Computational Geometry Applications in Clustering

—

Nikolaos Lekkas-Emmanouil Lykos
Spring 2019-20

What is Clustering?

Clustering is the task of grouping a set of objects in such a way that objects in the same group (called a **cluster**) are more similar (in some sense) to each other than to those in other groups (clusters).

More formally we can define **clustering** as follows:

Given **n** points $p_1, p_2, \ldots, p_n$ we should find **k** centers $c_1, c_2, \ldots, c_k$ that minimize the following formula:

$$\sum_{n}^{i=1} min_j distance(p_i, c_j)$$

Clustering Applications

Clustering is an interesting problem because has many applications in various fields such as:

1. Business
2. Biology
3. Computer Science
4. Social Sciences

Computational Geometry Contribution

Computational Geometry can contribute to the problem of **Clustering** in the two following ways:

1. Using some tools that can optimize some subtasks like range queries, nearest neighbors etc. of the already known algorithms such as k-means, hierarchical clustering etc.
2. Inventing new clustering algorithms using computational geometry principles such as Voronoi Diagram.

In this presentation firstly we will present the first type of applications and afterwards the second type.

DBSCAN

1. Is a **density-based** algorithm.
2. It produces good results if the radius and the necessary minimum number of points inside that radius are chosen wisely, and the clusters does not have varying densities.
3. Moreover, this algorithm has the advantage that identifies the noisy points of the dataset.
4. In this algorithm each point is classified in the following three categories.
    a. **Noise**: which are the points that does not have the necessary number of neighbors within the given radius.
    b. **Core**: the points that have the necessary number of neighbors within the given radius, and its neighbors might belong in the same cluster.
    c. **Border**: the points that does not have the necessary number of neighbors within the given radius, but are neighbors within that radius to a core point.

DBSCAN

Given radius **r** and the necessary number of neighbors within **r, minPts,** the algorithm is the following:

1. Current cluster **c**=0
2. For each point **p** in the **Database:**
    a. if **p** is not **unclassified** then proceed to next point
    b. Neighbors **N** = RangeQuery(**Database**, **p**, **r**)
    c. if size(**N**) < **minPts** then the point is classified as **Noise** and proceed to next point
    d. else, **c=c+1,** assign **p** to cluster **c** and insert the points in set **N\{p}** into a queue **Q**
    e. while **Q** is not empty
        i. Pop an element **q** from **Q**
        ii. if **q** is **Noise** then assign it to cluster **c**
        iii. if **q** is not **unclassified** then proceed to next point in **Q** else assign it to cluster **c**
        iv. Neighbors **N** = RangeQuery(**Database, q, r)**
        v. if size(**N**) >= **minPts** then the point **q** is a **core** point therefore the cluster **c** should "expand" at its neighbors in **N,** thus we add the elements of set **N\{q}** into **Q,** else the point is a **border** point, thus we do not do anything.

What is the **time complexity** of the above algorithm?

DBSCAN Complexity

1. We can observe that each point of the dataset can be examined, in order to get classified, once.
2. Thus, we observe that the complexity depends also by the complexity of the **Range Query.**
3. From the observation in the first bullet the complexity of the algorithms is

$$\mathcal{O}(n) * RangeQueryComplexity$$

But what is the complexity of the **Range Query**?

DBSCAN Complexity

The most naive solution for the **Range Query** is to make a loop through every point in the database and find the neighbors that have a distance less than a given distance. This solution have $\mathcal{O}(n)$ time complexity for each query thus DBSCAN will have $\mathcal{O}(n^2)$ complexity. But we can do better...

In [1], where DBSCAN is proposed the author writes:

"*Region queries can be supported efficiently by spatial access methods such as R*-trees ...which are assumed to be available in a SDBS for efficient processing of several types of spatial queries .... The height an R*-tree is O(log n) for a database of n points in the worst case and a query with a "small" query region has to traverse only a limited number of paths m the R -tree.*"

Thus, DBSCAN can be done with average time complexity equal to $\mathcal{O}(nlogn)$.

But, what is an **R-Tree** and an **R* Tree**?

What is R-Tree

- A **spatial indexing type.** Specifically, a data structure that is used by **Spatial Database Management Systems(SDBMS),** in order to answer efficiently spatial queries like Distance, Intersects, etc.
- A **dynamic index structure** proper for multi-dimensional objects of non-zero sizes like countries in the map, or buildings and streets in a aerial photograph.
- R-Trees are used in order to partition the multidimensional spaces into smaller rectangles and this smaller rectangles into other rectangles etc.
- R-Trees are much useful in Range Queries, for example in queries like “Find all countries that lie in a radius N of a specific point in the map”.

Why R-Tree

As said in [2] the reasons to prefer the **R-Tree** are the following:

- **Cell Methods**: The rectangle borders are determined in advance thus these methods are not dynamic.
- **Quad Trees** and **k-d Trees**: Do not deal with the management of secondary memory.
- **K-D-B Trees:** They deal with secondary memory but are useful only for point data(maybe we could use them instead of R-Tree in our case).
- **Index Intervals:** Are not proper for multi-dimensional data.
- **Corner stitching:** Are not efficient in multiple searches in large datasets.

R-Tree Structure

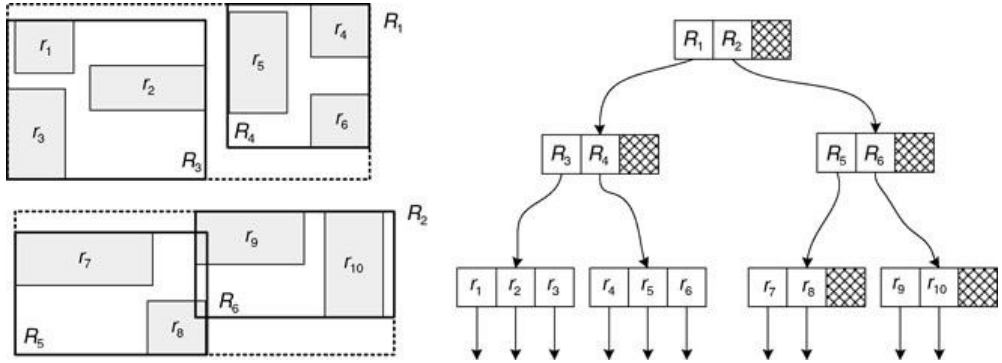


Figure 1: R-Tree Example

- It's a height-balanced tree like a B-Tree, and maintains some of its properties that we will mention later.
- Every record of a **leaf** node has the structure $(I, data)$ where $I = (I_1, I_2, \ldots, I_n)$ where I_i is an interval that denotes the bounds of the rectangle in the i-axis.
- Every record of an **internal** node has the following structure $(I, child - pointer)$ where $child - pointer$ is a pointer to its child node.

R-Tree Properties

- Every node except the **root** node has $[m, M]$ records where $2 \leq m \leq \frac{M}{2}$ and $m \in \mathbb{N}$.
- I of every record of a **leaf** node is the **minimum bounding box(MBR)** of the data that contains.
- I of every record of an **internal** node is the **MBR** of the I lists of the records in the child node.
- **Root** node contains at least 2 records(childs), with exception if root is the only node in the tree.
- All the **leaf** nodes are on the same level.

Afterwards, we will present the Basic Operations of an R-Tree.

R-Tree Operations

In the **Search** operation the user gives as input a bounding rectangle and we should find all objects that its bounding box has intersection with the given bounding rectangle. The **Search** algorithm is straightforward and is the following:

1. Input: **SearchBoundingBox**
2. **CurrNode = root**
3. If **CurrNode** is **leaf** node then
   a. For each record **R** check if its bounding box **R.bounding-box** intersects with **SearchBoundingBox**
      i. If the bounding boxes intersect then add this record to a list **L**
   b. Return **L**.
4. Else, for each record **R** that its bounding box intersects with **SearchBoundingBox** call recursively the **Search** procedure with **CurrNode = R.child-node**.
5. Concatenate the returned lists into a list **L** and return **L**.

R-Tree Operations

Insertions approximately are like the ones in B-Tree. The algorithm is the following:

1. Input: Bounding Box **B,** Object **data.**
2. **CurrNode** = **root**
3. While **CurrNode** is not **leaf** node
    a. **CurrNode** = ChooseSubtree(**CurrNode**)
4. If **CurrNode** has space for another entry, insert **Record(B,data)** and **N** = **CurrNode.**
5. Else, invoke **SplitNode(CurrNode)** to get the new nodes **N** and **NN.**
6. **CurrNode** = **CurrNode.parent.** We suppose that each node knows who is his parent node.
7. While **CurrNode** is not **None**
    a. Adjust the **Minimum Bounding Box** of the record in **CurrNode** that his child is the node **N**.
    b. If **NN** exists(the child was splitted) then create a new record **R** that has as bounding box the **MBR** of **NN** and its child pointer points to **NN.**
    c. If **CurrNode** has space for another record the insert **R** to **CurrNode** else split the node
    d. Change the values of **N** and **NN** like *Step 4 and 5*.
    e. **CurrNode** = **CurrNode.parent**
8. If **root** was splitted then create new **root** with childs the nodes **N** and **NN** and do the necessary adjustments.

R-Tree Operations

The delete operation of an R-Tree differs from the one of the B-Tree, because in R-Tree we do not fuse neighboring nodes of transfer the record of one node to another, instead R-Tree deletes the underfull nodes and keeps their records into a list and reinserts them. Precisely, the algorithm is the following

1. Input: Object **data** and its corresponding bounding box **B**
2. Invoke the **Search** operation to find the leaf node **CurrNode** that contains the record **R** that keeps **data**.
3. Delete **R** from **CurrNode**
4. **L** = []
5. While **CurrNode** is not **root**
    a. If **CurrNode** is underfull then delete him, delete its parent record from **CurrNode.parent** and add the remaining elements of **CurrNode** in **L**
    b. Else, adjust the bounding box of its parent record in **CurrNode.parent**
    c. **CurrNode = CurrNode.parent**
6. Reinsert all records of list **L** by invoking **Insert**, with difference that these records will be inserted in their proper level and not on the leaf level.
7. If **root** has only one child then delete the **root** and the new root will be **root.child**.

R-Tree Heuristics

- As you can observe we did not mentioned the **ChooseSubtree** and **SplitNode** routines.
- These functions can work the way we want with only requirement to be valid their return values.
- But, in **R-Tree** we want this function to optimize some value, so the **Search** algorithm visits the minimum number of nodes necessary, in order to its complexity will be near $\mathcal{O}(log(n))$, so the Range Query in DBSCAN will be indeed faster.
- **R-Tree** wants to minimize the sum of the areas of node's records bounding boxes.
- Other heuristics will be discussed later in **R*-Tree**.

R-Tree Heuristics

- **ChooseSubtree** method of an **R-Tree** simply chooses the record's child node that record's bounding box needs least are enlargement.
- Where ties are present we resolve them by choosing the record with the smallest area.

With this approach we can clearly see that whenever **ChooseSubtree** is invoked it chooses greedily the bounding box that its change will result to the smallest possible total area that the node's rectangles have.

R-Tree Node Splitting

- Our task is to split the node to 2 other nodes such that the sum of the areas of the Minimum Bounding Boxes of the 2 nodes will be minimized.
- Because the Brute Force algorithm has exponential time complexity, in [2], the following two approximate splitting methods were proposed:
    - Quadratic Split
    - Linear Split

R-Tree Node Splitting

1. **Quadratic Split** is a greedy splitting algorithm.
2. Firstly, chooses the first entry of each of the resulting nodes(seeds), by choosing 2 records, of the node to be splitted, P and Q that have the maximum value of $d = area(J) - (area(P.bounding\_box) + area(Q.bounding\_box))$ among all pairs (where J is the Minimum Bounding Box that encloses the bounding boxes of P and Q).
3. Afterwards, while we can choose for the rest of the records or we did not assign every node to some node, we choose the next record to be inserted to a group by determining the record that has the maximum difference of the area enlargement that is needed to be done by the two resulting nodes to enclose the new record.
4. Then the record to be inserted on a new nodes, get inserted to the node that needs least area enlargement to enclose it. Ties are resolved by choosing the node of minimum area.

R-Tree Node Splitting

Quadratic Split algorithm has $\mathcal{O}(M^2)$ time complexity, where M is the maximum number of records that a node can hold.

Proof

- The first step requires $\mathcal{O}(M^2)$ time because we should check every pair of nodes and determine which is the most suitable
- Assuming that we can find the new bounding boxes in constant time the loop of steps 3 and 4 needs $\mathcal{O}(M^2)$ time because in the worst case the loops run for all elements of the initial node, and we should check each of the remaining elements.

Therefore, quadratic split has $\mathcal{O}(M^2)$ time complexity.

R-Tree Node Splitting

1. **Linear Split** is a linear-cost splitting algorithm.
2. Obviously it is faster that the aforementioned **Quadratic Split** algorithm.
3. The algorithm works as follows:
    a. Along each axis find the rectangles(they can be the same rectangle) that have the highest low value and the lowest high value of their axis range.
    b. Normalize their separations by dividing with the total width along the current axis.
    c. The seeds(or the seed) are the rectangles(rectangle) that have the greatest normalized separation.
    d. Insert half of the entries into the first group and the other half into the second group.

This algorithm might be completely naive but in [2] it is shown that performs pretty well.

What is R*-Tree

- R-Tree variation.
- It is used to question the optimization criterion(see *R-Tree Heuristics)* used in R-Tree by using a combination of other heuristics, in order to achieve better retrieval performance.

- These heuristics are the following:
  a. Maximization of the minimum bounding box area covered by node's directory rectangles.
  b. Minimization of the overlap value of the directory rectangles in a node.
  c. Minimization of the margin value of a directory rectangle.
  d. Storage utilization.

- It is not mandatory to use all of the above heuristics for the following reasons:
  a. The use of one heuristic might serve another heuristic even if we do not use it.
  b. The use of all heuristics are not achieving the best retrieval performance, but a good combination of them does.

- Afterwards, we will present the R*-Tree operations that are done differently than the R-Tree.
- The presented operations are using the heuristics that the author in [3], found that result in the best retrieval performance.

R*-Tree Operations

- In order to discuss **ChooseSubtree** operation, firstly we should define the overlap value of a record R_k that belongs in a node that contains the records $R_1, R_2, \ldots, R_p$ which is equal to

$$overlap(R_k) = \sum_{i=1, i \neq k}^{p} area(R_k.bounding\_box \cap R_i.bounding\_box)$$

- **ChooseSubtree** works as follows:
  a. If **CurrNode** children are **leaf** nodes, then choose the child node of a record that its bounding box needs least overlap enlargement. Resolve ties by choosing the record that its bounding box needs least area enlargement.
  b. Else, choose the child node of a record like it is done at **ChooseSubtree** of an R-Tree**.**

- The above algorithms runs in quadratic time if **CurrNode** children are **leaf** nodes, but the algorithm can run in subquadratic time with an optimization that is referred in [3].

R*-Tree Node Splitting

The node splitting algorithm of an R*-Tree generally does the following:

1. Along each axis, sort the entries by their high and low value of their interval along that axis.
2. For each of the $M - 2m + 2$ distributions of each sorted list find their goodness values in order to determine the axis perpendicular to this the split will occur.
3. For each distribution of each sorted list that corresponds to the split axis we calculate its goodness value(it can be different from the one of the previous step) in order to determine the resulting split nodes which will be the two parts of the distribution.

R*-Tree Node Splitting

Goodness values can be the following:

1. **Area Value:** the sum of the area of the bounding box of first group and the bounding box of second group.
2. **Margin Value:** the sum of the margin of the bounding box of first group and the bounding box of second group. The **margin** of a rectangle is the sum of its edges.
3. **Overlap Value:** the area of the intersection of the bounding boxes of the two groups.

These **goodness** values can be used in order to find the most suitable distribution to take, as follows:

1. Take the distribution that has the minimum chosen **goodness** value.
2. Take the distribution that has the minimum sum of some of the **goodness** values.

R*-Tree Node Splitting

The goodness values that are used to choose the most suitable distribution in order to choose the split axis and the two resulting nodes are the following:

1. In order to choose the **split axis** we sum the margin values of all distributions along that axis and we choose the axis perpendicular of the axis.
2. In order to get the **two resulting nodes** we get all the distributions along the chosen axis and we choose the one that has the minimum overlap value. Ties are resolved by choosing the distribution with the minimum area value.

The node splitting algorithm of an R*-Tree requires $\mathcal{O}(Mlog(M))$ time because:

- The first and second step require $\mathcal{O}(Mlog(M))$ because it does four sorts(if we assume that data are 2D) which in total,require $\mathcal{O}(Mlog(M))$ and traverses each distribution once which requires $\mathcal{O}(M-2m+2)=\mathcal{O}(M)$ time.
- The third step requires $\mathcal{O}(Mlog(M))$time if we do not have ready the distributions, thus we need to sort them before we find the most suitable one. Else, we will only check which distribution has the minimum overlap value, which only requires $\mathcal{O}(M-2m+2)=\mathcal{O}(M)$ time.

R*-Tree Forced Reinsert

- The location of the R/R*-Tree entries are non-deterministic and probably depends from the sequence of the insertions like a Search Tree.
- Thus, an organization of the tree might achieve good retrieval performance, but a change in the tree such as a split or a deletion might cause that organization to have a bad retrieval performance.
- Therefore, we need a strategy to reorganize the tree, with purpose to keep its good retrieval performance.
- R-Tree, for instance, has the mentality of the reorganization in deletions.
- If we simply delete some records and after reinsert them we will achieve better retrieval performance, but for dynamic data we need more sophisticated methods.

R*-Tree Forced Reinsert

- The reinsertions in the R*-Tree are happening in some cases when an overfull node occurs through the insertion of one record.
- The insertion routine is the same as the insertion of an R-Tree with the following differences:
  - If an overfull node encountered we invoke a routine for overflow treatment.
  - The record is inserted to its proper level and not necessarily as a leaf.
- The overflow treatment routine just checks if the overfull node is the first overfull node that is encountered on the given level and if it is the reinsertion routine is invoked, else node splitting is performed.
- The reinsertion routine in an R*-Tree works as follows:
  - Sort the entries in decreasing order by their distance between the centers of their bounding rectangle and the minimum bounding box of their node.
  - Remove the first **p** percent entries from the current node, and adjust the minimum bounding box of its record on the parent node.
  - Invoke **Insert** to the deleted entries by starting with the entry with maximum distance(far reinsert) or with the entry with minimum distance(close reinsert).

R*-Tree Forced Reinsert

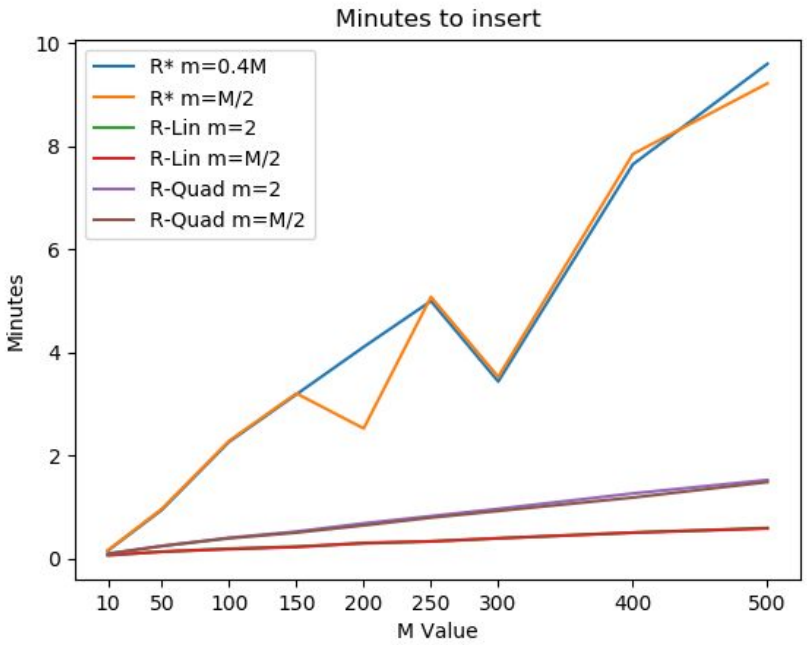
As said in [3], we can say the following about **Forced Reinsert**:

- Close Reinsert outperforms Far Reinsert.
- The optimal value of **p** is 30 percent of the max node records.
- Will not necessarily make the insertion routine more slow because less splits will happen.
- The overlap value decreases because some records are moved to other nodes.
- The shape of directory rectangles become more quadratic, thus the margin value decreases.
- Storage utilization is improved.

R-Tree Performance Compare

- We wrote Linear R-Tree, Quadratic R-Tree and R*-Tree in C++.
- Our experiments were conducted for 500000 points and 10000 Range Queries.
- Specifically, we inserted all the points one by one and after that we did the Range Queries.
- We compared the stats of these trees that gave the better performance as said in [2] and [3] and with which are:
  - For Linear R-Tree **m=2**
  - For Quadratic R-Tree **m=2**
  - For R*-Tree **m** is the 40% of **M**
- We also run the experiments of all trees with **m** equal to 50% of **M**.
- Note that because was hard to implement we didn't wrote the Forced Reinsert of an R*-Tree.

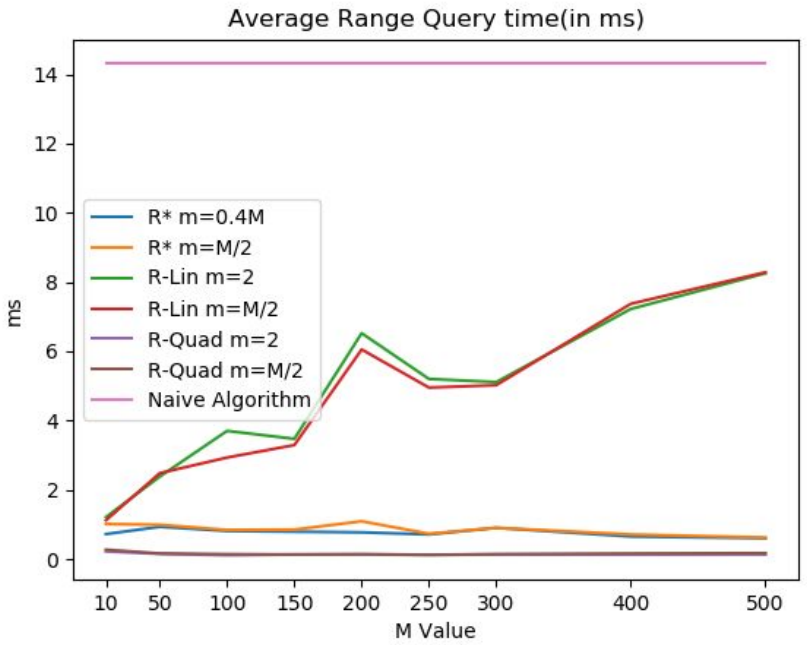
R-Tree Performance Compare



From the graph we can observe the following:

- R-Tree with Linear split is the faster solution to insert the records which is obvious because the node splitting algorithm runs in linear time.
- R*-Tree is the slowest solution to insert the records because:
  - Probably more splits are happening.
  - The **ChooseSubtree** algorithm is slower that the **ChooseSubtree** algorithm of the R-Tree, even with the optimization.

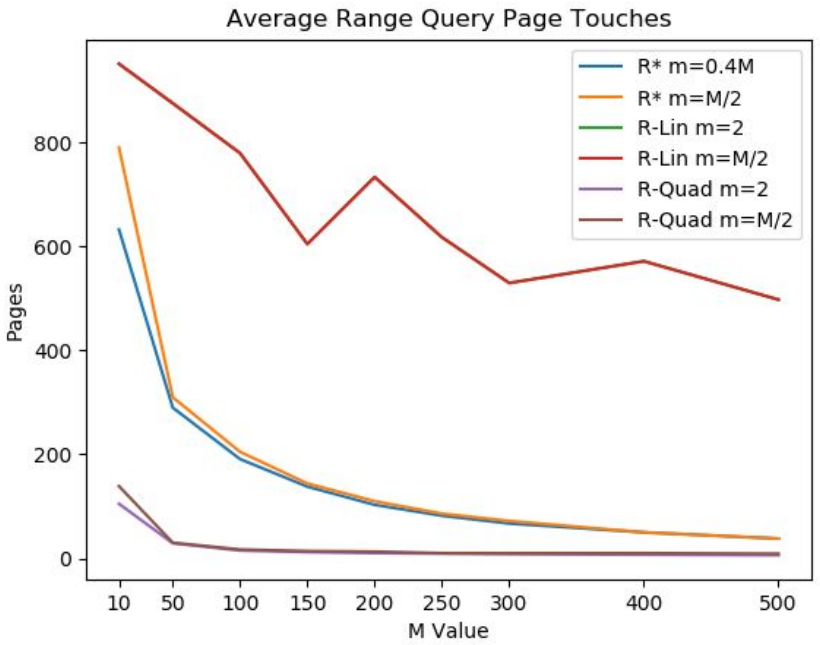
R-Tree Performance Compare



From the graph we can observe the following:

- Every R-Tree and its variation, R* Tree perform better than the Naive approach of the Range Query Problem.
- For each tree independently, the change of **m** does not affect in most cases very much the performance.
- R-Tree with linear split has increasing range query time while the other trees are keeping their times low.
- R-Tree with Quadratic split is faster than R*-Tree and keeps is average query time near 0 ms which indeed is true because the R-Tree answers all the range queries in 2 secs while R*-Tree needs 12 secs.

R-Tree Performance Compare



From the graph we can observe the following:

- All the different types of R-Trees does not show huge difference between their optimal value of **m** and the one that we gave explicitly, instead of both R-Trees where we can see in the start some difference where the optimal value performs better.
- R-Tree with Quadratic split starts with small value and continues with small value.
- R*-Tree starts with a high value but ends with a value near the one of the R-Tree with Quadratic Split which is optimal.

R-Tree Performance Compare

From the previous graphs we can conclude the following:

- The clear winner, paradoxically, is the **R-Tree with Quadratic Split** because:
  - It is much more faster in insertions that the R*-Tree.
  - It is much more faster in Range Queries than R*-Tree.
  - This might happen because R*-Tree does not do the Forced Reinsert.

- The Range Query average time of every R-Tree Variation is much more lower than the average time of the Naive Approach, thus the use of R-Trees are proved to be a good approach for answering Range Queries.
- Finally, we can say that the average time complexity of DBSCAN with the use of R-Trees, to answer the Range Queries is $\mathcal{O}(nlogn)$.

- Note that, that if we even have to insert the given data in an R-Tree before running DBSCAN, we don't have to add the data one by one but instead we can use efficient bulk loading algorithms as the one mentioned in [4].

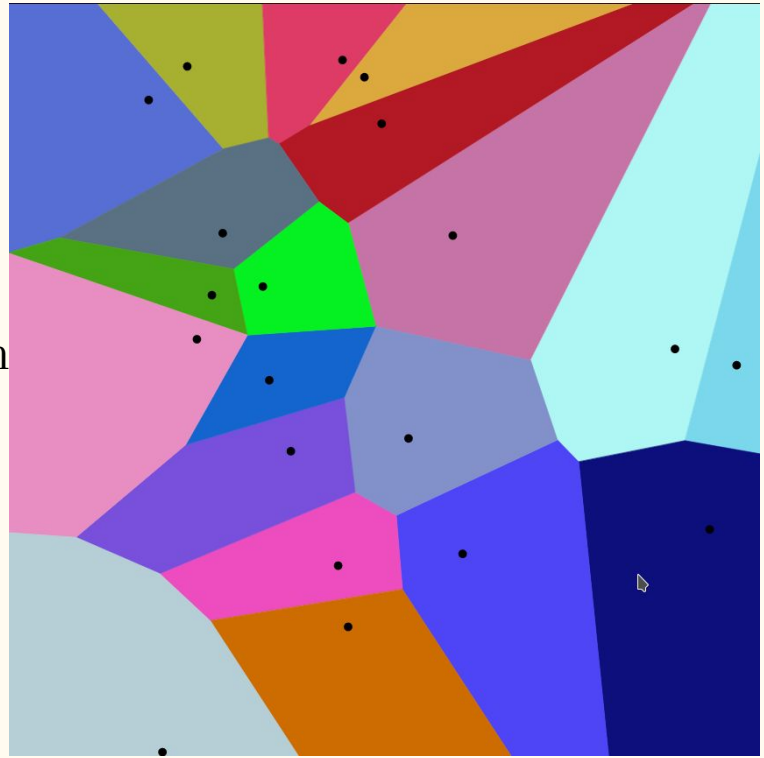
Voronoi Diagram

A Voronoi diagram is a partition of a plane into regions close to each of a given set of objects

It has a finite set of points $\{p_1, ..., p_n\}$ in the Euclidean plane.

Each cell includes a point or else a **site**

and it's called a **Voronoi Cell.**



Voronoi Diagram

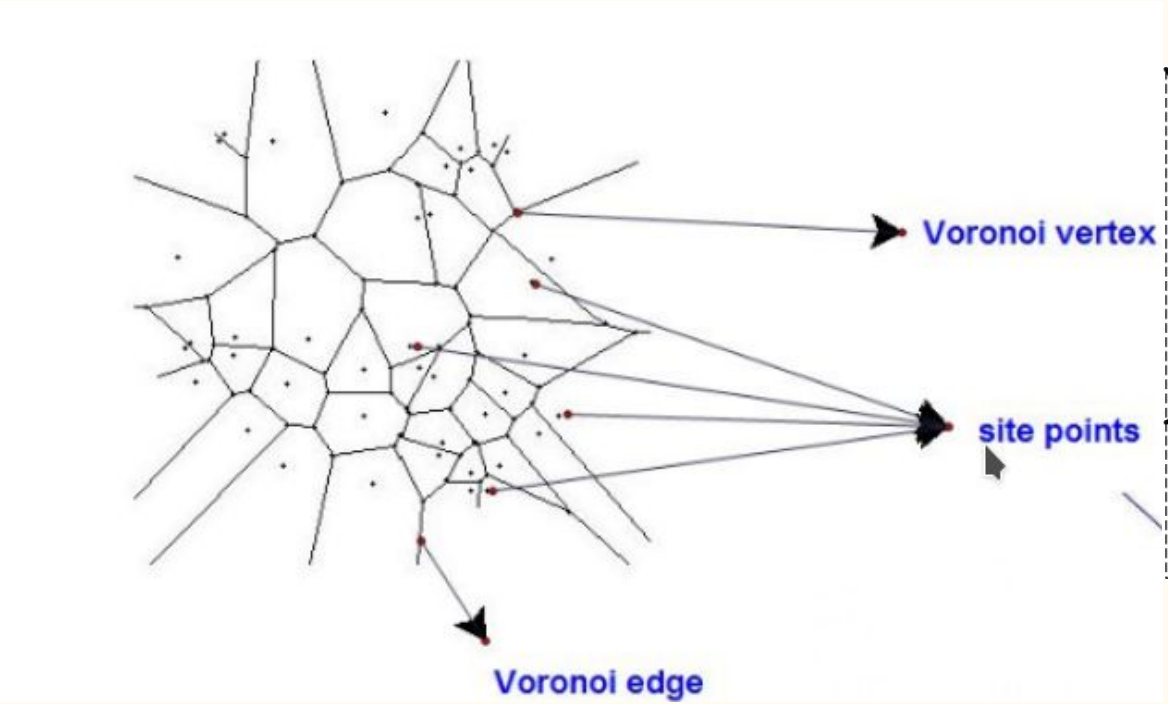
Definition:

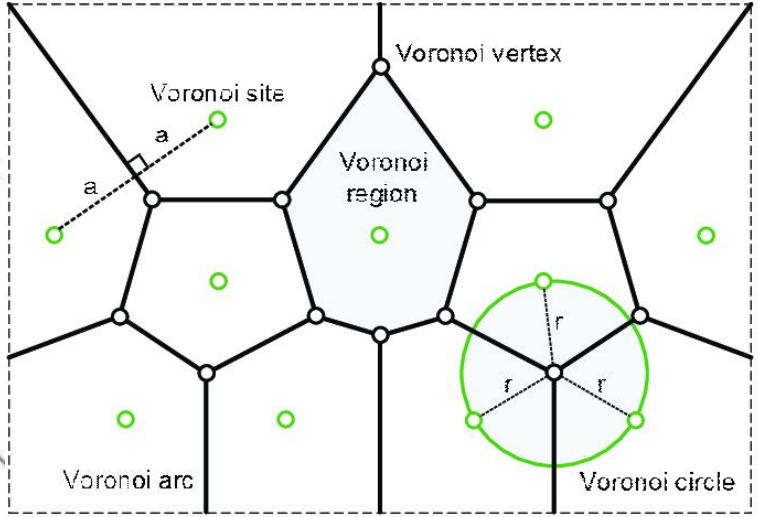
Let S = $\{p_1, ..., p_n\}$ be a given set of n points in an m-dimensional Euclidean space and let d(a, b) denote the distance between the points a and b in this space.

The Voronoi diagram of S is defined as the subdivision of the space into n cells, one for each point in S. A point u lies in the cell corresponding to the point pi if d(u, pi) $<$ d(u, pj) for each pj $\in$ S and j $\neq$ i.

Voronoi Diagram

Voronoi Diagram





Voronoi Diagram

An easy way to think of voronoi diagram is to create circles from each site with the same radius at the same time. As the circles become bigger, they collide with each other.

Two circles together will create a line between them which is called a **Voronoi edge.**

Three(or more) sites create a point in the intersection of their voronoi edges and this point is called a **Vertex.**

Voronoi Diagram

Properties:

- Two nearby cells have a common voronoi edge. The distance between the edge and the sites of the cells are equal from every point of the voronoi edge.
- Three cells create a point in the intersection of their voronoi edges. This point is called a vertex and it has the same distance from every site of these three cells.
- Each vertex has a circle which radius is the distance from the nearby sites. Thus the surroundings sites are on the boundary of that circle.

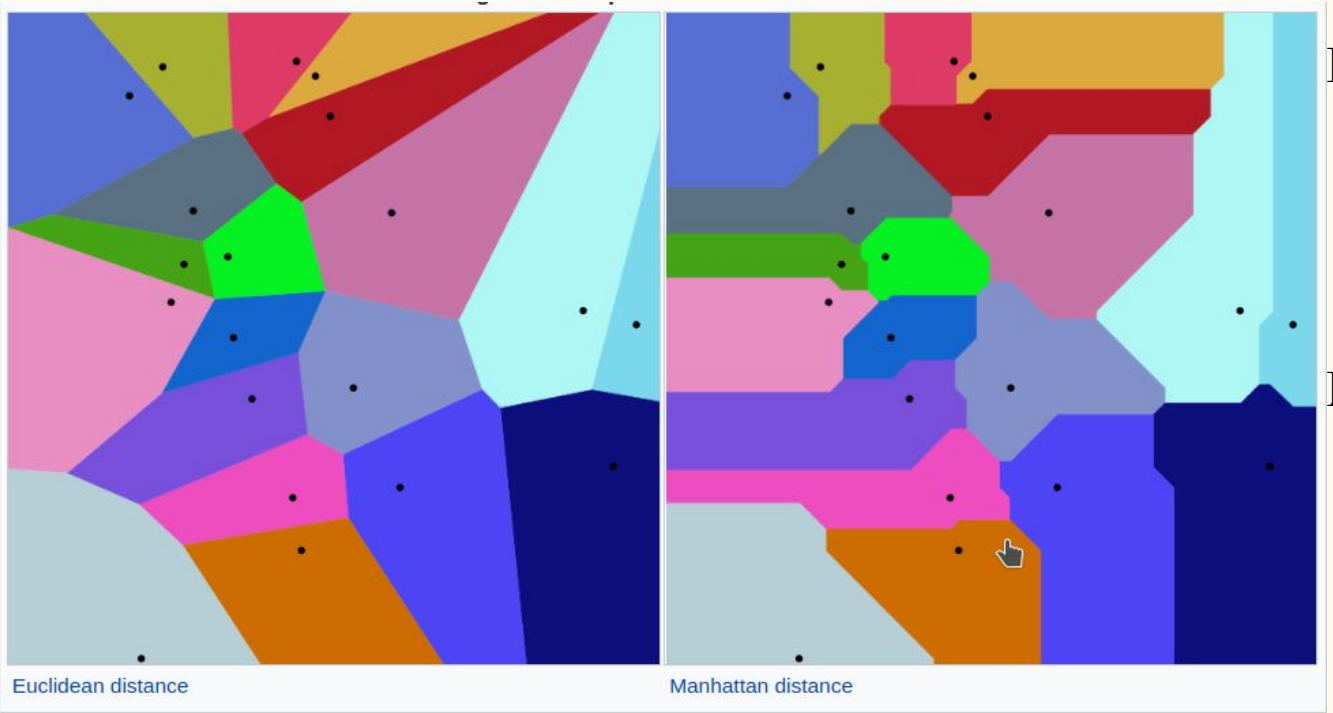
Voronoi Diagram

Properties:

- There are maximum $2n - 5$ vertices in a Voronoi diagram of n points.
- There are maximum 3n-6 edges in a Voronoi diagram of n points
- Number of vertices + Number of edges + Number of points = 1

Voronoi Diagram

To measure distance we have several metrics. The most common are:



Euclidean

Manhattan

Voronoi Diagram

Delaunay Triangulation:

For a given set P of discrete points in a plane **Delaunay Triangulation** is a triangulation such that no point in P is inside the circumcircle of any triangle.

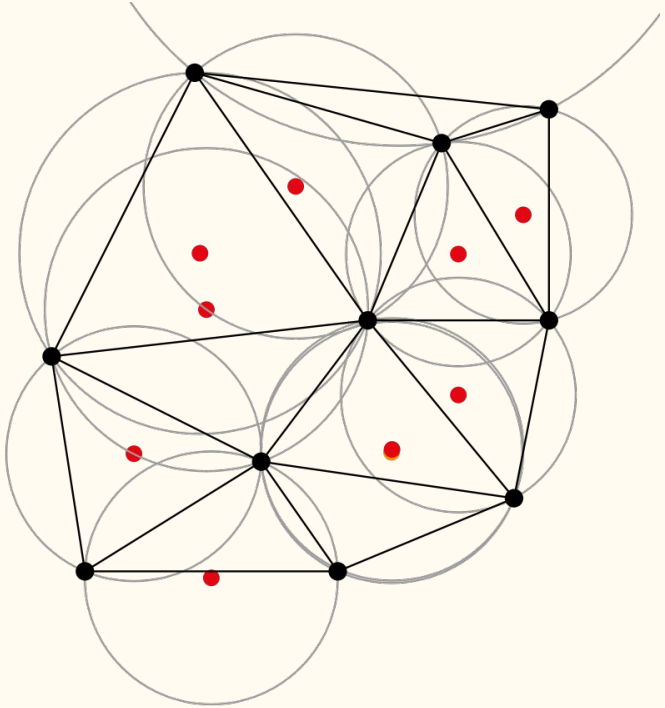
We can extract the delaunay triangulation from a voronoi diagram if we connect all the sites of the surrounding cells.

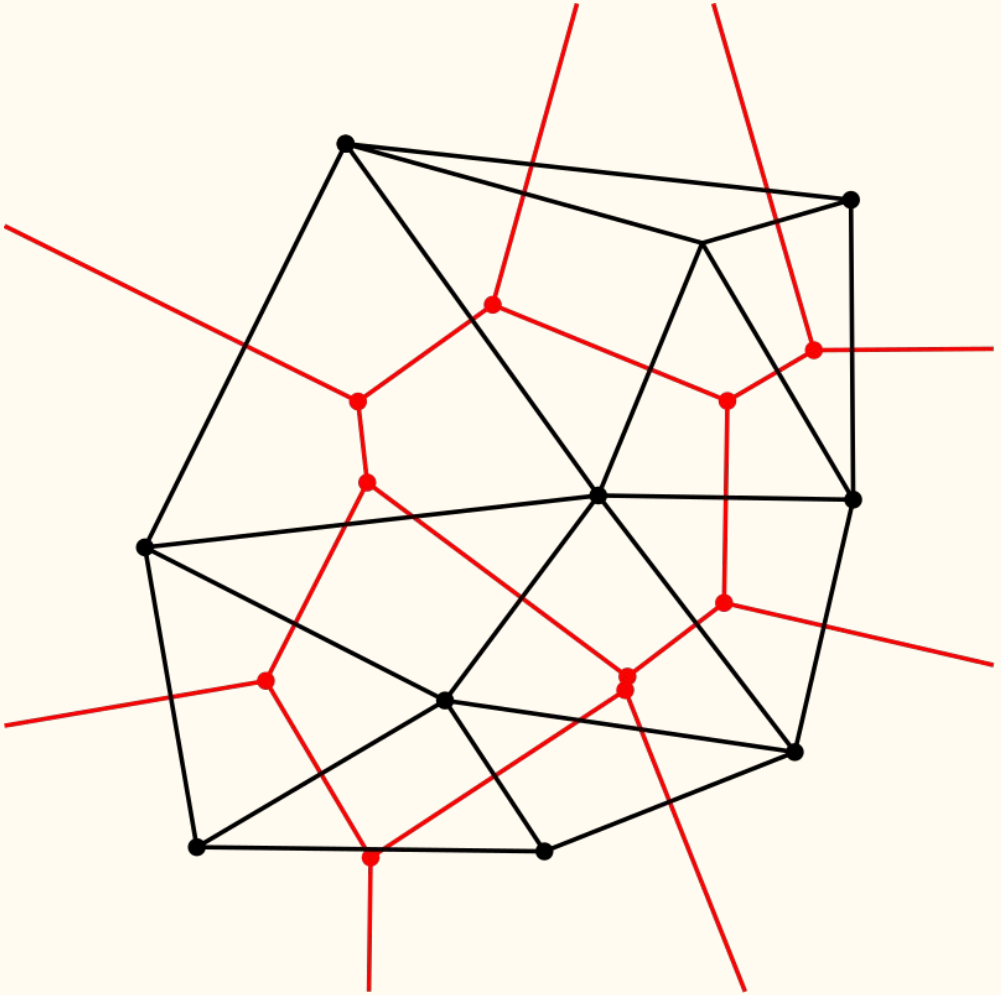
The vertices of the voronoi diagram are the nodes of the delaunay triangulation

It is unique for each voronoi diagram.

Voronoi Diagram

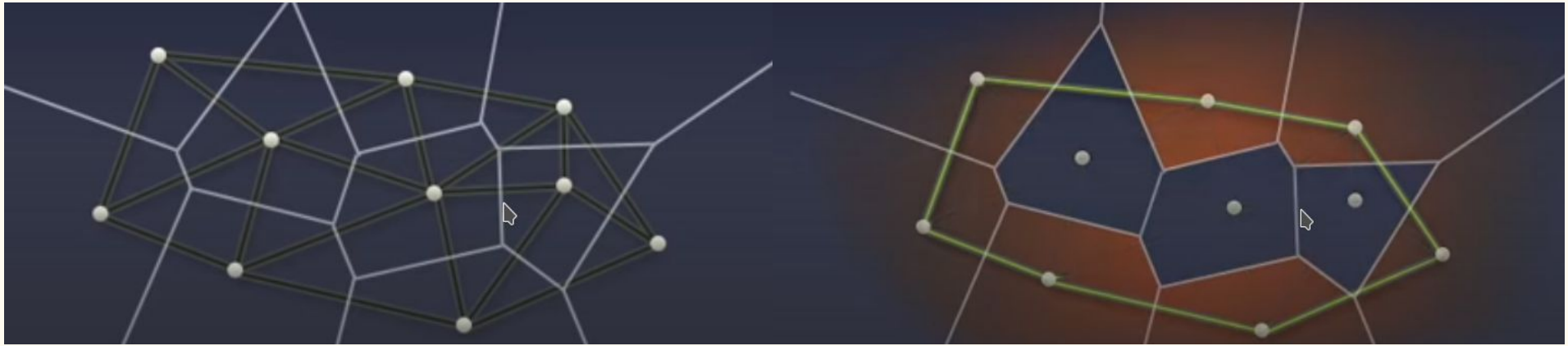
Delaunay Triangulation:





Voronoi Diagram

If we take the outer sites of the delaunay triangulation we have the convex hull



Voronoi Diagram

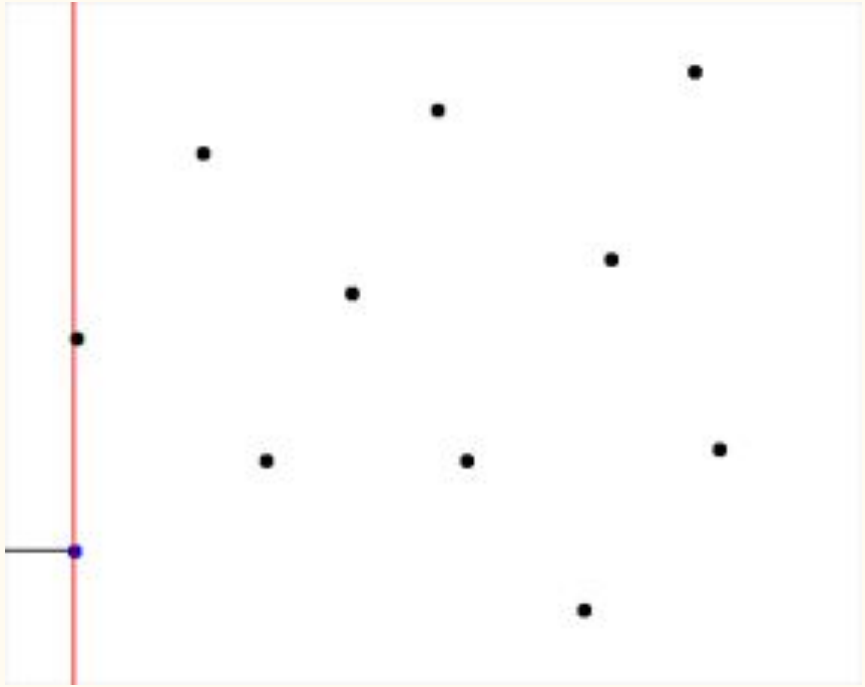
Algorithms:

Brute force : $O(n^3)$

Fortune's Algorithm: O(nlogn) (Space Complexity: O(n))

Bowyer-Watson: O(nlogn) Worst Case: $O(n^2)$

Voronoi Diagram

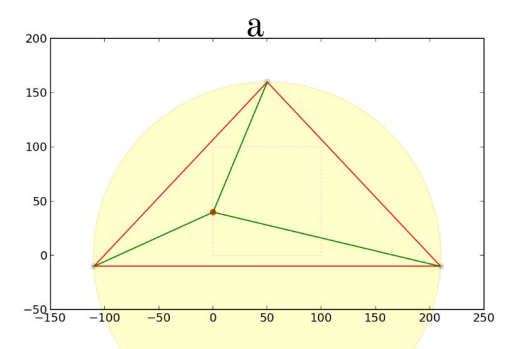


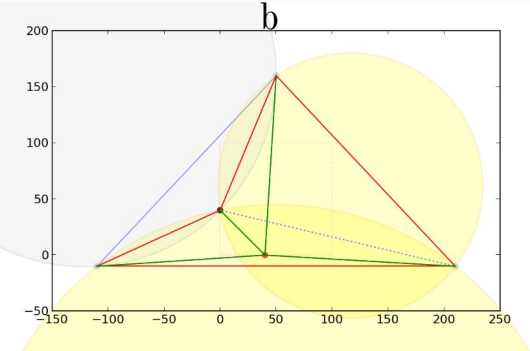
Fortune’s algorithm

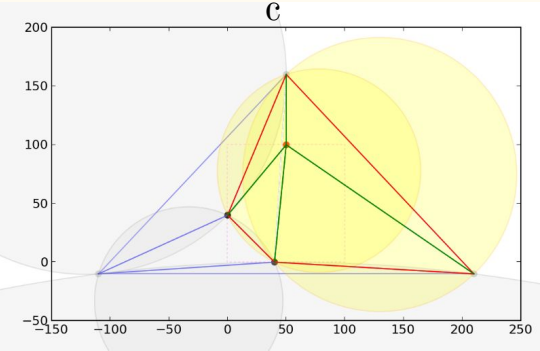
https://en.wikipedia.org/wiki/Fortune's_algorithm

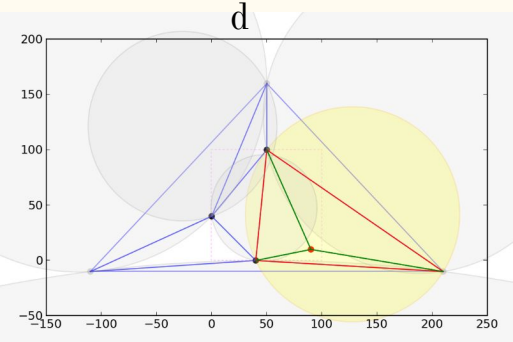
Voronoi Diagram

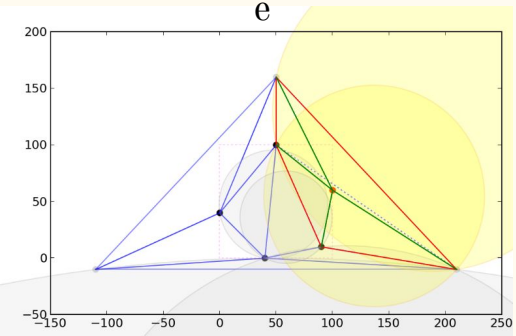
Bowyer-Watson https://en.wikipedia.org/wiki/Bowyer-Watson_algorithm

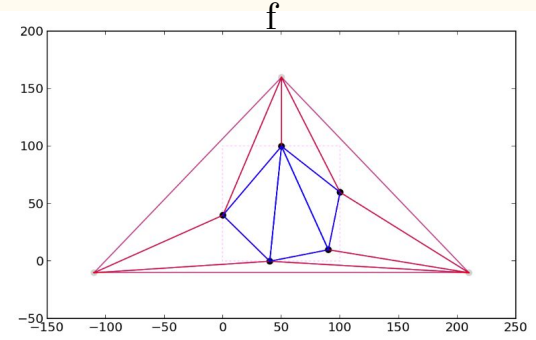












Clustering

Clustering is an efficient data mining technique to discover inherent structure of a given dataset.

The problem of clustering in general deals with partitioning a dataset consisting of n data points into k ($k \leq n$) distinct set of clusters such that the data points within the same cluster are more similar to each other than to the data points in other clusters.

K-Means Initialization with Voronoi Diagram

- Computational Geometry tools cannot be used only to make the algorithm more efficient in time perspective but they can also used to improve the results of an algorithm.
- For instance, K-Means algorithm may be a fast algorithm but the final results are dependent in the choice of the initial centers.
- Therefore, many algorithms for choosing effectively the initial centers have developed.
- Afterwards, we will present an algorithm that chooses wisely the initial centers with the use of Voronoi Diagrams

K-Means Initialization with Voronoi Diagram

The main steps of the algorithm as defined in [5] are the following:

- **Input:** Data Vectors, Number of Clusters k
- **Output:** k points that will work as the initial centers of k-means algorithm
- *ccenter:* set of points, initially empty.
- Creation of the Voronoi Diagram of the initial points.
- Sort the vertices of the Voronoi Diagram in descending order of the radius of their largest empty circle.

K-Means Initialization with Voronoi Diagram

- For each vertex, while *ccenter* has less than *k* points
  - Insert the points that are on the circumference of the largest empty circle into a set *Test*
  - If two points of set *Test* have distance less that the radius of the current circle we delete one of the points
  - If *ccenter* is empty then add the *Test*'s points in *ccenter*, then empty *Test*.
  - else, we store in a set *Temp* the points of *Test* that have distance with some *ccenter* point, less that the radius of current circle. After, we add the points of *Test-Temp* into *ccenter*. Finally, the two sets are emptied.

K-Means Initialization with Voronoi Diagram

The complexity of the algorithm is the following:

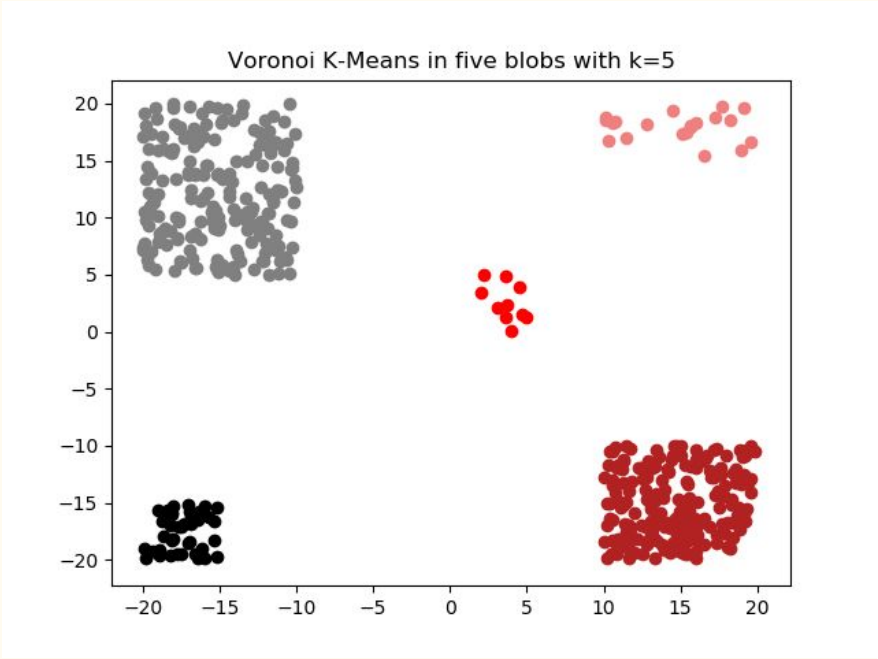
1. $\mathcal{O}(nlogn)$ for the creation of Voronoi Diagram.
2. $\mathcal{O}(nlogn)$ for the Sorting of the vertices by the radius of their Largest Empty circle(we suppose that we require logarithmic time to find the Largest Empty Circle radius and the points that lie on its circumference).
3. In the worst case the input points might form a circle so the algorithm will run only one iteration but in set *Test* might be all the points, so, the complexity of this step will be $\mathcal{O}(n^2)$
4. But this step on average does $\mathcal{O}(nlogn)$ time.

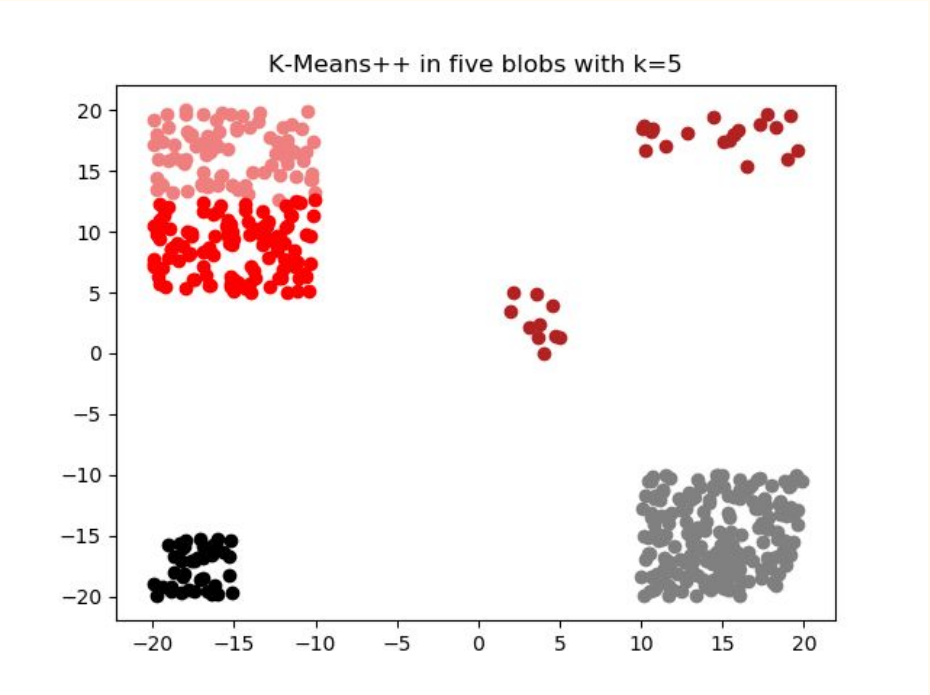
Therefore, the complexity of this algorithm is $\mathcal{O}(nlogn)$ on average, which might be dominated by the complexity of K-means which is $\mathcal{O}(kn\tau)$ where τ is the number of iterations.

K-Means Initialization with Voronoi Diagram

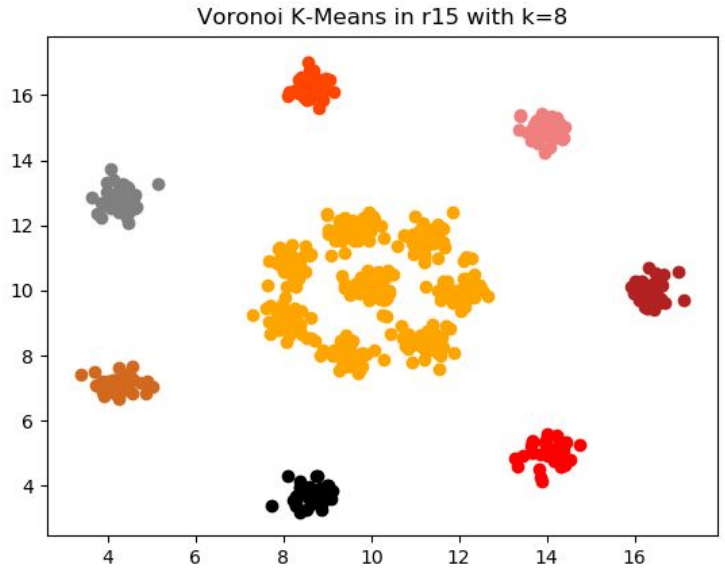
- We wrote the algorithm in Python.
- In order to check how good performs we did experiments in 3 datasets that K-Means should be efficient.
- We compared the performance of our algorithm with K-Means++ which **sklearn** library thinks that its best initialization method.

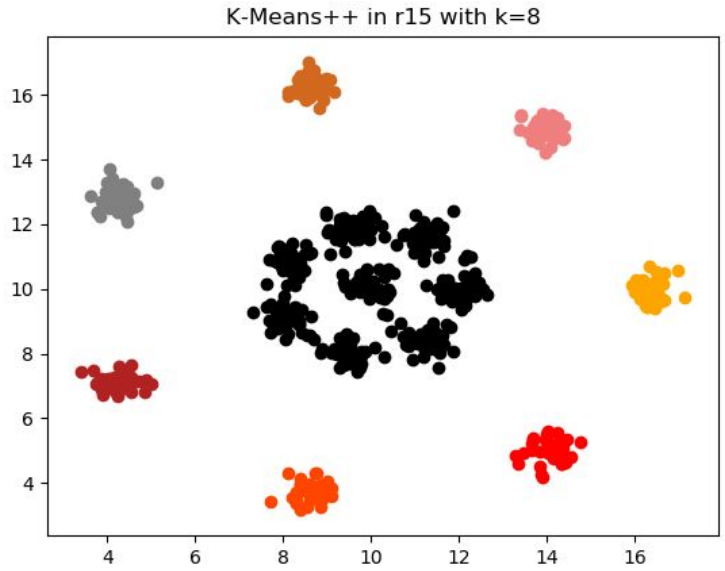
K-Means Initialization with Voronoi Diagram



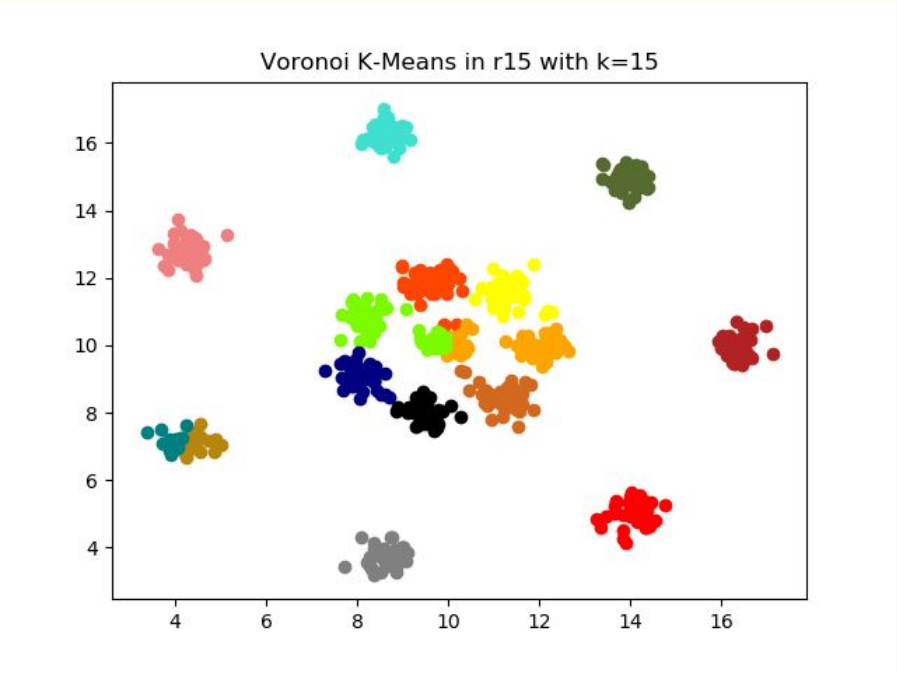


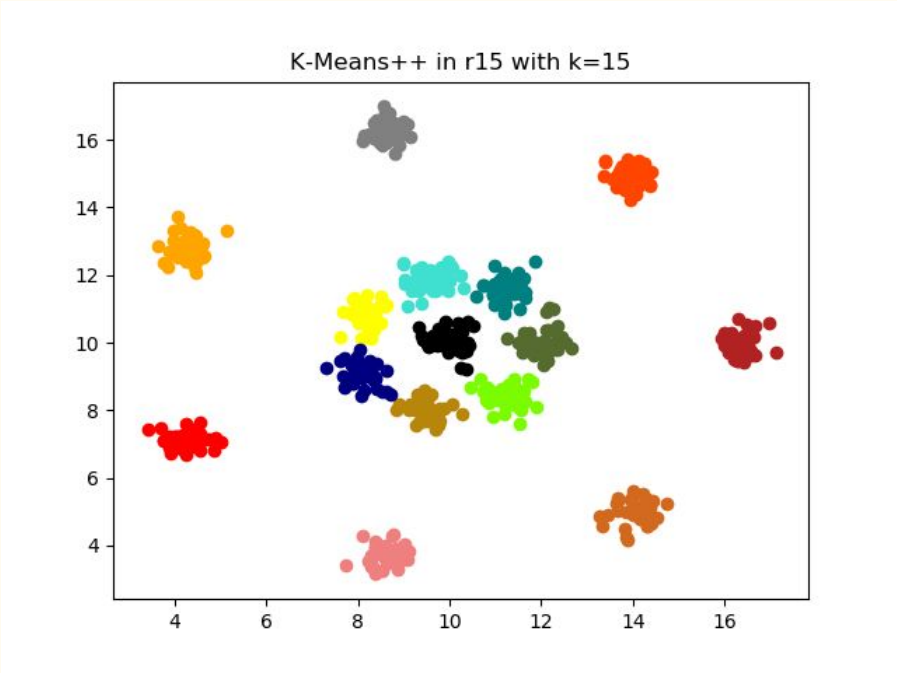
K-Means Initialization with Voronoi Diagram



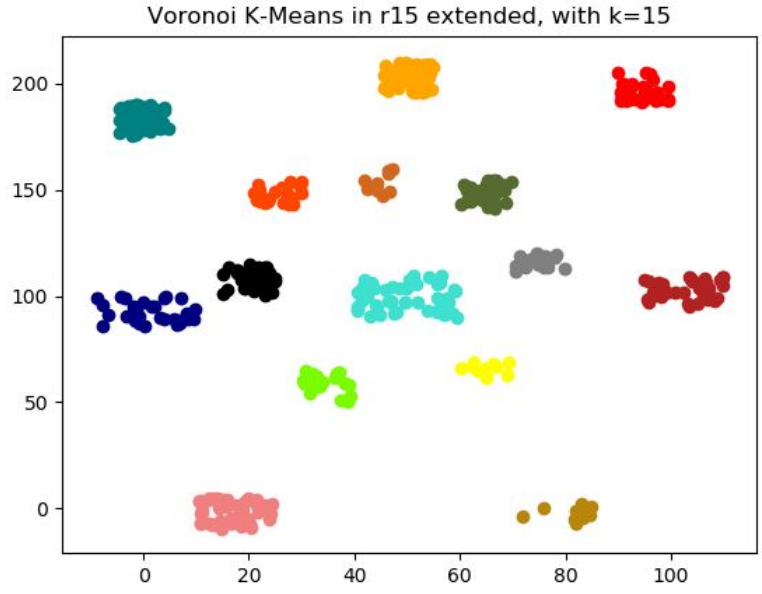


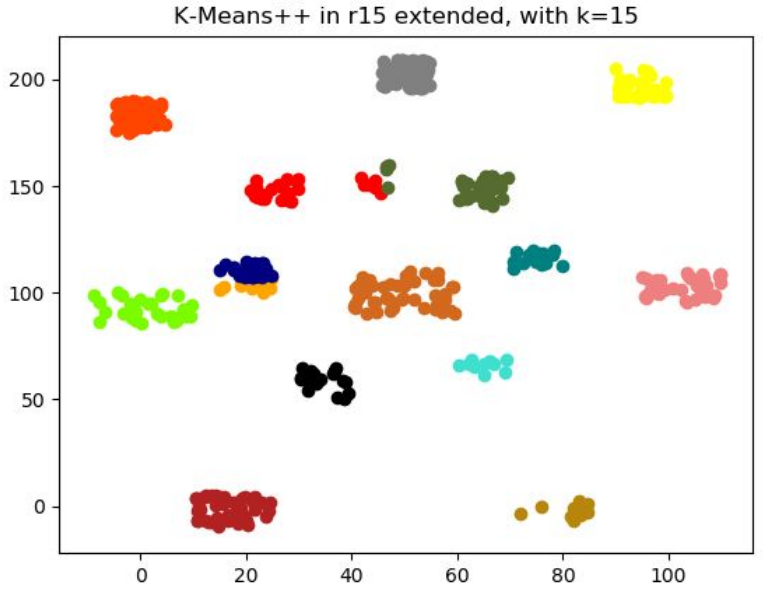
K-Means Initialization with Voronoi Diagram





K-Means Initialization with Voronoi Diagram





K-Means Initialization with Voronoi Diagram

From the results we conclude the following:

- The proposed algorithm at most times gives better results than K-Means++ provided that the parameter K has the proper value.
- From the results of **five blobs** and **R15 extended** we observe that the proposed algorithm works better than K-Means++ for clusters with varying number of samples.
- From the results of **R15** and **R15 Extended** we can see that the proposed algorithm is prone to the distance between the actual clusters because when we reproduced **R15** but with greater distances between the clusters (**R15 Extended**) the results were better.

Clustering Algorithm with Voronoi Diagram

Terms:

n, d: number of points, number of dimensions

S: a set of n points with d dimensions

Vor(S): the Voronoi Diagram of the set S

Vi: Ith voronoi vertex

CirS(v): Largest empty circle of v vertex (Empty circle is a circle where we don't have points inside it)

Clustering Algorithm with Voronoi Diagram

Terms:

R(v): Radius of CirS(v)

m: A threshold value. Maximum value of radius for the voronoi circle to separate the clusters

M: A threshold value. Maximum density(number of points) of clusters to detect the outliers

r: Temporary value of Ith radius

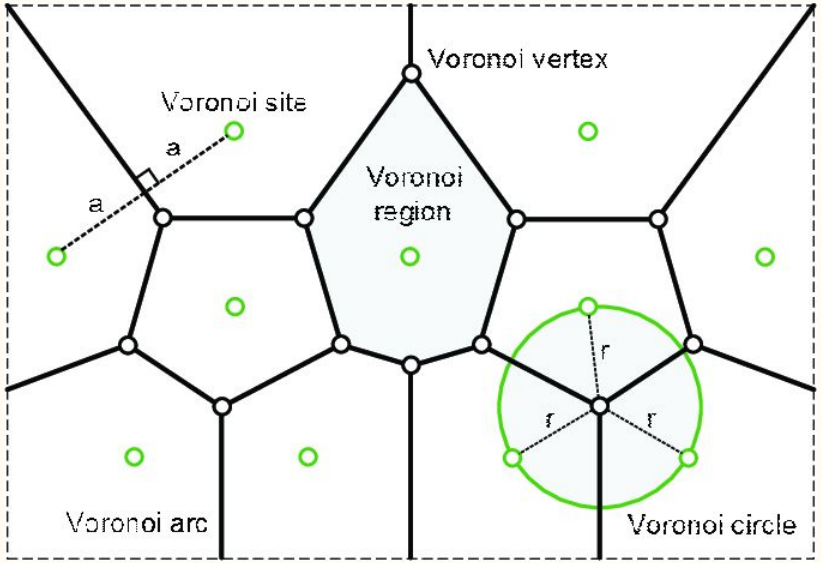
Clustering Algorithm with Voronoi Diagram

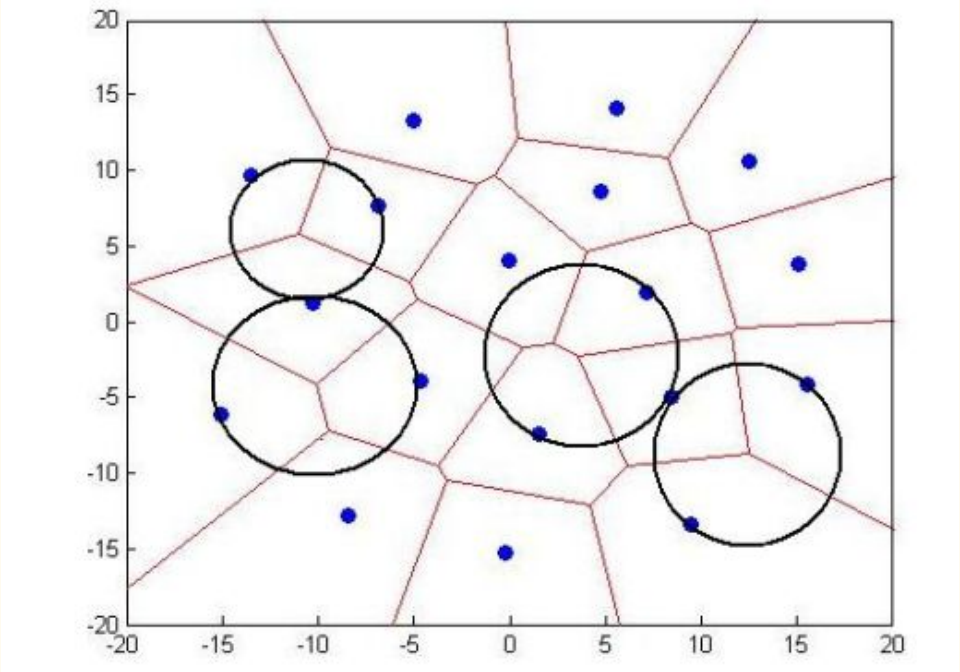
In brief - In simple words:

Initial Steps:

We start taking n points of a set S and we construct the Voronoi Diagram in order to create the cells in which these points are inside. From these cells we extract the Voronoi Vertices. We know that from these vertices empty circles are created. We call them empty because they do not have points inside them. We calculate the largest empty circles of each vertex and sort these circles based on their radius. The max number of these circles is 2n-5 because of the number of vertices.

Clustering Algorithm with Voronoi Diagram





Clustering Algorithm with Voronoi Diagram

In brief - In simple words:

For every circle:

Now that we have all the circles sorted we do the following steps from the smallest to the biggest circle:
If the radius of the Ith circle r $\leq$ m(radius threshold) then we find the points that are in the boundary of the Ith circle and add them to the prototype of that vertex. This vertex will be added in the set S'. If points that are found in the boundary of the circle are already found in other circles then ignore these points.

Clustering Algorithm with Voronoi Diagram

In brief - In simple words:

For every circle:

Now that we have all the circles sorted we do the following steps from the smallest to the biggest circle:
If the radius of the Ith circle r $>$ m(radius threshold) then we add the remaining points, the points that were not found in the prototypes of the previous circles, to the set S' and we stop searching for the next circles and thus break the loop.

Clustering Algorithm with Voronoi Diagram

In brief - In simple words:

This process is repeated until S=S'
If S is not the same with S' then we create the Voronoi diagram to the S' set, make the result our new set S and repeat the same process

With this algorithm we manage to have a cluster for every vertex that has a

radius $\leq$ m.

Clustering Algorithm with Voronoi Diagram

In brief - In simple words:

The final steps are to merge the clusters that we have created based on the m distance in order to create **k** clusters. In other words we are going to merge the vertices that are close together and thus have the points inside their circles look in a common point in the middle of their vertices. This point is going to be the centre of our new cluster.
Lastly we look all the clusters that we have and check if they have less density than the threshold value M. If they do we call them outliers.

Clustering Algorithm with Voronoi Diagram

Algorithm:

- Given the initial points we initialize the set S with these points
- We apply the Voronoi Diagram algorithm to that set S
- We want to find minimum number of vertices(prototypes) to cover the initial points

Clustering Algorithm with Voronoi Diagram

Algorithm:

- Having the voronoi vertices we sort the circles of these vertices by their circle's radius in ascending order
- We then consider the least radius circle with it's points in the boundary of the circle to form the initial cluster
- We do the same for the next circles, until all points are covered, and create the rest initial clusters

Clustering Algorithm with Voronoi Diagram

Algorithm:

- After the formation of all the initial clusters we merge them in a way to have the desired clusters.
- To achieve that we give the voronoi vertices as the initial points where each vertex is a prototype.
- This process is repeated until the voronoi diagrams in two iterations are same.

Clustering Algorithm with Voronoi Diagram

Pseudo Code:

Step 1:

Construct the Voronoi Diagram of n points in S

Step 2:

Sort all the voronoi vertices by the radius of their largest empty Voronoi circle(CirS(Vi)) and store these vertices in V[]

Clustering Algorithm with Voronoi Diagram

Pseudo Code:

Step 3:

For i = 1,2 .... 2n-5 repeat steps 4 to 6 (2n-5 maximum number of vertices)

Step 4:

Assign radius of the largest empty circle of Vi(CirS(Vi)) to r (r = R(V[i]))

Clustering Algorithm with Voronoi Diagram

Pseudo Code:

Step 5:

If r from step 4 is less or equal than m(radius threshold) then insert the vertex Vi in S' and add the points in the boundary of CirS(Vi) in the prototype Pi

Step 6:

If r is bigger than m then add the uncovered points in S' and exit the loop(hence go to step 8). Uncovered points are the points that are not in the prototype Pi

Clustering Algorithm with Voronoi Diagram

Pseudo Code:

Step 7:

If S = S’ go to step 8, else construct the Voronoi Diagram of S’(Vor(S’)) and make S the vertices of this Voronoi(S=Vor(S’)) and go to step 2

Step 8:

Merge the points of S’ based on m threshold to create k groups/clusters{C1,C2,...,Ck}.

Clustering Algorithm with Voronoi Diagram

Pseudo Code:

Step 9:

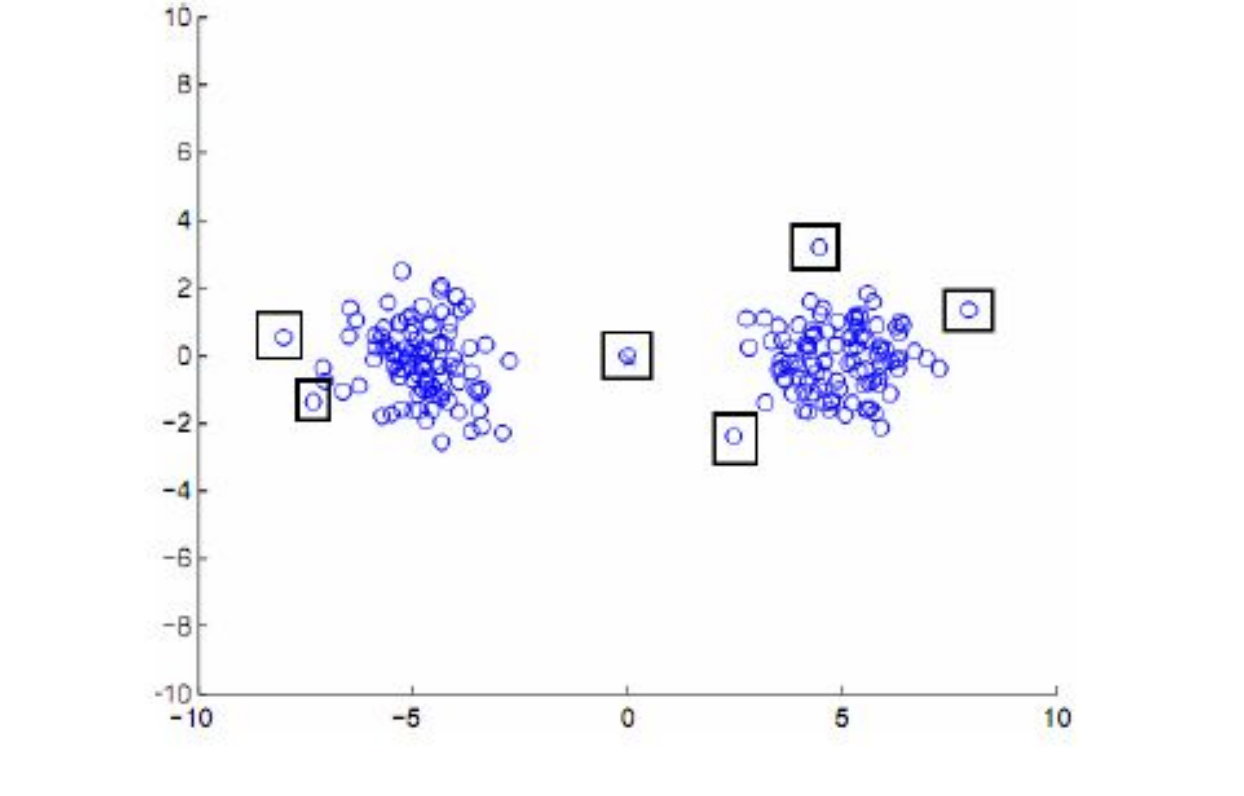
For i =1 to k

if Ci < M then Ci is an outlier

*Outlier: A cluster that is significantly different from the remaining clusters

Clustering Algorithm with Voronoi Diagram

Outlier example



Clustering Algorithm with Voronoi Diagram

Complexity:

Step 1:

O(nlogn) for the construction of Voronoi with n data

Step 2:

O(nlogn) for sorting

Step 3:

2n-5 times of the Steps 4, 5 and 6

Clustering Algorithm with Voronoi Diagram

Complexity:

Step 4, 5, 6:

O(1) to detect the prototypes, the uncovered points and add them in S'

Step 2-6:

$O(n\log n) + O(2n-5) = O(n\log n) + O(n) = O(n\log n)$

Step 2-7:

It's repeated a finite number of times, say k times

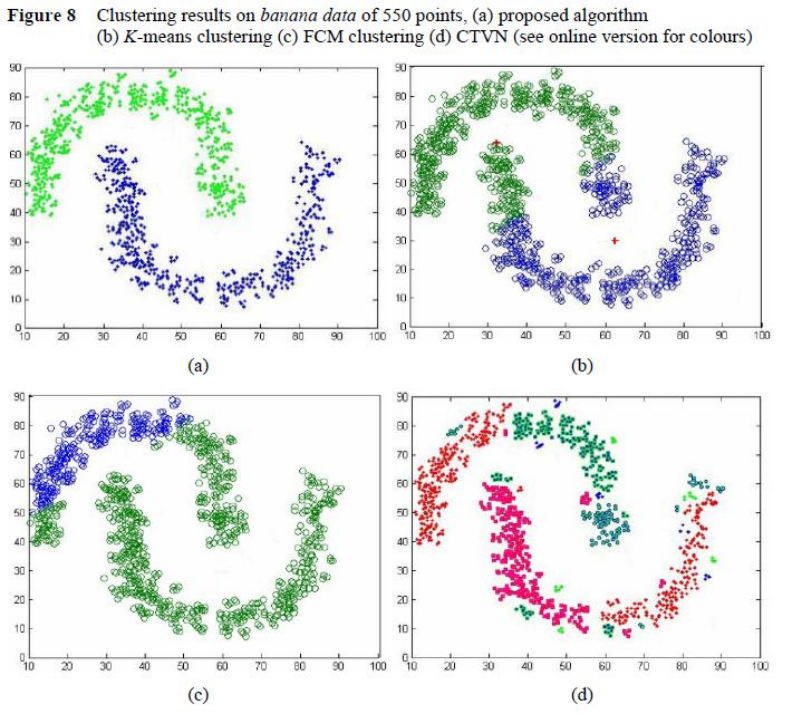
Clustering Algorithm with Voronoi Diagram

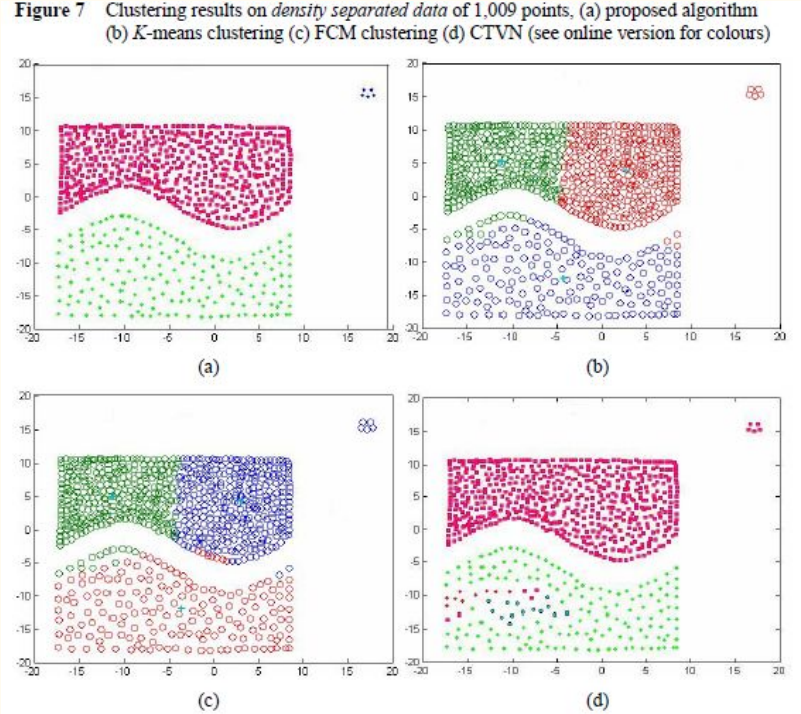
Complexity:

Overall complexity:

k*(O(nlogn)+O(nlogn)+O(n)) = **O(knlogn)**

Clustering Algorithm with Voronoi Diagram





Clustering Algorithm with Voronoi Diagram

Figure 5 Clustering results on *outlier data* of 1,500 points, (a) proposed algorithm (b) *K*-means clustering (c) FCM clustering (d) CTVN (see online version for colours)

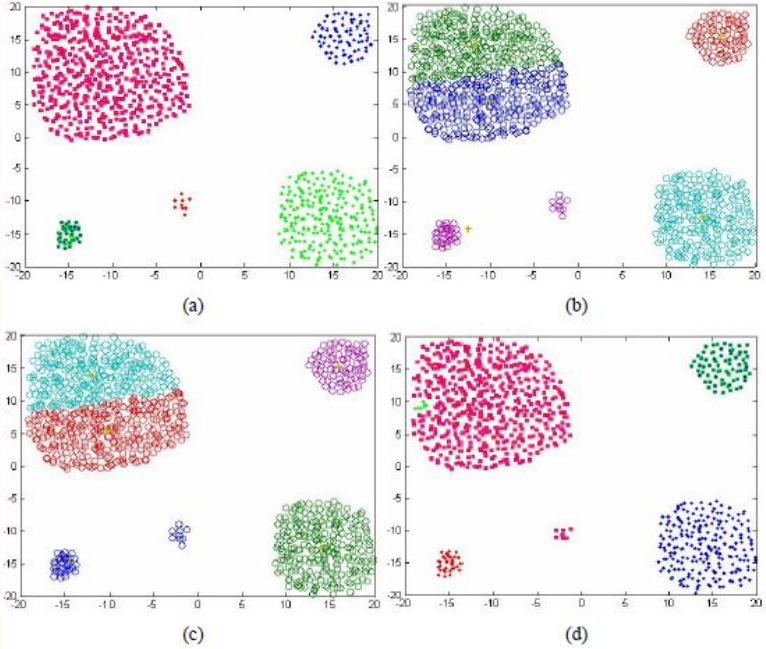
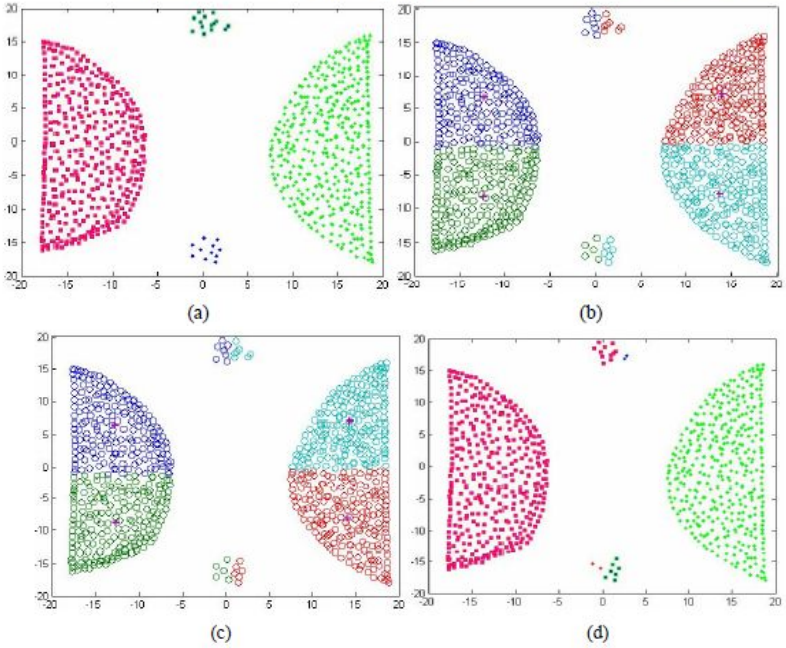


Figure 6 Clustering results on *half-circle data* of 1,250 points, (a) proposed algorithm (b) *K*-means clustering (c) FCM clustering (d) CTVN (see online version for colours)



Conclusion

- We presented how can the Computational Geometry field can contribute to the problem of Clustering.
- Computational Geometry tools can contribute to Clustering algorithms not only to make them faster, but also to make them produce better results.
- Furthermore, Computational Geometry can be used to invent new and efficient clustering algorithms

Thanks for your attention!!!

References

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

[2] Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: Proceedings of ACM management of data (SIGMOD), Boston, 18–21 June 1984, pp 47–57

[3] Sellis, Timos. (2000). Review - The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles.. ACM SIGMOD Digital Review. 2.

[4]Lee, Taewon and Sukho Lee. "OMT: Overlap Minimizing Top-down Bulk Loading Algorithm for R-tree." *CAiSE Short Paper Proceedings* (2003).

[5]Reddy, Damodar & Jana, Prasanta. (2012). Initialization for K-means Clustering using Voronoi Diagram. Procedia Technology. 4. 395–400. 10.1016/j.protcy.2012.05.061.

[6]Damodar Reddy, Prasanta K. Jana. "A new clustering algorithm based on Voronoi diagram", January 2014, International Journal of Data Mining Modelling and Management 6(1):49 - 64, DOI: 10.1504/IJDMMM.2014.059977