Project report

Project Title: Hierarchical Agglomerative Clustering

Course: Machine Learning and Data Mining

Authors: Anna Miletova 89231151, Mariia Kozlova 89231113

Date: July 2025

GitHub repository of the project: https://github.com/miletovaa/ml-dm-project/

Abstract

This project implements an interactive visualization tool for hierarchical agglomerative clustering using both single and complete linkage methods. Users can input 2D points and observe how clusters are merged step-by-step through updated distance matrices and a resulting dendrogram.

Introduction

Hierarchical clustering is a popular unsupervised learning technique used to group similar data points into nested clusters. Unlike methods such as k-means, hierarchical clustering does not require specifying the number of clusters in advance. This project focuses on agglomerative clustering, where each point starts as its own cluster and is iteratively merged with the nearest one.

The developed tool allows users to:

- Add 2D data points
- Choose between single or complete linkage
- Generate step-by-step distance matrices (based on Manhattan distances)
- Visualize clustering in the form of a dendrogram
- Calculate the number of resulting clusters and their content

Theoretical Background

Hierarchical agglomerative clustering works by:

- Starting with all data points as individual clusters
- Merging the closest pair of clusters based on a linkage metric:
  - **Single linkage**: minimum distance between any two points in different clusters
  - **Complete linkage**: maximum distance between any two points in different clusters

This process continues until all points are merged into one cluster, forming a **dendrogram**.

Distance matrices are recalculated at each step to reflect the updated distances between new clusters.

- To determine the final clustering results (i.e., the number of clusters and their composition), a cutting point is calculated. This point is defined as the midpoint of the largest distance between any two merged clusters during the hierarchical process. All the clusters below the cutting point are taken as a result of the clustering.

Used technologies

The project was built using a modern and modular front-end stack designed for interactive data visualization and responsiveness:

- **React (TypeScript)**

The core framework used to build the user interface. React's component-based architecture allows for clean separation of logic, UI, and state management. TypeScript adds type safety and improves code reliability during development.

- **Recoil**

  A state management library for React that simplifies sharing state across components. In this project, Recoil was used to manage the global clustering state, including the list of points and linkage type.

- **D3.js**

  Used for rendering the dendrogram. D3 provides fine-grained control over SVG elements, making it ideal for drawing hierarchical structures like cluster trees with animated transitions.

- **Tailwind CSS**

  Tailwind allows utility-first styling directly in JSX for fast prototyping and consistent design.

- **npm (Node.js)**

  Package manager used to install and manage project dependencies. The development environment is initialized and run using npm.

Project structure

```
app/
├── src/
│   ├── components/
│   │   ├── Dendrogram.tsx
│   │   ├── DistanceMatrixPlot.tsx
│   │   ├── InsertNewPoint.tsx
│   │   ├── PointsScatterPlot.tsx
│   │   └── UserInputContainer.tsx
│   ├── types/
│   │   ├── dendrogram.ts
│   │   ├── linkageType.ts
│   │   ├── matrix.ts
│   │   └── point.ts
│   ├── utils/
│   │   ├── clusteringAlgorithm.ts
│   │   └── dendrogram.ts
│   ├── state.ts
│   ├── App.tsx
│   └── index.tsx
```

Key functions

1. `calculateInitialDistanceMatrix(points: Points): DistanceMatrix` (clusteringAlgorithm.ts)

   Input: An array of 2D points.

   ```
   type Point = {
     id: string
     x: number
     y: number
   }

   type Points = Point[]
   ```

   Output: A distance matrix where each cell represents the Manhattan distance between two points.

   ```
   type Distance = {
     id: string
     distance: number
   ```

```
}

type Point = {
    id: string
    distances: Distance[]
}

export type DistanceMatrix = Point[]
```

2. `getClosestPoints(matrix: DistanceMatrix): ClosestPoints` (clusteringAlgorithm.ts)

   Input: A DistanceMatrix.

   Output: The ids of the pair of points with the smallest non-zero distance.

   ```
   type ClosestPoints = {
       pointAId: string
       pointBId: string
       minDist: number
   }
   ```

3. `calculateNewDistanceMatrix(oldMatrix, closestPoints, linkageType): DistanceMatrix` (clusteringAlgorithm.ts)

   Input: oldMatrix -- previous distance matrix, closestPoints, linkageType.

   ```
   type LinkageType = "S" | "C" // S = Single, C = Complete
   ```

   Output: An updated distance matrix after merging the closest pair.

4. `clusteringAlgorithm(initialPoints, linkageType): { matrices, nodes }` (clusteringAlgorithm.ts)

   Input: initialPoints -- an array of Point objects, linkageType.

   Output: An object with:

   - matrices: Distance matrices for each clustering step
   - nodes: Tree nodes used for building the dendrogram

   ```
   type DendrogramNode = {
       index: string
       height: number
       isLeaf?: boolean
       children?: DendrogramNode[]
   }
   ```

5. `getClusters(dendrogram: DendrogramNode): { cutHeight: number, clusters: string[] }` (clusteringAlgorithm.ts)

   Input: dendrogram root node

   Output: cut height (point where we "cut" the dendrogram from which we distinguish the number and content of resulting clusters) and clusters (mentioned resulting clusters)

6. `getDendrogram(data: DendrogramNode, options?): SVGElement` (dendrogram.ts)

   Input: data -- final dendrogram root node, options: display settings (e.g., size, cut height)

   Output: An SVG element with the drawn dendrogram.

State management

Atom	Type	Description	Used In

`pointsState`	`Points[]`	Array of user-defined points	`InsertNewPoint`, `Scatter`, `UserInputContainer`
`linkageTypeState`	`"S"` or `"C"`	Clustering method	`UserInputContainer`

UI Components

InsertNewPoint.tsx

- Handles adding new points to global state (pointsState)
- UI: Two inputs (x, y) + "+" button

PointsScatterPlot.tsx

- Plots points on a Recharts scatter plot
- Dynamically shows/hides when there are no points

UserInputContainer.tsx

- Lets user choose linkage type and run clustering
- Shows results:
- Dendrogram (using <Dendrogram />)
- Distance matrices (using <DistanceMatrixPlot />)

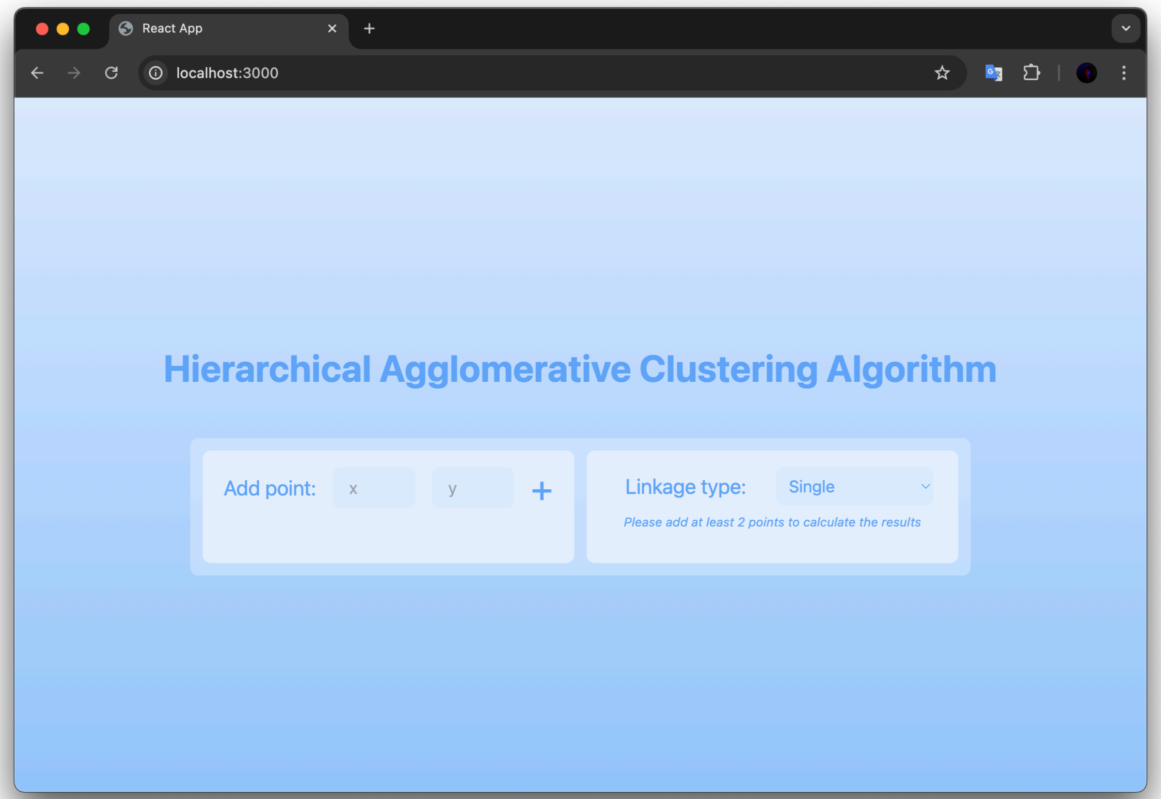
Dendrogram.tsx

- Uses getDendrogram() to render the final tree

DistanceMatrixPlot.tsx

- Renders a matrix table using <table> from matrix data

Project UI



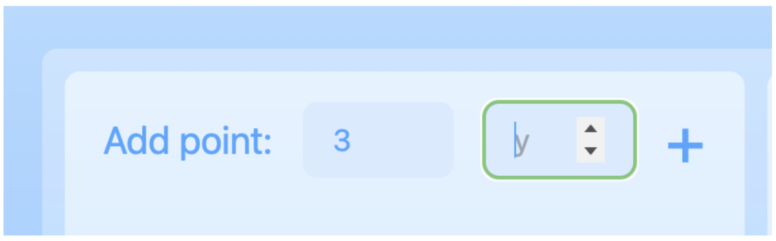
Hierarchical Agglomerative Clustering Algorithm

Linkage type: Single

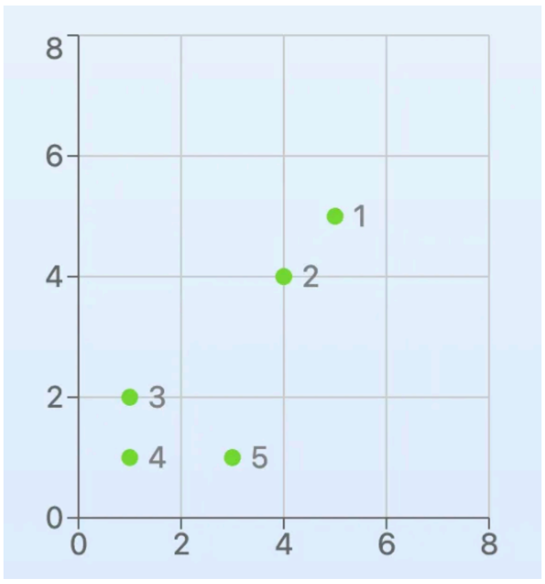
Please add at least 2 points to calculate the results

User interface of the application includes:

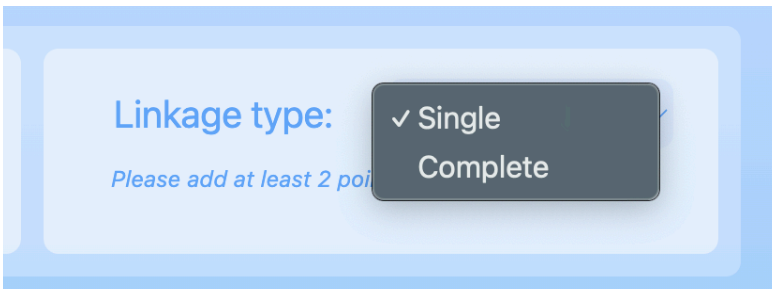
- Input of the 2D points with validation of non-positive values `(InsertNewPoint.tsx)`



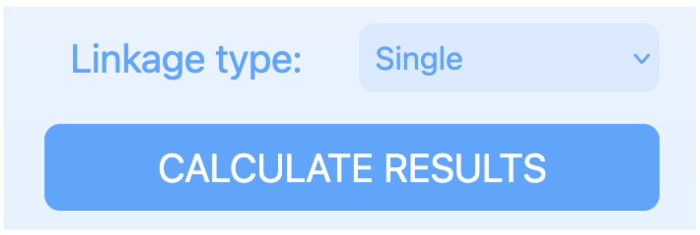
- Visualization of the inserted points `(PointsScatterPlot.tsx)`



- Select of the linkage type `(UserInputContainer.tsx)`



- Calculate button. `(UserInputContainer.tsx)` It is required to input at least 2 points to perform clustering. When the minimal number of points is entered, user can perform clustering by clicking “Calculate results” button.



Example of usage

The following dataset was used for the example.

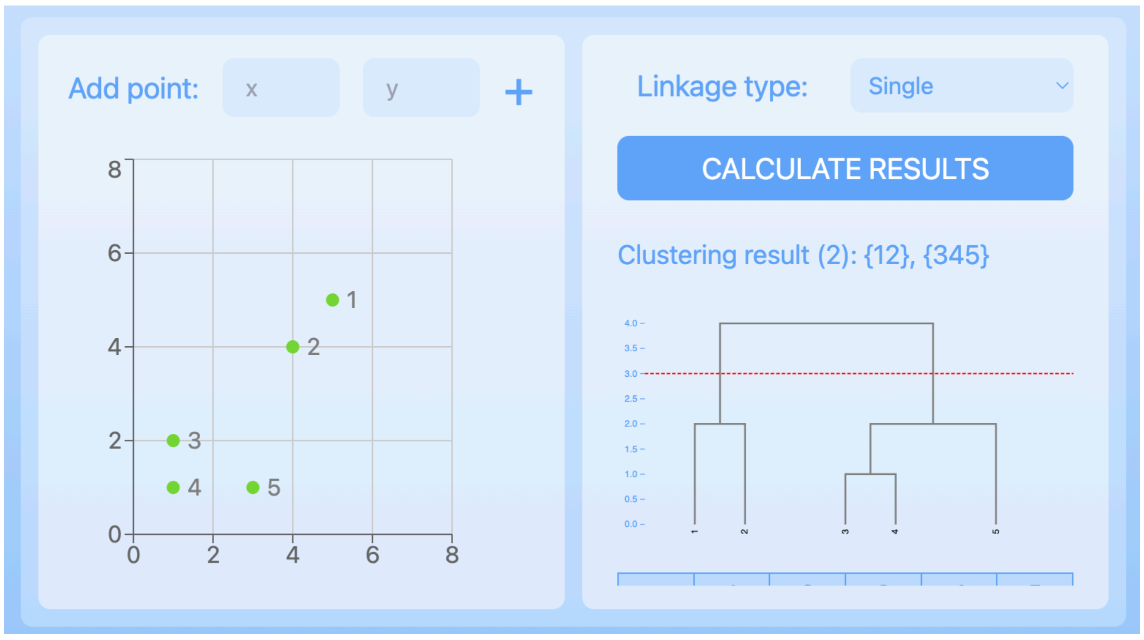
```
[
  { id: '1', x: 5, y: 5 },
  { id: '2', x: 4, y: 4 },
  { id: '3', x: 1, y: 2 },
  { id: '4', x: 1, y: 1 },
  { id: '5', x: 3, y: 1 }
]
```

After clustering was performed, user retrieves the results in a form of:

- Number of the resulting clusters below the cutting point

- Content of the clusters below the cutting point
- Distance matrixes for each step of the clustering process

Single linkage example:

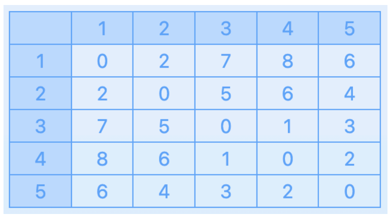


Add point:

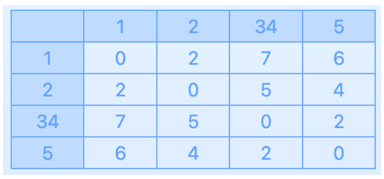
Linkage type: Single

CALCULATE RESULTS

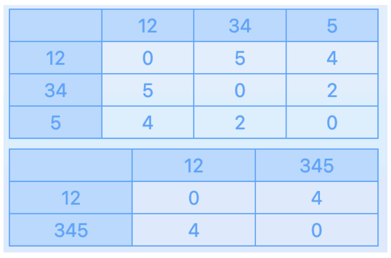
Clustering result (2): {12}, {345}



	1	2	3	4	5
1	0	2	7	8	6
2	2	0	5	6	4
3	7	5	0	1	3
4	8	6	1	0	2
5	6	4	3	2	0



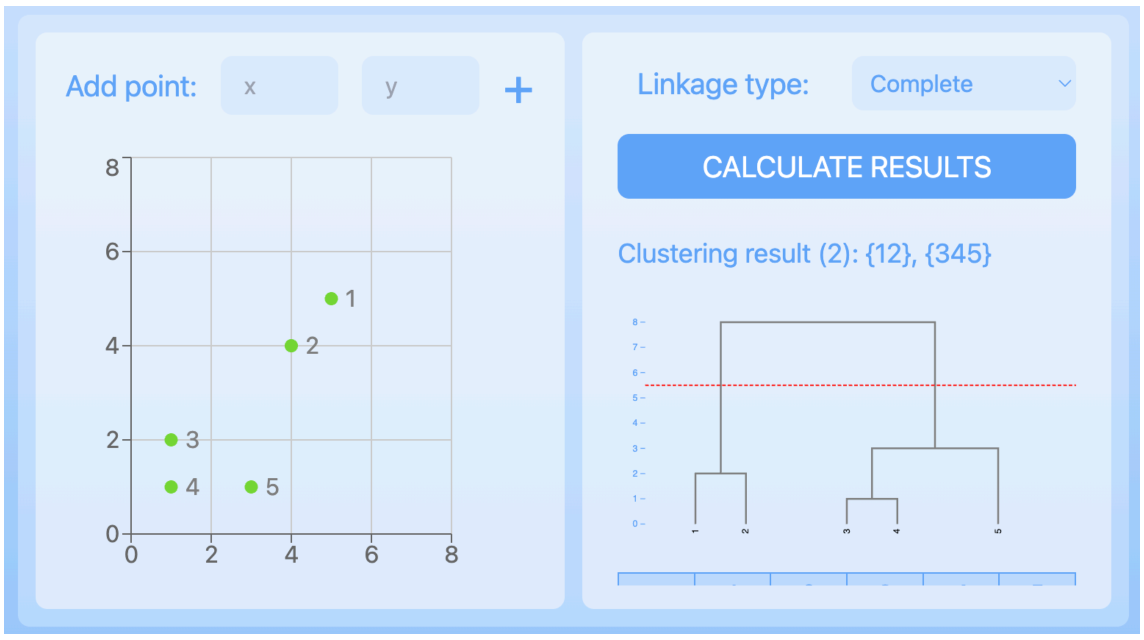
	1	2	34	5
1	0	2	7	6
2	2	0	5	4
34	7	5	0	2
5	6	4	2	0



	12	34	5
12	0	5	4
34	5	0	2
5	4	2	0

	12	345
12	0	4
345	4	0

Complete linkage example:

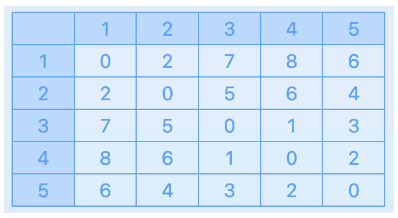


Add point:

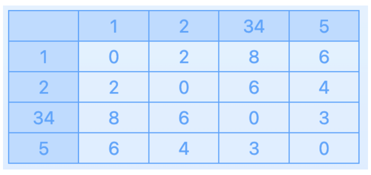
Linkage type: Complete

CALCULATE RESULTS

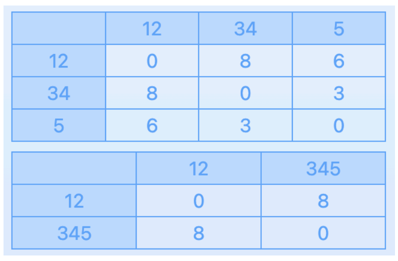
Clustering result (2): {12}, {345}



	1	2	3	4	5
1	0	2	7	8	6
2	2	0	5	6	4
3	7	5	0	1	3
4	8	6	1	0	2
5	6	4	3	2	0



	1	2	34	5
1	0	2	8	6
2	2	0	6	4
34	8	6	0	3
5	6	4	3	0



	12	34	5
12	0	8	6
34	8	0	3
5	6	3	0

	12	345
12	0	8
345	8	0

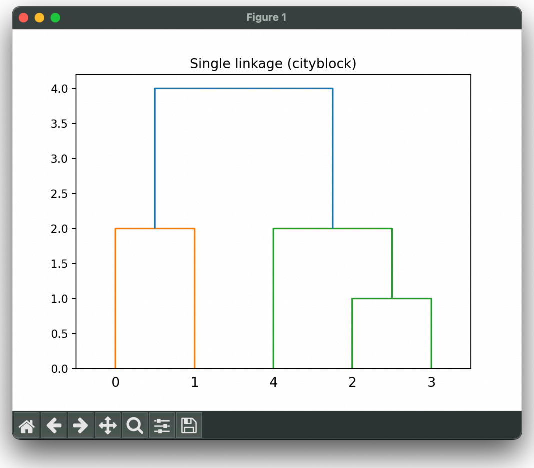
Testing and comparison with alternative libraries

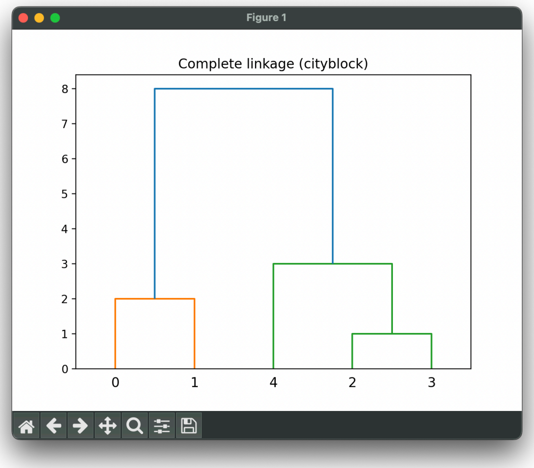
Testing was conducted by:

- Manually inputting predefined point sets
- Comparing calculated matrices with expected values
- Verifying correct cluster merges at each step
- Ensuring dendrogram structure matches clustering order
- Comparing the results of clustering with SciPy library (scipy.cluster.hierarchy)

Comparison with SciPy Clustering

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

data = [
    {'id': '1', 'x': 5, 'y': 5},
    {'id': '2', 'x': 4, 'y': 4},
    {'id': '3', 'x': 1, 'y': 2},
    {'id': '4', 'x': 1, 'y': 1},
    {'id': '5', 'x': 3, 'y': 1},
]

X = np.array([[d['x'], d['y']] for d in data], dtype=float)

## linkage method return array of a form:
## [cluster1, cluster2, distance, number of points in a new cluster]

Z_single = linkage(X, method='single', metric='cityblock')
Z_complete = linkage(X, method='complete', metric='cityblock')

print("Single linkage (cityblock):\n", Z_single)
# Output:
# Single linkage (cityblock):
#  [[2. 3. 1. 2.]
#  [0. 1. 2. 2.]
#  [4. 5. 2. 3.]
#  [6. 7. 4. 5.]]

print("\nComplete linkage (cityblock):\n", Z_complete)
# Output:
# Complete linkage (cityblock):
#  [[2. 3. 1. 2.]
#  [0. 1. 2. 2.]
#  [4. 5. 3. 3.]
#  [6. 7. 8. 5.]]

dendrogram(Z_single); plt.title('Single linkage (cityblock)'); plt.show()
dendrogram(Z_complete); plt.title('Complete linkage (cityblock)'); plt.show()
```





As it is visible from the data output and screenshots, the **results** of the HAC of our application and SciPy **are the same**, nevertheless out application has support of the calculation of the clusters and their content at the cutting point and presentation of all intermediate distance matrixes, that is not supported by the Python library.

Conclusion

In this project, we successfully implemented an interactive tool for hierarchical agglomerative clustering that combines educational clarity with functional depth. By allowing users to input 2D points and choose between single or complete linkage, the application provides a hands-on way to explore the clustering process in real time. The dynamic rendering of distance matrices and the dendrogram, along with automated cutting point detection, offers users both visual and analytical insight into how clusters are formed.

Our comparison with SciPy confirms the accuracy of the implementation, while our additional features—such as intermediate matrix visualization and cluster composition display—make this tool especially valuable for learning and debugging. Leveraging React, Recoil, D3.js, and Tailwind CSS, we created a modular, responsive, and intuitive interface that facilitates both understanding and experimentation.

Overall, this project not only deepened our understanding of clustering algorithms but also demonstrated how modern front-end technologies can be used to bring complex machine learning concepts to life in a transparent and engaging way.