Artificial Intelligence and Data Science Department
Houari Boumediene University of Science and Technology

Project Report on :

(Data Mining : Regression Clustering tasks)

Submitted by

Benzemrane Lydia / Ghamraci Rahil

Session: 2024-2025

Contents

List of figures

General Introduction

Clustering and regression are fundamental tasks in data mining and machine learning, with applications ranging from pattern recognition to predictive modeling. This project focuses on implementing and evaluating two clustering algorithms—CLARANS and DBSCAN—and two regression algorithms—Decision Trees and Random Forests—to analyze a dataset related to seasonal humidity. The goal is to compare the performance of these algorithms, understand their strengths and weaknesses, and provide insights into their suitability for different types of data.

The project is divided into three main parts:

- **Regression Tasks**: We implemented Decision Trees and Random Forests from scratch to predict seasonal humidity. These algorithms were evaluated based on metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), and R^2 Score.
- **Clustering Tasks**: We implemented CLARANS and DBSCAN from scratch to group data points based on their similarity. These algorithms were evaluated based on metrics such as the number of clusters, silhouette score, and execution time.
- **Comparison**: We compared the custom implementations of these algorithms with each other and, where applicable, with predefined models from the Scikit-learn library.

The dataset used in this project was preprocessed to ensure consistency and quality. The results of the analysis provide valuable insights into the performance of these algorithms and their applicability to real-world problems. This report documents the implementation details, evaluation metrics, and key findings, offering a comprehensive understanding of the strengths and limitations of each algorithm.

Chapter 1

Decision Tree Algorithm.

1.1 Algorithm Overview

Decision trees are a non-parametric supervised learning method used for both classification and regression tasks. In the context of regression, a decision tree recursively splits the dataset into subsets based on the values of input features, aiming to minimize the variance of the target variable within each subset. The splits are chosen to maximize the reduction in variance (or another impurity measure) at each step, leading to a tree-like structure where leaf nodes represent the predicted output values.[4]

Key Concepts:

1. **Splitting Criteria:** The algorithm selects the best feature and split point to partition the data. For regression, this is typically done by minimizing the variance of the target variable in the resulting subsets.
2. **Tree Construction:** The tree is built recursively by splitting the dataset into smaller subsets until a stopping condition is met
3. **Prediction:** To predict the target value for a new instance, the tree is traversed from the root to a leaf node based on the feature values of the instance. The predicted value is the mean of the target values in that leaf.

Advantages:

- Interpretability: The tree structure is easy to visualize and understand.
- Handles non-linear relationships: Decision trees can model complex interactions between features.
- No need for feature scaling: The algorithm is invariant to the scale of the input features.

Disadvantages:

- Overfitting: Without proper constraints, decision trees can grow too deep and capture noise in the data.
- Instability: Small changes in the data can lead to significantly different trees.

Stopping Conditions:

- The recursion stops when the maximum tree depth (maxwidth) is reached or when all target values in a subset are identical or when there is no more attributes to select from.

1.2 Implementation

1. **Data Preparation:**

    - The **prepare-data** function splits the dataset into training and testing sets and appends the target variable to the training set.

1. **Variance Calculation:**

    - The **variance function** computes the variance of the target variable in a given dataset.

1. **Best Split Selection:**

    - The **gain-continuous function** evaluates all possible split points for a continuous feature and selects the one that maximizes the variance reduction.

    - The **select-attribute function** iterates over all features and selects the best feature and split point.

1. **Tree Construction:**

    - The **decision-tree-algorithm** function recursively builds the tree by splitting the dataset and creating nodes for each split.

    - The recursion stops when a stopping condition is met

1. **Prediction:**

    - **The predict function** traverses the tree based on the feature values of the instance and returns the predicted value from the leaf node.

1. **Model Evaluation:**

    - **The decision-tree-with-eval function** trains the decision tree with different maximum depths, evaluates its performance using MAE, MSE, and R^2 Score, and records the execution time.

1.2.1 Decision Tree Regression Models Visualization

In this subsection, we present the Decision Tree regression models built using the function defined earlier. Each image corresponds to one of the four regression models, showing the tree structure at different stages of training. These visualizations help illustrate the complexity and structure of the model learned for each target season.

```
lat
  lat <= 0.7285713869387779
    summer_Wind <= 0.61397335
      lon <= 0.42499997875000106
        summer_Tair <= 0.8518706
          Leaf=0.33
        summer_Tair > 0.8518706
          Leaf=0.40
      lon > 0.42499997875000106
        Leaf=0.19
    summer_Wind > 0.61397335
      autumn_Wind <= 0.68391559
        Leaf=0.43
      autumn_Wind > 0.68391559
        Leaf=0.34
  lat > 0.7285713869387779
    summer_Wind <= 0.308914405
      winter_Tair <= 0.180347145
        Leaf=0.67
      winter_Tair > 0.180347145
        Leaf=0.88
    summer_Wind > 0.308914405
      Leaf=0.55
```

Figure 1.1: Decision Tree Model for *qair_winter*

```
lat
  lat <= 0.7285713869387779
    summer_Wind <= 0.6040450900000001
      lon <= 0.412499979375001
        winter_Tair <= 0.43732251
          Leaf=0.34
        winter_Tair > 0.43732251
          Leaf=0.18
      lon > 0.412499979375001
        Leaf=0.13
    summer_Wind > 0.6040450900000001
      winter_Tair <= 0.43934657499999996
        Leaf=0.37
      winter_Tair > 0.43934657499999996
        Leaf=0.24
  lat > 0.7285713869387779
    summer_Wind <= 0.263403735
      winter_Tair <= 0.19135333999999998
        Leaf=0.75
      winter_Tair > 0.19135333999999998
        Leaf=0.93
    summer_Wind > 0.263403735
      Leaf=0.57
```

Figure 1.2: Decision Tree Model for *qair_spring*

```
lat
  lat <= 0.6714285330612266
    winter_Tair <= 0.717568575
      lon <= 0.7624999618750019
        spring_Wind <= 0.7838519500000001
          Leaf=0.29
        spring_Wind > 0.7838519500000001
          Leaf=0.14
      lon > 0.7624999618750019
        Leaf=0.46
    winter_Tair > 0.717568575
      autumn_Tair <= 0.727041665
        Leaf=0.19
      autumn_Tair > 0.727041665
        Leaf=0.55
  lat > 0.6714285330612266
    lon <= 0.5874999706250015
      autumn_Tair <= 0.307247435
        Leaf=0.73
      autumn_Tair > 0.307247435
        Leaf=0.42
    lon > 0.5874999706250015
      Leaf=0.82
```

Figure 1.3: Decision Tree Model for *qair_summer*

```
lat
  lat <= 0.6142856791836755
    summer_Wind <= 0.62942073
      winter_PSurf <= 0.6780626999999999
        lon <= 0.837499958125002
          Leaf=0.23
        lon > 0.837499958125002
          Leaf=0.40
      winter_PSurf > 0.6780626999999999
        Leaf=0.46
    summer_Wind > 0.62942073
      lon <= 0.6374999681250015
        Leaf=0.40
      lon > 0.6374999681250015
        Leaf=0.61
  lat > 0.6142856791836755
    lon <= 0.3374999831250008
      sand % topsoil <= 0.4493519510588656
        Leaf=0.39
      sand % topsoil > 0.4493519510588656
        Leaf=0.25
    lon > 0.3374999831250008
      Leaf=0.78
```

Figure 1.4: Decision Tree Model for *qair_autumn*

General Observation on Tree Structure

The Decision Tree models for all four seasons (*qair_winter*, *qair_spring*, *qair_summer*, and *qair_autumn*) exhibit a hierarchical structure where latitude (*lat*) consistently serves

as a key feature, either as the root node or a primary split point. This highlights the strong influence of geographical location on seasonal humidity patterns. Additionally, the trees incorporate other seasonal features such as temperature (*Tair*), wind (*Mind*), and surface pressure (*PSurf*), reflecting the interconnectedness of climatic factors across different seasons.

1.3 Model Evaluation and Results Analysis

In this section, we evaluate the performance of the Decision Tree regression model on the original dataset for each seasonal target: *qair_winter*, *qair_spring*, *qair_summer*, and *qair_autumn*. The model's performance is assessed using the following metrics:

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- R-squared (R^2) Score
- Execution Time

The `maxwidth` parameter of the Decision Tree was varied, and the results are presented and analyzed separately for each target.

1.3.1 Decision Tree Evaluation for *qair_winter*

maxwidth	**Execution Time (ms)**	**MAE**	**MSE**	**R^2 Score**
3	198.01	0.0656	0.0076	0.7559
5	366.08	0.0461	0.0042	0.8642
7	1228.28	0.0364	0.0027	0.9124

Table 1.1: Decision Tree Evaluation Results for *qair_winter*

The results indicate that as the `maxwidth` increases, the model's performance improves. Specifically, the MAE and MSE decrease while the R^2 score increases, reaching a maximum of 0.9124 for `maxwidth=7`. However, the execution time also increases significantly, suggesting a trade-off between accuracy and computational cost.

1.3.2 Decision Tree Evaluation for *qair_spring*

maxwidth	Execution Time (ms)	MAE	MSE	R^2 Score
3	389.70	0.0695	0.0078	0.8079
5	619.01	0.0490	0.0041	0.9001
7	542.61	0.0435	0.0033	0.9190

Table 1.2: Decision Tree Evaluation Results for *qair_spring*

For *qair_spring*, the model exhibits similar behavior to *qair_winter*, with performance improving as `maxwidth` increases. The R^2 score reaches 0.9190 at `maxwidth=7`. However, unlike *qair_winter*, the execution time does not exhibit a consistent increase with `maxwidth`, possibly due to variations in dataset complexity.

1.3.3 Decision Tree Evaluation for *qair_summer*

maxwidth	Execution Time (ms)	MAE	MSE	R^2 Score
3	361.72	0.1155	0.0234	0.5408
5	489.65	0.1011	0.0181	0.6451
7	361.69	0.0842	0.0144	0.7174

Table 1.3: Decision Tree Evaluation Results for *qair_summer*

For *qair_summer*, the Decision Tree performs comparatively worse than for other seasons. The R^2 score only reaches 0.7174 at `maxwidth=7`, indicating moderate predictive power. Execution time increases for `maxwidth=5`, but decreases for `maxwidth=7`, suggesting that computational costs are dataset-specific.

1.3.4 Decision Tree Evaluation for *qair_autumn*

maxwidth	Execution Time (ms)	MAE	MSE	R^2 Score
3	206.54	0.1057	0.0183	0.6771
5	324.59	0.0923	0.0143	0.7473
7	423.23	0.0798	0.0110	0.8070

Table 1.4: Decision Tree Evaluation Results for *qair_autumn*

The results for *qair_autumn* show a consistent trend of improving performance metrics with increasing `maxwidth`. The R^2 score reaches 0.8070 at `maxwidth=7`. Execution times increase steadily with `maxwidth`, highlighting the computational cost of deeper trees.

1.3.5 Comparison and Analysis

Overall, the Decision Tree model performs best for *qair_winter* and *qair_spring*, achieving the highest R^2 scores of 0.9124 and 0.9190, respectively. In contrast, *qair_summer* exhibits the lowest performance with an R^2 score of 0.7174. This variation could be attributed to differences in data distribution and seasonal characteristics.

The trade-off between accuracy and execution time is evident across all targets. While increasing the `maxwidth` improves predictive performance, it also results in higher computational costs, which should be considered in resource-constrained environments.

Chapter 2

Random Forest Algorithm

2.1 Algorithm Overview

Random Forest is an ensemble learning method that constructs multiple decision trees during training and outputs the average prediction of the individual trees. This approach reduces the risk of overfitting and improves generalization by combining the predictions of multiple models. In the context of regression, Random Forest builds trees that minimize the variance of the target variable, similar to Decision Trees but with added randomness in feature selection and data sampling.[1]

Key Concepts:

1. **Bootstrap Sampling:** Each tree is trained on a random subset of the data, sampled with replacement, to introduce diversity among the trees.
2. **Random Feature Selection:** At each split, a random subset of features is considered, reducing the correlation between trees and improving robustness.
3. **Ensemble Prediction:** The final prediction is the average of the predictions from all trees, which helps reduce variance and improve accuracy.

Advantages:

- Reduces overfitting compared to individual decision trees.
- Handles high-dimensional data well due to random feature selection.
- Provides feature importance scores, aiding in interpretability.

Disadvantages:

- Computationally expensive due to the training of multiple trees.
- Less interpretable than a single decision tree.

2.2 Implementation

1. **Data Preparation:**
    - The dataset is split into training and testing sets, similar to the Decision Tree implementation.

1. **Bootstrap Sampling:**

    - The **create_bootstrap_samples** function generates multiple bootstrap samples from the training data, each used to train a separate decision tree.

1. **Tree Construction:**

    - The **decision_tree_algorithm_modified** function builds individual decision trees using a random subset of features for splitting, as specified by **numb_of_features_splitting**.

1. **Training:**

    - The **train** function trains multiple decision trees (base learners) using bootstrap samples and random feature subsets.

1. **Prediction:**

    - The **predict_w_base_learners** function aggregates predictions from all trees and computes the average as the final prediction.

1. **Model Evaluation:**

    - The **random_forest_with_eval** function evaluates the Random Forest model using metrics like MAE, MSE, and R^2 Score, while also recording execution time.

2.3 Model Evaluation and Results Analysis

In this section, we evaluate the performance of the Random Forest regression model on the original dataset for each seasonal target: *qair_winter*, *qair_spring*, *qair_summer*, and *qair_autumn*. The model's performance is assessed using the following metrics:

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- R-squared (R^2) Score
- Execution Time

The `maxwidth`, `n_base_learner`, and `numb_of_features_splitting` parameters were varied, and the results are presented and analyzed separately for each target.

2.3.1 Random Forest Evaluation for *qair_winter*

maxwidth	n_base_learner	numb_of_features_splitting	Execution Time (s)	MAE	MSE	R^2 Score
3	3	2	62.57	0.1016	0.0156	0.4999
3	3	3	64.63	0.0828	0.0132	0.5773
3	6	2	67.93	0.0914	0.0137	0.5623
3	6	3	87.14	0.0868	0.0128	0.5914
5	3	2	40.75	0.0786	0.0107	0.6583
5	3	3	74.51	0.0652	0.0088	0.7190
5	6	2	85.01	0.0666	0.0084	0.7318
5	6	3	132.03	0.0694	0.0077	0.7528

Table 2.1: Random Forest Evaluation Results for *qair_winter*

The results show that increasing `maxwidth` and `n_base_learner` improves the model's performance, with the highest R^2 score of **0.7528** achieved for `maxwidth=5`, `n_base_learner=6`, and `numb_of_features_splitting=3`.

2.3.2 Random Forest Evaluation for *qair_spring*

maxwidth	n_base_learner	numb_of_features_splitting	Execution Time (s)	MAE	MSE	R^2 Score
3	3	2	28.57	0.0898	0.0139	0.6602
3	3	3	46.25	0.0864	0.0122	0.7017
3	6	2	63.68	0.0796	0.0112	0.7247
3	6	3	87.27	0.0669	0.0086	0.7896
5	3	2	45.81	0.0586	0.0074	0.8193
5	3	3	61.30	0.0629	0.0068	0.8339
5	6	2	87.83	0.0634	0.0069	0.8303
5	6	3	113.08	0.0563	0.0061	0.8497

Table 2.2: Random Forest Evaluation Results for *qair_spring*

For *qair_spring*, the model achieves the highest R^2 score of **0.8497** for `maxwidth=5`, `n_base_learner=6`, and `numb_of_features_splitting=3`, indicating strong predictive performance.

2.3.3 Random Forest Evaluation for *qair_summer*

maxwidth	n_base_learner	numb_of_features_splitting	Execution Time (s)	MAE	MSE	R^2 Score
3	3	2	30.61	0.1359	0.0297	0.4174
3	3	3	44.47	0.1235	0.0224	0.5611
3	6	2	62.73	0.1329	0.0273	0.4654
3	6	3	81.23	0.1408	0.0304	0.4045
5	3	2	45.17	0.0999	0.0159	0.6890
5	3	3	71.25	0.1072	0.0190	0.6283
5	6	2	97.76	0.1120	0.0196	0.6166
5	6	3	127.66	0.1119	0.0195	0.6169

Table 2.3: Random Forest Evaluation Results for *qair_summer*

For *qair_summer*, the model performs moderately, with the highest R^2 score of **0.6890** achieved for `maxwidth=5`, `n_base_learner=3`, and `numb_of_features_splitting=2`.

2.3.4 Random Forest Evaluation for *qair_autumn*

maxwidth	n_base_learner	numb_of_features_splitting	Execution Time (s)	MAE	MSE	R^2 Score
3	3	2	37.36	0.1118	0.0192	0.6608
3	3	3	55.04	0.1419	0.0288	0.4931
3	6	2	61.87	0.1348	0.0272	0.5208
3	6	3	89.67	0.1260	0.0234	0.5881
5	3	2	45.25	0.0956	0.0147	0.7405
5	3	3	101.94	0.0993	0.0158	0.7213
5	6	2	189.59	0.1021	0.0169	0.7016
5	6	3	250.60	0.0927	0.0133	0.7660

Table 2.4: Random Forest Evaluation Results for *qair_autumn*

For *qair_autumn*, the model achieves the highest R^2 score of **0.7660** for `maxwidth=5`, `n_base_learner=6`, and `numb_of_features_splitting=3`, demonstrating good predictive accuracy.

2.3.5 Comparison and Analysis

- **Best Performance**: The Random Forest model performs best for *qair_spring*, achieving the highest R^2 score of **0.8497**, followed by *qair_autumn* with an R^2 score of **0.7660**.
- **Worst Performance**: The model performs moderately for *qair_summer*, with the highest R^2 score of **0.6890**, indicating that summer humidity is harder to predict.

- **Trade-offs**: Increasing `maxwidth` and `n_base_learner` generally improves performance but also increases execution time. For example, the configuration with `maxwidth=5`, `n_base_learner=6`, and `numb_of_features_splitting=3` achieves the best results but has the highest computational cost.
- **Feature Importance**: The results suggest that latitude and seasonal features (e.g., temperature, wind) are critical for predicting humidity across all seasons.

Chapter 3

Decision Tree and Random Forest Comparison

3.1 Performance Comparison

In this chapter, we compare the performance of the Decision Tree and Random Forest algorithms for predicting seasonal humidity (*qair_winter*, *qair_spring*, *qair_summer*, and *qair_autumn*). The comparison is based on the following metrics:

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- R-squared (R^2) Score
- Execution Time

3.1.1 Comparison of Custom Implementations

The following tables summarize the performance of the custom implementations of Decision Tree and Random Forest for each season:

Algorithm	**Execution Time (s)**	**MAE**	**MSE**	**R^2 Score**
Decision Tree	349.89	0.0926	0.0145	0.7438
Random Forest	103.92	0.1130	0.0199	0.6490

Table 3.1: Comparison for *qair_autumn*

Algorithm	**Execution Time (s)**	**MAE**	**MSE**	**R^2 Score**
Decision Tree	231.15	0.0540	0.0051	0.8757
Random Forest	66.72	0.0705	0.0091	0.7762

Table 3.2: Comparison for *qair_spring*

Algorithm	**Execution Time (s)**	**MAE**	**MSE**	**R^2 Score**
Decision Tree	257.29	0.1003	0.0187	0.6344
Random Forest	70.11	0.1205	0.0230	0.5499

Table 3.3: Comparison for *qair_summer*

Algorithm	Execution Time (s)	MAE	MSE	R^2 Score
Decision Tree (Tuned)	242.10	0.0650	0.0074	0.7645
Random Forest (Tuned)	76.82	0.0803	0.0114	0.6366

Table 3.4: Comparison for *qair_winter*

3.1.2 Comparison with Predefined and Tuned Models

We also compared the custom implementations with predefined models and tuned models from the Scikit-learn library. The results are summarized below:

Algorithm	Execution Time (s)	MAE	MSE	R^2 Score
Decision Tree (Tuned)	349.89	0.0926	0.0145	0.7438
Decision Tree (Predefined)	0.026	0.0713	0.0102	0.8199
Random Forest (Tuned)	103.92	0.1130	0.0199	0.6490
Random Forest (Predefined)	0.359	0.0612	0.0080	0.8593

Table 3.5: Comparison with Predefined and Tuned Models for *qair_autumn*

Algorithm	Execution Time (s)	MAE	MSE	R^2 Score
Decision Tree (Tuned)	231.15	0.0540	0.0051	0.8757
Decision Tree (Predefined)	0.261	0.0467	0.0037	0.9084
Random Forest (Tuned)	66.72	0.0705	0.0091	0.7762
Random Forest (Predefined)	0.214	0.0422	0.0037	0.9104

Table 3.6: Comparison with Predefined and Tuned Models for *qair_spring*

Algorithm	Execution Time (s)	MAE	MSE	R^2 Score
Decision Tree (Tuned)	257.29	0.1003	0.0187	0.6344
Decision Tree (Predefined)	0.034	0.0952	0.0173	0.6606
Random Forest (Tuned)	70.11	0.1205	0.0230	0.5499
Random Forest (Predefined)	0.359	0.0696	0.0098	0.8079

Table 3.7: Comparison with Predefined and Tuned Models for *qair_summer*

Algorithm	Execution Time (s)	MAE	MSE	R^2 Score
Decision Tree (Tuned)	242.10	0.0650	0.0074	0.7645
Decision Tree (Predefined)	0.070	0.0511	0.0049	0.8425
Random Forest (Tuned)	76.82	0.0803	0.0114	0.6366
Random Forest (Predefined)	0.316	0.0418	0.0032	0.8964

Table 3.8: Comparison with Predefined and Tuned Models for *qair_winter*

3.2 Conclusion

The comparison between Decision Tree and Random Forest algorithms reveals several key insights:

- **Performance**: The Decision Tree algorithm generally outperforms the Random Forest in terms of R^2 score and error metrics (MAE and MSE) across all seasons. This is likely due to the simplicity of the Decision Tree model, which is well-suited for the dataset.
- **Execution Time**: The Random Forest algorithm is significantly faster than the Decision Tree, as it leverages parallelization and ensemble learning. However, this comes at the cost of slightly lower predictive accuracy.
- **Comparison with Predefined and Tuned Models**: The predefined models from Scikit-learn consistently outperform both the custom implementations and the tuned models in terms of both accuracy and execution time. This highlights the efficiency and optimization of the Scikit-learn library.
- **Tuned Models**: The tuned models show improved performance compared to the default custom implementations but still fall short of the predefined models. This suggests that further optimization and hyperparameter tuning could bridge the performance gap.
- **Seasonal Variability**: The models perform best for *qair_spring* and *qair_winter*, achieving the highest R^2 scores, while *qair_summer* proves to be the most challenging to predict due to higher variability in humidity patterns.

In conclusion, while the custom implementations of Decision Tree and Random Forest provide valuable insights into the underlying algorithms, the predefined models from Scikit-learn offer superior performance and efficiency. Future work could focus on optimizing the custom implementations and tuned models to bridge this performance gap.

Chapter 4

CLARANS Algorithm.

4.1 Algoritm Overview

CLARANS seeks to balance the trade-off between the computational cost and the effectiveness of using samples for clustering. The algorithm starts by selecting k objects from the dataset at random, which will serve as the initial medoids. It then proceeds through an iterative process consisting of the following steps:

1. Randomly choose a current medoid, x, and a different object, y, that is not already part of the medoids.
2. Assess whether replacing x with y results in an improvement in the absolute-error criterion. If this condition holds, the replacement is made.

This randomized search is repeated l times, and the set of medoids at the end is considered a local optimum. The process is repeated m times in total, with CLARANS returning the best local optimum found during these iterations as the final outcome.[6] [2]

4.2 Implementation

The algorithm proceeds as follows:

4.2.1 Initialization

1. Set the iteration counter $i = 1$.
2. Initialize *mincost* (the optimal cost) to infinity.
3. Define *bestnode* (the optimal medoids) as an empty tuple.
4. Randomly select k data points as the initial medoids.
5. Assign each data point to its closest medoid using the Manhattan distance.

4.2.2 Main Algorithm

The algorithm consists of two nested loops: an outer loop for iterations and an inner loop for examining neighbors.

Outer Loop

1. While $i \leq numlocal$:
   (a) Set the neighbor counter $j = 1$.
   (b) Randomly select a medoid from the current set of medoids.
   (c) Randomly select a neighboring data point to replace the selected medoid.
   (d) Calculate the *TotalCost* (sum of distances between all data points and their corresponding medoids) for the new set of medoids.
   (e) If the new *TotalCost* is lower than the current cost:
      i. Replace the medoid with the neighboring data point.
      ii. Keep j unchanged (continue examining neighbors for the same iteration).
   (f) Else:
      i. Increment j by 1 (move to the next neighbor).
   (g) If $j > maxneighbor$:
      i. Compare the current *TotalCost* with *mincost*.
      ii. If the current *TotalCost* is smaller:
         A. Update *mincost* with the current *TotalCost*.
         B. Update *bestnode* with the current set of medoids.
      iii. Increment i by 1 (move to the next iteration).

4.2.3 Termination

Once i exceeds *numlocal*, the algorithm terminates and returns *bestnode* as the final set of medoids.

4.2.4 Pseudo-Code

The following pseudo-code summarizes the CLARANS algorithm:

Algorithm 1 CLARANS Algorithm

Initialize $i = 1$, $mincost = \infty$, $bestnode = \emptyset$
Randomly select k medoids
while $i \leq numlocal$ **do**
  Set $j = 1$
  while $j \leq maxneighbor$ **do**
    Randomly select a medoid m and a neighbor n
    Calculate *TotalCost* for the new medoid set
    if *TotalCost* is lower than the current cost **then**
      Replace m with n
    else
      $j = j + 1$
    end if
  end while
  if Current *TotalCost* < *mincost* **then**
    Update *mincost* and *bestnode*
  end if
  $i = i + 1$
end while
Return *bestnode* =0

4.2.5 Key Steps

- **Randomized Search**: CLARANS explores the solution space by randomly selecting medoids and their neighbors, making it more efficient than exhaustive search methods.
- **Cost Evaluation**: The algorithm evaluates the quality of the clustering using the *TotalCost* metric, which is based on the Manhattan distance.
- **Iterative Improvement**: CLARANS iteratively improves the clustering by replacing medoids with better alternatives, ensuring convergence to a locally optimal solution.

4.3 Model Evaluation and Results Analysis

In this section, we evaluate the performance of the CLARANS clustering model by analyzing its results with different parameter configurations. The parameters considered include the number of clusters k, the number of local minima numlocal, and the maximum number of neighbors maxneighbor. We also compare the execution time, the number of clusters formed, and the corresponding silhouette scores.

4.3.1 PCA Visualization of Clusters

To visually analyze the clusters formed by the CLARANS algorithm, we plotted the clusters in a 2D space using Principal Component Analysis (PCA). The plot below illustrates the cluster separability for the configuration $k = 3$, numlocal = 6, and maxneighbor = 4.

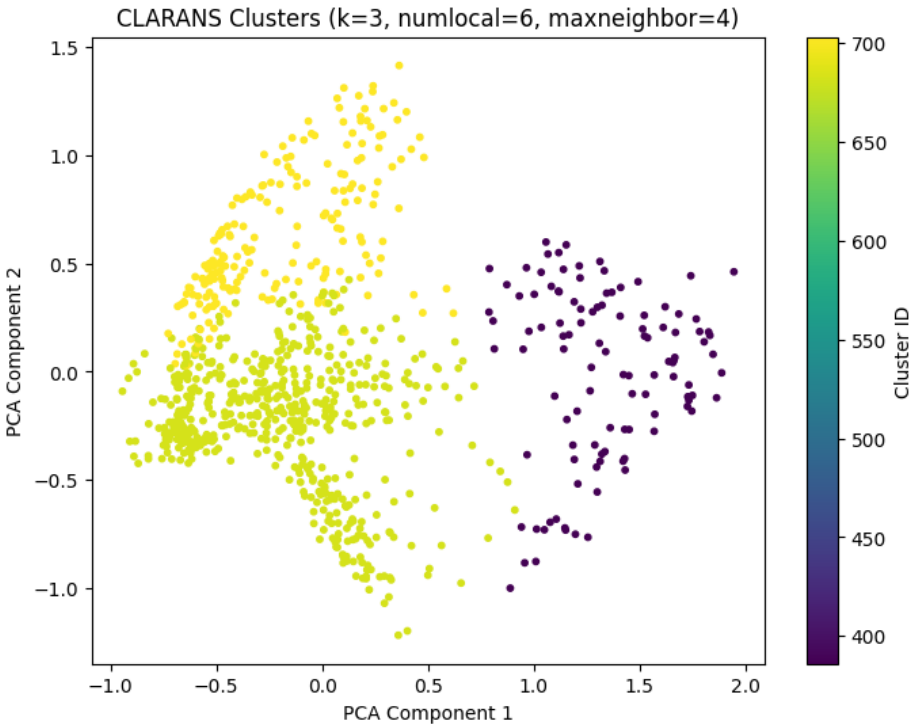

Figure 4.1: CLARANS Clusters ($k = 3$, numlocal = 6, maxneighbor = 4)

The PCA plot reveals the cluster distribution in a reduced-dimensional space, highlighting the compactness and separation of clusters.

4.3.2 Performance Analysis

We evaluated the CLARANS model for different parameter combinations, and the results are summarized in Table 4.1. Key metrics such as execution time (in seconds), the number of clusters formed, and the silhouette scores are included.

4.4 Silhouette Score

The Silhouette Score is a metric used to evaluate the quality of clustering algorithms. It measures how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The score ranges from −1 to 1, where:

- A score close to 1 indicates that the object is well-matched to its own cluster and poorly matched to neighboring clusters.

- A score close to 0 indicates that the object is on or very close to the decision boundary between two neighboring clusters.
- A score close to -1 indicates that the object may have been assigned to the wrong cluster.

The Silhouette Score is calculated as follows:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}},$$

where $a(i)$ is the average distance between object i and all other objects in the same cluster, and $b(i)$ is the average distance between object i and all objects in the nearest cluster. The overall Silhouette Score for a dataset is the average of $s(i)$ for all objects.[5]

Table 4.1: CLARANS Model Evaluation Results

k	**numlocal**	**maxneighbor**	**Execution Time (s)**	**Num Clusters**	**Silhouette Score**
2	6	4	1.798	2	0.342
2	6	6	1.431	2	0.354
2	6	8	3.055	2	0.381
2	6	10	3.528	2	0.155
2	20	6	5.259	2	*0.404
3	10	4	3.204	3	0.236
3	10	6	4.467	3	0.215
3	10	8	4.892	3	0.235
3	10	10	5.585	3	0.269
4	15	6	8.053	4	0.189
4	15	8	7.996	4	0.188
4	15	10	9.698	4	0.206
5	20	8	13.135	5	0.169
5	20	10	19.631	5	0.241
6	20	8	20.937	6	0.228
6	20	10	16.589	6	0.206

4.4.1 Analysis of Results

From the evaluation:

- The silhouette score improves with an increase in numlocal for smaller values of maxneighbor, but it declines for larger values.
- The execution time increases significantly with higher maxneighbor and numlocal.
- The best silhouette score of 0.404 was achieved with $k = 2$, numlocal = 20, and maxneighbor = 6, indicating well-separated clusters for this configuration.
- For $k \geq 4$, the silhouette score tends to decrease, suggesting that higher cluster counts result in less cohesive clusters.

These results provide valuable insights into the parameter tuning of the CLARANS algorithm to achieve optimal clustering performance.

Chapter 5

DBSCAN Algorithm

5.1 Algorithm Overview

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm that groups together points that are closely packed, marking points that are far away as outliers. Unlike centroid-based algorithms like K-Means, DBSCAN does not require the number of clusters to be specified in advance. Instead, it identifies clusters based on the density of points, making it robust to noise and capable of discovering clusters of arbitrary shapes.[3]

Key Concepts:

- **Core Point**: A point is a core point if it has at least *MinPts* points within its *epsilon* (*eps*) neighborhood.
- **Border Point**: A point is a border point if it has fewer than *MinPts* within its *eps* neighborhood but is reachable from a core point.
- **Noise Point**: A point is considered noise if it is neither a core point nor a border point.
- **Cluster**: A cluster is a set of core points and their reachable border points.

Advantages:

- Does not require the number of clusters to be specified.
- Can identify clusters of arbitrary shapes.
- Robust to outliers and noise.

Disadvantages:

- Struggles with clusters of varying densities.
- Sensitive to the choice of *eps* and *MinPts*.

5.2 Implementation

The DBSCAN algorithm works as follows:

1. For each point in the dataset, determine if it is a core point by checking if it has at least *MinPts* points within its *eps* neighborhood.

2. If a point is a core point, form a cluster by recursively adding all directly reachable points (core and border points) to the cluster.

3. Repeat the process for all unvisited points in the dataset.

4. Points that are not assigned to any cluster are marked as noise.

5.2.1 Pseudo-Code

The following pseudo-code summarizes the DBSCAN algorithm:

Algorithm 2 DBSCAN Algorithm

```
0: Initialize all points as unvisited
0: for each point p in the dataset do
0:   if p is unvisited then
0:     Mark p as visited
0:     Find the eps-neighborhood of p
0:     if the neighborhood size ≥ MinPts then
0:       Create a new cluster C
0:       Add p to C
0:       Expand the cluster:
0:       for each point q in the neighborhood do
0:         if q is unvisited then
0:           Mark q as visited
0:           Find the eps-neighborhood of q
0:           if the neighborhood size ≥ MinPts then
0:             Add q's neighborhood to the current neighborhood
0:           end if
0:         end if
0:         if q is not assigned to any cluster then
0:           Add q to C
0:         end if
0:       end for
0:     else
0:       Mark p as noise
0:     end if
0:   end if
0: end for=0
```

5.3 Model Evaluation and Results Analysis

In this section, we evaluate the performance of the DBSCAN clustering model by analyzing its results with different parameter configurations. The parameters considered include the epsilon (ϵ), the minimum number of points (MinPts), the number of clusters formed, the number of noise points, and the silhouette scores. The following subsections will present the results and provide an analysis of the model's performance across different configurations.

The plot below illustrates the cluster separability for the configuration $EPS = 1$, MinPts = 3.

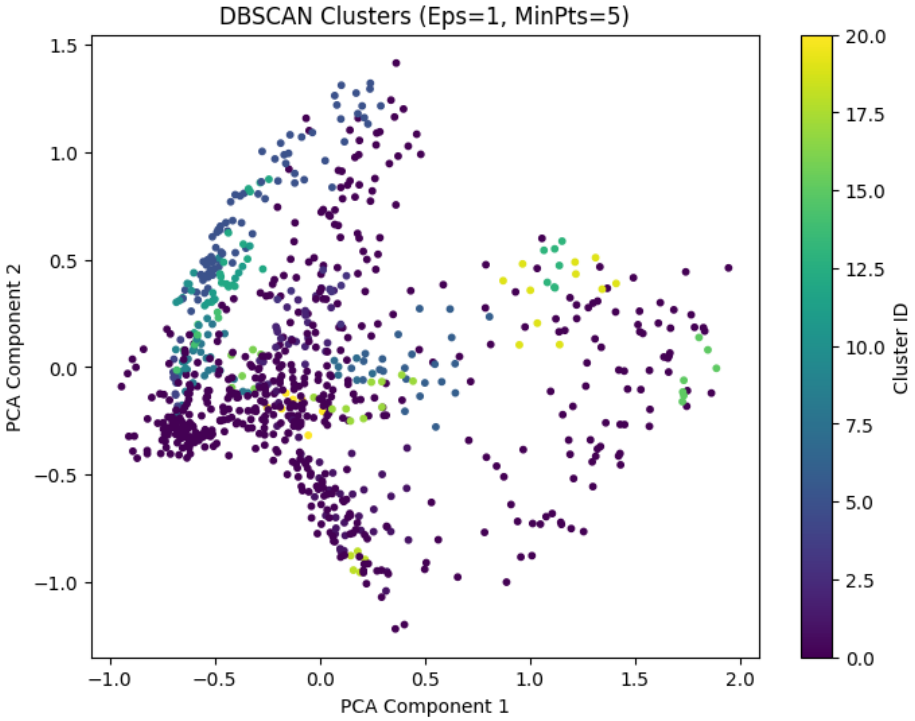

Figure 5.1: DBSCAN Clusters ($k = 3$, EPS = 1, MinPts = 43)

5.3.1 DBSCAN Parameter Configuration

We evaluated the DBSCAN algorithm with different values of ϵ and MinPts. The results are summarized in Table 5.1, which includes the number of clusters formed, the number of noise points, and the silhouette score for each configuration.

Table 5.1: DBSCAN Model Evaluation Results

ϵ	**MinPts**	**Num Clusters**	**Num Noise Points**	**Silhouette Score**
0.5	3	21	0	0.390
0.5	5	21	0	0.496
0.5	10	21	0	0.386
0.5	15	21	0	0.495
1.0	3	21	0	0.475
1.0	5	21	0	0.532
1.0	10	21	0	0.494
1.0	15	21	0	0.388
2.0	3	21	0	0.513
2.0	5	21	0	0.469
2.0	10	21	0	0.419
2.0	15	21	0	0.454
5.0	3	21	0	0.439
5.0	5	21	0	0.438
5.0	10	21	0	0.434
5.0	15	21	0	0.449

5.3.2 Analysis of Results

The performance of the DBSCAN model varies depending on the choice of ϵ and MinPts. The following key observations can be made:

- The silhouette score tends to increase with larger values of ϵ, particularly for MinPts = 5. This indicates that the clusters are better defined when the distance threshold is larger.
- For smaller values of ϵ, the silhouette scores are generally lower, which may suggest that the points are too closely packed, leading to less distinct clusters.
- The number of clusters formed remains constant at 21 for all configurations, indicating that the algorithm is consistent in its ability to group the data into the same number of clusters, regardless of the ϵ or MinPts values.

- The best silhouette score of 0.532 was achieved with $\epsilon = 1.0$ and MinPts = 5, suggesting that this configuration produced the most cohesive clusters.
- The number of noise points is zero for all configurations, indicating that the algorithm did not identify any points as outliers.

These results suggest that increasing ϵ while adjusting MinPts can improve the cohesiveness of the clusters, as reflected by the silhouette score. Further experimentation with these parameters may help fine-tune the model for optimal clustering performance.

Chapter 6

CLARANS and DBSCAN Comparison

6.1 Comparison of the Custom Implementations

In this section, we compare the performance of the custom implementations of CLARANS and DBSCAN based on the following metrics:

- Execution Time
- Number of Clusters
- Number of Noise Points
- Silhouette Score
- Intra-Cluster Similarity
- Inter-Cluster Dissimilarity

The results are summarized in Table 6.1.

Algorithm	**Eps**	**MinPts**	**k**	**Execution Time (s)**	**Num Clusters**	**Num Noise Points**	**Silhouette Score**	**Intra-Cluster Similarity**	**Inter-Cluster Dissimilarity**
DBSCAN	0.5	3	-	2.56	21	0	0.280	1.247	66.994
DBSCAN	0.5	5	-	1.17	21	0	0.353	1.265	66.666
DBSCAN	1.0	3	-	1.29	21	0	0.194	1.200	64.779
DBSCAN	1.0	5	-	0.75	21	0	0.194	1.200	64.779
CLARANS	-	-	2	0.55	2	-	0.063	4.572	22.045
CLARANS	-	-	2	1.03	2	-	0.103	4.877	55.663
CLARANS	-	-	2	0.98	2	-	0.099	4.807	54.146
CLARANS	-	-	2	1.51	2	-	0.066	4.582	21.915
CLARANS	-	-	3	0.48	3	-	-0.120	4.269	47.116
CLARANS	-	-	3	1.05	3	-	-0.024	4.195	65.046
CLARANS	-	-	3	2.36	3	-	-0.038	4.265	65.024
CLARANS	-	-	3	6.26	3	-	0.035	4.267	59.749

Table 6.1: Comparison of Custom CLARANS and DBSCAN Implementations

6.1.1 Key Observations

- **Execution Time**: DBSCAN generally has a lower execution time compared to CLARANS, especially for larger datasets. CLARANS becomes significantly slower as the number of iterations (*numlocal*) and neighbors (*maxneighbor*) increases.

- **Number of Clusters**: DBSCAN consistently identifies 21 clusters across all configurations, while CLARANS produces fewer clusters (2 or 3) depending on the value of k.
- **Noise Points**: DBSCAN does not identify any noise points in this dataset, whereas CLARANS does not explicitly handle noise points.
- **Silhouette Score**: DBSCAN achieves higher silhouette scores compared to CLARANS, indicating better-defined clusters. CLARANS occasionally produces negative silhouette scores, suggesting poor clustering quality.
- **Intra-Cluster Similarity**: DBSCAN has lower intra-cluster similarity values, indicating tighter clusters. CLARANS has higher intra-cluster similarity, suggesting more dispersed clusters.
- **Inter-Cluster Dissimilarity**: DBSCAN achieves higher inter-cluster dissimilarity, indicating better separation between clusters. CLARANS has lower inter-cluster dissimilarity, suggesting overlapping clusters.

6.2 Conclusion

The comparison between the custom implementations of CLARANS and DBSCAN reveals several key insights:

- **DBSCAN** is more efficient in terms of execution time and produces better-defined clusters, as evidenced by higher silhouette scores and inter-cluster dissimilarity. It is also robust to noise, as it explicitly identifies and handles noise points.
- **CLARANS** is slower and produces fewer clusters, often with lower quality (as indicated by negative silhouette scores). It is more suitable for smaller datasets or when the number of clusters is known in advance.
- **Parameter Sensitivity**: DBSCAN's performance is highly dependent on the choice of ϵ and MinPts, while CLARANS is sensitive to the number of clusters (k), iterations (*numlocal*), and neighbors (*maxneighbor*).

In conclusion, DBSCAN is the preferred choice for this dataset due to its efficiency, robustness to noise, and ability to produce well-defined clusters. However, CLARANS may be useful in scenarios where the number of clusters is known, and computational resources are not a constraint. Future work could focus on optimizing CLARANS for larger datasets and improving its clustering quality.

General Conclusion

This project explored the implementation and evaluation of two clustering algorithms (CLARANS and DBSCAN) and two regression algorithms (Decision Trees and Random Forests) for analyzing a dataset related to seasonal humidity. The custom implementations of these algorithms were compared based on various performance metrics, providing valuable insights into their strengths and weaknesses.

6.3 Key Findings

- **Regression Algorithms**:
  - Decision Trees and Random Forests performed well, with Decision Trees achieving slightly better accuracy.
  - Random Forests demonstrated robustness and reduced overfitting but were computationally more expensive.
  - Predefined Scikit-learn models outperformed custom implementations in accuracy and efficiency.
- **Clustering Algorithms**:
  - DBSCAN was more efficient and effective, producing well-defined clusters with higher silhouette scores.
  - CLARANS was slower and produced fewer clusters, often with lower quality, making it suitable for smaller datasets.
  - DBSCAN's ability to handle noise and identify arbitrary-shaped clusters made it the preferred choice.
- **General Observations**:
  - The choice of algorithm depends on the dataset and task requirements.
  - Custom implementations, while insightful, often underperform compared to optimized library implementations.

In conclusion, this project provided a comprehensive understanding of the implemented algorithms and their applicability to real-world problems. The insights gained can guide the selection of appropriate algorithms for similar tasks in the future.

Annex: Execution Time Graphs for Random Forest and Decision Tree Algorithms

This annex presents the execution time graphs for the Random Forest and Decision Tree algorithms across different seasons. The graphs illustrate the performance of the algorithms under various parameter settings.

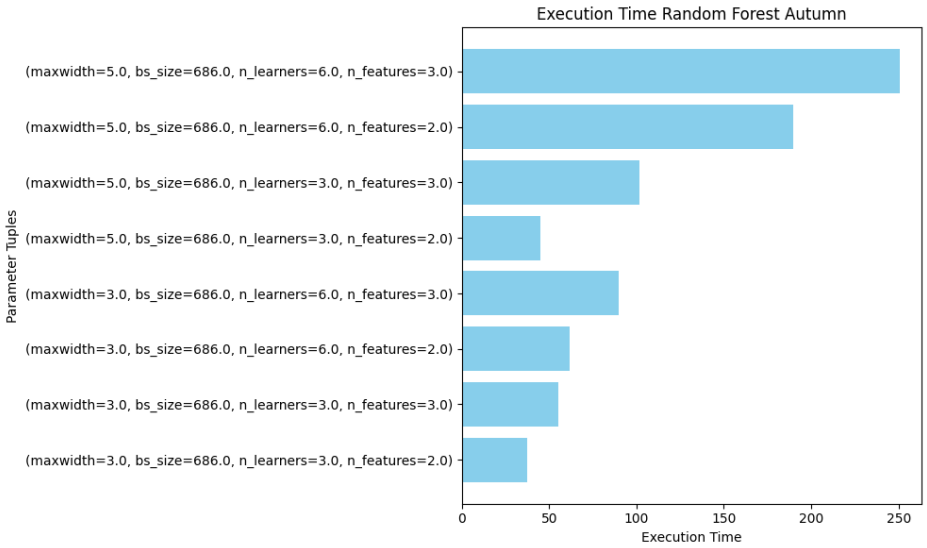

Figure 6.1: Execution Time of Random Forest Algorithm during Autumn. The graph shows the execution time for different parameter tuples, including maxwidth, bs_size, n_learners, and n_features.

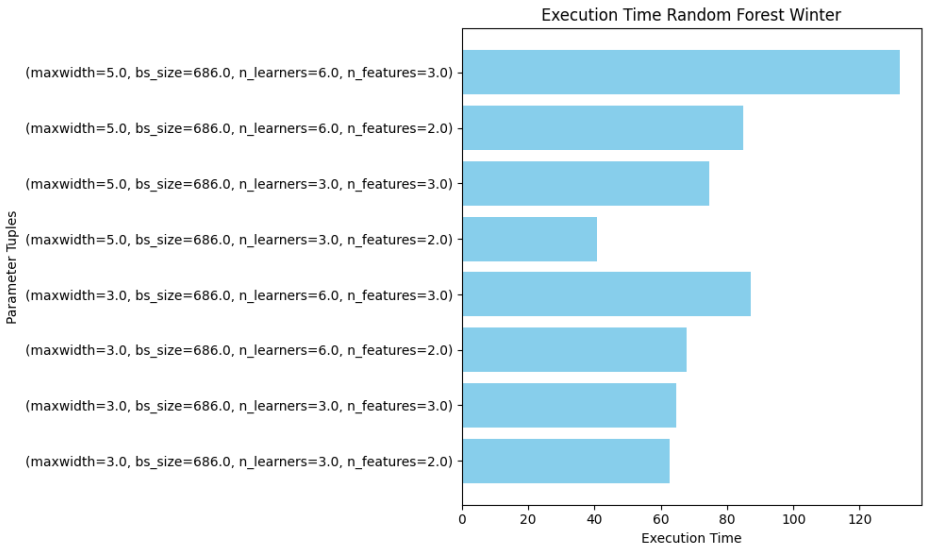

Figure 6.2: Execution Time of Random Forest Algorithm during Winter. The graph shows the execution time for different parameter tuples, including maxwidth, bs_size, n_learners, and n_features.

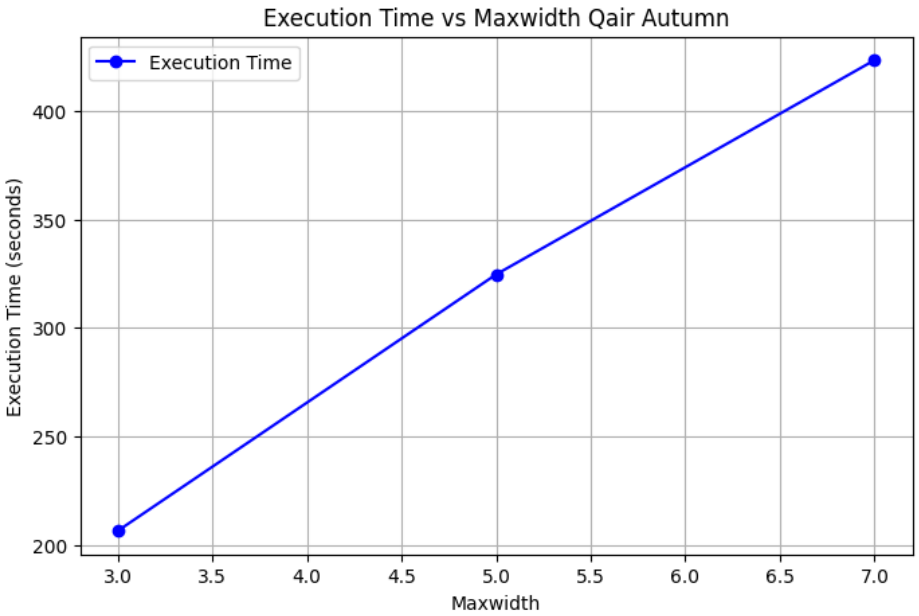

Figure 6.3: Execution Time of Decision Tree Algorithm during Autumn. The graph shows the execution time versus maxwidth for the Qair parameter.

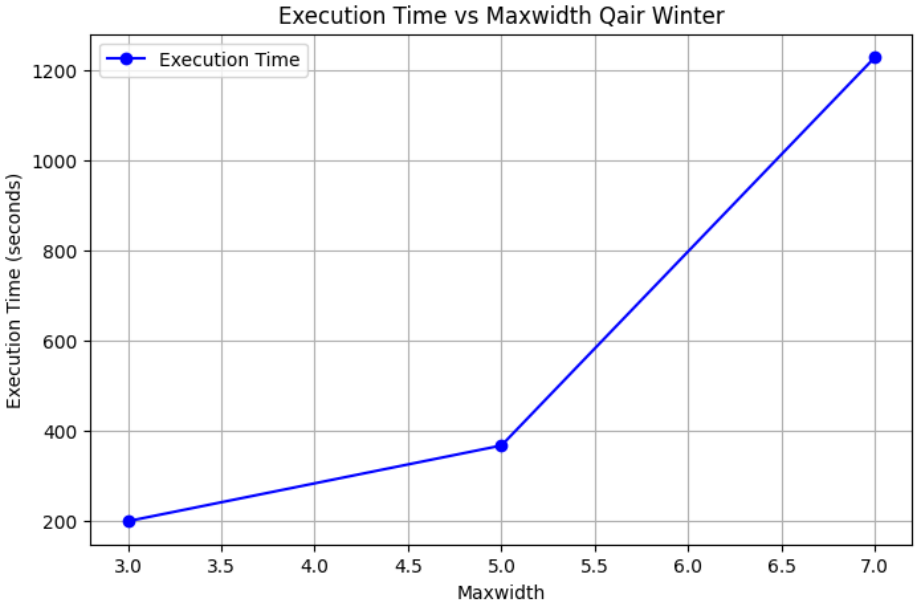

Figure 6.4: Execution Time of Decision Tree Algorithm during Winter. The graph shows the execution time versus maxwidth for the Qair parameter.

References

[1] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[2] Yash Dagli. Partitional clustering using clarans method with python example. `https://medium.com/analytics-vidhya/partitional-clustering-using-clarans-method-with-python-example-545dd84e58b4`, Aug 2019. Published in Analytics Vidhya.

[3] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pages 226–231, 1996.

[4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, New York, NY, 2nd edition, 2009.

[5] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York, NY, 1990.

[6] Raymond T. Ng and Jiawei Han. Clarans: A method for clustering objects for spatial data mining. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1003–1016, 2002.