | No |
| Sunny | Hot | High | TRUE | No |
| Overcast | Hot | High | FALSE | Yes |
| Rainy | Mild | High | FALSE | Yes |
| Rainy | Cool | Normal | FALSE | Yes |
| Rainy | Cool | Normal | TRUE | No |
| Overcast | Cool | Normal | TRUE | Yes |
| Sunny | Mild | High | FALSE | No |
| Sunny | Cool | Normal | FALSE | Yes |
| Rainy | Mild | Normal | FALSE | Yes |
| Sunny | Mild | Normal | TRUE | Yes |
| Overcast | Mild | High | TRUE | Yes |
| Overcast | Hot | Normal | FALSE | Yes |
| Rainy | Mild | High | TRUE | No |

**BAYESIAN NETWORK**

**AIM:** To write a program to construct a Bayesian network considering medical data and Use this model to demonstrate the diagnosis of heart patients using any standard Heart Disease Data Set.

**DESCRIPTION:**

"A Bayesian network is a probabilistic graphical model which represents a set of variables and their conditional dependencies using a directed acyclic graph."

It is also called a **Bayes network, belief network, decision network**, or **Bayesian model**.

Bayesian networks are probabilistic, because these networks are built from a **probability distribution**, and also use probability theory for prediction and anomaly detection.

Real world applications are probabilistic in nature, and to represent the relationship between multiple events, we need a Bayesian network. It can also be used in various tasks including **prediction, anomaly detection, diagnostics, automated insight, reasoning, time series prediction**, and **decision making under uncertainty**.

**PROGRAM:**

import pandas as pd

data=pd.read\_csv("/content/drive/MyDrive/heartdisease.csv")

heart\_disease=pd.DataFrame(data)

print(heart\_disease)

from pgmpy.models import BayesianNetwork

model=BayesianNetwork([

('age','Lifestyle'),

('Gender','Lifestyle'),

('Family','heartdisease'),

('diet','cholestrol'),

('Lifestyle','diet'),

('cholestrol','heartdisease'),

('diet','cholestrol')

])

from pgmpy.estimators import MaximumLikelihoodEstimator

model.fit(heart\_disease, estimator=MaximumLikelihoodEstimator)

from pgmpy.inference import VariableElimination

HeartDisease\_infer = VariableElimination(model)

print('For age Enter { SuperSeniorCitizen:0, SeniorCitizen:1, MiddleAged:2, Youth:3, Teen:4 }')

print('For Gender Enter { Male:0, Female:1 }')

print('For Family History Enter { yes:1, No:0 }')

print('For diet Enter { High:0, Medium:1 }')

print('For lifeStyle Enter { Athlete:0, Active:1, Moderate:2, Sedentary:3 }')

print('For cholesterol Enter { High:0, BorderLine:1, Normal:2 }')

q = HeartDisease\_infer.query(variables=['heartdisease'], evidence={

'age':int(input('Enter age :')),

'Gender':int(input('Enter Gender :')),

'Family':int(input('Enter Family history :')),

'diet':int(input('Enter diet :')),

'Lifestyle':int(input('Enter Lifestyle :')),

'cholestrol':int(input('Enter cholestrol :'))

})

print(q['heartdisease'])

OUTPUT:

age Gender Family diet Lifestyle cholestrol heartdisease

0 0 0 1 1 3 0 1

1 0 1 1 1 3 0 1

2 1 0 0 0 2 1 1

3 4 0 1 1 3 2 0

4 3 1 1 0 0 2 0

5 2 0 1 1 1 0 1

6 4 0 1 0 2 0 1

7 0 0 1 1 3 0 1

8 3 1 1 0 0 2 0

9 1 1 0 0 0 2 1

10 4 1 0 1 2 0 1

11 4 0 1 1 3 2 0

12 2 1 0 0 0 0 0

13 2 0 1 1 1 0 1

14 3 1 1 0 0 1 0

15 0 0 1 0 0 2 1

16 1 1 0 1 2 1 1

17 3 1 1 1 0 1 0

18 4 0 1 1 3 2 0

For age Enter { SuperSeniorCitizen:0, SeniorCitizen:1, MiddleAged:2, Youth:3, Teen:4 }

For Gender Enter { Male:0, Female:1 }

For Family History Enter { yes:1, No:0 }

For diet Enter { High:0, Medium:1 }

For lifeStyle Enter { Athlete:0, Active:1, Moderate:2, Sedentary:3 }

For cholesterol Enter { High:0, BorderLine:1, Normal:2 }

Enter age :1

Enter Gender :1

Enter Family history :0

Enter diet :1

Enter Lifestyle :0

Enter cholestrol :1

+----------------+---------------------+

| heartdisease | phi(heartdisease) |

+==========+===========+

| heartdisease\_0 | 0.0000 |

+----------------+---------------------+

| heartdisease\_1 | 1.0000 |

+----------------+---------------------+

**EM ALGORIHTM AND K-MEANS ALGORITHM**

**AIM**: To Apply EM algorithm to cluster a set of data stored in a .CSV file.and Use the same data set for clustering using k-Means algorithm also Compare the results of these two algorithms and comment on the quality of clustering.

**DESCRIPTION:**

The Expectation-Maximization (EM) algorithm is an iterative optimization method that combines different unsupervised machine learning algorithms to find maximum likelihood or maximum posterior estimates of parameters in statistical models that involve unobserved latent variables. The EM algorithm is commonly used for latent variable models and can handle missing data. It consists of an estimation step (E-step) and a maximization step (M-step), forming an iterative process to improve model fit.

* In the E step, the algorithm computes the latent variables i.e. expectation of the log-likelihood using the current parameter estimates.
* In the M step, the algorithm determines the parameters that maximize the expected log-likelihood obtained in the E step, and corresponding model parameters are updated based on the estimated latent variables.

**PROGRAM:**

from sklearn.cluster import KMeans

from sklearn import preprocessing

from sklearn.mixture import GaussianMixture

from sklearn.datasets import load\_iris

import sklearn.metrics as sm

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

dataset=load\_iris()

# print(dataset)

X=pd.DataFrame(dataset.data)

X.columns=['Sepal\_Length','Sepal\_Width','Petal\_Length','Petal\_Width']

y=pd.DataFrame(dataset.target)

y.columns=['Targets']

# print(X)

plt.figure(figsize=(14,7))

colormap=np.array(['red','lime','black'])

# REAL PLOT

plt.subplot(1,3,1)

plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[y.Targets],s=40)

plt.title('Real')

# K-PLOT

plt.subplot(1,3,2)

model=KMeans(n\_clusters=3)

model.fit(X)

predY=np.choose(model.labels\_,[0,1,2]).astype(np.int64)

plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[predY],s=40)

plt.title('KMeans')

# GMM PLOT

scaler=preprocessing.StandardScaler()

scaler.fit(X)

xsa=scaler.transform(X)

xs=pd.DataFrame(xsa,columns=X.columns)

gmm=GaussianMixture(n\_components=3)

gmm.fit(xs)

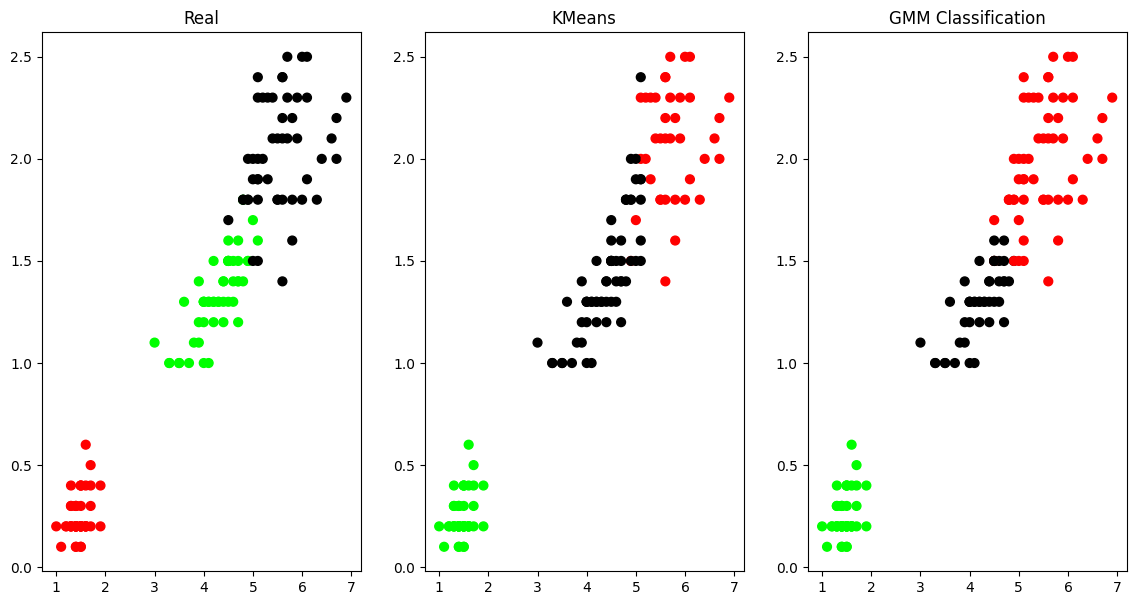
y\_cluster\_gmm=gmm.predict(xs)

plt.subplot(1,3,3)

plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[y\_cluster\_gmm],s=40)

plt.title('GMM Classification')

**OUTPUT:**



**KNN ALGORITHM**

**AIM:** To write a program to implement k-Nearest Neighbor algorithm to classify the iris data set and Print both correct and wrong predictions.

**DESCRIPTION:**

* K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
* K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
* K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.
* K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.
* K-NN is a **non-parametric algorithm**, which means it does not make any assumption on underlying data.
* It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.
* KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.

**PROGRAM:**

from sklearn.datasets import load\_iris

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model\_selection import train\_test\_split

import numpy as np

dataset=load\_iris()

#print(dataset)

X\_train,X\_test,y\_train,y\_test=train\_test\_split(dataset["data"],dataset["target"],random\_state=0)

kn=KNeighborsClassifier(n\_neighbors=1)

kn.fit(X\_train,y\_train)

for i in range(len(X\_test)):

x=X\_test[i]

x\_new=np.array([x])

prediction=kn.predict(x\_new)

print("TARGET=",y\_test[i],dataset["target\_names"][y\_test[i]],"PREDICTED=",prediction,dataset["target\_names"][prediction])

print(kn.score(X\_test,y\_test))

**OUTPUT:**

TARGET= 2 virginica PREDICTED= [2] ['virginica']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 2 virginica PREDICTED= [2] ['virginica']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 2 virginica PREDICTED= [2] ['virginica']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 2 virginica PREDICTED= [2] ['virginica']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 2 virginica PREDICTED= [2] ['virginica']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 2 virginica PREDICTED= [2] ['virginica']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 2 virginica PREDICTED= [2] ['virginica']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 2 virginica PREDICTED= [2] ['virginica']

TARGET= 2 virginica PREDICTED= [2] ['virginica']

TARGET= 1 versicolor PREDICTED= [1] ['versicolor']

TARGET= 0 setosa PREDICTED= [0] ['setosa']

TARGET= 1 versicolor PREDICTED= [2] ['virginica']

0.9736842105263158

**NON-PARAMETRIC LOCALLY WEIGHTED ALGORITHM**

**AIM:** To Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. And Select appropriate data set for your experiment and draw the corresponding graphs.

**DESCRIPTION:**

Linear regression is a supervised learning algorithm used for computing linear relationships between input (X) and output (Y)

I Locally weighted linear regression is a non-parametric algorithm, that is, the model does not learn a fixed set of parameters as is done in ordinary linear regression. Rather parameters are computed individually for each query point . While computing , a higher “preference” is given to the points in the training set lying in the vicinity of than the points lying far away from.

**PROGRAM:**

#non-parametric locally weighted algo

import numpy as np

import matplotlib.pyplot as plt

def kernel\_function(x, x\_i, tau):

# Gaussian kernel function

return np.exp(-((x - x\_i) \*\* 2) / (2 \* tau \*\* 2))

def locally\_weighted\_regression(x, X, y, tau):

# Add bias term to X

X\_bias = np.column\_stack([np.ones(len(X)), X])

# Initialize predictions

predictions = []

# Iterate over each query point

for query\_point in x:

weights = kernel\_function(query\_point, X, tau)

# Weighted least squares regression

W = np.diag(weights)

theta = np.linalg.inv(X\_bias.T @ W @ X\_bias) @ (X\_bias.T @ W @ y)

# Predict using the learned parameters

prediction = np.dot(np.array([1, query\_point]), theta)

predictions.append(prediction)

return np.array(predictions)

# Generate synthetic dataset

np.random.seed(42)

X = np.linspace(0, 10, 100)

y = np.sin(X) + np.random.normal(0, 0.1, size=X.shape)

# Set bandwidth parameter

tau = 0.1

# Predict using locally weighted regression

x\_values = np.linspace(0, 10, 1000)

y\_values = locally\_weighted\_regression(x\_values, X, y, tau)

# Plot original data and predictions

plt.figure(figsize=(10, 6))

plt.scatter(X, y, color='blue', label='Original Data')

plt.plot(x\_values, y\_values, color='red', label='Locally Weighted Regression')

plt.title('Locally Weighted Regression')

plt.xlabel('X')

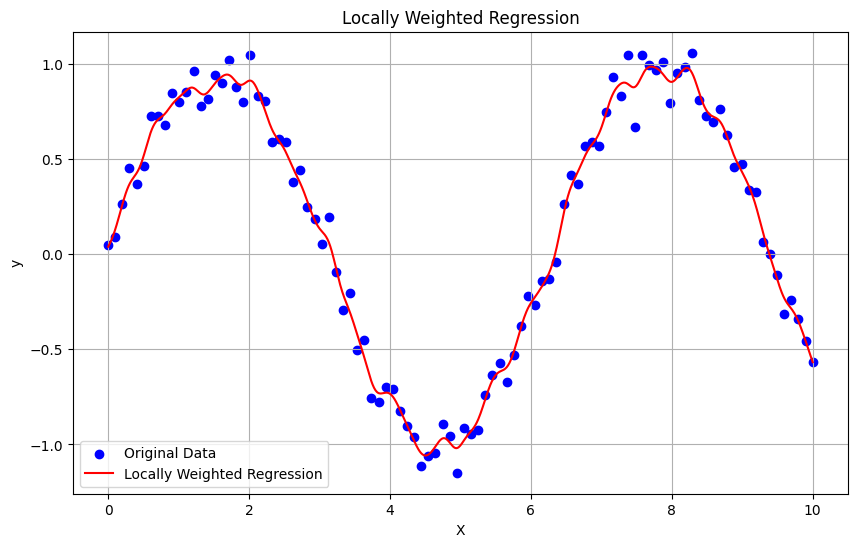
plt.ylabel('y')

plt.legend()

plt.grid(True)

plt.show()

**OUTPUT:**



**K-FOLD CROSS VALIDATION**

**AIM:** To Write a program to implement 5-fold cross validation on a given dataset and Compare the accuracy, precision, recall, and F-score for your data set for different folds.

**DESCRIPTION:**

Cross-validation is a technique for validating the model efficiency by training it on the subset of input data and testing on previously unseen subset of the input data. **We can also say that it is a technique to check how a statistical model generalizes to an independent dataset***.*

In machine learning, there is always the need to test the stability of the model. It means based only on the training dataset; we can't fit our model on the training dataset. For this purpose, we reserve a particular sample of the dataset, which was not part of the training dataset. After that, we test our model on that sample before deployment, and this complete process comes under cross-validation. This is something different from the general train-test split.

Hence the basic steps of cross-validations are:

* Reserve a subset of the dataset as a validation set.
* Provide the training to the model using the training dataset.
* Now, evaluate model performance using the validation set. If the model performs well with the validation set, perform the further step, else check for the issues.

**PROGRAM:**

from sklearn.model\_selection import cross\_val\_score, KFold

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import load\_iris

# Load example dataset (replace with your dataset)

data = load\_iris()

X, y = data.data, data.target

# Define the classifier (replace with your classifier)

clf = DecisionTreeClassifier()

# Define evaluation metrics

scoring = {

'accuracy': 'accuracy',

'precision': 'precision\_macro',

'recall': 'recall\_macro',

'f1': 'f1\_macro'

}

# Perform 5-fold cross-validation

kf = KFold(n\_splits=5, shuffle=True, random\_state=42)

for fold, (train\_index, test\_index) in enumerate(kf.split(X)):

X\_train, X\_test = X[train\_index], X[test\_index]

y\_train, y\_test = y[train\_index], y[test\_index]

clf.fit(X\_train, y\_train)

y\_pred = clf.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred, average='macro')

recall = recall\_score(y\_test, y\_pred, average='macro')

f1 = f1\_score(y\_test, y\_pred, average='macro')

print(f"Fold {fold+1}:")

print(f"Accuracy: {accuracy}")

print(f"Precision: {precision}")

print(f"Recall: {recall}")

print(f"F1 Score: {f1}")

print()

**OUTPUT:**

Fold 1:

Accuracy: 1.0

Precision: 1.0

Recall: 1.0

F1 Score: 1.0

Fold 2:

Accuracy: 1.0

Precision: 1.0

Recall: 1.0

F1 Score: 1.0

Fold 3:

Accuracy: 0.9333333333333333

Precision: 0.9333333333333332

Recall: 0.9333333333333332

F1 Score: 0.9259259259259259

Fold 4:

Accuracy: 0.9333333333333333

Precision: 0.9388888888888888

Recall: 0.9388888888888888

F1 Score: 0.9388888888888888

Fold 5:

Accuracy: 0.9333333333333333

Precision: 0.9487179487179488

Recall: 0.9444444444444445

F1 Score: 0.9419191919191919

**REINFORCEMENT LEARNING**

**AIM:** To write a simple program to implement Reinforcement Learning.

**DESCRIPTION:**

Reinforcement learning is an area of Machine Learning. It is about taking suitable action to maximize reward in a particular situation. It is employed by various software and machines to find the best possible behavior or path it should take in a specific situation. Reinforcement learning differs from supervised learning in a way that in supervised learning the training data has the answer key with it so the model is trained with the correct answer itself whereas in reinforcement learning, there is no answer but the reinforcement agent decides what to do to perform the given task. In the absence of a training dataset, it is bound to learn from its experience.

Reinforcement Learning (RL) is the science of decision making. It is about learning the optimal behavior in an environment to obtain maximum reward. In RL, the data is accumulated from machine learning systems that use a trial-and-error method. Data is not part of the input that we would find in supervised or unsupervised machine learning.

**PROGRAM:**

import numpy as np

# Define the environment

num\_states = 16 # 4x4 grid world

num\_actions = 4 # Up, Down, Left, Right

# Define rewards

rewards = np.array([

[-1, -1, -1, -1],

[-1, -1, -1, -1],

[-1, -1, -1, -1],

[-1, -1, -1, 10] # Goal

])

# Initialize Q-table

q\_table = np.zeros((num\_states, num\_actions))

# Q-learning parameters

learning\_rate = 0.1

discount\_factor = 0.9

epsilon = 0.1

num\_episodes = 1000

# Q-learning algorithm

for \_ in range(num\_episodes):

state = np.random.randint(num\_states) # Start from a random state

done = False

while not done:

if np.random.uniform(0, 1) < epsilon:

action = np.random.randint(num\_actions) # Explore

else:

action = np.argmax(q\_table[state, :]) # Exploit

next\_state = (state + 1) % num\_states # Transition to the next state

reward = rewards[state // 4, state % 4] # Get reward for current state

td\_target = reward + discount\_factor \* np.max(q\_table[next\_state, :])

td\_error = td\_target - q\_table[state, action]

q\_table[state, action] += learning\_rate \* td\_error

state = next\_state

if state == num\_states - 1:

done = True # Reach the goal

# Print the Q-table

print("Q-table:")

print(q\_table)

OUTPUT:

Q-table:

[[-4.70944673 -4.76643065 -4.78622887 -4.7412093 ]

[-5.38689152 -5.35088152 -5.36363495 -5.37319349]

[-5.30978315 -5.31359132 -5.3179846 -5.30656095]

[-4.95139646 -4.94662529 -4.94882623 -4.94459739]

[-4.4762014 -4.48068155 -4.48242096 -4.47825284]

[-3.95065999 -3.94631332 -3.94786459 -3.94947487]

[-3.33869477 -3.33797717 -3.33732121 -3.33939604]

[-2.67437946 -2.62940296 -2.63270142 -2.63035518]

[-1.95777588 -1.84064479 -1.8395852 -2.00439687]

[-1.40798553 -0.95838963 -1.29260792 -0.95708788]

[-0.14590318 0.01577939 -0.07200648 0.01880956]

[ 1.10997082 1.11120289 1.10726117 1.10875629]

[ 2.27516163 2.27281016 2.26890919 2.25501627]

[ 3.26706266 3.54087881 3.53141785 3.53719008]

[ 4.8910985 4.9013738 4.89073082 4.8934051 ]

[ 6.36468652 2.35385258 0. 0.90077054]]