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
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

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

plt.figure(figsize=(12, 8))
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
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

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

plt.figure(figsize=(12, 8))
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
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

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)

```
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
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

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 pandas as pd
import numpy as np
import seaborn as sns

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

# Preprocess the data
df = df.drop([ 'who', 'deck', 'embark_town', 'alive', 'class', 'adult_male',
'alone'], axis=1)
df['age'].fillna(df['age'].median(), inplace=True)
df['embarked'].fillna(df['embarked'].mode()[0], inplace=True)
df['sex'] = df['sex'].map({'female': 0, 'male': 1}).astype(int)
```

	```
	df['embarked'] = df['embarked'].map({'S': 0, 'C': 1, 'Q': 2}).astype(int)
	df = df.dropna()

	# Split the data into training and testing sets
	split = int(0.8 * len(df))
	train = df.iloc[:split]
	test = df.iloc[split:]

	# Calculate priors
	priors = train['survived'].value_counts(normalize=True).to_dict()

	# Calculate mean and standard deviation for conditionals
	conds = {}
	for feat in train.columns[train.columns != 'survived']:
	    conds[feat] = {}
	    for lbl in train['survived'].unique():
	        subset = train[train['survived'] == lbl][feat]
	        conds[feat][lbl] = (subset.mean(), subset.std())

	# Function to predict using Naive Bayes
	def predict(inst):
	    post = {lbl: np.log(priors[lbl]) for lbl in priors}
	    for feat, stats in conds.items():
	        x = inst[feat]
	        for lbl, (mean, std) in stats.items():
	            if std == 0:
	                std = 1e-6
	            exp = np.exp(-(x - mean) ** 2 / (2 * std ** 2))
	            like = (1 / (np.sqrt(2 * np.pi) * std)) * exp
	            post[lbl] += np.log(like)
	    return max(post, key=post.get)

	# Test the model
	y_true = []
	y_pred = []
	for _, inst in test.iterrows():
	    y_true.append(inst['survived'])
	    y_pred.append(predict(inst))

	# Calculate accuracy
	acc = np.sum(np.array(y_true) == np.array(y_pred)) / len(y_true)
	print("Accuracy:", acc)
	```

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 collections import Counter

	# Define distance functions
	def euc(x1, x2):
	    return np.sqrt(np.sum((x1 - x2)**2))

	def man(x1, x2):
	    return np.sum(np.abs(x1 - x2))

	df = pd.read_csv("glass.csv")
	X = df.drop("Type", axis=1).values
	y = df['Type'].values


	shf = np.random.permutation(len(X))
	split = int(0.7 * len(X))
	X_train, X_test = X[shf[:split]], X[shf[split:]]
	y_train, y_test = y[shf[:split]], y[shf[split:]]

	# KNN function
	def knn_predict(X_train, y_train, X_test,distance_fn):
	```

```
    pred = []
    for x in X_test:
      distances = [distance_fn(x, x_train) for x_train in X_train]
      k_indices = np.argsort(distances)[:3]
      k_labels = [y_train[i] for i in k_indices]
      most_common = Counter(k_labels).most_common(1)[0][0]
      pred.append(most_common)
    return pred


p1 = knn_predict(X_train, y_train, X_test, distance_fn=euc)
p2 = knn_predict(X_train, y_train, X_test, distance_fn=man)

def accuracy(y_true, y_pred):
    return np.sum(y_true == y_pred) / len(y_true)


acc = accuracy(y_test, p1)
print(f"Accuracy using Euclidean distance : {acc:.2f}")
acc = accuracy(y_test, p2)
print(f"Accuracy using manhattan distance : {acc:.2f}")
```

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
```

```
def kmeans(X, K, max_iters=100):
    centroids = X[:K]
    for _ in range(max_iters):
        expanded_x = X[:, np.newaxis]
        euc_dist = np.linalg.norm(expanded_x - centroids, axis=2)
        labels = np.argmin(euc_dist, axis=1)
        new_centroids = np.array([X[labels == k].mean(axis=0) for k in
range(K)])
        if np.all(centroids == new_centroids):
              break
        centroids = new_centroids
    return labels, centroids


X = load_iris() .data
K=3
labels, centroids = kmeans(X, K)
print("Labels:", labels)
print("Centroids:", centroids)

plt.scatter(X[:, 0], X[:, 1], c=labels)
plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', color='red', s=200)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('K-means Clustering of Iris Dataset')
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

class DimReduction:

    def __init__(self, method, n_components):
        self.method = method
        self.n_components = n_components
        self.projection = None

    def fit_transform(self, X, y=None):
        if self.method == 'PCA':
            mean = np.mean(X, axis=0)
            X_centered = X - mean
            cov = np.cov(X_centered.T)
            eigenvalues, eigenvectors = np.linalg.eig(cov)
            self.projection = eigenvectors[:, :self.n_components]
        elif self.method == 'LDA':
            class_labels = np.unique(y)
            mean_overall = np.mean(X, axis=0)
            SW = np.zeros((X.shape[1], X.shape[1]))
            SB = np.zeros((X.shape[1], X.shape[1]))
            for c in class_labels:
                X_c = X[y == c]
                mean_c = np.mean(X_c, axis=0)
                SW += (X_c - mean_c).T.dot((X_c - mean_c))
                n_c = X_c.shape[0]
                mean_diff = (mean_c - mean_overall).reshape(X.shape[1], 1)
                SB += n_c * (mean_diff).dot(mean_diff.T)
            A = np.linalg.inv(SW).dot(SB)
            eigenvalues, eigenvectors = np.linalg.eig(A)
            self.projection = eigenvectors[:, :self.n_components]
        return np.dot(X, self.projection)

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()

X, y = load_iris(return_X_y=True)

methods = ['PCA', 'LDA']
for method in methods:
    dr = DimReduction(method, 2)
    X_projected = dr.fit_transform(X, y)
    print(f"Shape of transformed data ({method}):", X_projected.shape)
    plot_data(X_projected, y, f"{method} Component 1", f"{method} Component 2")
```

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

```
import numpy as np

def train_perceptron(X, y, lr=0.1, epochs=100):
    X = np.c_[X, np.ones(len(X))]
    w = np.zeros(X.shape[1])

    for _ in range(epochs):
        for i in range(len(X)):
            pred = np.dot(X[i], w)
            err = y[i] - (1 if pred >= 0 else 0)
            w += lr * err * X[i]

    return w

def predict_perceptron(x, w):
    x = np.append(x, 1)
    pred = np.dot(x, w)
    return 1 if pred >= 0 else 0
```

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

print("Training AND Perceptron:")
w_and = train_perceptron(X_and, y_and)

# Test the AND perceptron
print("Testing AND Perceptron:")
for x in X_and:
    pred = predict_perceptron(x, w_and)
    print(f"Inputs: {x}, Prediction: {pred}")

# Example usage for OR function
X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_or = np.array([0, 1, 1, 1])

print("\nTraining OR Perceptron:")
w_or = train_perceptron(X_or, y_or)

# Test the OR perceptron
print("Testing OR Perceptron:")
for x in X_or:
    pred = predict_perceptron(x, w_or)
    print(f"Inputs: {x}, Prediction: {pred}")
```

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

```
import numpy as np

# Activation function - sigmoid
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)

# Define XOR inputs and labels
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([[0], [1], [1], [0]])

# Define AND-NOT inputs and labels
X_and_not = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_and_not = np.array([[1], [1], [0], [0]])

# Initialize weights and biases
np.random.seed(1)
input_size = X_xor.shape[1]
hidden_size = 4
output_size = 1
w_h = np.random.uniform(size=(input_size, hidden_size))
b_h = np.random.uniform(size=(1, hidden_size))
w_o = np.random.uniform(size=(hidden_size, output_size))
b_o = np.random.uniform(size=(1, output_size))

# Training function for MLP
def train_mlp(X, y, epochs=1000, lr=0.1):
    global w_h, b_h, w_o, b_o
    for epoch in range(epochs):
        # Forward propagation
        h_in = np.dot(X, w_h) + b_h
        h_out = sigmoid(h_in)
        o_in = np.dot(h_out, w_o) + b_o
        o_out = sigmoid(o_in)

        # Backpropagation
        error = y - o_out
        d_o = error * sigmoid_derivative(o_out)
        error_h = d_o.dot(w_o.T)
        d_h = error_h * sigmoid_derivative(h_out)

        # Update weights and biases
        w_o += h_out.T.dot(d_o) * lr
        b_o += np.sum(d_o, axis=0, keepdims=True) * lr
        w_h += X.T.dot(d_h) * lr
        b_h += np.sum(d_h, axis=0, keepdims=True) * lr

    return o_out

# Training XOR MLP
print("Training XOR MLP:")
out_xor = train_mlp(X_xor, y_xor)

# Training AND-NOT MLP
print("\nTraining AND-NOT MLP:")
out_and_not = train_mlp(X_and_not, y_and_not)

# Print final predictions
print("\nFinal predictions for XOR MLP:")
for i in range(len(X_xor)):
    print(f"Input: {X_xor[i]}, Predicted output: {out_xor[i][0]:.4f}")

print("\nFinal predictions for AND-NOT MLP:")
for i in range(len(X_and_not)):
    print(f"Input: {X_and_not[i]}, Predicted output: {out_and_not[i][0]:.4f}")
```