Data Dependent Dissimilarity Computation in Dynamic Datasets

A B. Tech Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Dharmesh Chourasiya , Kushal K S V S
(150101022,150101031)

under the guidance of

Dr. Amit Awekar

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Data Dependent Dissimilarity Computation in Dynamic Datasets**" is a bonafide work of **Dharmesh Chourasiya , Kushal K S V S** (**Roll No. 150101022,150101031**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Dr. Amit Awekar**

Assistant Professor,

November, 2019 Department of Computer Science &

Engineering,

Guwahati. Indian Institute of Technology Guwahati, Assam.

Contents

Abstract

With the rapid increase of data produced, we are using data-driven software applications more and more everyday. Essential algorithms like clustering, anomaly detection and classification are used on the data. It has been found that data-dependent measures work better in these cases than simple geometric euclidean distance to measure dissimilarity of the data point pairs. Our work includes improving the incremental algorithm proposed and devising a decremental algorithm which reduce the the space and time complexities of the dissimilarity calculation when data points are added or subtracted.

Chapter 1

Introduction

1.1 Importance of Data Dependent Disimilarity

Data Analytics mainly focuses on discovering patterns in data sets to get better insights from the data and the target entity by employing various qualitative and quantitative methods. Dissimilarity measure between two data points plays a vital role in this process. There are many methods proposed to get the dissimilarity measure between the points. Using geometric distances like euclidean distance between the two points is one such way to do density estimation, clustering, anomaly detection and classification. Irrespective of its widespread applications, geometric distance does not posses key property for dissimilarity i.e. two point in a sparse region tend to have more similarity than two points with same geometric distance in a dense region. It is observed that data dependent dissimilarity is a better measure because they tend to take the context of the data i.e. the data distribution into consideration.

1.2 Organization of The Report

In chapter 2, we discuss the MBSCAN algorithm introduced my Carman[1]. The data structures used and the algorithm are explained in detail. The usage of random forests and the definition of the dissimilarity measure is given.

In chapter 3, we propose the decremental algorithm. The algorithm aims to efficiently calculate the dissimilarity measures when data points are removed. We discuss the algorithm and the optimizations done. We will also take a look at the results which give an insight about the performance of the algorithm.

Chapter 2

Related Work

One of the first algorithms proposed in data dependent dissimilarity measures which took care of the context of the data and other data points in region was DBSCAN. This solution was particularly focused on clustering. This algorithm was slow and inefficient because of the complexity. A more robust and efficient algorithm for computing mass based dissimilarity is MBSCAN. Now we'll discuss the model used to compute the mass based dissimilarity between any pair of point in a Data set(D ,n) having size n.

2.1 MBSCAN Algorithm

iForests(isolation forests) are used to implement mass based dissimilarity. Isolation forests are formed using decision trees. Partitions are created by first selecting a random attribute/feature and then randomly selecting a split value for that attribute.

The dissimilarity measure used here is called mass value. Mass value of a pair of points is a number and it indicates the dissimilarity value. It is a probabilistic measure and is calculated as an average over many isolation trees. massMatrix is a matrix of mass values of all such pair of points in the space. The massMatrix can easily used by clustering, anomaly detection and classification algorithms by replacing the geometric distance.

The parameters that are required to create the iForests are:

- t - defines the number of *iTrees*.
- s - this is the sample size that is used to build each itree.

Every itree is a binary tree with a *maxHeight* of *log(s)*. We randomly select s number of points from the data set for each *iTree* and insert them at the root node. At each node containing some data point we randomly choose one *split attribute* from the various attributes that describe the data points. We select a *split point* which is a random value in range of minimum to maximum *split attribute* value that the attribute can take among those data points. We split the data points in the node into its left child right child based on these *split attribute* and *split point*. We stop splitting the nodes when it's a leaf node. A leaf node satisfies either of the two conditions. First, if the node is at height *maxHeight*. Second, if the node contains only single data point. We construct t number of *iTrees* using this method and each itree will have a random *split attribute* and *split point* value for each of it's nodes.

Now we insert all the data in data set D in all the *iTrees*. The data points use the previously computed *split attribute* and *split point* values to get partitioned into the left or the right child for each node. The partitioning stops when we reach leaf node of the *iTree*.

So after this step, every data point lies in exactly one leaf node in an *iTree*. The mass dissimilarity between two points is defined as the number of points in the minimum region containing both the data points. Here, we can clearly observe that minimum region with these two points together is their Least Common Ancestor of both of these leaf nodes. For each point pair x and y we find their Least Common Ancestor in an *iTree* T,let it be L_T. We define the mass of L_T as the number of data points in the node.

Then the dissimilarity measure(m_e(x,y)) between these two points can be defined as the average mass of L_T over all *iTrees*.

$$m_e(x, y) = \frac{\sum_{T=1}^{numOfTrees}(L_T)}{numOfTrees}$$

. This paper also defined μ-Neighbourhood Mass of a point x as the number of points y with $m_e(x,y) < \mu$ in the data set. This is also used in density based clustering algorithm.

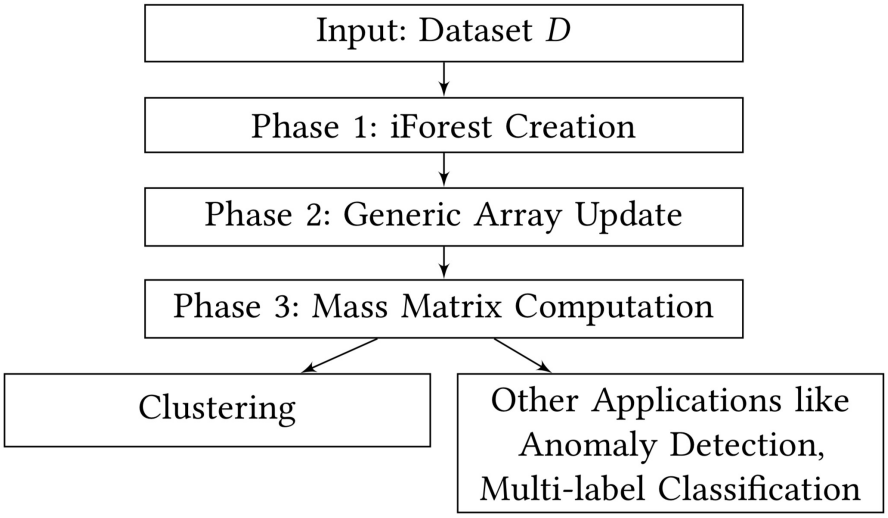

Fig 1: Steps in calculating the Mass Values

The author uses this mass value in place of geometrical distance in standard algorithms of clustering, multi-label classification and anomaly detection. The method to discover cluster using mass based dissimilarity measure MBSCAN is same as distance based DBSCAN with distance matrix replaced by mass matrix. The parameter μ in MBSCAN play the same role as ϵ in DBSCAN.

Chapter 3

Decremental Algorithm

Our work is focused on the decremental algorithm. Under dynamic environment, the geometrical distance between two points remains same but the dissimilarity value between the points might change due to the change in the density of the region containing the points. We focus on the aspect when a small proportion of the data points are removed from the space.

Let us say that we have data set D of size n and we have to remove some data points D' of size n'. We assume that n' is very small as compared to n because we don't want the structure of the *iTrees* to be invalidated because of the changes.

3.1 Types Of Data Points to be Deleted and their Deletion Mechanism

Removal of points will lead to one of the two scenarios.

- Points will change the structure of the iTree on removal.
- Points will not change the structure of the iTree on removal.

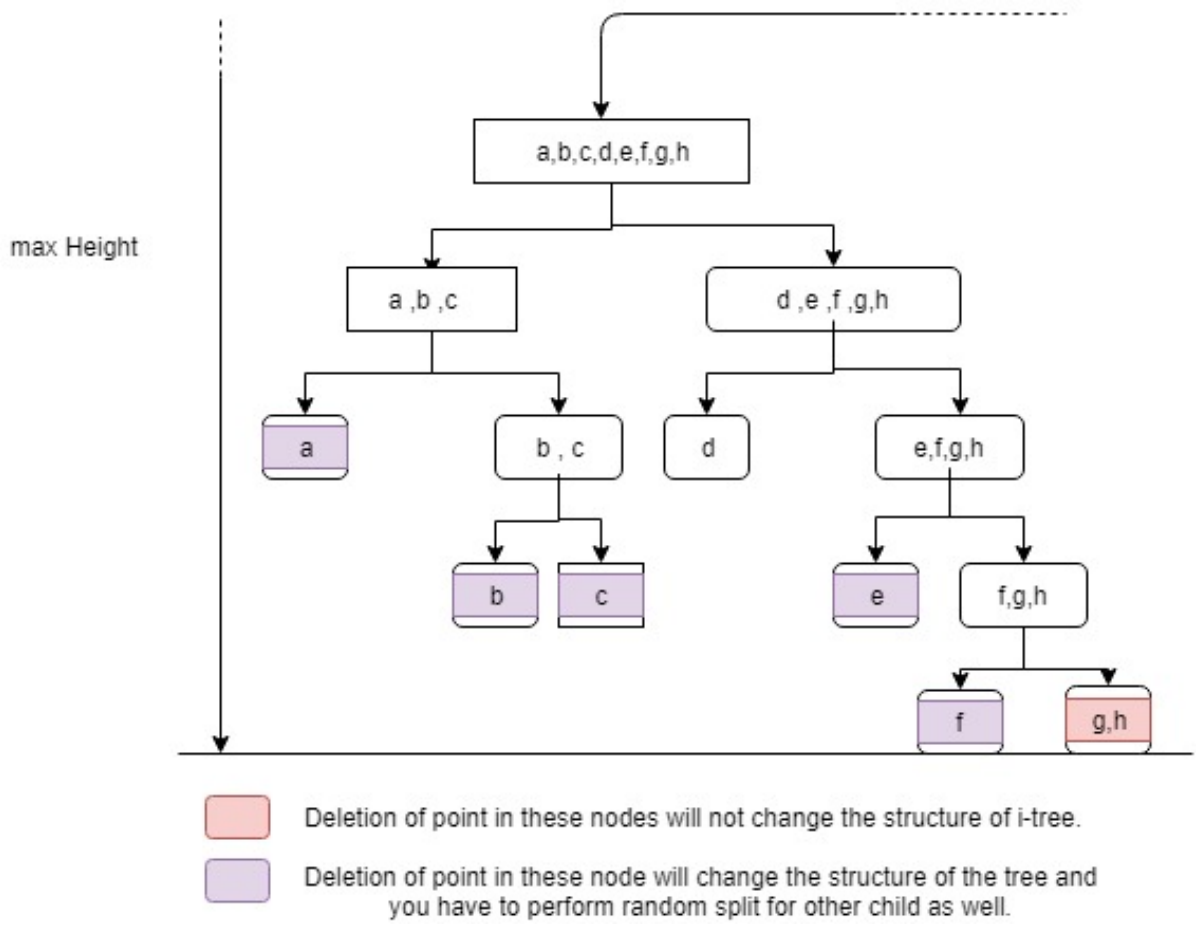

Fig 2: Different type of nodes to be deleted.

Figure 2 above shows a sub-tree of an *iTree*. Nodes are represented by rectangular boxes and alphabets represent the data points present in those nodes.

If we want to remove a data point x from the iTree, there can be two cases. First, all the points in the leaf node of data point x will be removed. Second, there is at least one point in the leaf node which will not be removed. In the first case, the node will be invalidated and the tree should be modified. We will use the term invalidated to point out that the node has no valid data points upon removal.

In *Fig 2*, removing one point from the leaf nodes which are coloured red will not change the structure of the tree. But removing a point from one if the blue nodes will change the structure. We can generalize this over removal of many points.

This algorithm handles both the cases and calculates efficiently the updated mass values.

3.2 Algorithm Flow

Arrays are used to represent nodes in the iTrees. We have the following iForest data after the static run.(Before the removal of points)

- **SplitAttribute** array - *SplitAttribute[i][j]* will point to the *split attribute* of the node j of the tree i.
- **SplitPoint** array - *SplitPoint[i][j]* will point to the *split point* value of the node j of the tree i.
- **TreeNode** array - *TreeNode[x][i]* will point to the leaf node of the data point x in the tree i.
- **TreeNodeMass** array - *TreeNodeMass[i][j]* will give the number of nodes in the node j of the tree i.

The steps to efficiently calculate the massMatrix after removal of the points are as follows. Note that all the tree traversals are done in level-order i.e. first the root is visited and then its children and so on.

3.2.1 iForest Update

We first remove the data points *D'* from *D*. This will give us the set of data points *D - D'*. This is done by storing the points to be removed in a set *S*. For all the points in *D*, put them in *D - D'* only if the point is not in the set *S*. Note that two points having the same set of attribute values cannot exist in the calculation of dissimilarity values.

We select this data set and we insert them into each iTree. We partition these points using the same *split attribute* and *split point* values as in the previous static run. Every point in this run will go through the same nodes as in the static run as the structure of the *iForest* has not changed. As we traverse down the tree, we will have one of the following scenarios.

1. Some of the points from the set *D - D'* reach the node.

    - Number of points reaching the node is equal to the *TreeNodeMass* of the node.
    - Number of points reaching the node is less than the *TreeNodeMass* of the node.

2. None of the points from the set *D - D'* reach the node.

We first look into the second case where the node is empty. This is an invalidated node meaning this has to be removed from the *iTree*. We can also see that all it's children will be invalidated too. So, the whole subtree is invalid. For every node like this, we make *splitAttribute* to -1 denoting that the node is not a part of the *iTree*. We don't need to update mass values for any of these pairs because all the points are to be removed.

Once we see such a node, we take it's sibling and replace the parent of the node with it. For example in *Fig 2*,

- if the node containing *[g,h]* is invalidated i.e. g and h are to be removed from the tree, then we replace the node *[f,g,h]* with *[f]*.
- In other case, if the node *[f]* is to be removed, we replace the node *[f,g,h]* with *[g,h]*. Now as the node *[g,h]* does not satisfy the leaf node conditions, we split the node by choosing a random *split attribute* and *split value*.

3.2.2 Mass Matrix Update

From the previous step, we have the nodes which are valid i.e. at least one point is in them from the data set *D - D'*. For all such nodes, we can simply state that the number of points in that node now is the new *TreeNodeMass* value of the node.

We traverse the modified *iTree* again. We skip the invalidated nodes by checking if the *split attribute* is -1. For all other nodes, we replace the *TreeNodeMass* value. We observe that the current node is the least common ancestor of all the pair wise values of it's child

nodes. We update the mass values of all the pair wise values. We do this by subtracting the old *TreeNodeMass* contribution and adding in the new *TreeNodeMass* contribution.

Finally, we remove all the pairs from the *massMatrix* which have the removed points. This is done easily by deleting the rows and columns of the removed nodes.

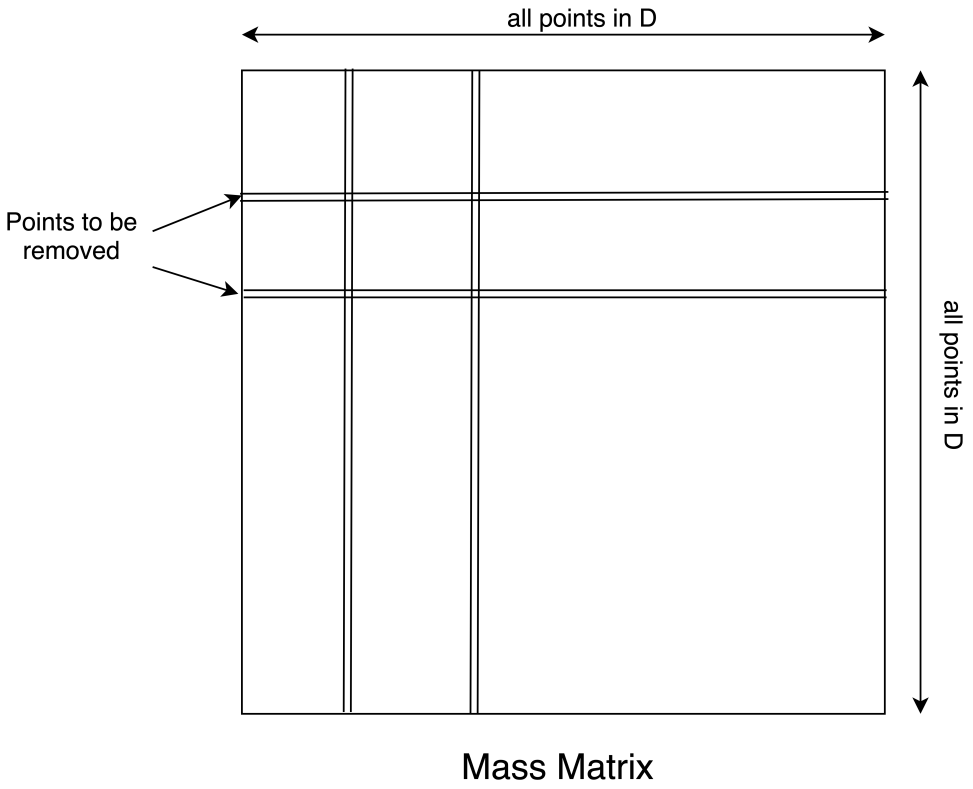

Fig 3: Mass Matrix

This algorithm is can be done with inserting *D + D'* instead of calculating *D - D'*. In this case, for every node where the points reach we update the *TreeNodeMass* values as follows. Let's take the old *TreeNodeMass* values as x and the new *TreeNodeMass* value as $x+y$. The required value is x-y. This means we have to do *TreeNodeMass* -= ($x+y$ - x) at every valid node. We found out that this is a bigger overhead on the performance than calculating *D - D'* in the first step when the *iTree* is big.

An optimization to the code allowed us to do both the steps in a single traversal decreasing the run-time. We also added a check to make sure the *split value* leads to division of points at a node.

3.3 Results

We used the data sets listed below to measure the performance. The metrics for the measure of performance are as follows. As the mass value is a probabilistic approximation, the values are averaged over ten runs.

Dataset	Size	Dimensions	Clusters
Aggregation	788	2	7
D31	3100	2	31
Iris	150	4	3
Libras	360	90	15
Segment	2310	19	7
S1	1000	2	4
S2	10000	2	4

Fig 4: The data sets used

- **Accuracy** - The percentage of mass values within the margin of error of the correct mass values because across multiple runs the values will be different.
- **Speed** - Measure of time taken to calculate the mass value. Here speed means *(static(D) + static(D'))* vs *(static(D) + dec(D'))*

The following are some useful terms used in the measurement of performance

- **Error** - Error percentage gives us the margin of error between two readings. This is used to calculate accuracy.
- **Dec percentage** - The percentage of points to be deleted from the data set. This will affect the structure and size of the tree.

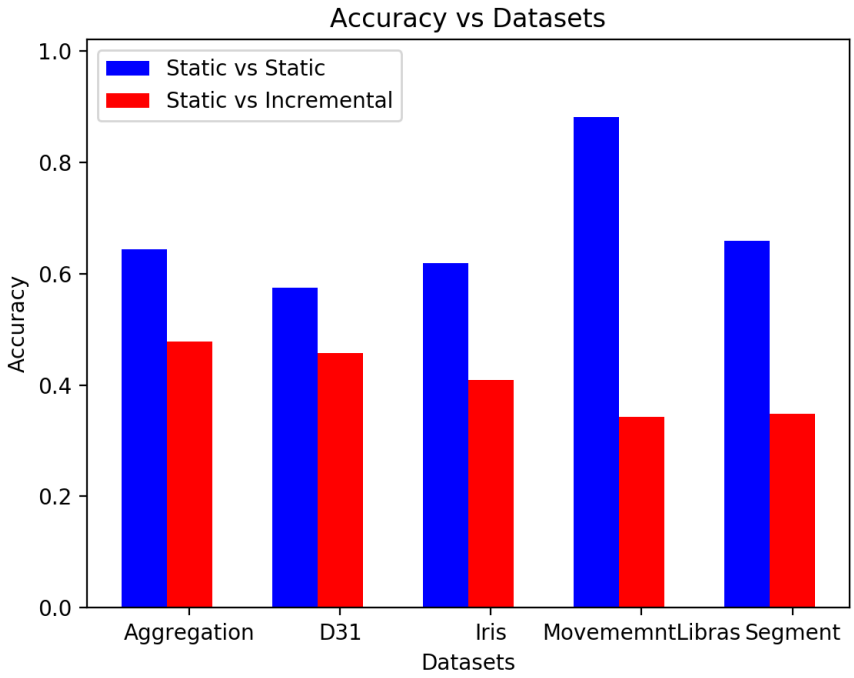

Fig 5: Accuracy with error at 5 percent

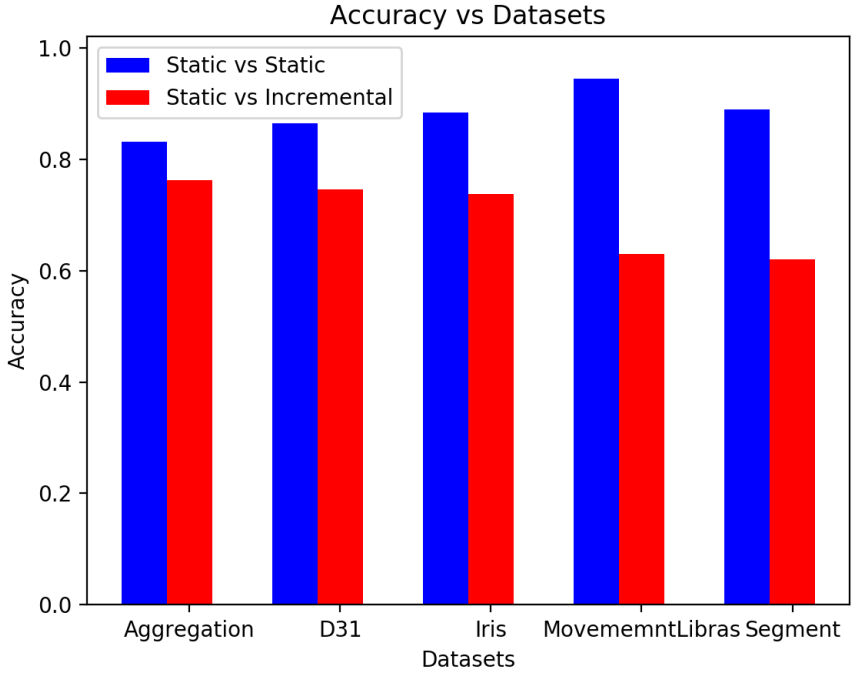

Fig 6: Accuracy with error at 10 percent

3.3.1 Observations

- Increase in the error percentage increases the accuracy making more runs come under the margin.
- Increase in the ψ value increases the depth of the tree which inturn increases the time complexity.
- With increase in the decremental percentage, the algorithm performs worse. This is because the more points deleted, the less valid will the structure of the tree will be.
- The number of attributes contributes less to the time complexity and the number of points directly is proportional to the time complexity.
- More the number of points to be deleted, less is the speedup advantage. Also leads to less accuracy.

Chapter 4

Conclusion and Future Work

In this paper, we propose a decremental algorithm for MBSCAN. Also, an optimization on the current incremental algorithm is implemented. We can see that the optimization is removing the redundant calculations being done in the mass matrix update step. We did this by using the previous iteration's *treeNode* and *treeNodeMass* values.

We implemented the version of decremental algorithm stated above with the optimization of running it in one traversal. The majority of the time goes in calculating the *massMatrix* values which the optimization helped reduce. We can see that the *massMatrix* least common ancestor of the pairs will not change even if the forest structure is updated. This leads to faster calculation as the *TreeNode* values are not changed.

References

[1] Kai Ming Ting, Ye Zhu, Mark Carman, *LaTeX: Overcoming Key Weaknesses of Distance-based Neighbourhood Methods using a Data Dependent Dissimilarity Measure*. Federation University Victoria 3842, Australia

[2] Martin Ester, Hans-Peter Kriegel, Jg Sander, Michael Wimmer, Xiaowei Xu *LaTeX: Incremental Clustering for Mining in a Data Warehousing Environment*. Institute for Computer Science,University of Munich Oettingenstr.67, D-80538 Mtinchen, Germany