Q.1:

Implement a function to train a linear regression model using stochastic gradient descent (SGD) with mini-batch updates. The function should include options for different learning rates, batch sizes, and a regularization term.

Input:

X = [[1, 1], [1, 2], [1, 3], [1, 4]]
y = [3, 4, 5, 6]
learning_rate = 0.01
batch_size = 2
num_iterations = 1000
regularization_term = 0.1
random_state=42

Expected Output:

Optimized coefficients: array([0, 1])

```
X = [[1, 1], [1, 2], [1, 3], [1, 4]]
y = [3, 4, 5, 6]
learning_rate = 0.01
batch_size = 2
num_iterations = 1000
regularization_term = 0.1
random_state=42

from sklearn.linear_model import SGDRegressor
from sklearn.preprocessing import StandardScaler

def train_linear_regression(X, y, learning_rate, batch_size, regularization_term, max_iterations):
    """
    Train a linear regression model using stochastic gradient descent (SGD) with mini-batch updates.

    Parameters:
        X (array-like): Input features with shape (n_samples, n_features).
        y (array-like): Target values with shape (n_samples,).
        learning_rate (float, optional): The learning rate for gradient descent. Default is 0.01.
        batch_size (int, optional): The size of the mini-batches used in training. Default is 32.
        regularization (str or None, optional): The type of regularization used. Can be 'l1', 'l2', or None. Default is None.
        max_iterations (int, optional): The maximum number of iterations for training. Default is 1000.

    Returns:
        sklearn.linear_model.SGDRegressor: The trained SGDRegressor model.
    """

    # Standardize the input features for better convergence
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    # Initialize the SGDRegressor with the given parameters
    model = SGDRegressor(
        loss='squared_error',
        learning_rate='constant',
        eta0=learning_rate,
        alpha=regularization_term,
        max_iter=max_iterations,
        tol=1e-3,
        random_state=0
    )

    # Train the model in mini-batches
    n_samples = X_scaled.shape[0]
    for iteration in range(max_iterations):
        start_idx = 0
        while start_idx < n_samples:
            end_idx = start_idx + batch_size
            X_batch = X_scaled[start_idx:end_idx]
            y_batch = y[start_idx:end_idx]
            model.partial_fit(X_batch, y_batch)
            start_idx = end_idx

    return model

model = train_linear_regression(X,
                                y,
                                learning_rate, batch_size,
                                regularization_term,
                                num_iterations)
model.coef_
```

```
array([0.        , 1.01715224])
```

Q.2:

Write a function to implement linear regression with Lasso regularization (L1 regularization) using coordinate descent. The function should allow for different regularization parameters and tolerance levels for convergence.

Input:

X = [[1, 1], [1, 2], [1, 3], [1, 4]]
y = [3, 4, 5, 6]
regularization_param = 0.1
tolerance = 0.001

Expected Output:

Optimized coefficients: [1.6, 0.8]

```
from sklearn.linear_model import Lasso

X = [[1, 1], [1, 2], [1, 3], [1, 4]]
y = [3, 4, 5, 6]

regularization_param = 0.1
tolerance = 0.001

def train_linear_regression_with_lasso(X, y, alpha, tol):
    """
    Train a linear regression model with Lasso (L1) regularization using coordinate descent.

    Parameters:
        X (array-like): Input features with shape (n_samples, n_features).
        y (array-like): Target values with shape (n_samples,).
        alpha (float, optional): The regularization strength (lambda). Default is 1.0.
        tol (float, optional): The tolerance for convergence. Default is 1e-4.

    Returns:
        sklearn.linear_model.Lasso: The trained Lasso model.
    """

    # Initialize the Lasso model with the given parameters
    model = Lasso(alpha=alpha,
                  tol=tol,
                  random_state=0)

    # Train the model
    model.fit(X, y)

    return model

model = train_linear_regression_with_lasso(X,
                                           y,
                                           alpha=regularization_param,
                                           tol=tolerance)
model.coef_
```

```
array([0.  , 0.92])
```

Q.3:

Create a program that performs logistic regression with L1 regularization (Lasso) using coordinate descent.

Input:

X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 1, 1]
regularization_param = 0.1
tolerance = 0.001

```
import numpy as np
from sklearn.linear_model import LogisticRegression

X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 1, 1]
regularization_param = 0.1
tolerance = 0.001

def logistic_regression_with_l1_regularization(X, y, regularization_param, tolerance):
    # Create the Logistic Regression model with L1 regularization
    model = LogisticRegression(penalty='l1',
                               C=1/regularization_param,
                               tol=tolerance,
                               solver='liblinear',
                               random_state=0)

    # Train the model
    model.fit(X, y)

    return model


model = logistic_regression_with_l1_regularization(X,
                                                   y,
                                                   regularization_param,
                                                   tolerance)
model.coef_
```

```
array([[2.23260945, 0.        ]])
```

Q.4:

Implement a function to perform logistic regression with L2 regularization (Ridge) using gradient descent.

Input:

X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 1, 1]
learning_rate = 0.01
num_iterations = 1000
regularization_param = 0.1

```
import numpy as np
from sklearn.linear_model import LogisticRegression

X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 1, 1]
num_iterations = 1000
regularization_param = 0.1
tolerance = 0.001

def logistic_regression_with_l2_regularization(X, y, tolerance, num_iterations, regularization_param):
    # Create the Logistic Regression model with L2 regularization
    model = LogisticRegression(penalty='l2',
                               tol=tolerance,
                               C=1/regularization_param,
                               max_iter=num_iterations,
                               random_state=0)

    # Train the model
    model.fit(X, y)

    return model

model = logistic_regression_with_l2_regularization(X, y, learning_rate, num_iterations, regularization_param)
model.coef_
```

```
array([[1.70966932, 1.73695077]])
```

Q.5:

Write a program to handle imbalanced classes in logistic regression by assigning class weights.

Input:

X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 0, 1]
class_weights = {0: 0.2, 1: 0.8}

```
from sklearn.linear_model import LogisticRegression

# Input data
X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 0, 1]

# Class weights
class_weights = {0: 0.2, 1: 0.8}

# Create a Logistic Regression model with class weights
log_reg_model = LogisticRegression(class_weight=class_weights,
                                   random_state=0)

# Fit the model to the data
log_reg_model.fit(X, y)

log_reg_model.coef_
```

```
array([[0.37945496, 0.3794771 ]])
```

Q.6:

Create a program to evaluate the multicollinearity among features in a logistic regression model using the variance inflation factor (VIF).

Input:

X = [[1, 2, 3], [2, 4, 6], [3, 6, 9], [4, 8, 12]]

```
import numpy as np
import pandas as pd
from statsmodels.stats.outliers_influence import variance_inflation_factor

X = np.array([[1, 2, 3],
              [2, 4, 6],
              [3, 6, 9],
              [4, 8, 12]])

def calculate_vif(X):
    """
    Calculate the variance inflation factor (VIF) for each feature in the dataset.
    """

    # VIF dataframe
    vif_data = pd.DataFrame()
    vif_data["feature"] = ["1st", "2nd", "3rd"]

    # calculating VIF for each feature
    vif_data["VIF"] = [variance_inflation_factor(X, i)
                       for i in range(len(vif_data))]

    return vif_data


# Calculate the VIF values
vif_values = calculate_vif(X)

# Print the VIF values for each feature
print("VIF values:")
print(vif_values)
```

```
VIF values:
  feature  VIF
0     1st  inf
1     2nd  inf
2     3rd  inf
C:\Users\murtuzad\Anaconda3\lib\site-packages\statsmodels\stats\outliers_influence.py:195: RuntimeWarning: divide by zero encountered in double_sca
lars
  vif = 1. / (1. - r_squared_i)
```

Q.7:

Write a program to calculate the area under the ROC curve (AUC) for a logistic regression model.

Input:

y_true = [0, 1, 1, 0, 1]
y_pred = [0.2, 0.8, 0.6, 0.3, 0.9]

```
from sklearn.metrics import roc_auc_score

y_true = [0, 1, 1, 0, 1]
y_pred = [0.2, 0.8, 0.6, 0.3, 0.9]

def calculate_auc(y_true, y_pred):
    auc = roc_auc_score(y_true, y_pred)
    return auc

auc_score = calculate_auc(y_true, y_pred)
print("Area Under the ROC Curve (AUC): {:.2f}".format(auc_score))
```

```
Area Under the ROC Curve (AUC): 1.00
```

Q.8:

Implement a program to perform logistic regression using stochastic gradient descent (SGD) with mini-batch updates.

Input:

X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 1, 1]
learning_rate = 0.01
batch_size = 2
num_iterations = 1000

```
from sklearn.linear_model import SGDClassifier
import numpy as np

# Input data
X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 1, 1]
learning_rate = 0.01
batch_size = 2
num_iterations = 1000

def logistic_regression_sgd(X, y, learning_rate=0.01, batch_size=2, num_iterations=1000):
    # Convert data to numpy arrays
    X = np.array(X)
    y = np.array(y)

    # Standardize the input features for better convergence
    scaler = StandardScaler()
    X = scaler.fit_transform(X)

    # Initialize the SGDClassifier with logistic loss and mini-batch updates
    clf = SGDClassifier(loss='log',
                        learning_rate='constant',
                        eta0=learning_rate,
                        random_state=0)

    # Perform mini-batch updates for the specified number of iterations
    for _ in range(num_iterations):
        # Randomly shuffle the data indices
        indices = np.random.permutation(len(X))

        # Divide the data into mini-batches and update the classifier
        for batch_start in range(0, len(X), batch_size):
            batch_indices = indices[batch_start:batch_start + batch_size]
            clf.partial_fit(X[batch_indices], y[batch_indices], classes=np.unique(y))

    return clf


# Train the logistic regression model using SGD
model = logistic_regression_sgd(X, y,
                                learning_rate,
                                batch_size,
                                num_iterations)

# Make predictions
predictions = model.predict(X)

model.coef_
```

```
array([[2.48201691, 2.48201691]])
```

Q.9:

Implement a function to perform logistic regression and display confusion matrix and classification report.

Input:

X_train = [[1, 2], [2, 3], [3, 4], [4, 5]]
y_train = [0, 0, 1, 1]
X_val = [[1, 3], [2, 4], [3, 5]]
y_val = [0, 1, 1]
learning_rate = 0.01
batch_size = 2
num_iterations = 1000

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report

# Input data
X_train = [[1, 2], [2, 3], [3, 4], [4, 5]]
y_train = [0, 0, 1, 1]
X_val = [[1, 3], [2, 4], [3, 5]]
y_val = [0, 1, 1]
learning_rate = 0.01
batch_size = 2
num_iterations = 1000

def logistic_regression_with_metrics(X_train, y_train, X_val, y_val, learning_rate, batch_size, num_iterations):

    # Standardize the input features for better convergence
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_val = scaler.transform(X_val)

    # Initialize the SGDClassifier with logistic loss and mini-batch updates
    model = SGDClassifier(loss='log',
                          learning_rate='constant',
                          eta0=learning_rate,
                          random_state=0)

    # Step 2: Train the model on the training data
    model.fit(X_train, y_train)

    # Step 3: Make predictions on the validation data
    y_pred = model.predict(X_val)

    # Step 4: Display the confusion matrix and classification report
    print("Confusion Matrix:")
    print(confusion_matrix(y_val, y_pred))

    print("\nClassification Report:")
    print(classification_report(y_val, y_pred))


# Call the function with the given input
logistic_regression_with_metrics(X_train, y_train, X_val, y_val, learning_rate, batch_size, num_iterations)
```

```
Confusion Matrix:
[[1 0]
 [0 2]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         2

    accuracy                           1.00         3
   macro avg       1.00      1.00      1.00         3
weighted avg       1.00      1.00      1.00         3
```

Q.10:

Write a program to calculate the log loss (binary cross-entropy) for a logistic regression model using vectorized operations.

Input:

y_true = [0, 1, 1, 0]
y_pred = [0.2, 0.8, 0.9, 0.3]

```
import numpy as np
from sklearn.metrics import log_loss

# Input data
y_true = np.array([0, 1, 1, 0])
y_pred = np.array([0.2, 0.8, 0.9, 0.3])

# Ensure y_pred is within [epsilon, 1-epsilon] to avoid log(0) and log(1) issues
epsilon = 1e-15
y_pred = np.clip(y_pred, epsilon, 1 - epsilon)

# Calculate the log loss (binary cross-entropy)
logloss = log_loss(y_true, y_pred)

print("Log Loss (Binary Cross-Entropy):", logloss)
```

```
Log Loss (Binary Cross-Entropy): 0.22708064055624455
```

Q.11

Write a program to build a decision tree classifier from scratch using the CART algorithm.

Input:

X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 1, 1]

```
from sklearn.tree import DecisionTreeClassifier

# Input data
X_train = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 1, 1]
X_val = [[2, 4], [1, 3], [3, 5]]

# Create the decision tree classifier with CART algorithm
clf = DecisionTreeClassifier(random_state=0)

# Train the model using the input data
clf.fit(X_train, y)

predictions = clf.predict(X_val)
predictions
```

```
array([1, 0, 1])
```

Q.12:

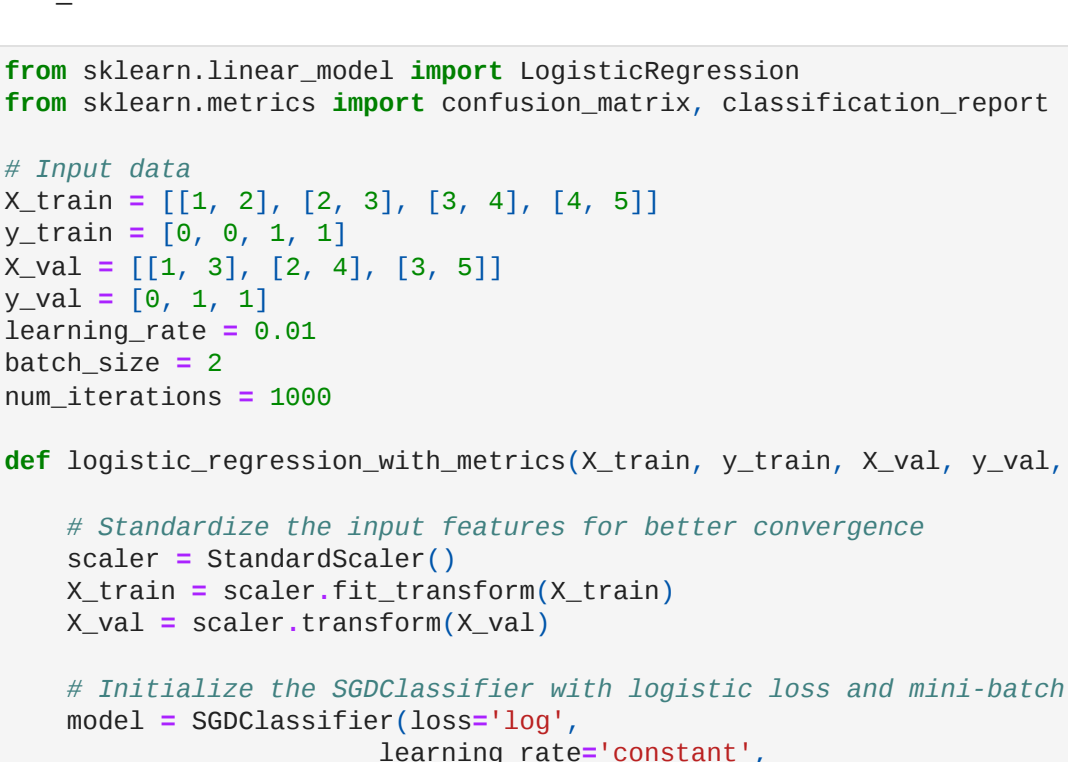
Create a function using sklearn to visualize a decision tree using a graph representation.

Input:

X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 1, 1]

```
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier, plot_tree, export_text
import numpy as np

# Input data
X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 1, 1]

def visualize_decision_tree(X, y):
    # Create the decision tree classifier with CART algorithm
    clf = DecisionTreeClassifier()

    # Train the model using the input data
    clf.fit(X, y)

    # Obtain a text representation of the decision tree
    tree_text = export_text(clf, feature_names=[f'Feature {i+1}' for i in range(len(X[0]))])
    print("Decision Tree Text Representation:")
    print(tree_text)

    # Create a graphical representation of the decision tree
    plt.figure(figsize=(10, 6))
    plot_tree(clf,
              filled=True,
              feature_names=[f'Feature {i+1}' for i in range(len(X[0]))],
              class_names=['Class 0', 'Class 1'])
    plt.show()


# Call the visualization function
visualize_decision_tree(X, y)
```

```
Decision Tree Text Representation:
|--- Feature 2 <= 3.50
|   |--- class: 0
|--- Feature 2 >  3.50
|   |--- class: 1
```



Q.13:

Write a program using sklearn to handle categorical features in a dataset by performing one-hot encoding.

Input:

dataset = [[1, 'A', 'Yes'], [1, 'B', 'Yes'], [0, 'B', 'No'], [0, 'A', 'No']]

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

# Input dataset
dataset = [[1, 'A', 'Yes'], [1, 'B', 'Yes'], [0, 'B', 'No'], [0, 'A', 'No']]

# Define the column indices of the categorical features
categorical_features = [1,2]

# Create a ColumnTransformer to perform one-hot encoding on the categorical features
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(), categorical_features)
    ],
    remainder='passthrough'
)

# Fit and transform the data using the preprocessor
dataset_encoded = preprocessor.fit_transform(dataset)
dataset_encoded
```

```
array([[1.0, 0.0, 0.0, 1.0, 1],
       [0.0, 1.0, 0.0, 1.0, 1],
       [0.0, 1.0, 1.0, 0.0, 0],
       [1.0, 0.0, 1.0, 0.0, 0]], dtype=object)
```

Q.14

Implement a program using sklearn to handle imbalanced classes in a decision tree by assigning class weights.

Input:

X = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 0, 1]
class_weights = {0: 0.2, 1: 0.8}

```
from sklearn.tree import DecisionTreeClassifier
import numpy as np

# Input data
X_train = [[1, 2], [2, 3], [3, 4], [4, 5]]
y = [0, 0, 0, 1]
X_val = [[2, 4], [1, 3], [3, 5]]

# Class weights
class_weights = {0: 0.2, 1: 0.8}

# Create the decision tree classifier with class weights
clf = DecisionTreeClassifier(class_weight=class_weights,
                             random_state=0)

# Fit the classifier on the data
clf.fit(X_train, y)

# Predict on new data
predictions = clf.predict(X_val)
predictions
```

```
array([0, 0, 1])
```

Q.15:

Write a program using sklearn to perform hierarchical clustering using the complete linkage method.

Input:

X = [[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]]

```
import numpy as np
from sklearn.cluster import AgglomerativeClustering
import matplotlib.pyplot as plt

# Input data
X = np.array([[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]])

# Perform hierarchical clustering using complete linkage
clustering = AgglomerativeClustering(linkage='complete')
clustering.fit(X)

# Get the cluster assignments for each data point
labels = clustering.labels_
labels
```

```
array([0, 0, 0, 0, 1, 1], dtype=int64)
```

Q.16:

Implement a program using sklearn to perform density-based clustering using the DBSCAN algorithm.

Input:

X = [[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]]
epsilon = 3
min_samples = 2

```
from sklearn.cluster import DBSCAN
import numpy as np

# Input data
X = np.array([[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]])
epsilon = 3
min_samples = 2

def perform_dbscan_clustering(X, epsilon, min_samples):
    # Create a DBSCAN object with the specified epsilon and min_samples
    dbscan = DBSCAN(eps=epsilon, min_samples=min_samples)

    # Perform clustering on the data
    clustering_labels = dbscan.fit_predict(X)

    # Return the clustering labels
    return clustering_labels

# Perform DBSCAN clustering
labels = perform_dbscan_clustering(X, epsilon, min_samples)
labels
```

```
array([0, 0, 1, 1, 2, 2], dtype=int64)
```

Q.17

Write a program using sklearn to perform k-means clustering using the k-means++ initialization method.

Input:

X = [[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]]
k = 2

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

# Input data
X = np.array([[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]])
k = 2

# Perform k-means clustering
kmeans = KMeans(n_clusters=k, init='k-means++', random_state=0)
y_kmeans = kmeans.fit_predict(X)

# Extract the cluster centers and labels
cluster_centers = kmeans.cluster_centers_
labels = kmeans.labels_
labels
```

```
array([1, 1, 1, 1, 0, 0])
```

Q.18:

Implement a program using sklearn to perform agglomerative clustering using the average linkage method.

Input:

X = [[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]]

```
from sklearn.cluster import AgglomerativeClustering

X = np.array([[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]])

# Perform hierarchical clustering using avergae linkage
model = AgglomerativeClustering(linkage='average')
model.fit(X)

model.labels_
```

```
array([0, 0, 0, 0, 1, 1], dtype=int64)
```

Q.19

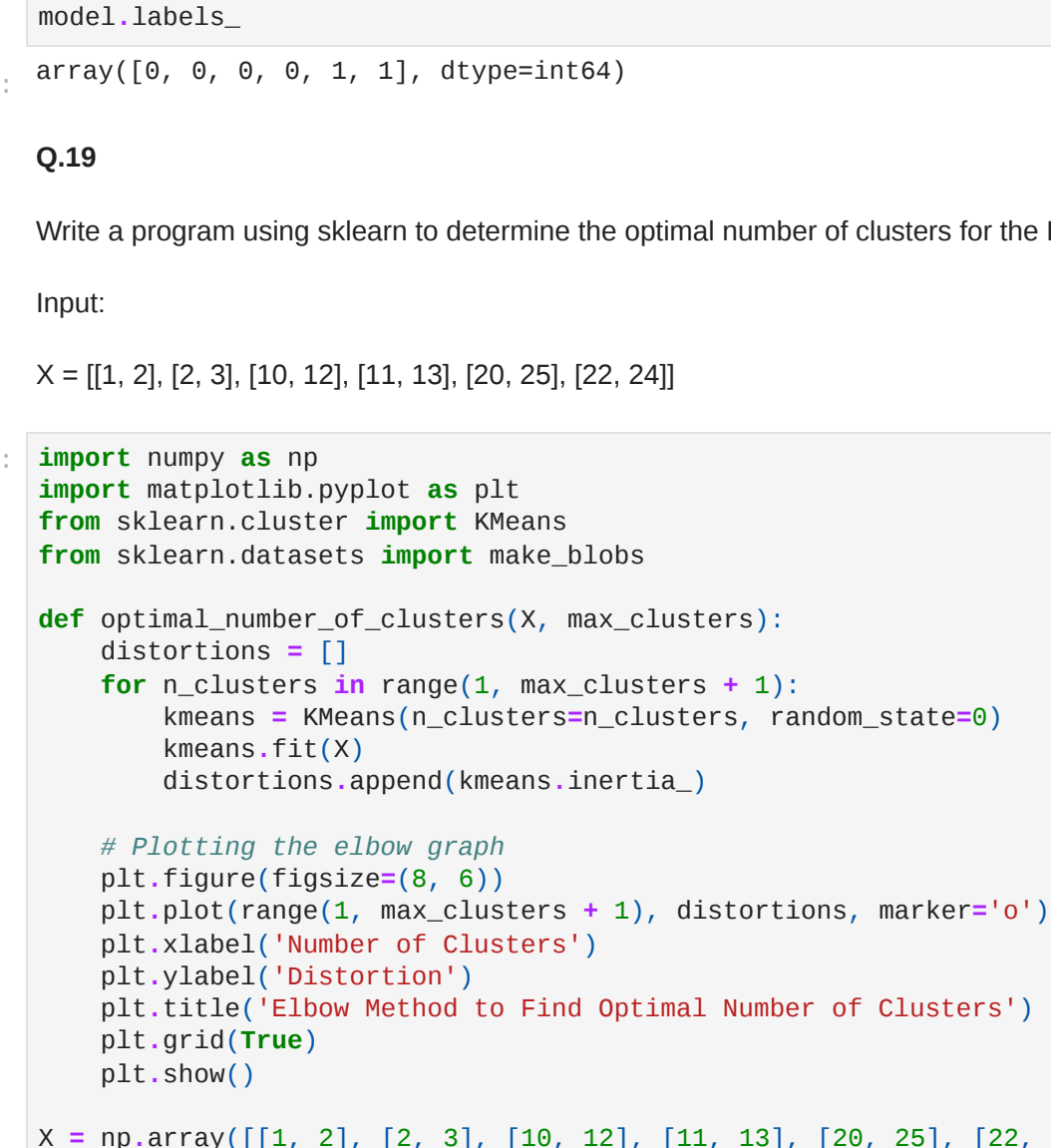
Write a program using sklearn to determine the optimal number of clusters for the KMeans algorithm.

Input:

X = [[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]]

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

def optimal_number_of_clusters(X, max_clusters):
    distortions = []
    for n_clusters in range(1, max_clusters + 1):
        kmeans = KMeans(n_clusters=n_clusters, random_state=0)
        kmeans.fit(X)
        distortions.append(kmeans.inertia_)

    # Plotting the elbow graph
    plt.figure(figsize=(8, 6))
    plt.plot(range(1, max_clusters + 1), distortions, marker='o')
    plt.xlabel('Number of Clusters')
    plt.ylabel('Distortion')
    plt.title('Elbow Method to Find Optimal Number of Clusters')
    plt.grid(True)
    plt.show()

X = np.array([[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]])
max_clusters = len(X)  # You can change this to your desired maximum number of clusters

optimal_number_of_clusters(X, max_clusters)
```

```
C:\Users\murtuzad\Anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:1036: UserWarning: KMeans is known to have a memory leak on Windows with M
KL, when there are less chunks than available threads. You can avoid it by setting the environment variable OMP_NUM_THREADS=1.
  warnings.warn(
```



Q.20:

Implement a program using sklearn to perform agglomerative clustering using the average linkage method and display the cluster dendogram.

Input:

X = [[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]]

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

# Input data
X = np.array([[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]])

# Agglomerative clustering with average linkage
agg_clustering = AgglomerativeClustering(linkage='average')
labels = agg_clustering.fit_predict(X)

# Create linkage matrix
Z = linkage(X, method='average')

# Display dendrogram
plt.figure(figsize=(10, 5))
dendrogram(Z, labels=labels)
plt.title('Agglomerative Clustering Dendrogram')
plt.xlabel('Data Points')
plt.ylabel('Distance')
plt.show()
```