EDAMI Project – Final Report

Implementation of the OPTICS algorithm

Daniel Bannister - daniel.bannister.stud@pw.edu.pl

1) Problem definition and introduction

Cluster analysis is a core method in data mining, designed to help uncover the natural groupings or structure within a dataset. It can serve as a standalone tool to gain insights into data distributions or as a preprocessing step for other algorithms that utilize detected clusters. Traditional clustering algorithms often rely on user-defined parameters that significantly impact results and may fail to capture the true structure of complex, high-dimensional data. Density-based clustering, including DBSCAN, identifies clusters as regions of high density separated by lower-density regions, offering flexibility in discovering clusters of arbitrary shape. However, global parameter settings in DBSCAN can lead to suboptimal clustering when densities vary across regions.

To address this, OPTICS (Ordering Points To Identify the Clustering Structure) was developed as a density-based clustering method that provides an ordered representation of the dataset's density-based structure. OPTICS does not produce explicit cluster memberships but instead generates a cluster-ordering that captures intrinsic clustering structures across a wide range of parameter settings, enabling a more versatile and insightful cluster analysis.

2) Description of the proposed algorithm

OPTICS is an extension of DBSCAN designed to overcome its limitations in handling datasets with varying densities. The algorithm constructs an ordering of the dataset based on its density-based clustering structure, allowing users to explore clusters interactively or automatically extract them at different density thresholds.

The key innovation of OPTICS lies in its use of **reachability-distance** and **core-distance**. For each data point, the core-distance defines the minimum distance required for it to become a core object, while the reachability-distance measures how closely connected a point is to its neighbors. By processing data points in order of increasing reachability distance, OPTICS generates a cluster-ordering that reveals the hierarchical clustering structure of the dataset.

This ordering is visualized using a **reachability plot**, where valleys correspond to dense clusters, and peaks indicate sparse regions or noise. The algorithm efficiently handles datasets with complex cluster structures and varying densities, offering significant advantages over traditional methods, including scalability, robustness to parameter choices, and the ability to detect nested clusters.

OPTICS does not perform properly speaking “clustering”, as it only orders the database and computes the reachability distances for all objects (so-called “reachability plots”). To extract clusters from the reachability plots, an additional algorithm is needed. Such an algorithm is described in [1] (`ExtractClusters`) but it was decided not to implement it in this project.

3) Description of the implementation

Considering the nature of the problem, I chose to use an object-oriented programming (OOP) language, in this case Java. OOP languages help describe objects in the real world and their relations to one another. Furthermore, Java has many advantages: platform independence, project portability, object-oriented features, and robust standard libraries. Its rich set of data structures, such as ArrayLists and PriorityQueues, simplifies managing datasets and cluster attributes. Additionally, Java's strong performance and built-in support for multithreading facilitate the efficient handling of large-scale data, which is crucial for computationally intensive clustering tasks like OPTICS.

A) Development

The project can be found in this Github repository. It was developed on Windows, using the Maven development framework. It requires Java and Python to work properly (see User’s Manual for more). The repository was made public and can be cloned either by SSH or HTTPS.

B) Java classes: methods and attributes

The first step of my implementation process was to list the necessary classes, their attributes, methods and their relations to one another.

i) Data types

`DObject` is the abstract class that all datatypes extend. Its attributes are basic information relative to its processing in the algorithm: ID, process information, neighbors, etc. Classes `Iris` and `Point2D` extend the `DObject` class. Their attributes are specific to the data they represent: `Iris` has sepal/petal length/width, `Point2D` has x/y coordinates. They implement the abstract method `distance()` that calculates the Euclidean distance between two objects of the same class.

ii) MyOPTICS

This class helps use the OPTICS algorithm in a practical case: instead of calling an `OPTICS()` method in the main function, we instantiate an object of `myOPTICS` and call it's `cluster()` method. This helps with readability and reproductivity: if we want to use OPTICS with multiple parameters, we simply create multiple `myOPTICS` objects. The attributes of this class are the arguments of the OPTICS algorithm (`epsilon` and `MinPoints`) and the `orderedSeeds` priority queue. Other methods include `ExpandcluserOrder()` and `OrderSeedsUpdate()`. They are set to private, as they are not used outside of the algorithm's scope.

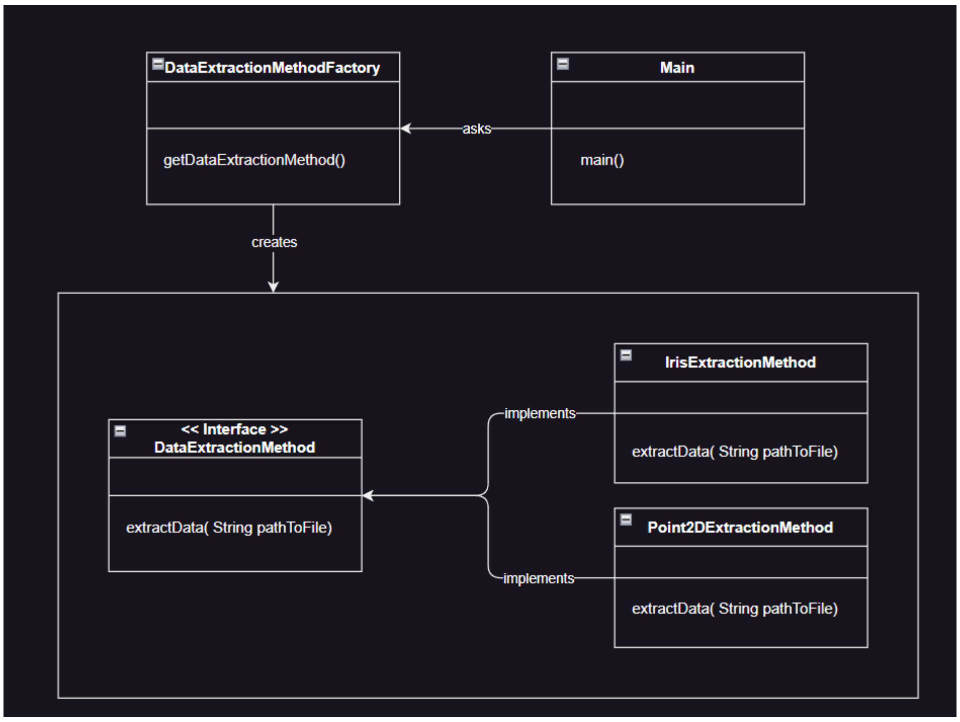
iii) Main

The `Main` class simply consists of the `main()` method. This function is the one that runs when calling the Java executable. It takes arguments from a CSV file. The arguments indicate which dataset to use and what arguments to call `myOPTICS.cluster()` with. After the OPTICS algorithm is performed on the dataset, the results are written in the output directory.

iv) Data extraction

`DataExtractionMethod` is an interface that helps extract data from a CSV file. The way the data is extracted from the file and inputted in a ArrayList of corresponding objects (eg, `Iris`) is determined by an argument of main(): category. Knowing what datatype the file contains; we can choose what method to use by calling `DataExtractionMethodFactory.getDataExtractionMethod()`

The Factory Design Pattern is a creational design pattern that provides an interface for creating objects without specifying their exact class. It allows subclasses or methods to decide which object to instantiate, promoting flexibility and reducing coupling in code:


DataExtractionMethodFactory
getDataExtractionMethod()
Main
main()
asks
creates
<< Interface >>
DataExtractionMethod
extractData(String pathToFile)
implements
IrisExtractionMethod
extractData(String pathToFile)
implements
Point2DExtractionMethod
extractData(String pathToFile)

DataExtractionMethod factory schematic

v) <u>DataSetStats</u>

This class provides helpful information on the dataset: number of objects, mean and standard deviation of the distances between objects. It also provides a method to print all information of the dataset.

<u>C) Comparison to reference implementation</u>

A few Java implementations of the OPTICS algorithm can be found [3][4], but they are usable for comparison to our implementation for the following reasons:

- They use entirely different frameworks, packages, methods, etc., which makes them difficult to interface with our project
- They do not provide access to core distances and/or reachability distances (methods set to `private` or `protected`). They perform clustering using cluster-extracting algorithms that are also set to `private` or `protected`
- Lack of documentation to adapt the cluster-extracting algorithms to our framework

To have a reference implementation to compare our implementation to, we use the Scikit-lean library in Python [5]. Using Python also helps with making graphs.

4) User’s manual

A) Requirements

The requirements to run the project are as following:

- Java 21.01
- Python 3.12.8 and libraries including: numpy, matplotlib, math, sklearn, sys, csv and panda. Most libraries are installed with Python, but some must be installed manually using pip.

B) Running the project

The main function of the project is to call our OPTICS implementation and it's Scikit-learn counterpart in Python. To run the project:

- Write in args.csv the arguments you will call the main() Java method with: type of data (“category”, can be Iris or Point2D), “dataset” name (see data/[category]), minPts and epsilon.
- Run the Java executable:
  ```
  java -jar .\EDAMI-Project-1.0-SNAPSHOT.jar
  ```
- Run the Python script:
  ```
  python3 .\src\test\python\compareResults.py
  ```

Note: the project was developed on Windows. Commands are different for MacOS and Linux.

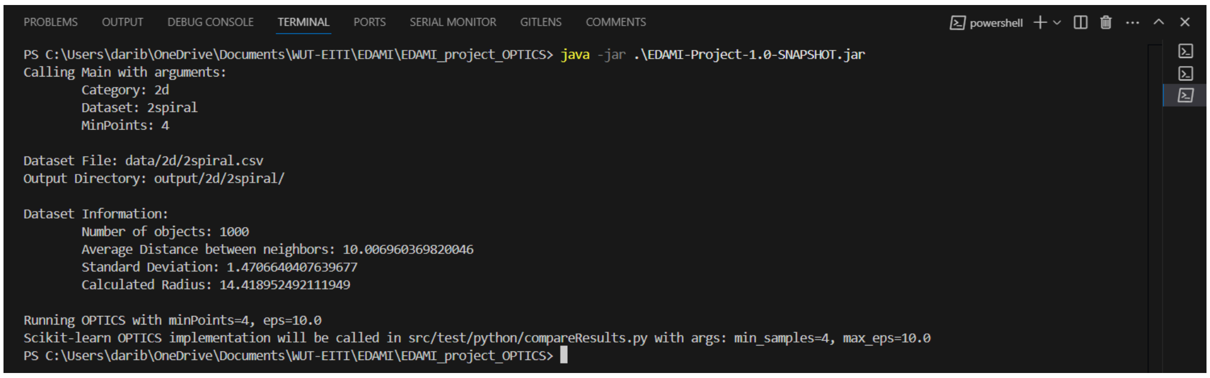
/!\\ /!\\ /!

It is recommended to first run the Java executable with `eps` before executing again with `eps` more or less equal to the mean distance between neighbors found by the first run.

/!\\ /!\\ /!

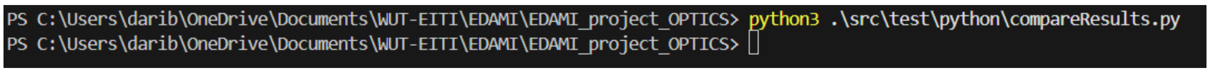
C) Example

- Open a VSCode terminal or command line prompt at the root of the project and execute the Java executable:



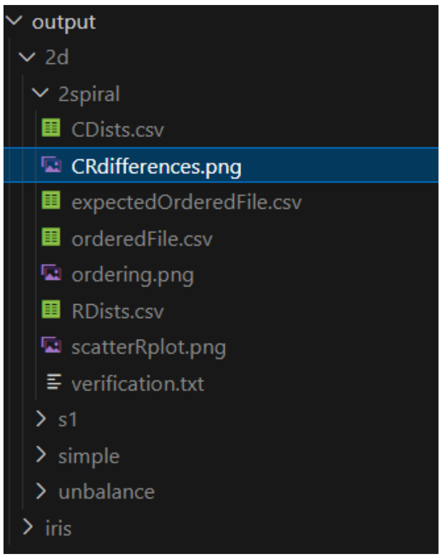
```
PS C:\Users\darib\OneDrive\Documents\WUT-EITI\EDAMI\EDAMI_project_OPTICS> java -jar .\EDAMI-Project-1.0-SNAPSHOT.jar
Calling Main with arguments:
        Category: 2d
        Dataset: 2spiral
        MinPoints: 4

Dataset File: data/2d/2spiral.csv
Output Directory: output/2d/2spiral/

Dataset Information:
        Number of objects: 1000
        Average Distance between neighbors: 10.006960369820046
        Standard Deviation: 1.4706640407639677
        Calculated Radius: 14.418952492111949

Running OPTICS with minPoints=4, eps=10.0
Scikit-learn OPTICS implementation will be called in src/test/python/compareResults.py with args: min_samples=4, max_eps=10.0
PS C:\Users\darib\OneDrive\Documents\WUT-EITI\EDAMI\EDAMI_project_OPTICS>
```

- Note that the average distance between neighbors is more or less equal to our argument `eps`. We don't have to change anything.
- Call the Python script:



```
PS C:\Users\darib\OneDrive\Documents\WUT-EITI\EDAMI\EDAMI_project_OPTICS> python3 .\src\test\python\compareResults.py
PS C:\Users\darib\OneDrive\Documents\WUT-EITI\EDAMI\EDAMI_project_OPTICS>
```

The results are found here:

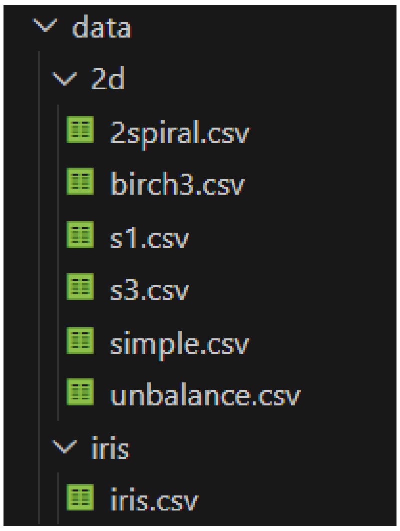


```
output
  2d
    2spiral
      CDists.csv
      CRdifferences.png
      expectedOrderedFile.csv
      orderedFile.csv
      ordering.png
      RDists.csv
      scatterRplot.png
      verification.txt
    s1
    simple
    unbalance
  iris
```

5) Description of the used datasets

A) Overview

The project is already provided with a few datasets in the `data/` folder. The folder is organized around the datatypes of each dataset:



```
data
  2d
    2spiral.csv
    birch3.csv
    s1.csv
    s3.csv
    simple.csv
    unbalance.csv
  iris
    iris.csv
```

B) Detailed descriptions of the datasets

The *Iris* dataset is a classic benchmark for clustering and classification tasks, consisting of 150 samples from three species of iris flowers: Setosa, Versicolor, and Virginica. Each sample has four features: sepal length, sepal width, petal length, and petal width. In clustering, the dataset is often used to evaluate algorithms by grouping data points based on feature similarities without prior knowledge of the species labels. Successful clustering should ideally separate the data into three distinct clusters, reflecting the natural divisions between the species. However, overlapping features between Versicolor and Virginica make this a non-trivial task, offering a good test of an algorithm's performance.

Datasets *bitch3*, *s1*, *s3* and *unbalance* are from an existing database used for the paper *"K-means properties on six clustering benchmark datasets"* [2]. Their description can be found here. The datasets chosen from the database for this project present interesting characteristics: different data density, linearly separable data, high data count, etc.

Datasets *simple* and *2spiral* are two basic 2D datasets used to debug the algorithm. They do not contain a lot of data, and clusters are easily identifiable on the scatter plot.

6) Experimental results

The results of each project run (calling of `myOPTICS.cluster()` and comparison to the `sklearn.OPTICS` implementation) are stored in `output/[category]/[dataset]`; `[category]` and `[dataset]` being the arguments passed for the project run in `args.csv`.

For instance, after running the project with arguments `category = 2d, dataset = 2spiral, minPts = 4` and `eps = 10.0` (as in the user's manual example), we can find graphs and CSV files in the corresponding output folder:

- `CDists.csv`: contains the core distances in object order (first line is the core distance of object ID 1, etc.)
- `CRdifferences.png`: 2 graphs representing the relative differences between expected and obtained distances in object order. One graph for core distances and another for reachability distances
- `ExpectedOrderedFile.csv`: ordering of the dataset obtained with the Scikit-learn OPTICS implementation (attribute `ordering_`)
- `OrderedFile.csv`: ordering obtained with our OPTICS implementation
- `Ordering.png`: graph plotting the ordering obtained with our implementation (obtained) and the Scikit-learn implementation (expected). For instance, if the object ID 6 is found at line 2 of orderedFile, a point (x=2, y=6) will be plotted
- `RDists.csv`: contains the reachability distances in object order
- `ScatterRplot.png`: graphs of the scattered dataset (only if 2D objects) and reachability plots (expected and obtained). The reachability plots are plotted with their respective reachability distances AND ordering.
- `Verification.txt`: indicates the arguments used in the project run, and if the obtained ordering is equal to the expected ordering.

We obtain the following experimental results:

	Core distance difference (%)		Reachability distance differences (%)			
Dataset	**mean**	**std**	**mean**	**std**	**Ordering**	**Reachability plots**
iris	0	0	0,23	5,47	incorrect	similar, but visible differences
simple	0	0	0	0	correct	identical
2spiral	0	0	0	0	correct	identical
s1	0	0	0,03	3,56	incorrect	very similar
unbalance	0	0	0,03	2,68	incorrect	very similar

For all datasets, the core distance differences are not null, but very small, in the order of 10^{-11} %. This is obviously a negligible difference that can be explained by the differences in which computations are done in Java and Python. Identical core distances prove **we correctly identify all the neighbors**.

Scikit-learn optimizes its OPTICS implementation (for instance when looking for the nearest neighbors), while our calculations are quite basic (using brute force). These optimizations are also reflected in the speed at which the Scikit-learn implementation clusters a dataset, compared to our implementation: usually less than a few second for Scikit-learn for all datasets, against a minute or so for the largest datasets with our implementation.

The reachability distances obtained are usually identical to the expected ones, as proven by the very small average relative difference. However, it seems that whenever the reachability distances differ from what is expected, they differ by a lot (great standard deviations compared to means). Since the ordering is done based on the reachability distances, it is normal to have incorrect ordering when having non-zero reachability distance differences.

7. Conclusions

Since the obtained reachability plots are very similar – if not identical – to the expected in all cases, we can consider that using the same cluster-extracting algorithm on the reachability plots obtained with our implementations and the Scikit-learn implementation, we would get almost identical clusters. In fact, information about clusters is intrinsic to the reachability plot [1], which means similar reachability plots will result in similar clustering. This is still a qualitative comparison; an implementation of a cluster-extracting algorithm would've helped compare the two implementations more precisely. Still, I think my implementation is satisfactory.

8. Bibliography

1 - Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, Jörg Sander, *"OPTICS: Ordering Points To Identify the Clustering Structure"*, Proceedings of the 1999 ACM SIGMOD international conference on Management of data (SIGMOD '99)

2 - P. Fänti and S. Sieranoja *"K-means properties on six clustering benchmark datasets"* Applied Intelligence, 48 (12), 4743-4759, December 2018

3 - https://github.com/tedgueniche/DataMiningSandbox/blob/master/src/cluster/optics/Optics.java

4 - https://github.com/EdwardRaff/JSAT/blob/master/JSAT/src/jsat/clustering/OPTICS.java

5 - https://scikit-learn.org/dev/modules/generated/sklearn.cluster.OPTICS.html