Aim :: Perform Normalization of data (Min-max and Z-score).

Code::

```
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <cmath>    // for std::sqrt
#include <limits>   // for numeric limits
#include <algorithm> // for std::min_element and std::max_element

// Function to split a line into tokens by a delimiter
std::vector<std::string> split(const std::string& line, char delimiter = ',') {
    std::vector<std::string> tokens;
    std::string token;
    std::istringstream tokenStream(line);
    while (std::getline(tokenStream, token, delimiter)) {
        tokens.push_back(token);
    }
    return tokens;
}

// Function to normalize a column using Min-Max and Z-Score formulas
void normalize_column_manually(const std::string& input_csv_file, const std::string&
output_csv_file, const std::string& column_name) {
    std::ifstream infile(input_csv_file);
    if (!infile.is_open()) {
        std::cerr << "Could not open the input file." << std::endl;
        return;
    }

    std::ofstream outfile(output_csv_file);
    if (!outfile.is_open()) {
        std::cerr << "Could not open the output file." << std::endl;
        return;
    }

    std::string line;
    std::vector<std::vector<std::string>> data;
    std::vector<double> column_data;
    size_t column_index = std::numeric_limits<size_t>::max();

    // Read the CSV file
    bool is_header = true;
    while (std::getline(infile, line)) {
        std::vector<std::string> row = split(line);
        if (is_header) {
            // Find the index of the column to normalize
```

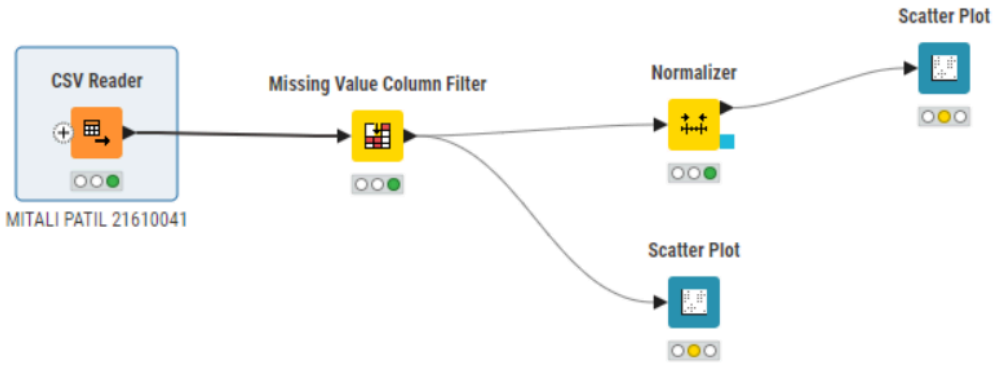
```
            for (size_t i = 0; i < row.size(); ++i) {
                if (row[i] == column_name) {
                    column_index = i;
                    break;
                }
            }
            if (column_index == std::numeric_limits<size_t>::max()) {
                std::cerr << "Column '" << column_name << "' does not exist in the CSV file." <<
std::endl;
                return;
            }
            // Append the new columns to the header row
            row.push_back(column_name + "_minmax");
            row.push_back(column_name + "_zscore");
            is_header = false;
        } else {
            // Add column data for normalization
            column_data.push_back(std::stod(row[column_index]));
        }
        data.push_back(row);
    }

    // Min-Max normalization
    double minA = *std::min_element(column_data.begin(), column_data.end());
    double maxA = *std::max_element(column_data.begin(), column_data.end());
    double new_minA = 0.0;  // Default new minimum
    double new_maxA = 1.0;  // Default new maximum

    // Z-Score normalization
    double sum = 0.0, sq_sum = 0.0;
    for (double value : column_data) {
        sum += value;
        sq_sum += value * value;
    }
    double mean = sum / column_data.size();
    double variance = (sq_sum / column_data.size()) - (mean * mean);
    double std_dev = std::sqrt(variance);

    // Write normalized data to the new CSV file
    for (size_t i = 0; i < data.size(); ++i) {
        if (i == 0) {
            // Write header row
            for (const std::string& token : data[i]) {
                outfile << token << ",";
            }
            outfile << std::endl;
        } else {
            // Write normalized rows
```

```
            for (size_t j = 0; j < data[i].size(); ++j) {
                outfile << data[i][j] << ",";
            }
            // Append Min-Max normalized value
            double normalized_minmax = ((column_data[i - 1] - minA) / (maxA - minA)) *
(new_maxA - new_minA) + new_minA;
            outfile << normalized_minmax << ",";
            // Append Z-Score normalized value
            double normalized_zscore = (column_data[i - 1] - mean) / std_dev;
            outfile << normalized_zscore << std::endl;
        }
    }

    infile.close();
    outfile.close();
    std::cout << "Normalized data saved to " << output_csv_file << std::endl;
}

int main() {
    std::string input_file = "input.csv"; // Replace with your input file
    std::string output_file = "normalized_output.csv"; // Output file

    // Take column name to be normalized from the user
    std::string column_to_normalize;
    std::cout << "Enter the name of the column to normalize: ";
    std::getline(std::cin, column_to_normalize);

    normalize_column_manually(input_file, output_file, column_to_normalize);

    return 0;
}
```

Output :

```
E:\DM KNIME\Normalization>Norm
Enter the name of the column to normalize: sepal_length
                    Data      sepal_length_minmax      sepal_length_zscore
------------------------------------------------------------------------------
                  5.1000                   0.1613                  -0.8351
                  4.6000                   0.0000                  -1.3207
                  6.4000                   0.5806                   0.4273
                  7.3000                   0.8710                   1.3013
                  5.8000                   0.3871                  -0.1554
                  7.7000                   1.0000                   1.6897
                  4.9000                   0.0968                  -1.0294
                  6.1000                   0.4839                   0.1360
                  6.8000                   0.7097                   0.8157
                  4.9000                   0.0968                  -1.0294
Normalized data saved to normalized_output.csv

E:\DM KNIME\Normalization>
```

Knime



Knime Output:

Min-Max Normalization

Rows: 10 | Columns: 5

#	RowID	sepal_length *Number (double)*
1	1	0.161
2	2	0
3	3	0.581
4	4	0.871
5	5	0.387
6	6	1
7	7	0.097
8	8	0.484
9	9	0.71
10	10	0.097

Z-score Normalization

Rows: 10 | Columns: 5

#	RowID	sepal_length *Number (double)*
1	1	-0.792
2	2	-1.253
3	3	0.405
4	4	1.235
5	5	-0.147
6	6	1.603
7	7	-0.977
8	8	0.129
9	9	0.774
10	10	-0.977

Aim : Find frequent item sets from given transaction data.

Code :

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <set>
#include <map>
#include <string>
#include <algorithm>
#include <iterator>

// Function to generate itemset support counts with support threshold pruning
std::map<std::set<std::string>, int> generate_itemset_support_count(int k, const
std::vector<std::set<std::string>>& transactions, int min_support) {
    std::map<std::set<std::string>, int> itemset_counts;

    // Generate all itemsets of size k
    for (const auto& transaction : transactions) {
        std::vector<std::string> items(transaction.begin(), transaction.end());
        std::sort(items.begin(), items.end());

        std::vector<bool> v(items.size());
        std::fill(v.begin(), v.begin() + k, true); // Set first k elements to true
        do {
            std::set<std::string> itemset;
            for (size_t i = 0; i < items.size(); ++i) {
                if (v[i]) {
                    itemset.insert(items[i]);
                }
            }
            itemset_counts[itemset]++;
        } while (std::prev_permutation(v.begin(), v.end()));
    }

    // Prune itemsets that do not meet the minimum support threshold
    for (auto it = itemset_counts.begin(); it != itemset_counts.end();) {
        if (it->second < min_support) {
            it = itemset_counts.erase(it);
        } else {
            ++it;
        }
    }

    return itemset_counts;
}
```

```
// Read transactions from CSV, skipping the first row and first column
std::vector<std::set<std::string>> read_transactions_from_csv(const std::string& file_path) {
    std::ifstream infile(file_path);
    std::vector<std::set<std::string>> transactions;
    std::string line;

    bool is_first_row = true; // Flag to skip the first row

    while (std::getline(infile, line)) {
        if (is_first_row) {
            is_first_row = false;
            continue; // Skip the first row
        }

        std::set<std::string> transaction;
        std::stringstream ss(line);
        std::string item;

        bool is_first_column = true; // Flag to skip the first column

        while (std::getline(ss, item, ',')) {
            if (is_first_column) {
                is_first_column = false; // Skip the first column
                continue;
            }
            if (!item.empty()) { // Ignore missing values
                transaction.insert(item);
            }
        }

        transactions.push_back(transaction);
    }

    return transactions;
}

int main() {
    std::string file_path;
    int min_support;

    // User input for file path and minimum support
    std::cout << "Enter the path to your CSV file: ";
    std::getline(std::cin, file_path);
    std::cout << "Enter the minimum support threshold: ";
    std::cin >> min_support;

    // Read transactions from CSV
```

```
    auto transactions = read_transactions_from_csv(file_path);

    // Determine the unique items
    std::set<std::string> unique_items;
    for (const auto& transaction : transactions) {
        unique_items.insert(transaction.begin(), transaction.end());
    }

    // Open output file to save results
    std::ofstream outFile("itemset_support_counts.txt");
    outFile << "Itemset Support Counts (min support = " << min_support << "):\n";

    // Generate support counts for itemsets of different sizes using the function
    for (int k = 1; k <= unique_items.size(); ++k) {
        auto itemset_counts = generate_itemset_support_count(k, transactions, min_support);

        if (!itemset_counts.empty()) {
            outFile << "\nItemsets of size " << k << ":\n";
            for (const auto& pair : itemset_counts) {
                const auto& itemset = pair.first;
                int count = pair.second;
                outFile << "{ ";
                for (const auto& item : itemset) {
                    outFile << item << " ";
                }
                outFile << "}, Support Count: " << count << "\n";
            }
        } else {
            outFile << "\nNo itemsets of size " << k << " meet the minimum support threshold.\n";
            // Stop processing further sizes since no itemsets meet the criteria
            break;
        }
    }

    outFile.close();
    std::cout << "Itemset support counts have been written to itemset_support_counts.txt\n";

    return 0;
}
```

Output:

code.cpp | itemset_support_counts.txt

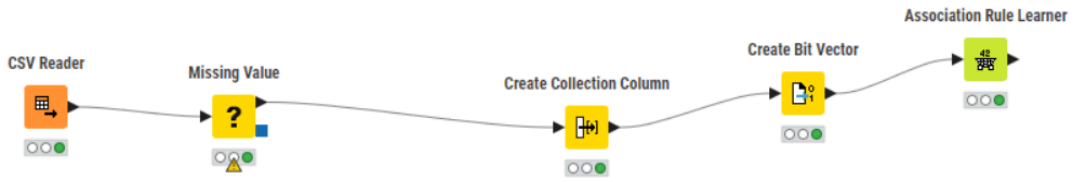
Frequent item set > itemset_support_counts.txt

```
Itemset Support Counts (min support = 3):

Itemsets of size 1:
{ A }, Support Count: 4
{ B }, Support Count: 4
{ C }, Support Count: 5
{ D }, Support Count: 5

Itemsets of size 2:
{ A C }, Support Count: 3
{ A D }, Support Count: 3
{ B C }, Support Count: 3
{ B D }, Support Count: 4
{ C D }, Support Count: 4

Itemsets of size 3:
{ B C D }, Support Count: 3

No itemsets of size 4 meet the minimum support threshold.
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
E:\DM KNIME\Frequent item set>code
Enter the path to your CSV file: apriori_dataset.csv
Enter the minimum support threshold: 3
Itemset support counts have been written to itemset_support_counts.txt

E:\DM KNIME\Frequent item set>
```

Knime:



Knime Output:

#	RowID	Support(0-1): *Number (double)*	Items *Set*
1	item ...	0.5	[A,C]
2	item ...	0.5	[A,D]
3	item ...	0.5	[B,D]
4	item ...	0.667	[A]
5	item ...	0.667	[C,D]
6	item ...	0.833	[C]
7	item ...	0.833	[D]

Aim : Find 5 no summary of dataset and obtain box plot

Code:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>
#include <algorithm>
#include <iomanip>
#include <stdexcept>

// Function to calculate quartiles and other statistics
void calculateStatistics(const std::vector<double>& sorted_data, double& Q1, double& Q2,
double& Q3, double& minimum, double& maximum, double& IQR, double& lower_bound,
double& upper_bound, std::vector<double>& outliers) {
    size_t n = sorted_data.size();

    // Calculate Q2 (Median)
    if (n % 2 == 0) {
        Q2 = (sorted_data[n / 2 - 1] + sorted_data[n / 2]) / 2;
    } else {
        Q2 = sorted_data[n / 2];
    }

    // Split data into lower and upper halves
    std::vector<double> lower_half(sorted_data.begin(), sorted_data.begin() + n / 2);
    std::vector<double> upper_half(sorted_data.begin() + (n + 1) / 2, sorted_data.end());

    // Calculate Q1 (lower half median)
    if (lower_half.size() % 2 == 0) {
        Q1 = (lower_half[lower_half.size() / 2 - 1] + lower_half[lower_half.size() / 2]) / 2;
    } else {
        Q1 = lower_half[lower_half.size() / 2];
    }

    // Calculate Q3 (upper half median)
    if (upper_half.size() % 2 == 0) {
        Q3 = (upper_half[upper_half.size() / 2 - 1] + upper_half[upper_half.size() / 2]) / 2;
    } else {
        Q3 = upper_half[upper_half.size() / 2];
    }

    minimum = sorted_data.front();
    maximum = sorted_data.back();
    IQR = Q3 - Q1;

    // Calculate whiskers (lower and upper bounds)
```

```
    lower_bound = Q1 - 1.5 * IQR;
    upper_bound = Q3 + 1.5 * IQR;

    // Identify outliers
    for (const auto& value : sorted_data) {
        if (value < lower_bound || value > upper_bound) {
            outliers.push_back(value);
        }
    }
}

// Function to read data from a CSV file for the specified column
std::vector<double> readColumnFromCSV(const std::string& file_path, const std::string&
column_name) {
    std::ifstream infile(file_path);
    std::string line;
    std::vector<double> data;
    std::vector<std::string> headers;

    // Read the header
    if (std::getline(infile, line)) {
        std::stringstream ss(line);
        std::string header;

        while (std::getline(ss, header, ',')) {
            headers.push_back(header);
        }
    }

    // Find the index of the specified column
    auto it = std::find(headers.begin(), headers.end(), column_name);
    if (it == headers.end()) {
        throw std::invalid_argument("Column name not found in the DataFrame.");
    }
    int column_index = std::distance(headers.begin(), it);

    // Read data from the specified column
    while (std::getline(infile, line)) {
        std::stringstream ss(line);
        std::string item;
        int current_index = 0;

        while (std::getline(ss, item, ',')) {
            if (current_index == column_index && !item.empty()) {
                try {
                    data.push_back(std::stod(item)); // Convert to double and add to the vector
                } catch (const std::invalid_argument&) {
                    // Ignore invalid numbers
```

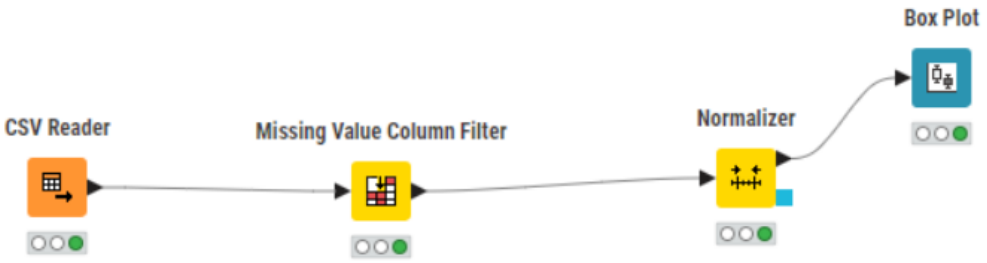
```
            }
        }
        current_index++;
    }
  }

  return data;
}

int main() {
  std::string file_path;
  std::string column_name;

  // User input for file path and column name
  std::cout << "Enter the path to your CSV file: ";
  std::getline(std::cin, file_path);
  std::cout << "Please enter the column name for the box plot: ";
  std::getline(std::cin, column_name);

  try {
    // Read data from the specified column
    std::vector<double> data = readColumnFromCSV(file_path, column_name);

    if (data.empty()) {
      std::cerr << "No valid data found in the specified column." << std::endl;
      return 1;
    }

    // Sort the data
    std::sort(data.begin(), data.end());

    // Calculate statistics
    double Q1, Q2, Q3, minimum, maximum, IQR, lower_bound, upper_bound;
    std::vector<double> outliers;
    calculateStatistics(data, Q1, Q2, Q3, minimum, maximum, IQR, lower_bound,
upper_bound, outliers);

    // Display results
    std::cout << "Statistics for column '" << column_name << "':\n";
    std::cout << "Min: " << minimum << "\n";
    std::cout << "Q1: " << Q1 << "\n";
    std::cout << "Median (Q2): " << Q2 << "\n";
    std::cout << "Q3: " << Q3 << "\n";
    std::cout << "Max: " << maximum << "\n";
    std::cout << "IQR: " << IQR << "\n";
    std::cout << "Lower Bound (Whisker): " << lower_bound << "\n";
    std::cout << "Upper Bound (Whisker): " << upper_bound << "\n";
    std::cout << "Outliers: ";
```

```
        for (const auto& outlier : outliers) {
            std::cout << outlier << " ";
        }
        std::cout << "\n";

    } catch (const std::invalid_argument& e) {
        std::cerr << "Error: " << e.what() << "\n";
    } catch (...) {
        std::cerr << "An unexpected error occurred.\n";
    }

    return 0;
}
```

Output:

```
E:\DM KNIME\box plot>code
Enter the path to your CSV file: input.csv
Please enter the column name for the box plot: sepal_length
Statistics for column 'sepal_length':
Min: 4.6
Q1: 4.9
Median (Q2): 6.1
Q3: 7.05
Max: 7.7
IQR: 2.15
Lower Bound (Whisker): 1.675
Upper Bound (Whisker): 10.275
```

Knime:



Knime Output:

Aim : Unsupervised Learning Methods - Cluster Analysis - partition based

Code :

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

double calculateMean(const vector<double>& points) {
    double sum = 0;
    for (double point : points) {
        sum += point;
    }
    return sum / points.size();
}

int findNearestCentroid(double value, const vector<double>& centroids) {
    int nearestCentroidIndex = 0;
    double minDistance = abs(value - centroids[0]);

    for (int i = 1; i < centroids.size(); i++) {
        double distance = abs(value - centroids[i]);
        if (distance < minDistance) {
            minDistance = distance;
            nearestCentroidIndex = i;
        }
    }
    return nearestCentroidIndex;
}

vector<double> getClusterPoints(const vector<double>& data, const vector<int>&
assignments, int clusterIndex) {
    vector<double> clusterPoints;
    for (int i = 0; i < data.size(); i++) {
        if (assignments[i] == clusterIndex) {
            clusterPoints.push_back(data[i]);
        }
    }
    return clusterPoints;
}

int main() {
    string csvFile = "data.csv";
    vector<double> weightValues;
```

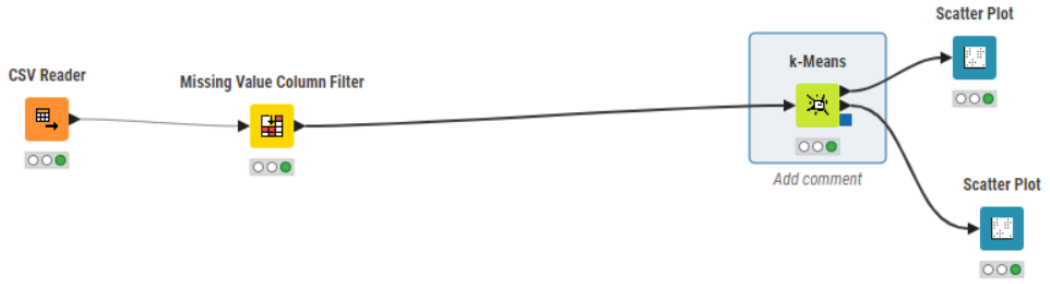
```
    ifstream file(csvFile);

    if (!file.is_open()) {
        cerr << "Error opening file." << endl;
        return 1;
    }

    string line;
    getline(file, line);

    while (getline(file, line)) {
        size_t comma = line.find(",");
        double weight = stod(line.substr(0, comma));
        weightValues.push_back(weight);
    }
    file.close();

    int k;
    cout << "Enter the number of clusters (K): ";
    cin >> k;

    vector<double> centroids(k);
    for (int i = 0; i < k; i++) {
        cout << "Enter initial centroid " << (i + 1) << ": ";
        cin >> centroids[i];
    }

    vector<int> clusterAssignments(weightValues.size());
    bool centroidsChanged = true;
    int iterations = 0;

    while (centroidsChanged && iterations < 100) {
        iterations++;
        centroidsChanged = false;

        // Assign points to the nearest centroid
        for (int i = 0; i < weightValues.size(); i++) {
            int nearestCentroidIndex = findNearestCentroid(weightValues[i], centroids);
            clusterAssignments[i] = nearestCentroidIndex;
        }

        // Update centroids
        for (int i = 0; i < k; i++) {
            vector<double> clusterPoints = getClusterPoints(weightValues, clusterAssignments,
i);
            if (!clusterPoints.empty()) {
                double newCentroid = calculateMean(clusterPoints);
                if (abs(centroids[i] - newCentroid) > 1e-4) {
```

```
                centroids[i] = newCentroid;
                centroidsChanged = true;
            }
        }
    }

    cout << "Iteration " << iterations << ": Centroids = ";
    for (double centroid : centroids) {
        cout << centroid << " ";
    }
    cout << endl;
}

cout << "Final Cluster Assignments:" << endl;
for (int i = 0; i < weightValues.size(); i++) {
    cout << "Value: " << weightValues[i] << " -> Cluster: " << clusterAssignments[i] << endl;
}

return 0;
}
```

Output:

```
E:\DM KNIME\Partitiion based>code
Enter the number of clusters (K): 2
Enter initial centroid 1: 28
Enter initial centroid 2: 35
Iteration 1: Centroids = 29.5 50.3333
Iteration 2: Centroids = 31.3333 58
Iteration 3: Centroids = 31.3333 58
Final Cluster Assignments:
Value: 28 -> Cluster: 0
Value: 35 -> Cluster: 0
Value: 31 -> Cluster: 0
Value: 63 -> Cluster: 1
Value: 53 -> Cluster: 1
```

Knime:



Knime Output:

#	RowID	ï»¿Marks *Number (integer)*	Cluster *String*
1	Row0	28	cluster_0
2	Row1	35	cluster_0
3	Row2	31	cluster_0
4	Row3	63	cluster_1
5	Row4	53	cluster_1

Weka Output:

Clusterer output

```
=== Clustering model (full training set) ===


kMeans
======

Number of iterations: 2
Within cluster sum of squared errors: 0.060952380952380945

Initial starting points (random):

Cluster 0: 63
Cluster 1: 35

Missing values globally replaced with mean/mode

Final cluster centroids:
                        Cluster#
Attribute    Full Data          0          1
                 (5.0)      (2.0)      (3.0)
=============================================
ï»¿marks             42         58    31.3333




Time taken to build model (full training data) : 0 seconds

=== Model and evaluation on training set ===

Clustered Instances

0       2 ( 40%)
1       3 ( 60%)
```

Aim : Unsupervised Learning Methods :Cluster Analysis - Density based

Code:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <sstream>

using namespace std;

void expandCluster(int pointIndex, const vector<int>& neighbors, vector<int>&
clusterAssignments,
                   int clusterId, const vector<double>& data, double epsilon, int minPts);

vector<int> getNeighbors(int index, const vector<double>& data, double epsilon);

int main() {
    string csvFile = "data.csv";
    vector<double> weightValues;
    string line;


    ifstream file(csvFile);
    if (!file.is_open()) {
        cerr << "Unable to open file" << endl;
        return 1;
    }
    getline(file, line);
    while (getline(file, line)) {
        stringstream ss(line);
        string weightStr;
        getline(ss, weightStr, ',');
        weightValues.push_back(stod(weightStr));
    }
    file.close();


    double epsilon;
    int minPts;
    cout << "Enter epsilon (ε): ";
    cin >> epsilon;
    cout << "Enter minimum points (minPts): ";
    cin >> minPts;

    vector<int> clusterAssignments(weightValues.size(), -1);
    int clusterId = 0;
```

```
    for (size_t i = 0; i < weightValues.size(); ++i) {
        if (clusterAssignments[i] == -1) {
            vector<int> neighbors = getNeighbors(i, weightValues, epsilon);

            if (neighbors.size() < minPts) {
                clusterAssignments[i] = -1;
            } else {
                ++clusterId;
                expandCluster(i, neighbors, clusterAssignments, clusterId, weightValues, epsilon,
minPts);
            }
        }
    }


    cout << "Results:" << endl;
    for (size_t i = 0; i < weightValues.size(); ++i) {
        if (clusterAssignments[i] == -1) {
            cout << "Value: " << weightValues[i] << " -> Noise" << endl;
        } else {
            cout << "Value: " << weightValues[i] << " -> Cluster " << clusterAssignments[i] <<
endl;
        }
    }

    return 0;
}

vector<int> getNeighbors(int index, const vector<double>& data, double epsilon) {
    vector<int> neighbors;
    for (size_t j = 0; j < data.size(); ++j) {
        if (fabs(data[index] - data[j]) <= epsilon) {
            neighbors.push_back(j);
        }
    }
    return neighbors;
}

void expandCluster(int pointIndex, const vector<int>& neighbors, vector<int>&
clusterAssignments,
                   int clusterId, const vector<double>& data, double epsilon, int minPts) {
    clusterAssignments[pointIndex] = clusterId;

    vector<int> expandedNeighbors = neighbors;

    for (size_t i = 0; i < expandedNeighbors.size(); ++i) {
        int neighborIndex = expandedNeighbors[i];
```

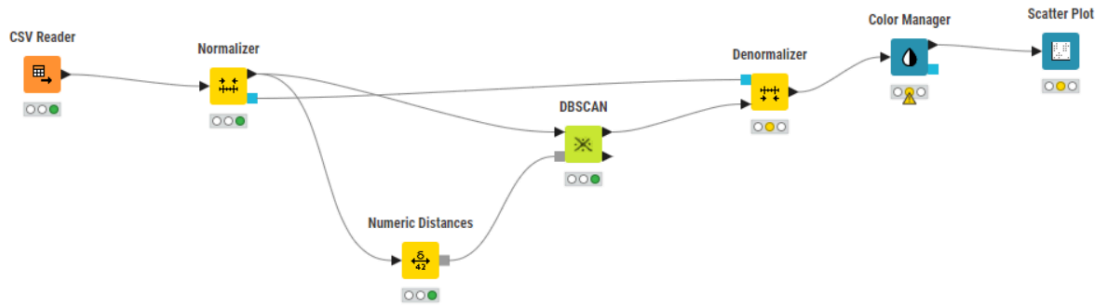
```
        if (clusterAssignments[neighborIndex] == -1) {
            clusterAssignments[neighborIndex] = clusterId;
        }

        if (clusterAssignments[neighborIndex] == 0) {
            clusterAssignments[neighborIndex] = clusterId;

            vector<int> neighborNeighbors = getNeighbors(neighborIndex, data, epsilon);
            if (neighborNeighbors.size() >= minPts) {
                expandedNeighbors.insert(expandedNeighbors.end(), neighborNeighbors.begin(),
neighborNeighbors.end());
            }
        }
    }
}
```

Code Output:

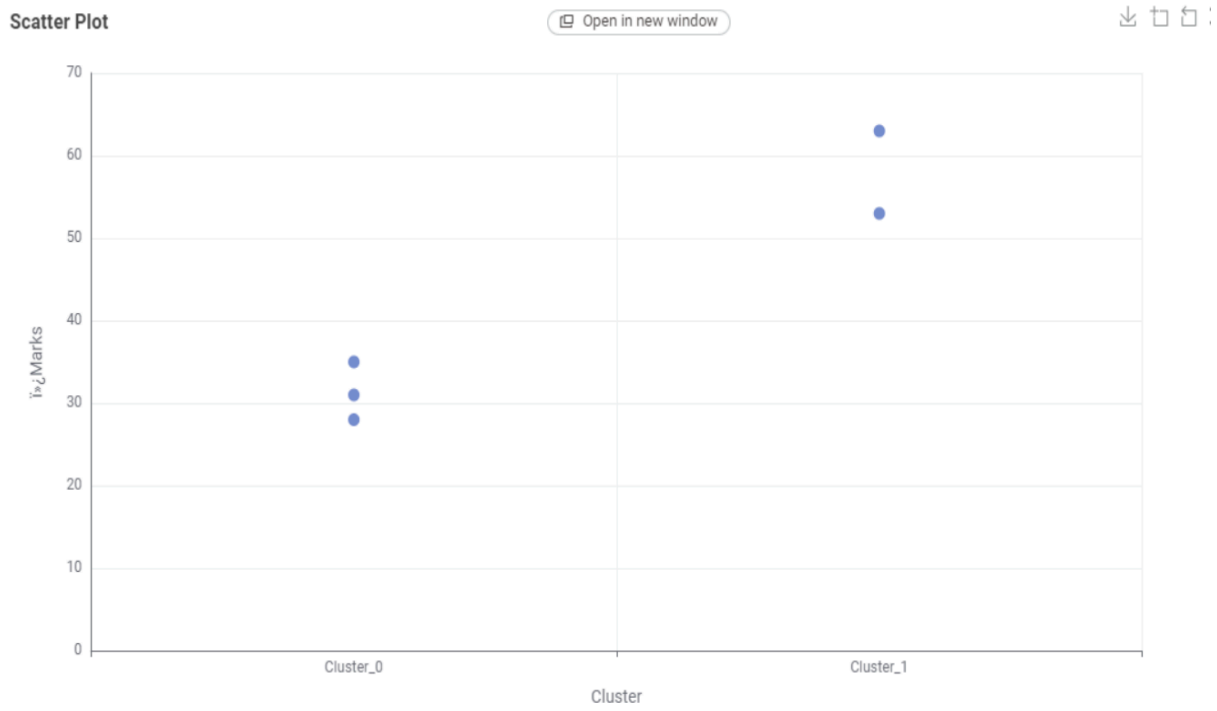
```
E:\DM KNIME\dbscan>code
Enter epsilon (╬): 10
Enter minimum points (minPts): 2
Results:
Value: 28 -> Cluster 1
Value: 35 -> Cluster 1
Value: 31 -> Cluster 1
Value: 63 -> Cluster 2
Value: 53 -> Cluster 2

E:\DM KNIME\dbscan>
```

ut:

Knime:



Knime Output:



Weka Output

Clusterer output

```
Missing values globally replaced with mean/mode

Final cluster centroids:
                        Cluster#
Attribute    Full Data          0          1
                 (5.0)      (2.0)      (3.0)
=============================================
ï»¿Marks            42         58    31.3333



Fitted estimators (with ML estimates of variance):

Cluster: 0 Prior probability: 0.4286

Attribute: ï»¿Marks
Normal Distribution. Mean = 58 StdDev = 5

Cluster: 1 Prior probability: 0.5714

Attribute: ï»¿Marks
Normal Distribution. Mean = 31.3333 StdDev = 2.8674


Time taken to build model (full training data) : 0 seconds
```

:

Aim : Unsupervised Learning methods : Cluster analysis : Hierarchical based

Code:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

double singleLinkage(const vector<double>& c1, const vector<double>& c2);
double completeLinkage(const vector<double>& c1, const vector<double>& c2);
double averageLinkage(const vector<double>& c1, const vector<double>& c2);

int main() {
    string file = "data.csv";
    vector<double> weights;
    int choice;

    cout << "Choose Linkage Method:" << endl;
    cout << "1. Single Linkage" << endl;
    cout << "2. Complete Linkage" << endl;
    cout << "3. Average Linkage" << endl;
    cin >> choice;


    ifstream infile(file);
    if (!infile.is_open()) {
        cerr << "Unable to open file" << endl;
        return 1;
    }

    string line;
    getline(infile, line);
    while (getline(infile, line)) {
        string::size_type sz;
        weights.push_back(stod(line, &sz));
    }
    infile.close();
```

```
vector<vector<double>> clusters;
for (double v : weights) {
    clusters.push_back({v});
}


while (clusters.size() > 1) {
    double minDist = numeric_limits<double>::max();
    int idx1 = -1, idx2 = -1;

    for (size_t i = 0; i < clusters.size(); ++i) {
        for (size_t j = i + 1; j < clusters.size(); ++j) {
            double dist = 0;

            switch (choice) {
                case 1:
                    dist = singleLinkage(clusters[i], clusters[j]);
                    break;
                case 2:
                    dist = completeLinkage(clusters[i], clusters[j]);
                    break;
                case 3:
                    dist = averageLinkage(clusters[i], clusters[j]);
                    break;
                default:
                    cout << "Invalid choice. Exiting." << endl;
                    return 1;
            }

            if (dist < minDist) {
                minDist = dist;
                idx1 = i;
                idx2 = j;
            }
        }
    }

    cout << "Merging clusters with minimum distance: " << minDist << endl;
    clusters[idx1].insert(clusters[idx1].end(), clusters[idx2].begin(), clusters[idx2].end());
    clusters.erase(clusters.begin() + idx2);


    cout << "Merged clusters: " << endl;
    for (const auto& cluster : clusters) {
        for (double v : cluster) {
            cout << v << " ";
```

```
            }
            cout << endl;
        }
    }


    cout << "Final Cluster: " << endl;
    for (double v : clusters[0]) {
        cout << v << " ";
    }
    cout << endl;

    return 0;
}

double singleLinkage(const vector<double>& c1, const vector<double>& c2) {
    double minDist = numeric_limits<double>::max();
    for (double v1 : c1) {
        for (double v2 : c2) {
            double dist = fabs(v1 - v2);
            if (dist < minDist) {
                minDist = dist;
            }
        }
    }
    return minDist;
}

double completeLinkage(const vector<double>& c1, const vector<double>& c2) {
    double maxDist = numeric_limits<double>::min();
    for (double v1 : c1) {
        for (double v2 : c2) {
            double dist = fabs(v1 - v2);
            if (dist > maxDist) {
                maxDist = dist;
            }
        }
    }
    return maxDist;
}

double averageLinkage(const vector<double>& c1, const vector<double>& c2) {
    double totalDist = 0.0;
    int count = 0;
    for (double v1 : c1) {
        for (double v2 : c2) {
            totalDist += fabs(v1 - v2);
            count++;
```

```
        }
    }
    return totalDist / count;
}
```

Code Output:

1. **Single Linkage**

```
Choose Linkage Method:
1. Single Linkage
2. Complete Linkage
3. Average Linkage
1
Merging clusters with minimum distance: 3
Merged clusters:
28 31
35
63
53
Merging clusters with minimum distance: 4
Merged clusters:
28 31 35
63
53
Merging clusters with minimum distance: 10
Merged clusters:
28 31 35
63 53
Merging clusters with minimum distance: 18
Merged clusters:
28 31 35 63 53
Final Cluster:
28 31 35 63 53

E:\DM KNIME\hierar>
```

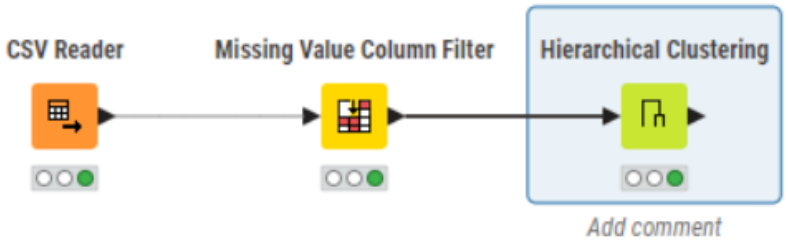
2. **Complete Linkage:**

```
Choose Linkage Method:
1. Single Linkage
2. Complete Linkage
3. Average Linkage
2
Merging clusters with minimum distance: 3
Merged clusters:
28 31
35
63
53
Merging clusters with minimum distance: 7
Merged clusters:
28 31 35
63
53
Merging clusters with minimum distance: 10
Merged clusters:
28 31 35
63 53
Merging clusters with minimum distance: 35
Merged clusters:
28 31 35 63 53
Final Cluster:
28 31 35 63 53
```

3. **Average Linkage**

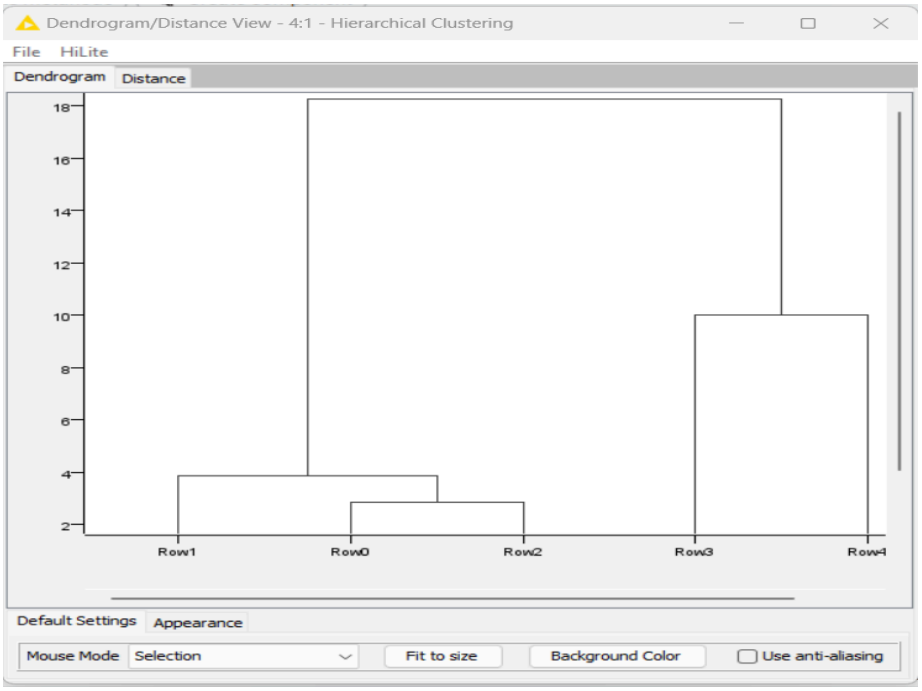
```
Choose Linkage Method:
1. Single Linkage
2. Complete Linkage
3. Average Linkage
3
Merging clusters with minimum distance: 3
Merged clusters:
28 31
35
63
53
Merging clusters with minimum distance: 5.5
Merged clusters:
28 31 35
63
53
Merging clusters with minimum distance: 10
Merged clusters:
28 31 35
63 53
Merging clusters with minimum distance: 26.6667
Merged clusters:
28 31 35 63 53
Final Cluster:
28 31 35 63 53

E:\DM KNIME\hierar>
```

Knime:

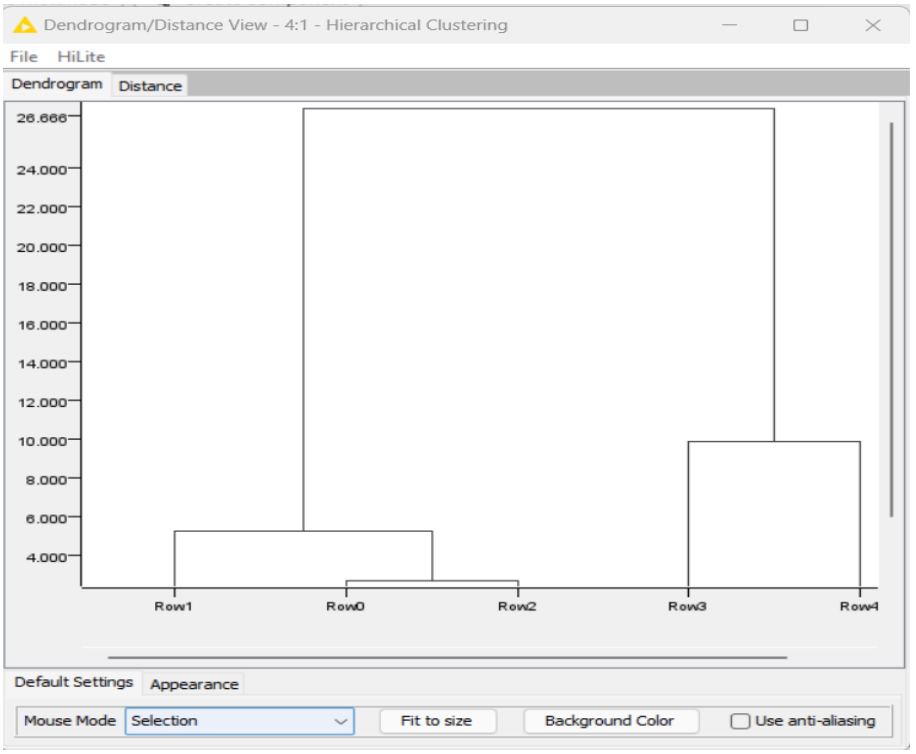


Knime Output:

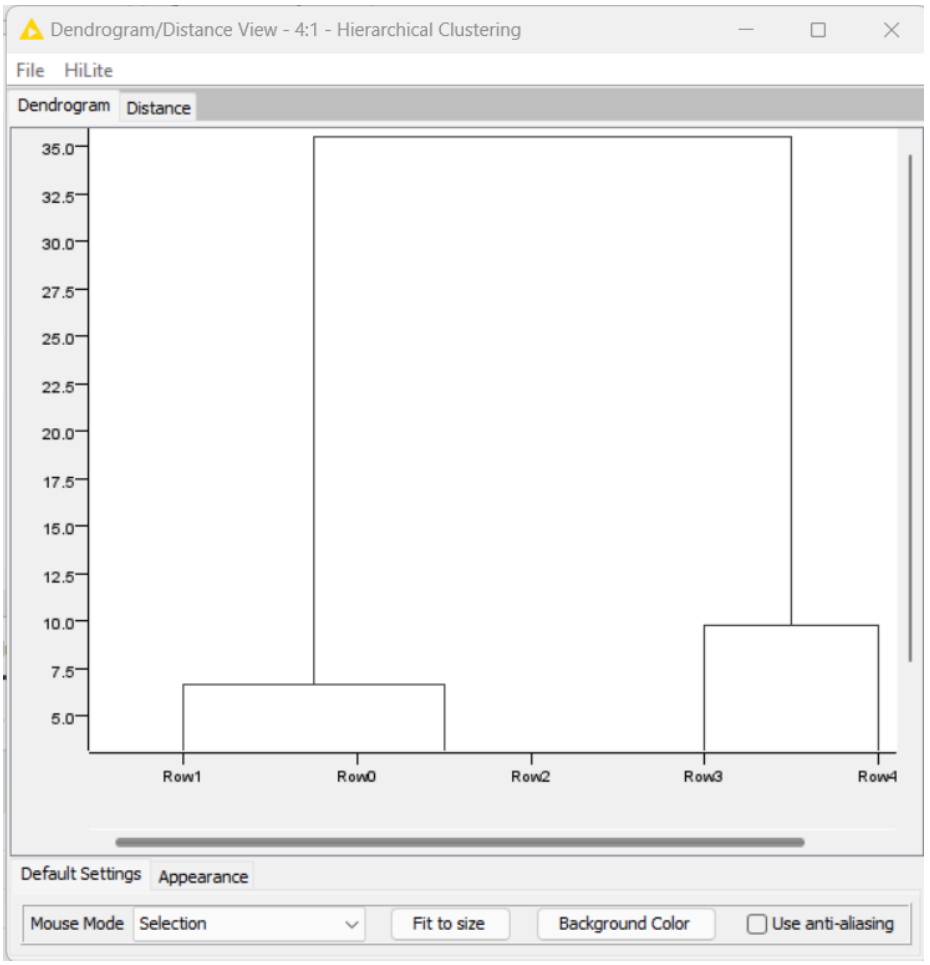
1. **Single Linkage**



2. Average Linkage



3. Complete Linkage



Weka Output

```
=== Run information ===

Scheme:       weka.clusterers.HierarchicalClusterer -N 2 -L SINGLE -P -A "weka.core.EuclideanDistance -R first-last"
Relation:     k_means_data
Instances:    5
Attributes:   1
              i»¿marks
Test mode:    evaluate on training data


=== Clustering model (full training set) ===

Cluster 0
((28.0:0.08571,31.0:0.08571):0.02857,35.0:0.11429)

Cluster 1
(63.0:0.28571,53.0:0.28571)



Time taken to build model (full training data) : 0 seconds

=== Model and evaluation on training set ===

Clustered Instances

0      3 ( 60%)
1      2 ( 40%)
```