KD-Tree Algorithm for Propensity Score Matching

John R Hott, Nathan Brunelle, abhi shelat, Jeremy Rassen

Abstract

Propensity scores have been widely used in epidemiology to control for confounding bias in post-marketing studies, but the standard technique of matching patients by score becomes computationally impractical with studies of three or more treatment groups. We present a multi-category propensity score matching algorithm that 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 propensity score, balancing the distribution of confounders in each treatment group and thereby removing bias. Our algorithm outperforms the brute force approach in the expected case, requiring only O(n) space and time compared with brute force's time, for k treatment groups. This difference is clearly seen in our simulation study of 1000 participants in 3 groups: our algorithm matches in 4.5 seconds compared to brute force's 17.5 hours.

In the vast majority of cases, we can accomplish matching on propensity score 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. Background and Significance

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 [8]. 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 [5]. 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 [17].

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 [18], ensuring equivalence or "exchangeability" [6] between the groups under study at baseline is a first step to a valid assessment of the treatments' comparative safety and effectiveness. Further, as more drugs come on the market, the usual Drug A versus Drug B 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 [13].

Propensity scores have been widely used in epidemiology [2] 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 [16]. 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. As an example, if age is a component of the score, then in a study of cox-2 inhibitors such as celecoxib (Celebrex) versus non-selective non-steroidal anti-inflammatory drugs (ns-NSAIDS) such as ibuprofen, the process of propensity score matching will yield approximately equal ages in each of the two groups. If ages are approximately equal, age cannot confound the treatment-to-outcome association under study. With three or more treatment groups, a patient will have multiple propensity scores, one for each treatment. Because of the constraint that the total probability is 1, there are $k-1$ scores to match on for k treatment groups under study. Matching thus requires extending the usual two-group (single value) approach to an approach that can efficiently and correctly identify those patients with similar value of multiple scores.

Matching two groups of patients is not complex, though 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. In this paper, we propose and evaluate an efficient algorithm for simultaneously matching three or more groups of patients by propensity or other continuous-value score.

r: Citation ideas for bg material: Imbens, who adapted the propensity score to more than two groups: [7]. Foster, who used Imbens' result to compare number of mental health visits to a child's well-being: [4]. Rosenbaum's initial work on expanding the propensity score to two groups [14] [15]. Commentary on propensity scores from Rosenbaum and Joffe (background material) [9].

In section 2, we formally describe and discuss the problem. Section 3 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 4, followed by theoretical and empirical results in section 5. We conclude in section 6.

2. Objective

Informally, we would like to match participants of similar traits across multiple treatment groups for a given study. 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 consider the following variable definitions:

k	the number of patient groups considered
n	the number of patients per group
d	the number of facets to match per patient; for propensity scores $d = k - 1$

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 grous, 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 points.

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.

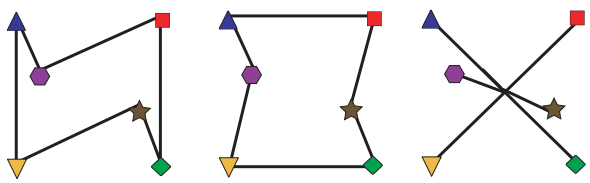
2.1. Definition of Smallest Match

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.

2.1.1. 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 $\frac{(k-1)!}{2}$ 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-complete Traveling Salesman Problem [1]. 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.

2.1.2. 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 colors 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

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

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.

3. Algorithm Design

3.1. Brute Force Algorithm

A naive approach for a matching algorithm would include matching all participants in each group, 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^3 total matches for 3 groups, requiring $O(n^3)$ space and $O(n^3 \log n)$ time to process the input since sorting the matches is necessary. If the trial contains 300 participants, 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 $O(n^4)$. This algorithm loops over k from 0 to $n-1$ to examine all $(n-k)^3$ remaining matches, only stores the smallest per iteration, and removes the used points from the pool. This approach can be seen in Algorithm 1.

3.2. 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 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.

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

Input: k sets of n points
Output: set of n ordered smallest matches of k points each

```
read input from file;
for i = 1 : n do
    smallest = MAX;
    m_smallest = null;
    foreach G_1 do
        foreach G_2 do
            ...
            foreach G_k do
                if size(m = {k_1, k_2, ..., k_k} < smallest
                then
                    m_smallest = m;
                    smallest = size(m);
    M_ans ← m_smallest;
    remove k_1m, k_2m, ..., k_km from G_1, G_2, ..., G_k;
return M_ans
```

Kd-tree data structures store points in arbitrary dimensions in a binary tree as described in [11]. For the 2-dimensional case, [11] 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 $O(\log n)$ [12].

Algorithm 2 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

Algorithm 2: kd-tree algorithm

Input: k sets of n points
Output: set of n ordered smallest matches of k points each

```
1  G_1 = read input from file;
2  for i ← 2 to k do
3      G_i = read input from file;
4      T_i = makeKDTree(G_i);
5  pq = new PriorityQueue;
6  matches = new ArrayList;
7  foreach p_i ∈ G_1 do
8      addPutativeMatches(pq);
9  while pq not empty do
10     m = pq.poll();
11     if all points in m are unused then
12         foreach i ≤ k do
13             T_i.remove(m.i);
14         matches.add(m);
15     else
16         if point from G_1 is unused then
17             if no more matches available then
18                 addPutativeMatches(pq);
```

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 3, 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 of G_s such that $dist(p_r, p_i) \leq k \times max(m)$. 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 2.

3.3. *Simulation Study*

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

Algorithm 3: addPutativeMatches subroutine

Input: PriorityQueue pq, current point pt_1 from color 1, kd-trees for each colors 2 to k
Output: list of 10 smallest matches for point pt_1

```
1  for i ← 2 to k do
2      pt_i = T_i.getnearest(pt_{i-1});
3  small = size(pt_1, pt_2, ... , pt_k);
4  search = get search distance from small;
5  tq = new PriorityQueue;
6  tq.add(match(pt_1,pt_2, ... ,pt_k));
7  for i ← 2 to k do
8      list_i = T_i.getnearest(pt_{i-1}, search);
9  foreach list_2 as pt_2 do
10     foreach list_3 as pt_3 do
11         ... foreach list_k as pt_k do
12             dist = size(pt_1,pt_2, ... ,pt_k);
13             if dist ≤ small then
14                 tq.add(match(pt_1,pt_2, ... ,pt_k));
15 for i ← 1 to 10 do
16     m = tq.poll();
17     pq.add(m);
```

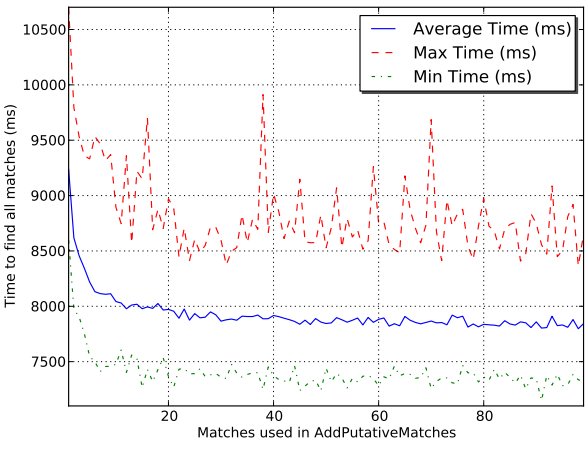


Figure 2: Empirical results varying the number of possible matches returned by `addPutativeMatches`.

Parameter	Test Values
Number of Treatment Groups	3-4
Participants per Treatment Group	50, 100, 200, 300, 400, 500, 750, 1000

Table 1: Empirical test configurations.

experiment was conducted on 1.6GHz dual-core AMD Opteron processors with 3GB of RAM. Both algorithms were coded in Java, with the kd-tree algorithm utilizing the WLU kd-tree implementation [10]. 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.

4. 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 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 correctness 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.

4.1. 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. □

4.2. 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

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

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 $ks^2 \leq \frac{r^2}{2}$.

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

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

5. 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.

5.1. Worst Case

The worst case example for this algorithm occurs when every point is contained within the search radius for a given point: for example, when all points are concident with each other. For this case, starting at any point from Group 1, finding a point in Group 2, then the closest in Group 3 will result in a search radius containing all points. Therefore, for any point, we must consider every possible match between all points in other groups.

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

of the algorithm. The time complexity of `addPutativeMatches` is r: Clean up the sums in the analyses

$$\begin{aligned} T_{apm} &= O\left(2\sqrt{n} + \log n \right. \\ &\quad +2(n-1+n\sqrt{n}) + n(n(\log n)) \\ &\quad \left. +10(\log n + \log n)\right) \\ &= O(2\sqrt{n} + \log n + 2n - 2 + 2n^{3/2} + n^2 \log n \\ &\quad +20\log n) \\ &= O(n^2 \log n + 2n^{3/2} + 2n + 21\log n - 2) \\ &= O(n^2 \log n). \end{aligned}$$

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

$$\begin{aligned} T_{part1} &= O\left(nT_{apm}\right) \\ &= O(n^3 \log n) \\ T_{part2} &= O\left(n^3 \log n + 2n\log n + \frac{n^3 - n}{10} T_{apm}\right) \\ &= O\left(n^3 \log n + 2n \log n \right. \\ &\quad \left. + \frac{n^3 - n}{10}(n^2 \log n)\right) \\ &= O\left(\frac{1}{10}n^5 \log n + \frac{9}{10}n^3 \log n + 2n \log n\right) \\ &= O(n^5 \log n), \end{aligned}$$

with the algorithm's total time complexity

$$\begin{aligned} T_{kdtree} &= T_{part1} + T_{part2} \\ &= O\left(n^5 \log n + n^3 \log n\right) \\ &= O(n^5 \log n) \end{aligned}$$

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

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

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.

5.2. *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

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

with the total time complexity reducing to

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

5.3. 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 δ-sized proportion of the total points of each G_i to be included in any ε-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 [3], we break the expected case complexity into 2-dimensional and multi-dimensional costs.

5.3.1. 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

$$\begin{aligned} T_{kdtree} &= T_{build_{kds}} + T_{part1_{k,2}} + T_{part2_{k,2}} \\ &= O\left((k-1)(n\log n) + kn\log n \right. \\ &\quad \left. +(k+1)n\log n\right) \\ &= O(kn\log n). \end{aligned}$$

5.3.2. 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

$$\begin{aligned} T_{kdtree} &= T_{build_{kds}} + T_{part1_{k,d}} + T_{part2_{k,d}} \\ &= O\left((k-1)(n\log n) + kdn^2 \right. \\ &\quad \left. +(k+1)n\log n\right) \\ &= O(kdn^2). \end{aligned}$$

5.4. Simulation Study Results

Finally, we give empirical results for our algorithm compared against the space-efficient brute force algorithm discussed earlier. Our kd-tree algorithm performs as expected as the number of participants increases. For example, for 1000 participants per group, the kd-tree algorithm matches all participants in 4.5 seconds, versus the brute force algorithm's 17.5 hours. 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 3 and enumerated in Table 2.

Precisely, as the number of participants grows, the kd-tree algorithm grows in relation to n^2 as expected. Likewise, the brute force algorithm grows exponentially. Therefore, as the number of participants grows, the speedup over brute force grows exponentially, from 17.8x for 50 participants per group to 13,897.7x for 1000.

6. 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 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] DL Applegate. *The traveling salesman problem: a computational study*. Princeton series in applied mathematics. Princeton University Press, 2006.

[2] PC Austin. A critical appraisal of propensity-score matching in the medical literature between 1996 and 2003. *Stat Med*, 27:2037–49, 2008.

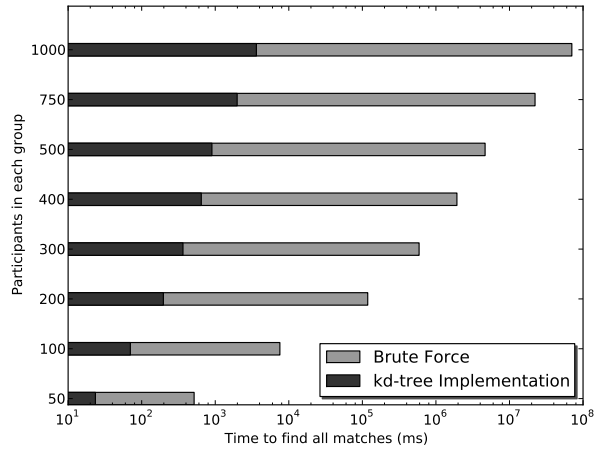
[3] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11:397–409, December 1979.

[4] EM Foster. Propensity score matching: An illustrative analysis of dose response. *Medical Care*, 41(10):1183–1192, 2003.

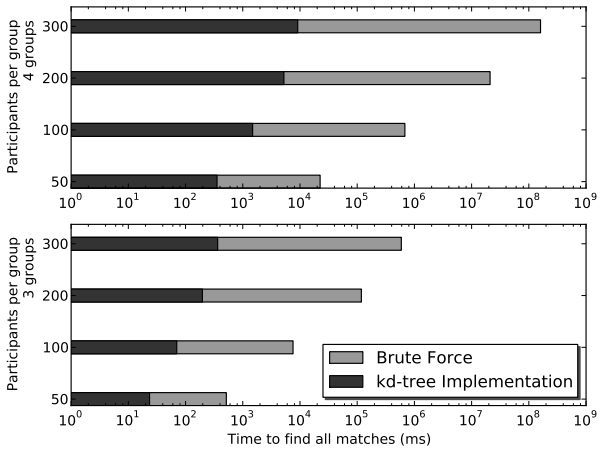
[5] NH Goldberg, S Schneeweiss, MK Kowal, and JJ Gagne. Availability of comparative efficacy data at the time of drug approval in the united states. *JAMA*, 305:1786–9, 2011.

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

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



(a)



(b)

Figure 3: Average running time for our kd-tree implementation vs the space-efficient brute force implementation for (a) up to 1000 participants in 3 groups considering 3 propensity scores and (b) up to 300 participants in 3 and 4 groups considering 3 propensity scores. Note the x-axis is log-scale.

[6] S Greenland and JM Robins. Identifiability and exchangeability for direct and indirect effects. *Epidemiology*, 3:143–55, 1992.

[7] GW Imbens. The role of the propensity score in estimating dose-response functions. *Biometrika*, 87(3):706–710, Sep 2000.

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

[9] M Joffe and P Rosenbaum. Invited commentary: propensity scores. *Am. J. Epidem*, pages 1–7.

[10] Simon D Levy. Kd-tree implementation in java and c#.

[11] Andrew W. Moore. An introductory tutorial on kd-trees, 1991.

[12] Oracle. Priorityqueue (java platform se 6). http://download.oracle.com/javase/6/docs/api/java/util/PriorityQueue.h 2011.

[13] JA Rassen, DH Solomon, RJ Glynn, et al. Simultaneously assessing intended and unintended treatment effects of multiple treatment options: A pragmatic "matrix design". *Pharmacoepidemiol Drug Saf*, 20:675–683, 2011.

[14] P Rosenbaum and DB Rubin. The central role of hte propensity score in observational studies for causal effects. *Biometrika*, pages 41–55.

[15] P Rosenbaumi and DB Rubin. Reducing bias in observational studies using subclassification on the propensity score. *J. Am. Statist. Assoc*, pages 516–524.

[16] DB Rubin and PRR Rosenbaum. The central role of the propensity score in observational studies for causal effects. *Biometrika*, 70:41–55, 1983.

[17] S Schneeweiss, JJ Gagne, RJ Glynn, M Ruhl, et al. Assessing the comparative effectiveness of newly marketed medications: Methodological challenges and implications for drug development. *Clin Pharmacol Ther*, 90:777–790, 2011.

[18] AM Walker. Confounding by indication. *Epidemiology*, 7:335–6, 1994.

Reviewer Comments

6.1. Associate Editor

Development of more efficient and effective methods for post-market surveillance of drugs is an important topic. However, as pointed out by the reviewers, the authors have to do much more to place their work in the context of prevoius related studies and to compare their method to other potential approaches in order to make the paper useful to the JAMIA audience.

6.2. Reviewer 1

The authors present an algorithm for computing propensity scores matching of patients, that is much more efficient than a naive matching algorithm. The paper is clear in most places, and the results are convincing. My main issue with the paper is one of fit to the journal. As of now, the paper reads like an abstract, not a manuscript. It would need to be widely expanded to be considered for this journal.

- Propensity scores: since this is such an essential part of the work, the authors need to ensure the reader undersands what these are about. One citation is not enough. Please expand with both relevant citations, examples, etc.
- Same for the concept of matching patients, especially in the case of different groups. Please provide more background, citations, and examples to explain better the task at hand.
- Is this paper the first to tackle this question? Can the authors provide more previous work on the task of patient matching?
- If the main contribution of the paper is the complexity of the algorithm through the use of an appropriate data structure, then a lot of the information in the appendix needs to be integrated in the body of the paper. r: Proof of Correctness and Time Complexity return to the paper

6.3. Reviewer 2

Hott et al. propose a new algorithm for propensity score matching with more than two treatments and compare this algorithm to a brute force approach. They find, both theoretically and by simulation, that their algorithm outperforms the brute force approach in realistic situations. The paper is clearly written and pleasantly short.

While their algorithm is of potential interest in medical research I would (before recommending the paper for publication) like to see to react Hott et al. to the following comments.

- I feel that the current standard approach for propensity score analysis with more than two treatments is the IPTW-approach as proposed by Imbens (Biometrika, 2000, 87(3):706-710) which is, for example, used by Foster (Medical Care 2003, 41(10):1183-1192). Can the author comment on their approach as compared to the Imbens approach? r: These seem to be more pre-work. We expand on these?
- While I understand that computing time is of most interest for a computer scientist when assessing a matching algorithm, the central measure of interest from a clinical viewpoint is the balance of covariates in the matched treatment groups. Is the authors' algorithm superior to the brute force approach also in terms of covariate balance? r: These matchings (and therefore balance of covariates are identical–addressed in the Proof of Equivalence section
- I wonder how the other matching approaches (Optimal matching, Greedy matching, Kernel matching, Genetic matching, ...) that are used with two treatment groups compare to the authors' proposal. Is there no possibility to use them in the more than two treatment situations? If yes, these algorithms are probably the better competitors for the authors' proposal than a simple brute force approach.
- Can the matching algorithm of Hott et al. deal with different (not necessarily 1:1:1 ...) matching ratios? Consider, for example, three treatments A, B, and C, where 10% of all patients have been treated with A, 10% with B, and 80% with C. Matching the three groups by a ratio 1:1:1 would necessarily omit the information of at least 70% of patients. Moreover, what about variable matching ratios, that is matching ratios 1:b:c, where "b" and "c" can be different numbers for patients from treatments B and C that are matched to one single patient treated with A.
- What about software? My experience is that a new computational method would not be used unless the proponents would provide a software tool that can be used with at least one of the standard software packages. Is there such an implementation of Hott's algorithm and are the authors willing to share it with the community? r: we have the Java source code that we could release.
- If you had asked me about a PS matching algorithm for more than two treatments, I would probably have come up with the following ad hoc algorithm using standard two treatment matching algorithms: First, match patients from treatments A and B, then set aside the matched patients from B and match the already matched patients from A with the patients from C (but do not allow dropping already matched A patients). How does this algorithm compare to that of Hott et al.? r: This doesn't work. Should we include the 7th son problem to illustrate the issue?