*Division of Pharmacoepidemiology*
*And Pharmacoeconomics*
Technical Report Series

*Year:* 2012 #005

# KD-Tree Algorithm for Propensity Score Matching With Three or More Treatment Groups

John R. Hott[a], Nathan Brunelle[a], Jessica A. Myers[b], Jeremy Rassen[b], abhi shelat[a]

a.) Department of Computer Science, University of Virginia, Charlottesville, VA
b.) Division of Pharmacoepidemiology and Pharmacoeconomics, Department of Medicine, Brigham and Women's Hospital and Harvard Medical School, Boston, MA

**Series Editors:**

Sebastian Schneeweiss, MD, ScD
Jerry Avorn, MD
Robert J. Glynn, ScD, PhD
Niteesh K. Choudhry, MD, PhD
Jeremy A. Rassen, ScD
Josh Gagne, PharmD, ScD

**Contact Information:**

Division of Pharmacoepidemiology and Pharmacoeconomics
Department of Medicine
Brigham and Women's Hospital and Harvard Medical School
1620 Tremont St., Suite 3030
Boston, MA 02120
Tel: 616-278-0930 Fax: 617-232-8602

# KD-Tree Algorithm for Propensity Score Matching With Three or More Treatment Groups

John R Hott[a], Nathan Brunelle[a], Jessica A Myers[b], Jeremy Rassen[b], abhi shelat[a]

[a]*Department of Computer Science, University of Virginia*
[b]*Brigham and Women's Hospital and Harvard Medical School*

**Abstract**

Propensity scores (PS) have been widely used in epidemiology to control for confounding bias in non-experimental comparative studies of drugs, but the technique of matching patients by score becomes computationally impractical with studies of three or more treatment groups. Imbens' generalized propensity score (GPS) provides a method for comparing multiple treatments through regression, weighting, and other approaches. We present a multi-category matching algorithm that matches patients across multiple groups under any number of normalized factors, including the PS and GPS. The algorithm's time complexity is expected-case quadratic on the number of participants per group.

Utilizing kd-tree data structures to provide efficient queries for nearby points and a search radius related to a best-guess match between participants in each treatment group, we reduce the number of participants that must be considered for each matching. We then match patients by the $k$ factors defined, balancing the distribution of confounders in each treatment group and thereby removing bias. Our algorithm outperforms the brute force matching approach in the expected case, requiring only $O(n)$ space and $O(n^2)$ time compared with brute force's $O(n^{k+1})$ time, for $k$ treatment groups. This difference is clearly seen in our simulation study of 1000 participants in 3 groups: our algorithm matches on propensity score in 4.5 seconds compared to brute force's 17.5 hours, using commodity hardware available in 2012.

In the vast majority of cases, we can accomplish matching with three or more treatment groups without the constraint of exponential growth of the search space. Considering four groups of 5,000 patients, that is a reduction from 625 trillion matches to 100 million and orders of magnitude shorter computation time.

## 1. Introduction

Drugs that are marketed in the United States have gone through substantial testing, from Phase I clinical trials that examine basic pharmacology and toxicity to wide-scale Phase III trials testing the safety and effectiveness of a new agent against placebo or an existing agent [1]. However, even the largest Phase III trials are statistically underpowered to detect rare safety events, and placebo-controlled trials of most new agents will not yield information on how the drug compares to existing standards of care [2]. To evaluate these key factors, non-randomized post-marketing studies that give clinicians insight on the comparative safety and effectiveness of the the full range of possible treatment options are crucial [3].

A major determinant of the success of these studies is how well confounding is controlled; that is, given that patients in non-randomized settings are prescribed drugs because of their need for therapy [4], ensuring equivalence or "exchangeability" [5] between the groups under study at baseline is a first step to a valid assessment of the treatments' comparative safety and effectiveness. Further, when there are multiple available treatments, the usual pairwise comparison study is insufficient to give meaningful information to regulators, policymakers, and clinicians: for them, a study that compares three, four, or more active treatments to each other can help identify the best choices from a wide panel of available treatments [6].

Propensity scores have been widely used in epidemiology [7] to control for confounding. For each treatment under study, a patient's propensity score is simply his or her predicted probability of receiving that treatment (as opposed to the other treatments under study), as a function of all confounding factors measured for that patient [8]. Propensity scores have been shown to be "balancing scores", meaning that if the score is correctly specified and patients are correctly matched, then the factors that make up the score will be on average balanced between the treatment groups. This balancing removes confounding, since it creates treatment groups that are comparable at baseline.

Matching two groups of patients is not complex, though

*Email addresses:* jh2jf@virginia.edu (John R Hott), njb2b@virginia.edu (Nathan Brunelle), jrassen@partners.org (Jeremy Rassen), shelat@virginia.edu (abhi shelat)

a brute-force approach to it becomes less practical as the number of patients increases. Adding additional treatment groups makes a brute-force approach infeasible; indeed, with four groups of patients and just 5,000 patients in each group, a brute-force algorithm would have to test 625 trillion possible matched sets. Similarly, a naive greedy approach used for two groups loses correctness when considering more than two groups.

In this paper, we propose and evaluate an efficient algorithm for simultaneously matching three or more groups of patients by any normalized score. Specifically, we can match on propensity score, generalized propensity score, or any number of normalized patient characteristics such as age, weight, or height.

In section 2, we discuss similar multi-treatment approaches, as compared to the matching approach. Section 3 formally describes and discusses the problem. Section 4 defines both the brute force solution and pseudocode for our kd-tree algorithm. A proof that the kd-tree algorithm correctly matches brute force is given in section 5, followed by theoretical and empirical results in section 6. We conclude in section 8.

## 2. Related Work

Rosenbaum and Rubin's original definition of a propensity score defined the probability of a single binary treatment choice [8, 9, 10]. Continuous treatments, ordinal, and categorical treatments were not specifically considered with more than 2 groups.

Imbens [11] proposed the generalized propensity score, an extension of Rosenbaum and Rubin's definition for multiple levels of treatment. This approach allows for estimation of average outcomes using only the generalized propensity score for three or more levels of treatment. Feng, et al, [12] discuss Imbens' approach to estimate treatment effect through weighting and regression adjustment specifically in the context of a multi-categorical treatment.

The literature describing the generalized propensity scores notes methods for regression, stratification, and weighting using the GPS. However, Imbens notes that "matching approaches ... appear less well suited to the multi-valued treatment case" [11]. This may be in part due to the computational challenges of matching with more than two groups.

If using a matching caliper to restrict match distance and thereby eliminate patients that do not resemble patients in other treatment groups, then a benefit of matching is that it restricts the estimation sample to only patients who could plausibly have received any of the treatment options under study. Those who were not "at risk" for all of the treatments will be excluded from the analysis. This method is similar in concept the design of clinical trials, in which participation is limited to only those patients who could reasonably be randomized to any of the treatments. Matching, therefore, will often give a more valid estimate of the treatment effect of interest – the average effect in patients that could have received either treatment – but may do so at the cost of precision. This is particularly important when treatment effects are not homogenous across the population, such that the sub-population of inference becomes an important analytic choice.

We designed our matching algorithm, therefore, to accept and match on existing propensity score and generalized propensity score definitions as well as any number of normalized facets of the data set considered.

## 3. Objective

Informally, we would like to match participants of similar traits across multiple treatment groups for a given study. Requiring all matched sets to include a patient from each treatment group will guarantee that only patients that could have reasonably received any of the available treatments will be included in the matched sample. Closest or *smallest* matches—matches where the participants from each group are the most similar—are considered first, then matches of increasingly larger sizes, until either all points are consumed or a threshold of participants is included.

To define the problem formally, let us construct the following notation: let $k$ be the number of patient groups considered, $d$ be the number of facets to match per patient, and $n$ be the number of patients per group. For simplicity in analysis, we assume equal-sized groups; however, in practice, this assumption need not hold.

We define a patient as a $d$-dimensional point in the real numbers. Each treatment group is defined as a set, $G_1, ..., G_k$, with $n$ points each. The set of all patients, $\mathcal{P}$, is the union of all treatment groups, and contains $kn$ total points. Next, we define a match $m$ as a collection of one point from each $G_i$ as follows: $m \subseteq \mathcal{P}$ such that $\forall G_i, |m \cap G_i| = 1$. We define $\mathcal{M}$ to be the set of all such matches. Note that the size of $\mathcal{M}$ is exponential with respect to $n$, that is $\mathcal{M}$ contains $n^k$ matches.

Since we want to find the *smallest* matches, we must define this term specifically. Intuitively, we would like to pick matches such that all the points are close together and or alternatively, the size of the grouping in $d$-dimensional space looks to be the smallest. Therefore, let us define a function $size(m)$ as a measurement function on the matches

$$size(m) : \mathcal{M} \to \mathbb{R}.$$

We discuss in the next section possible concrete definitions of this function, but we note now that it must be well defined: any match must have an unique measure. Specifically, $m_1 = m_2$ implies $size(m_1) = size(m_2)$. Now, let us define $M$ as a **match covering** of $\mathcal{P}$ such that $M$ has $n$ total matches and each point in $\mathcal{P}$ is used

only once in $M$. Without loss of generality, let us assume that $M$ is sorted on the size of matches. Specifically, for any two matches $m_i, m_j$ in $M$, where $i < j$ then $size(m_i) \leq size(m_j)$. Next we define an ordering $<_M$ on match coverings such that $M_0 <_M M_1$ if for some index $i$ into the sorted match coverings, $size(m_{0,i}) < size(m_{1,i})$ and for all smaller matches $j < i, size(m_{0,j}) = size(m_{1,j})$, where $m_{0,j}$ is the $j$-th element of $M_0$ and likewise for $M_1$.

We must find the minimal match covering, $M_0$, such that $\forall i > 0, M_0 <_M M_i$. Therefore, it suffices to extract matches from $\mathcal{P}$ in order of *size*, smallest to largest, until no points are left.

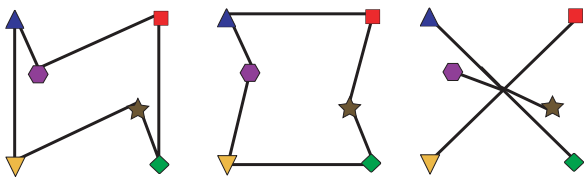
### 3.1. Definition of Smallest Match

We must first consider an appropriate definition of the $size(m)$ function, as described above. We consider 2 possibilities: perimeter and sum of squared distances to the centroid of the points in $m$. The choice between these definitions depends on consistency in determining the size given points in different orders (order independence of the points) and complexity to compute.

#### 3.1.1. Perimeter

The perimeter of a match is point-order-independent and unique in defining match size with 3 or fewer points, since any path through the points will yield the same perimeter. Therefore for 3 points, it only requires 3 distance calculations, making it constant-time to compute.

For more than 3 points, perimeter is no longer point-order-independent since different routes through the points would arrive at different perimeters, as shown in Figure 1. One solution is to require $size(m)$ be the perimeter of the smallest $k$-gon formed by the points. However, there are $(k-1)!/2$ possible $k$-gons for each match $m$, and thus the naive method for finding the minimal-perimeter order would take factorial-time in $k$. More clever routines to find such an ordering also seem unlikely as the problem reduces to the notoriously difficult NP-hard Traveling Salesman Problem [13]. Therefore, we limit the use of perimeter to the case when $k \leq 3$.



(a) perimeter $x$ (b) perimeter $y$ (c) centroid

Figure 1: Two different perimeters ($x \neq y$) and distance to the centroid for 6 points.

#### 3.1.2. Distance to the Centroid

We propose using distance to the centroid of the points instead of perimeter, since the centroid is neither dependent on the ordering of points nor the dimensionality of the space. Likewise, distance to the centroid can be calculated in $O(k)$ time with respect to the number of points per match.

For a $d$-dimensional space and $k$ points $p_1, ..., p_k$ per match, the centroid of a match $m$ is defined as

$$c(m) = \frac{1}{k} \sum_{i=1}^{k} p_i.$$

Our $size(m)$ function, using sum of squared distances to the centroid, is defined as

$$size(m) = \sum_{i=1}^{k} \sum_{j=1}^{d} \left(p_{i,j} - c_j(m)\right)^2,$$

where $c_j$ and $p_{i,j}$ define the $j$-th dimension of $c$ and $p_i$, respectively.

This definition is not only linear on both $k$ and $d$, but also captures the semantic definition of size: it measures how far away the points are from their average, the centroid. Unlike perimeter, with this definition of $size(m)$, the points will be mutually close to each other, discriminating against a closer grouping with one or two outliers. Specifically, sum of squared distances to the centroid will favor equilateral triangles over highly acute and obtuse triangles.

## 4. Matching Algorithms

We start by examining existing two-group and $k$-group solutions, followed by a naive algorithm to perform true $k$-way $k$-group matching and our kd-tree algorithm.

### 4.1. Two-group Matching Algorithms

Rubin [14, 15] proposed two approaches to two-group matching: mean-matching and pair-matching. The mean-matching approach matches subjects from two groups into a match covering $M$ such that the average $size(m)$ is minimized for all $m \in M$. Rubin provides a greedy approximation algorithm that, for each $p_i \in G_1$, chooses the $p_j \in G_2$ such that the average $size(m)$ is minimized for previous matches $m_0, ..., m_i$. The pair-matching approach, which we utilize, matches participants subject by subject. This method is both computationally faster than mean-matching and allows for extra analysis, including confidence limits, tests of significance, and the ability to study each match individually.

The initial definition of pair-matching is also a greedy approach. Rubin's algorithm either randomly permutes or sorts the first group $G_1$ by the facet to match, $X$, low-to-high or high-to-low. It then performs only one pass through the participants $p_i$ in $G_1$, matching each with

the closest point in $G_2$. An example pseudocode of his algorithm can be seen in Algorithm 1. This particular portrayal is a $O(n^2)$ implementation, however a more sophisticated algorithm will achieve the greedy approach $O(n \log n)$. According to Rubin's description, neither of these approaches produces the global in-order smallest pair matching; they rely on the initial ordering of $G_1$.

Rosenbaum [16] extended the work to utilize an optimal bipartite matching, which finds the match covering $M$ that minimizes $\sum size(m)$ for all $m \in M$. By definition, a bipartite matching only matches across two groups. Both of these methods, therefore, are unscalable to larger $k$ without significant modification.

**Algorithm 1:** Pseudocode of Rubin's pair-matching algorithm. *sort* may be either low-to-high on $X$, high-to-low on $X$, or random.

**Input**: 2 sets of $n$ points: $G_1, G_2$
**Output**: set of $n$ matches of 2 points each

```
G1 = sort(G1)
foreach p_i ∈ G1 do
    smallest = MAX
    foreach p_j ∈ G2 do
        if size(m = {p_i, p_j}) < smallest then
            m_smallest = m
            smallest = size(m)
    M_ans ← m_smallest
    remove p_i, p_j ∈ m_smallest from G1, G2
return M_ans
```

guarantee that the patients in the matched sample have clinical equipoise with respect to all treatments.

Although a $k$-way $k$-group matching seems simple given the above definition and algorithms, it is worth noting that simple, intuitive solutions may not yield a correct answer for more than two groups. For example, a natural first approach for 3 groups might be to start with any point in $G_1$, find its closest neighbor in a different group, and then find its closest neighbor in the third group. Hade [19] introduced triplet matching using this approach. She compares two methods of 3-way matching: the nearest neighbor approach and a sub-optimal matching approach. Both rely on the assumption that two closest pairs will form a closest triple. Figure 2 illustrates the incorrectness of these and other similarly "greedy" approaches. A match made with the closest neighbors to a starting point does not necessarily result in the smallest match for that point.

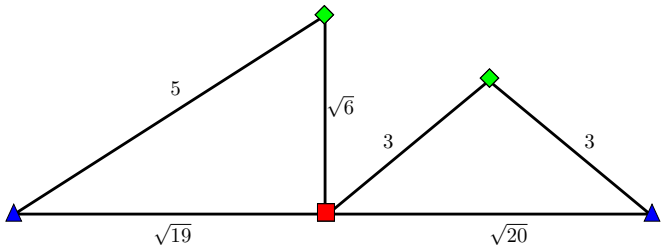



Figure 2: The blue (triangle) and green (diamond) points are closer to the red (square) point in match $m$ on the left, compared with match $m'$ on the right. However, $size(m) \approx 16.6667$ and $size(m') \approx 12.6667$.

### 4.2. Existing k-group Algorithms

Current $k$-group matching algorithms utilize a pair-matching technique to match both in and across treatment groups. The standard approach [17] uses non-bipartite matching, in which the two-group requirement is relaxed and pairings are allowed across two of many groups.

Lu et al. [17] discuss using non-bipartite matching to pair participants for various study settings. One parameter to their solution is our $k$-group matching problem. Their algorithm performs well, achieving a time complexity of $O((kn)^3)$ through the use of Derigs' nonbipartite matching FORTRAN implementation [18]. However, it is impractical for larger data sets and ill-suited for the $k$-way matching problem. First, it requires $O((kn)^2)$ memory to store the computed pair-wise distance matrix, which is prohibitive as $k$ and $n$ become moderately large, such as 5000 patients in 3 groups. Derigs' implementation also requires that all distances must be integer values, forcing reduced precision. Most importantly, however, each participant is paired with only one other member in the remaining $k-1$ groups rather than with one per group. A $k$-way matching is the only way to

### 4.3. Brute Force Algorithms

The robust naive approach, therefore, would include forming all potential $k$-way matches, sorting them based on the size of the match, then picking matches in order ensuring that no person is used twice. Assuming there are $n$ participants in each group, this will lead to $n^k$ total matches for $k$ groups, requiring $O(n^k)$ space and $O(n^k \log n)$ time to process the input since sorting the matches is necessary. If the trial contains 300 participants in each of 3 groups, that would be $100^3 = 1,000,000$ matches to store. As $n$ grows, current hardware becomes unable to store the data efficiently.

An alternative brute force algorithm exists that requires only $O(n)$ space, but has an increased time complexity of $O(n^{k+1})$. This algorithm considers all $n^k$ matches to find the smallest, removes those $k$ points from the pool, and repeats until $n$ matches are made. This approach can be seen in Algorithm 2.

### 4.4. KD-Tree Algorithm

Our algorithm reduces the brute force cost by considering only matches within a neighborhood of each point. By searching this neighborhood and ignoring all other

**Algorithm 2:** Brute Force Algorithm requiring $O(n)$ space, $O(n^{k+1})$ time

**Input**: $k$ sets of $n$ points: $G_1, ..., G_k$
**Output**: set of $n$ ordered smallest matches of $k$ points each

```
for i = 1 : n do
    smallest = MAX
    m_smallest = null
    foreach p_1 ∈ G_1 do
        foreach p_2 ∈ G_2 do
            ...
            foreach p_k ∈ G_k do
                if
                size(m = {p_1, p_2, ..., p_k}) < smallest
                then
                    m_smallest = m
                    smallest = size(m)
    M_ans ← m_smallest
    remove p_{1_m}, p_{2_m}, ..., p_{k_m} from G_1, G_2, ..., G_k
return M_ans
```

**Algorithm 3:** kd-tree algorithm

**Input**: $k$ sets of $n$ points: $G_1, ..., G_k$
**Output**: set of $n$ ordered smallest matches of $k$ points each

```
for i ← 2 to k do
    T_i = makeKDTree(G_i)
pq = new PriorityQueue
foreach p_i ∈ G_1 do
    addPutativeMatches(pq, p_i, T_2...T_k)
while pq not empty do
    m = pq.poll()
    if all p_i ∈ m are unused then
        foreach i ≤ k do
            T_i.remove(p_{i_m})
        M_ans ← m
    else
        if p_{1_m} ∈ m ∩ G_1 is unused then
            if no more matches available then
                addPutativeMatches(pq, p_{1_m}, T_2...T_k)
```

points, we seek to reduce the time complexity in the average case. To find a neighborhood, we first consider an approximate best match for a point in $G_1$: the closest points of $G_i$ for $2 \leq i \leq k$. Our algorithm uses kd-trees, with an expected case look-up time of $O(\log n)$, to find these $k-1$ close points.

Kd-tree data structures store points in arbitrary dimensions in a binary tree as described in [20]. For the 2-dimensional case, [20] showed that the time complexity for building a kd-tree is $O(n \log n)$, with expected-case nearest neighbor lookup cost $O(\log n)$ and worst case $O(n^{1-1/d})$ for $d$ dimensions. Therefore, in the 2-dimensional case, any request to find the nearest patient will be $O(\log n)$ in the average case, with the cost of making all kd-trees $O(kn \log n)$. The algorithm also uses `PriorityQueues` as implemented in Java. `PriorityQueues` can be queried and added to with a time of $O(\log n)$ [21].

Algorithm 3 provides the pseudocode for our kd-tree algorithm. A kd-tree $T_i$ containing the points in $G_i$ is created for $i = 2, ..., k$. The algorithm then creates a `PriorityQueue` for matches ordered on $size(m)$ and an array to store the final matches. Starting with $G_1$, the algorithm finds the smallest matches within a given search radius for each point $p_i \in G_1$, using the `addPutativeMatches` subroutine described below. Then, the algorithm considers all matches found, presorted ascending according to $size(m)$ due to the `PriorityQueue` construction. For each match, if all its points have not been used to make a match which has already been considered, then this match is a smallest match. The algorithm stores the match and removes these points from their respective kd-trees. Alternatively, if any of the points have been used for another match, the match is discarded. When all matches in the queue for any $p_i \in G_1$ have been exhausted, `addPutativeMatches` is called to add more matches onto the queue with the remaining points. When this process terminates, we have the $n$ smallest matches.

`addPutativeMatches` as outlined in Algorithm 4, starts with the given $p_i \in G_1$ and produces the 10 smallest matches for that point. First, it performs kd-tree queries to find a $p_j \in G_2$ closest to $p_i$, a $p_l \in G_3$ closest to $p_j$, and so on for all subsets of $\mathcal{P}$, using Euclidian distance. These points together become an initial naive smallest match $m$, with $size(m)$ as the sum of squared distance to the centroid of the match. Let $max(m)$ denote the maximum distance from a point to the centroid in this match. The algorithm queries each kd-tree $T_s$, $2 \leq s \leq k$, for points $p_r \in G_s$ such that $dist(p_r, p_i) \leq search = k \times max(m)$. These points $p_r$ are then used to create all possible matches in this search radius with $p_i$. The smallest 10 matches $m_t$ with $size(m_t) \leq size(m)$ are returned.

We only consider the smallest 10 matches from `addPutativeMatches`, since the execution time asymptotically converges as the number of matches returned increases as shown in Figure 3.

## 5. Proof of Equivalence

To show that this algorithm produces identical matchings to brute force, it suffices to show that for any given point in $G_1$, the smallest match for that point will reside in the search radius considered for that point. Since all $p_i \in G_1$ will be considered, we are guaranteed to

**Algorithm 4:** addPutativeMatches subroutine

**Input**: PriorityQueue $pq$, current point $p_1 \in G_1$, kd-trees for each group $T_2, ..., T_k$
**Output**: list of 10 smallest matches for point $p_1$

```
for i ← 2 to k do
    p_i = T_i.getnearest(p_{i−1})
small = size(m = {p_1, p_2, ..., p_k})
search = getSearchRadius(small)
tq = new PriorityQueue
tq.add(m)
for i ← 2 to k do
    L_i = T_i.getnearest(p_{i−1}, search)
foreach p_2 ∈ L_2 do
    foreach p_3 ∈ L_3 do
        ... foreach p_k ∈ L_k do
            dist = size(m' = {p_1, p_2, ..., p_k})
            if dist ≤ small then
                tq.add(m')
for i ← 1 to 10 do
    m = tq.poll()
    if m not null then
        pq.add(m)
```

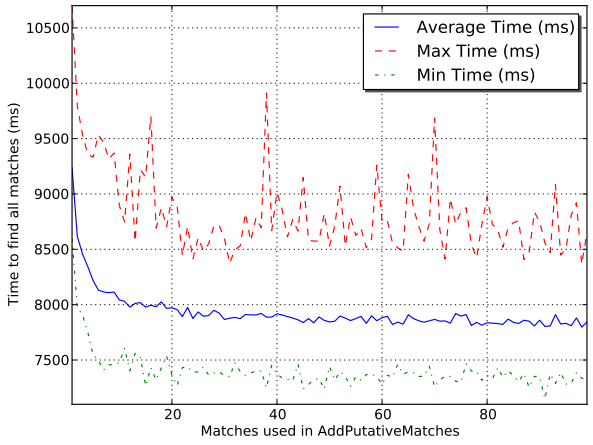



Figure 3: Empirical results varying the number of possible matches returned by `addPutativeMatches`.

| Parameter | Test Values |
|---|---|
| Number of Treatment Groups ($k$) | 3-5 |
| Number of Facets Per Patient ($d$) | 2-4 |
| Participants per Treatment Group ($n$) | 50, 100, 200, 300, 400, 500, 750, 1000 |

Table 1: Empirical test configurations.

end with the smallest match for each $p_i$. When points are used, the process will be repeated with larger search radii. Therefore, to prove the equivalence of the kd-tree algorithm, we will show that for an initial neighborhood of a match containing $p_i$, the smallest match will reside in that neighborhood for both definitions of $size(m)$: perimeter and sum of squared distance to the centroid.

### 5.1. Perimeter

**Theorem 1.** *Given an initial match $m$ containing point $p_i \in G_1$, which contains random points $p_{r_j}$, one per $G_j$ with $j \geq 2$, the perimeter will define the size of the match. Let us assume that $size(m) = q \in \mathbb{R}$. Our search radius will then be the disc centered at $p_i$ with radius $\frac{q}{2}$. This area will contain the smallest match for $p_i$.*

*Proof.* For contradiction, assume there is one point $p_j$ that resides outside of this disc that makes a smaller match $m'$ for $p_i$. We note that the euclidean distance from $p_i$ to $p_j$ is greater than $\frac{q}{2}$. Therefore, the smallest match $m'$ containing both points would be at least colinear with each point at an endpoint, making $size(m') > 2 * \frac{q}{2} = q$, however $size(m) = q$ and therefore $m'$ is larger than $m$. □

### 5.2. Centroid

**Theorem 2.** *Given an initial match $m$ containing point $p_i \in G_1$, which contains random points $p_{r_j}$, one per $G_j$ with $j \geq 2$, the sum of squared distances to the centroid will define the size of the match. Our search radius will be the disc centered at $p_i$ with radius $k * s$ where $s$ is the max distance to centroid. This area will contain the smallest match for $p_i$.*

*Proof.* We want to find $r$ such that given $m$ with $k$ points,

$$s = max\left(\sum_{l=1}^{d}(p_l - c_l(m))^2\right), \forall p \in m,$$

where s is the maximum Euclidean distance to centroid.

a) First let us discuss $size(m) \leq ks^2$. In our definition of $size(m)$, we know that

$$size(m) = \sum_{l=1}^{k}(p_l - c(m))^2,$$

where each $|p_l - c(m)| \leq s$, since $s$ is the maximum. Therefore this inequality is trivially true.

b) Assume there exists $p_j$ outside of our radius $r$, such that $p_j, p_i \in m'$. We will call the centroid of this match $c(m')$. Let us define $x$ to be the distance from $p_i$ to $c(m')$ and $y$ to be the distance from $p_j$ to $c(m')$. Note that $x + y \geq r$, since $p_j$ and $p_i$ are at least a distance of $r$ apart. Also, $size(m) \leq ks^2$ by part a. Therefore, to show that $size(m) \leq size(m')$ it suffices to show that $ks^2 \leq x^2 + y^2$, since then we will have

$$size(m) \leq ks^2 \leq x^2 + y^2 \leq size(m').$$

Let us assume the minimal $x + y$, namely that $x + y = r$. Then, it is clear that $\frac{r^2}{2} \leq x^2 + y^2 \leq r^2$. Now, to show $ks^2 \leq x^2 + y^2$, it suffices to show that $ks^2 \leq \frac{r^2}{2}$.

$$\begin{aligned} ks^2 &\leq \frac{r^2}{2} \\ ks^2 &\leq \frac{k^2 s^2}{2} \\ 2 &\leq k. \end{aligned}$$

Therefore for all $m'$ with at least one point outside of $r = ks^2$, $size(m') \geq size(m)$, where $k \geq 2$. □

## 6. Theoretical Results

We first discuss theoretical results for the algorithm, both in the worst case and the average expected case, followed by the empirical study results.

### 6.1. Worst Case

Initially, we consider the 3-group case in 2-dimensions. The time complexity of the `addPutativeMatches` subroutine in Algorithm 4 is an integral part of the complexity of the algorithm. The time complexity of `addPutativeMatches` is

$$\begin{aligned} T_{apm} &= O(n^2 \log n + 2n^{3/2} + 2n + 21 \log n - 2) \\ &= O(n^2 \log n). \end{aligned}$$

We divide the kd-tree algorithm into two independent parts to simplify the analysis. *part*1 consists of the initial $n$ calls to `addPutativeMatches`, while *part*2 contains the main loop of the algorithm. The time complexity of each part is

$$\begin{aligned} T_{part1} &= O\left(nT_{apm}\right) \\ &= O(n^3 \log n) \\ T_{part2} &= O\left(n^3 \log n + 2n \log n + \frac{n^3 - n}{10} T_{apm}\right) \\ &= O(n^5 \log n), \end{aligned}$$

with the algorithm's total time complexity

$$T_{kdtree} = T_{part1} + T_{part2} = O(n^5 \log n)$$

Generalizing the worst case analysis to an arbitrary number of dimensions and groups, we derive the time complexity of each part as follows:

$$\begin{aligned} T_{apm_{k,d}} &= O(n^{k-1} \log n + kdn^2 + kdn) \\ T_{part1_{k,d}} &= O\left(nT_{apm_{k,d}}\right) \\ &= O\left(n^k \log n + kn^2 \log n + kdn^2\right) \\ T_{part2_{k,d}} &= O\left(n^{2k-1} \log n + kn^{k+1} \log n + kdn^{k+1}\right) \end{aligned}$$

Assuming a relatively small $k$ and $d$, this leads to an overall time complexity of

$$O(n^{2k-1} \log n).$$

Fixing $n$, as $k$ and $d$ increase, the complexity will increase exponentially with respect to $k$, but only linearly (in the insignificant terms) with respect to $d$. The complete derivations are available in the Appendix.

### 6.2. Improved Worst Case

Ignoring space constraints, we could store all previous match sizes seen to reduce the cost of duplicating calculations in `addPutativeMatches`. Therefore, if a match has already been computed, `addPutativeMatches` becomes a constant-time lookup. This would require $O(n^k)$ space to store all possible matches, but would reduce the re-call time complexity cost of `addPutativeMatches` to $O(1)$. Therefore, our partial time complexities become

$$\begin{aligned} T_{part1_{k,d}} &= O\left(n^k \log n + kn^2 \log n + kdn^2\right) \\ T_{part2_{k,d}} &= O\left(n^k \log n + (k-1)n \log n \right. \\ &\qquad \left. + \frac{n^k - n}{10} T_{apm_{k,d}}\right) \\ &= O\left(n^k \log n\right), \end{aligned}$$

with the total time complexity reducing to

$$T_{part1_{k,d}} + T_{part2_{k,d}} = O(n^k \log n).$$

### 6.3. Expected Case

In order to evaluate an average case complexity of our kd-tree algorithm, let us define the following assumption for the distribution of points over the space.

*Assumption.* We have an even scattering of points with a uniform density. Specifically, $\exists\delta$ such that $\forall\varepsilon$-sized areas, there are $\delta\varepsilon n$ points in that region. That is, the number of points in a region is in constant proportion to the area of that region. The search radius of an initial guess for any point $p_i$ is proportional to the density of points available in that area, and therefore there will be a constant number of points within that fixed search radius.

This assumption eliminates the worst case, in which each $G_i$'s points must be grouped together or coincident, forcing the algorithm to include all possible matches in each search radius, then make and subsequently discard $n^{k-1}$ matches as invalid. Assuming a $\delta$-sized proportion of the total points of each $G_i$ to be included in any $\varepsilon$-sized area will guarantee that any small search radius will not contain most or all the points from any $G_i$, only a constant amount proportional to the area.

Under this assumption, the `addPutativeMatches` subroutine completes in constant amortized time, since for

each fixed search radius it must only consider a constant number of points. Noting that kd-trees provide average case $O(\log n)$ lookups in 2 dimensions, but are only guaranteed $O(dn^{1-\frac{1}{d}})$ in higher dimensions [22], we break the expected case complexity into 2-dimensional and multi-dimensional costs.

*6.3.1. 2-dimensional case*

In two dimensions, we find that

$$\begin{aligned} T_{apm_{k,2}} &= O(4(k-1)\log n + \log n) = O(k \log n) \\ T_{part1_{k,2}} &= O\left(nT_{apm_{k,2}}\right) = O(kn \log n) \\ T_{part2_{k,2}} &= O\left(n \log n + nk \log n\right) \\ &= O((k+1)n \log n) \end{aligned}$$

with the total time complexity reducing to

$$\begin{aligned} T_{kdtree} &= T_{build_{kds}} + T_{part1_{k,2}} + T_{part2_{k,2}} \\ &= O(kn \log n). \end{aligned}$$

*6.3.2. d-dimensional case*

However, for the general case of $d$-dimensions, the algorithm is not expected to perform quite as well. We find that

$$\begin{aligned} T_{apm_{k,d}} &= O(2(k-1)dn^{1-\frac{1}{d}} + \log n) = O(kdn) \\ T_{part1_{k,d}} &= O\left(nT_{apm_{k,d}}\right) = O(kdn^2) \\ T_{part2_{k,d}} &= O\left(n \log n + nk \log n\right) \\ &= O((k+1)n \log n) \end{aligned}$$

with the total time complexity reducing to

$$\begin{aligned} T_{kdtree} &= T_{build_{kds}} + T_{part1_{k,d}} + T_{part2_{k,d}} \\ &= O(kdn^2). \end{aligned}$$

## 7. Simulation Study

We conducted a simulation study comparing our kd-tree algorithm against the space-efficient brute force algorithm to supplement our theoretical analysis.

### *7.1. Experimental Setup*

This experiment was conducted on 1.6GHz dual-core AMD Opteron processors with 3GB of RAM running Ubuntu Linux 10.04. Both algorithms were coded in Java, compiled and run using the Oracle™ Java™ SE runtime environment version 1.6.0_26, with the kd-tree algorithm utilizing the WLU kd-tree implementation [23]. Table 1 shows the configurations for each experimental run, where the participants are sampled by a Gaussian distribution over the varying confounding factors. Each test was repeated 50 times for each algorithm.

### *7.2. Simulation Results*

Our kd-tree algorithm performs as expected as the number of participants increases. For example, for three groups of 1000 participants, the kd-tree algorithm matches all participants on propensity score in 4.5 seconds versus the brute force algorithm's 17.5 hours as depicted in Figure 5. In this fixed $k$ and $d$ example, the kd-tree algorithm compute time grows quadratically as $n$ varies from 50 to 1000. Similarly, the brute force algorithm's compute time grows relative to $n^4$ over the same interval.

Varying the number of facets matched, $d$, does not significantly alter the compute time of either algorithm, as shown in Figure 4(b). This behavior is expected, since $d$ is not a dominant term in the complexity of either algorithm. However, varying the number of groups, $k$, with a constant $d$ shows significant slowdown. For 5 groups, the kd-tree algorithm completed the matchings two order of magnitudes faster than the brute force algorithm.

Precisely, as the number of participants grows, the kd-tree algorithm grows relative to $n^2$ as expected. Likewise, the brute force algorithm grows relative to $n^{k+1}$. Therefore, as the number of participants grows, the speedup over brute force grows significantly, from 17x for 50 participants per group to 13,897x for 1000. The results, including all runs in which brute force completed on our cluster within the allotted CPU time of 168 hours, are depicted in Figure 4 and enumerated in Table 2.

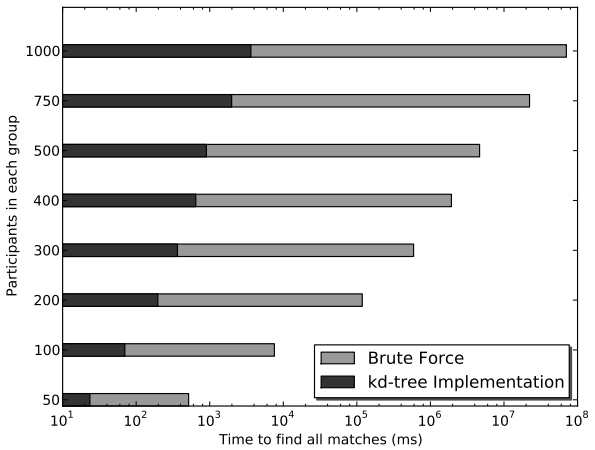
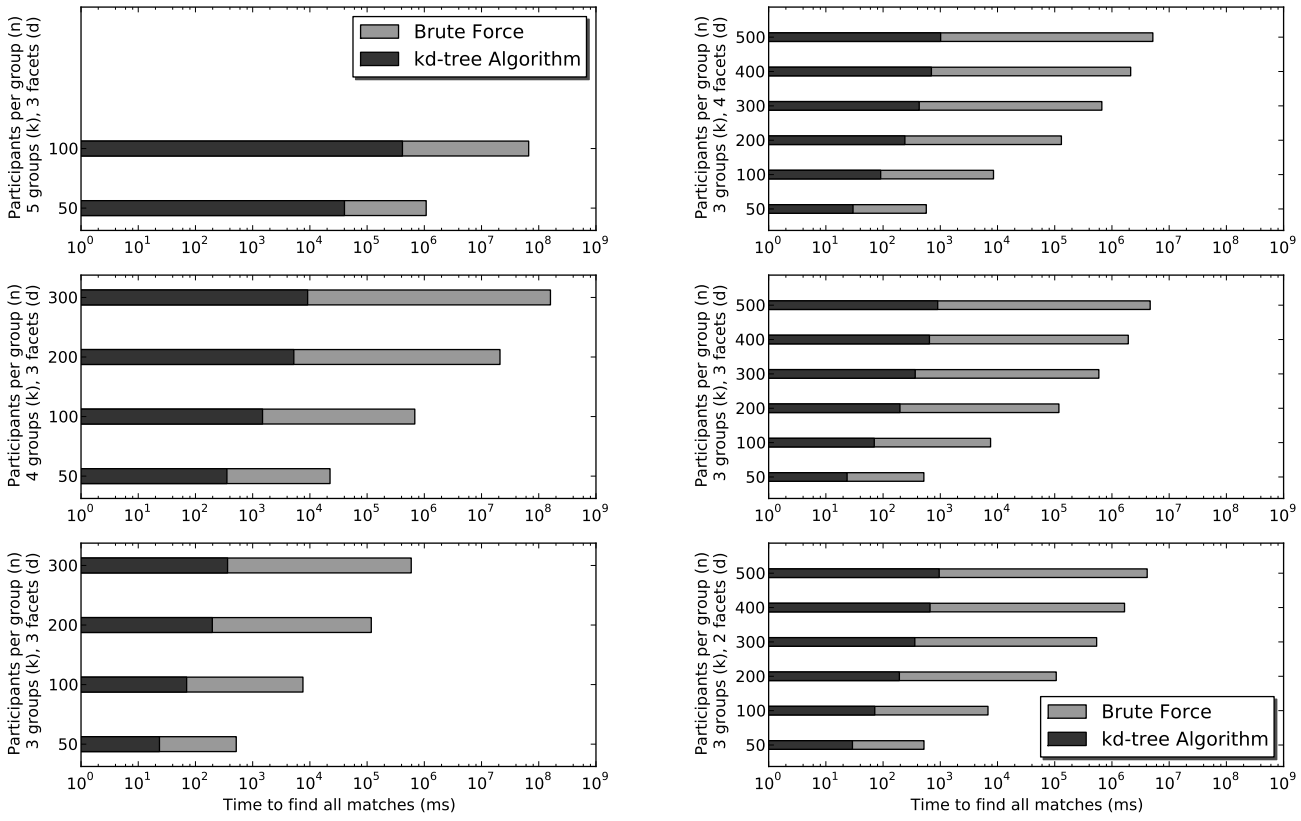



Figure 5: Average running time for our kd-tree implementation vs the space-efficient brute force implementation for up to 1000 participants in 3 groups considering 3 propensity scores. Note the $x$-axis is log-scale.

## 8. Conclusion

Given $k$ sets of participants, matching against $d$ traits, we have defined an algorithm that can produce the matches in an expected $O(kdn^2)$ time assuming the traits are uniformly distributed. Moreover, since our algorithm only uses kd-trees, a list of final matches, and an intermediary




(a) (b)

Figure 4: Average running time for our kd-tree implementation vs the space-efficient brute force implementation for (a) up to 300 participants in 3 to 5 groups considering 3 facets and (b) up to 500 participants in 3 groups considering 2-4 facets. Brute force for the results not shown did not complete within the allotted time. Note the $x$-axis is log-scale.

| Num Participants | Kd-tree Algorithm | | | Brute Force Algorithm | | | Speedup (kd-tree / brute force) | | |
|---|---|---|---|---|---|---|---|---|---|
| | 3 groups | 4 groups | 5 groups | 3 groups | 4 groups | 5 groups | 3 groups | 4 groups | 5 groups |
| 50 | 29.15 | 354.73 | $4.02 \times 10^4$ | 518.45 | $2.25 \times 10^4$ | $1.08 \times 10^6$ | 17.79x | 63.42x | 26.80x |
| 100 | 71.56 | 1487.52 | $4.15 \times 10^5$ | 6814.07 | $6.82 \times 10^5$ | $6.67 \times 10^7$ | 95.22x | 458.22x | 160.85x |
| 200 | 192.64 | 5238.16 | - | $1.06 \times 10^5$ | $2.10 \times 10^7$ | - | 551.57x | 4013.17x | - |
| 300 | 358.60 | 9134.80 | - | $5.40 \times 10^5$ | $1.60 \times 10^8$ | - | 1506.61x | 17530.88x | - |
| 400 | 661.36 | - | - | $1.66 \times 10^6$ | - | - | 2509.11x | - | - |
| 500 | 956.00 | - | - | $4.13 \times 10^6$ | - | - | 4320.33x | - | - |
| 750 | 2135.27 | - | - | $1.99 \times 10^7$ | - | - | 9332.74x | - | - |
| 1000 | 4519.80 | - | - | $6.28 \times 10^7$ | - | - | 13897.70x | - | - |

Table 2: Average time (ms) to match all partiicpants with the kd-tree and brute force algorithm for 3 facets, including the speedup of the kd-tree algorithm over the brute force algorithm.

list, we only use $O(n)$ space. While our approach does not match the performance of brute force in the worst case, we are no longer exponentially constrained by the number of groups in the expected case. We have substantially reduced the space and time required to compute complex, multi-group matches; we feel that this methodology can help clinicinas, regulators, and patients make more informed decisions based on the comparative safety and effectiveness across the full range of available treatment options.

## References


[1] B. Randall, IV, "The u.s. drug approval process: A primer," *Congressional Research Service*, June 1, 2001.

[2] N. Goldberg, S. Schneeweiss, M. K. Kowal, and J. J. Gagne, "Availability of comparative efficacy data at the time of drug approval in the united states," *JAMA*, vol. 305, pp. 1786–9, 2011.

[3] S. Schneeweiss, J. J. Gagne, R. J. Glynn, M. Ruhl, and J. A. Rassen, "Assessing the comparative effectiveness of newly marketed medications: Methodological challenges and implications for drug development," *Clin Pharmacol Ther*, vol. 90, pp. 777–790, 2011.

[4] A. M. Walker, "Confounding by indication," *Epidemiology*, vol. 7, pp. 335–6, 1994.

[5] J. M. Robins and S. Greenland, "Identifiability and exchangeability for direct and indirect effects," *Epidemiology*, vol. 3, pp. 143–55, 1992.

[6] J. A. Rassen, D. H. Solomon, R. J. Glynn, and S. Schneeweiss, "Simultaneously assessing intended and unintended treatment effects of multiple treatment options: A pragmatic "matrix design"," *Pharmacoepidemiol Drug Saf*, vol. 20, pp. 675–683, 2011.

[7] P. C. Austin, "A critical appraisal of propensity-score matching in the medical literature between 1996 and 2003," *Stat Med*, vol. 27, pp. 2037–49, 2008.

[8] P. R. Rosenbaum and D. B. Rubin, "The central role of the propensity score in observational studies for causal effects," *Biometrika*, vol. 70, pp. 41–55, 1983.

[9] P. R. Rosenbaum and D. B. Rubin, "Reducing bias in observational studies using subclassification on the propensity score," *J. Am. Statist. Assoc*, vol. 79, pp. 516–524, 1984.

[10] M. M. Joffe and P. R. Rosenbaum, "Invited commentary: propensity scores," *Am. J. Epidem*, pp. 1–7.

[11] G. W. Imbens, "The role of the propensity score in estimating dose-response functions," *Biometrika*, vol. 87, pp. 706–710, Sep 2000.

[12] P. Feng, X.-H. Zhou, Q.-M. Zou, M.-Y. Fan, and X.-S. Li, "Generalized propensity score for estimating the average treatment effect of multiple treatments.," *Statistics in medicine*, vol. 31, pp. 681–697, Feb. 2012.

[13] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The traveling salesman problem: a computational study*. Princeton series in applied mathematics, Princeton University Press, 2006.

[14] D. B. Rubin, "Matching to remove bias in observational studies," *Biometrics*, pp. 159–183, 1973.

[15] D. B. Rubin, "The use of matched sampling and regression adjustment to remove bias in observational studies," *Biometrics*, pp. 185–203, 1973.

[16] P. R. Rosenbaum, "Optimal matching for observational studies," *Journal of the American Statistical Association*, vol. 84, no. 408, pp. 1024–1032, 1989.

[17] B. Lu, R. Greevy, X. Xu, and C. Beck, "Optimal nonbipartite matching and its statistical applications," *The American Statistician*, vol. 65, no. 1, pp. 21–30, 2011.

[18] U. Derigs, "Solving non-bipartite matching problems via shortest path techniques," *Annals of Operations Research*, vol. 13, no. 1, pp. 225–261, 1988.

[19] E. M. Hade, *Propensity score adjustment in multiple group observational studies: comparing matching and alternative methods.* PhD thesis, The Ohio State University, 2012.

[20] A. W. Moore, "An introductory tutorial on kd-trees," 1991.

[21] Oracle, "Priorityqueue (java platform se 6)." http://download.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html, 2011.

[22] J. L. Bentley and J. H. Friedman, "Data structures for range searching," *ACM Comput. Surv.*, vol. 11, pp. 397–409, December 1979.

[23] S. D. Levy, "Kd-tree implementation in java and c#."