Computational Geometry Applications in Clustering

Nikolaos Lekkas, Emmanouil Lykos

National & Kapodistrian University of Athens

Abstract

Clustering is a data mining tool that has been researched for a long time. We introduce here three applications that make existing clustering algorithms more efficient. These are R-Tree, R*-Tree and Voronoi K-means Initialization. R and R*-Tree are used in order to answer efficiently range queries, which is a common subtask in clustering algorithms. The Initialization algorithm helps us give better initialized points in K-means in order to have better clustering results. Then we introduce a new clustering algorithm which is based on Voronoi diagram. The use of Voronoi diagram is that it provides a partition of space into cells and therefore clusters. In our approach we search the largest empty Voronoi circles in order to locate closer points(prototypes) represented by the Voronoi vertices. The points represented by these prototypes are going through a new Voronoi diagram iteratively to produce the desired clusters. We perform experiments on several datasets in order to see the desired results of the algorithms. The proposed algorithm performs better in discovering the desired clusters and it is able to detect the outliers.

Keywords: Computational Geometry, Clustering, Spatial Indexing, Voronoi Diagram

1. Introduction

Computational Geometry can be applied to various settings, such as computer graphics, machine learning, robotics etc. For instance computational Geometry can be applied to unsupervised learning. Unsupervised learning is the type of machine learning that its task is to learn patterns from data that are not attached with any labels. Thus, unsupervised learning contains different families of algorithms. One family of these algorithms are the clustering algorithms. Clustering is the task of grouping objects, in order to, the most similar belong in the same group while the most different belong to a different one. More formally given n points $p_1, p_2, \ldots, p_n$, k centers $c_1, c_2, \ldots, c_k$ should be found in order to minimize the following function:

$$\sum_{i=1}^{n} \min_{j} \left\{ distance\left(p_i, c_j \right) \right\}$$

Because clustering has various applications in different fields, such as business and marketing, it has become a subject of vast research, therefore, it would be useful to present how the field of

Computational Geometry contributes to it. Computational geometry can be applied to the clustering problem by using its tools to make the existing clustering algorithms more efficient or by inventing new algorithms that are using computational geometry tools. The rest of the paper is organized as follows. Section 2, presents the theoretical background needed to understand the rest of the paper. Section 3, presents how Computational Geometry can contribute to improve the performance of existing clustering algorithms. Section 4, presents a clustering algorithm that uses the Voronoi Diagram in order to cluster the given points. Section 5, presents the experimental evaluation of the mentioned approaches against the naive or existing applications that seem to be the best ones. Section 6, presents the conclusions deducted from our work.

2. Background

2.1 DBSCAN

DBSCAN[1] is a density based clustering algorithm which seems to perform well when the data does not have varying densities and when radius and minimum points inside that radius were chosen wisely. Another advantage of this algorithm is that he identifies the noisy points of the dataset, therefore, the noisy points cannot affect negatively the result(when the hyperparameters of the algorithm are chosen wisely). The pseudocode of the algorithm as defined in [1] is the following:

1. Input: Radius r, Minimum points inside r, $minPts$.
2. Current cluster $c = 0$
3. For point p in the database
    a. If p is not unclassified continue
    b. Else do a Range Query to get the set N that contains all its neighbors within radius r
    c. If $|N| < minPts$ the point will be classified as Noise
    d. Else we begin a new cluster $c = c + 1$ and we insert the $N \backslash \{p\}$ into a queue Q
    e. While Q is not empty
        i. Pop a point q from Q
        ii. If point is Noise or Unclassified then assign it into cluster c, else continue to next point.
        iii. Do a Range Query to get the set N that contains all neighbors of q within radius r.
        iv. If $|N| < minPts$ then q is a border point, thus we do not do anything, else q is a core point and we add $N \backslash \{q\}$ into Q.

Because, the algorithm calls the Range Query once per point the complexity of the algorithm depends on the complexity of the Range Query. If the Naive method, that checks the distance of all points from the given point, is used to answer the Range Query the complexity of the algorithm is $O(n^2)$.

2.2 Voronoi Diagram

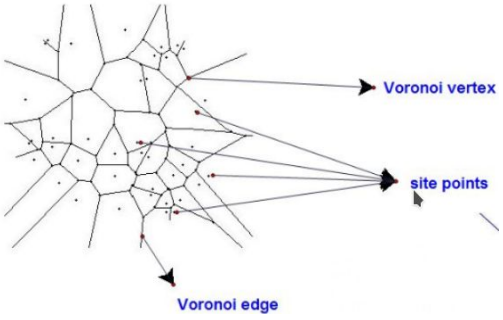
Definition:

Let $S = \{p_1, \ldots, p_n\}$ 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 p_i if $d(u, p_i) < d(u, p_j)$ $\forall pj \in S \; and \; j \neq i$.

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 $\{p_1, \ldots, p_n\}$ in the Euclidean plane. Each cell includes a point or else a site and it's called a Voronoi Cell. 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.



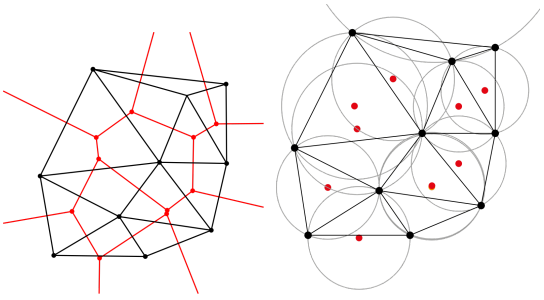
The basic properties of the Voronoi Diagram are the following:

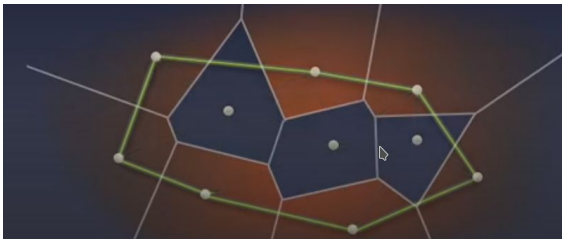
- Two nearby cells have a common voronoi edge. The distance between the edge and the sites of the cells are equal from every point of the voronoi edge.
- Three cells create a point in the intersection of their voronoi edges. This point is called a vertex and it has the same distance from every site of 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.
- 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

To measure distance we have several metrics. The most common are the Euclidean and Manhattan distance.

2.3 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.





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

3. Improving Algorithms

3.1. R-Tree

In [1], is proposed the use of R-Tree in order to improve the time complexity of the algorithm from $O(n^2)$ to $O(n\log(n))$ because R-Tree's height with n points is at most $O(\log(n))$. R-Tree[2] is 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, Intersection and more. Moreover, Because R-Tree partitions the multidimensional space into smaller rectangles and these rectangles into other rectangles until we reached the objects, it is suitable for multi-dimensional objects of non-zero size like countries and building on a map and are pretty effective in queries like "Find all countries that lie in a radius N of a specific point in a map". As said in [2] R-Tree is mostly preferred over other spatial indexing types because the rectangle borders are dynamically configured and because they are used in SDBMS, they should deal with secondary memory, which they do. In terms of structure, R-Tree is a height-balanced tree like a B-Tree-that's why also it is called multidimensional B-Tree-and maintains some of its properties. Every 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^{th} axis. Every internal node has the structure

$$(I, child-pointer)$$

where $child-pointer$ is a pointer to its child node. An example R-Tree is shown in Figure 1.

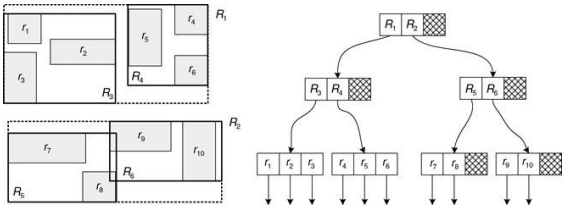


Figure 1: R-Tree Example

R-Tree has the following properties:

- Every node except the root node has $[m, M]$ records where $2 \le m \le M/2$ 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.

R-Tree Operations

The R-Tree Search algorithm just does a recursive search by checking at each node's record if the query bounding box has intersection with record's bounding box. If it has, the algorithm is called recursively to his child node if it is not leaf else the object that holds is returned. Though that the search algorithm in worst case can achieve worst performance than just linearly searching the rectangles, it is proven that has an average complexity of the height of the tree, specifically $O(\log(n))$. Note that, that this can be the complexity of the Range Query because if we want to get the points that lie in a radius R from some point (x, y) we just form the rectangle that has bounds $[[x-R, x+R], [y-R, y+R]]$ and from the points that Search returned we take the

ones that has distance less or equal than R from the given point.

The R-Tree's deletion and insertion algorithm are mentioned in [2] and they are like the B-Tree respective algorithms with difference that in insertion the tree is traversed with a user-customed way until a leaf node is reached and in deletion when an underfull node is encountered, R-Tree does not apply a fusion strategy, but instead the records are reinserted in the tree.

R-Tree Heuristics

The goodness of R-Tree's organization depends clearly on the routine that chooses the best subtree to insert the new record, and on the routine that splits an overfull node. These routines, if they give valid results, can do whatever we want, for example, choose subtree routine might choose the first subtree of a node and the split node routine can just create the first node by taking the first half of records while the second node will be the rest records. But, these methods will not result in a proper organization of the tree, which is of paramount importance because a good organization will result in faster Range Query execution. Therefore, R-Tree node records should optimize some value, which is called a heuristic. Specifically, R-Tree wants to minimize the sum of the areas of node's records bounding boxes. Thus, the choose subtree routine just chooses greedily the child node of a record that its bounding box needs the least area enlargement among the other node's bounding boxes. Ties are resolved by choosing the child node of the record who has the smallest area.

R-Tree node splitting algorithm has the purpose to split the overfull node into two groups, such that the sum of the areas of the minimum bounding boxes of the two groups is minimized. In [2], three approaches are suggested. The first one is the brute force approach which is a slow solution because it has exponential time complexity. The second solution is called Quadratic split and is a greedy approach. The Quadratic Split algorithm is the following:

1. 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 $d = area(J) - (area(P.boundingBox) + area(Q.boundingBox))$ among all pairs (where J is the Minimum Bounding Box that encloses the bounding boxes of P and Q).
2. Afterwards, while we can choose from 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.
3. 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.

Quadratic split algorithm has $O(M^2)$ time complexity, where M is the maximum number of records that a node can hold. In order to prove that we should check the complexity of each step:

1. The first step requires $O(M^2)$ time because we should check every pair of nodes and determine which is the most suitable.
2. Assuming that we can find the new bounding boxes in constant time the loop of steps 3 and 4 needs $O(M^2)$ 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.

Thus, the Quadratic split algorithm requires in total $O(M^2)$ time.

Linear Split is a splitting algorithm that has linear time complexity and obviously is faster than the

Quadratic Split algorithm, and can be used when we have too many data to insert on the R-Tree, thus we want a fast solution if we do not want to use a fast bulk-loading approach as the one mentioned in [4]. This algorithm might be naive but in [2] it is shown that performs well even in search queries. The steps of the algorithm are the following:

1. 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.
2. Normalize their separations by dividing with the total width along the current axis.
3. The seeds(or the seed) are the rectangles(rectangle) that have the greatest normalized separation.
4. Insert half of the entries into the first group and the other half into the second group.

3.2. R*-Tree

R*-Tree[3] is a variation of the aforementioned R-Tree that questions the heuristic used by the R-Tree by using a combination of the following heuristics:

- Maximization of the minimum bounding box area covered by node's directory rectangles. Specifically, we want to minimize the "dead space" of node's minimum bounding box.
- Minimization of the overlap value of the directory rectangles in a node. Intuitively this is a must-do heuristic because the less the rectangles overlap, the less of unnecessary child nodes the search procedure will called recursively given a search rectangle.
- Minimization of the margin(perimeter) value of a directory rectangle in order to be more quadratic the directory rectangles because given an area value, square has the minimum perimeter.
- Storage utilization, which is important because R-Tree deals with secondary memory, and will keep the query cost low because the height will be low.

But, it is not mandatory to use them all, for two reasons. Firstly, the use of one heuristic will affect positively the other without using it. For example, if we want to minimize the overlap then maybe the rectangles are distant one another, thus, the covered area of the minimum bounding box of a node can be maximized, so, it will be redundant to use explicitly the first heuristic. Secondly, the use of all heuristics simultaneously does not achieve the best retrieval performance, but a good combination of them does.

R*-Tree Operations

The insertion and deletion operation are the same as the R-Tree ones. Thus, because R*-Tree deals with different heuristics, the choose subtree and node splitting routine will be presented. Note that, these operations can be done with different combination of heuristics but the operations will be presented with the heuristics that gave the best retrieval performance as said in [3]. Before showing the choose subtree routing we should define the overlap value of a node's record. Given the records of a node $R_1, R_2, \ldots, R_p$ the overlap value of a record R_k is

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

Choose subtree routine does the following:

1. If current node's children are leaves, then the child node of the record that will have the minimum overlap enlargement is chosen. Ties are resolved by choosing the record that its bounding box needs least area enlargement.
2. Else, the next subtree is chosen like it is chosen in R-Tree.

Finding the overlap value of a record needs linear time. Therefore, the choose subtree algorithm needs quadratic time to choose the next subtree for the last before last level of the tree which will be too detrimental to the insertion performance, especially if the tree does many insertions. Fortunately, an optimization exists that makes the choose subtree routine run in subquadratic time because the algorithm does only a sort and retrieves the overlap value of a constant number of records. The optimization algorithm that is used is the following:

1. Sort the records in ascending order based on the area enlargement needed to contain

the new rectangle.
2. Choose from the first p records, the one that needs least overlap enlargement to contain the new entry. Resolve ties by choosing the record that needs least area enlargement.

In [3] it is mentioned that for two dimensions(which will be the dimensionality of the data that we will do experiments) the optimal value of p is 32.

The concept behind the Node Splitting algorithm of an R*-Tree is the following. Firstly, the records are sorted by their low and high value along each axis. Secondly, the goodness value, of each distribution of the sorted record lists, is determined, in order to find the perpendicular axis that the split will occur.

Finally, from the sorted lists that correspond to the split axis we calculate the goodness value-which can be different from the previous step- of each distribution in order to determine the two resulting groups. Note that, distribution of a list is defined as the result of splitting the list at some index. Therefore, given that a node should have more or equal than m and less or equal than M records and that node has M records, there are $M - 2m + 2$ distributions. The following goodness values can be used as defined in [3]:

1. Area Value: is defined as the sum of the areas of the minimum bounding boxes of the two groups.
2. Margin Value: is defined as the sum of the perimeters of the minimum bounding boxes of two groups.
3. Overlap Value: is defined as the area of the intersection between the two minimum bounding boxes of the two groups.

These values can be used to find the most suitable distribution by choosing the distribution that has the minimum chosen goodness value or by taking an evaluation method that combines some of the aforementioned goodness values. Specifically, the Node Splitting algorithm of an R*-Tree is the following:

1. The records are sorted by their low and high value along each axis.
2. For each sorted list, the sum of margin value of each distribution is calculated and we find the axis that corresponds to the sorted list with the minimum value. The axis perpendicular to the aforementioned axis, is the split axis.
3. The two resulting nodes are the two groups that are formed from the distribution along the split axis with the minimum overlap Value.

Node Splitting algorithm requires subquadratic time because in the first step are done 4 sortings(assuming that data are 2-dimensional). The second step does linear time because it just finds in constant time the margin value of each distribution and the total number of distributions is linear. Assuming, that we saved the distributions in a structure, the third step requires linear time because it just finds in constant time the overlap value of each distribution and the total number of distributions is linear.

An extra operation of an R*-Tree that elevates its performance is the Forced Reinsert operation. The reason behind this operation is that the organization of a tree is non-deterministic and depends on the sequence that the records will be inserted. However, a tree's organization might achieve a good retrieval performance, but, when the tree will be "shaken" by a split or a deletion, the performance might not be optimal. Thus, the tree suffers from its old entries and an reorganization mechanism should be determined. As said in [3], by reinserting half of the records achieved a better performance, but there is a need of a more sophisticated and dynamic method.

Firstly, Forced Reinsert method is invoked when an overfull node is encountered and it is the first one that was encountered on the current level of the tree, in different case the Node Splitting operation is invoked. The Forced Reinsert routine works as follows:

1. Sorting the records in decreasing order by their distance between the centers of record's bounding box and the minimum bounding box of the node.
2. The first p entries from the tree are removed and the minimum bounding boxes changes are propagated all the way up to the root.
3. The removed entries are reinserted in the tree by starting either from the maximum distance entry(far reinsert), either from the minimum distance entry(close reinsert). The reinsertion method is the same as the R-Tree Insert method with only difference that the record is not inserted in the bottom level, but on its appropriate one.

As said in [3], close reinsert outperforms far reinsert and the optimal value of p is the 30 percent of the maximum node records. Furthermore, this operation it is not guaranteed that it will make the insertion routine slower, because the organization of the tree is better, thus less splits will occur. Finally, overlap value is getting decreased because the records are moved to other nodes, the bounding rectangles become more quadratic, thus the margin value decreases and storage utilization is improved.

3.3. Voronoi K-Means Initialization

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.

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.
- 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.

The complexity of the algorithm is the following:

1. O(nlogn) for the creation of Voronoi Diagram.
2. 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 $O(n^2)$
4. But this step on average does O(nlogn) time.

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

4. Inventing new algorithms

4.1. Voronoi Diagram Clustering

The point of the algorithm is to create the desired clusters with the help of the Voronoi Diagram. Using the voronoi vertices we construct the initial clusters. Based on the radius threshold value of the voronoi circles we create the desired clusters with the voronoi vertices as their center.

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

V_i : i_{th} voronoi vertex

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

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 i_{th} radius

Algorithm In brief

1. 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.
2. 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 i_{th} circle r ≤ m (radius threshold) then we find the points that are in the boundary of the i_{th} 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.
    - Else If the radius of the i_{th} 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.
3. 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 ≤ m.
4. 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.

Pseudocode

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(V_i)) and store these vertices in V[]

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 V_i (CirS(V_i)) to r (r = R(V[i]))

Step 5:

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

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

$\{C_1, C_2, \cdots, C_k\}$.

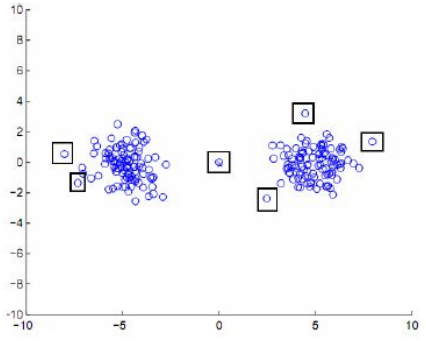
Step 9:

For i =1 to k

If C_i < *M* then C_i is an outlier

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

Outlier Example



Complexity Analysis

Step 1:

O(nlogn) for the construction of Voronoi Diagram from *n* points

Step 2:

O(nlogn) for sorting

Step 3:

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

Step 4, 5, 6:

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

Step 2-6:

O(nlogn) + O(2n-5) = O(nlogn) + O(n) = O(nlogn)

Step 2-7:

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

Overall complexity:

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

5. Experimental Evaluation

5.1 R/R* Tree

We wrote R*-Tree and R-Tree with Linear and Quadratic Split in the C++ Programming Language. In the following experiments we inserted sequentially 500 thousand points and did 10 thousand Range Queries. The values of m that are used for each type of R-Tree were the optimal that were determined from the experiments conducted in [2] and [3] and the 50 percent of M just to test values other than the optimal.

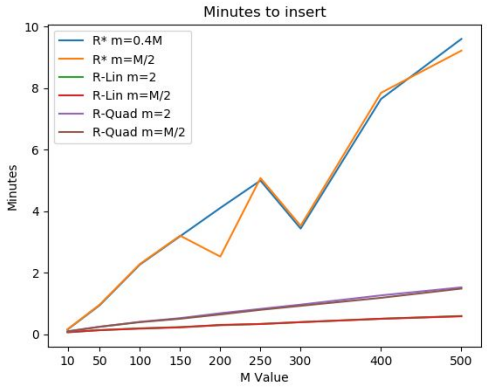


Figure 3: Total Insertion times

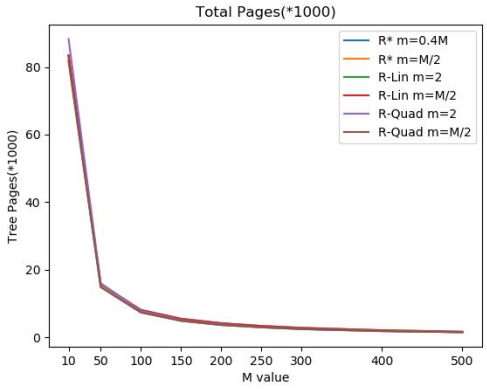


Figure 4: Total Pages

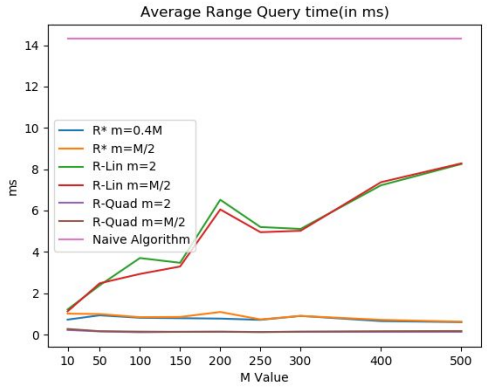


Figure 5: Average Query Time(ms)

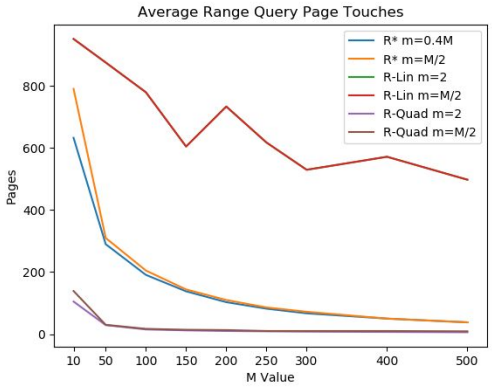


Figure 6: Average Page Touches in Query

As we can observe in Figure 3, the faster structure for insertions is the R-Tree that performs Linear split, and this might be obvious because the Linear Split algorithm is the fastest node splitting approach because it runs in linear time. Furthermore, R*-Tree is much slower, probably because more splits may happen and R*-Tree's choose subtree routine is slower than R-Tree's one, even with the optimization. Also, note that, that R*-Tree does not perform the Forced Reinsert method, thus it might change the results. In Figure 4, we can see that all variations have the same number of pages approximately, thus we can tell that in terms of space utilization are almost identical.

From Figure 5, we can observe, that every R-Tree variation has better performance than the Naive method to answer range queries, thus we can say that the complexity of the range query reaches logarithmic time, thus DBSCAN runs in subquadratic time. Furthermore, we observe that the value of m does not affect the performance much. Finally, we observe that R-Tree with Quadratic Split and R*-Tree are keeping their times low, contrary to R-Tree with Linear Split. In Figure 6, it seems that the value of m does not affect the number of pages touched at all. Moreover, we can see that R*-Tree needs a larger node capacity in order to reach the performance of Quadratic R-Tree while at small node capacities it has a very bad performance. Concluding, the best R-Tree variation, paradoxically, is the R-Tree with Quadratic Split because it is faster than R*-Tree in both insertions and range

queries. Finally, it is shown that every R-Tree variation performs better than the naive approach in answering range queries, thus the DBSCAN's algorithm performance is improved, and since R-Tree's average search complexity is $O(\log(n))$ the DBSCAN's time complexity is $O(n\log(n))$.

5.2 K-Means Initialization with Voronoi Diagram

Initially, 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. Afterwards, we compared the performance of our algorithm with K-Means++, which sklearn library thinks that its best initialization method.

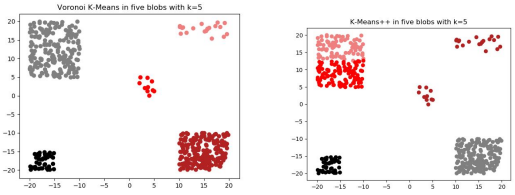


Figure 7: Left: Voronoi K-Means Initialization Right: K-Means++ on five blobs with k=5

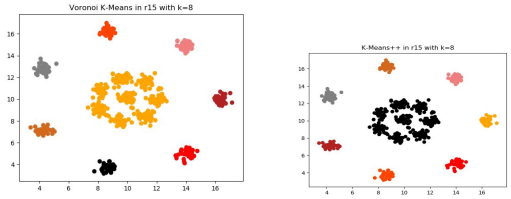


Figure 8: Left: Voronoi K-Means Initialization Right: K-Means++ on r15 with k=8

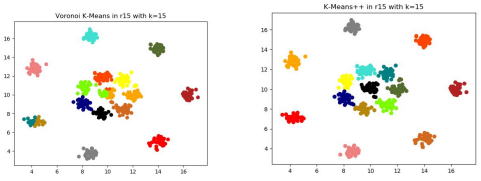


Figure 9: Left: Voronoi K-Means Initialization Right: K-Means++ on r15 with k=15

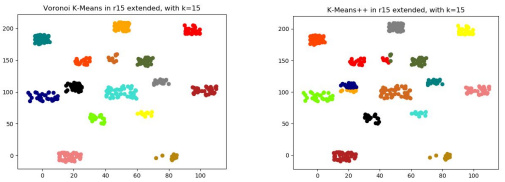


Figure 10: Left: Voronoi K-Means Initialization Right: K-Means++ on r15 extended with k=15

As we can observe from the figures 7-10, the proposed algorithm at most times gives better results than K-Means++ provided that the parameter *k* has the proper value. Furthermore, 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. Finally, from the results of R15 and R15 Extended(figures 9 and 10), we can observe 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 optimal, while K-Means++ performed poorly.

5.3 Clustering Algorithm with Voronoi Diagram

Firstly, we wrote the algorithm in Python. The experiments are on 4 datasets. The banana data, the density separated data, outlier data and half circle data. The algorithm can be applied to more common found datasets like Iris.

Figure 8 Clustering results on *banana data* of 550 points, (a) proposed algorithm (b) *K*-means clustering (c) FCM clustering (d) CTVN (see online version for colours)

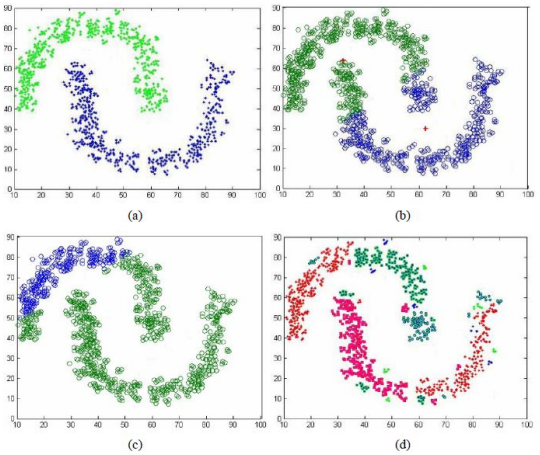


Figure 6 Clustering results on *half-circle data* of 1,250 points, (a) proposed algorithm (b) *K*-means clustering (c) FCM clustering (d) CTVN (see online version for colours)

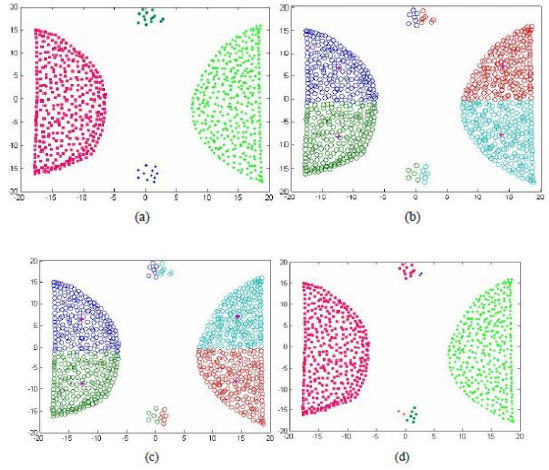


Figure 7 Clustering results on *density separated data* of 1,009 points, (a) proposed algorithm (b) *K*-means clustering (c) FCM clustering (d) CTVN (see online version for colours)

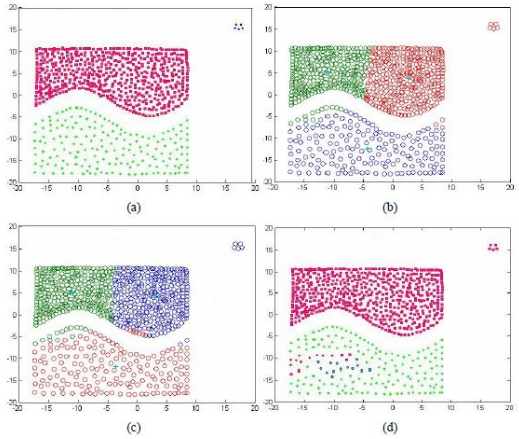
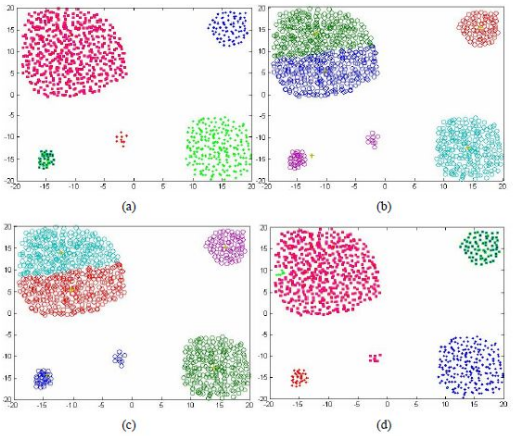


Figure 5 Clustering results on *outlier data* of 1,500 points, (a) proposed algorithm (b) *K*-means clustering (c) FCM clustering (d) CTVN (see online version for colours)



As we can see from the figures which we referenced [6], the proposed algorithm at most times gives better results than K-means, FCM and CTVN given the right parameters.

6. Conclusion

In this paper we illustrated two ways on how can Computational Geometry field can contribute to the Clustering Problem. Firstly, we presented how we can optimize existing algorithms not only in terms of time, but also in terms of results, by showing a spatial indexing type, called R-Tree that can be used in order to answer range queries more efficiently, and an algorithm that used the Voronoi Diagram in order to initialize better the initial centers of the K-Means algorithm, in order to produce better clusters. Secondly, we presented a new clustering algorithm that uses the Voronoi Diagram in order to produce the resulting clusters by merging nearby points based on the radius of the Voronoi vertices. Finally, the experimental results were encouraging because they shown that these contributions can be used in real world in order to optimize the existing algorithms and to address better algorithms to the clustering problem.

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.2014.05.