1. **Visualize the n-dimensional data using 3D surface plots.**
Write a program to implement the Best First Search (BFS) algorithm

(a)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

X, Y = np.meshgrid(np.linspace(0, 1, 100), np.linspace(0, 1, 100))
Z = np.sin(X * np.pi) * np.cos(Y * np.pi)
ax = plt.subplot(2, 3, 5, projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, edgecolor='none')
plt.colorbar(surf)

plt.tight_layout()
plt.show()
```

(b)

```
def best_first_search(graph,start,goal,heuristic, path=[]):
    open_list = [(0,start)]
    closed_list = set()
    closed_list.add(start)

    while open_list:
        open_list.sort(key = lambda x: heuristic[x[1]], reverse=True)
        cost, node = open_list.pop()
        path.append(node)

        if node==goal:
            return cost, path

        closed_list.add(node)
        for neighbour, neighbour_cost in graph[node]:
            if neighbour not in closed_list:
                closed_list.add(node)
                open_list.append((cost+neighbour_cost, neighbour))

    return None


graph = {
    'A': [('B', 11), ('C', 14), ('D',7)],
    'B': [('A', 11), ('E', 15)],
    'C': [('A', 14), ('E', 8), ('D',18), ('F',10)],
    'D': [('A', 7), ('F', 25), ('C',18)],
    'E': [('B', 15), ('C', 8), ('H',9)],
    'F': [('G', 20), ('C', 10), ('D',25)],
    'G': [],
    'H': [('E',9), ('G',10)]
}

start = 'A'
goal = 'G'

heuristic = {
    'A': 40,
    'B': 32,
    'C': 25,
    'D': 35,
    'E': 19,
    'F': 17,
    'G': 0,
    'H': 10
}

result = best_first_search(graph, start, goal, heuristic)

if result:
    print(f"Minimum cost path from {start} to {goal} is {result[1]}")
```

```
    print(f"Cost: {result[0]}")
else:
    print(f"No path from {start} to {goal}")
```

2. **Visualize the n-dimensional data using contour plots.**
Write a program to implement the A* algorithm

(a)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

X, Y = np.meshgrid(np.linspace(0, 1, 100), np.linspace(0, 1, 100))
Z = np.sin(X * np.pi) * np.cos(Y * np.pi)
contour = plt.contour(X, Y, Z, 20, cmap='viridis')
plt.colorbar(contour)

plt.tight_layout()
plt.show()
```

(b)

```
def h(n):
    H = {'A': 3, 'B': 4, 'C': 2, 'D': 6, 'G': 0, 'S': 5}
    return H[n]

def a_star_algorithm(graph, start, goal):
    open_list = [start]
    closed_list = set()
    g = {start:0}
    parents = {start:start}
    while open_list:
        open_list.sort(key=lambda v: g[v] + h(v), reverse=True)
        n = open_list.pop()
        if n == goal:
            reconst_path = []
            while parents[n] != n:
                reconst_path.append(n)
                n = parents[n]
            reconst_path.append(start)
            reconst_path.reverse()
            print(f'Path found: {reconst_path}')
            return reconst_path
        for (m, weight) in graph[n]:
            if m not in open_list and m not in closed_list:
                open_list.append(m)
                parents[m] = n
                g[m] = g[n] + weight
            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n
                    if m in closed_list:
                        closed_list.remove(m)
                        open_list.append(m)
        closed_list.add(n)
    print('Path does not exist!')
    return None


graph = {
    'S': [('A', 1), ('G', 10)],
    'A': [('B', 2), ('C', 1)],
    'B': [('D', 5)],
    'C': [('D', 3),('G', 4)],
    'D': [('G', 2)]
}
a_star_algorithm(graph, 'S', 'G')
```

3. **Visualize the n-dimensional data using heat-map.**

Write a program to implement Min-Max algorithm.

(a)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

data = np.random.rand(100, 3)
df = pd.DataFrame(data, columns=['Feature 1', 'Feature 2', 'Feature 3'])

plt.figure(figsize=(12, 8))
corr = df.corr()
sns.heatmap(corr, annot=True, cmap='coolwarm', vmin=-1, vmax=1)

plt.tight_layout()
plt.show()
```

(b)

```
class TreeNode:
  def __init__(self,value,children=[]):
    self.value = value
    self.children = children

def minimax(node, depth, maximizing_player):
    if depth == 0 or not node.children:
        return node.value, [node.value]

    if maximizing_player:
        max_value = float("-inf")
        max_path = []
        for child_node in node.children:
            child_value, child_path = minimax(child_node, depth - 1, False)
            if child_value > max_value:
                max_value = child_value
                max_path = [node.value] + child_path
        return max_value, max_path
    else:
        min_value = float("inf")
        min_path = []
        for child_node in node.children:
            child_value, child_path = minimax(child_node, depth - 1, True)
            if child_value < min_value:
                min_value = child_value
                min_path = [node.value] + child_path
        return min_value, min_path


game_tree = TreeNode(0, [
    TreeNode(1, [TreeNode(3),TreeNode(12)]),
    TreeNode(4, [TreeNode(8),TreeNode(2)])
])


optimal_value, optimal_path = minimax(game_tree, 2, True)

print("Optimal value:", optimal_value)
print("Optimal path:", optimal_path)
```

4. **Visualize the n-dimensional data using Box-plot.**
Write a program to implement Alpha-beta pruning algorithm.

(a)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

data = np.random.rand(100, 3)
df = pd.DataFrame(data, columns=['Feature 1', 'Feature 2', 'Feature 3'])
```

```
plt.figure(figsize=(12, 8))
sns.boxplot(data=df)

plt.tight_layout()
plt.show()

(b)
MAX, MIN = 1000, -1000

def alphabeta_minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):

      if depth == 3:
            return values[nodeIndex]

      if maximizingPlayer:
            best = MIN
            for i in range(0, 2):
                  val = alphabeta_minimax(depth + 1, nodeIndex * 2 + i, False,
values, alpha, beta)
                  best = max(best, val)
                  alpha = max(alpha, best)

                  if beta <= alpha:
                        break
            return best

      else:
            best = MAX
            for i in range(0, 2):
                  val = alphabeta_minimax(depth + 1, nodeIndex * 2 + i, True,
values, alpha, beta)
                  best = min(best, val)
                  beta = min(beta, best)

                  if beta <= alpha:
                        break

            return best


values = [3, 5, 6, 9, 1, 2, 0, -1]
print("The optimal value is :", alphabeta_minimax(0, 0, True, values, MIN, MAX))
```

5. **Write a program to develop the Naive Bayes classifier on Titanic dataset.**

```
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.model_selection import train_test_split, LeaveOneOut, cross_val_score
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

# Load Titanic dataset
df = sns.load_dataset('titanic')

# Preprocessing
df.drop(columns=['deck', 'embark_town', 'alive', 'who', 'class', 'adult_male',
'alone'], inplace=True)
df.dropna(inplace=True)
df['sex'] = df['sex'].map({'male': 0, 'female': 1})
df['embarked'] = df['embarked'].map({'C': 0, 'Q': 1, 'S': 2})
df['embarked'].fillna(df['embarked'].mode()[0], inplace=True)

# Features and target variable
X = df.drop(columns=['survived'])
y = df['survived']

# Function to train and evaluate Naive Bayes classifier
def naive_bayes(X, y):
```

```
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7,
random_state=42)
    model = GaussianNB()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy = {accuracy:.4f}")

naive_bayes(X,y)

# Leave-One-Out Cross-Validation
loo = LeaveOneOut()
model = GaussianNB()
accuracies = cross_val_score(model, X, y, cv=loo)
print(f"\nLeave-One-Out Cross-Validation: Accuracy = {np.mean(accuracies):.4f}")
```

6. **Write a program to develop the KNN classifier with Euclidean distance and Manhattan distance for the k values as 3 based on split up of training and testing dataset as 70-30 on Glass dataset.**

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Load Glass dataset
df = pd.read_csv("glass.csv")

# Features and target variable
X = df.drop(columns=['Type'])
y = df['Type']

# Function to train and evaluate KNN classifier
def knn(X, y):
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7)
    metrics = ['euclidean','manhattan']
    for x in metrics:
       model = KNeighborsClassifier(n_neighbors=3,metric=x)
       model.fit(X_train, y_train)
       y_pred = model.predict(X_test)
       accuracy = accuracy_score(y_test, y_pred)
       print(f"metric={x}: Accuracy = {accuracy:.4f}")

knn(X,y)
```

7. **Write a program to develop a decision tree classifier based on weather forecasting dataset.**

```
from sklearn import preprocessing
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import pandas as pd
import matplotlib.pyplot as plt

# Create DataFrame
df = pd.read_csv('weather_forecast.csv')

# Convert categorical variables to numerical
le = preprocessing.LabelEncoder()
for column in df.columns:
    df[column] = le.fit_transform(df[column])

# Features and target variable
X = df.drop(columns=['Play'])
y = df['Play']

# Split dataset into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Decision tree classifier
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)

# Predictions on testing set
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Plot decision tree
plt.figure(figsize=(10, 6))
plot_tree(clf, filled=True, feature_names=df.columns[:-1], class_names=['No',
'Yes'])
plt.show()
```

8. **Write a program to perform unsupervised K-means clustering techniques**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans

# Load the Iris dataset
X = load_iris().data

# Number of clusters
K = 3

# Perform K-means clustering
kmeans = KMeans(n_clusters=K, random_state=0)
kmeans.fit(X)
labels = kmeans.labels_
centroids = kmeans.cluster_centers_

# Print the results
print("Labels:", labels)
print("Centroids:", centroids)

# Plot the results
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', color='red', s=200,
label='Centroids')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('K-means Clustering of Iris Dataset')
plt.legend()
plt.show()
```

9. **Write a program to perform agglomerative clustering based on single-linkage, complete-linkage criteria**

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.datasets import load_iris

iris = load_iris()
data = iris.data[:6]

def proximity_matrix(data):
    n = data.shape[0]
    proximity_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(i+1, n):
            proximity_matrix[i, j] = np.linalg.norm(data[i] - data[j])
```

```
            proximity_matrix[j, i] = proximity_matrix[i, j]
    return proximity_matrix

def plot_dendrogram(data, method):
    linkage_matrix = linkage(data, method=method)
    dendrogram(linkage_matrix)
    plt.title(f'Dendrogram - {method} linkage')
    plt.xlabel('Data Points')
    plt.ylabel('Distance')
    plt.show()

# Calculate the proximity matrix
print("Proximity matrix:")
print(proximity_matrix(data))

# Plot the dendrogram using single-linkage
plot_dendrogram(data, 'single')

# Plot the dendrogram using complete-linkage
plot_dendrogram(data, 'complete')
```

10. **Write a program to develop Principal Component Analysis(PCA) and Linear Discriminant Analysis(LDA) algorithms**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.preprocessing import StandardScaler

def plot_data(X_projected, y, xlabel, ylabel):
    plt.scatter(X_projected[:, 0], X_projected[:, 1], c=y, cmap="jet")
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.show()

# Load the Iris dataset
X, y = load_iris(return_X_y=True)

# Perform data preprocessing - Standardization
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Define the number of components
n_components = 2

# PCA
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X_scaled)
print(f"Shape of transformed data (PCA): {X_pca.shape}")
print(f"Transformed data: {X_pca}")
plot_data(X_pca, y, "PCA Component 1", "PCA Component 2")

# LDA
lda = LinearDiscriminantAnalysis(n_components=n_components)
X_lda = lda.fit_transform(X_scaled, y)
print(f"Shape of transformed data (LDA): {X_lda.shape}")
print(f"Transformed data: {X_lda}")
plot_data(X_lda, y, "LDA Component 1", "LDA Component 2")
```

11. **Write a Program to develop simple single layer perceptron to implement AND, OR Boolean functions.**

```
import numpy as np
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score

# Data for AND function
X_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
```

```
y_and = np.array([0, 0, 0, 1])  # A AND B

# Data for OR function
X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_or = np.array([0, 1, 1, 1])  # A OR B

# AND Perceptron
perceptron_and = Perceptron(max_iter=1000,random_state=0)
perceptron_and.fit(X_and, y_and)
y_pred_and = perceptron_and.predict(X_and)
accuracy_and = accuracy_score(y_and, y_pred_and)
print(f"AND Function Accuracy: {accuracy_and * 100}%")
print(f"AND Predictions: {y_pred_and}")

# OR Perceptron
perceptron_or = Perceptron(max_iter=1000, random_state=0)
perceptron_or.fit(X_or, y_or)
y_pred_or = perceptron_or.predict(X_or)
accuracy_or = accuracy_score(y_or, y_pred_or)
print(f"OR Function Accuracy: {accuracy_or * 100}%")
print(f"OR Predictions: {y_pred_or}")
```

12. **Write a program to develop Multi-layer perceptron to implement AND-NOT, XOR Boolean functions.**

```
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Data for AND-NOT function
X_and_not = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_and_not = np.array([0, 1, 1, 0])
# Data for XOR function
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([0, 1, 1, 0])

# AND-NOT MLP
mlp_and_not = MLPClassifier(hidden_layer_sizes=(3,), activation='relu',
solver='adam', max_iter=1000)
mlp_and_not.fit(X_and_not, y_and_not)
y_pred_and_not = mlp_and_not.predict(X_and_not)
accuracy_and_not = accuracy_score(y_and_not, y_pred_and_not)
print(f"AND-NOT Function Accuracy: {accuracy_and_not * 100}%")
print(f"AND-NOT Predictions: {y_pred_and_not}")

# XOR MLP
mlp_xor = MLPClassifier(hidden_layer_sizes=(3,), activation='relu', solver='adam',
max_iter=1000)
mlp_xor.fit(X_xor, y_xor)
y_pred_xor = mlp_xor.predict(X_xor)
accuracy_xor = accuracy_score(y_xor, y_pred_xor)
print(f"XOR Function Accuracy: {accuracy_xor * 100}%")
print(f"XOR Predictions: {y_pred_xor}")
```