The School of Mathematics

THE UNIVERSITY
of EDINBURGH

Minimum Enclosing Balls with Outliers

by

Thomas Holmes

Dissertation Presented for the Degree of
MSc in Operational Research with Computational Optimization

August 2021

Supervised by
Dr E. Alper Yıldırım

For Mum

Abstract

The minimum enclosing ball with outliers problem is that of finding the ball with the smallest radius that covers at least $\eta\%$ of n many points in $\mathbb{R}^d$. Solving this model optimally is incredibly computationally difficult and in fact this problem is known to be NP-hard, and as such in this paper we investigate heuristic and approximate methods which aim to provide a feasible solution to this problem in a reasonable time, as well as develop improvement heuristics which improve an existing feasible but sub-optimal solution. To ascertain the effectiveness of each method, we run computational experiments on a variety of randomly generated data while varying each parameter of the problem. We also test the application of this problem to finding a binary classifier which we use as an outlier recognition model on the MNIST data set of hand-written digits, comparing the performance of balls found by our methods with a previously known construction method, as well as with some other baseline algorithms commonly used for outlier recognition. An implementation of our work is publicly available on GitHub.

Acknowledgments

My biggest thanks to my supervisor, Dr. E. Alper Yıldırım for his invaluable guidance during this dissertation. Without him, this process would not have gone so smoothly and the work inside would look very different. I would also like to thank Toby Yıldırım.

Thank you to my family and all of my friends, both in Edinburgh and elsewhere, for their support over this very strange, challenging, but ultimately rewarding year. Thank you, Kevin and Benjamin, may our EP go platinum.

Own Work Declaration

I declare that this dissertation was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(Thomas Holmes)

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

The minimum enclosing ball with outliers problem is one that may be stated simply: given a set of numerical data, find the smallest ball which contains some percentage of that data (see Figure 1.1.1). Like many problems within mathematics, however, a simple problem statement does not necessarily imply the existence of a simple solution. The minimum enclosing ball with outliers problem is an extension of the well-known minimum enclosing ball problem within computational geometric optimization, which is to totally enclose a set of data within a ball of minimum radius, and this is a well-studied problem with known methods to solve it quickly and efficiently. The difficulty when extending to the problem of that with outliers is in the choice of which points to exclude as outliers, which depending on the number of points to exclude leads to a combinatorial explosion of individual minimum enclosing ball problems. Indeed, this problem is NP-hard, and it is known that no fully polynomial-time approximation scheme exists in Euclidean space unless P = NP [30].

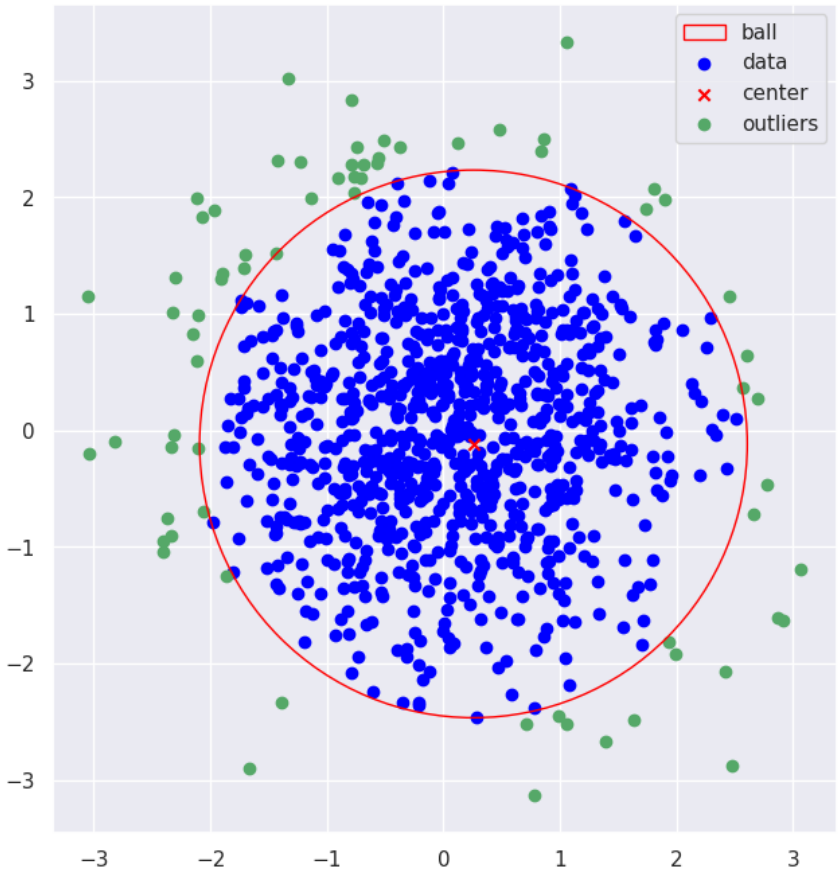


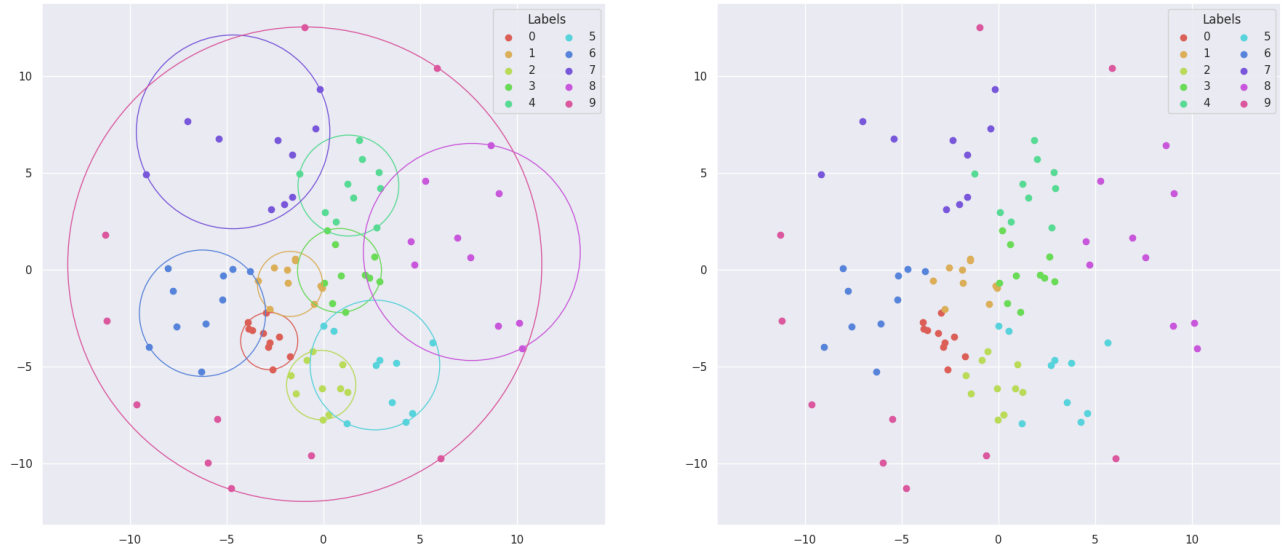
Figure 1.1.1: Example of a MEBwO

One may ask why should we concern ourselves with the minimum enclosing ball with outliers problem, considering the significant increase in difficulty when compared to the minimum enclosing ball problem. The reason is in the name of the problem: outliers can have a significant negative effect on the optimal solution of the minimum enclosing ball problem, which reduces the effectiveness of a minimum enclosing ball in applications. Outliers can come from many sources, for example by human error in data entry or by measurement error due to a faulty sensor. The minimum enclosing ball with outliers problem is able to identify these outliers and return a more robust ball whose solution is closer to that of the "true" data without outliers.

There are many applications of the minimum enclosing ball with outliers problem. In two or three dimensions there are some obvious spatial applications, for example finding the optimal location of a service facility such as a shop such that the maximum distance from homes/businesses to the shop is minimized. A single building which is significantly further from the main cluster of buildings could have a significant effect on the location of the shop to the detriment of the majority of customers, so a minimum enclosing ball with outliers applied to this problem would be able to prioritize this majority. Beyond three dimensions, we often turn to problems within data science and specifically machine learning.

A ball may be used as a binary classification model, where points are labelled based on whether they are inside or outside the ball. This type of model is easy to interpret and requires very little information to construct it, only a radius and a d-dimensional vector. Ball classifiers are similar in concept to support-vector machines [7], which aim to linearly divide a space into two regions and use this rule to determine labels, similar to how a ball may divide a space into two regions. A ball classifier may be used as an outlier recognition model, where points which lie outside the ball are considered to be outliers, and we will benchmark this application of the minimum enclosing ball with outliers problem in Section 5.4.

Another application of the minimum enclosing ball with outliers problem is to clustering, where if we have n data points we may find the minimum enclosing ball with outliers which contains k many points, then removing these points from the data set and repeating this process until all data is enclosed within a ball and labelled. An example of this application can be seen in Figure 1.1.2, though in this paper we do not investigate the effectiveness of this method compared to other known clustering algorithms.



(a) With Balls

(b) Without Balls

Figure 1.1.2: MEBwO Clustering Example on Normal Data from $N(0, 5)$

Fortunately, there are many heuristic and approximation methods which attempt to solve the minimum enclosing ball with outliers problem in a reasonable time. Many methods are already known in the literature, and in this paper we will investigate one of these known methods and propose three

of our own of varying complexity. An implementation of our work is available on GitHub[1] and we discuss this implementation further in Section 4.1.2.

1.2 Outline

In Chapter 2 we make some important definitions as well as discuss a few theorems and propositions. We explicitly define the minimum enclosing ball and minimum enclosing ball with outliers problems, as well as discuss the implications of the big-M parameter within the optimization model formulation of the minimum enclosing ball with outliers problem. We close this chapter with a literature review of the historical and current progress on this problem.

Chapter 3 is concerned with the statement and derivation of the heuristic and approximation methods we discuss within this paper, as well as their expected time complexities. First we present the construction methods which find a feasible solution to the minimum enclosing ball with outliers problem, and then we have the improvement heuristics which aim to improve an existing feasible solution.

We discuss our implementation of the algorithms and software used in this paper in Section 4, and we also present the data we use to solve the minimum enclosing ball with outliers problem for. We use a variety of randomly generated data sets as well as the widely known MNIST data set of hand-written digits [21].

In Chapter 5 we benchmark both our construction methods and our improvement heuristics, the former by finding feasible solutions to the minimum enclosing ball with outliers problem and recording the runtime and feasible solution, and the latter by recording the improvement made to an initial feasible solution. We also investigate an application of our construction methods as an outlier recognition model using the MNIST data set.

Finally, in Chapter 6 we make our conclusions and provide recommendations supported by our benchmarks in Chapter 5. We also discuss some directions for further research.

[1] `https://github.com/tomholmes19/Minimum-Enclosing-Balls-with-Outliers` [15]

Chapter 2

Problem Definition and Literature Review

2.1 Preliminaries

In this paper, we shall denote our data set of finite vectors as $\mathcal{A} = \{a^1, \ldots, a^n\} \subseteq \mathbb{R}^d$ for $n, d \in \mathbb{N}$. The centre of a ball is represented by a vector $c \in \mathbb{R}^d$, the radius a scalar $r \in \mathbb{R}$, and the squared radius $\gamma = r^2$. We denote the percentage of inliers for a minimum enclosing ball with outliers (MEBwO) by $\eta \in (0, 1]$, i.e. if $\eta = 0.9$ then we seek a ball which covers 90% of our data. We will write $\eta\%$ to mean 90% rather than 0.9% in this case. While it may be easier for us to think practically of the MEBwO as covering $\eta\%$ of our data, it is easier mathematically to consider a MEBwO which covers $n - k$ out of n data points, where $k = \lfloor n(1 - \eta) \rfloor$ is the number of outliers.

We would now like to make some formal definitions.

Definition 2.1.1. Let $c \in \mathbb{R}^n$ and $r \in \mathbb{R}$. Then the *ball* with centre c and radius r is the set

$$B(c, r) = \{x \in \mathbb{R}^n : \|x - c\| \leq r\}$$

where $\|\cdot\| : L \to \mathbb{R}$ denotes the standard Euclidean norm on a vector space L (in this paper, $L = \mathbb{R}^n$).

Definition 2.1.2. The *minimum enclosing ball* of $\mathcal{A}$, denoted by $\text{MEB}(\mathcal{A})$, is the ball $B(c^*, r^*)$ where $\mathcal{A} \subseteq B(c^*, r^*)$ and if any $B(c, r)$ exists such that $\mathcal{A} \subseteq B(c, r)$ then $r^* \leq r$.

Theorem 2.1.1. For a given set $\mathcal{A}$, $\text{MEB}(\mathcal{A})$ exists and is unique.

Proof. See [40, page 5]. □

Now, we are interested in the idea that adding or removing data from a data set may have an effect on the radius of the MEB of that data set.

Proposition 2.1.1. Let $a' \in \mathbb{R}^n$ where $a' \notin \mathcal{A}$. Suppose $B(c, r) = \text{MEB}(\mathcal{A})$ and $B(c', r') = \text{MEB}(\mathcal{A} \cap \{a'\})$. Then $r \leq r'$.

Proof. Clearly we have $\mathcal{A} \subseteq \mathcal{A} \cap \{a'\}$, then since $\mathcal{A} \cap \{a'\} \subseteq B(c', r')$ we have $\mathcal{A} \subseteq B(c', r')$ by transitivity. Thus $r \leq r'$ with equality only if $a' \in B(c, r)$ by uniqueness. □

Proposition 2.1.2. Suppose $a' \in \mathcal{A}$. Let $B(c, r) = \text{MEB}(\mathcal{A})$, $B(c', r') = \text{MEB}(\mathcal{A} \setminus \{a'\})$. Then $r' \leq r$.

Proof. Clearly we have $\mathcal{A} \setminus \{a'\} \subseteq \mathcal{A}$ then since $\mathcal{A} \setminus \{a'\} \subseteq B(c', r')$ and $\mathcal{A} \setminus \{a'\} \subseteq B(c, r)$ we have $r' \leq r$ since $B(c', r')$ is the MEB for $\mathcal{A} \setminus \{a'\}$, with equality only if $\text{MEB}(\mathcal{A} \setminus \{a'\} = \text{MEB}(\mathcal{A})$ by uniqueness. □

These two propositions tell us that when we add data to a set, the resulting MEB is either unchanged or bigger. Conversely, when we remove data from a set, the resulting MEB is either unchanged or smaller.

In the interest of contextualizing Algorithm 1, we define the $(1 + \epsilon)$-approximation to an MEB and core-sets.

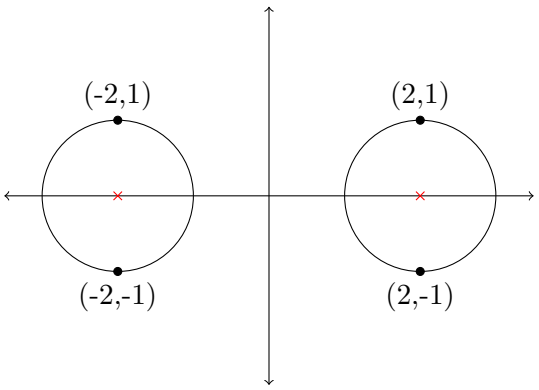


Figure 2.1.1: Example of MEBwO non-uniqueness

Definition 2.1.3 ([19, page 2])**.** Let $\epsilon > 0$. Let r^* be the radius of MEB($\mathcal{A}$). A ball $B(c, (1+\epsilon)r)$ is a $(1+\epsilon)$-*approximation* of MEB($\mathcal{A}$) if $r \leq r^*$ and $\mathcal{A} \subseteq B(c, (1+\epsilon)r)$.

Definition 2.1.4 ([19, page 2])**.** Let $\epsilon > 0$. A subset $X \subseteq \mathcal{A}$ is a ϵ-*core-set* (typically just referred to as a *core-set*) of $\mathcal{A}$ if $\mathcal{A} \subset B(c, (1+\epsilon)r)$.

Finally, the following two definitions allow us to formally define the MEBwO.

Definition 2.1.5. For a set $\mathcal{A}$, the set of k-subsets of $\mathcal{A}$ is the set $\mathfrak{K} = \{\mathcal{K} \in \mathscr{P}(\mathcal{A}) : |\mathcal{K}| = k\}$ where $\mathscr{P}(\mathcal{A})$ is the usual power set of $\mathcal{A}$.

Definition 2.1.6. For some $\eta \in [0, 1]$, let $k = \lfloor n(1-\eta) \rfloor$ be the number of outliers in $\mathcal{A}$. Then the *minimum enclosing ball with outliers* of $\mathcal{A}$, denoted by MEBwO($\mathcal{A}, \eta$), which covers $\eta\%$ of $\mathcal{A}$, is the ball $B(c^*, r^*) = \text{MEB}(\mathcal{K}^*)$ for $\mathcal{K}^* \in \mathfrak{K}$ a k-subset of $\mathcal{A}$ where for all other k-subsets $\mathcal{K} \in \mathfrak{K}$, $\mathcal{K} \neq \mathcal{K}^*$, if $B(c, r) = \text{MEB}(\mathcal{K})$ then $r^* \leq r$.

We may deduce that for a given data set $\mathcal{A}$, an MEBwO exists since an MEBwO is simply the MEB of some k-subset of $\mathcal{A}$, which by Theorem 2.1.1 exists and is unique for that k-subset. However, MEBwO's are not unique, as we can have two distinct k-subsets of $\mathcal{A}$ which have MEB's of the same radius.

For a simple example, consider the data set $\mathcal{A} = \{(-2,-1), (2,-1), (2,1), (-2,1)\}$ with $\eta = \frac{1}{2}$. Then $B((-2,0),1)$ and $B((2,0),1)$ are both MEBwO's for $\mathcal{A}$, and MEB's for the 2-subsets $\{(-2,-1),(-2,1)\}$, $\{(2,-1),(2,1)\}$ of $\mathcal{A}$ respectively. See Figure 2.1.1 for a visualisation of this example.

2.2 Minimum Enclosing Ball

Definition 2.2.1. The optimization model formulation of the Minimum Enclosing Ball (MEB) problem is as follows:

$$\begin{aligned} \min_{c,r} \quad & r \\ \text{s.t.} \quad & \left\|c - a^i\right\| \leq r, \quad i = 1, \ldots, n. \end{aligned}$$

where $c \in \mathbb{R}^d$ and $r \in \mathbb{R}$ are the decision variables corresponding to the centre and radius of the ball respectively.

We can formulate the MEB problem as a second-order cone program (SOCP) by considering the constraints as second-order cone constraints, i.e.

$$C_i = \left\{(r, c - a^i) \in \mathbb{R}^{d+1} : \left\|c - a^i\right\| \leq r\right\}$$

As detailed in [40], the MEB problem can be transformed to a convex quadratically constrained program (QCQP) by squaring the constraints and defining a new decision variable $\gamma = r^2$:

$$\begin{aligned} \min_{c,\gamma} \quad & \gamma \\ \text{s.t.} \quad & \left(a^i\right)^T a^i - 2\left(a^i\right)^T c + c^T c \leq \gamma, \quad i = 1, \ldots, n. \end{aligned}$$

Since this problem can be modelled as a SOCP we know it may be solved by interior-point methods [2]. This problem has $d+1$ variables and n constraints, so it quickly becomes slow to find an optimal solution for large n and d when using a solver such as Gurobi [11], but many alternative approaches are present in the literature which offer very fast solutions within a guaranteed accuracy ([19, 40]).

We will discuss here one such algorithm to solve the MEB problem from [19] that will be used in algorithms in Chapter 3. The central idea is that of the *core-set* [5], which is a small set of points that approximate the shape of a larger set of points. For example, a circle in two dimensions can be represented by a core-set of three points which lie on the boundary of the circle (see Figure 2.2.1).

The idea of the algorithm is to create a candidate core-set, check if the MEB of this core-set contains all the input data, and if so return the MEB, if not grow the core-set by adding the furthest point from the centre of the candidate MEB. For further details, please see [19].

Algorithm 1: Core-Set Algorithm for the MEB Problem [19]

Input: Data set $\mathcal{A} = \left\{a^1, \ldots, a^n\right\}$, error tolerance $\epsilon > 0$
Output: $(1+\epsilon)$-approximation to MEB($\mathcal{A}$) and an $O(1/\epsilon)$-size core-set

Let p be any point in $\mathcal{A}$ (can be chosen randomly);
$q = \arg\max_{a \in \mathcal{A}} \|p - a\|$;
$q' = \arg\max_{a \in \mathcal{A}} \|q - a\|$;
$X := \{q, q'\}$;
$\delta := \epsilon^2/163$;
while *True* **do**
  Let $B(c', r')$ denote the $(1+\delta)$-approximation to MEB(X) returned by solver;
  if $\mathcal{A} \subseteq B(c', (1+\epsilon/2)r')$ **then**
    break;
  else
    $p := \arg\max_{a \in \mathcal{A}} \|a - X\|$
  end
  $X := X \cup \{a\}$
end
return $B(c', (1+\epsilon/2))$, X

Algorithm 1 runs in $O\left(\frac{nd}{\epsilon} + \frac{d^2}{\epsilon^{3/2}}\left(\frac{1}{\epsilon} + d\right)\log\frac{1}{\epsilon}\right)$ time [19, Page 6]. Whenever we use this algorithm, we will set $\epsilon = 1 \times 10^{-4}$.

2.3 Minimum Enclosing Ball with Outliers

2.3.1 Formulation

Definition 2.3.1. We may extend Definition 2.2.1 to the Minimum Enclosing Ball with Outliers (MEBwO) problem as follows:

$$\begin{aligned} \min_{c,r,\xi} \quad & r \\ \text{s.t.} \quad & \left\|c - a^i\right\| \leq r + M\xi_i, \quad i = 1, \ldots, n, \\ & \sum_{i=1}^{n} \xi_i \leq k, \\ & \xi_i \in \{0, 1\}, \qquad i = 1, \ldots, n. \end{aligned}$$

where ξ_i are binary variables corresponding to the distance constraint on each variable, with $M \in \mathbb{R}$ a sufficiently large scalar. One can interpret this as, if $\xi_i = 1$ for some $i \in \{1, \ldots, n\}$, then a_i does not need to be inside the ball. The constraint $\sum_{i=1}^{n} \xi_i \leq k$ where $k = \lfloor n(1-\eta) \rfloor$ ensures that $\eta\%$ of the data is covered.

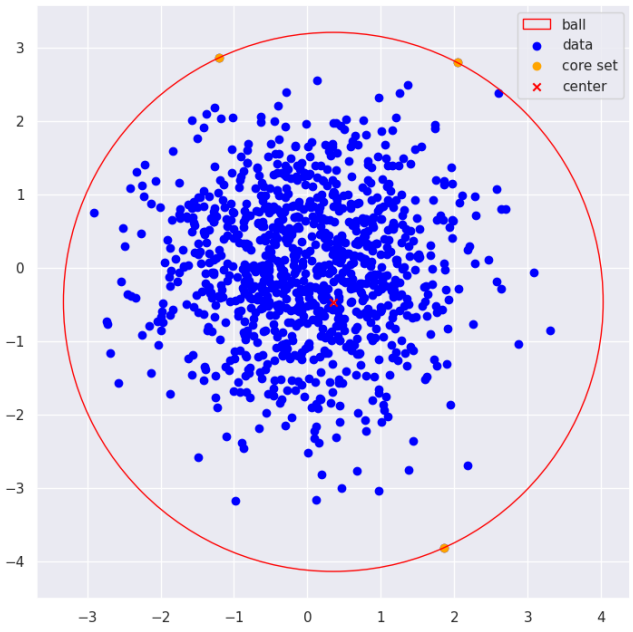


Figure 2.2.1: Example of a Core-Set for an MEB

We can instead extend the QCQP in Definition 2.2.1 to get the following mixed-integer QCQP (MIQCQP) formulation:

$$\begin{aligned} \min_{c,\gamma,\xi} \quad & \gamma \\ \text{s.t.} \quad & \left(a^i\right)^T a^i - 2\left(a^i\right)^T c + c^T c \leq \gamma + M\xi_i, \quad i = 1, \ldots, n, \\ & \sum_{i=1}^{n} \xi_i \leq k, \\ & \xi_i \in \{0, 1\}, \qquad\qquad i = 1, \ldots, n. \end{aligned}$$

Remark. One may note that when we relax ξ_i to be continuous variables, i.e. $0 \leq \xi_i \leq 1$ for $i = 1, \ldots, n$, we have again a convex QCQP since we have then added only continuous variables and linear constraints to the base QCQP.

This model gives us a way to solve the MEBwO problem optimally for a given data set. However, by examining the structure of this model, we can expect the solution times to be unreasonable for any meaningfully large instance. Note that, of n many binary variables ξ_i, we can choose up to $k \leq n$ many of them to have a value of 1, though by Proposition 2.1.2 we know that removing data gives us an MEB with potentially lower radius, meaning we will always pick the highest possible number of points (k many) to be excluded. Thus, by a total brute force search, we may expect to explore $\binom{n}{k}$ many individual MEB problems. Modern optimization solvers will work more efficiently than this, utilizing techniques such as branch-and-bound. Regardless, we expect a very large solution space which leads to exponentially larger solution times as we will see in Section 5.1.

2.3.2 On the Big-M Parameter

The "big-M constraint" in Definition 2.3.1 is a commonly known technique within Operational Research, and an important question is that of what M do we choose? A sufficiently large M is required such that when the corresponding binary variable is equal to 1, the constraint is effectively nullified. But, as we will see within this section, a value of M which is too large will lead to a longer solution time with worse solutions.

An Upper Bound on M

Finding a suitable value of M depends heavily on the nature of the problem the constraint is being applied to. For the minimum enclosing ball with outliers problem, where we are concerned with

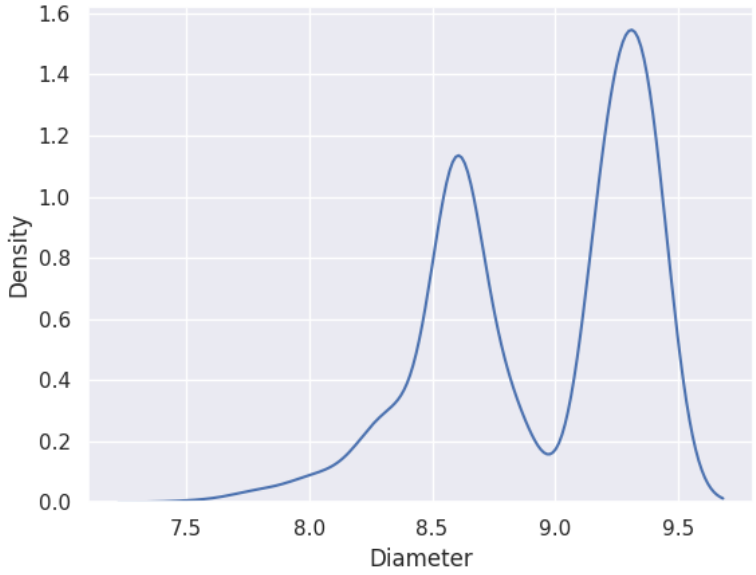


Figure 2.3.1: Density Plot of Approximate Diameters from Algorithm 2

the distances $\|c - a^i\|$ being less than the radius r, we look to add a value to r such that for any reasonable c, this constraint is always satisfied. Thus, a candidate upper bound for M may be found by calculating each pairwise distance within the data and recording the largest distance. Formally, this may be written as $M := \max\{\|a^i - a^j\| : i, j = 1, \ldots, n, i \neq j\}$.

An immediately apparent issue with this approach is that computing M in this way will run in $O(n^2 d)$ time. For smaller data sets this is manageable, but for any significantly sized data sets the time taken to compute M is often longer than the time taken to solve the MEBwO problem on the same data.

We can look to [19, Lemma 1] for a $1/\sqrt{3}$-approximation to the diameter of $\mathcal{A}$ which runs in $O(nd)$ time, by picking a random point $p \in \mathcal{A}$, finding q the furthest point in $\mathcal{A}$ from p, finding q' the furthest point in $\mathcal{A}$ from q, then setting $D := \|q - q'\|$ which is our $1/\sqrt{3}$-approximation to the diameter of $\mathcal{A}$. For a candidate M value we can simply set $M := \sqrt{3}D$.

Algorithm 2: Diameter Approximation Algorithm [19, Lemma 1]

Input: Data set $\mathcal{A}$, point $p \in \mathcal{A}$
Output: Approximate diameter $D \in \mathbb{R}$
$q := \arg\max_{a \in \mathcal{A}} \|p - a\|$;
$q' := \arg\max_{a \in \mathcal{A}} \|q - a\|$;
$D := \|q - q'\|$;
return D;

A caveat to this approach is that the value of M will change depending on the initial random point chosen, and can vary significantly. To demonstrate this, we generate a standard normal data set $\mathcal{A}$ with $n = 1000$, $d = 10$ (see Section 4.2), and by running Algorithm 2 for each $a \in \mathcal{A}$ we construct a distribution of approximate diameters. From `src/diameter_approx_analysis.py` we find an average diameter of 8.939 with variance 0.170 and a density plot is shown in Figure 2.3.1.

For more robust benchmarking, we would prefer a method which always returns the same M value for a given data set, so this method is unsuitable for our purposes. This is because, as seen in the previous section, a higher value of M has a negative effect on the run time of the solver, so we would prefer to minimise the effect of randomness on our benchmarks.

Our chosen method, which gives reliable estimates for M in a reasonable time, is to simply solve the MEB problem for $\mathcal{A}$ and set $M := 2r$ for $B(c, r) = \text{MEB}(\mathcal{A})$. We know from Theorem 2.1.1 that an MEB always exists and is unique for a given data set, so can always expect to receive the same M value from this method. Using a heuristic approach such as Algorithm 1 gives us a $(1 + \epsilon)$-

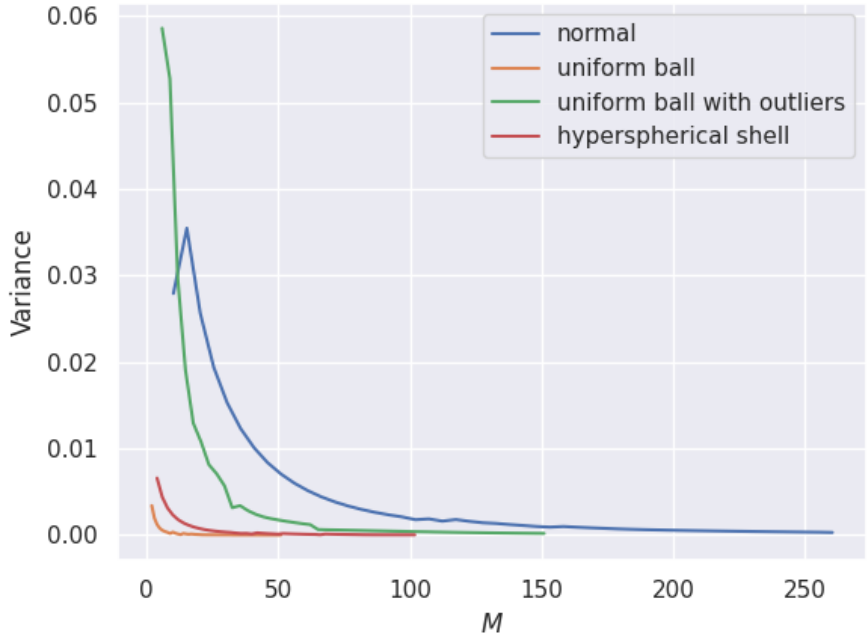


Figure 2.3.2: Variance of Relaxed ξ_i Variables for a Sequence of M Values.

approximation to MEB($\mathcal{A}$), so we will receive an M value that is slightly larger than needed, but the effect on run time is insignificant and given that the returned M value is consistent we do not introduce unneeded randomness into our methodology that is not already inherently present due to using randomly generated data.

Solution Times

One concern with our choice of M is the effect on the solution time of the solver. Typically, a larger than necessary value of M will result in longer solution times, and so this is a concern given that our heuristics are intended to return a feasible solution to the MEBwO problem in reasonably fast time. In our benchmarking of solving the exact model, we observe a significant increase in solution time as the value of M increases when solving the exact model on the same data set (Figure 5.1.1d).

Quality of ξ_i Solutions in the QCQP Relaxation

Another concern regarding the value of M for the exact model is the effect on the optimal solutions for the relaxed ξ_i variables in the QCQP relaxation. This is important as good quality relaxed solutions are essential to Algorithm 5, which uses high-valued ξ_i as a proxy for how far a data point lies from the optimal centre, and so for how much of an outlier that point is.

We can evaluate the quality and diversity of relaxed solutions by solving the relaxed model on the same data set with different values of M. We generate random data sets (see Section 4.2) with $n = 1000$, $d = 10$, and the standard parameters described in Section 4.2, then calculate a lower bound for M^* using Algorithm 2 for each data set, then solve the model for a sequence of multiples of M^* and record the variance of the list of relaxed solutions. We choose to test 50 different values of M in the sequence $\langle M^*(1 + k/2)\rangle_{k=0,\ldots,49}$. The implementation can be found in `src/xi_analysis.py` (see Section 4.1.2) and a plot of the variances can be seen in Figure 2.3.2.

We can see in Figure 2.3.2 that, for each data set, the variance of the relaxed solutions for ξ_i decreases sharply as M increases and tends to zero. Interestingly, there is actually a slight increase in variance at first for the normal data, but like the rest this does still tend to zero.

2.4 Literature Review

We will now conduct a review of the current literature on the MEBwO problem.

Early approaches were developed specifically for the $d = 2$ case of finding the minimum enclosing circle with outliers. One such paper is [1] which discusses computing Voronoi cells [35] and then performing a search of these cells. [10] makes use of a parametric search technique proposed by [24], and [23] uses a randomized search algorithm. These papers all gave exact solutions to the MEBwO problem in the two-dimensional case, while [12] presents some approximate algorithms.

Some more recent approaches are as follows. [30] proves the NP-hardness of the MEBwO problem in Euclidean space (Theorem 1) and also that no fully polynomial time approximation scheme exists for the MEBwO problem unless P = NP (Theorem 3). Shenmaier's paper also proposes two algorithms for solving the MEBwO problem in $\mathbb{R}^d$ which both have an approximation guarantee. We implement and benchmark Algorithm 1 in this paper, which Shenmaier proves to give 2-approximate solutions, meaning we may get a lower bound on the optimal value of the MEBwO problem for a given data set. This algorithm works by computing the k-nearest points to each point in the data set and returning the smallest ball found. The second algorithm, not implemented in this paper, is an extension of the prior algorithm where a grid of candidate centres is constructed in a local neighbourhood of each point. This algorithm provides a tighter approximation guarantee but has significantly worse time complexity. Shenmaier also discusses the dual of the MEBwO problem, and the hardness of this dual is an interesting area of further research.

[6] presents an algorithm built on branch-and-bound [20] which orders the nodes of the tree in such a way that along with a first-in-first-out search strategy results in a small number of nodes explored. [8] also presents a branch-and-bound algorithm, this time using the idea of core-sets and a bi-criteria approximation with respect to both the radius and percentage of points covered. This paper also utilizes the algorithm as a classification model on the MNIST [21] data set (for further details, please see Section 4.2).

Due to the time constraints of this project and the difficulty of implementing branch-and-bound methods, we will not be implementing these branch-and-bound algorithms in this paper, and instead focus on methods which work from a more geometric mindset. We will implement Shenmaier's approximation and also investigate some improvement heuristics which may be run quickly in order to improve the initial solution provided from this method.

Chapter 3

Heuristic and Approximation Algorithms

In this chapter we propose some non-exact construction methods to solving the MEBwO problem which return a feasible solution. We also investigate some improvement heuristics which, given a feasible MEB or MEBwO, seek to locally find a new centre such that a smaller radius can be found.

3.1 Construction Methods

3.1.1 Average Point Heuristic

Statement

The Average Point Heuristic (APH) works by finding the average point, i.e. $c = \frac{1}{n}\sum_{i=1}^{n} a^i$, then finding the k closest points and returning the ball $B(c, r)$ where r is the greatest distance from c to each of the k points.

Algorithm 3: Average Point Shrinking Heuristic

Input: Data set $\mathcal{A}$, $\eta \in [0, 1]$
Output: Ball $B(c, r)$

1 $k := \lfloor n(1-\eta) \rfloor$;
2 $c := \frac{1}{n}\sum_{i=1}^{n} a^i$;
3 $D := \{\|c - a\| : a \in \mathcal{A}\}$;
4 Let D' be D sorted in ascending order;
5 Let $\delta = D'[k-1]$ be the $(k-1)$th element of D' (indexing from 0);
6 $\mathcal{A}' = \left\{a^i \in \mathcal{A} : D[i] \leq \delta, i = 1, \ldots, n\right\}$;
7 $r = \max_{a \in \mathcal{A}'} \|c - a\|$;
8 **return** $B(c, r)$

Complexity

This algorithm runs extremely quickly, with the dominating term in the time complexity being $O(nd \log n)$ on average as a result of sorting the list of distances on line 4. This is far faster than any other construction method and is more resistant to outliers than the MEB shrinking heuristic.

3.1.2 MEB Shrinking Heuristic

Statement

The MEB Shrinking Heuristic works by computing $\text{MEB}(\mathcal{A}) = B(c, r)$, then simply shrinking the radius until $\eta\%$ of points are covered. This is done by calculating the k closest points to c and

returning the ball $B(c, r')$ where r' is the greatest distance from c to each of the $n - k$ points.

Algorithm 4: MEB Shrinking Heuristic

Input: Data set $\mathcal{A}$, $\eta \in [0, 1]$, $\epsilon > 0$
Output: Ball $B(c, r)$
1 $k := \lfloor n(1 - \eta) \rfloor$;
2 Let c be the centre of the MEB returned by a heuristic for MEB($\mathcal{A}$);
3 $D := \{\|c - a\| : a \in \mathcal{A}\}$;
4 Let D' be D sorted in ascending order;
5 Let $\delta = D'[k - 1]$ be the $(k - 1)$th element of D' (indexing from 0);
6 $\mathcal{A}' = \left\{a^i \in \mathcal{A} : D[i] \leq \delta, i = 1, \ldots, n\right\}$;
7 $r = \max_{a \in \mathcal{A}'} \|c - a\|$;
8 **return** $B(c, r)$;

Complexity

The complexity of this heuristic mostly depends on the choice of heuristic for solving the MEB problem on $\mathcal{A}$. The dominant term in lines 3 though 7 is on average $O(n \log n)$ from sorting the list of distances, so the overall complexity is the greater out of this and the complexity of the heuristic. Our implementation uses Algorithm 1, which in this case is $O\left(\frac{nd}{\epsilon} + \frac{d^2}{\epsilon^{3/2}}\left(\frac{1}{\epsilon} + d\right) \log \frac{1}{\epsilon}\right)$ [19, Page 6].

3.1.3 Relaxation-Based Heuristic

Statement

This heuristic works by first solving the relaxed QCQP formulation for the MEBwO problem in Definition 2.3.1, and then making the assumption that a higher value of ξ_i (i.e. closer to 1) means that the model treats the data point a^i as "more" of an outlier. From this assumption, we pick the largest k values of ξ_i and treat each corresponding a^i as an outlier, therefore treating the remaining data as inliers. We then solve the MEB problem for this remaining data. A more formal outline of this algorithm is detailed in Algorithm 5.

This method will always return a feasible solution that covers $\eta\%$ of the data as we will always remove $k = \lfloor n(1 - \eta) \rfloor$ many points. As discussed in the previous chapter, solving the final MEB problem is relatively easy and selecting outliers using the MEBwO QCQP relaxation is trivial, but the difficulty lays in solving the initial QCQP relaxation, and as such a heuristic method which can solve this model quickly and return approximate solutions to each ξ_i is a valuable direction for further research in order to speed up this method.

Algorithm 5: Relaxation-Based Heuristic

Input: Data set $\mathcal{A} = \left\{a^1, \ldots, a^n\right\}$, percentage of data to be covered $\eta \in [0, 1]$, error tolerance for MEB heuristic $\epsilon > 0$, big-M parameter $M > 0$
Output: Ball $B(c, r)$
1 Let $\xi = [\xi_1, \ldots, \xi_n]$ be the relaxed binary variables returned by relaxed MEBwO solver for $\mathcal{A}$, η, and M;
2 Let ξ' be the smallest $k = \lfloor \eta \cdot n \rfloor$ elements of ξ;
3 Let $\mathcal{A}' := \left\{a^i \in \mathcal{A} : \xi_i \in \xi',\ i = 1, \ldots, n\right\}$;
4 Let c, r be the centre and radius of MEB($\mathcal{A}'$) returned by heuristic;
5 **return** $B(c, r)$;

Complexity

The complexity of this heuristic is difficult to determine, as the relaxed model must be solved by an optimization solver. Typically, and in our case, these solvers make use of interior point methods which are able to reach an optimal solution in polynomial time. We will benchmark this algorithm in Section 5.2 to gain some insight into how the runtime scales as the problem size varies.

3.1.4 Shenmaier's Approximation

Statement

This approximation scheme developed by Vladimir Shenmaier [30, Algorithm 1] is a brute-force algorithm for solving the MEBwO problem with $O(n^2 d)$ time complexity. It works by considering each point in the input set, finding the k-closest points to that point, then returning the point-distance pair such that the maximum distance from each point to each other point in its k-closest points is minimised. See Algorithm 6 for a detailed outline of this algorithm.

Algorithm 6: Shenmaier's Approximation [30, Algorithm 1]

Input: Data set $\mathcal{A}$, $\eta \in [0, 1]$
Output: Ball $B(c, r)$

1 $k = \lfloor n(1 - \eta) \rfloor$;
2 **for** $i = 1, \ldots, n$ **do**
3 $\quad D := \left\{ \left\| a^i - a \right\| : a \in \mathcal{A} \right\}$;
4 $\quad$ Let D' be D sorted in ascending order;
5 $\quad$ Let δ_i be the $(k-1)$th element of D' (indexing from 0);
6 $i^* := \arg\min_{i=1,\ldots,n} \delta_i$;
7 $c := a^{i^*}$;
8 $r := \delta_{i^*}$;
9 **return** $B(c, r)$;

Approximation Guarantee

From [30, Theorem 4] we know that Algorithm 6 returns a 2-approximation solution to the MEBwO problem. What this means is that if we have a minimum radius of r returned by Algorithm 6 for a data set $\mathcal{A}$, then the true optimal value to the MEBwO problem on that data is no less than $r/2$. This is useful for assessing how close a heuristic is to an optimal solution in the worst case when we have a solution via Algorithm 6, but not an optimal solution via solving the exact model.

Complexity

Referring to Algorithm 6, on line 1 we have an assignment which runs in $O(1)$ time. On lines 2 through 5 we have a `for` loop which runs n many times, and on line 3 we compute a list of n many distances, which for d-dimensional vectors runs in $O(nd)$ time. Line 4 involves sorting the aforementioned list, so the time complexity depends on the sorting algorithm used. It is well known within computer science that algorithms are often subject to a time-memory trade off, meaning that it is possible to design algorithms that run faster as a result of using less memory, and vice-versa. We can then assume that a sensible implementation of Algorithm 6 would use the fastest available sorting algorithm given that the size of our data sets are small enough that we do not run into memory constraints, so the recommended sorting algorithm is Quicksort [14] which runs on average in $O(n \log n)$ time. Line 5 simply accesses the $(k-1)$th element of the sorted list and so runs in $O(1)$ time.

Line 6 finds the minimum element from a set of values and runs in $O(n)$ time. Lines 7 and 8 are simple assignments and run in $O(1)$ time. Finally, our overall time complexity is then $O(n^2(d + \log n))$. This will usually reduce to $O(n^2 d)$ as typically $d > \log n$, though for particularly large n and very small d the alternative reduction of $O(n^2 \log n)$ is of course possible.

3.2 Improvement Heuristics

We now consider some improvement heuristics. The two heuristics described here are, strictly speaking, designed to improve an existing ball without considering outliers. Quick and effective methods exist for solving the MEB problem and so improvement heuristics have not been needed thus far, while

methods which inexactly solve the MEBwO problem may return a solution which has considerable space for improvement. These improvement heuristics may be used after a method which attempts to solve the MEBwO problem where the initial solution is not an MEB, since if we do have an MEB, that ball is by definition the minimum radius ball which may be fit to the given subset of data. So, in our implementation, we may use improvement heuristics after the average point heuristic, MEB shrinking heuristic, and Shenmaier's approximation since these methods do not necessarily return an MEB.

The two improvement heuristics work in a similar way, by considering an improving direction in which moving the centre of the MEB in that direction while keeping the same radius retains a feasible solution, then shrinking the radius to the furthest point from the new centre. The Direction-Constrained Single Step Heuristic (DCSSH) works by pure geometric reasoning to find a new centre, while the Direction-Constrained MEB (DCMEB) uses an optimization solver such as Gurobi to find an optimal step in the improving direction to minimise the radius of the new ball.

The inspiration for these heuristics comes from Shenmaier's Approximation, which returns a ball with some centre $c \in \mathcal{A}$. Our assumption is that a better solution exists in a local neighbourhood around c, i.e. we find a new centre that does not have to lie on a point in $\mathcal{A}$. Both algorithms as written here consider only a single step, but as they are quick to run, we may run each algorithm in succession to build a sequence of steps that lead to an improved solution. There are various stopping criteria one could consider in an implementation of these heuristics, such as setting an iteration limit or stopping once each iteration yields an improvement below some threshold. For ease of implementation, we will choose the iteration limit since each iteration is computationally cheap.

For an example of this refer to Figure 3.2.4 which shows the new radius after each iteration of each improvement heuristic on each data set for $n = 1000$, $d = 100$, and the standard parameters for each data type (see Section 4.2). These examples were run on one data set each rather than averaged over multiple data sets and so should not be used to draw any conclusions.

3.2.1 Direction-Constrained Single Step Heuristic

Derivation

For a visual aid to this derivation, please refer to Figure 3.2.1. Consider $B(\hat{c}, r)$ which contains $\mathcal{A}$. To find an improving direction we choose the furthest point from c, $\hat{a} = \arg\max_{a \in \mathcal{A}} \|c - a\|$, then the (negative) improving direction is defined to be $\beta = \hat{c} - \hat{a}$. We choose the direction $\hat{a} \to \hat{c}$ as opposed to $\hat{c} \to \hat{a}$ in order to simplify later derivations.

Now that we have a direction β, we can move the initial ball in this direction by simply subtracting βx from $\hat{c}$ where $x \in \mathbb{R}^{\geq 0}$ is some scalar, i.e. our new centre is $c = \hat{c} - \beta x$. We would like to choose x such that the new ball remains feasible by ensuring that $\left\|c - a^i\right\| \leq r$ for $i = 1, \ldots, n$, and we can do this by considering the distance from each a^i in the direction of β to the surface of the initial ball and choose x such that each of these distances is greater than $\|\beta x\|$.

More formally, for each $a^i \in \mathcal{A}$ let x_i be the scalar such that $a^i + \beta x_i$ lies on the surface of the initial ball, i.e. $\left\|(a^i + \beta x^i) - \hat{c}\right\| = r$. Again, to simplify derivations, denote the direction $\hat{c} \to a^i$ by $\alpha_i = a^i - \hat{c}$. To calculate each x_i, we square this equation and set $\gamma = r^2$ to get

$$\begin{aligned} & \alpha_i^T \alpha_i + 2x_i \alpha_i^T \beta + x_i^2 \beta^T \beta = \gamma \\ \implies & \beta^T \beta x_i^2 + 2\alpha_i^T \beta x_i + \alpha_i^T \alpha_i - \gamma = 0. \end{aligned}$$

Using the quadratic formula we then have

$$\begin{aligned} x_i &= \frac{-2\alpha_i^T \beta \pm \sqrt{4\left(\alpha_i^T \beta\right)^2 - 4\beta^T \beta\left(\alpha_i^T \alpha_i - \gamma\right)}}{2\beta^T \beta} \\ &= \frac{-\alpha_i^T \beta \pm \sqrt{\left(\alpha_i^T \beta\right)^2 - \beta^T \beta\left(\alpha_i^T \alpha_i - \gamma\right)}}{\beta^T \beta}. \end{aligned}$$

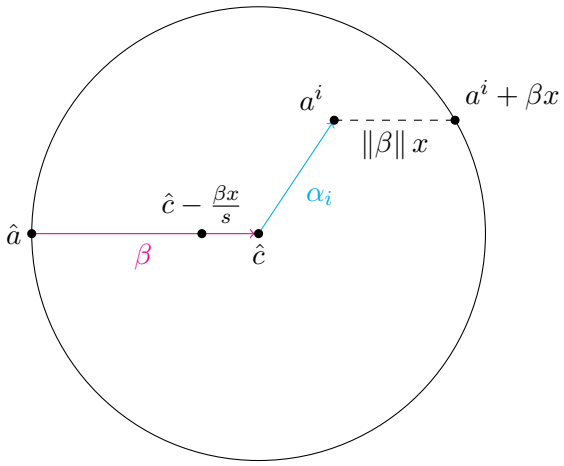


Figure 3.2.1: Visual Aid for the Direction-Constrained Single Step Heuristic

The two solutions to x_i corresponding to the positive and negative discriminants represent the steps in the positive and negative β directions respectively, then since from our derivations we are only interested in the positive β direction we can disregard the negative solution for x, so we can define the function $Q : \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}$ by

$$Q(\alpha_i, \beta, \gamma) = \frac{-\alpha_i^T \beta + \sqrt{\left(\alpha_i^T \beta\right)^2 - \beta^T \beta \left(\alpha_i^T \alpha_i - \gamma\right)}}{\beta^T \beta}.$$

Then let $x = \min \{Q(\alpha_i, \beta, \gamma) : i = 1, \ldots, n\}$, so $\|\beta x\|$ is the smallest distance from each a^i to the surface of the ball in the direction of β. Finally, by moving by some scalar multiple of $-\beta x$ from $\hat{c}$, we get a new feasible centre $c = \hat{c} - \beta x/s$ for some scalar $s \geq 1$. We give a simple proof that this new centre with unchanged radius indeed yields a feasible solution, and a visual aid is provided in Figure 3.2.2. See Algorithm 7 for a detailed outline of this algorithm.

Proposition 3.2.1. Consider the MEB problem for a data set $\mathcal{A}$ denoted by (P). Let $B(\hat{c}, r)$ be a feasible solution to (P). Then the ball $B(\hat{c} - \beta x/s, r)$ with β, x, and s as described above, is also a feasible solution to (P).

Proof. For each $a^i \in \mathcal{A}$ let $\hat{a}_\beta^i$ be the point on the surface of $B(\hat{c}, r)$ in the direction of β from a^i. Let $a_\beta^i = \hat{a^i}_\beta - \beta x/s$ be the point on the surface of $B(\hat{c} - \beta x/s, r)$ in the direction of β from a^i. Let $c = \hat{c} - \beta x/s$. Then, by the triangle inequality, we have

$$\begin{aligned} \left\|c - a_\beta^i\right\| &\geq \left\|c - a^i\right\| + \left\|a^i - a_\beta^i\right\| \\ \implies \left\|c - a^i\right\| &\leq \left\|c - a_\beta^i\right\| - \left\|a^i - a_\beta^i\right\| \\ &= r - \left\|a^i - a_\beta^i\right\| \\ &\leq r \end{aligned}$$

since $\left\|a^i - a_\beta^i\right\| \geq 0$. Hence $\left\|c - a^i\right\| \leq r$ for $i = 1, \ldots, n$ and so $B(c, r)$ is feasible for (P). □

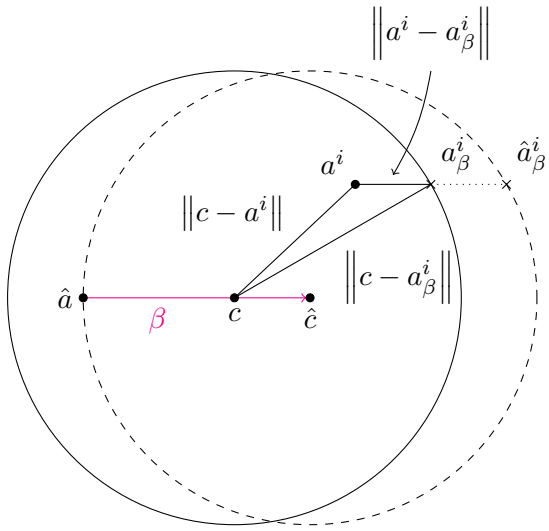


Figure 3.2.2: Visual Aid for Proposition 3.2.1

Algorithm 7: Direction-Constrained Single Step Heuristic

Input: Data set $\mathcal{A}$, Ball $B(\hat{c}, \hat{r})$, scalar $s \geq 1$
Output: Ball $B(c, r)$

1 $\hat{a} := \arg\max_{a \in \mathcal{A}} \|\hat{c} - a\|$;
2 $\beta := \hat{c} - \hat{a}$;
3 $\gamma := \hat{r}^2$;
4 **for** $i = 1, \ldots, n$ **do**
5 $\quad \alpha_i := a^i - \hat{c}$;
6 $\quad x_i := Q(\alpha_i, \beta, \gamma)$;
7 $x := \min\{x_i : i = 1, \ldots, n\}$;
8 $c := \hat{c} - \beta x / s$;
9 $r := \max_{i=1,\ldots,n} \left\|c - a^i\right\|$;
10 **return** $B(c, r)$;

Complexity

The complexity of the DCSSH is easy to determine. Referring to Algorithm 7, on line 1 we have a loop over each data point which runs in $O(n)$ time. On lines 4 through 6 we have a `for` loop which runs n many times, with the operations on lines 5 and 6 both being $O(d)$ as they are each single operations on d-dimensional vectors, so overall this loop runs in $O(nd)$ time. On lines 7 and 8 we find the minimum and maximum of two lists respectively, again $O(n)$. Finally, on line 8, a single operation on d-dimensional vectors runs in $O(d)$ time. Our dominating term is then $O(nd)$.

3.2.2 Direction-Constrained MEB Heuristic

Derivation

For a visual aid to this derivation, please refer to Figure 3.2.3. Consider $B(\hat{c}, \hat{r})$ which contains $\mathcal{A}$. Similarly to the DCSSH, for an improving direction we choose the furthest point in $\mathcal{A}$ from $\hat{c}$, $\hat{a} = \arg\max_{a \in \mathcal{A}} \|\hat{c} - a\|$, and let $\beta = \hat{a} - \hat{c}$ be this direction. Note that this is the negative of β used in the derivation of the DCSSH, as we are now considering the direction $\hat{c} \to \hat{a}$. Now, instead of calculating distances between each point in data and the surface of the ball to find a feasible step length from $\hat{c}$ towards $\hat{a}$, we formulate an optimization model to find an optimal step length such that the new ball still contains all data, but has a smaller radius.

We modify the optimization model for the MEB problem, adding the constraint that c must lie on the direction β from $\hat{c}$. It is possible to formulate this exactly using the constraint $c = x(\hat{c} + \beta)$,

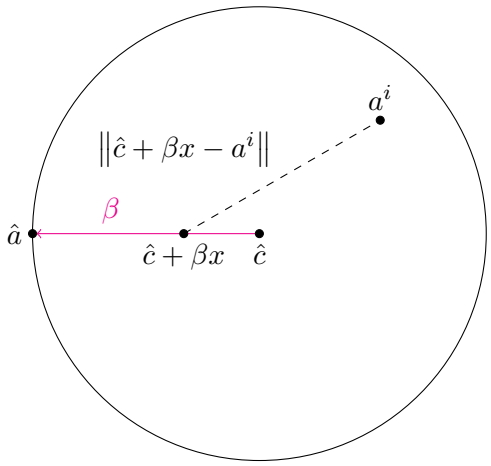


Figure 3.2.3: Visual Aid for the Derivation of the Direction-Constrained MEB

however a better formulation is to consider a variable $x \in \mathbb{R}^{\geq 0}$, then write $c = \hat{c} + \beta x$. Obviously, $\hat{c}$ is a constant vector, so this reduces the problem of finding a new d-dimensional vector c to that of finding one variable x. The optimization model for the DCMEB problem is then as follows:

$$\begin{array}{ll} \min\limits_{x,r} & r \\ \text{s.t.} & \|\hat{c} + \beta x - a^i\| \leq r, \quad i = 1, \ldots, n, \\ & x \geq 0. \end{array}$$

By setting $\gamma = r^2$, $\alpha^i = \hat{c} - a^i$ for $i = 1, \ldots, n$, and squaring the norm constraints, we get the following QCQP:

$$\begin{array}{ll} \min\limits_{x,\gamma} & \gamma \\ \text{s.t.} & \beta^T \beta x^2 + 2\alpha_i^T \beta x + \alpha_i^T \alpha_i \leq \gamma, \quad i = 1, \ldots, n, \\ & x \geq 0. \end{array}$$

This model has $n + 1$ constraints and, most crucially, 2 decision variables. This makes the model extremely computationally easy to solve by modern solvers such as Gurobi. In fact, like the original MEB problem, we may formulate the DCMEB problem as a SOCP by considering the cones

$$C_i = \left\{ (r, \beta x + \hat{c} - a^i) \in \mathbb{R}^{d+1} : \|\beta x + \hat{c} - a^i\| \leq \gamma \right\}.$$

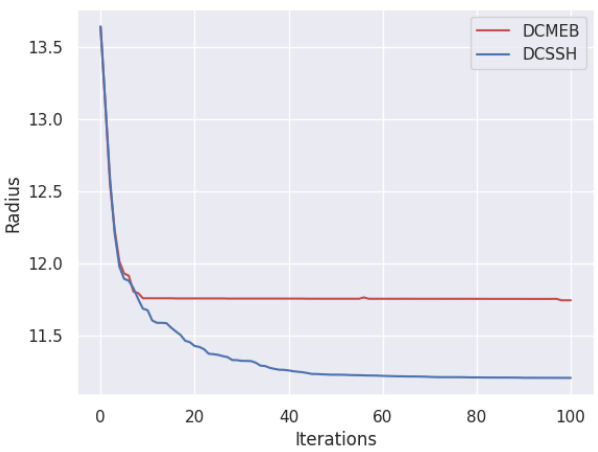
and as such, the DCMEB problem may be solved by interior-point methods [2]. The DCMEB heuristic is then to simply repeatedly solve the DCMEB problem given a data set $\mathcal{A}$ and initial ball $B(\hat{c}, \hat{r})$, using the new ball of each solved DCMEB problem for the next iteration.

Algorithm 8: Direction-Constrained MEB Heuristic

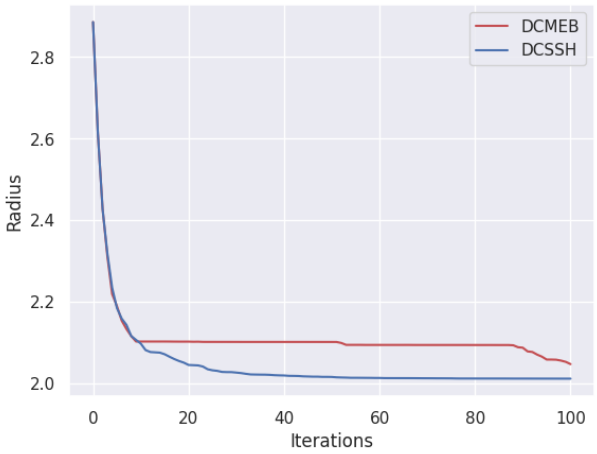
Input: Data set $\mathcal{A}$, Ball $B(\hat{c}, \hat{r})$
Output: Ball $B(c, r)$
$\hat{a} := \arg\max_{a \in \mathcal{A}} \|\hat{c} - a\|$;
Let x, r be the solutions returned by solver for the DCMEB problem for $B(\hat{c}, \hat{r})$;
$c := c + x(\hat{a} - \hat{c})$;
return $B(c, r)$;

Complexity

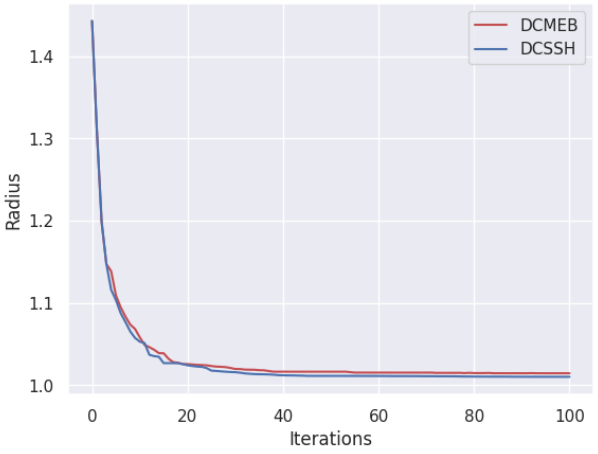
The operations in this heuristic are simply finding a maximally distant point which runs in $O(nd)$ time, solving the DCMEB problem, and an assignment which runs in $O(1)$ time. So, the interest is in the time complexity of the method used to solve the DCMEB problem. As this problem is a SOCP, it makes sense to apply interior-point methods, which may then solve it in polynomial time.



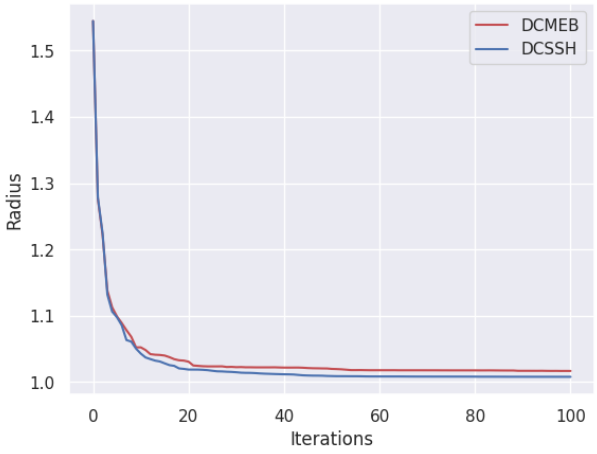
(a) Normal



(b) Hyperspherical Shell



(c) Uniform Ball



(d) Uniform Ball with Outliers

Figure 3.2.4: Reduction of Radius by each Iteration of Improvement Heuristics

Chapter 4

Implementation and Data

4.1 Code

4.1.1 Software

We implement our algorithms and data generation using Python 3.8 [34]. Python is a high-level procedural programming language which is commonly used in data science and mathematics for its low barrier to entry, wide-scale adoption, and rich package ecosystem. One drawback of using Python is that it is known to be slow compared to other languages.

An implementation of our work in a faster programming language such as C [17] will almost certainly result in faster run times—in fact many Python packages such as scikit-learn [26] and NumPy [13] utilize C to run the more time-consuming processes, and the Gurobi [11] solver is written in C while Python and other languages with Gurobi support are merely interfaces to this solver. A new up-and-coming programming language is Julia [3] which promises similar run times to C and has a mature library for interfacing with optimization solvers known as JuMP [9]. Due to the time constraints on this paper, we make use of the author's prior knowledge in Python for the implementation as previously mentioned, but again acknowledge that there are indeed potentially superior choices for this type of work.

Packages within Python we make use of in the implementation are NumPy [13] for numerical computation and linear algebra, pandas [27, 38] for data manipulation, Matplotlib [16] and Seaborn [36] for visualisation and plotting, and finally TensorFlow Datasets [31] to access the MNIST [21] data set.

For modelling and solving our exact models for the MEBwO and DCMEB problems, we use the Gurobi optimization solver [11]. At the time of writing, Gurobi is one of the fastest and most powerful mathematical programming solvers available and in particular is able to formulate and solve the QCQP and MIQCQP problems we require, so Gurobi was an easy choice of solver. Gurobi also has an easy-to-use Python interface which we make use of in our implementation.

4.1.2 Implementation

Our implementation can be found on GitHub[1] within the `src` folder and overall consists of around 2000 lines of code. All scripts used to run benchmarks and create figures are contained within this folder, and we have also written three packages contained therein. The `meb` package contains an object-oriented framework for MEB objects, defining `Ball`, `MEB`, and `MEBwO` objects within `ball.py` with a simple class hierarchy as seen in Figure 4.1.1.

These objects have attributes such as a centre and radius and allow us to define an MEB or MEBwO as a single variable as an instance of their corresponding object, which allows for more streamlined implementation, as opposed to tracking values such as the centre and radius within separate, unlinked variables. These objects also implement methods such as `.fit()` which, given a data input and a chosen method and relevant parameters, will solve the MEB or MEBwO problem on that data set. Other methods are implemented, and the reader may refer to the aforementioned `ball.py` module for

[1] `https://github.com/tomholmes19/Minimum-Enclosing-Balls-with-Outliers` [15]

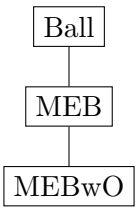


Figure 4.1.1: Class Hierarchy of Ball Objects

a comprehensive guide. All other modules within this package are used by `ball.py` and are simply used to abstract methods out of this module so as to adhere to good programming principles. Our exact models are formulated and solved using Gurobi in `gurobi_solvers.py`.

The `data` package contains modules regarding the data used in this paper. The `generation.py` module contains functions which generate each type of data set described in Section 4.2, as well as some others which were used in the testing and development of our methods but are not mentioned further within this paper. `loading.py` contains a few utility functions regarding loading and saving data consistently for ease of implementation. `benchmarking_data.py` can be run to generate the data used in our benchmarking, and must be run if the reader wishes to obtain our data as the data sets generated are cumulatively too large to be uploaded to GitHub.

The `benchmarking` package contains the `trials.py` module which contains functions used for benchmarking. The other module in this package, `utils.py`, simply abstracts various functions away from `trials.py`.

4.2 Data

In this section, we present and discuss the types of data we will use for bench-marking our algorithms for solving the MEBwO problem. Functions written and used for this paper can be found in the module `src/data/generation.py`.

On Random Number Generation

In order to generate our data, we must have some means of creating random numbers. To generate uniform random variates from the distribution $U(0,1)$, the method of linear congruential generators [32, 28] is well known within the literature, and provide us a computationally trivial way of generating pseudo-random uniform numbers.

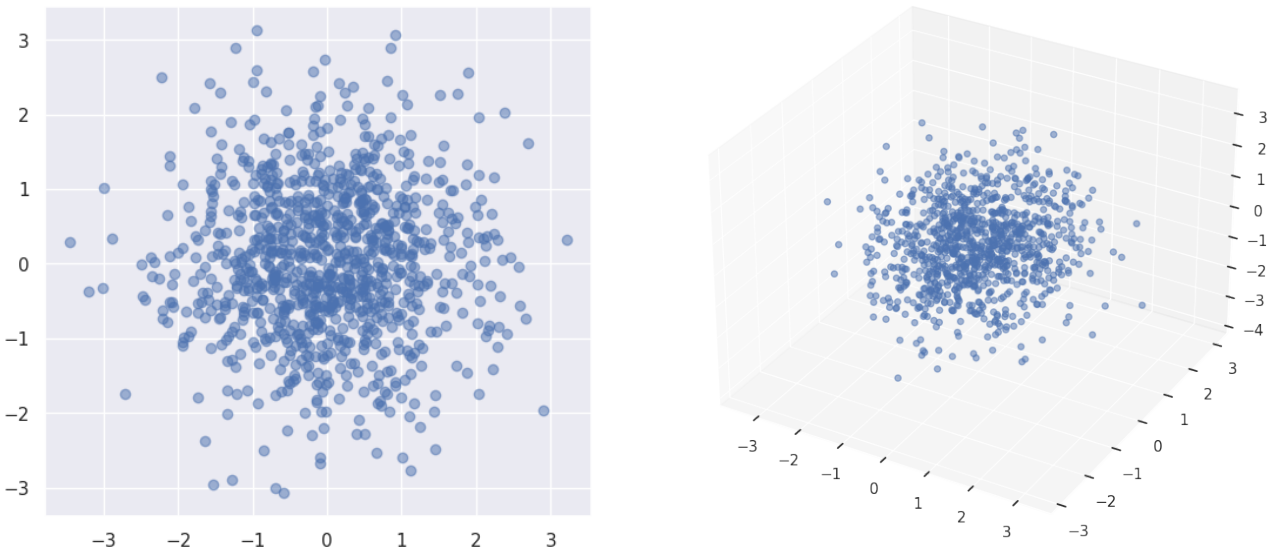
With a source of pseudo-random uniform numbers, it is then possible to create a wide variety of different random variates from various distributions. For example, to sample a random variate from the distribution $U(a,b)$ where $a > b$, one must simply generate a random variate U, from the distribution $U(0,1)$, then the random variates $U(b-a)+a$ follows the distribution $U(a,b)$. To generate standard normal random variates from uniform random variates, one can use methods such as the Box-Muller transform [4].

Using linear congruential generators and then applying transformations to obtain other random variates allows the user direct control over the randomness in their computations, allowing for very easy reproducibility. In our implementation, we instead use the `random` package from NumPy [13]. This allows us to generate uniform random numbers using `np.random.uniform` and `np.random.normal` easily, and we may set a seed for the pseudo-random generator using `np.random.seed` for reproducibility.

4.2.1 Normal

Our first data set is constructed by generating standard normal variates, that is, random numbers from the normal distribution $X \sim N(0,1)$ with mean 0 and variance 1. Many examples of real-life data have been shown to be normally distributed, and the normal distribution is symmetric around its mean in each dimension, so it is an obvious choice to be fit with an MEBwO.

This can be easily generated using NumPy [13] by calling the function `np.random.normal(0, 1, (n,d))` to generate n many d-dimensional standard normal vectors. See Figure 4.2.1 for an example.



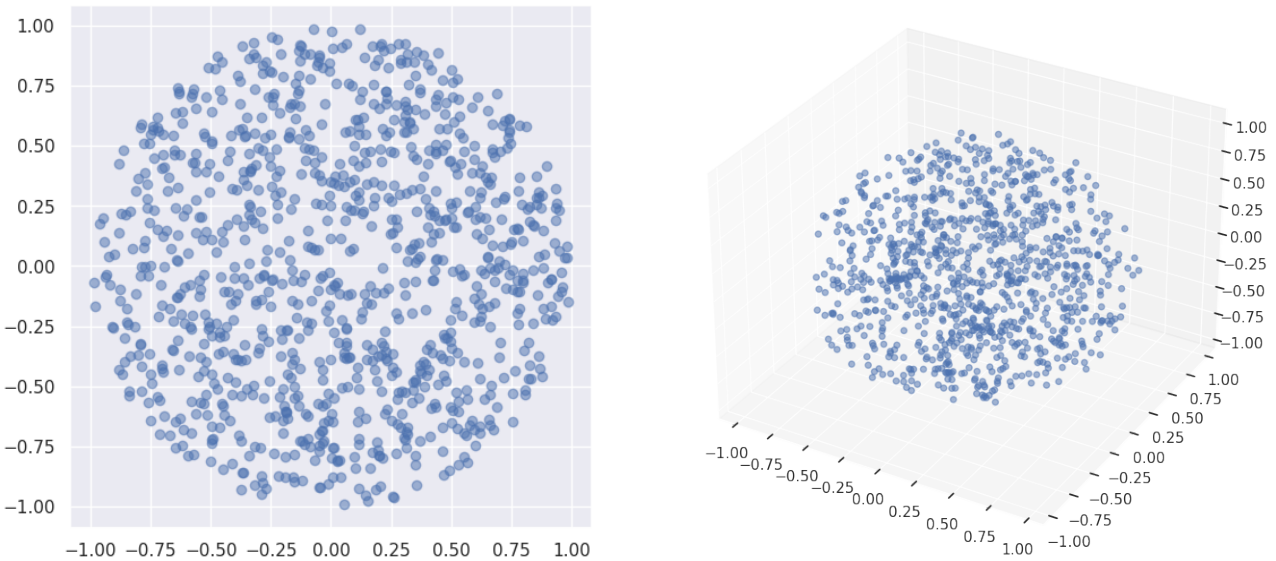
(a) Two Dimensions

(b) Three Dimensions

Figure 4.2.1: Example of Standard Normal Data

4.2.2 Uniform Ball

Our next data type consists of points sampled uniformly from a hypersphere of dimension d, and in this paper we will specifically choose to uniformly sample points within the unit ball centred on the zero vector, $B(\mathbf{0}, 1) \subseteq \mathbb{R}^d$. See Figure 4.2.2 for an example. Like with the choice of normal data, spherical data such as that of a uniform ball is a reasonable choice for the MEBwO problem.



(a) Two Dimensions

(b) Three Dimensions

Figure 4.2.2: Example of a Uniform Ball

To generate points uniformly within a unit ball, we must first generate points uniformly on the unit hypersphere. [25] provides a fast and easy to implement method of doing this, by first generating the

standard normal vector $\mathbf{x} = (x_1, \ldots, x_d)$ where $x_i \sim N(0,1)$ for $i = 1, \ldots, d$, and then the distribution of the normalized vectors $\mathbf{x}/\|\mathbf{x}\|$ is uniform over the hypersphere of dimension d.

A naive approach would then be to multiply the point on the surface of the hypersphere by a uniform random variate $U \sim U(0,1)$ to obtain a point inside the unit ball, and indeed this does generate points within the unit ball, but they will not be uniformly distributed and points will be more densely located near the centre of the ball. Instead, from [33] we find that we can instead multiply our point on the surface of the hypersphere by $U^{1/d}$ to have our points be uniformly distributed within the hypersphere. More generally, multiplying our point on the surface by $r \cdot U^{1/d}$ for some radius $r > 0$ gives us a point within the hypersphere of radius r. See Algorithm 9 for a detailed outline of this method.

Algorithm 9: Algorithm for Generating Points in a Hypersphere of Radius r

Input: Dimension $d \in \mathbb{N}$, radius $r \in \mathbb{R}^{\geq 0}$, centre $c \in \mathbb{R}^d$
Output: Point $x \in \mathbb{R}^d$

Generate $x := (x_1, \ldots, x_d)$ where $x_i \sim N(0,1)$;
$x := x/\|x\|$;
Generate $U \sim U(0,1)$;
$x := r \cdot x \cdot \left(U^{1/d}\right)$;
$x := x + c$;
return x

One benefit of using this data is that when we are bench-marking the Relaxation-Based Heuristic, we do not need to spend time calculating a lower bound for M. Since we know our data set is contained within a unit ball, our diameter can be at most 2. The true diameter of any given uniform ball data set will be some value marginally close to 2, but crucially, less than 2. This means an M value of 2 will ensure that our big-M constraints work correctly, and the closeness of the true diameter to 2 means that the increase in solution time for the exact model is negligible.

4.2.3 Hyperspherical Shell

Now we will discuss the data type we call the Hyperspherical Shell. This consists of points sampled uniformly between the surfaces of two concentric hyperspheres with the same centres, where one is strictly contained within the other. We may explicitly define this region as

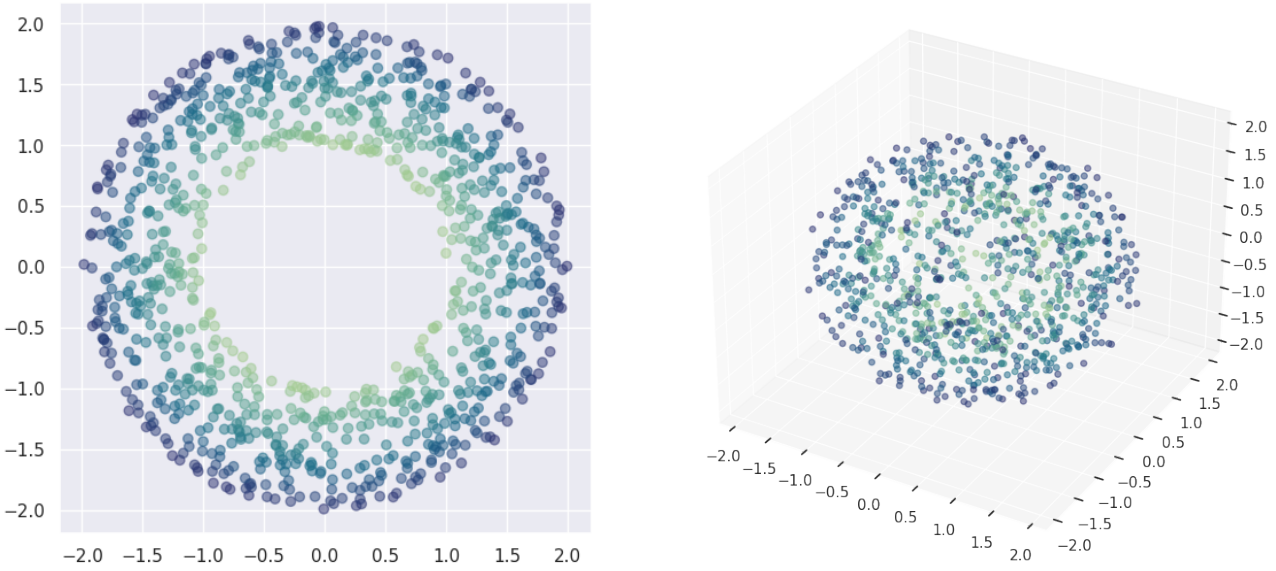
$$H(c, r_1, r_2) = \left\{ x \in \mathbb{R}^d : r_1 \leq \|x - c\| \leq r_2 \right\}$$

In this paper we choose the zero vector as our centre, an inner radius of 1, and an outer radius of 2. See Figure 4.2.3 for an example. Note that we colour the points in this plot by their distance to the origin, so that the shape of the data may be seen in the three-dimensional plot. In two dimensions the shape approximated by this data is known as an annulus, and in three dimensions a spherical shell, so for the general d-dimensional case we opt to call it a hyperspherical shell.

This data type is chosen as the optimal centre of the MEBwO on any sufficiently large data set is guaranteed to be "far" away from any point in the data set, specifically it will be near the zero vector. This poses issues for Shenmaier's approximation, which always returns a centre which is a point within the data set.

To generate points within this set, we can make use of Algorithm 9 to generate points uniformly in a hypersphere whose radius is our chosen outer radius, then use an acceptance-rejection method to reject points whose distance from our chosen centre is less than our chosen inner radius. A common drawback of acceptance-rejection methods is that of efficiency, i.e. of the space in which we are generating points, what percentage of this space is our acceptance region. Depending on the amount of points desired, a low efficiency can result in very long computation times due to the number of points being rejected.

Acceptance-rejection methods are usually unsuitable for generating high-dimensional data. For example, if we wanted to generate points within a unit ball using an acceptance-rejection method by first generating points uniformly inside a hypercube, the ratio of the volume of a hypersphere inscribed



(a) Two Dimensions

(b) Three Dimensions

Figure 4.2.3: Example of a Hyperspherical Shell

within a hypercube to the volume of that hypercube tends to 0 as $d \to \infty$ [22].

Fortunately, in our case, the ratio of the acceptance region to the total volume of the hypersphere tends to 1 as $d \to \infty$. To see this, note that the volume of a d-dimensional hypersphere with radius r is

$$V_d(r) = \frac{S_d r^d}{d}$$

where S_d is the hyper-surface area of the unit hypersphere in d dimensions [37]. Now, for inner radius r_1 and outer radius r_2 where $r_2 > r_1$, we have

$$\begin{aligned} \frac{V_d(r_1)}{V_d(r_2)} &= \frac{\left(\frac{S_d r_1^d}{d}\right)}{\left(\frac{S_d r_2^d}{d}\right)} \\ &= \frac{r_1^d}{r_2^d} \\ &= \left(\frac{r_1}{r_2}\right)^d \end{aligned}$$

which tends to 0 as $d \to \infty$ since $r_2 > r_1$. So, for sufficiently large d, the efficiency of this acceptance-rejection method is close to 1. We outline our acceptance-rejection method in Algorithm 10

Algorithm 10: Acceptance-Rejection Method for Generating Points in a Hyperspherical Shell

Input: Dimension $d \in \mathbb{N}$, inner radius $r_1 \in \mathbb{R}^{\geq 0}$, outer radius $r_2 \in \mathbb{R}^{\geq 0}$, center $c \in \mathbb{R}^d$
Output: Point $x \in \mathbb{R}^d$

```
while True do
    Generate x from Algorithm 9 with r = r_2;
    if ||x − c|| > r_1 then
        break;
return x;
```

Like with the Uniform Ball data, Hyperspherical Shell data will have a guaranteed upper bound

to the diameter, and so we will not need to calculate M.

4.2.4 Uniform Ball with Outliers

Our final randomly generated data set is the Uniform Ball with Outliers. This data type consists of a uniform ball with a set amount of "outliers" which we generate ourselves. The main benefit of using this data type to benchmark our algorithms is that since we can choose the exact number of points outside the uniform ball, if we set $\eta\%$ of our n points to be outside the ball with radius r then $\mathrm{MEBwO}(\mathcal{A}, \eta)$ will have a radius close to r.

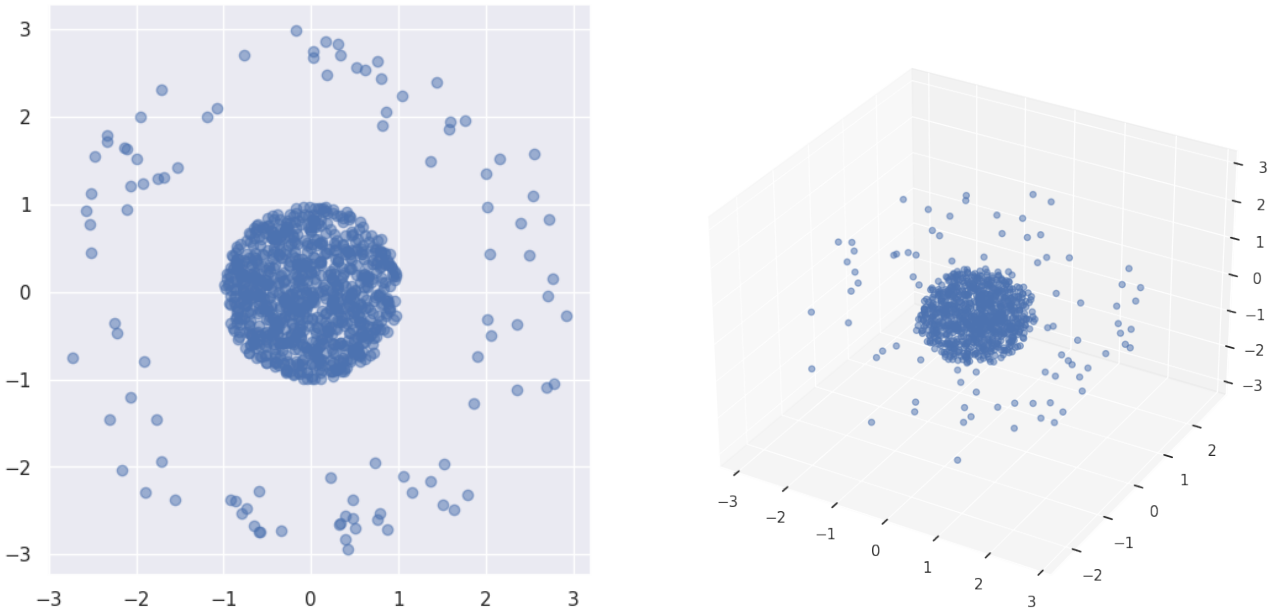
To generate this data, we simply use Algorithm 9 to generate a uniform unit ball with radius 1 which contains $\eta\%$ of our data, and then use Algorithm 10 with inner radius 2 and outer radius 3 which contains $(1-\eta)\%$ of our data. We choose an inner radius of 2 so that there is a distinct gap between the two parts of the data in the interest of having a clear inner ball. See Figure 4.2.4 for an example. Like before, we do not need to calculate an M value for this data as we have a guaranteed upper bound to the diameter.

Algorithm 11: Algorithm for Generating a Uniform Ball with Outliers

Input: Number of points $n \in \mathbb{N}$, dimension $d \in \mathbb{N}$, percentage of inliers $\eta \in [0,1]$, radius of ball $r \in \mathbb{R}^{\geq 0}$, inner radius of shell $r_1 \in \mathbb{R}^{\geq 0}$, outer radius of shell $r_2 \in \mathbb{R}^{\geq 0}$

Output: Data set $\mathcal{A} \subseteq \mathbb{R}^d$

$n_1 := \lfloor \eta \cdot n \rfloor$;
$n_2 := \lfloor (1-\eta) n \rfloor$;
Let $\mathcal{A}_1$ be the data set generated by running Algorithm 9 n_1 many times for $r = r$;
Let $\mathcal{A}_2$ be the data set generated by running Algorithm 10 n_2 many times for $r_1 = r_1$, $r_2 = r_2$;
$\mathcal{A} := \mathcal{A}_1 \cup \mathcal{A}_2$;
return $\mathcal{A}$



(a) Two Dimensions

(b) Three Dimensions

Figure 4.2.4: Example of a Uniform Ball with Outliers

4.2.5 MNIST

In the interest of easy reproducibility and an example of a real-life application, we would like to test our construction methods on the MNIST data set [21]. This data set consists of 70000 hand-written

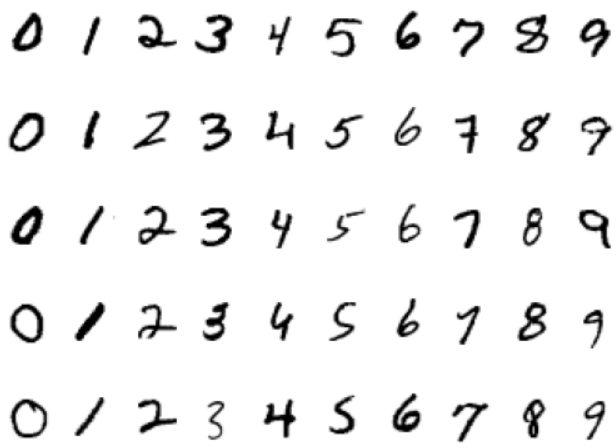


Figure 4.2.5: Example of MNIST Data

digits from 0–9 which are size-normalized and centred in 28×28 pixel greyscale images, and examples can be seen in Figure 4.2.5. MNIST is a popular data set for testing digit recognition models, a specific application of the broader class of image recognition models within computer vision. We may fit an MEBwO to the subset of data for a specific digit, and this ball may then be used for outlier recognition or as a more robust binary classification model than if one were to use an MEB. Further to this, separate balls may be fit to each digit and used to make multi-class classifications, either in a voting-based ensemble model or as part of a decision tree. In Section 5.4 we will apply two of our algorithms as an outlier recognition method.

In order to make this data usable by our algorithms, we would like to express each image as a vector. To do this, we may simply transform each pixel value (an integer in the range 0–255) to an element of a 784-dimensional array, and this is easily achieved using the `np.ndarray.flatten` method from NumPy.

Chapter 5

Experiments

In this chapter, we present our experimental results from applying our algorithms to different types of data. For each class of benchmarks, we first discuss the methodology used to gain results and then present and discuss those results.

As discussed in Section 4.1.1 we implement our algorithms in Python 3.8. Results were obtained on a HP ENVY x360-15-dr0005na[1] laptop running Ubuntu 20.04.2 LTS, with an Intel® Core™ i7 8565U processor and 16 GB DDR4-2400 SDRAM.

5.1 Exact Model

5.1.1 Methodology

Our interest in benchmarking the exact solver is simply to demonstrate the impracticality of using it to solve any reasonably sized problems, rather than an assessment of how this method is able to solve different types of problems. So, in order to save time, we benchmark the solver just on normal data.

Parameters chosen for n were $n = 300$ to $n = 750$ in steps of 50 with $d = 10$ and $\eta = 0.9$ fixed. For d we choose $d = 2$ to $d = 20$ in steps of 2 with $n = 300$ and $\eta = 0.9$ fixed, and for η we choose $\eta = 0.5$ to $\eta = 1$ in steps of 0.05 with $n = 300$ and $d = 10$ fixed. Note that the $\eta = 1$ case reduces to the standard MEB problem.

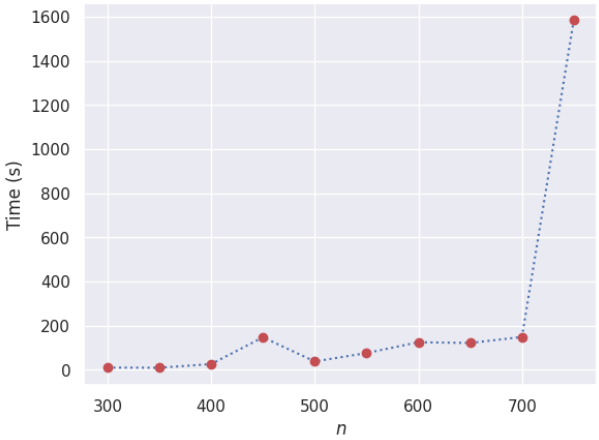
In Section 2.3.2 we discussed the effects of the big-M parameter on the solution time of the solver, and so we also benchmark the solution time when this parameter is varied. To calculate a lower bound on M which ensures that our constraints work as intended, we use the maximum pairwise distance as discussed in the aforementioned section. This is a suitable choice as the size of data sets which we use to benchmark the solver are relatively small and so calculating each pairwise distance does not take too much time. Then, we solve the model on the same data set while increasing M in steps of $M/2$.

We generate a large data set (1000×100000) of random normal variates, from which we may sample n many d-dimensional vectors to use as a data set. For each choice of parameters, we solve 5 different models and record details such as the centre and radius, elapsed time from the solver, and more to a log file. While we choose 5 for our number of trials, more is always better in this context, but due to the time constraints of the project we deemed 5 to be a sufficient amount of trials. In presenting our results, we give the average elapsed time, though exact times for each solved instance may be found in the log files in the `benchmarks` folder in the GitHub repository.

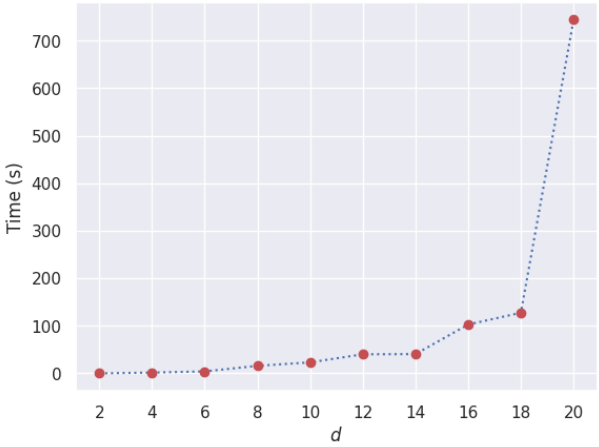
5.1.2 Results

In Figure 5.1.1a and Figure 5.1.1b we observe gradual increases in runtime as each parameter increases, until we see a sharp increase in runtime. This is what we expected due to the highly combinatorial nature of this problem, though we did not expect such a large and sudden change. One potential explanation to this is optimizations made by the Gurobi solver, which make smaller instances quicker to solve, but eventually lose effectiveness as problem size increases. These optimizations are how

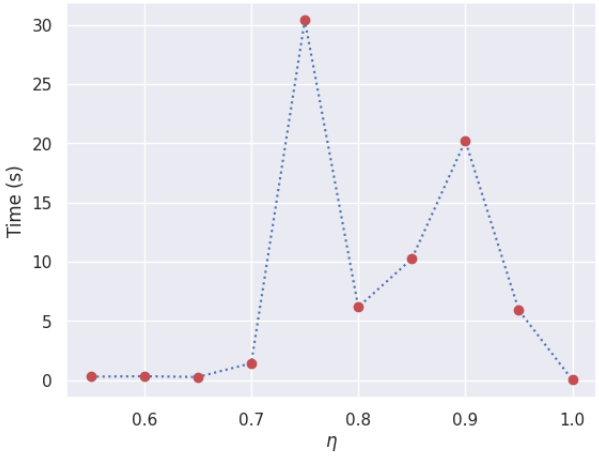
[1] `https://support.hp.com/gb-en/document/c06378310`



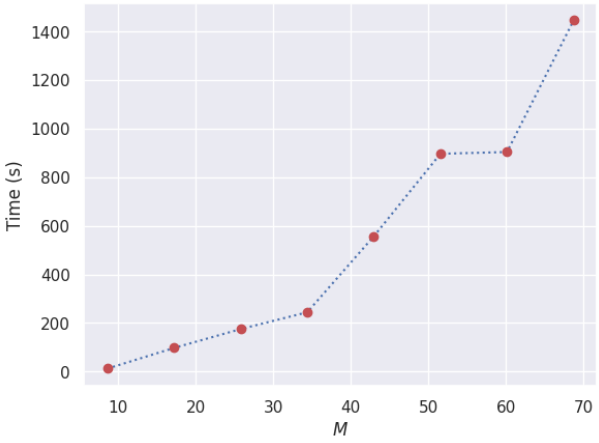
(a) By n ($d = 10$, $\eta = 0.9$)



(b) By d ($n = 300$, $\eta = 0.9$)



(c) By η ($n = 300$, $d = 10$)



(d) By M ($n = 300$, $d = 10$, $eta = 0.9$)

Figure 5.1.1: Runtimes for the Exact Model on Normal Data

Gurobi Optimization make their money and are not publicly known, so we are not able to comment on exactly how they achieve this.

Figure 5.1.1c also produces strange results, as we would expect the maximum number of feasible solutions to be available when $\eta = 0.5$, but the solver is actually able to solve these instances from 0.5 up to 0.7 extremely quickly. We observe spikes at $\eta = 0.75$ and $\eta = 0.9$, and again the exact behaviour we observe here is hard to explain due to the closed nature of the Gurobi solver.

Figure 5.1.1d shows exactly what we expect, which is increasing runtime as M increases beyond what is necessary. This provides justification for some sort of optimized approach to calculating a lower bound, as we have done in order to reduce run times as discussed in Section 2.3.2.

5.2 Construction Methods

5.2.1 Methodology

To benchmark our construction methods, we take a similar approach to that of benchmarking the exact model. We generate separate data sets of each type for a given set of parameters, so that each method is benchmarked on exactly the same data sets.

For benchmarking n we choose $n = 1000$ to $n = 28000$ in steps of 3000 with $d = 30$ and $\eta = 0.9$ fixed. For d we choose $d = 10$ to $d = 100$ in steps of 10 with $n = 10000$ and $\eta = 0.9$ fixed. For η we choose $\eta = 0.5$ to $\eta = 0.9$ in steps of 0.1 with $n = 10000$, $d = 30$ fixed.

Like with the exact model, we solve the MEBwO problem with our methods on 5 different data sets for each choice of parameters, and again would prefer to do more, but we chose 5 due to the

time restrictions. Each solved instance has its solutions, runtime and other details recorded to a log file. Runtime is recorded using Python's `timeit` library, calling the timer before and after running each algorithm. When we use the Relaxation-Based Heuristic, we calculate a lower bound for the M value required to solve the relaxed model by solving the MEB problem and setting $M := 2r$ where r is the solution for the radius of the MEB problem, as discussed in Section 2.3.2. This is only actually necessary for the normal data, as each of the other data types has a maximum diameter determined by the parameters chosen to generate it (see Section 4.2).

In presenting our results, we again report average radii and runtimes while exact solutions and runtimes are available in the log files in the `benchmarks` folder in the GitHub repository and tabular results may be found in the Appendix .2. We provide graphs which group results by data type and algorithm, so one may compare the performance of each algorithm on one data type as well as the performance of one algorithm on each data type. When grouping by data type, we also present the radii returned by each method, so we may compare the radii returned by each method on the same data, while it does not make sense to compare radii returned by one algorithm on different data types. Note that in some plots, some methods are not visible, but this is because they are covered by the markers for other methods when they give the same solutions. We do not report runtimes for the average point heuristic as this runs extremely quickly, but runtime plots are given separately in Appendix .1 for completeness. When grouping by function, we are interested in how the runtime varies so that we may verify our time complexity derivations and do not report solutions.

5.2.2 Results

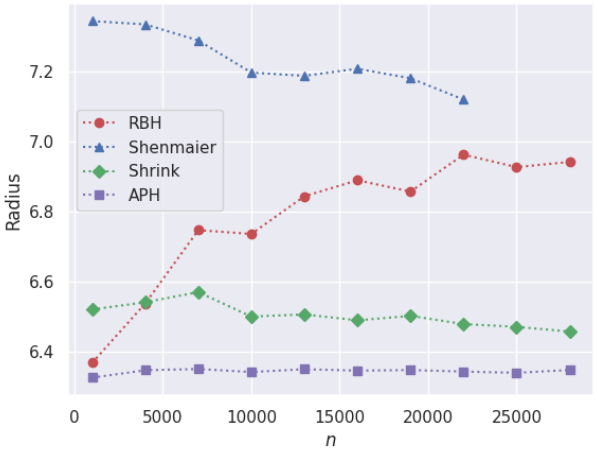
By Data Type

On normal data (Figure 5.2.1), we observe that Shenmaier's approximation gives strictly worse solutions as both n and d vary. Shenmaier's approximation also has the longest run time for each n, while for d it runs in less time than the relaxation-based heuristic for d greater than 50. The average point heuristic tends to perform the best on normal data, while also sporting near-instant runtimes.

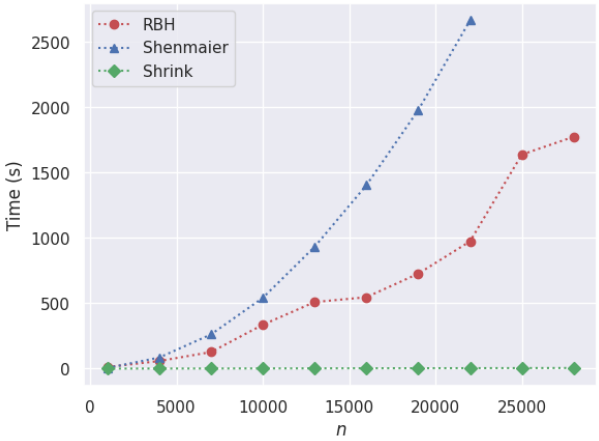
On uniform ball data (Figure 5.2.2), the relaxation-based heuristic and shrink heuristic give the same solutions as each parameter varies, but the shrink heuristic has strictly shorter runtimes for each parameter. The average point heuristic also gives the same solutions with insignificant runtimes. Shenmaier's approximation has shorter runtimes for d greater than 70, but has significantly worse radii on these instances (around 40% worse).

On hyperspherical shell data (Figure 5.2.3), we observe a similar story to that of uniform ball data, with the relaxation-based heuristic, shrink heuristic, and average point heuristic again giving the same results, with the shrink heuristic running in less time than the relaxation-based heuristic for each parameter. Shenmaier's approximation again is the fastest method for d greater than 70, but like before has strictly worse performance.

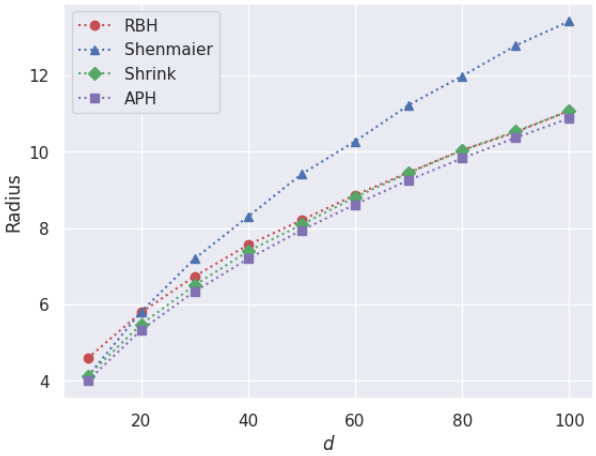
On uniform ball with outliers data (Figure 5.2.4), the relaxation-based heuristic is our best performing method, though we observe a larger than usual increase in runtime as the dimension increases, with a runtime of a little over an hour for each instance at $d = 80$, so we were unable to solve instances for $d = 90$ and $d = 100$ within the time constraints. The shrink heuristic as usual gives short runtimes with only marginally worse radii, though also marginally worse radii than the average point heuristic for small n. Each of these methods have very similar solutions for large n and all d. Recall that the optimal solution for our uniform ball with outliers data will be a radius of 1, which is indeed returned by the relaxation-based heuristic, while the shrink heuristic returns radii marginally above 1. Shenmaier's approximation gives far worse than expected solutions, on average around 55% worse than optimal.



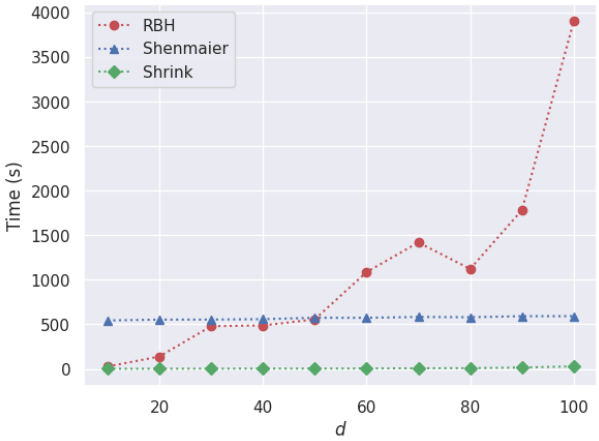
(a) Radius as a Function of n ($d = 30$, $\eta = 0.9$)



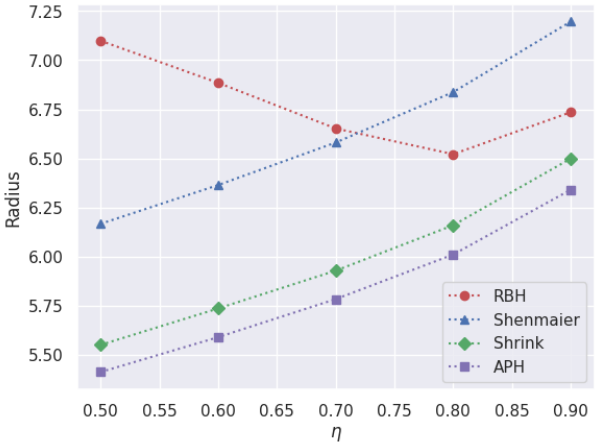
(b) Runtime as a Function of n ($d = 30$, $\eta = 0.9$)



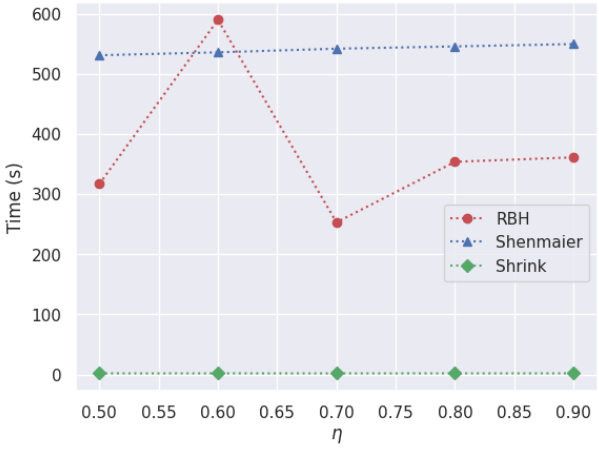
(c) Radius as a Function of d ($n = 10000$, $\eta = 0.9$)



(d) Runtime as a Function of d ($n = 10000$, $\eta = 0.9$)

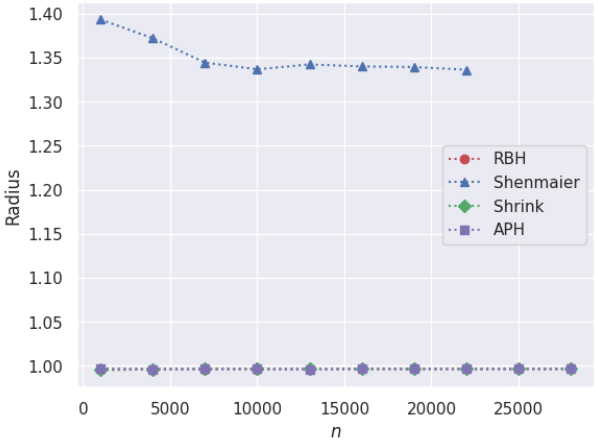


(e) Radius as a Function of η ($n = 10000$, $d = 30$)

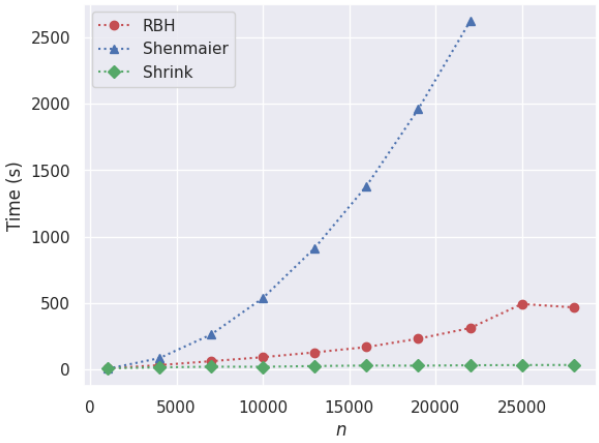


(f) Runtime as a Function of η ($n = 10000$, $d = 30$)

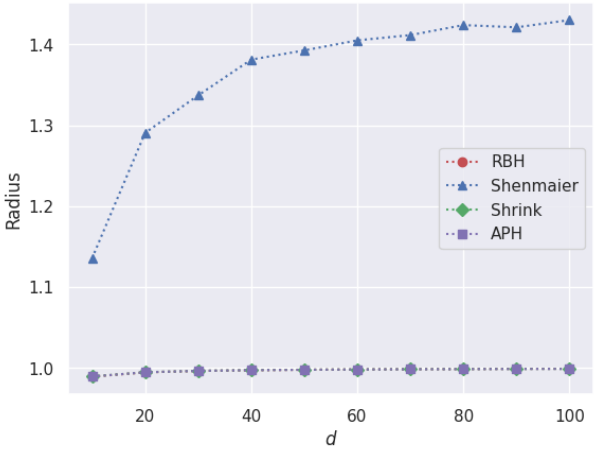
Figure 5.2.1: Results for each Algorithm on Normal Data



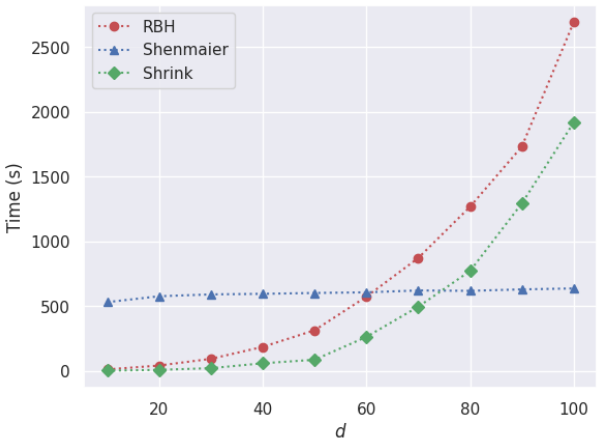
(a) Radius as a Function of n ($d = 30$, $\eta = 0.9$)



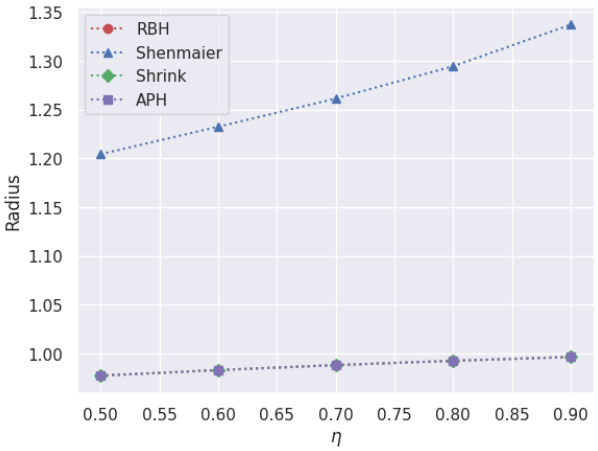
(b) Runtime as a Function of n ($d = 30$, $\eta = 0.9$)



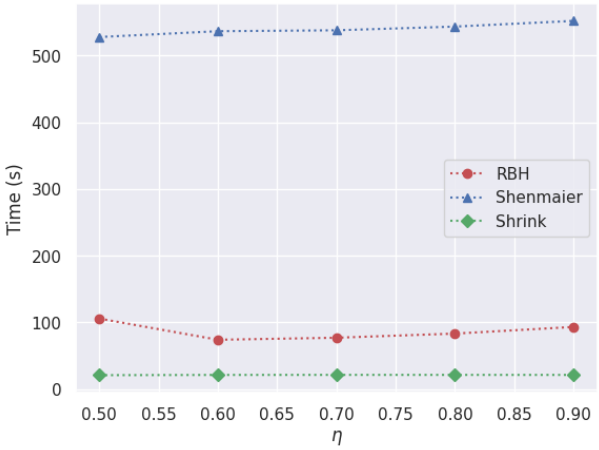
(c) Radius as a Function of d ($n = 10000$, $\eta = 0.9$)



(d) Runtime as a Function of d ($n = 10000$, $\eta = 0.9$)

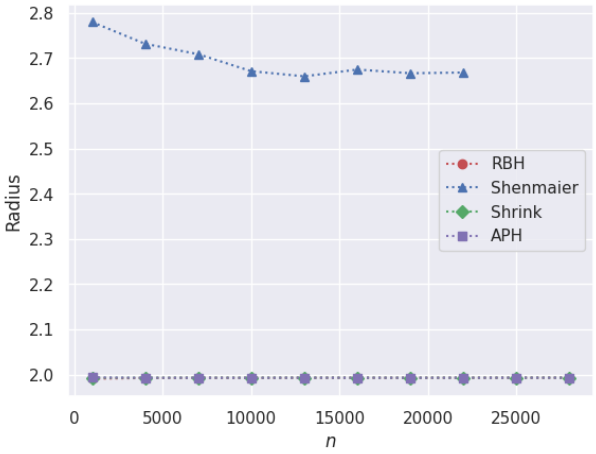


(e) Radius as a Function of η ($n = 10000$, $d = 30$)

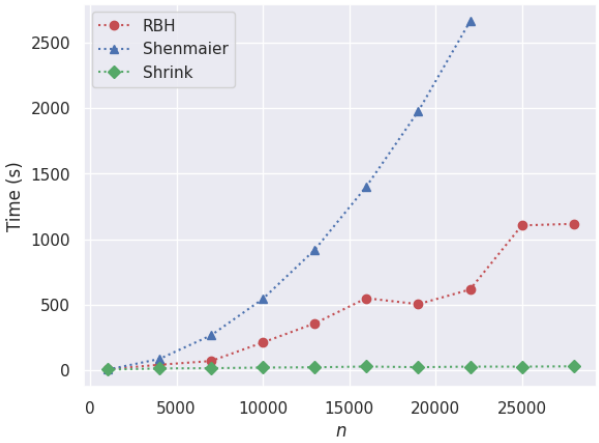


(f) Runtime as a Function of η ($n = 10000$, $d = 30$)

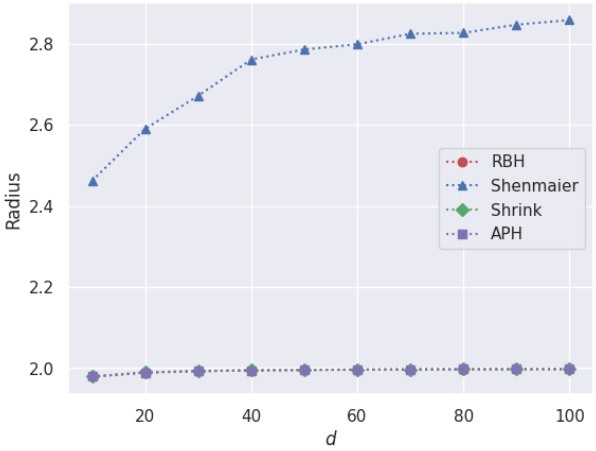
Figure 5.2.2: Results for each Algorithm on Uniform Ball Data



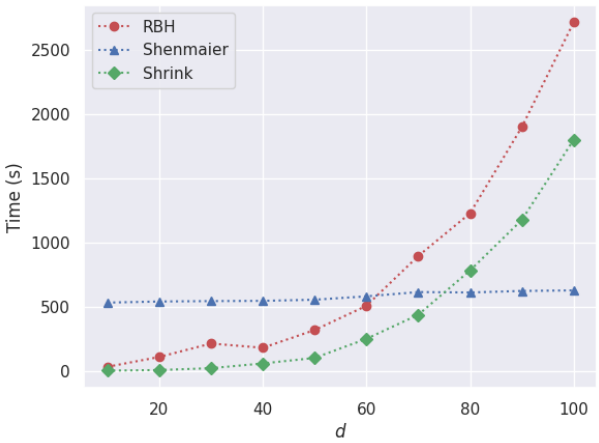
(a) Radius as a Function of n ($d = 30$, $\eta = 0.9$)



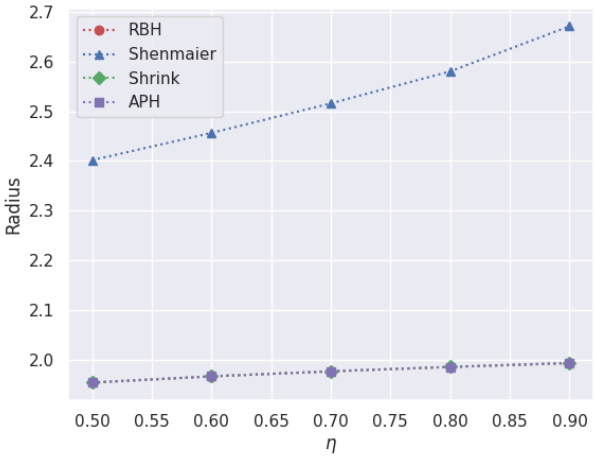
(b) Runtime as a Function of n ($d = 30$, $\eta = 0.9$)



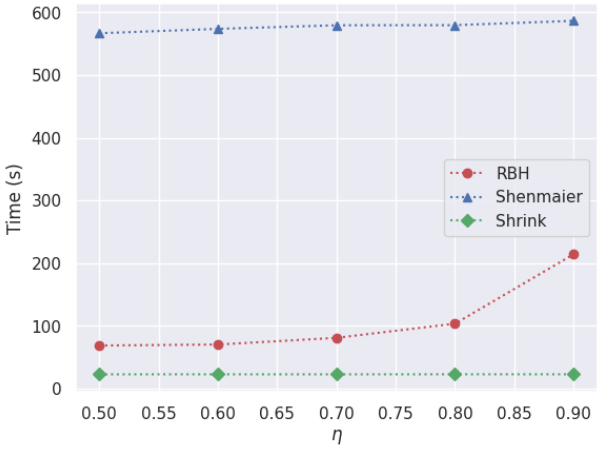
(c) Radius as a Function of d ($n = 10000$, $\eta = 0.9$)



(d) Runtime as a Function of d ($n = 10000$, $\eta = 0.9$)

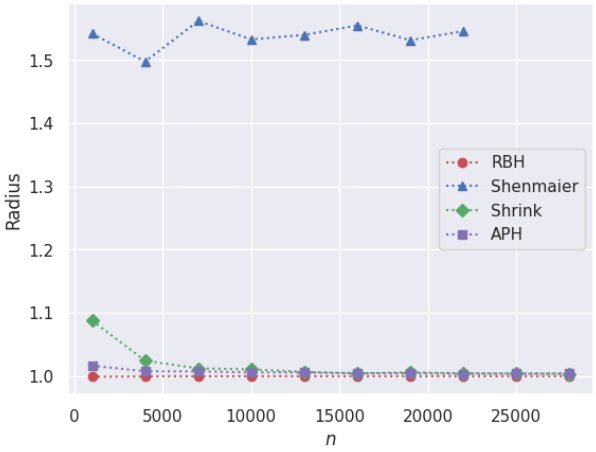


(e) Radius as a Function of η ($n = 10000$, $d = 30$)

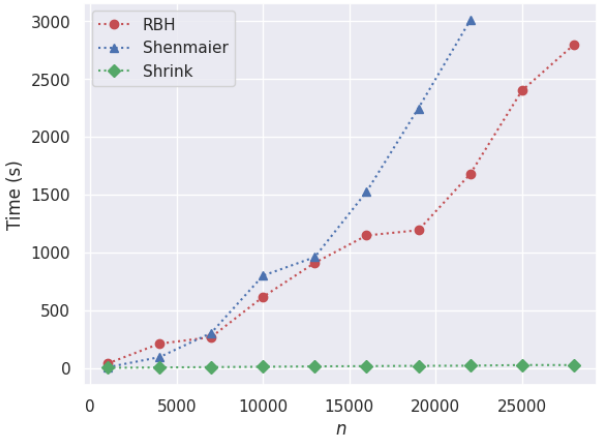


(f) Runtime as a Function of η ($n = 10000$, $d = 30$)

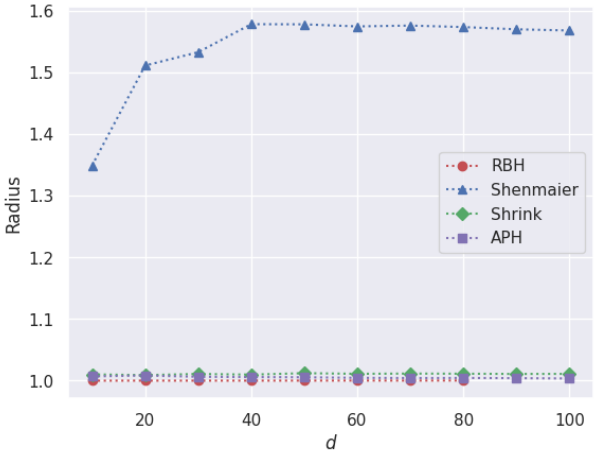
Figure 5.2.3: Results for each Algorithm on Hyperspherical Shell Data



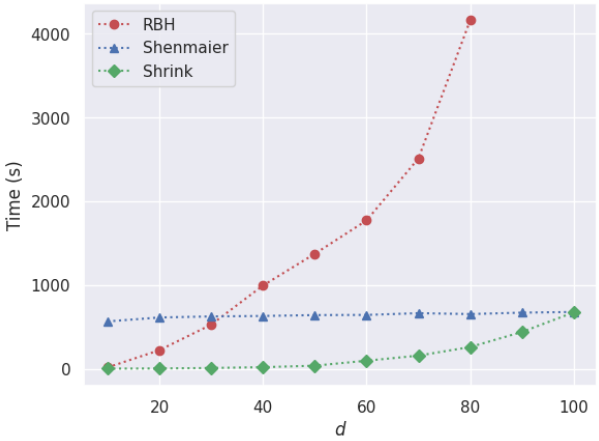
(a) Radius as a Function of n ($d = 30$, $\eta = 0.9$)



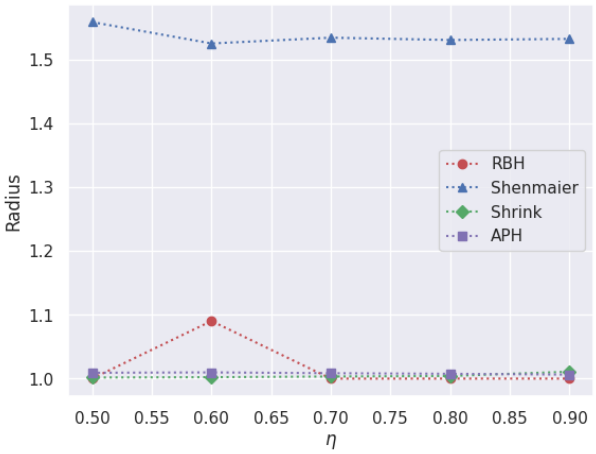
(b) Runtime as a Function of n ($d = 30$, $\eta = 0.9$)



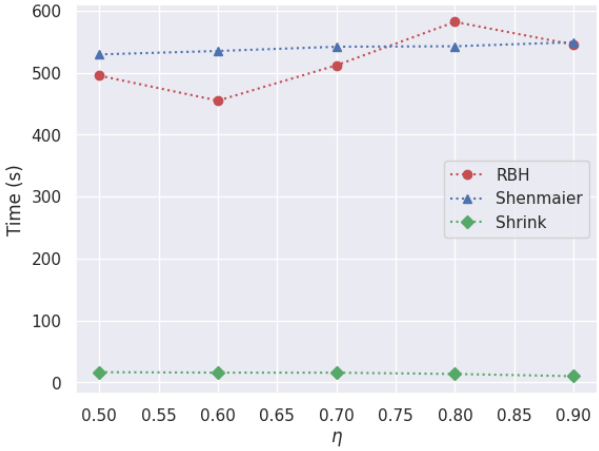
(c) Radius as a Function of d ($n = 10000$, $\eta = 0.9$)



(d) Runtime as a Function of d ($n = 10000$, $\eta = 0.9$)



(e) Radius as a Function of η ($n = 10000$, $d = 30$)



(f) Runtime as a Function of η ($n = 10000$, $d = 30$)

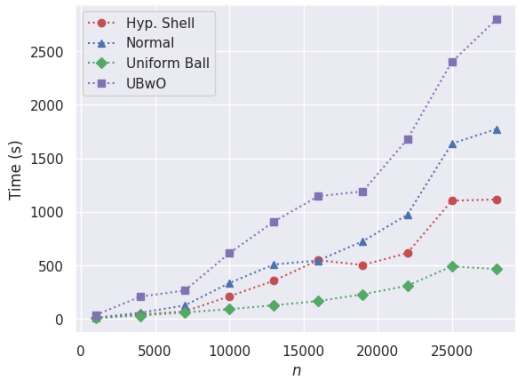
Figure 5.2.4: Results for each Algorithm on Uniform Ball with Outliers Data

By Algorithm

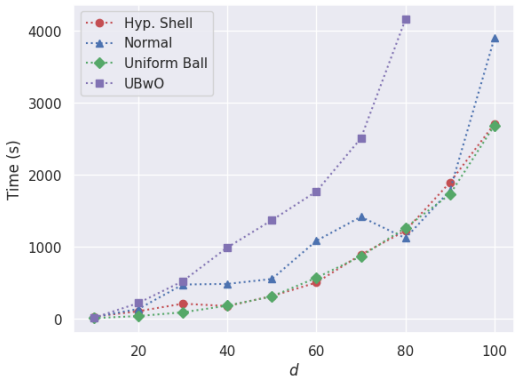
The complexity of the relaxation-based heuristic is hard to assess due to being solved by an optimization solver, so these plots (Figure 5.2.5) are able to give us some insight. With respect to n, we can observe a trend somewhere between linear and polynomial. With respect to d, the trend appears to be polynomial, and with respect to η there does not appear to be any strong trend.

We know that the average time complexity of Shenmaier's approximation will be, $O(n^2 d)$ which is fortunately exactly what we observe in Figure 5.2.6. We would expect no significant difference in runtimes when comparing data types since Shenmaier's approximation works by iterating over each point, but we do see slight differences in runtime then varying the dimension of the data. This could be explained by the sorting of distances which takes place when finding the k closest points, as this list will have a different distribution when sorted for each data type. The overall trend as dimension is increased is linear, but we observe a lesser increase in runtime than expected. This is likely because of optimizations made by NumPy, where the computations for arrays/vectors are written in C.

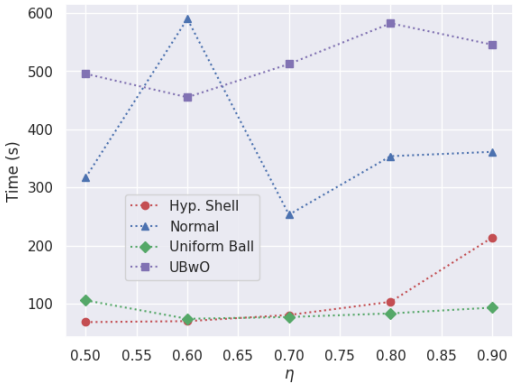
Recall that the shrink heuristic's time complexity depends on the algorithm used to solve the initial MEB problem, which in our implementation is Algorithm 1 from [19] with time complexity $O\left(\frac{nd}{\epsilon} + \frac{d^2}{\epsilon^{3/2}}\left(\frac{1}{\epsilon} + d\right)\log\frac{1}{\epsilon}\right)$. In Figure 5.2.7, we indeed see a linear relationship between runtime and n and a polynomial relationship between runtime and d. Interestingly, we have almost no relationship between runtime and d for normal data, and the reason for this is unclear. There is no relationship observed between runtime and η as expected, except for uniform ball with outliers data, which makes sense considering the structure of the data is dependent on η which evidently has an effect on the runtime of this algorithm, which works by finding a core-set for the data which we would assume is easier to find when there are fewer points in the outer shell.



(a) By n ($d = 30$, $\eta = 0.9$)

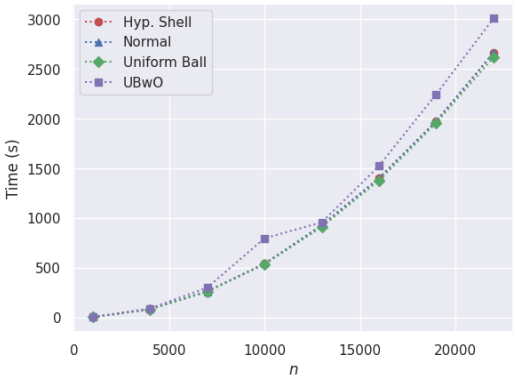


(b) By d ($n = 10000$, $\eta = 0.9$)

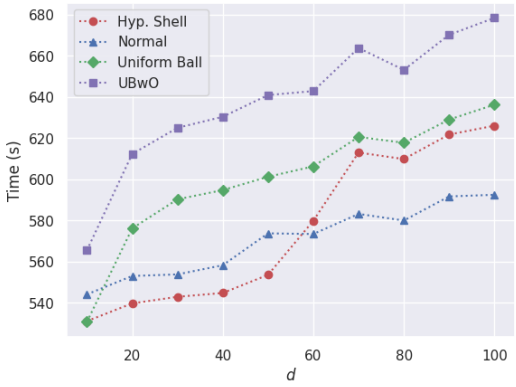


(c) By η ($n = 10000$, $d = 30$)

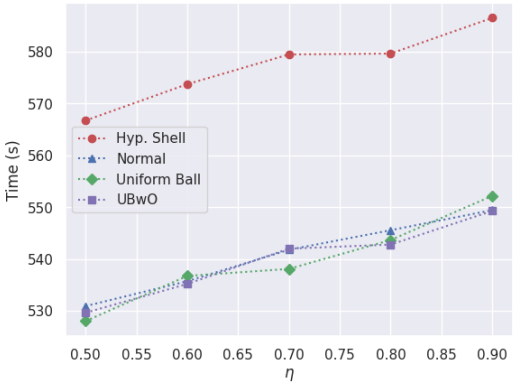
Figure 5.2.5: Runtimes for the Relaxation-Based Heuristic on each Data Type



(a) By n ($d = 30$, $\eta = 0.9$)

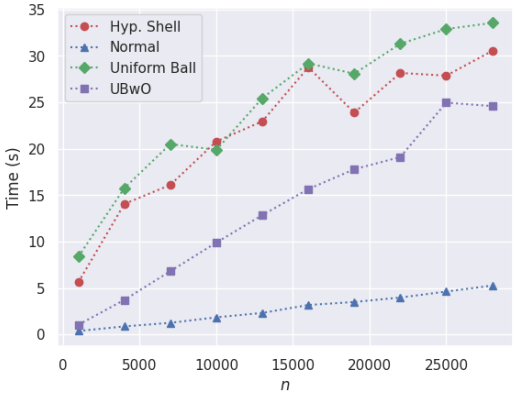


(b) By d ($n = 10000$, $\eta = 0.9$)

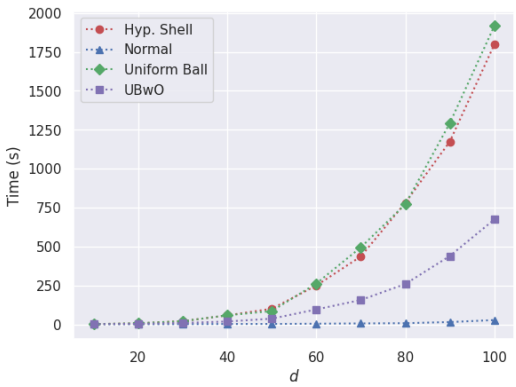


(c) By η ($n = 10000$, $d = 30$)

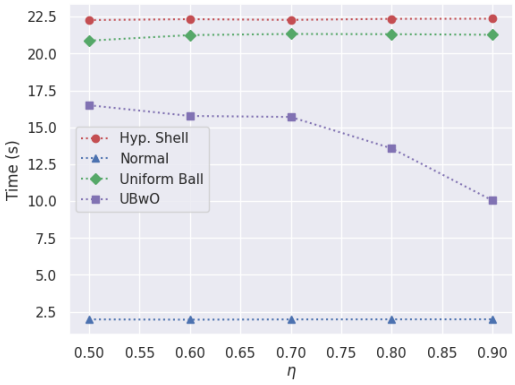
Figure 5.2.6: Runtimes for Shenmaier's Approximation on each Data Type



(a) By n ($d = 30$, $\eta = 0.9$)



(b) By d ($n = 10000$, $\eta = 0.9$)



(c) By η ($n = 10000$, $d = 30$)

Figure 5.2.7: Runtimes for the Shrink Heuristic on each Data Type

5.3 Improvement Heuristics

5.3.1 Methodology

Our interest in benchmarking our improvement heuristics is in how well these heuristics improve an existing solution. As discussed in Section 3.2 we know that both heuristics will run very quickly, however we are interested in the differences in runtimes between the heuristics. The DCSSH will run in polynomial time while the DCMEB is expected to run in polynomial time, as is typical of interior point methods, so we will benchmark the runtimes of the solver for this model.

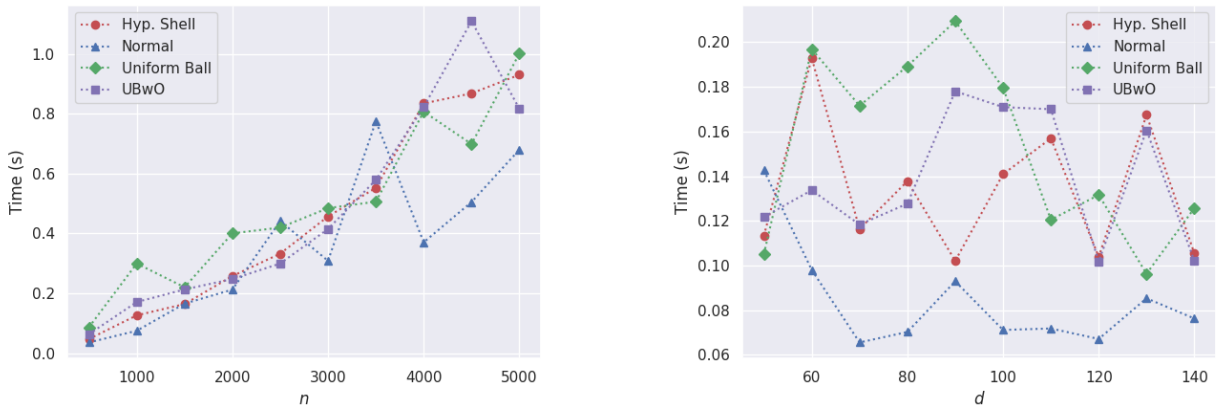
These heuristics were designed to be used after gaining a solution from Shenmaier's approximation, so we first find this solution for the given data set and then run an improvement heuristic 100 times. As discussed in Section 3.2 other stopping criteria could be implemented, but we opt for an iteration limit for ease of implementation and simplicity.

As seen in the previous section, Shenmaier's algorithm is not a particularly fast method, so with regard to the time constraint of this project we choose to benchmark our improvement heuristics on separate, smaller data sets than those we used to benchmark our construction methods. We follow the same methodology in generating and saving each data set. For benchmarking n we choose $n = 500$ to $n = 5000$ in steps of 500 with $d = 100$ fixed and for d we choose $d = 10$ to $d = 150$ in steps of 10 with $n = 1000$ fixed. The improvement heuristics only improve a ball on a given subset of data, so there is no η parameter to vary.

In each instance we solve, we record the initial radius returned by Shenmaier's approximation, and after running the improvement heuristic 100 times we record the new improved radius. For benchmarking the runtime of the solver for the DCMEB solver, we will record the runtime from the model instance after running Shenmaier's approximation. In the results section we report average improvement for each heuristic, where if r is the initial radius and $\hat{r}$ is the improved radius, the improvement percentage is calculated as $(1 - \hat{r}/r) \cdot 100$. Tabular results may be found in Appendix .3.

5.3.2 Results

DCMEB Runtime



(a) By n ($d = 100$)

(b) By d ($n = 1000$)

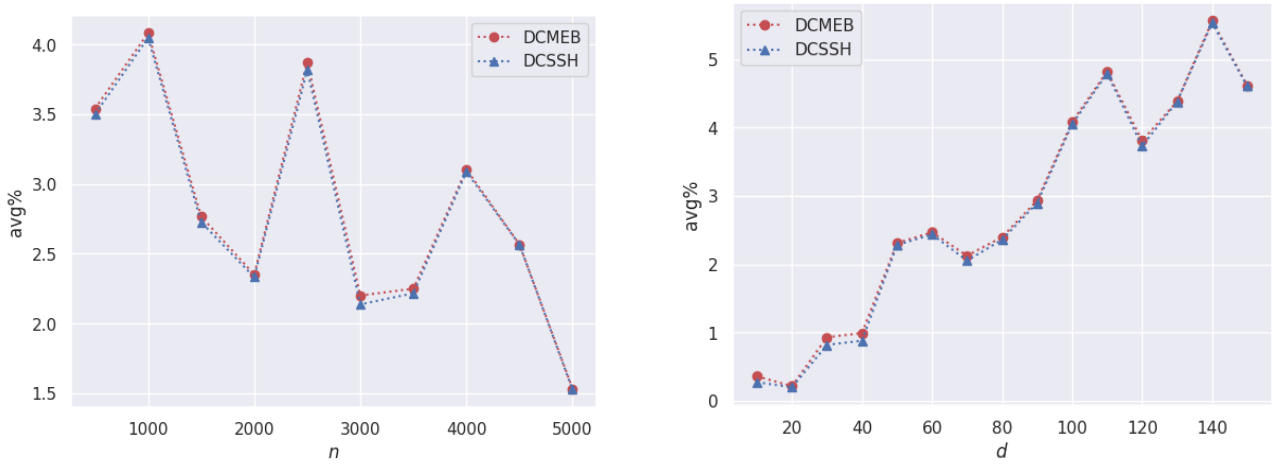
Figure 5.3.1: Runtimes for the DCMEB Model on each Data Type

From Figure 5.3.1 we see what appears to be a polynomial relationship between runtime and n as expected when using an interior point method, and no relationship between runtime and d. Although the longest that a single model took to solve was about 1.1 seconds, since this model is solved many times (100 in our case) this leads to non-insignificant runtimes when applying the heuristic, especially when compared to the DCSSH which runs in linear time.

Performance

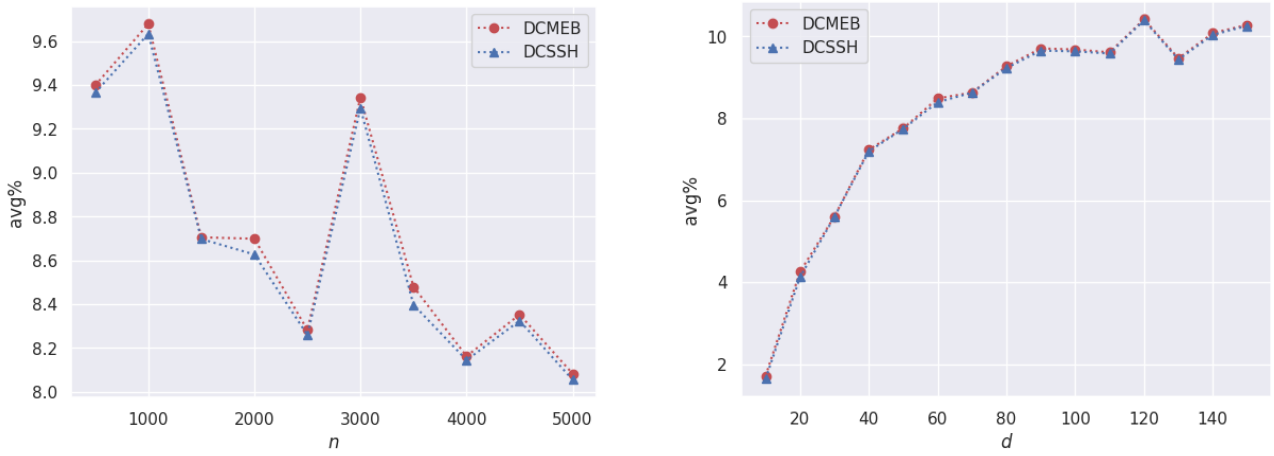
From the first three plots (Figures 5.3.2, 5.3.3, 5.3.4), we see the same trends, where the average improvement decreases as n increases and average improvement increases as d increases. This is likely due to more free space between points near the surface of the ball and that surface as dimension increases, while increasing the number of points will mean there are more points which are near the surface, thus giving a lower minimal distance between each point and the surface of the ball in the opposite improving direction. Interestingly, we observe the exact opposite trend for the uniform ball with outliers data in Figure 5.3.5, and it is uncertain why this is, though it is likely due to the unique structure of this data type.

Both heuristics give very similar performances, though the DCSSH appears to always be marginally better than the DCMEB. We can then conclude that, out of these two heuristics, one should always use the DCSSH over the DCMEB since the DCSSH runs in linear time as opposed to the polynomial time of the DCMEB, and for ease of implementation as the DCSSH does not require the use of an optimization solver which requires specific knowledge of how to use and in a commercial setting will in most cases require a payment of sorts.



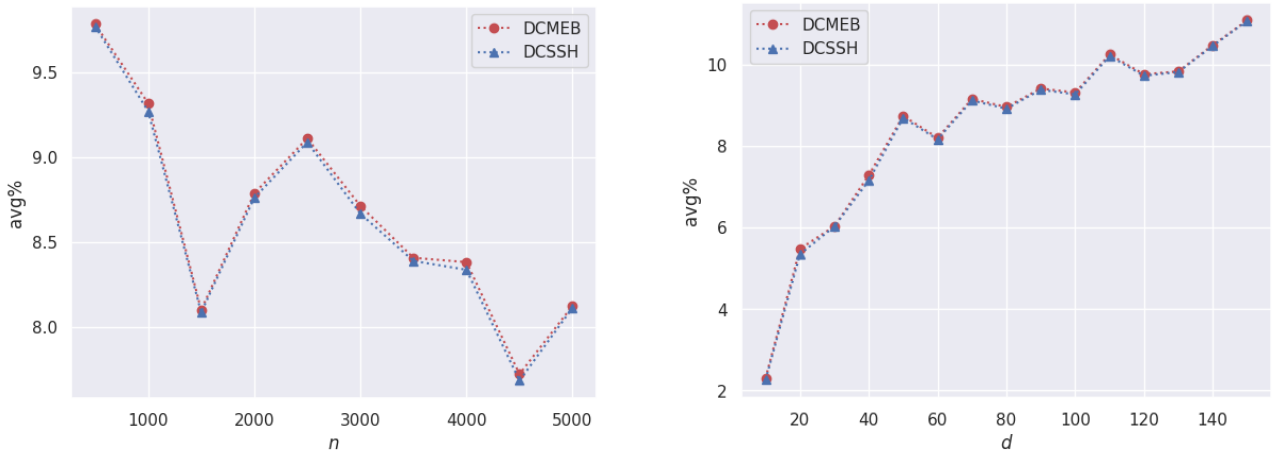
(a) By n (b) By d

Figure 5.3.2: Average Improvement Performance on Normal Data as a Function of n and d



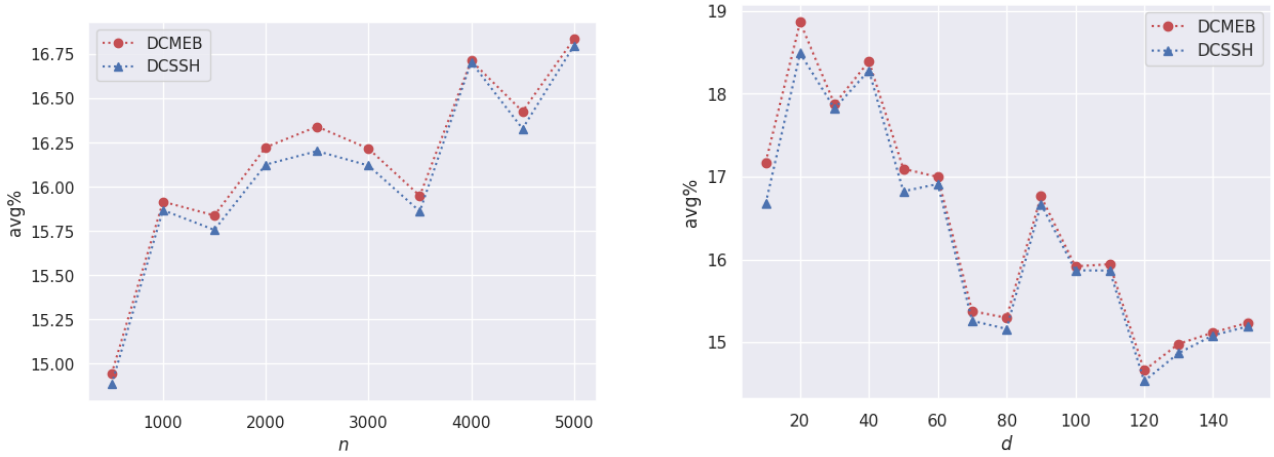
(a) By n (b) By d

Figure 5.3.3: Average Improvement Performance on Uniform Ball Data as a Function of n and d



(a) By n (b) By d

Figure 5.3.4: Average Improvement Performance on Hyperspherical Shell Data as a Function of n and d



(a) By n (b) By d

Figure 5.3.5: Average Improvement Performance on Uniform Ball with Outliers Data as a Function of n and d

5.4 Outlier Recognition

5.4.1 Methodology

We largely follow the methodology in [8, Section 4.2] by Hu Ding and Mingquan Ye for benchmarking their algorithm on the MNIST data set. For each digit 0–9, we construct a base data set by taking all points corresponding to that digit, then randomly sample points from the remaining digits as outliers. So, suppose we have n many points which are labelled 0, we then sample $(1-\eta)n$ many outliers from the digits 1–9. We use the entire 70000-point data set for this as we are not interested in a test-train split for an outlier recognition problem.

Once we have solved each instance, to measure the performance of the MEBwO as an outlier recognition model we calculate the F_1 score which is defined as

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

where the precision and recall are defined as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

and TP, FP, and FN are the number of true positives, false positives, and false negatives respectively found by the model. In our context, true positives are points which are inside the ball and are the desired digit, false positives are points which are inside the ball and are not the desired digit, and false negatives are points which are outside the ball and are the desired digit. The F_1 score is a common metric used to assess binary classification models such as outlier recognition models, and varies from 0 to 1 with an F_1 score of 1 indicating perfect performance.

For each method and each parameter, we solve the MEBwO problem for each digit and then report the average F_1 score across each digit. Due to the random sampling in constructing each instance we would prefer to report average performance for each digit across multiple instances, but due to time constraints this was not possible, and we hope that averaging across each digit provides enough robustness to the issues associated with random sampling.

The algorithm in [8] is compared against three outlier recognition methods, which are angle-based outlier detection (ABOD) [18], one-class support vector machine (OCSVM) [29], and discriminative reconstructions in an auto-encoder (DRAE) [39]. See [8] and each respective paper for further details. We provide an extension to Table 2 of [8], referring to their algorithm as HDMY. The total time reported is the total running time for each value of η and we assume this is averaged over each of their 20 instances. It should be noted that the time for sampling is included in the runtimes in their paper, which we do not do, but in our experiments we find this takes at most a few seconds, so the effect on the results is negligible considering the total runtime for each algorithm is reported on the order of hours. Additionally, a different computer is used to compute these results, and this an unavoidable (short of buying the same machine) limiting factor on the comparisons we may draw.

We present our results in tabular form as in [8] but also provide the results as a plot for the reader's enjoyment. It should be noted that HDMY and each of our algorithms are all MEBwO models, and we are comparing the effectiveness and runtime of each algorithm used to create the MEBwO. Unfortunately, Gurobi was unable to solve the relaxed model required for the relaxation-based heuristic, reaching the memory limit on our machine (16 GB) before the model could be instantiated.

5.4.2 Results

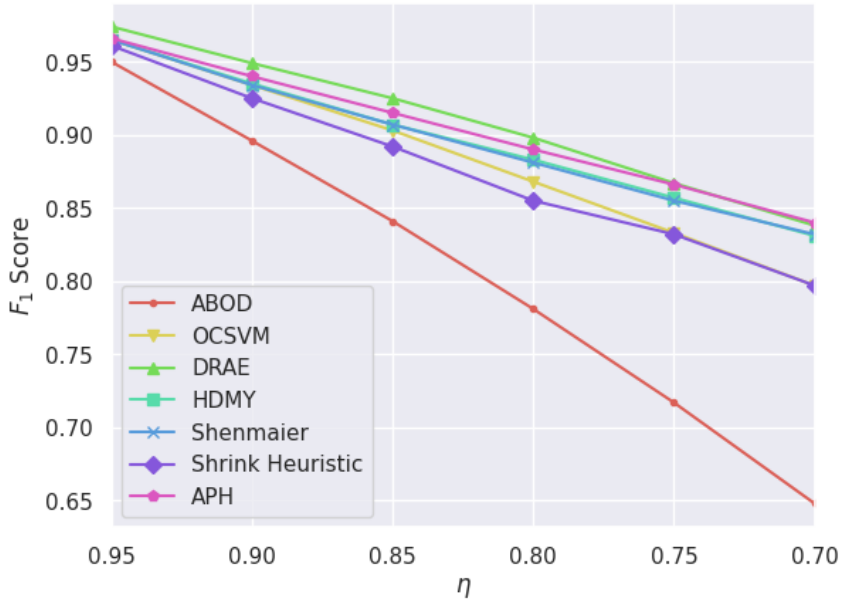


Figure 5.4.1: F_1 Score and Runtime of each Method for MNIST Outlier Recognition

η	ABOD	OCSVM	DRAE	HDMY	Shenmaier	Shrink Heuristic	APH
0.95	0.95	0.965	0.974	0.965	0.965	0.961	0.966
0.9	0.896	0.934	0.949	0.935	0.934	0.925	0.94
0.85	0.841	0.903	0.925	0.907	0.907	0.892	0.915
0.8	0.781	0.868	0.898	0.883	0.881	0.855	0.89
0.75	0.717	0.833	0.867	0.857	0.855	0.832	0.866
0.7	0.648	0.797	0.838	0.831	0.832	0.797	0.84
Runtime (s)	2×10^4	3.4×10^3	1×10^5	1.6×10^4	5.1×10^4	1.6×10^4	131.7

Table 5.1: F_1 Score and Runtime of each Method for MNIST Outlier Recognition

Comparing just our algorithms, both Shenmaier's approximation and the average point heuristic achieve similar performance, interestingly with the average point heuristic being marginally better—a surprise to be sure, but a welcome one, considering the very short runtime of the average point heuristic (two minutes compared to fourteen hours). The shrink heuristic finds worse performance than each of our other algorithms, despite taking four hours in total to run.

Each of our algorithms is able to outperform the ABOD model, and all but the shrink heuristic outperform the OCSVM model. Each algorithm gives worse performance than the DRAE model, which was also true for the algorithm in [8]. This algorithm gives similar performance to Shenmaier's approximation, but Shenmaier's approximation takes a little over three times as long to run. The average point heuristic is able to outperform HDMY which again is interesting considering the short runtime. It should again be noted that we were unable to run averages for these scores over multiple samplings of the MNIST data set, so it is possible that the average point heuristic on average has a worse performance than in our results, and we have randomly sampled a more favourable set of instances for this algorithm.

Chapter 6

Conclusion

6.1 Discussion

In this paper, we have explored the minimum enclosing ball with outliers problem and discussed the difficulties when trying to solve this problem optimally by way of an optimization solver. We were interested in the viability of alternative methods which may produce a feasible solution in a reasonable time. We have benchmarked these methods on various data and have recorded the runtimes and solutions when each parameter is varied. From this, we are able to draw conclusions about the effectiveness and viability of each method.

In our results, we consistently observe Shenmaier's approximation giving worse results than the other three algorithms, while often having longer runtimes. This is particularly true for large values of n, though Shenmaier's approximation scales linearly with d so for very high dimensional data relative to the number of data points this method may be preferable. The comparatively poor objective function value may be improved by applying an improvement heuristic in these cases.

The relaxation-based heuristic often performs well, and in some cases the best. This method appears to scale linearly with n, however we observe a polynomial relationship between runtime and d, with the worst case taking on average slightly over an hour to solve each iteration. Due to this, we may conclude that this method is preferable when the dimension of the data set is small.

In most cases, the shrink heuristic gives a similar objective function value to that of the relaxation-based heuristic, and giving better results on normal data for n above 4000. This makes a case for using the shrink heuristic instead of the relaxation-based heuristic, however the shrink heuristic is particularly susceptible to extreme outliers (that could be a result of measurement error, for example a faulty sensor) which would have a significant effect on the objective function value due to the impact of these outliers on the initial MEB. An investigation into data of this type would potentially reveal this difference, however under the time constraints of this project we were unable to give concrete evidence for this speculation.

The average point heuristic also gives similar objective function value, and for normal data even consistently giving the best value, while having near-instant runtimes. However, like with the shrink heuristic, extreme outliers could have a significant effect on the average point and thus the objective function value.

Our improvement algorithms yielded almost exactly the same results, with the DCSSH giving marginally better results over the DCMEB heuristic, and as discussed in Section 5.3.2 that one should always use the DCSSH due to the longer runtimes required to solve many DCMEB models and the requirement of an optimization solver.

We compared three of our methods to the tree-based construction method (HDMY) in [8] in an application to outlier recognition, and found that the balls returned by Shenmaier's approximation and the average point heuristic were able to provide similar performance to that of the HDMY method, though Shenmaier's approximation takes three times as long to run on this data and the average point heuristic takes very little time. Again, it should be noted that our algorithms were benchmarked on a different machine and were not run over multiple instances to reduce the impact of random sampling, so it would be unwise to draw concrete conclusions from these benchmarks.

6.2 Future Work

There are many directions for further research, one of note which may continue the work done in this paper is whether approximation methods exist which may solve the relaxed optimization model for the minimum enclosing ball with outliers problem within some approximation guarantee. Currently, this is a large bottleneck in both the runtime and the capabilities of the relaxation-based heuristic, seeing as the model was unable to be instantiated for the high-dimensional MNIST data with the available memory on our machine. An algorithm to solve this specific model in a similar manner as that of the simple minimum enclosing ball problem in [40] could enable the relaxation-based heuristic to tackle larger problems in less time.

Another interesting direction of research is in the development of improvement heuristics which are able to choose which points to include and exclude from the initial ball. This could work in multiple ways, for example by first finding a ball which excludes "obvious" outliers first, giving a ball that covers more than $\eta\%$ of the data, and then choosing points to remove which reduce the percentage of points covered until that percentage equals $\eta\%$. Another way would be to find an initial ball which does cover $\eta\%$ of the data, and then "swap" points which are considered by the model to be inliers and outliers in such a way that after each step the exact number of points inside the ball remains the same but the objective function value is reduced.

As we have seen, the minimum enclosing ball with outliers problem is not a simple one. However, we have also seen that it is not impossible to solve. In the big data era we live in, models and algorithms like those discussed in this paper are ever-present in today's society, with no sign of going away. When our work then has the effect on people and society as it does, it is imperative that we ensure these methods are applied effectively, robustly, and ethically. I am certain that, when adhering to this principle, ideas such as those in this paper and many more may be used to help improve the world and enhance the way in which we live.

Bibliography

[1] Alok Aggarwal, Hiroshi Imai, Naoki Katoh, and Subhash Suri. Finding k points with minimum diameter and related problems. *Journal of Algorithms*, 12(1):38–56, 1991.

[2] Erling D Andersen, Cornelis Roos, and Tamas Terlaky. On implementing a primal-dual interior-point method for conic quadratic optimization. *Mathematical Programming*, 95(2):249–277, 2003.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[4] G. E. P. Box and Mervin E. Muller. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, 29(2):610 – 611, 1958.

[5] Mihai Bundefineddoiu, Sariel Har-Peled, and Piotr Indyk. Approximate clustering via core-sets. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 250–257, New York, NY, USA, 2002. Association for Computing Machinery.

[6] M. Cavaleiro and F. Alizadeh. A branch-and-bound algorithm for the minimum radius k-enclosing ball problem. *ArXiv*, abs/1707.03387, 2017.

[7] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[8] Hu Ding and Mingquan Ye. Solving minimum enclosing ball with outliers: Algorithm, implementation, and application, April 2018.

[9] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

[10] Alon Efrat, Micha Sharir, and Alon Ziv. Computing the smallest k-enclosing circle and related problems. In Frank Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures*, pages 325–336, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[11] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021.

[12] Sariel Har-Peled and Soham Mazumdar. Fast algorithms for computing the smallest k-enclosing disc. In Giuseppe Di Battista and Uri Zwick, editors, *Algorithms - ESA 2003*, pages 278–288, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[13] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[14] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[15] Thomas Holmes. Minimum enclosing balls with outliers, 8 2021.

[16] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[17] Brian W Kernighan and Dennis M Ritchie. The c programming language, 2006.

[18] Hans-Peter Kriegel, Matthias Schubert, and Arthur Zimek. Angle-based outlier detection in high-dimensional data. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, page 444–452, New York, NY, USA, 2008. Association for Computing Machinery.

[19] Piyush Kumar, Joseph S. B. Mitchell, and E. Alper Yildirim. Approximate minimum enclosing balls in high dimensions using core-sets. *ACM J. Exp. Algorithmics*, 8:1.1–es, December 2004.

[20] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[21] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[22] MathWorks. Example a. volume of a hypersphere inscribed in a hypercube. Accessed 2021/08/04.

[23] Jiří Matoušek. On enclosing k points by a circle. *Information Processing Letters*, 53(4):217–221, 1995.

[24] Nimrod Megiddo. Linear-time algorithms for linear programming in r^3 and related problems. *SIAM Journal on Computing*, 12(4):759–776, 1983.

[25] Mervin E. Muller. A note on a method for generating points uniformly on ¡i¿n¡/i¿-dimensional spheres. *Commun. ACM*, 2(4):19–20, April 1959.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] Jeff Reback, jbrockmendel, Wes McKinney, Joris Van den Bossche, Tom Augspurger, Phillip Cloud, Simon Hawkins, gfyoung, Sinhrks, Matthew Roeschke, Adam Klein, Terji Petersen, Jeff Tratner, Chang She, William Ayd, Patrick Hoefler, Shahar Naveh, Marc Garcia, Jeremy Schendel, Andy Hayden, Daniel Saxton, Marco Edward Gorelli, Richard Shadrach, Vytautas Jancauskas, Ali McMaster, Fangchen Li, Pietro Battiston, Skipper Seabold, attack68, and Kaiqi Dong. pandas-dev/pandas: Pandas 1.3.0, July 2021.

[28] A. Rotenberg. A new pseudo-random number generator. *J. ACM*, 7(1):75–77, January 1960.

[29] Bernhard Schölkopf, Robert Williamson, Alex Smola, John Shawe-Taylor, and John Platt. Support vector method for novelty detection. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, page 582–588, Cambridge, MA, USA, 1999. MIT Press.

[30] Vladimir Shenmaier. Complexity and approximation of the smallest k-enclosing ball problem. *European Journal of Combinatorics*, 48:81–87, 2015. Selected Papers of EuroComb'13.

[31] TensorFlow Datasets, a collection of ready-to-use datasets. `https://www.tensorflow.org/datasets`.

[32] W. E. Thomson. A Modified Congruence Method of Generating Pseudo-random Numbers. *The Computer Journal*, 1(2):83–83, 01 1958.

[33] Nate Eldredge (`https://math.stackexchange.com/users/822/nate-eldredge`). Picking random points in the volume of sphere with uniform probability.

[34] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[35] Georges Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. premier mémoire. sur quelques propriétés des formes quadratiques positives parfaites. *Journal für die reine und angewandte Mathematik*, 133:97–178, 1908.

[36] Michael L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021.

[37] Eric W. Weisstein. Hypersphere. From MathWorld—A Wolfram Web Resource. Accessed 2021/08/04.

[38] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

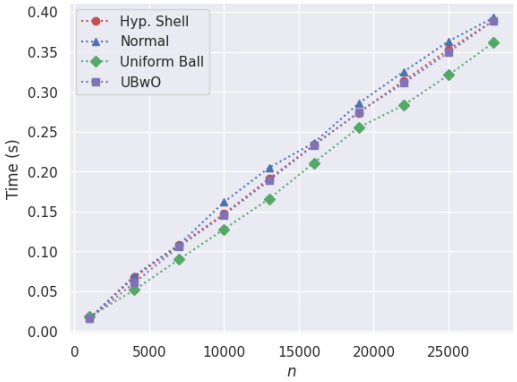
[39] Yan Xia, Xudong Cao, Fang Wen, Gang Hua, and Jian Sun. Learning discriminative reconstructions for unsupervised outlier removal. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[40] E. Alper Yildirim. Two algorithms for the minimum enclosing ball problem. *SIAM J. on Optimization*, 19(3):1368–1391, November 2008.

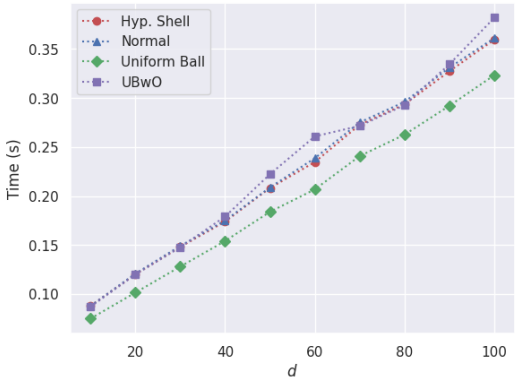
Appendices

.1 Average Point Heuristic Runtime Plots

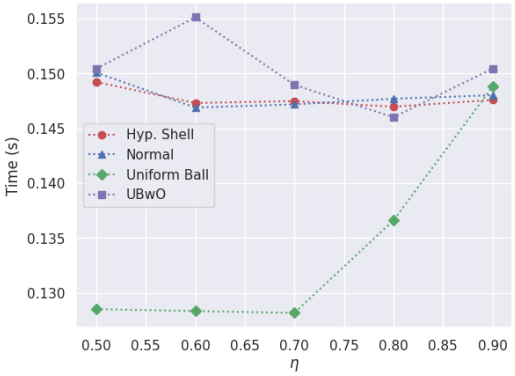
Here we present the runtime plots for the average point heuristic. Since this heuristic has a time complexity of $O(nd \log n)$ we know it will run extremely quickly, and so the runtimes were not included in the main text, but we include them here for completeness.



(a) By n ($d = 30$, $\eta = 0.9$)



(b) By d ($n = 10000$, $\eta = 0.9$)



(c) By η ($n = 10000$, $d = 30$)

Figure .1.1: Runtimes for the Average Point Heuristic on each Data Type

.2 Construction Method Tables

Here we present all average results for each method, parameter, and data type in tabular form. In order to fit the tables within the page width, values were rounded to two decimal places and method names in the column headers were shortened. For each individual result rather than rounded averages, please refer to the `benchmarks` folder in the GitHub repository[1].

The tables are organised by data type, and each table contains results for a single varied parameter, with each pair of columns the radius and runtime for each method. Runtimes are given in seconds.

.2.1 Normal

n	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
1000	6.37	12.4	7.34	5.38	6.52	0.37	6.32	0.02
4000	6.54	58.21	7.33	85.02	6.54	0.86	6.35	0.07
7000	6.75	127.11	7.29	262.29	6.57	1.25	6.35	0.11
10000	6.74	335.94	7.2	541.61	6.5	1.84	6.34	0.16
13000	6.84	509.54	7.19	932.43	6.51	2.31	6.35	0.2
16000	6.89	546.73	7.21	1403.68	6.49	3.17	6.34	0.24
19000	6.86	726.53	7.18	1978.03	6.5	3.49	6.35	0.29
22000	6.96	973.81	7.12	2664.51	6.48	3.97	6.34	0.33
25000	6.93	1637.88	—	—	6.47	4.61	6.34	0.36
28000	6.94	1774.19	—	—	6.46	5.26	6.35	0.39

Table 1: Results as a Function of n for each Algorithm on Normal Data

d	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
10	4.58	26.2	4.11	544.15	4.12	0.99	3.99	0.09
20	5.79	139.09	5.81	553.07	5.48	1.52	5.33	0.12
30	6.74	479.08	7.2	553.84	6.5	1.91	6.34	0.15
40	7.56	487.73	8.3	558.3	7.39	2.89	7.2	0.17
50	8.21	555.81	9.41	573.78	8.09	3.1	7.95	0.21
60	8.87	1086.75	10.27	573.47	8.81	4.79	8.62	0.24
70	9.46	1419.23	11.22	583.22	9.44	6.45	9.26	0.28
80	10.04	1122.77	11.99	580.02	10.04	7.95	9.83	0.3
90	10.51	1783.04	12.78	591.72	10.53	15.29	10.37	0.33
100	11.08	3902.45	13.42	592.52	11.07	28.02	10.88	0.36

Table 2: Results as a Function of d for each Algorithm on Normal Data

η	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
0.5	7.1	316.75	6.17	530.9	5.55	1.98	5.41	0.15
0.6	6.88	589.73	6.36	535.7	5.74	1.97	5.59	0.15
0.7	6.65	253.06	6.58	541.77	5.93	1.98	5.78	0.15
0.8	6.52	353.67	6.84	545.55	6.16	1.99	6.01	0.15
0.9	6.74	361.1	7.2	549.43	6.5	1.99	6.34	0.15

Table 3: Results as a Function of η for each Algorithm on Normal Data

[1] `https://github.com/tomholmes19/Minimum-Enclosing-Balls-with-Outliers` [15]

.2.2 Uniform Ball

n	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
1000	1.0	8.04	1.39	4.84	1.0	8.37	1.0	0.02
4000	1.0	32.25	1.37	84.61	1.0	15.76	1.0	0.05
7000	1.0	62.13	1.34	262.36	1.0	20.5	1.0	0.09
10000	1.0	92.07	1.34	536.42	1.0	19.87	1.0	0.13
13000	1.0	128.33	1.34	912.68	1.0	25.44	1.0	0.17
16000	1.0	168.09	1.34	1379.14	1.0	29.18	1.0	0.21
19000	1.0	231.08	1.34	1957.49	1.0	28.08	1.0	0.26
22000	1.0	310.96	1.34	2618.51	1.0	31.3	1.0	0.28
25000	1.0	492.28	—	—	1.0	32.89	1.0	0.32
28000	1.0	466.8	—	—	1.0	33.54	1.0	0.36

Table 4: Results as a Function of n for each Algorithm on Uniform Ball Data

d	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
10	0.99	11.42	1.14	530.74	0.99	2.08	0.99	0.07
20	0.99	41.34	1.29	576.21	0.99	7.91	0.99	0.1
30	1.0	93.18	1.34	590.32	1.0	20.88	1.0	0.13
40	1.0	185.42	1.38	594.8	1.0	58.73	1.0	0.15
50	1.0	313.6	1.39	601.28	1.0	85.84	1.0	0.18
60	1.0	572.94	1.4	606.33	1.0	262.42	1.0	0.21
70	1.0	871.02	1.41	620.63	1.0	494.76	1.0	0.24
80	1.0	1266.66	1.42	617.78	1.0	770.84	1.0	0.26
90	1.0	1730.96	1.42	629.06	1.0	1292.3	1.0	0.29
100	1.0	2689.86	1.43	636.3	1.0	1915.89	1.0	0.32

Table 5: Results as a Function of d for each Algorithm on Uniform Ball Data

η	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
0.5	0.98	105.74	1.2	528.0	0.98	20.87	0.98	0.13
0.6	0.98	73.97	1.23	536.72	0.98	21.25	0.98	0.13
0.7	0.99	77.04	1.26	538.08	0.99	21.33	0.99	0.13
0.8	0.99	83.24	1.29	543.65	0.99	21.31	0.99	0.14
0.9	1.0	93.22	1.34	552.18	1.0	21.27	1.0	0.15

Table 6: Results as a Function of η for each Algorithm on Uniform Ball Data

.2.3 Hyperspherical Shell

n	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
1000	1.99	8.61	2.78	5.27	1.99	5.62	1.99	0.02
4000	1.99	42.17	2.73	85.99	1.99	14.05	1.99	0.07
7000	1.99	70.81	2.71	266.54	1.99	16.1	1.99	0.11
10000	1.99	212.18	2.67	544.19	1.99	20.79	1.99	0.15
13000	1.99	357.39	2.66	917.93	1.99	22.94	1.99	0.19
16000	1.99	549.89	2.67	1401.93	1.99	28.76	1.99	0.23
19000	1.99	504.16	2.67	1974.51	1.99	23.89	1.99	0.27
22000	1.99	616.03	2.67	2660.51	1.99	28.16	1.99	0.31
25000	1.99	1104.94	—	—	1.99	27.87	1.99	0.35
28000	1.99	1116.81	—	—	1.99	30.53	1.99	0.39

Table 7: Results as a Function of n for each Algorithm on Hyperspherical Shell Data

d	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
10	1.98	32.05	2.46	531.14	1.98	2.31	1.98	0.09
20	1.99	108.26	2.59	539.75	1.99	6.62	1.99	0.12
30	1.99	212.84	2.67	542.96	1.99	21.2	1.99	0.15
40	1.99	179.66	2.76	544.81	1.99	58.01	1.99	0.17
50	2.0	317.79	2.79	553.62	2.0	100.81	2.0	0.21
60	2.0	506.3	2.8	579.88	2.0	249.42	2.0	0.23
70	2.0	893.33	2.82	613.06	2.0	436.27	2.0	0.27
80	2.0	1225.61	2.83	609.86	2.0	780.93	2.0	0.29
90	2.0	1896.26	2.85	621.8	2.0	1174.07	2.0	0.33
100	2.0	2710.25	2.86	626.08	2.0	1796.83	2.0	0.36

Table 8: Results as a Function of d for each Algorithm on Hyperspherical Shell Data

η	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
0.5	1.95	68.28	2.4	566.73	1.95	22.27	1.95	0.15
0.6	1.97	69.93	2.46	573.75	1.97	22.33	1.97	0.15
0.7	1.98	80.55	2.52	579.48	1.98	22.28	1.98	0.15
0.8	1.98	103.16	2.58	579.65	1.99	22.35	1.98	0.15
0.9	1.99	213.85	2.67	586.54	1.99	22.36	1.99	0.15

Table 9: Results as a Function of η for each Algorithm on Hyperspherical Shell Data

.2.4 Uniform Ball with Outliers

n	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
1000	1.0	37.9	1.54	5.8	1.09	1.04	1.02	0.02
4000	1.0	209.2	1.5	92.97	1.02	3.72	1.01	0.06
7000	1.0	266.03	1.56	302.46	1.01	6.81	1.01	0.11
10000	1.0	614.47	1.53	798.67	1.01	9.89	1.01	0.15
13000	1.0	909.65	1.54	958.93	1.01	12.85	1.01	0.19
16000	1.0	1146.58	1.55	1525.25	1.0	15.65	1.0	0.23
19000	1.0	1192.01	1.53	2247.41	1.01	17.79	1.0	0.27
22000	1.0	1677.9	1.55	3008.78	1.0	19.1	1.0	0.31
25000	1.0	2400.67	—	—	1.0	24.96	1.0	0.35
28000	1.0	2797.06	—	—	1.0	24.59	1.0	0.39

Table 10: Results as a Function of n for each Algorithm on Uniform Ball with Outliers Data

d	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
10	1.0	15.96	1.35	565.55	1.01	1.95	1.01	0.09
20	1.0	221.79	1.51	612.36	1.01	4.01	1.01	0.12
30	1.0	526.7	1.53	625.06	1.01	9.5	1.01	0.15
40	1.0	993.85	1.58	630.42	1.01	18.86	1.01	0.18
50	1.0	1372.71	1.58	640.97	1.01	36.98	1.01	0.22
60	1.0	1771.32	1.57	642.91	1.01	96.25	1.0	0.26
70	1.0	2510.24	1.58	663.83	1.01	156.8	1.0	0.27
80	1.0	4159.73	1.57	653.19	1.01	259.48	1.0	0.29
90	—	—	1.57	670.26	1.01	439.24	1.0	0.33
100	—	—	1.57	678.45	1.01	673.86	1.0	0.38

Table 11: Results as a Function of d for each Algorithm on Uniform Ball with Outliers Data

η	RBH r	RBH t	Shenm. r	Shenm. t	Shrink r	Shrink t	APH r	APH t
0.5	1.0	495.3	1.56	529.62	1.0	16.5	1.01	0.15
0.6	1.09	454.97	1.53	535.17	1.0	15.77	1.01	0.16
0.7	1.0	512.09	1.53	541.98	1.0	15.7	1.01	0.15
0.8	1.0	582.12	1.53	542.77	1.0	13.57	1.01	0.15
0.9	1.0	545.39	1.53	549.27	1.01	10.05	1.01	0.15

Table 12: Results as a Function of η for each Algorithm on Uniform Ball with Outliers Data

.3 Improvement Heuristic Tables

Here, we present results for each instance of benchmarking the improvement heuristic. We ran 5 separate instances for each particular benchmark, and r_i represents the initial radius of the ith instance returned by Shenmaier's approximation, while $\hat{r}_i$ represents the improved radius. The final column is the average improvement percentage, which is presented in the main text. Figures are rounded to two decimal places to fit within the page width, and the complete solutions may be found in the GitHub repository[2].

.3.1 Normal

n	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
500	13.2	12.98	13.92	13.11	13.8	13.3	13.79	13.49	13.62	13.03	3.54
1000	13.84	13.15	13.75	13.23	13.54	12.81	13.79	13.32	13.7	13.33	4.08
1500	13.78	13.36	13.5	13.31	13.56	13.21	13.67	13.25	13.72	13.2	2.77
2000	13.57	13.25	13.49	13.41	13.81	13.33	13.53	13.15	13.5	13.17	2.35
2500	13.56	12.99	13.51	13.1	13.69	13.09	13.42	13.03	13.71	13.05	3.87
3000	13.32	13.04	13.6	13.31	13.46	13.08	13.72	13.64	13.6	13.16	2.2
3500	13.52	13.25	13.61	13.21	13.33	13.03	13.53	13.18	13.35	13.15	2.25
4000	13.5	13.02	13.55	12.98	13.65	13.21	13.37	13.05	13.64	13.36	3.11
4500	13.52	13.09	13.61	13.26	13.52	13.24	13.66	13.43	13.69	13.23	2.57
5000	13.48	13.25	13.49	13.33	13.26	13.01	13.7	13.32	13.06	13.05	1.53

Table 13: Results as a Function of n for the DCMEB Heuristic on Normal Data

n	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
500	13.2	12.98	13.92	13.12	13.8	13.3	13.79	13.51	13.62	13.03	3.5
1000	13.84	13.15	13.75	13.24	13.54	12.81	13.79	13.33	13.7	13.33	4.05
1500	13.78	13.36	13.5	13.32	13.56	13.22	13.67	13.26	13.72	13.2	2.73
2000	13.57	13.26	13.49	13.41	13.81	13.33	13.53	13.15	13.5	13.17	2.33
2500	13.56	12.99	13.51	13.1	13.69	13.11	13.42	13.05	13.71	13.05	3.82
3000	13.32	13.05	13.6	13.32	13.46	13.1	13.72	13.64	13.6	13.16	2.14
3500	13.52	13.25	13.61	13.21	13.33	13.03	13.53	13.19	13.35	13.15	2.22
4000	13.5	13.02	13.55	12.99	13.65	13.21	13.37	13.05	13.64	13.36	3.09
4500	13.52	13.09	13.61	13.26	13.52	13.24	13.66	13.43	13.69	13.23	2.57
5000	13.48	13.25	13.49	13.33	13.26	13.01	13.7	13.32	13.06	13.05	1.53

Table 14: Results as a Function of n for the DCSSH on Normal Data

[2] `https://github.com/tomholmes19/Minimum-Enclosing-Balls-with-Outliers` [15]

d	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
10	4.24	4.24	4.11	4.07	4.29	4.28	4.28	4.26	4.27	4.25	0.36
20	6.04	6.03	5.91	5.89	6.06	6.05	6.11	6.11	6.02	6.0	0.21
30	7.49	7.44	7.39	7.3	7.35	7.32	7.5	7.38	7.29	7.24	0.93
40	8.61	8.5	8.55	8.42	8.67	8.58	8.42	8.36	8.49	8.45	0.99
50	9.77	9.45	9.81	9.54	9.88	9.65	9.64	9.53	9.34	9.16	2.31
60	10.4	10.04	10.72	10.2	10.59	10.31	10.38	10.3	10.43	10.37	2.47
70	11.46	11.29	11.51	11.13	11.46	11.06	11.43	11.4	11.35	11.11	2.13
80	12.06	11.95	12.26	11.83	12.25	12.07	12.16	11.94	12.3	11.78	2.4
90	12.92	12.38	12.99	12.81	12.98	12.67	13.12	12.58	13.04	12.71	2.93
100	13.84	13.15	13.75	13.23	13.54	12.81	13.79	13.32	13.7	13.33	4.08
110	14.26	13.61	14.45	13.87	14.34	13.55	14.56	13.87	14.5	13.74	4.82
120	14.9	14.32	14.74	14.05	15.2	14.61	15.07	14.61	15.04	14.51	3.81
130	15.84	15.06	15.79	15.28	15.47	14.83	15.89	14.9	15.74	15.2	4.39
140	16.1	15.25	16.26	15.07	16.28	15.61	16.51	15.46	16.19	15.41	5.56
150	16.86	16.31	16.72	15.94	16.69	15.87	16.82	16.03	16.6	15.67	4.62

Table 15: Results as a Function of d for the DCMEB Heuristic on Normal Data

d	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
10	4.24	4.24	4.11	4.09	4.29	4.28	4.28	4.27	4.27	4.25	0.27
20	6.04	6.03	5.91	5.89	6.06	6.06	6.11	6.11	6.02	6.0	0.2
30	7.49	7.45	7.39	7.3	7.35	7.32	7.5	7.4	7.29	7.25	0.82
40	8.61	8.52	8.55	8.44	8.67	8.59	8.42	8.37	8.49	8.45	0.88
50	9.77	9.45	9.81	9.54	9.88	9.65	9.64	9.54	9.34	9.16	2.28
60	10.4	10.04	10.72	10.21	10.59	10.32	10.38	10.3	10.43	10.37	2.44
70	11.46	11.31	11.51	11.13	11.46	11.06	11.43	11.4	11.35	11.13	2.05
80	12.06	11.95	12.26	11.84	12.25	12.07	12.16	11.94	12.3	11.79	2.36
90	12.92	12.39	12.99	12.82	12.98	12.67	13.12	12.58	13.04	12.71	2.9
100	13.84	13.15	13.75	13.24	13.54	12.81	13.79	13.33	13.7	13.33	4.05
110	14.26	13.61	14.45	13.87	14.34	13.57	14.56	13.87	14.5	13.74	4.8
120	14.9	14.32	14.74	14.06	15.2	14.65	15.07	14.62	15.04	14.51	3.73
130	15.84	15.06	15.79	15.28	15.47	14.84	15.89	14.9	15.74	15.2	4.38
140	16.1	15.26	16.26	15.08	16.28	15.61	16.51	15.46	16.19	15.42	5.53
150	16.86	16.31	16.72	15.94	16.69	15.87	16.82	16.04	16.6	15.67	4.61

Table 16: Results as a Function of d for the DCSSH on Normal Data

.3.2 Uniform Ball

n	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
500	1.44	1.31	1.45	1.31	1.45	1.29	1.41	1.29	1.42	1.3	9.4
1000	1.44	1.29	1.43	1.29	1.44	1.29	1.43	1.29	1.45	1.34	9.68
1500	1.44	1.3	1.43	1.3	1.42	1.31	1.44	1.31	1.42	1.31	8.7
2000	1.43	1.29	1.44	1.31	1.44	1.31	1.43	1.3	1.44	1.33	8.7
2500	1.43	1.31	1.44	1.33	1.43	1.33	1.44	1.31	1.42	1.29	8.28
3000	1.43	1.3	1.44	1.31	1.44	1.3	1.44	1.3	1.42	1.3	9.34
3500	1.43	1.32	1.44	1.33	1.44	1.31	1.43	1.3	1.44	1.31	8.48
4000	1.43	1.32	1.43	1.31	1.43	1.31	1.43	1.31	1.43	1.33	8.16
4500	1.44	1.31	1.44	1.31	1.43	1.32	1.42	1.29	1.43	1.33	8.35
5000	1.43	1.31	1.44	1.33	1.43	1.31	1.43	1.3	1.43	1.33	8.08

Table 17: Results as a Function of n for the DCMEB Heuristic on Uniform Ball Data

n	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
500	1.44	1.31	1.45	1.32	1.45	1.29	1.41	1.29	1.42	1.3	9.36
1000	1.44	1.29	1.43	1.29	1.44	1.29	1.43	1.29	1.45	1.34	9.63
1500	1.44	1.3	1.43	1.3	1.42	1.31	1.44	1.31	1.42	1.31	8.7
2000	1.43	1.3	1.44	1.31	1.44	1.31	1.43	1.3	1.44	1.33	8.63
2500	1.43	1.31	1.44	1.33	1.43	1.33	1.44	1.31	1.42	1.29	8.26
3000	1.43	1.31	1.44	1.31	1.44	1.3	1.44	1.3	1.42	1.3	9.29
3500	1.43	1.32	1.44	1.33	1.44	1.31	1.43	1.31	1.44	1.31	8.4
4000	1.43	1.32	1.43	1.31	1.43	1.31	1.43	1.31	1.43	1.33	8.14
4500	1.44	1.31	1.44	1.31	1.43	1.32	1.42	1.29	1.43	1.34	8.32
5000	1.43	1.31	1.44	1.33	1.43	1.31	1.43	1.3	1.43	1.33	8.05

Table 18: Results as a Function of n for the DCSSH on Uniform Ball Data

d	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
10	1.21	1.2	1.19	1.17	1.25	1.21	1.26	1.23	1.2	1.2	1.72
20	1.33	1.27	1.36	1.3	1.3	1.24	1.3	1.26	1.36	1.29	4.25
30	1.38	1.29	1.39	1.32	1.38	1.31	1.39	1.32	1.39	1.31	5.6
40	1.42	1.29	1.42	1.32	1.41	1.28	1.38	1.32	1.41	1.31	7.23
50	1.43	1.29	1.42	1.31	1.43	1.32	1.41	1.32	1.43	1.31	7.75
60	1.41	1.29	1.41	1.3	1.44	1.3	1.42	1.3	1.44	1.32	8.48
70	1.43	1.3	1.43	1.31	1.44	1.33	1.45	1.32	1.43	1.31	8.63
80	1.42	1.3	1.43	1.29	1.44	1.3	1.42	1.29	1.45	1.32	9.27
90	1.45	1.31	1.44	1.29	1.45	1.32	1.44	1.29	1.45	1.31	9.7
100	1.44	1.29	1.43	1.29	1.44	1.29	1.43	1.29	1.45	1.34	9.68
110	1.45	1.28	1.44	1.32	1.44	1.3	1.44	1.32	1.44	1.3	9.61
120	1.45	1.29	1.45	1.29	1.45	1.3	1.44	1.3	1.45	1.29	10.41
130	1.45	1.3	1.43	1.28	1.45	1.32	1.45	1.31	1.45	1.34	9.47
140	1.44	1.32	1.45	1.29	1.44	1.29	1.43	1.29	1.45	1.29	10.08
150	1.45	1.29	1.45	1.3	1.44	1.29	1.45	1.3	1.44	1.31	10.28

Table 19: Results as a Function of d for the DCMEB Heuristic on Uniform Ball Data

d	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
10	1.21	1.2	1.19	1.18	1.25	1.21	1.26	1.23	1.2	1.2	1.64
20	1.33	1.27	1.36	1.3	1.3	1.25	1.3	1.26	1.36	1.29	4.12
30	1.38	1.29	1.39	1.32	1.38	1.31	1.39	1.32	1.39	1.31	5.6
40	1.42	1.29	1.42	1.32	1.41	1.28	1.38	1.32	1.41	1.31	7.2
50	1.43	1.29	1.42	1.31	1.43	1.32	1.41	1.32	1.43	1.31	7.74
60	1.41	1.29	1.41	1.3	1.44	1.3	1.42	1.3	1.44	1.32	8.39
70	1.43	1.3	1.43	1.31	1.44	1.33	1.45	1.32	1.43	1.31	8.62
80	1.42	1.3	1.43	1.29	1.44	1.3	1.42	1.29	1.45	1.32	9.23
90	1.45	1.31	1.44	1.29	1.45	1.32	1.44	1.3	1.45	1.31	9.65
100	1.44	1.29	1.43	1.29	1.44	1.29	1.43	1.29	1.45	1.34	9.63
110	1.45	1.28	1.44	1.32	1.44	1.3	1.44	1.32	1.44	1.31	9.58
120	1.45	1.29	1.45	1.29	1.45	1.3	1.44	1.3	1.45	1.29	10.39
130	1.45	1.3	1.43	1.28	1.45	1.32	1.45	1.31	1.45	1.34	9.43
140	1.44	1.32	1.45	1.29	1.44	1.29	1.43	1.29	1.45	1.29	10.04
150	1.45	1.29	1.45	1.3	1.44	1.29	1.45	1.3	1.44	1.31	10.26

Table 20: Results as a Function of d for the DCSSH on Uniform Ball Data

.3.3 Hyperspherical Shell

n	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
500	2.91	2.57	2.9	2.64	2.86	2.57	2.87	2.61	2.87	2.61	9.78
1000	2.88	2.59	2.87	2.62	2.89	2.64	2.91	2.63	2.87	2.6	9.32
1500	2.88	2.63	2.89	2.7	2.88	2.63	2.9	2.65	2.84	2.62	8.1
2000	2.85	2.61	2.87	2.64	2.84	2.59	2.89	2.63	2.9	2.62	8.79
2500	2.86	2.61	2.86	2.6	2.9	2.63	2.87	2.59	2.85	2.61	9.11
3000	2.86	2.6	2.87	2.62	2.86	2.62	2.83	2.61	2.88	2.61	8.71
3500	2.86	2.65	2.87	2.64	2.89	2.61	2.86	2.61	2.88	2.64	8.41
4000	2.87	2.6	2.83	2.6	2.85	2.63	2.87	2.65	2.88	2.64	8.38
4500	2.87	2.64	2.87	2.61	2.85	2.6	2.83	2.68	2.88	2.66	7.72
5000	2.86	2.61	2.86	2.62	2.83	2.65	2.87	2.62	2.86	2.62	8.12

Table 21: Results as a Function of n for the DCMEB Heuristic on Hyperspherical Shell Data

n	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
500	2.91	2.57	2.9	2.64	2.86	2.57	2.87	2.61	2.87	2.61	9.77
1000	2.88	2.59	2.87	2.63	2.89	2.64	2.91	2.63	2.87	2.6	9.27
1500	2.88	2.63	2.89	2.7	2.88	2.63	2.9	2.65	2.84	2.62	8.08
2000	2.85	2.61	2.87	2.64	2.84	2.59	2.89	2.63	2.9	2.62	8.76
2500	2.86	2.61	2.86	2.6	2.9	2.63	2.87	2.59	2.85	2.61	9.09
3000	2.86	2.6	2.87	2.63	2.86	2.62	2.83	2.61	2.88	2.61	8.67
3500	2.86	2.65	2.87	2.64	2.89	2.61	2.86	2.62	2.88	2.64	8.39
4000	2.87	2.6	2.83	2.6	2.85	2.63	2.87	2.65	2.88	2.64	8.34
4500	2.87	2.65	2.87	2.61	2.85	2.6	2.83	2.68	2.88	2.66	7.68
5000	2.86	2.61	2.86	2.62	2.83	2.65	2.87	2.62	2.86	2.62	8.11

Table 22: Results as a Function of n for the DCSSH on Hyperspherical Shell Data

d	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
10	2.64	2.59	2.51	2.39	2.47	2.45	2.53	2.48	2.47	2.41	2.3
20	2.72	2.58	2.7	2.56	2.66	2.54	2.74	2.57	2.74	2.56	5.48
30	2.8	2.61	2.78	2.64	2.83	2.71	2.74	2.5	2.75	2.61	6.03
40	2.77	2.57	2.82	2.55	2.67	2.56	2.81	2.56	2.79	2.61	7.28
50	2.86	2.63	2.86	2.63	2.88	2.62	2.89	2.58	2.85	2.61	8.74
60	2.88	2.61	2.86	2.65	2.88	2.64	2.85	2.62	2.85	2.62	8.2
70	2.87	2.58	2.85	2.57	2.86	2.6	2.88	2.63	2.87	2.64	9.16
80	2.88	2.59	2.87	2.59	2.87	2.63	2.88	2.65	2.85	2.6	8.96
90	2.9	2.64	2.87	2.57	2.86	2.58	2.86	2.59	2.89	2.64	9.41
100	2.88	2.59	2.87	2.62	2.89	2.64	2.91	2.63	2.87	2.6	9.32
110	2.9	2.57	2.86	2.57	2.88	2.64	2.89	2.56	2.89	2.61	10.25
120	2.88	2.61	2.88	2.59	2.87	2.6	2.88	2.64	2.91	2.58	9.76
130	2.91	2.63	2.9	2.62	2.86	2.58	2.91	2.61	2.9	2.6	9.84
140	2.91	2.59	2.88	2.56	2.9	2.61	2.89	2.58	2.9	2.61	10.48
150	2.89	2.58	2.89	2.59	2.9	2.57	2.91	2.59	2.89	2.55	11.09

Table 23: Results as a Function of d for the DCMEB Heuristic on Hyperspherical Shell Data

d	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
10	2.64	2.59	2.51	2.39	2.47	2.45	2.53	2.48	2.47	2.41	2.26
20	2.72	2.58	2.7	2.57	2.66	2.54	2.74	2.57	2.74	2.56	5.35
30	2.8	2.61	2.78	2.64	2.83	2.71	2.74	2.5	2.75	2.61	6.02
40	2.77	2.58	2.82	2.55	2.67	2.57	2.81	2.56	2.79	2.61	7.15
50	2.86	2.63	2.86	2.63	2.88	2.62	2.89	2.59	2.85	2.61	8.69
60	2.88	2.61	2.86	2.65	2.88	2.64	2.85	2.62	2.85	2.62	8.17
70	2.87	2.58	2.85	2.57	2.86	2.6	2.88	2.63	2.87	2.64	9.12
80	2.88	2.59	2.87	2.6	2.87	2.63	2.88	2.66	2.85	2.6	8.92
90	2.9	2.64	2.87	2.57	2.86	2.58	2.86	2.59	2.89	2.64	9.38
100	2.88	2.59	2.87	2.63	2.89	2.64	2.91	2.63	2.87	2.6	9.27
110	2.9	2.57	2.86	2.58	2.88	2.64	2.89	2.56	2.89	2.61	10.21
120	2.88	2.61	2.88	2.59	2.87	2.6	2.88	2.64	2.91	2.58	9.73
130	2.91	2.63	2.9	2.62	2.86	2.58	2.91	2.61	2.9	2.6	9.82
140	2.91	2.59	2.88	2.56	2.9	2.61	2.89	2.58	2.9	2.62	10.46
150	2.89	2.58	2.89	2.59	2.9	2.57	2.91	2.59	2.89	2.55	11.07

Table 24: Results as a Function of d for the DCSSH on Hyperspherical Shell Data

.3.4 Uniform Ball with Outliers

n	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
500	1.53	1.29	1.53	1.3	1.51	1.3	1.53	1.3	1.53	1.31	14.94
1000	1.54	1.3	1.55	1.3	1.55	1.31	1.54	1.3	1.53	1.28	15.92
1500	1.54	1.3	1.55	1.28	1.55	1.32	1.55	1.3	1.54	1.32	15.84
2000	1.55	1.29	1.55	1.32	1.53	1.3	1.54	1.28	1.55	1.27	16.22
2500	1.56	1.29	1.54	1.32	1.54	1.29	1.55	1.29	1.56	1.29	16.34
3000	1.55	1.32	1.56	1.3	1.54	1.28	1.53	1.29	1.55	1.29	16.22
3500	1.55	1.3	1.57	1.35	1.57	1.31	1.55	1.3	1.56	1.31	15.95
4000	1.57	1.3	1.56	1.3	1.56	1.3	1.56	1.28	1.54	1.31	16.71
4500	1.56	1.29	1.57	1.33	1.56	1.3	1.56	1.3	1.56	1.32	16.43
5000	1.52	1.26	1.56	1.29	1.54	1.28	1.54	1.3	1.55	1.29	16.83

Table 25: Results as a Function of n for the DCMEB Heuristic on Uniform Ball with Outliers Data

n	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
500	1.53	1.29	1.53	1.31	1.51	1.3	1.53	1.3	1.53	1.31	14.88
1000	1.54	1.3	1.55	1.3	1.55	1.31	1.54	1.3	1.53	1.28	15.87
1500	1.54	1.3	1.55	1.28	1.55	1.32	1.55	1.3	1.54	1.32	15.76
2000	1.55	1.29	1.55	1.32	1.53	1.3	1.54	1.29	1.55	1.28	16.13
2500	1.56	1.29	1.54	1.32	1.54	1.3	1.55	1.29	1.56	1.3	16.2
3000	1.55	1.32	1.56	1.3	1.54	1.28	1.53	1.29	1.55	1.3	16.12
3500	1.55	1.3	1.57	1.35	1.57	1.31	1.55	1.3	1.56	1.31	15.86
4000	1.57	1.3	1.56	1.3	1.56	1.3	1.56	1.28	1.54	1.31	16.71
4500	1.56	1.29	1.57	1.33	1.56	1.3	1.56	1.3	1.56	1.32	16.33
5000	1.52	1.26	1.56	1.29	1.54	1.28	1.54	1.3	1.55	1.29	16.8

Table 26: Results as a Function of n for the DCSSH on Uniform Ball with Outliers Data

d	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
10	1.32	1.13	1.47	1.18	1.37	1.13	1.42	1.18	1.43	1.18	17.17
20	1.51	1.27	1.54	1.23	1.5	1.25	1.55	1.19	1.57	1.28	18.86
30	1.54	1.31	1.53	1.24	1.59	1.26	1.56	1.25	1.57	1.32	17.88
40	1.58	1.31	1.57	1.28	1.53	1.3	1.56	1.23	1.59	1.27	18.39
50	1.57	1.33	1.58	1.27	1.53	1.3	1.54	1.29	1.55	1.26	17.09
60	1.55	1.31	1.53	1.27	1.52	1.25	1.57	1.29	1.57	1.31	17.0
70	1.56	1.32	1.56	1.31	1.55	1.28	1.53	1.32	1.53	1.32	15.38
80	1.55	1.27	1.51	1.32	1.55	1.34	1.55	1.3	1.56	1.31	15.3
90	1.55	1.3	1.55	1.28	1.56	1.29	1.54	1.29	1.56	1.29	16.76
100	1.54	1.3	1.55	1.3	1.55	1.31	1.54	1.3	1.53	1.28	15.92
110	1.52	1.3	1.54	1.29	1.53	1.28	1.55	1.29	1.54	1.3	15.94
120	1.54	1.32	1.52	1.26	1.55	1.31	1.54	1.36	1.53	1.29	14.66
130	1.54	1.31	1.54	1.32	1.54	1.31	1.53	1.29	1.54	1.31	14.98
140	1.52	1.34	1.52	1.3	1.54	1.3	1.53	1.28	1.54	1.27	15.12
150	1.53	1.29	1.53	1.3	1.53	1.3	1.54	1.29	1.53	1.3	15.23

Table 27: Results as a Function of d for the DCMEB Heuristic on Uniform Ball with Outliers Data

d	r_1	$\hat{r}_1$	r_2	$\hat{r}_2$	r_3	$\hat{r}_3$	r_4	$\hat{r}_4$	r_5	$\hat{r}_5$	Avg%
10	1.32	1.14	1.47	1.18	1.37	1.13	1.42	1.21	1.43	1.18	16.68
20	1.51	1.27	1.54	1.23	1.5	1.25	1.55	1.21	1.57	1.29	18.49
30	1.54	1.31	1.53	1.24	1.59	1.26	1.56	1.26	1.57	1.32	17.82
40	1.58	1.31	1.57	1.28	1.53	1.3	1.56	1.24	1.59	1.27	18.28
50	1.57	1.33	1.58	1.27	1.53	1.31	1.54	1.29	1.55	1.27	16.82
60	1.55	1.31	1.53	1.27	1.52	1.25	1.57	1.29	1.57	1.31	16.91
70	1.56	1.32	1.56	1.31	1.55	1.28	1.53	1.32	1.53	1.32	15.26
80	1.55	1.27	1.51	1.32	1.55	1.34	1.55	1.31	1.56	1.31	15.16
90	1.55	1.3	1.55	1.29	1.56	1.29	1.54	1.29	1.56	1.29	16.67
100	1.54	1.3	1.55	1.3	1.55	1.31	1.54	1.3	1.53	1.28	15.87
110	1.52	1.3	1.54	1.29	1.53	1.28	1.55	1.29	1.54	1.3	15.86
120	1.54	1.33	1.52	1.27	1.55	1.31	1.54	1.36	1.53	1.29	14.53
130	1.54	1.31	1.54	1.32	1.54	1.31	1.53	1.29	1.54	1.31	14.87
140	1.52	1.34	1.52	1.3	1.54	1.3	1.53	1.28	1.54	1.27	15.08
150	1.53	1.29	1.53	1.3	1.53	1.3	1.54	1.29	1.53	1.3	15.19

Table 28: Results as a Function of d for the DCSSH on Uniform Ball with Outliers Data