Master of Science Thesis

Machine Learning Approximation Techniques Using Dual Trees

Denis Ergashbaev

Supervisor:
Oriol Pujol Vila
Dept. Matemtica Aplicada i Anlisi, University of Barcelona (UB)

April 2015

ABSTRACT

This master thesis explores a dual-tree framework with underlying kd-tree space partitioning data structure as applied to a particular class of machine learning problems that are collectively referred to as generalized n-body problems. We propose a novel algorithm based on the dual-tree framework to accelerate the task of discovering characterizing boundary points (CBP) – a set of data points defined by geometry rules and representing an optimal robust interclass boundary. Designed with support for both approximate and exact computations, experimental results confirm superior runtime properties of the algorithm compared to a state-of-the-art solution. Furthermore, we propose an improvement of the Boosted Geometry-Based Ensembles algorithm that constructs a CBP-based strong classifier. Owning to our modification of the original learner, the scalability of the algorithm is improved to be able to operate on the datasets of higher size and dimensions while improving speed and maintaining high accuracy of classification.

ACKNOWLEDGEMENTS

This work would not have been possible without the firm support of my final thesis supervisor Oriol Pujol. I am deeply indebted to him for his encouragement, guidance, and mentorship.

I am thankful to my family for their support of my delayed endeavor of pursuing a master's degree. They have both pushed me forward and held me from behind when I was about to give up.

Being so remote in distance, my American family has stayed close to me over the years. I miss you so much!

My cool boss Thomas for letting me go and keeping me at the same time. It helped a lot. To the PD department – Oliver, Gordon, Stefan, Steffen, and Martin – for fixing my bugs and covering up for me. Thanks.

"Katta rahmat" to my friend Sarvar for endless discussions about academic life and computer science, our shared adventures in Barcelona, and his critical role in making me commit to this master's program.

I am grateful and obliged to Dilshod Ibragimov who always supports my various aspirations, including this master's program, and helps me to keep on growing. Thank you.

Thanks goes to my long-time flatmates – Lorenzo and Hadi – for friendship, many fun parties, adventures, and incessant tea talks. This is not to forget El Arbi and Pablo, my first ones, living with you was so much enjoyable and helped me to get on terms with living in the big city again.

Pablo, Jeroni, Lorenzo, and Iosu, our sleepless all-nighters, crunching the brutal assignments and coffees in the morning helped me survive the first semester. Being able to rely on you in a foreign country made my stay so much more enjoyable. Pablo, for introducing me to the Spanish legal system.

Dear members of the machine learning beers community – Anna, Alex, Eloi, Carles and Oriol – for beers and geeky talks. Carles and Alex, for setting up Elbereth and letting me run my experiments there. "Cheers!"

The classes of Javier Larrosa and Albert Oliveras were amazingly good, useful, and involving. I learned so much.

My Russian/Ukranian friends – Oleg, Pavel, Grigoriy, Aleksandr, Igor – made my stay in Spain to feel more like at home, while the great Spanish class team – Xavier, Maria, Vladimir, Svetlana, Arancha & Ainhoa made learning Spanish so fun. Simran, for her birthday cake and good times.

I know I have missed many more people without whom the whole master's would not be possible or at least much less enjoyable: "muchas gracias".

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

CBP Characterizing Boundary Points

GE Geometry-Based Ensembles

Boosted OGE Boosted Geometry-Based Ensembles

Kd-trees K-dimensional Trees

1

INTRODUCTION

1.1 MOTIVATION

Many of the machine learning methods – including all-nearest-neighbors problem, range search, kernel density estimation, and two-point correlation – are naively quadratic in the number of data points [11]. This time complexity compromises their use in the large-scale machine learning applications thus demanding more efficient solutions to accelerate the naive approaches.

One commonly used method to reduce the time complexity of solving these problems is application of space-partitioning data structures, such as kd-trees, and use of branch-and-bound algorithms to improve the runtime speed [7]. This idea has been further developed for a subset of the problems falling in the class of *generalized n-body problems* by applying a dual-tree framework.

Initially introduced by Alexander Grey [12], the dual-tree framework has been claimed and theoretically proven to achieve a near-linear performance for a range of n-body problems. Nevertheless, beyond a narrow circle of researchers, the dual-tree approach has not yet been adopted in broader scientific community. With relative scarce coverage which is limited by a few papers and presentations, we believe that dual trees deserve a more exploration.

1.2 OBJECTIVES

This thesis sets forward several objectives:

1. To implement a dual-tree framework based on the general algorithm provided in [7, 12] and to apply it to one of the machine learning problems that would benefit from complexity reduction.

2. To explore the behavior of the proposed algorithm on the datasets of different size and dimensionality comparing the performance with a state-of-the-art solution.

3. To provide an approximate solution based on dual trees as an acceptable trade-off between computational complexity of the algorithm and accuracy of the results.

1.3 CONTRIBUTIONS OF THE RESEARCH

This work makes several contributions:

1. It provides a workable reference implementation of the dual-tree framework based on the kd-tree data structure and written in python programming language.
2. The reference implementation of the dual tree framework is applied to one of the machine learning problems: finding characterizing boundary points [24, 23]. Possible dual-tree architectures as well as relevant pruning and local search strategies are compared.
3. The performance of the devised framework is verified against datasets of different size and dimensionality, reporting the gains in speed, worst and real cases of required computations.
4. Along with the exact solution an approximate technique to find characterizing boundary points is developed and applied to the Boosted OGE problem [23] in order to achieve computational tractability for datasets of bigger size and higher dimensions.

1.4 ORGANIZATION

This report takes off with the review of kd-tree (Section 2.1), a space-partitioning data structure, and moves on to introduce a dual-tree framework (Section 2.3) that integrates kd-trees into a new higher-order divide-and-conquer algorithm to solve generalized n-body problems. A description of the Geometry-Based Ensembles, a new classifier based on the *characterizing boundary points* (CBP), ensues in Section 2.4.

Following review of the state-of-the-art in Section 3.1 we formulate a proposal for a novel algorithm to construct CBPs that applies the dual-tree framework with custom developed pruning strategies and a local search in order to gain speed improvements over the state-of-the art technique. Subsequently, in Section 3.2, we propose two alternative models to the Boosted OGE classifier (that uses CBPs as its building blocks) that significantly improve its time performance on larger datasets.

Peculiarities of the new algorithm for CBP computation and proposed Boosted OGE models are analyzed in Section 4.1, while the

summary of the experimental results is provided in Section 4.2.

Lastly, in Chapter 5 we conclude the report with the short summary of the scope of work conducted and the new areas of research that this master thesis has opened.

2

BACKGROUND AND STATE OF THE ART

The following sections will present a common space partitioning data structure called kd-tree enabling execution of fast data queries. It is then followed by discussion of the dual-tree approach that builds on top of kd-trees to provide an efficient algorithm for a range of n-body problems. The section concludes with an overview of the promising geometry-based ensemble technique used for classification tasks.

2.1 KD-TREES

Kd-tree (short for k-dimensional tree) is a space-partitioning data structure, which can be viewed as special case of the binary space partitioning tree. It was invented by J.L. Bentley [1] to provide support for efficient range and nearest neighbor searches in multidimensional spaces.

Generalizing the binary search tree from one to higher dimensions, the kd-trees recursively partition space into half-spaces at each level of the tree. Each node in a kd-tree represents a hyperrectangle whose faces are aligned with the axes of coordinates.

Algorithm 1 lists the main steps involved in the construction of the kd-tree. Given a set S of data points in space $\mathbb{R}^d$ the algorithm proceeds to recursively bisect the space into adjacent cells. Partitioning of a relevant branch stops once the cell defined by the bounding hyperrectangle contains at most a given number of data points.

Figure 1 is a visual representation of the resulting space partitioning when applied to a three-dimensional data set. The root cell is bisected with a red hyperplane into two halfspaces, which are then recursively split by other hyperplanes (marked with green and blue colors).

There are two important considerations that need to be taken into account when constructing a kd-tree:

Leaf size. The algorithm stops dividing the space as soon as the current node has at most the amount of data points specified by this

Algorithm 1: *Kd − tree*. Kd-tree construction

Input: Set of $data\_points = \{x_i\},\ x_i \in \mathbb{R}^d$, depth
Output: kd-tree
begin
  $axis := select\_axis()$
  $splitting\_point :=$ by $axis$ from $data\_points$
  `// create node and subnodes`
  $node := new\_node()$
  $node.split = splitting\_point$
  $new\_depth := depth + 1$
  $node.left\_child := kdtree(points \in data\_points \leq$
  $splitting\_point, new\_depth)$
  $node.right\_child := kdtree(points \in data\_points >$
  $splitting\_point, new\_depth)$
end

parameter. While a too low value for the leaf (or bucket) size would increase overhead of traversing the nodes [22], letting this value approach the size of the dataset makes queries become a brute force computation. As it will be demonstrated later, in the context of the dual-tree framework, the leaf size will have a direct impact on the ability to perform efficient pruning.

Splitting rules. The exact structure of the tree and associated spacial subdivision depends on the procedure called *splitting rule* which selects a splitting hyperplane at each recursive step [17].

Standard split. The original paper on kd-trees has proposed to determine the axis to split on by the formula $D = L \ mod \ k + 1$; D is the axis to be chosen for the node at level L for the dataset dimensionality k [1, 9], whereas the partition point at the established dimension is chosen to be a random value. If, for instance, we have a two dimensional dataset, then the algorithm will split on the x dimension at the first level, on y at the second level, and then on x again. The splitting stops when the value of points held by each leaf is reached.

A further extension of the standard splitting rule is to split on the median value of the selected axis [2] in order to achieve a balanced kd-tree where each node is equally distanced from the root.

Midpoint split. The splitting hyperplane bisects the longest side of the cell. In contrast to the standard splitting rule, the spread measure has no importance in that case.

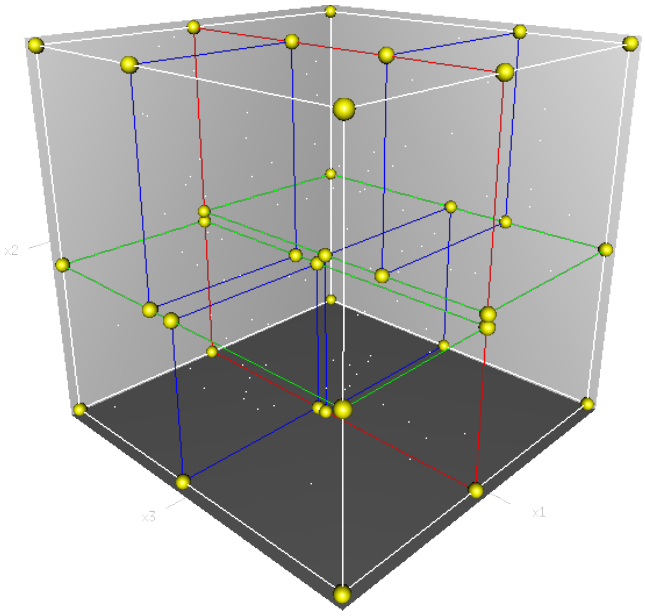

Figure 1.: A 3-dimensional kd-tree [3]

Sliding-midpoint. This rule is a combination of the previous two strategies, where first a provisional split is made along the longest side of the cell. In case that all the points appear only on one side of the hyperplane, the splitting hyperplane is moved along the axis to capture some of them. The result of application of this rule is that the cells satisfy a packing constraint, that bounds the number of disjoint cells of a given size that can overlap a ball of certain radius [17].

Figure 2 compares the effect of using the three splitting rules in the kd-trees. In this work, the kd-trees were built using the sliding-midpoint rule for the reasons that will become obvious later on.

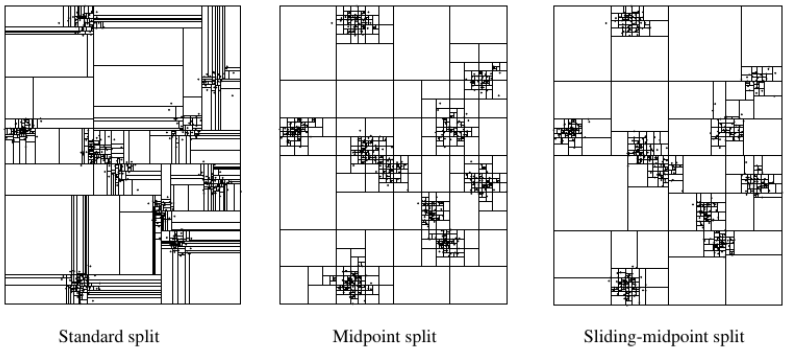

Figure 2.: Splitting rules [17]

2.2 NEAREST-NEIGHBOR SEARCH

As has been mentioned, one of the primary applications of the kd-trees is nearest-neighbor search. Given a query dataset X_Q containing N_Q points and a point x_r from a reference dataset X_R of size N_R, the nearest neighbor algorithm is defined as follows (a most commonly used euclidean distance is assumed):

$$Compute\ NN\ (x_q) = \arg\min ||x_q - x_r|| \tag{1}$$

The most straightforward solution for the problem – a linear search has a running time of $O(dN)$, where d is a dimensionality of the metric space.

Meantime, it has been shown that one of the two primary applications of the kd-trees – finding a nearest point – has $O(log\ N)$ complexity given random distributed points [9].

Algorithm 2 provides a highly abstract perspective of nearest neighbor queries with kd-trees. It is implemented as a 2-way recursion where first the root node of the kd-tree is queried for the point x_q. To find the exact nearest neighbor it proceeds to traverse through its child nodes provided that the distance of the nearest neighbor found so far to the query point x_q is larger then the distance of the current node to the query point.

Algorithm 2: *NN*. Nearest-neighbor search

Input: query point x_q,
node root node on first entry,
d initial distance set to ∞ on entry,
x_{best} best candidate found so far,
d_{best} corresponding best distance
Output: x_{best}, d_{best}
begin
  if $d > d_{best}$ **then**
    `// disregard (prune) the node`
    return
  end
  if $is\_leaf\_node(node)$ **then**
    $[x_{best}, d_{best}] = NNBase(x_q, node, d_{best})$
  else
    $[node_{closer}, d_{closer}, node_{further}, d_{further}] =$
    $order\_by\_dist(x_q, node.left\_child, node.right\_child)$
    $NN(x_q, node_{closer}, d_{closer}, x_{best}, d_{best})$
    $NN(x_q, node_{further}, d_{further}, x_{best}, d_{best})$
  end
end

2.3 DUAL TREES

There is a class of problems in machine learning, such as all-nearest-neighbors, kernel density estimation, n-point correlation that require comparison of each pair of points individually in order to provide a solution. This class of problems was described by Alexander G. Gray in his Ph.D. dissertation and was given a name *generalized N-body problems* [12].

The particularity of these problems is that it is generally not possible to determine the relationships between individual pairs of points analytically without considering each pair of the points individually [12]. Thus, the straightforward solution dictates an $O(N^2)$ time complexity.

However, several approaches in computational geometry achieve an $O(N\ log\ N)$ runtime performance [12]. Consider for instance the nearest neighbor problem described in Equation (1). A related 'N-body problem' would be finding all nearest neighbors:

$$\forall x_q,\ Compute\ NN(x_q) = \arg\min ||x_q - x_r|| \tag{2}$$

Using kd-trees as a supporting structure for the task will reduce the computation cost from $O(N^2)$ to $O(N\ log\ N)$ [12]. The assumptions in that case is that $N_R = N_Q = N$ and we can ignore the cost of constructing a kd-tree as it will be amortized over time. However, even this cost is prohibitive for the massive datasets in machine learning.

In his Ph.D. thesis and subsequent publications A. Grey devises a dual-tree framework that builds on top of the existing space partitioning data structures, such as kd-trees but also ball trees, and applies divide-and-conquer algorithmic strategy to bring forward a $O(N)$ expected runtime algorithm [12]. It exploits the characteristic of N-body problems that each data point in the reference dataset N_R has to be compared with all of the data points from the query dataset Q_R and introduces pruning rules to be able to reduce the required number of operations. Instead of using a single space-partitioning data structure, the algorithm utilizes two trees and extends a single tree traversal to a dual tree traversal. It needs to be mentioned, that oftentimes, the reference and query datasets are the same ($N_R = N_Q$). The dual tree framework is best exemplified on the all-nearest neighbor search.

The fundamental idea is to process the points in chunks instead of making individual queries for each point x_q. As the space-partitioning data structures group similar points together, work of finding nearest

neighbor will be shared among the similar query points.

Algorithm 3 extends the single tree Algorithm 2 to support a dual-tree traversal. In contrast to the single tree traversal where an input query point x_q is checked against the nodes of the kd-tree, the dual-tree version compares the tree nodes to each other, that is complete chunks of points at once that a particular tree node references.

The main modifications as compared to the single tree traversal Algorithm 2 are:

1. $x_q \rightarrow query\_node$
2. $dist(x_q, node) \rightarrow dist(query\_node, reference\_node)$
3. 2-way recursion transforms to the 4-way recursion

As the result, the runtime complexity of the algorithm is reduced from $O(N\ log\ N)$ to $O(N)$.

Figure 3 visualizes the difference between single and dual tree traversal algorithms. Whereas in the single-tree traversal individual query points are compared to the nodes in order to find the nearest neighbor, the dual tree traversal compares chunks of points (referenced by the tree nodes) first and resolves to brute force point-to-point comparisons only when the traversal reaches two leaf nodes.

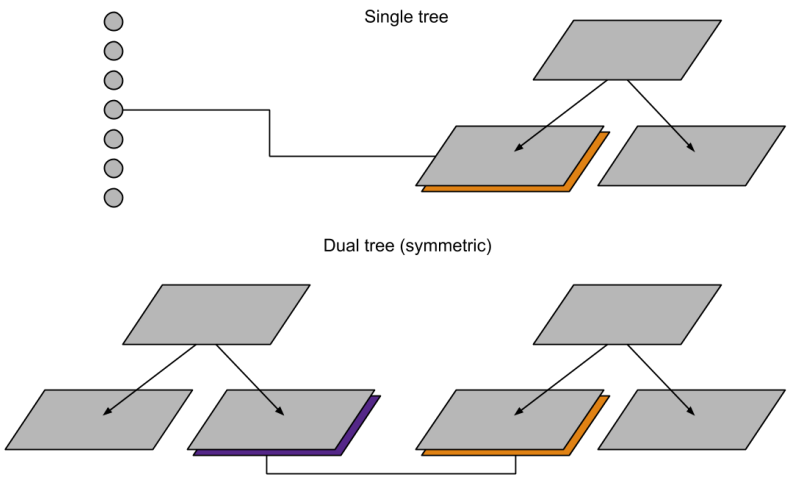

Figure 3.: Single-tree vs dual-tree traversal [13]

Two important operations in dual tree traversal is the exclusion and inclusion. First set to infinity, as we traverse the nodes, we store the maximum distance between two nodes as a future threshold value.

Algorithm 3: *AllNN*. All-nearest-neighbor search [12, 13]

Input: query point x_q,
query_node query node (set to the tree root on the first entry),
reference_node reference node (set to the tree root on the first entry), d initial distance set to ∞ on entry,
x_{best} best candidate found so far,
d_{best} corresponding best distance
Output: x_{best}, d_{best}
begin
  if $d > Q.d_{best}$ **then**
    // disregard (prune) the node
    return
  end
  if *is_leaf_node*(*query_node*) *and is_leaf_node*(*reference_node*)
  then
    $[x_{best}, d_{best}] =$
    $AllNNBase(query\_node, reference\_node, d_{best})$
    $Q.d_{best} = \max_Q d_{best}$
  else if
  !*is_leaf_node*(*query_node*) *and is_leaf_node*(*reference_node*)
  then
    ...
  else if
  is_leaf_node(*query_node*) *and* !*is_leaf_node*(*reference_node*)
  then
    ...
  else if
  !*is_leaf_node*(*query_node*) *and* !*is_leaf_node*(*reference_node*)
  then
    $[reference\_node_{closer}, reference\_d_{closer},$
    $reference\_node_{further}, reference\_d_{further}] =$
    $order\_by\_dist(query\_node.left\_child,$
    $reference\_node.left\_child, reference\_node.right\_child)$
    $AllNN(query\_node.left\_child, reference\_node_{closer},$
    $reference\_d_{closer}, x_{best}, d_{best})$
    $AllNN(query\_node.left\_child, reference\_node_{further},$
    $reference\_d_{further}, x_{best}, d_{best})$

    $[reference\_node_{closer}, reference\_d_{closer},$
    $reference\_node_{further}, reference\_d_{further}] =$
    $order\_by\_dist(query\_node.right\_child,$
    $reference\_node.left\_child, reference\_node.right\_child)$
    $AllNN(query\_node.right\_child, reference\_node_{closer},$
    $reference\_d_{closer}, x_{best}, d_{best})$
    $AllNN(query\_node.right\_child, reference\_node_{further},$
    $reference\_d_{further}, x_{best}, d_{best})$
    $Q.d_{best} = \max(Q.left\_child_{best}, Q.right\_child_{best})$
  end
end

Subsequently the value is updated to a smaller number if the maximum distance between two nodes is below the threshold.

We do not need to explore a branch, therefore pruning it, when the minimum distance between corresponding nodes is greater than the maximum distance of two nodes for the candidate nearest neighbors. Conversely, the branch needs to be explored further if the minimum node-to-node distance is smaller than the threshold value. This rule exploits properties of triangle inequality.

APPLICATIONS OF THE DUAL TREE FRAMEWORK The dual tree framework has been credited to be applicable to a range of machine learning tasks some of these applications are referenced below. In "'N-Body' Problems in Statistical Learning" [11] and later in ""Fast Algorithms and Efficient Statistics: N-Point Correlation Functions"" [21] the authors present a range searching and a two-point correlation algorithm based on single-tree traversal, followed by dual-tree approaches to the algorithms. Along with the proposed transition from a single to dual tree implementation of the algorithms, the paper also introduces several important concepts:

- Use of the cached sufficient statistics [19] – extra information that can be stored inside of the kd-tree nodes, such as number of points belonging to the node and coordinates of the bounding rectangles – to accelerate the computations.
- Redundancy elimination during the dual-tree traversal to avoid computations which have already been compared in reverse order.
- Extendibility of the-dual tree framework to a multiple tree approach as dictated by the problem.

The speed-ups achieved by the dual-tree approach on the two-point correlation function have reduced the time complexity from $O(N^2)$ to $O(N\sqrt{N})$ which translates to a computation time of 10 hours for a database of 10^7 compared with $10,000$ hours ($>$ 1 year) using a naive method [21].

In "Nonparametric Density Estimation: Toward Computational Tractability" [14] the dual-tree framework is applied to kernel density estimation rendering an empirically estimated runtime performance of $O(N)$, which is significantly faster than existing approaches. The authors propose to make use of the kd-trees as an underlying data structure for the datasets of up to 10 dimensions. For higher dimensions, where kd-trees are known to perform much worse due to exponential requirements as to the size of underlying dataset, a better scaling data structure called ball-tree [20] should be used.

"Tree-Independent Dual-Tree Algorithms" [7] presents a meta-algorithm to allow for the development of the dual-tree algorithms in a tree-independent manner. In particular, the meta-algorithm consists of the four parts: a space partitioning tree, pruning dual-tree traversal, point-to-point base case, and a pruning rule. The paper formalizes a tree traversal (single and dual) as well as a pruning dual-tree tree traversal. The meta-algorithm is then applied to the tree problems: all-k-nearest neighbor search, range search, Boruvka's algorithm, and kernel density estimation.

2.4 GEOMETRY-BASED ENSEMBLES

We have described kd-trees and the dual-tree framework that enables extending a single tree to a dual tree traversal with custom inclusion/exclusion rules to achieve a near-linear performance for a number of n-body problems. This section puts spotlight on one promising machine learning classification technique which could potentially benefit from the application of the dual trees.

CHARACTERIZING BOUNDARY POINTS. Geometry-based ensembles (GE) is a simple classification algorithm that exhibits superior performance compared to some machine-learning techniques, while being commensurate to the kernel methods. GE are build upon the notion of *characterizing boundary points* (CBP) – dataset points that form an optimal boundary between two classes based on a specific notion of robustness and margin [23].

Consider an illustrative example depicted by Figure 4. The CBPs are the black-colored points (b) that are formed at half the distance between the generating points that belong to opposing classes $\{+1, -1\}$. The set CBPs forms an optimal geometric boundary (a) between two classes.

Formally, given a labeled training set of M points $S = (x_i, y_i)$, where $x_i \in \mathbb{R}^d$ belonging to class $y_i \in +1, -1, i = 1...M$, a characteristic boundary point $x_{cp}^{i,j} \in \mathbb{R}^d$ is defined between any two training points (x_i, x_j) that fulfill following conditions:

1. x_i and x_j belong to different classes.
2. There is no other point in the dataset M that is located closer to the $x_{cp}^{i,j}$ than the corresponding x_i and x_j that form it. Given any point $x_k : \{x_k \in \mathbb{R}^d | (x_k, y_k) \in S\}$, then:

$$||x_i - x_{cp}|| \leq ||x_k - x_{cp}|| \tag{3}$$

$$||x_j - x_{cp}|| \leq ||x_k - x_{cp}|| \tag{4}$$

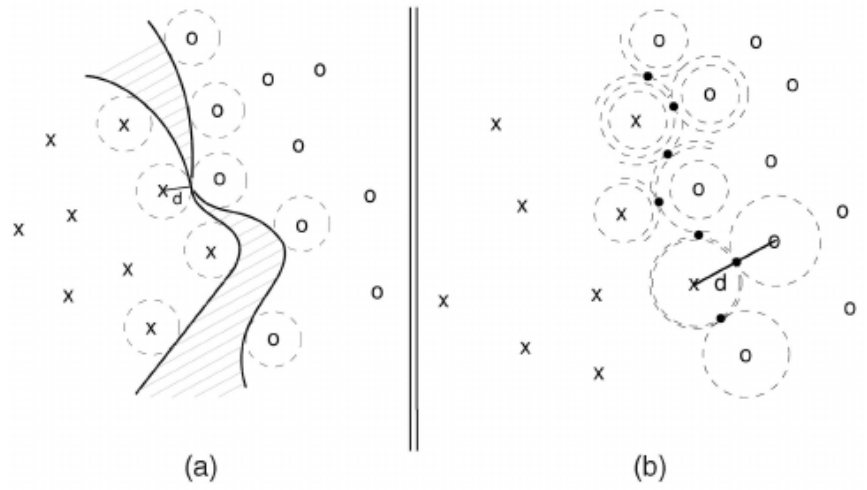

Figure 4.: Decision boundary formed by the CBPs (a) and their areas of influence (b) [24]

3. The characteristic boundary point is located at half distance from both of the generating points (middle point between x_i and x_j):

$$x_{cp}^{i,j} = \frac{1}{2}(x_i + x_j) \tag{5}$$

These conditions defining CBPs are illustrated in Figure 5. Point $x_{cp}^{i,j}$ is a CBP, as there is no other point – including x_k – which is located closer to $x_{cp}^{i,j}$ than x_i and x_j are. Note that x_i and x_j belong to different classes.

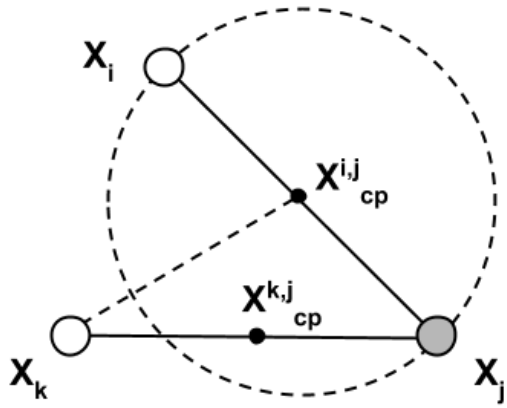

Figure 5.: Illustration of the characteristic boundary point definition [23]

The construction of CBPs is based on the restricted version of the Gabriel graph [10], where edges are only allowed to form between

points that belong to different classes. Figure 6 illustrates a full Gabriel graph constructed on the dataset.

For reasons that will become apparent later it is important to establish the place that Gabriel graph takes in the hierarchy of proximity graphs [6, 25]:

$$\begin{aligned} &\textit{Nearest neighbor graph} \\ \subseteq\ &\textit{Euclidean minimum spanning tree} \\ \subseteq\ &\textit{Relative neighborhood graph} \\ \subseteq\ &\textit{Gabriel Graph} \\ \subseteq\ &\textit{Delauney triangulation} \end{aligned} \tag{6}$$

Nearest neighbor graph is built by connecting two vertices x_i and x_j with an edge of length $d(i,j)$ only if there exists no other vertex x_k whose distance with x_i is smaller than $d(i,j)$. Being more restrictive in terms of connecting the vertices than the Gabriel graph, the nearest neighbor graph is a subgraph of it. Figure 7 illustrates the result of the more restrictive nature of the nearest neighbor graph as contrasted to the previously shown Gabriel graph (Figure 6).

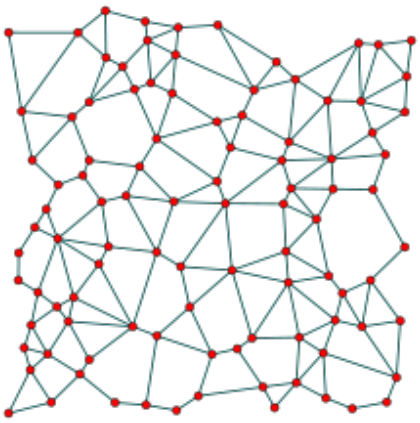

Figure 6.: Gabriel graph [5]

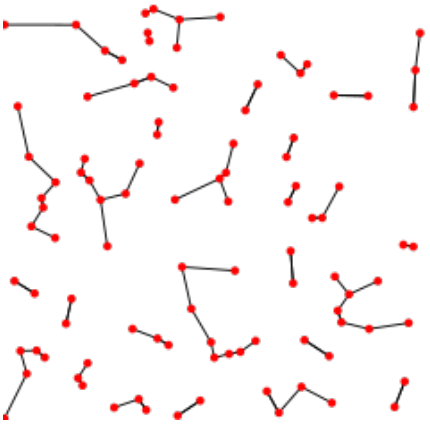

Figure 7.: Nearest neighbor graph [4]

A naive approach to computing CPBs would be to build a provisional $x_{cp}^{i,j}$ for all the points in the dataset M and then check that there exists no other point x_k in the dataset M whose distance to one of the generating points x_i or x_j is smaller than the distance between $x_{cp}^{i,j}$ and x_i or x_j. The complexity of this solution is $O(dM^3)$.

However, we could derive a better algorithm based on the formula that Matula and Sokal provide to define the Gabriel graph edges [18].

$$d(i,j)^2 < d(i,k)^2 + d(j,k)^2 \quad \forall i,j,k \in M, i \neq j \neq k \tag{7}$$

Accordingly, i and j form a valid edge of the Gabriel graph, if the squared distance $d(i,j)^2$ is smaller than the sum of the squared distances of i and j to any other point in the dataset. Adopting this formula to the task of locating CBPs, we obtain a faster Algorithm 4. In contrast to the naive brute-force approach it exhibits a lower complexity of $O(M^3 + dM^2)$ as no explicit computation of provisional CBPs is required any longer.

Algorithm 4: *Sokal*. Sokal algorithm for CBP construction

Input: Set of data points $S = \{x_i, y_i\} \in \mathbb{R}^d$ belonging to class $y_i \in \{+1, -1\}$
Output: Set of tuples of indexes $\{(i,j)\}$ that identify the generating data points of CBP
Compute squared Euclidean distance $d(i,j)^2$ from each point x_i to x_j in M
Initialize the set of indexes that define the CBP, $E = \{\}$ **begin**

```
    foreach x_i | y_i = +1 do
        foreach x_j | y_j = −1 do
            foreach x_k | y_k = +1 or y_k = −1 do
                cbp_found = true;
                if x_i ≠ x_k and x_j ≠ x_k then
                    if d(i,j)^2 ≥ d(i,k)^2 + d(j,k)^2 then
                        cbp_found = false;
                        break;
                    end
                end
            end
            if cbp_found then
                E = E ∪ {(x_i, x_j)}
            end
        end
    end
end
```

BOOSTED OGE. Boosted Geometry-Based Ensembles [23] is a result of applying gradient boosting to the original Geometry-Based Ensembles (GE) framework [24] in order to achieve controlled complexity of the model.

GE constructs an additive model based on the linear combination of classifiers. Based on the obtained CBPs, a set of base linear classifiers is constructed in the following way [23]:

$$\pi_{x_{cp}^{i,j}}(x) = (x - x_{cp}^{i,j})\vec{n}_{x_{cp}^{i,j}} \qquad \vec{n}_{x_{cp}^{i,j}} = \frac{x_i - x_j}{\| x_i - x_j \|} \tag{8}$$

Having obtained the classifiers, the ensemble is created via an additive model F:

$$\begin{gathered} F(x) = \sum_{k=1}^{N} \rho_k h_k(x) = \sum_{k=1}^{N} \rho_k sign(\pi_k(x)) \\ \text{where } N \text{ is the number of CBPs} \\ \text{and } \rho \text{ is a weighting vector} \end{gathered} \tag{9}$$

The simplest approach to gradient boosting uses L2-penalized least squares incremental residual minimization as described in Algorithm 5.

Algorithm 5: *BoostedOGE*. L2-penalized least squares GE boost

begin
  $A(k,i) = sign(\pi_k(x_i)) \quad k \in \{1...N\}, i \in \{1...M\}$;
  $F_0(x) = y$;
  for $t = 1$ *to* T **do**
    $\hat{y} = y_i - F_{t-1}(x_i) \quad i = 1, M$;
    $\rho = \frac{\hat{y}^T A}{M+\lambda}$;
    $a_t = \arg\min_a \sum_{i=1}^{M} (\hat{y}_i - \rho_a h(x_i; a))^2$;
    $F_t(x) = F_{t-1}(x) + \rho_t h(x; a_t)$;
  end
end

According to the experimental results [23] the regularized versions of Boosted OGE with L2-penalization performed on par with the reference classifiers SVM and OGE, while outperforming Adaboost with decision stumps.

Boosted OGE has a nice property that creation of the set of classifiers (predefined by CBPs) and the incremental optimizations are decoupled. This makes the algorithm useful for online and parallel extensions [23].

Nevertheless, Boosted OGE has clear inferior performance to the reference classifiers on some of the datasets. The author contemplates, that it could be caused by the limited space of base classifiers in the selection process. Being a subset of Gabriel graph, the number of CBPs is limited by the same strict geometric rules narrowing the space of potential classifiers. A relaxation of the CBP definition that

would allow k points to intrude the hypersphere is recommended as a potential strategy to expand the space of classifiers.

3

PROPOSAL

This master thesis proposes a novel application of a dual-tree framework to a task of computing characteristic boundary points. Due to the inherent $O(N^3)$ computational complexity of the state-of-the-art Sokal algorithm the objective of devising a faster solution gains importance for medium and large sized datasets.

3.1 FASTER CBP COMPUTATIONS

Let us define the reference dataset X_R as data points belonging to class -1 and the query dataset X_Q as the points of class $+1$. While the task of computing CBPs is not strictly an n-body problem, it does share one characteristic common to this class of problems: in the worst case deriving a solution requires comparison of each data point in the reference set X_R to each data point in the query set Q_R. Furthermore, according to the Equation 7, we also need to compare the distances between the pairs of X_R and X_Q to the points $X_R \cup X_Q$ in order to verify validity of a CBP.

WEAKNESSES OF SOKAL ALGORITHM Consider for instance a reduced *banana* dataset plotted in Figure 8. The Sokal algorithm is largely oblivious to the spatial location of the data points:

1. Even if there are interfering points between x_i and x_j that clearly break their generating ability, the algorithm will still perform the comparison.

2. When comparing the distance between x_i and x_j to the rest of the dataset $X_R \cup X_Q$, no heuristics is applied to ensure that the point most capable of breaking the CBP tie are considered first. The algorithm iterates sequentially through the elements of $X_R \cup X_Q$.

DUAL-TREE ALGORITHM TO FIND CBPS Observing this relationship, it is reasonable to argue that bringing space-awareness to the algorithm would make it more efficient by reducing excessive computations. While we could start designing a point-to-point algorithm based on a single kd-tree, our awareness that the task of computing

CBPs is related to n-body class of problems suggests applicability of a dual-tree approach. Algorithm 6 provides a conceptual skeleton for the dual-tree algorithm we have developed to deal with CBP computations.

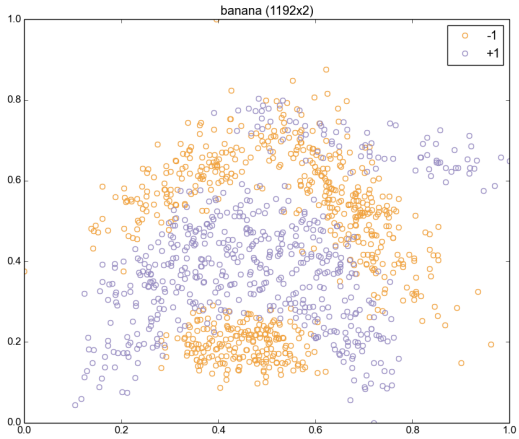

Figure 8.: banana (1192x2). Representation of the reduced banana dataset.

Following the structure laid out by the all-nearest-neighbors Algorithm 3, we start a depth-first dual-tree traversal where the reference nodes are compared to the leaf nodes:

- At each step a possibility to prune the current pair (N_R and N_Q) is evaluated and the pruning is performed: *CBP_Prune* (Section 3.1.2)
- If the pair can not be pruned the recursive traversal through the current node descendants ($N_R.less/greater$ and $N_Q.less/greater$) is continued until two leafs are reached.
- As soon as the reference and query nodes represent leaf nodes, the *CBP_Local_Search*() (Section 3.1.6) is called. Its task is to locate the potential generating points (i and j) as well as potential intruding points k and extend the relevant lists with their indexes.

After the dual-tree traversal has completed and we have filled lists of generating and intruding points, we feed these lists to the baseline Sokal Algorithm 4 (slightly modified to work with adjusted inputs). Thus, the preprocessing with dual trees will help to reduce the search space that the Sokal has to operate on when applied to the raw datasets positively affecting the computation time.

Algorithm 6: *CBP_Traversal*. Dual-Tree traversal algorithm to find characterizing boundary points

Input: N_R reference node, N_Q query node; $i = []$, $j = []$, $k = []$ three multidimensional lists, empty at the start

Output: 3 multidimentional lists: i, j, and k representing the indexes of the data points from M that need to be evaluated by the Sokal algorithm

```
begin
    // see Section 3.1.2
    if CBP_Prune(N_R, N_Q) then
        return;
    end
    if is_leaf_node(N_R) and is_leaf_node(N_Q) then
        // see Section 3.1.6
        CBP_Local_Search(N_R, N_Q, i, j, k);
    else if is_leaf_node(N_R) and !is_leaf_node(N_Q) then
        CBP_Traversal(N_R, N_Q.less, i, j, k);
        CBP_Traversal(N_R, N_Q.greater, i, j, k);
    else if !is_leaf_node(N_R) and is_leaf_node(N_Q) then
        CBP_Traversal(N_R.less, N_Q, i, j, k);
        CBP_Traversal(N_R.less, N_Q, i, j, k);
    else if !is_leaf_node(N_R) and !is_leaf_node(N_Q) then
        CBP_Traversal(N_R.less, N_Q.less, i, j, k);
        CBP_Traversal(N_R.less, N_Q.greater, i, j, k);

        CBP_Traversal(N_R.greater, N_Q.less, i, j, k);
        CBP_Traversal(N_R.greater, N_Q.greater, i, j, k);
    end
end
```

3.1.1 *Dual-Tree Architecture*

We have considered two different choices to partition the data. The first one is to construct two distinct kd-trees: one for X_R and the other one for X_Q. An alternative solution is to build one tree on the whole dataset M irrespective of the class labels and reuse it during the dual-tree traversal. The result of both architectures are visualized in Figures 9 and 10. The advantage of using two trees is the simplicity of implementation and the resulting dual-tree traversal. However, using a single tree on the whole dataset produces uniform tree cells which would prove critical during for the pruning strategies. In order to fully benefit from pruning, we have settled for the architecture where the same tree data structure is used for query and reference nodes.

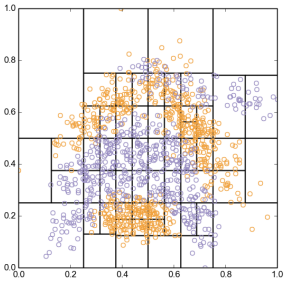

Figure 9.: Dual-tree architecture: one tree

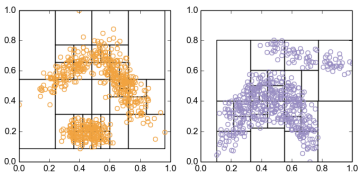

Figure 10.: Dual-tree architecture: two trees

3.1.2 *Pruning Rules*

We have devised several pruning rules to achieve two parallel objectives: 1) to implement a strict pruning rule that reduces the dataset by discarding only illegitimate points of the datasets, 2) to bring forth a more relaxed pruning rule that would offer a good compromise between improved efficiency and accuracy of the results.

3.1.3 *Exact CBP Computation*

Given that the Sokal algorithm finds all CBPs, the processing of the points after application of the strict pruning rule should provide results identical to the baseline solution.

Conservative maximum distance pruning. Having partitioned the datasets either with a single or multiple tree, we can access the tree cells which contain data points clustered according to their spacial location. Consider a query and reference nodes as depicted in Figure 11. We want to prune these two nodes if it is certain that the

other points located between the two nodes render their capacity to generate CBPs void. We can reverse the inequality Equation 7 in order to identify non-generating points and substitute point-wise distances with distance measurements between the nodes, which renders Equation 10.

$$d_{min}(N_R, N_Q)^2 \geq d_{max}(N_R, k)^2 + d_{max}(N_Q, k)^2 \tag{10}$$

Thus, the first version of the pruning rule states that nodes N_R and N_Q – and therefore any points they contain – are not generating nodes if there exists a point k in the dataset the sum of maximum squared distances to each one of the nodes N_R and N_Q is less than the minimum squared distance between those nodes. This rule is visualized in Figure 11.

Remember, that we only have two cheap distance functions at our disposal: minimum and maximum distance. Having that in mind, note the role that the choice of min/max distance plays in the definition of the rule:

1. Minimum distance between two hyperrectangles $d_{min}(N_R, N_Q)^2$ ensures that we consider the shortest possible distance between any two points belonging to the respective nodes. While the distance can be calculated, there is no information as to the exact location of those points.

2. Instead of the minimum distance, a maximum distance between hyperrectangles and point k – $d_{max}(N_R, k)^2$ and $d_{max}(N_Q, k)^2$ – is used to consider the distances between k and the furthest points in the hyperrectangle in order to ensure that point k would break generating capacity of the most distant data points to it. Compare that to the alternative – using minimum distance – as indicated in on Figure 11 in red color. In that case, the formula would indicate that the two nodes can be pruned whereas that action would eliminate valid CBPs.

While this rule is valid, it is overly conservative by considering the whole hyperrectangle areas when calculating the maximum distances $d_{max}(N_R, k)^2$ and $d_{max}(N_Q, k)^2$. It is, in fact sufficient to calculate the maximum distance between k and the side of the hyperrectangle that faces the other hyperrectangle: see Figure 12.

Maximum distance pruning. One possible solution could be to calculate all four distances in two dimensions, sort them, and take the second smaller distance which should be the furthest distance of the side facing the other node. However, this approach quickly becomes unfeasible in higher dimensions where the number of vertices

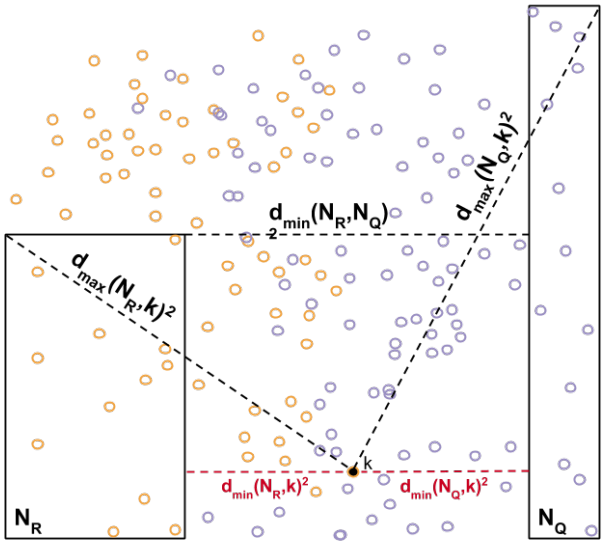

Figure 11.: Pruning rule

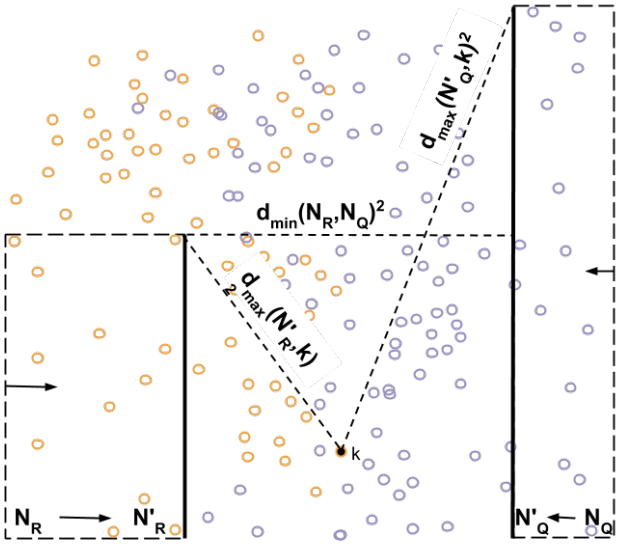

Figure 12.: Tighter pruning rule

is 2^d [26]. Instead, the proposed idea is to "squeeze" the hyperrectangles towards each other in order to produce modified tight rectangles. Afterwards we can apply a cheap maximum distance function to calculate the distance from the side of the rectangle to the point k (Figure 12). Figure 13 shows a possible worst case to justify the use of the maximum distance measure to the k from the tight nodes N'_R and N'_Q: using the minimum distance (indicated by dashed red colored lines) may in this case misleadingly suggest to prune the node

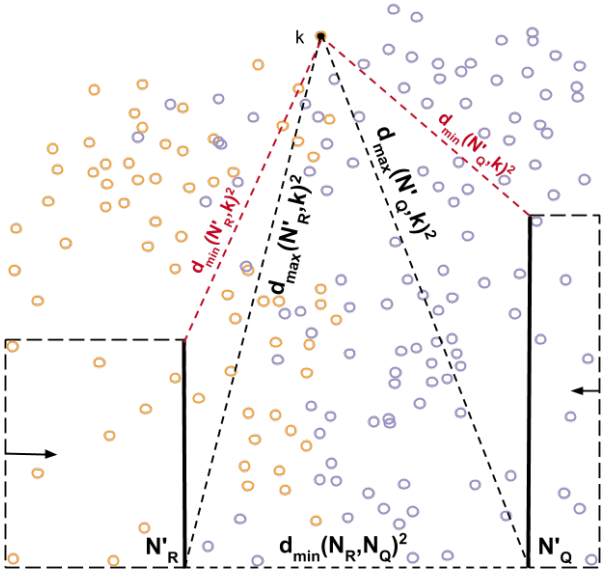

Figure 13.: Worst case

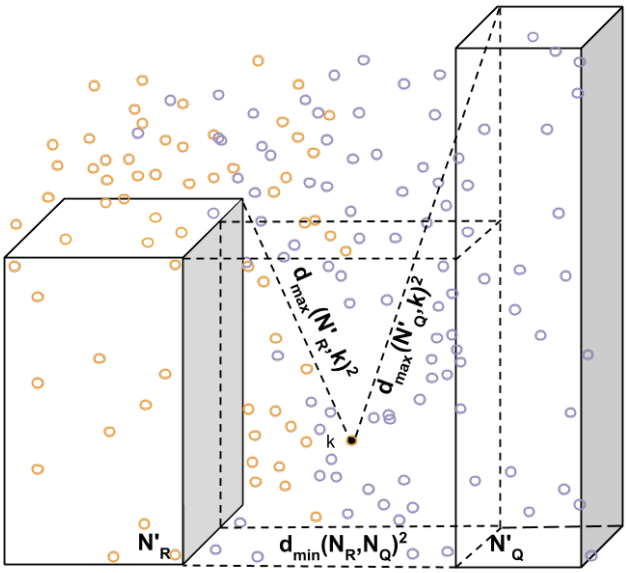

Figure 14.: Strict pruning in 3D

pair, while a more conservative maximum distance measure ensures that the furthest points within the nodes are considered during the computation.

3.1.4 *Approximate CBP Computation*

Although a proposed conservative pruning strategy would lead to the exact CBP computation, as the experiments show, the rate of suc-

cessful prunings quickly degrades as we go into the higher dimensions. Besides "curse of dimensionality" (see Section 4.1.4), another reason for that is the maximum distance measure we use to establish the validity of pruning: $d_{max}(N_R, k)^2$ and $d_{max}(N_Q, k)^2$. Observe for instance Figure 14: it demonstrates that while in certain cases the minimum distances between two nodes remain unchanged as we ascend into higher dimensions (when the hyperrectangles face each other), the node-point distances grow due to the added dimension component. Thus the result of same equation that defines if two nodes can be pruned could be positive in 2d and negative in 3d, as the maximum distances to k would grow disproportionately to the distances between the nodes.

Therefore, we propose alternative pruning rules, whose purpose is two-fold: 1) guarantee execution of pruning in higher dimensions where the first strategy has its limitations, 2) provide a more relaxed pruning rule in order to accelerate computation of CBPs at the expense of accuracy. In this respect, two less strict pruning strategies have been devised.

Minimum distance pruning. Lets revisit Figure 13. It was noted, that we need to use the maximum distance between point k and the nodes to ensure that k breaks the generating capacity of the most remote points in N_R and N_Q. In order to relax the rule we now substitute maximum with the minimum distance (indicated in red color). The effect of that relaxation will be felt most in the higher dimensions, as indicated in Figure 15, as the distance between the k will not grow disproportionately to the minimum distances between the nodes. Equation 11 formalizes the new relaxed pruning rule.

$$d_{min}(N_R, N_Q)^2 \geq d_{min}(N'_R, k)^2 + d_{min}(N'_Q, k)^2 \tag{11}$$

Nonadjacent pruning. The second strategy takes the relaxation of the pruning rule one step further. Instead of computing the intra-node and point-node distances we prune two nodes if they are not adjacent: that is, they have no common point of contact with each other. Although, this strategy is severely aggressive, we can justify it by expecting that a node N_{IB} would contain enough points to break the generating capacity of points within N_R and N_Q. In order to establish whether two nodes are adjacent, we use the minimum distance between them: if the returned value is greater than 0 the nodes are not adjacent. Figure 16 illustrates this pruning rule, formalized by the Equation 12.

$$d_{min}(N_R, N_Q)^2 > 0 \tag{12}$$

While the two relaxed pruning rules are expected to prune approximately equal amount of nodes for the majority of trivial cases, the

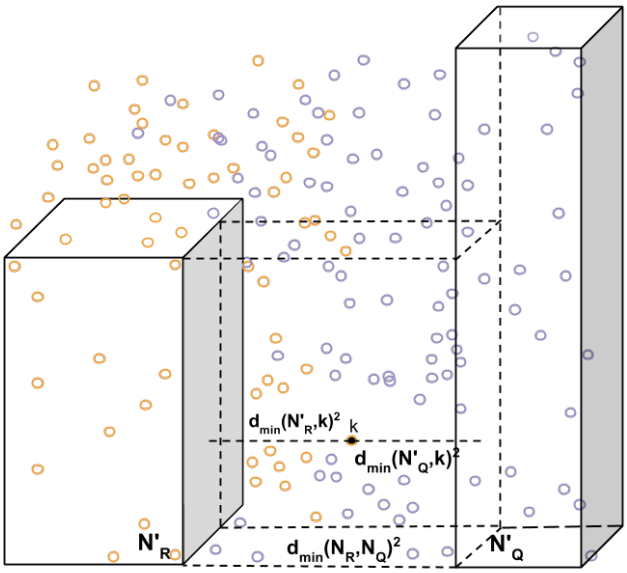

Figure 15.: Relaxed pruning rule: minimum distance to point

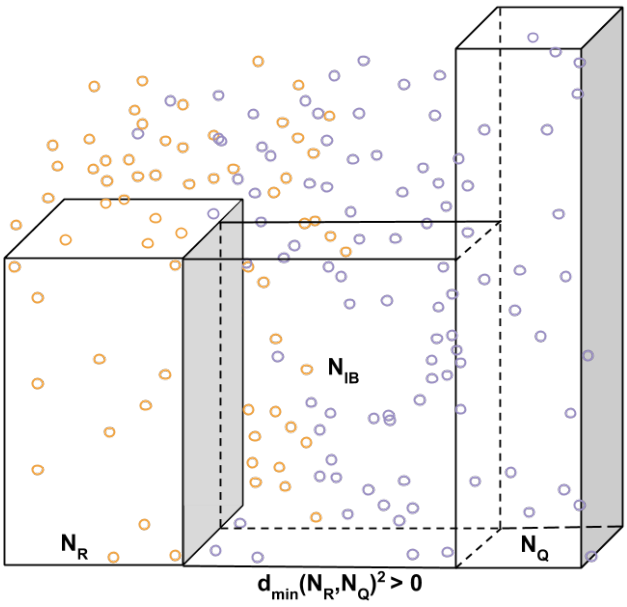

Figure 16.: Relaxed pruning rule: nonadjacent hyperrectangles

nonadjacent pruning is computationally cheaper because we reduce the number of distance measurements required from 3 to 1, but also free ourselves up from the requirement of finding k.

3.1.5 *Finding Reference Point k*

Earlier the discussion has centered around computing the intra-node and point-node distances required for the pruning rules. We have, however, not addressed the task of finding point k based on which the ability of nodes N_R and N_Q to generate CBPs is judged. Intuitively, a good choice for the point would be a centroid of the area that encloses two nodes. Any point close enough to the centroid would be a strong candidate to activate pruning.

Thus, we start with the most straightforward approach: constructing a minimum bounding hyperrectangle that includes both N_R and N_Q. As each node represented by the hyperrectangle is defined by two vectors – minimum and maximum coordinates of all the points that the node contains – we can easily devise Algorithm 7 that rearranges the coordinates to produce two new vectors that define the bounding hyperrectangle.

Algorithm 7: *BoundingHyperrectangle*. Constructing bounding hyperrectangle that encloses nodes N_R and N_Q

Input: N_R reference node, N_Q query node
Output: *bounding_hyperrectangle* bounding hyperrectangle
begin
  $bounding\_mins =$
  $min(N_R.mins, N_R.maxes, N_Q.mins, N_Q.maxes)$
  $bounding\_maxes =$
  $max(N_R.mins, N_R.maxes, N_Q.mins, N_Q.maxes)$
  `// Use scipy.spatial.Rectangle`
  $bounding\_hyperrectangle =$
  $Rectangle(bounding\_maxes, bounding\_mins)$
end

The result of the algorithm in 2D is illustrated by Figure 17. We can observe, however, that due to the unequal sizes of N_R and N_Q the bounding hyperrectangle is biased toward fatter and taller boxes and therefore has suboptimal centroid according to our intuition. Instead, we want the tightest possible hyperrectangle that captures the space between the nodes and does not include their outer sides. This desired hyperrectangle and its centroid C are depicted in Figure 18.

Algorithm 8 details the procedure of finding the tight rectangle visualized in Figure 18: 1) it merges the vectors of nodes N_R and

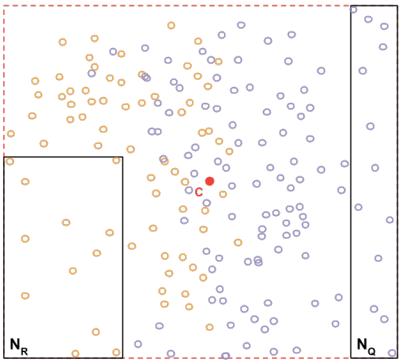

Figure 17.: Bounding hyperrectangle

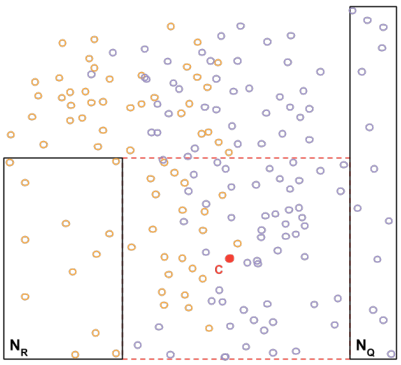

Figure 18.: Tight hyperrectangle

N_Q together, 2) sorts them in ascending order, 3) then the algorithm attempts to change the minimum coordinates (initially set to the minimum bounding hyperrectangle by Algorithm 7): if the *first two* successive sorted values correspond to minimum and maximum of the same hyperrectangle we can shift the boundary, because it would not cross the axes of another hyperrectangle, 4) the procedure is analogous for the vector of maximum values.

Algorithm 8: *TightHyperrectangle*. Constructing tight hyperrectangle that includes the minimum required side segments of nodes N_R and N_Q

Input: N_R reference node, N_Q query node
Output: *tight_hyperrectangle*
begin
  `// concatenate and sort points`
  $sorted\_points =$
  $sort([N_R.mins, N_R.maxes, N_Q.mins, N_Q.maxes])$
  $tight\_mins = copy\ BoundingHyperrectangle(N_R, N_Q).mins;$
  $next\_min = sorted\_points[1, :];$
  if $(tight\_mins = N_R.mins\ and\ next\_min = N_R.maxes)$
  $or\ (tight\_mins = N_Q.mins\ and\ next\_min = N_Q.maxes)$
  then
    $tight\_mins[idx] = next\_min;$
  end
  `// analogous but in reverse order for maxes`
  ...
  $tight\_hyperrectangle = Rectangle(tight\_maxes, tight\_mins);$
end

Based on the tight hyperrectangle found, we calculate its centroid. We then use the one nearest-neighbor query of kd-tree upon which

our dual tree is based to find a point in the dataset closest to the centroid. One caveat is that both nodes N_R and N_Q should be excluded from the search.

3.1.6 *Local Search*

To recap, two noted weaknesses of the Sokal algorithm (see Paragraph 3.1) where 1) excessive comparisons of potential generating points and 2) absence of meaningful heuristics when validating the generating points against potential intruders. Both result from the unawareness of the spacial structure of the dataset.

PROXIMITY HEURISTICS Pruning strategies are used to resolve the first weakness, whereas local search will be mainly beneficial for the heuristics that it brings to the Sokal Algorithm 4 when comparing the generating points with a list of intruding points. If we come back to the illustration demonstrating the concept of CBPs (Figure 5), then we could draw following conclusion: ordering potential intruding points according to their distance from prospective $x_{cp}^{i,j}$ beforehand will result in Sokal performing a more optimal reduced search. Clearly, the points that would break the capacity of i and j to generate a CBP are located close enough to the center $x_{cp}^{i,j}$.

According to the dual-tree representation of the partitioned space (Figure 18), the most intuitive center is the centroid of the tight hyperrectangle between N_R and N_Q. Thus, if we sort the intruding points in the dataset according to the approximate remoteness from the centroid C_{tight}, we would achieve a similar helping effect from a higher perspective. Fortunately, the k-nearest neighbor query of kd-trees returns the list of matching points ordered by the distance from the query point x (our centroid).

MINIMUM LOCAL BOUNDARY Aside from the introduced metaheuristics, we can also further limit the dataset to the points that can potentially break emerging CBPs. Figure 19 exemplifies how we restrict the space by a sphere boundary. The sphere boundary has its origin at the centroid of the bounding hyperrectangle with the radius set to the space diagonal [27] of this hyperrectangle: when performing a k-nearest neighbor search, the query will return only the points of the hyperrectangles within the boundary.

The choice of the rather large value for the sphere – diameter, as opposed to half of diagonal – is intentional. It is designed to prevent border cases, as depicted by Figure 20, where potential CBPs can be formed by the generating points at the vertices of the hyperrectangles.

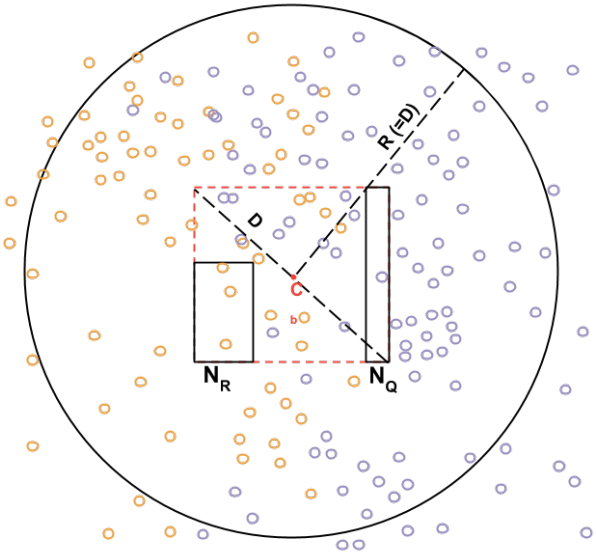

Figure 19.: Local Search

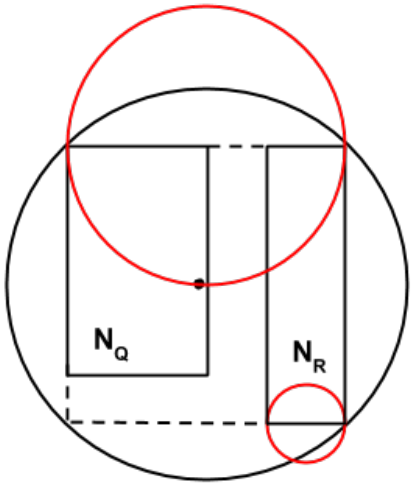

Figure 20.: Overly restrictive spheric boundary

We have extended the implementation of the kd-tree provided by scipy scientific package of python programming language [16] to support the following n-nearest-neighbor query operations:

- List of blocked node IDs tells the query to skip the passed nodes when performing a search. This is done to accelerate the operation as we already know which points belong to the nodes N_R and N_Q.

- In order to accelerate the query performance, we have added an option to the kd-tree to perform approximate nearest-neighbor queries: instead of strictly fetching the specified by k-nearest neighbors, the query returns all the points within the cells that are at the specified distance.

Setting n parameter to the size of the dataset achieves the desired result of filling a list of potential intruding points with indexes.

3.2 A CBP-BASED CLASSIFIER

One of the weaknesses of the original Boosted OGE classifier is its poor scalability in respect to the surge in dimension and size of underlying datasets. This goes back to the fact that the base classifiers are constructed on top of the CBPs.

The author argues that the Boosted OGE's subprime results on some of the datasets could be attributed to low number of base classifiers, due to the stringent rules of building CBPs. However, as we increase the dataset sizes and dimensions the number of CBPs has a drastic and unpredictable growth. Tables 1 and 2 expose the CPB growth.

Table 1.: Growth of CBPs as dataset gains more dimensions

EEGeye		CBPs		Twonorm		CBPs	
Size	Dim.	(count)	(%)	Size	Dim.	(count)	(%)
7200	2	9942		6660	2	3485	
7200	4	11199	112.64	6660	4	9237	265.05
7200	6	17276	154.26	6660	6	22095	239.20
7200	8	34494	199.66	6660	8	44796	202.74
7200	10	48507	140.62	6660	10	78571	175.40
7200	12	69765	143.82	6660	12	132153	168.20
7200	14	69854	100.13	6660	14	201042	152.13

Table 2.: Growth of CBPs as dataset increases in size

EEGeye		CBPs		Twonorm		CBPs	
Size	Dim.	(count)	(%)	Size	Dim.	(count)	(%)
1800	14	10751		1665	14	46354	
3600	14	58785	546.79	3330	14	115334	248.81
5400	14	45827	77.96	4995	14	201042	174.31
7200	14	69854	152.43	6660	14	303220	150.82

Consider, for instance, that we want to run Boosted OGE on EEG-eye dataset with 7200 instances and 10 dimensions, then the original Boosted OGE algorithm would need 1) to construct a matrix A of base classifiers applied to the training dataset (resulting dimension is 7200x48507) and 2) to calculate the best performer at each boosting iteration in order to augment the set of the strong classifiers. Clearly, that introduces both immense storage but also computational complexity making the use of the classifier limited to the smaller datasets.

We propose two alternative modifications to Boosted OGE to address its scalability issues (visualized in Figures 21 and 22).

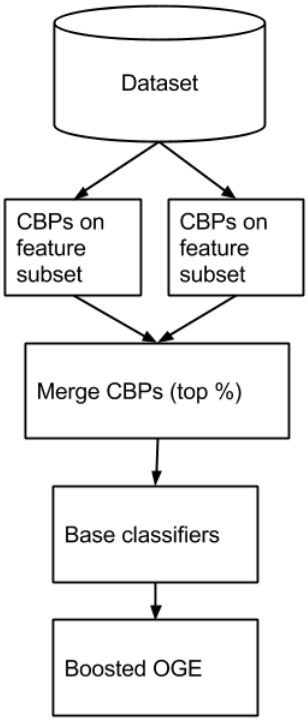

Figure 21.: Boosted OGE: filtered merge

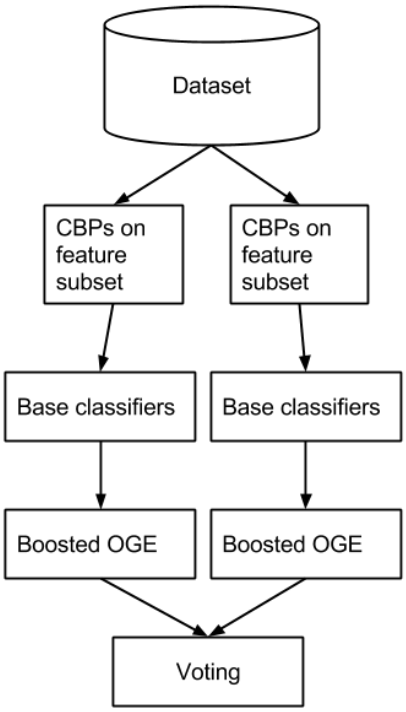

Figure 22.: Boosted OGE: cascade

Both modifications apply ideas from divide-and-conquer algorithm design paradigm to reduce computational and storage costs. They, however, approach the task in somewhat different manner.

Filtered merge (Figure 21). The first strategy is to split dataset in random feature subsets and to compute CBPs on these subsets. Afterwards, all the computed CBPs are merged into a single set and the base linear classifiers are built on top of them. While the default strategy is to merge all the CBPs together into one single set, we propose a selection criteria by which only a certain percentage of CBPs ranked according to their strength are chosen. We define strength

as frequency with which the same CBP appears in different feature subsets; if there is no overlap between CBPs from different feature subsets, then the algorithm simply selects a given share of CBPs.

Cascade (Figure 22) This modification resembles more the approach taken by random forest. Again, random feature sets are extracted from the dataset and the CBPs are computed based on them. However, instead of merging the CBPs into a single set, the *cascade* proceeds to generate base classifiers on each individual set, runs the boosting OGE on top of it and finally utilizes majority voting to make the prediction of the final label.

Both of the proposed solutions should provide computational speed improvements and reduction of storage requirements.

1. As we have seen, we can expect less CBPs in lower dimensions than in higher. Although *filtered merge* by combining the CBPs into a single set can potentially undo the gain of operating in lower dimensions, the filtering procedure ensures that a threshold value for CBP count is not exceeded.

2. Having less base classifiers reduces the computational cost of boosting iterations. Moreover, it adds a nice property that portions of the algorithm can be trivially parallelized.

Thus, we have obtained two models, tuned by the following parameters: 1) size of feature subsets, 2) number of CBP generators (*filtered merge*) or CBP generators/classifiers (*cascade*).

4

EXPERIMENTS AND RESULTS

4.1 UNDERSTANDING THE MODEL

Prior to jumping to the final results obtained with our algorithm, this section will attempt to give insight into the various factors of the model that are otherwise too easy to overlook if only the final values are considered. To support the findings, we would interchangeably consider only a small selection of the datasets as they adequately reflect general patterns of the model.

Specifically, we will discover the effects that leaf size of the underlying kd-trees, dimensionality of the dataset, and local search have on our dual-tree algorithm for faster CBP computations. This will be followed by the analysis of aspects related to the proposed modifications of the Boosted OGE classifiers: effect of thresholding on the CBP count, performance of the classifier based on genuine or weak CBPs.

Table 3.: Datasets

Dataset	Size	Dim
banana	5300	2
EEGEye	14980	14
Example	1014	5
magic	19020	10
ringnorm	7400	20
skin	245057	3
svmguide1	3089	4
transfusion	748	4
Twonorm	7400	20
waveform	5000	21

Table 3 lists the original datasets that have been used in the experiments (Table 11 in Appendix A provides the sources of these datasets). In order to explore effects of dimensionality and size but also to ensure that available hardware resources are able to operate

on the datasets, these where reduced in dimensions and size. To provide reader with information about which particular version of the dataset is used in an experiment, we follow the convention of writing the name of the dataset followed by "(size **x** dimension)" information.

4.1.1 *Effect of Leaf Size*

We have conjectured (Section 2.1) about the influence that the leaf size setting can have on our algorithm: while small value for the number of data points in leaf nodes leads to expensive dual-tree traversals, too large values would convert the kd-tree queries to a brute force procedure. However, the true situation in context of the CBP computations appears to have even more variability.

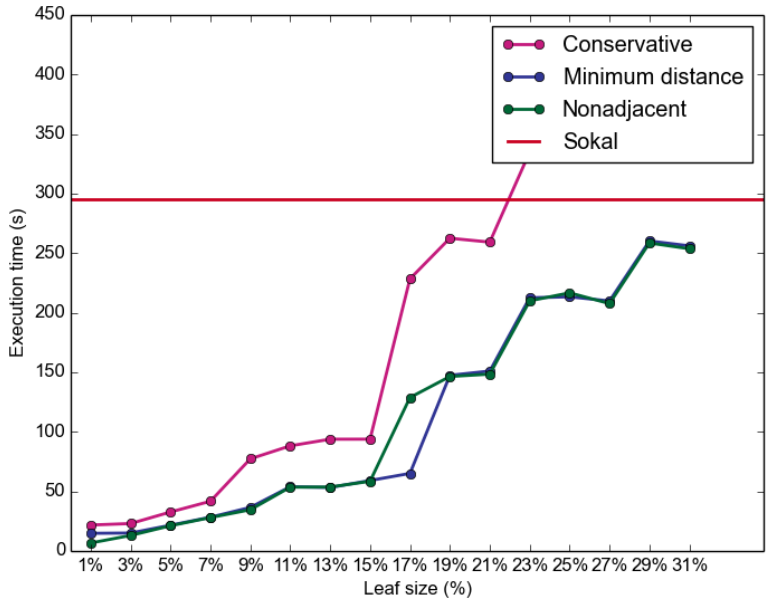

Figure 23.: EEGEye (7200x2): Leaf size growth

Figures 23 and 24 demonstrate the impact that the set leaf size (as percentage of the dataset size) values have on the time performance of the dual-tree algorithms compared to the baseline. While the relation between leaf size and execution ratio is clearly defined in a two-dimensional space, the picture is more vague in four dimensions. After a sharp rise in computation cost further increases in leaf sizes (19-31%) seem to have no correlation with algorithm performance.

We can take a look at the leaf size problem from a different perspective plotting the breakdown between dual-tree time and CBP computation time: Figures 25 and 26 represent performance of the

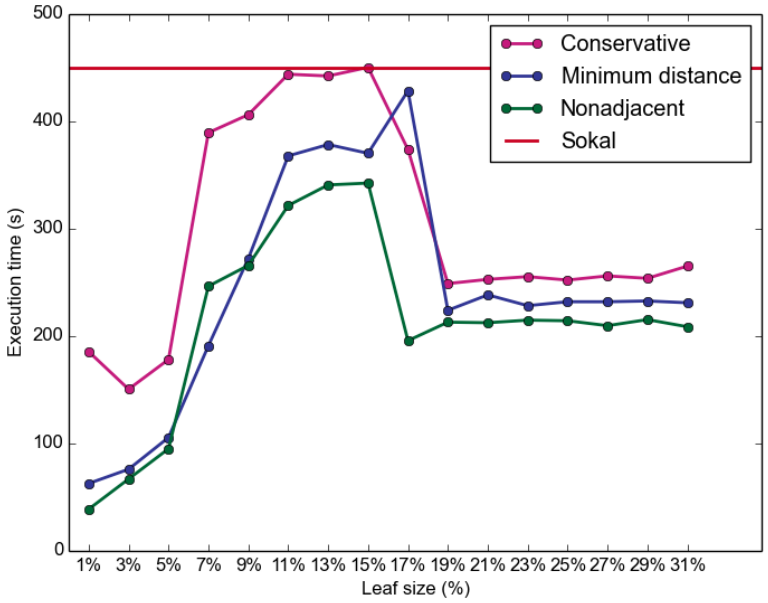

Figure 24.: EEGEye (7200x4): Leaf size growth

conservative strategy for different leaf sizes.

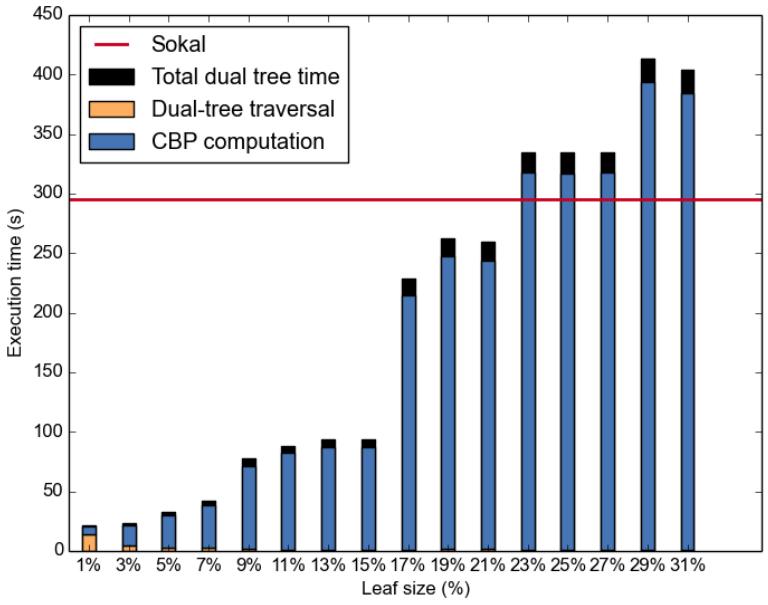

Figure 25.: EEGEye (7200x2): Break-down of dual-tree algorithm (conservative pruning) time consumption.

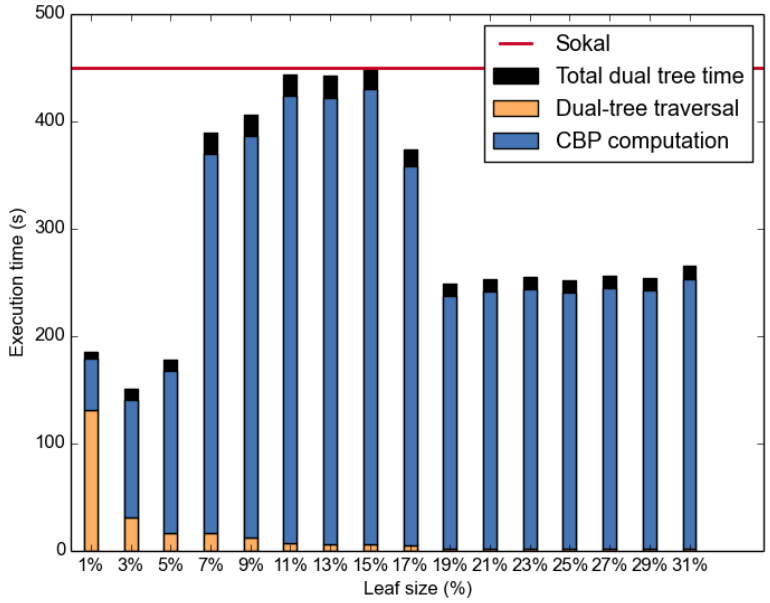

Figure 26.: EEGEye (7200x4): Break-down of dual-tree algorithm (conservative pruning) time consumption.

Both in two and four dimensional datasets dual-tree traversal corresponds to a significant portion of the total execution time. This implies that the algorithm is hard at work pruning the dataset and performing local search leading to the reduced time needed for the CBP computation. This relative ratio of dual-tree to CBP computation components gets distorted when the nodes become larger limiting the capacity of dual-tree traversal.

However, the question of the sudden performance plunge in EEGEye (7400x4) as well as reasonably good results for higher leaf sizes are still puzzling until we look at Figures 27 and 27.

The graphs demonstrate percentage of worst-case computations that the dual-tree versions have to perform as a percentage of the baseline Sokal worst case. Whereas the Sokal worst case is calculated as $|N_R| * |N_Q| * |N_R \cup N_Q|$, the dual tree worst case is defined by $\sum_{s=1}^{|S|} |N_{R_s}| * |S_{Q_s}| * |S_{R_s} \cup S_{Q_s}|$, where S is a set of subspaces that are extracted from N_R, N_Q, and $N_Q \cup N_R$ during the local search stage of the dual-tree traversal.

As can be seen, growing node sizes correspond to the general increase of computation count. While the trend is stronger in two dimensions, it is still notable in the four-dimensional dataset. The execution time bump reported in the EEGEye (7200x4) dataset is likely

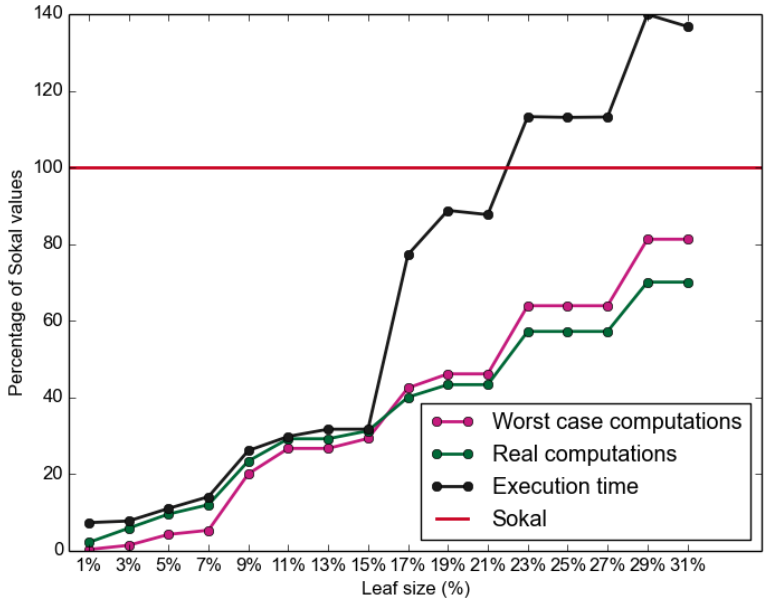

Figure 27.: EEGEye (7400x2). Worst case vs real computations in relation to the leaf size

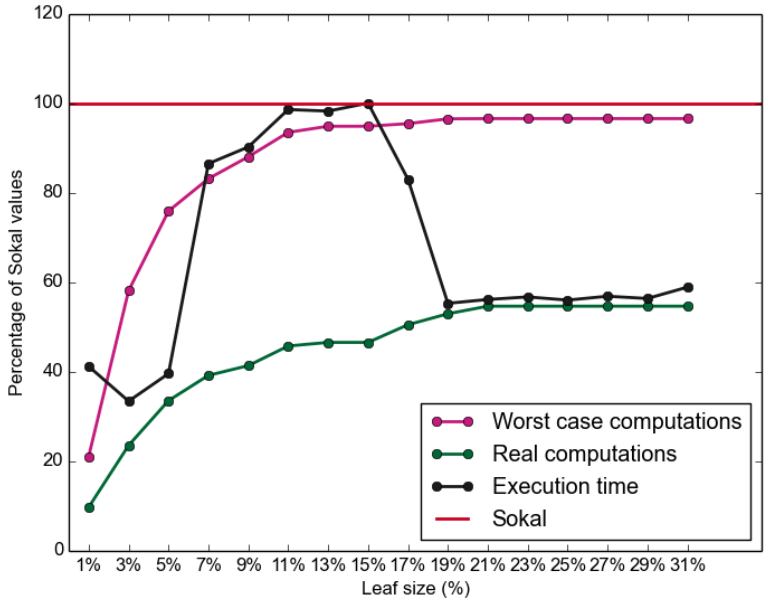

Figure 28.: EEGEye (7400x4). Worst case vs real computations in relation to the leaf size

an outlier that can be ignored, as the computation time diverges from the real computations line without any underlying cause (for instance, the dual-tree traversal time at this leaf size can be neglected, as indi-

cated in Figure 26). The sudden rise could be caused by numeric calculations made by the Sokal subtask.

Another interesting observation is the ever increasing gap between worst case and real computations. This particularity has less to do with the leaf size, but is rather a property of local search which is investigated in Section 4.1.3.

We have settled to run the experiments with a leaf size set to 3% of the dataset as it seems to be a value at which the combination of dual-tree traversal and CBP computations are well-balanced and exhibit superior performance. Nevertheless, while outside of the report scope, the issue of optimal setting for leaf sizes as proportion of the dataset size and dimension needs further investigation.

4.1.2 *Effect of Dimensionality*

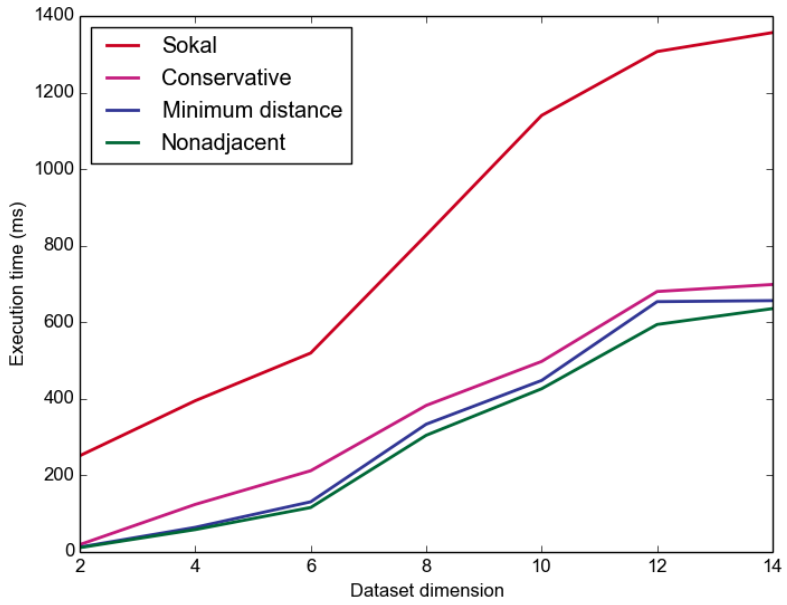

Figure 29.: Absolute time performance of Sokal and Dual-Tree algorithms as dataset dimensionality increases.

Figures 29 and 30 demonstrate a general behavior of Sokal and dual-trees with three pruning strategies. While the data is based on dataset EEGEye, the trend of worsening performance is observed across all datasets. Two aspects are of particular importance to us: 1) the increase of computational time of the Sokal algorithm as the dimensions grow, 2) the steep increase of the dual-tree solutions. In a 2-dimensional space the dual-tree algorithm demonstrate over a one

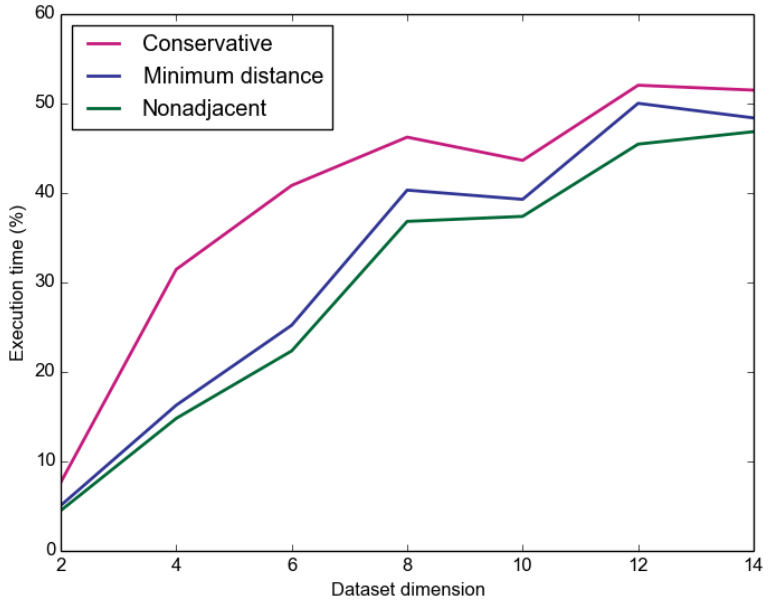

Figure 30.: Relative time performance of Dual-Tree algorithms compared to Sokal in as dataset dimensionality increases.

order of magnitude performance improvement over the Sokal. That gain, however washes off as we go into the higher dimensions until the relative performance of the dual-tree even outs at 12 to 14 dimensions to be merely twice as fast as Sokal.

One valuable observation that we can gain from Figure 30 is that at around 10 dimensions, all three versions of the dual-tree algorithms – conservative, minimum distance and nonadjacent pruning strategies – gravitate toward similar execution times. We can conclude than, that at that point, it is not the pruning strategies that distinguish each version but rather some other common factor that aligns their performance. Section 4.1.3 looks deeper into the role the local search plays in the performance yields we have witnessed.

4.1.3 *Effect of Local Search*

As mentioned in Section 3.1.6, local search fulfills two goals: in lower dimensions due to application of the sphere boundary we are able to achieve the reduction of the potential intruding points. This additional pruning ability is not sustainable in higher dimensions due to the "curse of dimensionality" (see Section 4.1.4).

In addition, the underlying k-nearest neighbor query introduces the space-aware metaheuristic which the baseline Sokal lacks: the

retrieval of potential candidates in an ordered fashion (Section 3.1.6) helps the Sokal algorithm to discard potential generating points sooner than otherwise would be possible.

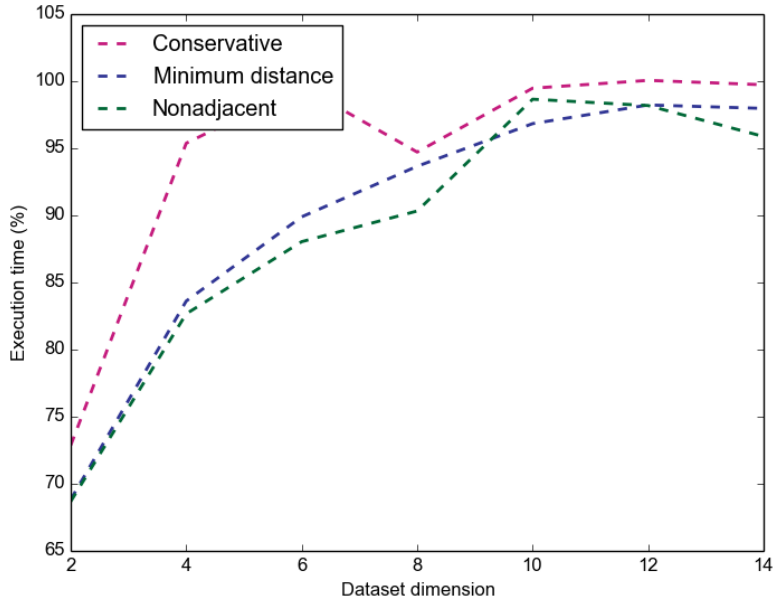

Figure 31.: Relative time performance of Dual-Tree algorithms without local search compared to Sokal in as dataset dimensionality increases.

Table 4 and Figure 31 confirm the dramatic effect that local search has on the performance of the dual-tree algorithm. As can be seen, without local search, even the best case performance of the dual-tree algorithm (in a 2-dimensional setting) is only approximately 33% better than the baseline, which contrasts with the stellar improvement of 95% on a 2d data with local search.

However, apart from the awareness of its significance, we can not trace back which specific property of local search has stronger impact without some further analysis. To do this, let us take a look at two graphs: the worst case computations and the real computations made by the Sokal sub-task of the dual-tree algorithms as a proportion of those made by the pure algorithm.

The solid lines in Figure 32 represent dual tree versions and the dotted lines their correspond to the same algorithm with local search turned off. One critical observation is that at around 6-8 dimensions, both dotted and the solid lines merge. This signifies that the local search ceases making any additional contributions in terms of search

Table 4.: Dataset EEGEye (7200 instances). Effect of dimensionality growth on the performance of Sokal and Dual-Tree algorithms without local search

	Sokal	Conservative		Minimum		Nonadjacent	
Dim.	(s)	(s)	(%)	(s)	(%)	(s)	(%)
2	233	170	72.87	160	68.83	160	68.69
4	359	342	95.38	300	83.62	297	82.67
6	492	490	99.59	442	89.89	433	88.04
8	780	739	94.72	731	93.66	705	90.33
10	1011	1005	99.48	979	96.84	997	98.66
12	1282	1282	100.06	1259	98.23	1258	98.18
14	1291	1288	99.73	1265	97.97	1238	95.88

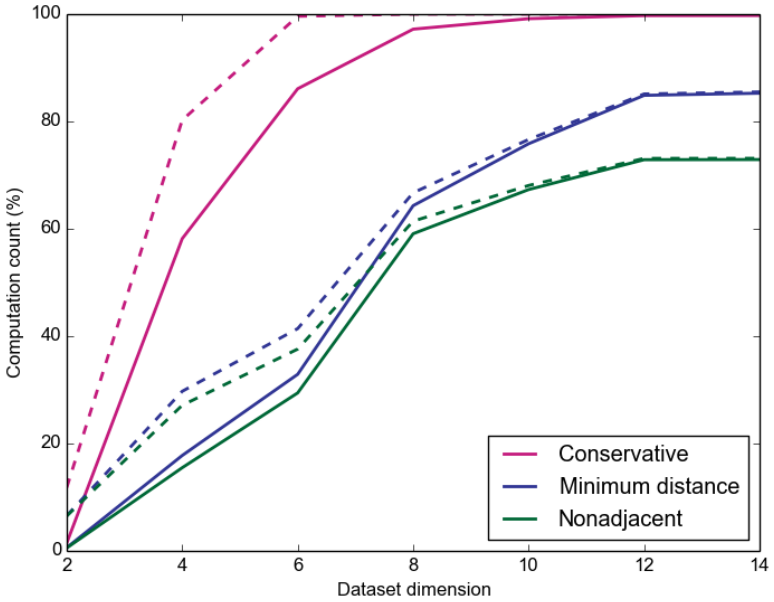

Figure 32.: Worst case computation ratio.

space reduction. Another point, is the relative insignificant contribution of the local search towards improving the worst case scenario. Finally, as the dimensions increase so does the worst case computation count, as the pruning and local search pruning abilities lessen.

However, if we plot the real computation counts as a proportion of the baseline Sokal (Figure 33), a far more divisive picture shapes up. The gap between the computation count achieved by the dual-tree strategies as compared to the Sokal baseline is compellingly large. The difference is distinct from the worst case plots in three points identified before: 1) the divergence between dual-trees with and with-

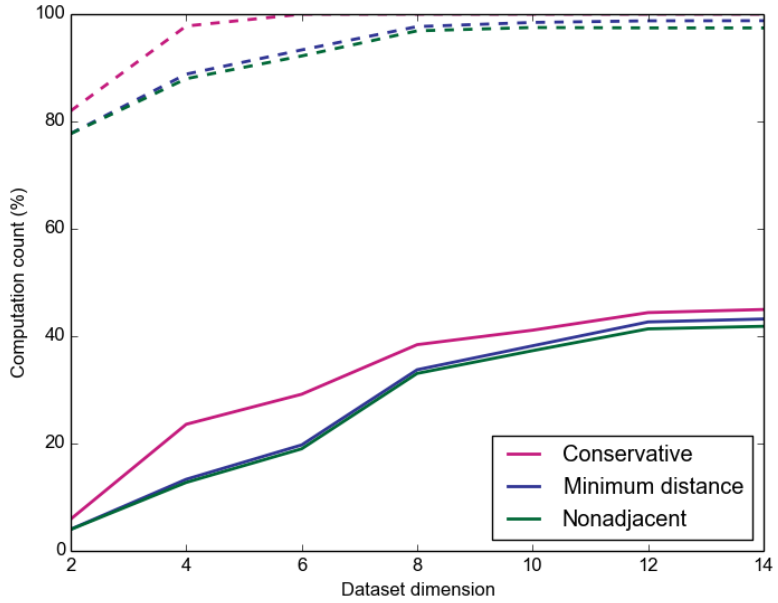

Figure 33.: Real computation ratio

out local search continues even after 6-8 dimensions whereas the corresponding worst cases align, 2) unlike the marginal contribution of the local search on the worst case computations, it is compellingly large in the real case, ca. 60%, 3) the real computations never reach the baseline values.

Our investigative experiments bring us to several conclusions regarding the local search: 1) it is an adequate additional pruning technique in the window of 1 to 6 dimensions, 2) its metaheuristic properties are of prime importance to reducing the runtime computation count of the Sokal subtask in the dual-tree algorithm, 3) this metaheuristic effect is equally potent across all the dataset dimensions, improving on the worst case scenario and outperforming Sokal even in higher dimensions.

4.1.4 *Curse Of Dimensionality*

As the dimensions of the dataset grow, curse of dimensionality [8, 15] becomes more of a problem. It affects our dual-tree algorithm in several ways:

- *Pruning.* The conservative pruning strategy is the first one to be hit, as the k-point-to-node distance becomes much larger than the distances between nodes.

The number of sides of the kd-tree, and consecutively adjacent nodes, becomes exponential to the dimensionality [26]. Therefore even the most aggressive pruning – nonadjacent – becomes of less use as ever less nodes become non-neighboring ones.

- *Local Search.* As the distances in high dimensions become very big, the local search stops fulfilling one of its two properties – additional data reduction mechanism.
- *Nearest-neighbor query.* In high dimensions querying becomes an ever more expensive operation eventually equaling linear search.

Therefore, to stay effective as dimensions grow, the dual tree requires corresponding growth of the dataset size. The exact correlation between dimensionality and desired dataset size is not explored in this work.

4.1.5 *A CBP-Based Classifier*

Experimental results lead us to several observations. Consider Figure 34 that highlights several particularities of the presented strategies.

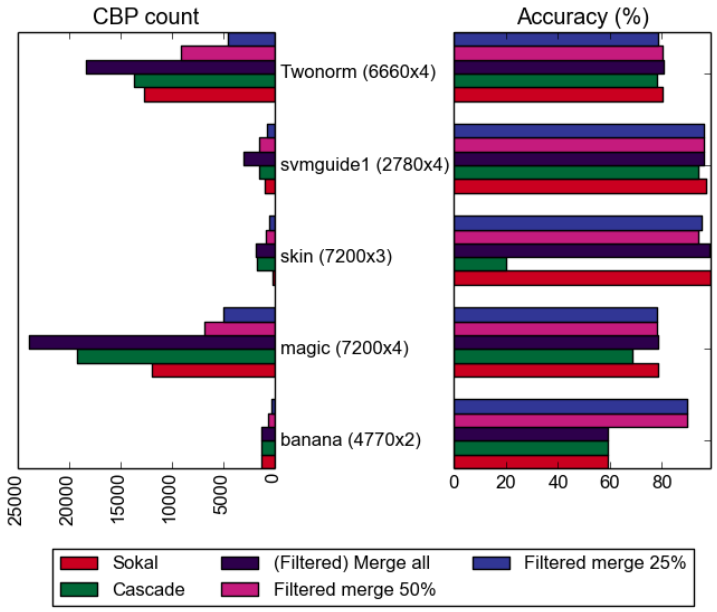

Figure 34.: CBP count vs classifier accuracy on selected datasets

Filtered merge without a threshold value controlling the number of the strong CBPs that should be retained – that is, practically a *full*

merge – runs against the same limitations that the original Boosted OGE does. The number of CBPs in the merged set is too high and hits the storage restrictions, on some datasets it even exceeds the CBP count of the original Boosted OGE. Therefore thresholding is critical. Although the bars of *cascade* strategy are the highest, the values are not generally comparable with the other strategies: they represent the sum of CBPs of all the classifiers and are not merged into one set of CBPs as is the case in other strategies.

Raising the threshold to select only 25% instead of 50% of the strongest CBPs does not compromise the quality of prediction. Thus, we can conclude that even more aggressive thresholding would be possible. Moreover, when applied to one particular dataset – banana (4770x2) – thresholding improves accuracy of prediction as compared to original Boosted OGE or a full merge / cascade strategies, as reducing CBP leads to improved generalization.

Table 5.: Percentage of CBPs that proposed classifiers share with original Boosted OGE

	B. OGE	Cascade		Merge all	
Dataset	(count)	(count)	(%)	(count)	(%)
banana (4770x2)	1266	1266	100.00	1266	100.00
magic (7200x4)	11913	19282	13.32	23918	17.01
skin (7200x3)	204	1727	6.37	1758	12.25
svmguide1 (2780x4)	981	1502	8.66	3035	19.78
Twonorm (6660x4)	12733	13668	5.34	18325	6.91

	B. OGE	F. merge 50%		F. merge 25%	
Dataset	(count)	(count)	(%)	(count)	(%)
banana (4770x2)	1266	633	50.00	316	24.96
magic (7200x4)	11913	6825	4.90	4976	4.49
skin (7200x3)	204	863	3.43	544	3.92
svmguide1 (2780x4)	981	1496	10.81	736	7.03
Twonorm (6660x4)	12733	9071	3.66	4547	2.08

If we analyze how many of the genuine CBPs which Boosted OGE operates on are shared by the proposed classifiers (Table 5), a striking picture appears: in datasets beyond 2 dimensions the proposed classifiers are built mostly on non-genuine CBPs. This seems to have no adverse affect on accuracy. As we have mentioned before, author of Boosted OGE [23] has proposed to relax definition of CBPs in order to increase the space of candidate classifiers. This relaxation is effectively taking place when only a subset of features are considered.

Cascade strategy is very susceptible to the choice of feature subset size. In particular, building a cascade of classifiers based on subsets of features with poor predicting ability produces a majority of unsatisfactory base predictions, which the voting procedure can not improve. The *cascade* predictors are self-contained and restricted to the prediction power that a particular set of features gives them, whereas *filtered merge* benefits from the full multidimensionality.

Table 6.: Classifiers performance based on available features

Dataset	Boosted OGE Acc. (%)	Filtered merge (25%) Acc. (%)	Cascade Acc. (%)
Skin (7200x2)	20.00	20.00	20.00
Skin (7200x3)	98.88	91.38	20.00

To make this claim more convincing, consider Table 6 which shows particularly poor prediction ability (20% accuracy) that *cascade* strategy has on the skin dataset with three dimensions, whereas other *filtered merge* and original Boosted OGE predict with over 90% accuracy. If we, however, only leave the first two dimensions, then the performance of the other methods plunges to the same depth of 20%.

4.2 RESULTS

4.2.1 *CBP Computation*

Table 9 in Appendix A provides detailed summary of the dual-tree performance compared to the baseline Sokal algorithm. We have used Jaccard similarity coefficient to calculate accuracy of the CBPs found by dual trees:

$$\frac{|\{CBP_{Sokal}\} \cap \{CBP_{Dual-Tree}\}|}{|\{CBP_{Sokal}\} \cup \{CBP_{Dual-Tree}\}|} \tag{13}$$

Seeing that most of the accuracy values for all three pruning strategies are around 99%, we can focus on the time performance parameter of the dual trees as illustrated in Figure 35. As could be expected, *nonadjacent pruning* strategy has the best time performance across almost the whole sample of the datasets. That allows us to focus our attention in the further course of results analysis on this top-performer.

Table 7 presents the results of validating the approximate CBPs (found with nonadjacent pruning strategy) based on the Boosted OGE algorithm. Although nonadjacent pruning is only an approximation mechanism to find the CBPs, the classifiers based on the exact CBPs

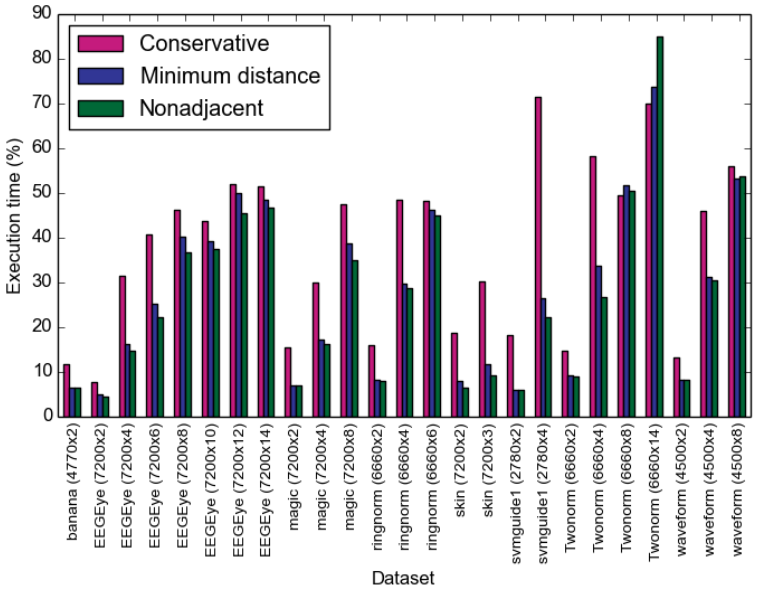

Figure 35.: Time performance of the three dual trees in relation to Sokal

(found by Sokal) and those based on the approximate points render almost identical results in respect to the accuracy percentage. Simultaneously, the dual-tree solution manages to beat the Sokal-based ensemble on each dataset in terms of time: the mere exception is dataset Example (912x5) where due to the poor data-samples-to-feature-count ratio the kd-trees become inefficient. Some of the values are not available (N/A) due to the growing storage cost: as we ascend into higher dimensions, the Boosted OGE algorithm has to create classifiers based on the growing number of points which does not fit into the memory leading to the algorithm failure.

Table 7.: Validation of the CBPs found with dual-tree nonadjacent pruning against baseline Boosted OGE classifier

Dataset	Sokal		Dual Tree (nonadjacent)		
	Time (s)	Acc. (%)	Time (s)	Rel.time (%)	Acc. (%)
banana (4770x2)	73	59.25	9	12.44	59.25
EEGEye (7200x2)	273	54.38	36	13.16	54.38
EEGEye (7200x4)	427	65.63	88	20.60	65.63
EEGEye (7200x6)	603	67.88	179	29.64	67.88
EEGEye (7200x8)	1016	77.25	465	45.75	77.25
EEGEye (7200x10)	1296	78.38	676	52.17	78.38
EEGEye (7200x12)	1503	N/A	792	52.70	N/A

Table 7.: Validation of the CBPs found with dual-tree nonadjacent pruning against baseline Boosted OGE classifier

Dataset	Sokal		Dual Tree (nonadjacent)		
	Time (s)	Acc. (%)	Time (s)	Rel.time (%)	Acc. (%)
EEGEye (7200x14)	1485	N/A	788	53.07	N/A
Example (912x2)	3	62.75	1	19.44	62.75
Example (912x4)	5	88.24	4	76.90	88.24
Example (912x5)	5	100.00	8	157.95	100.00
magic (7200x2)	376	72.50	52	13.95	72.50
magic (7200x4)	668	78.63	130	19.45	78.63
ringnorm (6660x2)	311	73.78	49	15.72	73.78
ringnorm (6660x4)	675	83.24	134	19.81	83.24
ringnorm (6660x6)	823	84.59	420	51.00	84.59
skin (7200x2)	131	20.00	23	17.43	20.00
skin (7200x3)	88	98.88	19	21.05	98.25
svmguide1 (2780x2)	17	94.17	1	8.70	94.17
svmguide1 (2780x4)	23	97.09	12	51.79	97.09
transfusion (673x2)	8	82.67	2	28.95	82.67
transfusion (673x4)	5	82.67	1	27.81	82.67
Twonorm (6660x2)	216	71.62	32	14.86	71.62
Twonorm (6660x4)	437	80.54	171	39.20	80.54
Twonorm (6660x14)	4138	N/A	2346	56.70	N/A
waveform (4500x2)	148	68.40	16	10.66	68.40
waveform (4500x4)	400	67.00	137	34.17	67.00
waveform (4500x8)	672	80.80	406	60.46	80.80

4.2.2 *A CBP-Based Classifier*

Having confirmed comparable accuracy achieved by the dual tree algorithms we would now proceed to the analysis of the alternative ensemble classifiers. The reader can refer to the detailed results in Table 10 of Appendix A whereas this section concentrates on the main aspects.

Among the three proposed classifiers – *merge all*, *filtered merge* (25% and 50%), and *cascade* – only the last two will be extensively analyzed in this section. The reason for this is, as also visible in Table 10, that we can discard two classifiers from the detailed analysis: 1) *full merge* tends to amass disproportionately many CBPs leading to the excessive memory load; 2) as *filtered merge* with thresholds 25% and 50% exhibit comparable performance we will only consider the former pa-

rameterization of the classifier.

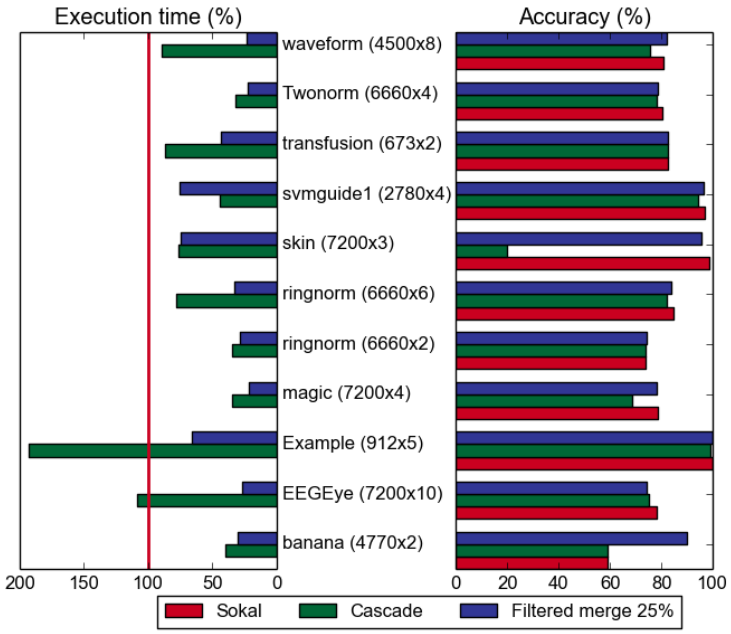

Figure 36.: Comparison of relative computation time with classifier accuracy

Figure 36 puts a dual perspective on the performance of the classifiers: relative computation time compared to Sokal vs classifier accuracy. Instead of presenting the datasets with different feature subsets as was done in 7, Figure 36 selects one particular feature subset of the dataset. Generally, a dataset with most complete feature set was taken except when it was impractical as at least one of the classifiers would collapse due to the storage cost of fitting multitude of resulting CBPs.

We observe that typically both *cascade* and *filtered merge* classifiers outperform Sokal in terms of computational time. A notable exception is dataset Example (912x5) where due to the relatively small size of the dataset, the new classifiers can not keep up with Sokal.

Barring for the datasets skin and banana, both *cascade* and *filtered merge* have commensurate performance with Sokal in terms of accuracy. While *cascade* performance is sensitive to the value of the features used for the subclassifiers, being only 3-dimensional, dataset skin leaves no space for the classifier improvement which could be said to be unsuitable for the particular problem. On the other hand, *filtered merge* displays admirable outstanding accuracy on the 2-d dataset, where both Sokal and *cascade* are about 30% behind.

Table 8.: Ranking of classifiers according to accuracy

Dataset	Sokal	Filtered merge 25%	Cascade	Random Forest
banana (4770x2)	59.25	89.81	59.25	91.32
EEGEye (7200x10)	78.38	74.38	75.00	86.00
Example (912x5)	100.00	100.00	99.02	100.00
magic (7200x4)	78.63	78.38	68.63	78.38
ringnorm (6660x6)	84.59	83.78	82.03	86.89
skin (7200x3)	98.88	95.50	20.00	99.75
svmguide1 (2780x4)	97.09	96.44	94.17	96.76
transfusion (673x2)	82.67	82.67	82.67	78.67
Twonorm (6660x4)	80.54	78.78	78.38	76.08
waveform (4500x8)	80.80	82.00	75.40	77.60
Rank	1.6	2.2	3.1	1.7

Given that Sokal falls behind on the larger datasets in terms of time, we aim the analysis on the accuracy aspect for the datasets that are solvable by Sokal. Table 8 is a ranking of the classifiers. We provide reference values for the results obtained with the random forest classifier in order to have an additional benchmark values. Sokal has a narrow lead over *filtered merge* strategy with *cascade* trailing behind.

It is interesting to observe, that filtered merge, although ranked to be the 3rd if only considering the raw values, has a very close general performance with the random forest classifier, should we allow for a tolerance value of 5% with respect to the accuracy values.

Figure 37 gives another view angle on the classification of the datasets which the Boosted OGE classifier failed to crunch due to the storage capacity required by the growing number of CBPs. The time values are absolute because a baseline measurements are missing. We can, nevertheless, see that the filtered merge outperforms cascade in computation by a wide margin while both classifiers have achieved similar accuracy rates.

Furthermore, the algorithms are able to capitalize on growing number of features and consequently outperform the Boosted OGE values achieved on the lower dimensional datasets. Accordingly, as we go into higher dimensions and the storage cost becomes unbearable for the baseline Boosted OGE, we can resort to the proposed alternative ensemble-based models. Based on the experimental data, the transition from an "integral" to an ensemble-based model incurs no significant penalty in terms of prediction ability of the classifiers.

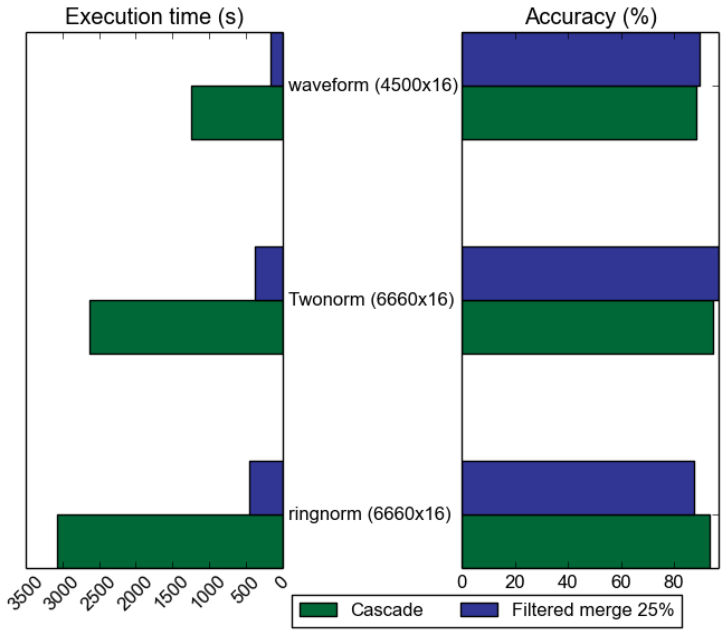

Figure 37.: Comparison of computation time with classifier accuracy for higher dimensions

5

CONCLUSIONS AND FUTURE WORK

CONCLUSIONS. This work has started off with the exploration of the dual-tree framework that was originally devised to bring tractability to the class of generalized n-body problems. We have been able to devise a dual-tree based algorithm to deal with the problem of finding characterizing boundary points (CBP), a special case of the Gabriel graph.

The proposed algorithm was devised in incremental fashion, proposing and evaluating alternative solutions to the dual-tree architecture, pruning rules, and a local search strategy. In this course, we have been able to devise two types of pruning – exact and approximate – offering a trade-off between accuracy of CBP computation and speed gains.

Owing to the powerful metaheuristics implemented by the local search and reduction of the dataset brought about by pruning, the developed dual-tree algorithm has been able to both lower worst-case and real-case computation measurements as compared to the baseline Sokal solution. Based on the experimental evaluation, we have been able to achieve about an order of magnitude performance increases on the two-dimensional datasets. Gains in efficiency were also evident in higher dimensions ranging from 70% to 50% improvement of the state-of-the-art method.

Having built a new algorithm for finding CBPs we have proceeded to Boosted OGE [23], a CBP-based classification algorithm. We have set a goal of improving one of the algorithm's weaknesses – poor scalability in higher dimensions. In order to do that, two alternative models where proposed that integrated original problem-solving approach of the classifier into an ensemble-based framework. Both models – *filtered merge* and *cascade* – were shown to be on par with the original classifier in terms of predictive power on almost all datasets, while outperforming it in computational speed. Above all, however, the two ensemble-based models have been able to take up larger datasets which the original Boosted OGE algorithm fails to tackle due to the high memory cost it requires.

FUTURE WORK. This work has opened up a number of areas that further research should address.

CBP Computation. The proposed dual-tree algorithm is superior to Sokal in performance, but also more complex as it requires parameterizing the node size. Experimentally shown to be an important factor for the dual-tree performance and a non-trivial task, a stricter methodology or a set of general observations should be developed to determine optimal values for the leaf sizes. This would inevitably depend on the implicit dimensionality of the datasets and their sizes and is likely to be specific to the chosen dual-tree architecture.

It has been demonstrated that approximation strategies do not affect the performance of the CBP-based classifier. Therefore, we could hypothesize that even more aggressive pruning rules and restricted local search could bring computational speed dividends without sacrifices of the performance. Having observed the non-linear growth of CBPs attributed to the rise of dimensionality, we can even argue that tightening the number of discovered CBPs may implicitly obtain needed regularization in higher dimensions.

More awareness about the required size of the datasets in respect to their implicit dimension is needed to be able to conclude beforehand about the expected performance gain of the dual-tree algorithms compared to Sokal. Due to the hardware considerations we were restricted in the size of the datasets under test.

CBP-Based Classifier. The two developed ensemble-based models proposed have a comparable performance to the original Boosted OGE classifier. However, inspecting the CBPs that the ensemble learners operate on has shown that they share only a small margin of the "genuine" CBPs. This fact seems to have no adverse effect on the classification, but needs a more rigorous study of its own. Being just a heuristic notion of goodness, CBPs do not necessarily represent the most optimal imaginary classification boundary, whereas the newly found "weak CBPs" could embody a better approximation.

The proposed *filtered merge* has proven to be robust to thresholding as we have reduced the number of points retained from 50% to 25%. Reducing the number of CBPs further will cut down the dimensions of the ensuing weak classifiers, therefore it is certainly worthy to test the limits of thresholding. Furthermore, instead of setting a percentage threshold the *filtered merge* can be set to retain a certain number of CBPs based on the underlying dataset size and dimension.

BIBLIOGRAPHY

[1] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Commun. ACM* 18 (9 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: http://doi.acm.org/10.1145/361002.361007.

[2] Mark de Berg et al. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. ISBN: 3540779736, 9783540779735.

[3] Wikimedia Commons. *A 3-dimensional k-d tree*. File: 3dtree.png. 2015. URL: http://commons.wikimedia.org/wiki/File:3dtree.png.

[4] Wikimedia Commons. *Gabriel graph*. File: Gabriel_graph.svg. 2015. URL: http://commons.wikimedia.org/wiki/File:Gabriel_graph.svg.

[5] Wikimedia Commons. *Nearest neighbor graph*. File: Nearest_neighbor_graph.svg. 2015. URL: http://commons.wikimedia.org/wiki/File:Nearest_neighbor_graph.svg.

[6] Jacob E. Goodman Csaba D. Toth Joseph O'Rourke. *Handbook of Discrete and Computational Geometry, Second Edition*. Apr. 13, 2004.

[7] Ryan R. Curtin et al. "Tree-Independent Dual-Tree Algorithms". In: *CoRR* abs/1304.4327 (2013). URL: http://dblp.uni-trier.de/db/journals/corr/corr1304.html#abs-1304-4327.

[8] Pedro Domingos. "A Few Useful Things to Know About Machine Learning". In: *Commun. ACM* 55.10 (Oct. 2012), pp. 78–87. ISSN: 0001-0782. DOI: 10.1145/2347736.2347755. URL: http://doi.acm.org/10.1145/2347736.2347755.

[9] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. "An Algorithm for Finding Best Matches in Logarithmic Expected Time". In: *ACM Trans. Math. Softw.* 3.3 (Sept. 1977), pp. 209–226. ISSN: 0098-3500. DOI: 10.1145/355744.355745. URL: http://doi.acm.org/10.1145/355744.355745.

[10] K. R. Gabriel and R. R. Sokal. "A new statistical approach to geographic variation analysis". In: *Syst. Zool.* (1969), pp. 259–278.

[11] Alexander Gray and Andrew Moore. "'N-Body' Problems in Statistical Learning". In: *Advances in Neural Information Processing Systems 13*. MIT Press, 2000, pp. 521–527.

[12] Alexander G. Gray. "Bringing Tractability to Generalized N-Body Problems in Statistical and Scientific Computation". PhD thesis. Carnegie Mellon University, 2003.

[13] Alexander G Gray. *Fast N-Body Algorithms for Massive Datasets*. https://www.siam.org/meetings/sdm08/TS3.ppt. (Visited on 04/15/2015).

[14] Alexander G. Gray and Andrew W. Moore. "Nonparametric Density Estimation: Toward Computational Tractability". In: *SDM*. Ed. by Daniel Barbar and Chandrika Kamath. SIAM, Aug. 26, 2003. ISBN: 0-89871-545-8. URL: http://dblp.uni-trier.de/db/conf/sdm/sdm2003.html#GrayM03.

[15] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[16] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 2015-04-06]. 2001. URL: http://www.scipy.org/.

[17] Songrit Maneewongvatana and David M. Mount. "It's Okay to Be Skinny, If Your Friends Are Fat". In: *Center for Geometric Computing 4th Annual Workshop on Computational Geometry*. 1999.

[18] David W. Matula and Robert R. Sokal. "Properties of Gabriel Graphs Relevant to Geographic Variation Research and the Clustering of Points in the Plane". In: *Geographical Analysis* 12.3 (1980), pp. 205–222. ISSN: 1538-4632. DOI: 10.1111/j.1538-4632.1980.tb00031.x. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1538-4632.1980.tb00031.x/abstract.

[19] Andrew Moore and Mary Soon Lee. "Cached Sufficient Statistics for Efficient Machine Learning with Large Datasets". In: *Journal of Artificial Intelligence Research* 8 (1997), pp. 67–91.

[20] Andrew W. Moore. "The Anchors Hierarchy: Using the Triangle Inequality to Survive High Dimensional Data". In: *In Twelfth Conference on Uncertainty in Artificial Intelligence*. AAAI Press, 2000, pp. 397–405.

[21] AndrewW. Moore et al. "Fast Algorithms and Efficient Statistics: N-Point Correlation Functions". English. In: *Mining the Sky*. Ed. by AnthonyJ. Banday, Saleem Zaroubi, and Matthias Bartelmann. ESO ASTROPHYSICS SYMPOSIA. Springer Berlin Heidelberg, 2001, pp. 71–82. ISBN: 978-3-540-42468-0. DOI: 10.1007/10849171_5. URL: http://dx.doi.org/10.1007/10849171_5.

[22] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[23] Oriol Pujol. "Boosted Geometry-Based Ensembles." In: *MCS*. Ed. by Neamat El Gayar, Josef Kittler, and Fabio Roli. Vol. 5997. Lecture Notes in Computer Science. Springer, Apr. 14, 2010, pp. 195–204. ISBN: 978-3-642-12126-5. URL: http://dblp.uni-trier.de/db/conf/mcs/mcs2010.html#Pujol10.

[24] Oriol Pujol and David Masip. "Geometry-Based Ensembles: Toward a Structural Characterization of the Classification Boundary." In: *IEEE Trans. Pattern Anal. Mach. Intell.* 31.6 (2009), pp. 1140–1146. URL: http://dblp.uni-trier.de/db/journals/pami/pami31.html#PujolM09.

[25] Remco C. Veltkamp. *Closed Object Boundaries from Scattered Points*. Nov. 30, 1994.

[26] Eric W. Weisstein. *Hypercube. From MathWorld—A Wolfram Web Resource*. Last visited on 4/4/2015. URL: http://mathworld.wolfram.com/Hypercube.html.

[27] Eric W. Weisstein. *Space diagonal. From MathWorld—A Wolfram Web Resource*. Last visited on 4/4/2015. URL: http://mathworld.wolfram.com/SpaceDiagonal.html.

A

DETAILED RESULTS

Table 9.: Comparison of the dual-tree performance with different strategies to the baseline Sokal

Dataset	Sokal (s)	Dual Tree (strategy)	Time (s)	(%)	Acc. (%)
banana (4770x2)	76	Conservative	9	11.74	100.00
		Minimum	5	6.47	100.00
		Nonadjacent	5	6.62	100.00
EEGEye (7200x2)	251	Conservative	19	7.66	100.00
		Minimum	13	5.07	100.00
		Nonadjacent	11	4.51	99.99
EEGEye (7200x4)	395	Conservative	124	31.46	100.00
		Minimum	64	16.28	100.00
		Nonadjacent	59	14.81	99.96
EEGEye (7200x6)	520	Conservative	212	40.85	100.00
		Minimum	131	25.23	99.99
		Nonadjacent	116	22.36	99.92
EEGEye (7200x8)	828	Conservative	383	46.25	100.00
		Minimum	334	40.32	100.00
		Nonadjacent	305	36.83	99.86
EEGEye (7200x10)	1140	Conservative	498	43.65	100.00
		Minimum	448	39.30	99.99
		Nonadjacent	426	37.39	99.83
EEGEye (7200x12)	1307	Conservative	680	52.05	100.00
		Minimum	654	50.03	99.87
		Nonadjacent	594	45.46	99.46
EEGEye (7200x14)	1357	Conservative	699	51.50	100.00
		Minimum	657	48.39	99.87
		Nonadjacent	636	46.86	99.46
Example (912x2)	3	Conservative	1	25.91	100.00
		Minimum	0	14.86	100.00
		Nonadjacent	0	14.68	100.00
Example (912x4)	4	Conservative	6	166.44	100.00
		Minimum	3	85.87	99.74

Table 9.: Comparison of the dual-tree performance with different strategies to the baseline Sokal

Dataset	Sokal (s)	Dual Tree (strategy)	Time (s)	(%)	Acc. (%)
		Nonadjacent	3	77.01	99.74
Example (912x5)	3	Conservative	6	199.49	100.00
		Minimum	4	138.84	99.78
		Nonadjacent	4	109.96	99.57
magic (7200x2)	203	Conservative	31	15.45	100.00
		Minimum	14	7.11	100.00
		Nonadjacent	14	6.89	100.00
magic (7200x4)	328	Conservative	98	29.98	100.00
		Minimum	57	17.25	100.00
		Nonadjacent	53	16.25	99.96
magic (7200x8)	685	Conservative	326	47.62	100.00
		Minimum	266	38.80	99.93
		Nonadjacent	239	34.90	99.56
ringnorm (6660x2)	182	Conservative	29	15.96	100.00
		Minimum	15	8.16	100.00
		Nonadjacent	14	7.96	100.00
ringnorm (6660x4)	350	Conservative	170	48.62	100.00
		Minimum	104	29.73	100.00
		Nonadjacent	101	28.77	100.00
ringnorm (6660x6)	638	Conservative	307	48.13	100.00
		Minimum	294	46.17	99.98
		Nonadjacent	287	44.96	99.97
skin (7200x2)	92	Conservative	17	18.72	100.00
		Minimum	7	8.12	100.00
		Nonadjacent	6	6.52	99.93
skin (7200x3)	67	Conservative	20	30.25	100.00
		Minimum	8	11.63	97.85
		Nonadjacent	6	9.14	96.77
svmguide1 (2780x2)	9	Conservative	2	18.22	100.00
		Minimum	1	6.11	100.00
		Nonadjacent	1	6.01	100.00
svmguide1 (2780x4)	19	Conservative	14	71.53	100.00
		Minimum	5	26.40	100.00
		Nonadjacent	4	22.33	100.00
transfusion (673x2)	3	Conservative	1	40.32	100.00
		Minimum	1	22.88	95.60
		Nonadjacent	0	17.64	79.69
transfusion (673x4)	2	Conservative	5	244.35	100.00
		Minimum	1	75.63	99.94
		Nonadjacent	1	46.93	95.43

Table 9.: Comparison of the dual-tree performance with different strategies to the baseline Sokal

Dataset	Sokal (s)	Dual Tree (strategy)	Time (s)	(%)	Acc. (%)
Twonorm (6660x2)	177	Conservative	26	4.11	100.00
		Minimum	16	9.19	100.00
		Nonadjacent	16	8.89	100.00
Twonorm (6660x4)	353	Conservative	206	58.33	100.00
		Minimum	119	33.80	100.00
		Nonadjacent	94	26.70	100.00
Twonorm (6660x8)	1026	Conservative	509	49.56	100.00
		Minimum	530	51.66	100.00
		Nonadjacent	519	50.58	100.00
Twonorm (6660x14)	3426	Conservative	2402	70.12	100.00
		Minimum	2524	73.68	100.00
		Nonadjacent	2911	84.96	100.00
waveform (4500x2)	91	Conservative	12	13.17	100.00
		Minimum	8	8.36	100.00
		Nonadjacent	8	8.27	100.00
waveform (4500x4)	195	Conservative	90	46.02	100.00
		Minimum	61	31.15	99.99
		Nonadjacent	60	30.60	100.00
waveform (4500x8)	449	Conservative	252	56.11	100.00
		Minimum	240	53.32	99.99
		Nonadjacent	241	53.68	99.99

Table 10.: Detailed results of alternative Boosted OGE classifiers

	Sokal			Merge all					Filtered merge (25%)					Filtered merge (50%)					Cascade					RF
Dataset	CBP (count)	Time (s)	Acc. (%)	CBP (count)	Time (s)	Time (%)	Acc. (%)	Diff.	CBP (count)	Time (s)	Time (%)	Acc. (%)	Diff.	CBP (count)	Time (s)	Time (%)	Acc. (%)	Diff.	CBP (count)	Time (s)	Time (%)	Acc. (%)	Diff.	Acc. (%)
banana (4770x2)	1266	73	59.25	1266	24	32.61	59.25	0.00	316	22	30.23	89.81	30.57	633	22	30.46	89.62	30.38	1266	29	39.52	59.25	0.00	91.32
EEGEye (7200x2)	9267	273	54.38	9266	69	25.37	54.38	0.00	2316	55	20.09	54.38	0.00	4633	60	22.12	54.38	0.00	9266	108	39.47	54.38	0.00	58.25
EEGEye (7200x4)	11281	427	65.63	52041	258	60.48	66.38	0.75	12088	148	34.60	65.25	-0.38	27379	192	44.94	66.13	0.50	48353	273	64.00	54.38	-11.25	71.00
EEGEye (7200x6)	17442	603	67.88	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	78.38
EEGEye (7200x8)	34883	1016	77.25	102126	252	24.76	N/A	N/A	25700	315	30.96	75.00	-2.25	55474	364	35.79	75.13	-2.13	72917	753	74.11	76.13	-1.13	84.88
EEGEye (7200x10)	49128	1296	78.38	137786	330	25.48	N/A	N/A	32799	345	26.65	74.38	-4.00	72775	303	23.36	N/A	N/A	129660	1407	108.55	75.00	-3.38	84.00
EEGEye (7200x12)	72230	1503	N/A	185964	403	26.81	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	86.00
EEGEye (7200x14)	72290	1485	N/A	169811	450	30.31	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	88.00
Example (912x2)	718	3	62.75	718	2	48.91	62.75	0.00	179	2	44.36	58.82	-3.92	359	2	45.68	57.84	-4.90	718	2	59.10	62.75	0.00	61.76
Example (912x4)	1177	5	88.24	1897	3	64.35	85.29	-2.94	445	3	55.51	71.57	-16.67	890	3	72.02	85.29	-2.94	2290	7	140.71	78.43	-9.80	82.35
Example (912x5)	598	5	100.00	1727	3	60.04	100.00	0.00	590	3	65.69	100.00	0.00	1214	3	67.58	100.00	0.00	2906	9	192.73	99.02	-0.98	100.00
magic (7200x2)	4533	376	72.50	4533	135	35.84	72.50	0.00	1133	111	29.52	71.63	-0.88	2266	86	22.86	72.38	-0.13	4533	144	38.15	72.50	0.00	70.63
magic (7200x4)	11913	668	78.63	23918	211	31.60	78.50	-0.13	4976	142	21.22	78.38	-0.25	6825	144	21.49	78.00	-0.63	19282	228	34.12	68.63	-10.00	78.38
ringnorm (6660x2)	4643	311	73.78	4643	115	36.87	73.78	0.00	1160	87	28.09	74.19	0.41	2321	59	18.97	73.78	0.00	4643	107	34.39	73.78	0.00	71.76
ringnorm (6660x4)	12188	675	83.24	17670	216	31.96	81.76	-1.49	4456	126	18.67	77.84	-5.41	8947	118	17.44	80.00	-3.24	17813	234	34.75	75.95	-7.30	81.76
ringnorm (6660x6)	29529	823	84.59	31158	247	29.97	82.57	-2.03	7838	267	32.46	83.78	-0.81	15529	207	25.18	85.14	0.54	42198	643	78.07	82.03	-2.57	86.89
skin (7200x2)	1304	131	20.00	1304	61	46.28	20.00	0.00	326	51	38.68	20.00	0.00	652	59	45.32	20.00	0.00	1304	61	46.61	20.00	0.00	97.00
skin (7200x3)	204	88	98.88	1758	76	86.30	98.25	-0.63	544	66	74.45	95.50	-3.38	863	79	88.84	94.25	-4.63	1727	68	76.52	20.00	-78.88	99.75
svmguide1 (2780x2)	258	17	94.17	258	7	41.47	94.17	0.00	64	5	26.94	95.47	1.29	129	4	21.86	95.15	0.97	258	4	24.59	94.17	0.00	96.44
svmguide1 (2780x4)	981	23	97.09	3035	20	87.16	96.12	-0.97	736	17	75.60	96.44	-0.65	1496	18	79.85	96.44	-0.65	1502	10	43.75	94.17	-2.91	96.76
transfusion (673x2)	4239	8	82.67	3603	4	56.42	82.67	0.00	900	3	43.49	82.67	0.00	1801	4	46.68	82.67	0.00	3603	7	86.85	82.67	0.00	78.67
transfusion (673x4)	1784	5	82.67	13241	18	354.61	82.67	0.00	3310	35	696.07	82.67	0.00	6238	21	421.72	82.67	0.00	12476	25	494.02	82.67	0.00	81.33
Twonorm (6660x2)	4650	216	71.62	4650	72	33.40	71.62	0.00	1162	60	27.67	72.30	0.68	2325	63	28.95	72.03	0.41	4650	88	40.66	71.62	0.00	68.51
Twonorm (6660x4)	12733	437	80.54	18325	127	29.13	80.68	0.14	4547	99	22.68	78.78	-1.76	9071	111	25.48	80.41	-0.14	13668	141	32.29	78.38	-2.16	76.08
Twonorm (6660x14)	304930	4138	N/A	67934	352	8.51	N/A	N/A	15915	319	7.71	95.54	N/A	33415	362	8.75	95.81	N/A	174843	1923	46.47	93.65	N/A	91.22
waveform (4500x2)	4123	148	68.40	4123	36	24.19	68.40	0.00	1030	35	23.30	68.40	0.00	2061	33	22.00	68.40	0.00	4123	77	51.72	68.40	0.00	63.40
waveform (4500x4)	12748	400	67.00	7691	74	18.44	68.60	1.60	3839	71	17.77	66.40	-0.60	5688	73	18.39	65.80	-1.20	7904	103	25.80	68.40	1.40	62.60
waveform (4500x8)	48854	672	80.80	22826	173	25.67	80.40	-0.40	6115	157	23.39	82.00	1.20	11475	131	19.42	78.20	-2.60	75037	603	89.67	75.40	-5.40	77.60

Table 11.: Datasets with sources

Dataset	Size	Dim	Source
banana	5300	2	http://mldata.org/repository/data/viewslug/banana-ida/
EEGEye	14980	14	https://archive.ics.uci.edu/ml/datasets/EEG+Eye+State
Example	1014	5	http://www.maia.ub.es/~oriol/Personal/downloads.html
magic	19020	10	https://archive.ics.uci.edu/ml/datasets/MAGIC+Gamma+Telescope
ringnorm	7400	20	http://mldata.org/repository/data/viewslug/ringnorm-ida/
skin	245057	3	http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html
svmguide1	3089	4	http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html
transfusion	748	4	https://archive.ics.uci.edu/ml/datasets/Blood+Transfusion+Service+Center
Twonorm	7400	20	http://mldata.org/repository/data/viewslug/twonorm-ida/
waveform	5000	21	http://mldata.org/repository/data/viewslug/waveform-ida/