Principles of Algorithm Design
Final Exam

Student Name

Hamed Araab

Student Number

9925003

Contents

Question 1

Introduction

This documentation provides an overview of the code implementation for the K-Means Clustering algorithm. The code performs clustering on a list of points based on the specified number of clusters. The implementation follows the principles of Algorithm Design. The purpose of this code is to find the shortest inter-cluster distance using the K-Means Clustering algorithm.

Code Structure

The code consists of the following components:

1. `KMeansClustering` class: Performs the K-Means Clustering algorithm on a list of points.
2. `main` function: Provides the user interface for input and output handling.

`KMeansClustering` Class

Class Description

The `KMeansClustering` class encapsulates the K-Means Clustering algorithm. It performs clustering on a list of points and calculates the shortest inter-cluster distance.

Class Attributes

- `_points`: A list of points provided as input.
- `_k`: The specified number of clusters.
- `_means`: The means of each cluster.
- `_clusters`: The clusters formed during the clustering process.

Class Methods

`_get_distance(point1, point2)`

- Description: Returns the Euclidean distance between two points.
- Parameters:
  - `point1`: The first point.
  - `point2`: The second point.
- Returns: The Euclidean distance between the two points.

`__init__(self, points, k)`

- Description: Initializes the `KMeansClustering` class with the given points and number of clusters.
- Parameters:
    - `points`: A list of points.
    - `k`: The number of clusters.

`points`

- Description: Getter method for the `_points` attribute.
- Returns: The list of points.

`k`

- Description: Getter method for the `_k` attribute.
- Returns: The number of clusters.

`means`

- Description: Getter method for the `_means` attribute.
- Returns: The means of each cluster.

`clusters`

- Description: Getter method for the `_clusters` attribute.
- Returns: The clusters formed during the clustering process.

`min_inter_cluster_distance`

- Description: Calculates the shortest inter-cluster distance.
- Returns: The shortest distance between any two points in different clusters.

`_get_initial_means()`

- Description: Returns `k` random points as the initial values for means.
- Returns: A list of `k` randomly selected points from the input.

`_get_clusters()`

- Description: Assigns each point to its nearest mean and forms clusters.
- Returns: A list of clusters, where each cluster contains at least one point.

`_get_means()`

- Description: Calculates the means of all clusters.
- Returns: A list of means for each cluster.

`_get_wcss()`

- Description: Calculates the Within-Cluster Sum of Squares (`WCSS`) metric for the current clustering result.
- Returns: The `WCSS` value.

`_run()`

- Description: Finds the best clustering result for the given instance.

`main` Function

Function Description

The `main` function is the entry point of the program. It handles user input and output.

Function Steps

1. Reads the number of points from the user.
2. Reads the points from the user.
3. Reads the number of clusters from the user.
4. Creates an instance of the `KMeansClustering` class with the given points and number of clusters.
5. Prints the shortest inter-cluster distance.

Usage Example

```
# Start of getting user inputs
points_count = int(input())
points = [list(map(float, input().

split(" "))) for _ in range(points_count)]
clusters_count = int(input())
# End of getting user inputs

clustering = KMeansClustering(points, clusters_count)
print("{:.7f}".format(clustering.min_inter_cluster_distance))
```

The user is prompted to provide the number of points, the coordinates of each point, and the number of clusters. The code then calculates the shortest inter-cluster distance using the K-Means Clustering algorithm and displays the result.

Conclusion

This documentation provided an overview of the code implementation for the K-Means Clustering algorithm. It described the class structure, methods, and the main function responsible for user interaction. The code allows users to perform clustering on a set of points and obtain the shortest inter-cluster distance.

Question 2

Introduction

This documentation provides an overview of the code implementation for solving the Traveling Salesman Problem (TSP). The code uses the principles of Algorithm Design to find the minimum cost of touring a set of nodes in the TSP. The implementation follows a combination of Dijkstra's algorithm and the Heap Queue (Priority Queue) algorithm to efficiently compute the minimum costs.

Code Structure

The code consists of the following components:

1. `TSPSolver` class: Solves the Traveling Salesman Problem using the provided arcs and nodes count.
2. `main` function: Provides the user interface for input and output handling.

`TSPSolver` Class

Class Description

The `TSPSolver` class encapsulates the logic for solving the Traveling Salesman Problem.

Class Attributes

- `_arcs`: A set of arcs representing connections between nodes.
- `_nodes_count`: The total number of nodes.
- `_graph`: The graph representation of the arcs.
- `_all_min_costs`: A dictionary storing the minimum costs for traveling from a start node to other nodes in the graph.

Class Methods

`_get_graph(self)`

- Description: Returns a graph representation using the provided arcs and nodes count.
- Returns: A dictionary representing the graph, where each node is associated with a set of target nodes and their respective costs.

`_get_min_costs(self, start_node)`

- Description: Calculates the minimum costs of traveling from the `start_node` to other nodes using Dijkstra's algorithm.
- Returns: A dictionary containing the minimum costs for traveling from the `start_node` to other nodes.

`get_min_cost_of_tour(self, selected_nodes)`

- Description: Calculates the minimum cost of touring the `selected_nodes` in the Traveling Salesman Problem.
- Returns: The minimum cost of touring the selected nodes.

`main` Function

Function Description

The `main` function is the entry point of the program. It handles user input and output.

Function Steps

1. Reads the number of nodes and arcs from the user.
2. Reads the arcs and stores them in the `_arcs` attribute.
3. Creates an instance of the `TSPSolver` class with the provided arcs and nodes count.
4. Prints "Ready" to indicate that the solver is ready to process further input.
5. Reads the number of travels from the user.
6. Reads the selected nodes for each travel and calculates the minimum cost of touring them using the `get_min_cost_of_tour` method.
7. Prints the minimum cost for each travel.

Usage Example

```
# Start of getting part one of user inputs.
nodes_count, arcs_count = tuple(map(int, input().split(" ")))
arcs = set()

for _ in range(arcs_count):
    user_input = input().split(" ")
    arc = (int(user_input[0]), int(user_input[1]), float(user_input[2]))

    arcs.add(arc)
# End of getting part one of user inputs.

tsp_solver = TSPSolver(arcs, nodes_count)
print("Ready")
```

```
# Start of getting part two user inputs.
travels_count = int(input())
travels = [list(map(int, input().split(" ")))[1:] for _ in
range(travels_count)]
# End of getting part two user inputs.

for selected_nodes in travels:
    print(tsp_solver.get_min_cost_of_tour(selected_nodes))
```

The user is prompted to provide inputs in two parts. In the first part, they input the number of nodes and arcs, followed by the arcs themselves. In the second part, they input the number of travels and the

selected nodes for each travel. The code then calculates and displays the minimum cost of touring the selected nodes for each travel.

Conclusion

This documentation provided an overview of the code implementation for solving the Traveling Salesman Problem. It described the class structure, methods, and the main function responsible for user interaction. The code allows users to input arcs and calculate the minimum cost of touring selected nodes using a combination of Dijkstra's algorithm and the Heap Queue algorithm.

Question 3

Introduction

This documentation provides an overview of the code implementation for finding the Longest Common Subsequence (LCS) between two strings. The code uses the principles of Algorithm Design and employs the Dynamic Programming approach to efficiently compute the LCS.

Code Structure

The code consists of the following components:

1. `print_lcs_details` function: Finds the LCS between two strings and prints the start index of LCS in each string and the length of LCS.
2. `main` function: Provides the user interface for input and output handling.

`print_lcs_details` Function

Function Description

The `print_lcs_details` function calculates the LCS between two strings and prints the start index of LCS in each string and the length of LCS.

Function Parameters

- `string1`: The first input string.
- `string2`: The second input string.

Function Steps

1. Initialize variables `string1_length` and `string2_length` with the lengths of `string1` and `string2`, respectively.
2. Create a 2D array `lengths` with dimensions `(string1_length + 1) x (string2_length + 1)`. Initialize all elements to 0.
3. Initialize variables `max_length`, `start1_index`, and `start2_index` to track the maximum length of LCS and the corresponding start indices.
4. Iterate over the indices `i` from 1 to `string1_length` and `j` from 1 to `string2_length`.
5. If the characters at indices `i-1` in `string1` and `j-1` in `string2` are equal, update `lengths[i][j]` by adding 1 to `lengths[i-1][j-1]`.

6. If `lengths[i][j]` is greater than `max_length`, update `max_length` and set `start1_index` to `i - max_length` and `start2_index` to `j - max_length`.
7. Print the values of `start1_index`, `start2_index`, and `max_length` in the format `start1_index start2_index max_length`.

`main` Function

Function Description

The `main` function is the entry point of the program. It handles user input and output.

Function Steps

1. Reads two strings separated by a space from the user.
2. Calls the `print_lcs_details` function with the two input strings.

Usage Example

```
print_lcs_details(*input().split(" "))
```

The user is prompted to provide two strings separated by a space. The code then calculates the LCS between the two strings and prints the start index of LCS in each string and the length of LCS.

Conclusion

This documentation provided an overview of the code implementation for finding the Longest Common Subsequence (LCS) between two strings. The code follows the principles of Algorithm Design and employs the Dynamic Programming approach to efficiently compute the LCS. The provided `print_lcs_details` function accepts two strings, calculates the LCS, and prints the start indices and length of LCS.

Question 6

Introduction

This documentation provides an overview of the code implementation for performing linear regression using the principles of Algorithm Design. The code defines a `LinearRegressor` class that fits a linear regression model to the given data and provides methods for prediction and evaluation.

Code Structure

The code consists of the following components:

1. `LinearRegressor` class: Represents a linear regression model.
2. `__init__` method: Initializes the linear regression model by fitting the data and calculating coefficients, RMSE (Root Mean Square Error), and R2 score.
3. `coefficients` property: Returns the coefficients of the linear regression model.
4. `rmse` property: Returns the Root Mean Square Error (RMSE) of the linear regression model.
5. `r2` property: Returns the R2 score (coefficient of determination) of the linear regression model.
6. `_fit_scaler_X` method: Fits the scaler for input feature normalization.
7. `_fit_scaler_Y` method: Fits the scaler for target variable normalization.
8. `_normalize_X` method: Normalizes the input features using the fitted scaler.
9. `_normalize_Y` method: Normalizes the target variable using the fitted scaler.
10. `_undo_normalization_Y` method: Reverses the normalization of the target variable.
11. `predict` method: Predicts the target variable for a given set of input features.

`LinearRegressor` Class

Class Description

The `LinearRegressor` class represents a linear regression model. It provides methods for fitting the model to the given data, calculating model evaluation metrics, and making predictions.

`__init__` Method

Method Description

The `__init__` method initializes the `LinearRegressor` object. It fits the data, calculates the coefficients, RMSE, and R2 score of the linear regression model.

Method Parameters

- `X`: The input features for the linear regression model.
- `Y`: The target variable for the linear regression model.

Method Steps

1. Convert the input features `X` and target variable `Y` to arrays.
2. Append a column of ones to the input features `X` to incorporate the intercept term in the linear regression model.
3. Fit the scaler for input feature normalization using the `_fit_scaler_X` method.
4. Normalize the input features `X` using the `_normalize_X` method.
5. Fit the scaler for target variable normalization using the `_fit_scaler_Y` method.
6. Normalize the target variable `Y` using the `_normalize_Y` method.
7. Calculate the coefficients of the linear regression model using the formula: `inv(X.T @ X) @ X.T @ Y`.
8. Perform prediction using the normalized input features `X` and the calculated coefficients.
9. Undo the normalization of the predicted target variable using the `_undo_normalization_Y` method.
10. Undo the normalization of the target variable `Y`.
11. Calculate the Root Mean Square Error (RMSE) using the formula: `((Y - Y_prediction) ** 2).mean() ** 0.5`.
12. Calculate the R2 score (coefficient of determination) using the formula: `1 - (((Y - Y_prediction) ** 2).sum() / ((Y - Y.mean()) ** 2).sum())`.

`coefficients` Property

Property Description

The `coefficients` property returns the coefficients of the linear regression model.

`rmse` Property

Property Description

The `rmse` property returns the Root Mean Square Error (RMSE) of the linear regression model.

`r2` Property

Property Description

The `r2` property returns the R2 score (coefficient of determination) of the linear regression model.

`_fit_scaler_X` Method

Method Description

The `_fit_scaler_X` method fits the scaler for input feature normalization.

Method Parameters

- `X`: The input features for the linear regression model.

Method Steps

1. Calculate the mean of the input features `X` and store it in the `_mean_X` variable.
2. Calculate the standard deviation of the input features `X` and store it in the `_std_X` variable.

`_fit_scaler_Y` Method

Method Description

The `_fit_scaler_Y` method fits the scaler for target variable normalization.

Method Parameters

- `Y`: The target variable for the linear regression model.

Method Steps

1. Calculate the mean of the target variable `Y` and store it in the `_mean_Y` variable.
2. Calculate the standard deviation of the target variable `Y` and store it in the `_std_Y` variable.

`_normalize_X` Method

Method Description

The `_normalize_X` method normalizes the input features using the fitted scaler.

Method Parameters

- `X`: The input features for the linear regression model.

Method Steps

1. Subtract the mean of the input features `_mean_X` from the input features `X`.
2. Divide the result by the standard deviation of the input features `_std_X`.
3. Return the normalized input features.

`_normalize_Y` Method

Method Description

The `_normalize_Y` method normalizes the target variable using the fitted scaler.

Method Parameters

- `Y`: The target variable for the linear regression model.

Method Steps

1. Subtract the mean of the target variable `_mean_Y` from the target variable `Y`.
2. Divide the result by the standard deviation of the target variable `_std_Y`.
3. Return the normalized target variable.

`_undo_normalization_Y` Method

Method Description

The `_undo_normalization_Y` method reverses the normalization of the target variable.

Method Parameters

- `Y`: The normalized target variable.

Method Steps

1. Multiply the normalized target variable `Y` by the standard deviation of the target variable `_std_Y`.
2. Add the mean of the target variable `_mean_Y`.

3. Return the unnormalized target variable.

`predict` Method

Method Description

The `predict` method predicts the target variable for a given set of input features.

Method Parameters

- `X`: The input features for which to predict the target variable.

Method Steps

1. Convert the input features `X` to an array.
2. Append a column of ones to the input features `X` to incorporate the intercept term in the linear regression model.
3. Normalize the input features `X` using the `_normalize_X` method.
4. Perform prediction using the normalized input features `X` and the coefficients of the linear regression model.
5. Undo the normalization of the predicted target variable using the `_undo_normalization_Y` method.
6. Return the predicted target variable.

Usage Example

```
model1 = LinearRegressor(X[["sqft_living"]], Y)
print(f"R2: {model1.r2}")
print(f"RMSE: {model1.rmse}")

model2 = LinearRegressor(X, Y)
print(f"R2: {model2.r2}")
print(f"RMSE: {model2.rmse}")
```

The code creates two instances of the `LinearRegressor` class. The first instance (`model1`) fits a linear regression model using only the "sqft_living" feature from the input features `X` and the target variable`Y`. The second instance (`model2`) fits a linear regression model using all the input features from `X`and the target variable`Y`. The R2 score and RMSE are then printed for both models.