Friedrich-Alexander-Universität
Technische Fakultät

Department of Computer Science
Computer Science 6
(Data Management)

Knowledge Discovery in Databases with Exercises
Summer Semester 2025

Submission 3: Clustering

About this Assignment

In this assignment, your task is to implement the algorithms for K-means and DBSCAN. For this purpose, you have access to a basic code skeleton, some helper classes, and several test cases.

Key Data

- **Max. Group Size:** 3
- **Max. Points:** 30
- **Estimated Workload:** 3 - 4 hours

How to Work on the Assignment

To start working on the assignment, you'll need to accept the assignment via GitHub Classroom by clicking the provided link. This will set up a new GitHub repository for your group, packed with all the necessary files for the assignment. If you're joining an existing group, it'll add you to that group's repository.[1]

Once that's done, you have two main options for working on your assignment. You can clone the repository[2] to your local machine by navigating to `Code` → `Local`, which allows you to work directly from your computer. Alternatively, you might prefer using GitHub Codespaces by selecting `Code` → `Codespaces` for a virtual online environment, complete with the ability to run Python through the `Terminal` provided.

Whichever method you choose, it's crucial to commit and push your changes back to the repository to submit your solution[2]. After your submission, GitHub Actions takes over to automatically grade your solution and provide feedback. You'll find this feedback in the `Actions` tab of your repository. If you didn't receive full points, you can improve your solution and push the changes back to the repository to trigger a reevaluation.

[1] Each student must join individually. You can join groups while accepting an assignment.
[2] If you're unfamiliar with Git or GitHub, check out this helpful guide: `https://github.com/git-guides/`

How to Prepare the Transfer the Points to StudOn

In addition to joining the GitHub Classroom, you also need to register your GitHub username on StudOn. This is necessary to transfer the points you've earned on GitHub to StudOn. To do this, enter your GitHub username in `Submission 3 - GitHub Username`. Make sure to enter your username correctly, as otherwise, the points cannot be transferred.

After submission deadline, the points you've earned on GitHub will be transferred to StudOn. This process is not immediate and may take a few days. If you have any questions or issues, please contact us via the StudOn forum.

Restrictions

Within the scope of your implementation, you are not permitted to modify the helper classes, the test cases, or the provided GitHub Actions.

This will be checked on a random basis, and any attempt to do so will result in zero points for the involved group, similar to the consequences of plagiarism.

Task 1: K-means

K-means is a simple and widely used clustering algorithm. It partitions a dataset into k clusters by iteratively assigning each data point to the cluster with the nearest centroid and updating the centroids based on the mean of the data points in the cluster.

Task 1.1 (3 Points)

The first step of the K-means algorithm is to distribute the data points to the k partitions. This partition can be done randomly or by using a more sophisticated method. All partitions have to be non-empty after the initialization and each data point has to be assigned to exactly one partition.

Open `kmeans.py` and implement `_initialize_partitions`, which initializes the partitions for the K-means algorithm:

_initialize_partitions

```
def _initialize_partitions(
    self,
    points: List[Point]
):
```

Description:

- Initializes the partitions (self.partitions) by assigning each point to a cluster/partition. All clusters/partitions are non empty after this method is called.

Parameters:

- `points` (List[Point]): The points to cluster.

Returns:

- None - The method stores the partitions in `self.partitions`

The method expects a list of `Points` and does not return anything. The method should put each point into one of the partitions available at `self.partitions`. The number of partitions is given by the variable `self.k`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/kmeans/test_initialize_partitions.py
```

Task 1.2 (3 Points)

A second important part of the K-means algorithm is to update the centroids of the partitions. The centroid of a partition is the mean of all points in the partition.

Open `kmeans.py` and implement `_update_centroids`, which updates the centroids of the partitions:

_update_centroids

```
def _update_centroids(self):
```

Description:
- Updates the centroids of the partitions and writes the new centroids into self.centroids.

Parameters:
- None

Returns:
- None - The method updates `self.centroids`

The method does not expect any parameters and does not return anything. The method should update the `self.centroids` based on the mean of all points in the partition. The i-th centroid in `self.centroids` should refer to the i-th partition in `self.partitions`.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/kmeans/test_update_centroids.py
```

Task 1.3 (5 Points)

The last step of the K-means algorithm is to assign each data point to the cluster with the nearest centroid.

Open `kmeans.py` and implement `_reassign_points`, which assigns each data point to the partition with the nearest centroid:

_reassign_points

```
def _reassign_points(self) -> bool:
```

Description:

- Reassigns each point to the partition with the closest centroid. Ensures that each partition is non-empty after reassigning the points, by randomly reassigning a single point from a random non-empty partition with more than one element into each empty partition.

Parameters:

- None

Returns:

- `bool`: True if the reassignment changed the partitions, False otherwise

The method does not expect any parameters and returns a boolean. The method should remove all points from their previous partition and add them to the partition with the nearest centroid if there is a closer centroid. If the reassignment changed the partitions, the method should return `True`, otherwise `False`.

If there are empty partitions after the reassignment, the method should randomly reassign a single point from a random non-empty partition with more than one element into each empty partition. This is necessary to avoid empty partitions, which would lead to K-means not producing k clusters, but $k - n$ clusters (n being the number of empty partitions).

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/kmeans/test_reassign_points.py
```

Task 1.4 (3 Points)

The K-means algorithm is iterative. The algorithm stops if the partitions do not change anymore or if a maximum number of iterations is reached.

Open `kmeans.py` and implement `fit`, which combines the previous steps to implement the K-means algorithm:

fit

```
def fit(
    self,
    points: List[Point]
):
```

Description:
- Fit the K-Means clustering instance to the given points.

Parameters:
- `points` (List[Point]): The points to cluster.

Returns:
- None

The method expects a list of `Point`s and does not return anything. The method should implement the K-means algorithm by calling the methods `_initialize_partitions`, `_update_centroids`, and `_reassign_points`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/kmeans/test_fit.py
```

Task 2: DBSCAN

DBSCAN is a density-based clustering algorithm that groups together points that are closely packed together. It is based on two parameters: ε and $MinPts$.

Task 2.1 (2 Points)

A core part of the DBSCAN algorithm is to find all points that are within a distance of ε of a given point, the so-called ε-Neighborhood.

Open `dbscan.py` and implement `_get_neighborhood`, which returns all points that are within a distance of ε of a given point:

_get_neighborhood

```
def _get_neighborhood(
    self,
    point: Point,
    points: List[Point]
) -> List[Point]:
```

Description:

- Get the neighborhood of a point.

Parameters:

- `point` (Point): The point to get the neighborhood of.
- `points` (List[Point]): The points to consider.

Returns:

- `List[Point]`: The points in the neighborhood.

The method expects the `Point` for which the neighborhood should be determined and a list of all `Points` that should be considered. The method should return all points that are within a distance of ε of the given point as a list of `Points`. ε is given by the variable `self.epsilon`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/dbscan/test_get_neighborhood.py
```

Task 2.2 (14 Points)

The function `_get_neighborhood` can be used to implement the DBSCAN algorithm. The algorithm can be implemented either iteratively or recursively.

Open `dbscan.py` and implement `fit`, which implements the DBSCAN algorithm:

fit

```
def fit(
    self,
    points: List[Point]
) -> None:
```

Description:
- Fit the DBSCAN clustering instance to the given points.

Parameters:
- `points` (List[Point]): The points to cluster.

Returns:
- None - The method stores the found clusters in `self.clusters` and the noise points in `self.noise`

The method expects a list of `Point`s and does store the found clusters in the variable `self.clusters` and the noise points in the variable `self.noise`.

The method can be implemented either iteratively or recursively. Additional helper methods can be implemented if necessary, but they will not be tested and therefore award no points.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/dbscan/test_fit.py
```

Appendices

In Task 1 and Task 2 test cases are provided and used to grade the submission.

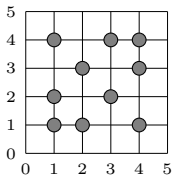
Dataset(s)

The most test cases are based on the following data sets:

Small Point Dataset

All test cases starting with the prefix `test_with_small_point_dataset` are based on the small dataset of 2D Points known from Exercise Sheet 5 - Task 1/Task 2.

The dataset is structured as follows:



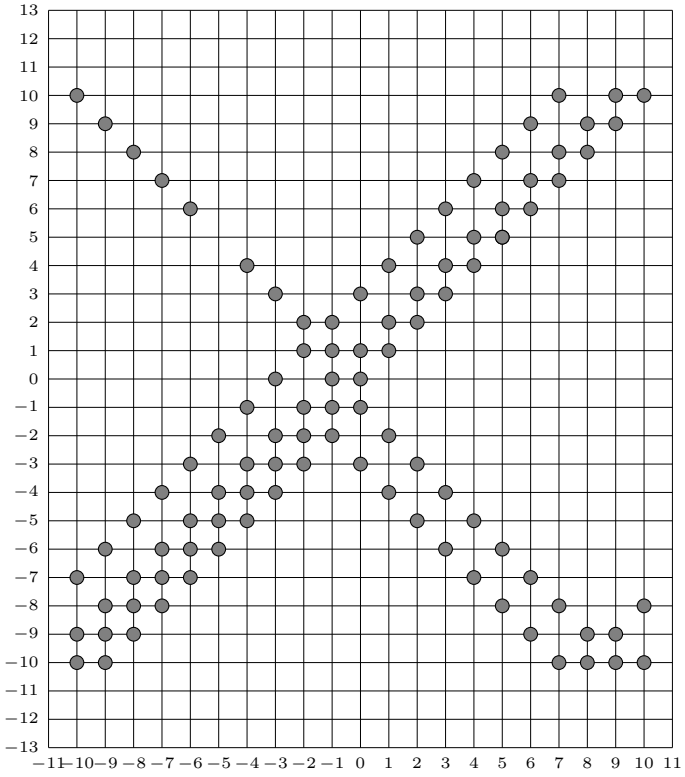
Points:

(1,1)	(1,4)	(2,3)	(3,4)	(4,3)
(1,2)	(2,1)	(3,2)	(4,1)	(4,4)

Bigger Point Dataset

All test cases starting with the prefix `test_with_bigger_point_dataset` are based on a bigger dataset of one-hundred 2D Points.

The dataset is structured as follows:



Points:

(-10,-10)	(-6,-6)	(-2,-1)	(2,2)	(6,-7)
(-10,-9)	(-6,-5)	(-2,1)	(2,3)	(6,6)
(-10,-7)	(-6,-3)	(-2,2)	(2,5)	(6,7)
(-10,10)	(-6,6)	(-1,-2)	(3,-6)	(6,9)
(-9,-10)	(-5,-6)	(-1,-1)	(3,-4)	(7,-10)
(-9,-9)	(-5,-5)	(-1,0)	(3,3)	(7,-8)
(-9,-8)	(-5,-4)	(-1,1)	(3,4)	(7,7)
(-9,-6)	(-5,-2)	(-1,2)	(3,6)	(7,8)
(-9,9)	(-4,-5)	(0,-3)	(4,-7)	(7,10)
(-8,-9)	(-4,-4)	(0,-1)	(4,-5)	(8,-10)
(-8,-8)	(-4,-3)	(0,0)	(4,4)	(8,-9)
(-8,-7)	(-4,-1)	(0,1)	(4,5)	(8,8)
(-8,-5)	(-4,4)	(0,3)	(4,7)	(8,9)
(-8,8)	(-3,-4)	(1,-4)	(5,-8)	(9,-10)
(-7,-8)	(-3,-3)	(1,-2)	(5,-6)	(9,-9)
(-7,-7)	(-3,-2)	(1,1)	(5,5)	(9,9)
(-7,-6)	(-3,0)	(1,2)	(5,5)	(9,10)
(-7,-4)	(-3,3)	(1,4)	(5,6)	(10,-10)
(-7,7)	(-2,-3)	(2,-5)	(5,8)	(10,-8)
(-6,-7)	(-2,-2)	(2,-3)	(6,-9)	(10,10)

Helper Classes

The following helper classes are provided in the `classes/` folder to support your implementation. Each class serves a specific purpose in the clustering algorithms.

Basic Data Structures

Point (`classes/point.py`)

Represents a single point in 2D space with additional flags for DBSCAN.

- **Attributes:**
  - `x` (float) - The x-coordinate of the point
  - `y` (float) - The y-coordinate of the point
  - `visited` (bool) - Flag for DBSCAN to indicate if the point has been visited (default: False)
  - `clustered` (bool) - Flag for DBSCAN to indicate if the point is part of a cluster (default: False)
- **Key Methods:**
  - `get_x()` - Returns the x-coordinate
  - `get_y()` - Returns the y-coordinate
  - `get_distance(other)` - Calculates Euclidean distance to another point
  - `is_visited()` / `set_visited(visited)` - Check/set visited flag
  - `is_clustered()` / `set_clustered(clustered)` - Check/set clustered flag
- **Usage:** Create points like `Point(1.0, 2.0)` or `Point(3.5, 4.2, visited=True)`
- **Example:**

```
point1 = Point(1.0, 2.0)
point2 = Point(4.0, 6.0)
distance = point1.get_distance(point2)  # Calculate distance
point1.set_visited(True)  # Mark as visited for DBSCAN
print(f"Point: {point1}")  # Output: (1.0, 2.0)
```

Cluster (`classes/cluster.py`)

Represents a cluster (or partition) of points. Can be used for both K-means partitions and DBSCAN clusters.

- **Attributes:**
  - `points` (List[Point]) - List of points in the cluster
- **Key Methods:**
  - `add_point(point)` - Add a point to the cluster
  - `remove_point(point)` - Remove a point from the cluster
  - `get_points()` - Returns all points as a list

  - `__contains__(point)` - Check if point is in cluster (enables `point in cluster`)
  - `__len__()` - Get number of points in cluster (enables `len(cluster)`)
  - `__iter__()` - Iterate over points (enables `for point in cluster`)
- **Usage:** Essential for storing and managing groups of points in both algorithms
- **Example:**

```
cluster = Cluster()
cluster.add_point(Point(1.0, 2.0))
cluster.add_point(Point(3.0, 4.0))

print(f"Cluster size: {len(cluster)}")  # Output: 2
for point in cluster:
    print(f"Point: {point}")

if Point(1.0, 2.0) in cluster:
    print("Point found in cluster")
```

Practical Tips

- **Distance Calculation:**
  Use `point1.get_distance(point2)` for Euclidean distance between points
- **DBSCAN Flags:**
  The `visited` and `clustered` flags in Point are specifically designed for DBSCAN implementation
- **K-means Partitions:**
  Use Cluster objects to represent the k partitions in K-means
- **DBSCAN Clusters:**
  Use Cluster objects to store the final clusters found by DBSCAN
- **Iteration:**
  Both Point and Cluster classes support Python's standard iteration patterns
- **Point Equality:**
  Points are considered equal if they have the same x and y coordinates
- **Debugging:**
  Both classes have `__str__` methods for easy printing and debugging
- **Neighborhood Search:**
  For DBSCAN, use `get_distance()` to find points within epsilon distance