# Distributed K-Nearest Neighbors

Henry Neeb and Christopher Kurrus

June 5, 2016

## 1 Introduction

K nearest neighbor is typically applied as a classification method. The intuition behind it is given some training data and a new data point, you would like to classify the new data based on the class of the training data that it is close to. Closeness is defined by a distance metric (e.g. the Euclidean distance, absolute distance, some user defined distance metric, etc...) applied to the feature space. We find the $k$ closest such data points across the whole training data and classify based on a majority class of the $k$ nearest training data.

The K nearest neighbor method of classification works well when similar classes are clustered around certain feature spaces [1]. However, the major downside to implementing the K nearest neighbor method is it is computationally intense. In order to find our new data's $k$ closest observations in the training data, we will need to compare its distance to every single training data point. If we have training data size $N$, feature space size $P$, and assuming you choose a distance metric that is linear in the feature space, we need to perform $O(NP)$ computations to determine a single data point's $k$ nearest neighbors [2]. This computation will only grow linearly with the amount of new observations you wish to predict [3]. With large datasets, this classification method becomes prohibitively expensive for traditional serial computing paradigms [3].

K nearest neighbor also has applications beyond direct classification. It can be used as a way to approximate the geometric structure of data [2]. It does this by finding each data point's $k$ closest data points and "drawing" lines between them. This is represented by a graph data structure [4], but visually it can represent manifolds and geometric structures (see figure 1). Performing this type of operation is used predominately in manifold learning techniques, such as isomaps [4].

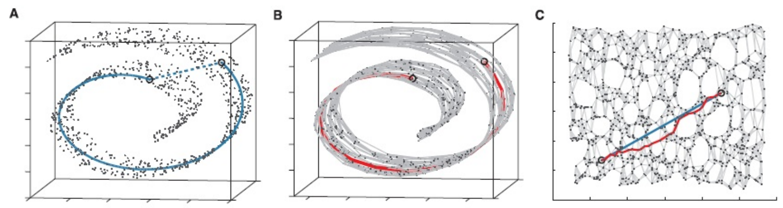

Figure 1: An application of isomaps on a "swiss roll" dataset. Notice the shape of the data. KNN is used to approximate this shape, then isomaps are used to reduce the dimensionality. *Source: Lydia E. Kavraki, "Geometric Methods in Structural Computational Biology"*

This application of K nearest neighbors is even more computationally heavy because now we must compare every point in our data set to every other point. Again, a naive implementation would require $O(NP)$ computations per data point. However, we would need to do this step $O(N)$ times for each data point, thus ultimately requiring $O(N^2P)$ computations.

We will investigate distributed methods implementing classification with K nearest neighbor as well as the geometry of data application. The first has a very direct deterministic distributive method. The latter will require a randomized approximate approach in order to compute in a distributive environment. For the geometry of data application, we will be mostly relaying information learned from *"Clustering Billions of Images with Large Scale Nearest Neighbor Search"* [6], where they talk about the use of implementing an approximate K nearest neighbor search in a distributed enviornment using hybrid spill trees.

## 2 Classification

For a classification problem, we are given a dataset of size $N$ where each data point falls into one of $C$ classes. Each data point contains $P$ features. For simplicity, we will assume that each feature is a continuous real number. We believe that each class $C$ will appear in clusters. That is, we think that the differences between the feature values within the class are smaller than the differences out of class. In addition, we have an additional dataset of size $M$ of which we do not know their classes. We would like to predict these data points classes using our dataset of size $N$ (our training data). Given this information, and assuming that our feature dimension $P$ is sufficiently small, or we have data size $N >> P$ such that we can overcome the curse of dimensionality associated with distance problems, we would like to apply the K nearest neighbors classification scheme [2].

The general idea is we will compare each data point in our classification data set to all of the data in our training set. We compute the distance between the data we wish to classify and all of the training data. The distance measure can be a variety of metrics (e.g. the Euclidean distance, absolute distance, some user defined distance metric, etc...). For our purposes, we will use the Euclidean distance metric for distances. The $K$ training data points with the smallest computed distance are the data points we will classify on. We classify the point to one of the $C$ classes that appears the most in the $K$ selected training points.

**Serial Classification - Algorithm**

We can iterate through each point you would wish to classify, compute its distance from every training data point, select the closest $K$ points, and return the class that appears the most of those $K$ points. The psuedo code for this process is as follows:

**Algorithm 1:** Serial KNN Method 1

```
Data: $X = N \times P$ matrix of training points
      $Y = N \times 1$ vector of the classes in $X$
      $A = M \times P$ matrix of data to classify
Result: $B = M \times 1$ vector of classifications for $M$ data set $A$
begin
    $B \leftarrow$ empty $M \times 1$ vector
    for data point $y_i \in Y$ do
        $l \leftarrow$ empty $K$ length list
        for data point $x_j \in X$ do
            $d \leftarrow Dist(y_i, x_j)$
            $c \leftarrow Y[j]$
            if $|l| < k$ then
                place $(c, d)$ in $l$ in descending order of $d$
            else if $l[k][2] < d$ then
                remove element $l[k]$
                place $(c, d)$ in $l$ in descending order of $d$
        $B[i] \leftarrow$ mode of $c$'s in $l$
```

**Serial Classification - Analysis**

On a single machine, this algorithm will take $O(NPM \log K)$ because:

- For each data point we wish to classify, we need to compute a distance metric, which will be $O(P)$
- We do this computation against $N$ data points, so $O(NP)$ for one data point to classify.
- We then need to compare the distance just computed to all other $K$ distances cached and insert. This can be done in $O(\log K)$ with a binary search. This is done at each computation of a distance, so our running complexity so far is $O(NP \log K)$
- We now need to do this for all of the data we wish to classify, so this is in total $O(NPM \log K)$

**Distributed Classification - Algorithm**

We can modify this algorithm to run on a distributed network. The most direct method of performing this is as follows:

- Compute the cross product between the data we wish to classify and our training data.
- Ship the data evenly across all of our machines.
- Compute the distance between each pair of points locally.
- Reduce for each data point we wish to classify that data point and the $K$ smallest distances, which we then use to predict

Assume we have $S$ machines. The above is easily implemented in a MapReduce framework.

**Distributed Classification - Analysis**

We will now analyze the time complexity, shuffle size, and communication of this process. For the mapper, all we are doing is pairing up and duplicating our data we wish to classify with our training data. Each pairing can be done in constant time, so the time complexity is proportional to the total data size, or $O(NM)$. Our shuffle size is also proportional to the amount of data we produce, which again is $O(NM)$. To distribute our data to all $S$ machines, we must perform a one-to-all communication. We cannot avoid not transmitting all $O(NM)$ data points, so our cost for communication is $O(NM)$.

**Algorithm 2:** Distributed KNN Method 1 - Mapper

```
Data: $X = N \times P$ matrix of training points, with $x_j$ the $j$th point
      $Y = N \times 1$ vector of the classes in $X$, with $y_j$ the class of
          the $j$th point
      $A = M \times P$ matrix of data to classify, with $p_i$ the $i$th point
Result: $M \times N$ tuples of form $(i, (p_i, x_j, y_j))$
begin
    Append $Y$ to $X$ and compute the cross product with $A$
    For each cross product, emit a tuple $(i, (p_i, x_j, y_j))$
```

**Algorithm 3:** Distributed KNN Method 1 - Reducer

```
Data: $M \times N$ tuples of form $(i, (p_i, x_j, y_j))$
Result: $M$ tuples of the form $(i, classification)$
begin
    for each input tuple do
        $d \leftarrow Dist(p_i, x_j)$
        form new tuple $(i, (d, y_j))$
    for each new tuple local on each machine do
        combine each tuple with same key such that we keep
        smallest $K$ $d$ values.
    Send all $(i, \text{List}[(d, y_j)])$ to one machine to be combined
    for each key value do
        Sort List$[(d, y_j)]$ by descending $d$.
        Keep only the smallest $K$ entries in sorted List$[(d, y_j)]$
        $c_i \leftarrow$ mode of remaining $y_j$'s in the list.
        return $(i, c_i)$
```

For our reducer's time complexity, we will investigate a single machine. Assume we can map data roughly evenly across all $S$ machines, so we have $\lceil \frac{NM}{S} \rceil$ data on a single machine. Our analysis for the time complexity for this process is similar to our sequential analysis:

- We need to compute a distance for each of our $\lceil \frac{NM}{S} \rceil$ data points, which will be $O(P\lceil \frac{NM}{S} \rceil)$ for all of our data on a single machine.
- The combination step increase our work, but will reduce our communication cost (analyzed later). If all same-type keys are on the same machine, this step requires appending our data to a single sorted list, which can be done in $O(N \log N)$ with a mergesort.
- When we reduce to one machine, each key will have a list of distance, classification tuple pairs of at least $K$ and at most $KS$ if the key was

distributed across all $S$ machines. If the size is only $K$, we just need to compute the mode of the classes, which should be $O(K)$. If not, we need to sort the list on our distances first and then compute a mode. Assuming our worst case, we can sort the list in $O(KS\log KS)$ with quicksort. Performing this for each data point will total $O(MKS\log KS)$ for the worst case and $O(MK)$ for the best case.

- Our total time complexity is then $O(P\lceil\frac{NM}{S}\rceil + N\log N + MKS\log KS)$

Putting all of our data from our $S$ machines onto one machine will require an all to one communication. Based on the above analysis, each machine will have to transmit at most $\lceil\frac{NM}{S}\rceil$ each, for a total of $O(NM)$ communication cost. This corresponds to no combining of data. However, if we do combine, the best case would be if all data with the same key is on the same machine for all keys. Then we would transmit only one tuple for each key, meaning our communication cost would be $O(M)$.

**Distributed Versus Serial Classification**

For situations where all of our data can fit on one machine, we want to know which methodology performs better. The computational complexity advantage of using a distributed cluster is apparent. In serial, our computational complexity is $O(NPM\log K)$. In our best case distributed situation (all keys on the same machine), the computing complexity is $O(P\lceil\frac{NM}{S}\rceil + N\log N) = O(N\log N)$ assuming $N > P\lceil\frac{NM}{S}\rceil$. The ratio of the serial time complexity to the distributed complexity is $O(PM\frac{\log K}{\log N})$, meaning we have approximately a $O(PM)$ speed up in computational complexity.

However, to do a thorough analysis, we also need to factor in the communication cost. It is worth noting that determining whether it is worth using a distributed setup depends on your computing cluster, including how fast of a network you have and how many machines you have. Note that we have a shuffle size of $O(NM)$ in our map step, which depending on your network can eclipse your computational complexity savings. During your reduce, you could also have another $O(NM)$ data transfer, although optimally it will be $O(M)$. These are not trivial communication costs, and deciding if to distribute or not will depend on this analysis.

# 3 Metric Trees and Spill Trees

Please note that for this section we will be explaining how hybrid spill trees operate, so that we can then implement them in our final algorithm. As such, we will be paraphrasing and borrowing heavily from *" An Investigation of Practical Approximate Nearest Neighbor Algorithms"*, which is cited below. Any fine details below originate from [5].

As mentioned earlier in the report, performing K nearest neighbor search for data geometry applications is computationally infeasible for large data sets. If we were to apply a modified version of the serial algorithm mentioned in the prior section, our time complexity would be $O(N^2P)$. For a modified distributed classification algorithm, our time complexity would be $O(P\left\lceil \frac{N^2}{S} \right\rceil$ assuming $N < P\left\lceil \frac{N^2}{S} \right\rceil$. To alleviate this problem, we must approximate the solution. There are a variety of approximate solution techniques for K nearest neighbor search. We will focus on using hybrid spill trees and metric trees as our primary data structure.

Spill trees are specialized data storage constructs that are designed to efficiently organize a set of multidimensional points. Spill trees operate by constructing a binary tree. The children in a full spill tree are a sub-set of the original data that empirically have the property of being close to one another. In other words, the spill tree is a way to look up data in which we can determine which data point is close to other observations.

Each parent node stores a classification method for whether a data point should go "left" or "right" along the node split. The classification method is done on whether the data point we wish to classify appears on either the left or right side of a boundary drawn through the data set (we'll talk about this boundary later in this report). The idea is that this boundary is constructed such that after a data point traverses the whole tree, it will have a high likelihood of ending up in a subset of data that is close to that point.

**Spill Tree - Algorithm**

The method for constructing a spill tree is as follows:

- Starting at the root of the tree, we consider our whole data set. At each parent node, say $v$, we consider the subset of data that abide by the decision bound created in the previous parent nodes.
- For each parent, we define two pivot nodes $p.lc$ and $p.rc$ such that $p.lc$ and $p.rc$ have the maximum distance between them compared to all other data points.

- We construct a hyperplane between *p.lc* and *p.rc* such that the line between the two points is normal to the hyperplane and every point on the hyperplane is equidistant from *p.lc* and *p.rc*.
- Define $\tau$ as the overlap size, with $0 \leq \tau < \infty$. $\tau$ is a tuning parameter in our construction of the spill tree. We use $\tau$ to construct two additional planes that are parallel to our initial hyperplane and are distance $\tau$ away from this initial boundary hyperplane.
- We define the two new hyperplane boundaries as follows: the hyperplane that is closest to *p.lc* is $\tau.lc$ and the other closer to *p.rc* is $\tau.rc$.
- We finally split the data at node $v$ based on the rule that if the data point is to the left of $\tau.rc$, we send it to the right child and if it is to the right of $\tau.lc$ we send it to the left child. If a data point satisfies both conditions, it is sent to both the right and left child.
- We keep splitting the data until the data size in each node hits a specified threshold.

Notice that the use of the $\tau$ parameter allows for data to be duplicated if it appears within $\tau$ of the initial boundary split. The use of this parameter is the defining feature of a spill tree. Spill trees that set $\tau = 0$ are called metric trees. The purpose of using $\tau$ is to get increased accuracy that we pick a subset of data in which points near the separation boundary is actually closest to. The larger the tau, the more accurate the splits, but also the more duplications of data we make and the slower the process.

The pseudo code for implementation of our algorithm $SpillTree$ is:

**Algorithm 4:** Spill Tree

```
Data: $X = N \times P + 1$ matrix of data, with each data $x_i$ having ID $i$.
          Have column $P + 1$ contain the class of $x_i$
      $U$ = a specified threshold upper bound to stop splitting
      $\tau$ = our buffer parameter
      $D = N \times N$ matrix of all distances between $x_i$'s.
          Let $D[i, j]$ = distance between $x_i$ and $x_j$.
      $T_{acc}$ = our accumulated tree. Initialized to just a root node.
      $n_{curr}$ = a pointer to current node of $T_{acc}$ we are considering.
Result: $T$ = our spill tree
begin
    If $D$ is empty (first call) compute $D$ with $X$
    if $|X| < U$ then
        return $T_{acc}$
    else
        $d_{max} \leftarrow max(D[i, j])$
        $x_{i-max}, x_{j-max} \leftarrow x_i, x_j$ s.t. $dist(x_i, x_j) == d_{max}$
        $plane \leftarrow$ separating plane midway and equidistant
            between $x_{i-max}, x_{j-max}$
        $plane_{-\tau}, plane_{+\tau} \leftarrow$ planes parallel to $plane$
            and $+/-\tau$ distance from $plane$   $D_r, D_l \leftarrow D$
        $X_r, X_l \leftarrow X$
        for $x_i \in X$ do
            if $x_i$ is to the left of $plane_{+\tau}$ and not to the right of $plane_{-\tau}$
            then
                drop $i$th row and column from $D_r$ and $X_r$
            else
                IF: $x_i$ is to the right of $plane_{-\tau}$ and not to the
                    left of $plane_{+\tau}$
                drop the $i$th row and column from $D_l$ and $X_l$
        create two new nodes $n_{curr.l}$ and $n_{curr.r}$ in $T_{acc}$
        create two directional edges from $n_{curr}$ to $n_{curr.l}$ and $n_{curr.r}$
        assign split criteria to $n_{curr}$ based on $plane_{-\tau}$ and $plan_{+\tau}$ return:
        $SpillTree(X_r, U, \tau, D_r, T_{acc}, n_{curr.r})+$
            $SpillTree(X_l, U, \tau, D_l, T_{acc}, n_{curr.l})$
```

**Spill Tree - Analysis**

We will now analyze the time complexity for constructing a spill tree. First, note that the complexity of constructing a tree depends on the parameters that you use to construct the tree and your underlying data. Exact runtime will vary between use of data sets. For example, closely clustered data and large $\tau$, we duplicate a lot of data, which will cause us to create a deeper tree with a longer termination time. We will mainly focus on the time complexity that will be independent of your data.

We also note that this is a recursive algorithm, so when we talk about datasize, we mean the size of the data within the current recursive step.

- Constructing $D$ for the first time will be $O(N^2P)$ because we need to create a distance for each pair of points in $X$. Consequently passing and computing $D_r$ and $D_l$ will not cost much of anything except dropping a pointer to the row and column of data omitted in the recursive call.
- Finding the maximum distance across all pairwise points will be $O(N^2)$
- Constructing the split criteria for the current node will be $O(P)$, which corresponds to constructing a plane of $P$ dimensions.
- Classifying each $x_i$ to the right and/or left split is $O(NP)$, corresponding to iterating through each point and evaluating which side of the hyperplane it lies on.
- Dropping columns and rows of $D_r$, $D_l$, $X_r$, and $X_l$ are all $O(1)$ since it corresponds to just dropping pointers.
- If we store $T$ as an adjacency list with pointers to the subset of data sets, adding completely new nodes and edges corresponds to just adding new rows, each with two pointers. This is a $O(1)$ operation.

The above has an initial cost of $O(N^2P)$ outside of the recursive calls associated with the initial construction of our $D$ matrix. Within our recursive calls, the dominating operation is $O(N^2)$. We then can construct the following recursive relationship: $T(N) = T(f(N)) + N^2$, where $f(N)$ dictates how quickly our data is being depleted. The base case for this recursive relationship is for $N < U$ and is $O(1)$.

Bounding $f(N)$ requires us to analyze our data structure and some of the parameters that we use to construct our spill tree (namely $\tau$). In fact, some of our choices of $\tau$ could possibly cause non-termination. For example, consider our selection of $\tau > max(D)$. At each step, we would be sending our data to both the left and right split each time. We would never reach our threshold $U$ and thus would never terminate.

## 4 Distributed K Nearest Neighbor

We would like to use a spill tree model on our data to categorize data with some likelihood of being close to our chosen data point. Once we construct the spill tree, classifying a point would then be as simple as performing a look up for the point (which would have complexity $O(PD)$ where $D =$ the depth of the spill tree) and then computing K Nearest Neighbor on the subset of the data, which if we set a threshold of the size of our child node equal to $U$ would be $O(U^2P)$ for a total time complexity of $O(PD + U^2P)$. Compared to our naive way of doing K Nearest Neighbor, this classification scheme with $U << N$, our runtime on classification is much less.

However, there are numerous problems when implementing a spill tree directly on our whole data set.

- Constructing our $D$ matrix in our spill tree algorithm is already $O(N^2P)$, so implementing a spill tree may help us for lookup, but will not help us in avoiding the time complexity of actually making the spill tree.
- For data size large enough that it will not fit on a single machine, calculating a spill tree directly in a distributed environment will be near-impossible. Like K Nearest Neighbor, to construct a spill tree with perfect precision, we need to compare each data point to every other data point to determine the furthest two data points. On a distributed setting, this is impossible without a lot of computer communication.

Instead of constructing a spill tree on all of our data, we will instead implement an approximate hybrid spill tree. This approximation method will construct a sampled hybrid spill tree on one machine. This spill tree will classify our data into chunks of data that can fit on single machines. From there, we construct a spill tree on each of the distributed data sets [6].

The main problem with using a normal spill tree as our data structure is that if $\tau$ is too large, the depth of the spill tree can go to infinity [6]. To prevent this, we can use a hybrid spill tree. Hybrid spill trees operate identically to spill trees, except they have an additional parameter $\rho$, with $0 \leq \rho < 1$ such that if the overlap proportion when *P.lc* and *P.rc* are constructed is $> \rho$, then *P.lc* and *P.rc* are instead simply split on the initial bound with no overlap [6]. These nodes are marked as non-overlap, which will become important later on. This guarantees that we will not have an infinite depth tree, and makes spill trees usable for our application.

We will be using a hybrid spill tree to store our data, because it will allow us to make efficient approximate K nearest neighbor searches. On a standard spill tree with overlaps we would use defeatist search to implement K nearest neighbors, but it is important to remember that we now have non-overlap nodes scattered throughout, for which we would instead use the standard metric tree depth first search.

This gives us an efficient data structure to use for our K nearest neighbors algorithm, but it is still designed to be used in a serial environment, and we want to use this in a distributed environment [6]. To accomplish this we will have to somehow split the dataset among multiple machines. This is problematic, as to generate a hybrid spill tree, you need to have all of the data accessible in memory. We solve this by using the following procedure [6]:

**Distributed KNN - Algorithm**

- Sample our original data each with some probability.
- Construct a spill tree using the algorithm in the previous section on the sampled data. We call this the top tree.
- Classify all of our data according to the top tree. When we reach a leaf in the top tree, we will have a subset of the original data which we will send to a single machine on a distributed cluster.
- On each distributed machine compute a spill tree on the subset of the data, again using the algorithm of the previous section. These are the bottom trees.
- For each leaf of all the bottom trees, compute K nearest neighbor of the subset data. store the K closest points, classes, and distances.
- If all of the data can fit on one machine, reduce the data to one machine. Duplicate data that went to more than one bottom tree will need to sort their list of values based on distances and return the K closest points.
- If the data cannot fit on one machine, it us to the algorithm designer on how they wish to handle duplicate points. Data can be left distributed and accessed via top and bottom trees with duplicate data being compared upon data retrieval. We can mark duplicate data to be reduced to a machine to find the best update and then modify our top and bottom trees to send our classification down the correct path for the optimal closest points.

We will provide an algorithm for when all of our data can fit on one machine:

```
Algorithm 5: Distributed Spill Trees
Data: $X = N \times P + 1$ matrix of data, with each data $x_i$ having ID $i$.
          Have column $P + 1$ contain the class of $x_i$
      $U$ = a specified threshold upper bound to stop splitting
      $\tau$ = our buffer parameter
      $D = N \times N$ matrix of all distances between $x_i$'s.
          Let $D[i, j]$ = distance between $x_i$ and $x_j$.
      $T_{acc}$ = our accumulated tree. Initialized to just a root node.
      $n_{curr}$ = a pointer to current node of $T_{acc}$ we are considering.
      $\frac{1}{M}$ = our sampling probability.
Result: $T$ = our spill tree
begin
    $S \leftarrow$ our sample dataset of $x_i$ sampled with probability $\frac{1}{M}$
    $Top \leftarrow SpillTree(S)$
    MAPPER:
    Label each leaf of $Top$ with a unique ID.
    Classify $X$ according to $Top$
    For each $x_i$ in each child with ID $c_{id}$, emit $(c_{id}, (i, x_i))$
    REDUCER:
    $Bottom \leftarrow SpillTree(X$ s.t. $c_{id}$'s are all equal$)$
    For each leaf in $Bottom$ compute Naive K-NN
    emit ($i$, List[closest $K$ point ids, distances, and classes])
```

To assist, we have provided a flow chart to illustrate what we are doing. See figure 2.

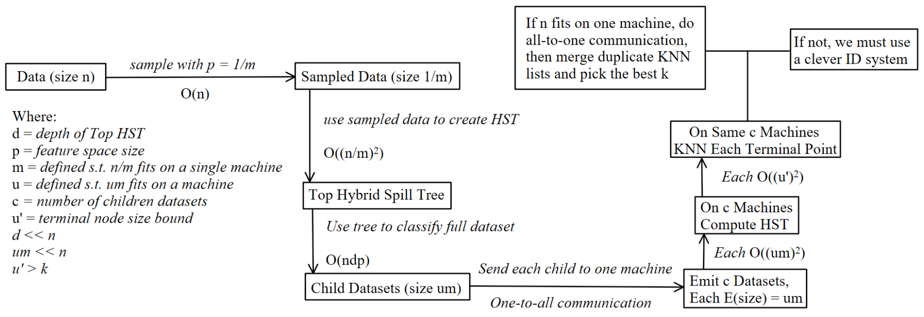
Data (size n)
sample with p = 1/m
O(n)
Sampled Data (size 1/m)
use sampled data to create HST
O((n/m)²)
Top Hybrid Spill Tree
Use tree to classify full dataset
O(ndp)
Child Datasets (size um)
Send each child to one machine
One-to-all communication
Emit c Datasets, Each E(size) = um
Each O((um)²)
On c Machines Compute HST
Each O((u')²)
On Same c Machines KNN Each Terminal Point
If n fits on one machine, do all-to-one communication, then merge duplicate KNN lists and pick the best k
If not, we must use a clever ID system
Where:
d = depth of Top HST
p = feature space size
m = defined s.t. n/m fits on a single machine
u = defined s.t. um fits on a machine
c = number of children datasets
u' = terminal node size bound
d << n
um << n
u' > k

Figure 2: A visualization of our algorithm to solve distributed K Nearest Neighbors

## Distributed KNN - Analysis

This procedure works because the hybrid spill trees yield children subsets that, with high probability, contain nodes that are close together [6]. This is true for hybrid spill trees more so than metric trees because in a metric tree, two points that are very close together can be split into two different subtrees and never be compared, but this does not happen in a hybrid spill tree [6]. In our algorithm, even if two similar points happen to be separated in one of the children resulting from a split, they will still be together in the other child, and so during evaluation it will still be considered. This property can be seen below:

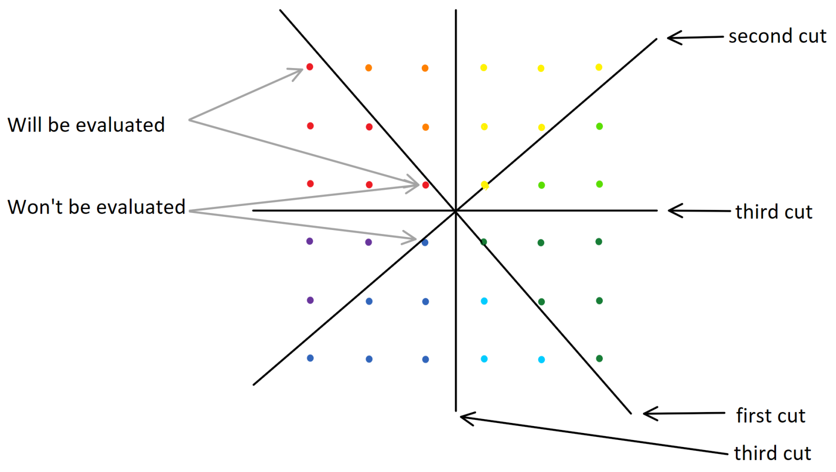

Figure 3: In the metric tree, two points that are very close together will not be evaluated, whereas two points much further apart will be, because they happen to be in the same split

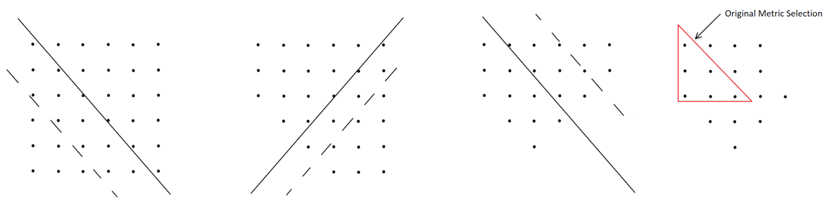

Figure 4: In the spill tree, we can see that when comparing the corresponding subset, there are much better odds of close together points being evaluated together

In our algorithm we incur a one to all communication. We note that each leaf is of size $O(UM)$. Our total communication cost will be the sum of the size of each classified leaf. While we cannot guarantee that size given that that would depend on the construct of your data and the $\tau$ parameter you use, we know that it is at least of size $O(N)$. If we employ the use of the $\rho$ parameter to reduce duplications, we know that at most we will tolerate $\rho$ duplications per split. We can then bound our communication cost above by $N(1+\rho)^d$. These are rough bounds - the real bounds heavily depend on the data set. Regardless, if your $N$ data cannot fit on a single machine, you would be hard pressed to compute K nearest neighbor without incurring a large amount of communication cost.

We also have an all to one communication towards the end where we send all of the K nearest neighbor lists to a machine, combining lists with identical keys. Without combining, the size of this cost will be equal to the initial communication cost because we have not gotten rid of any of our data at this point. When we combine, however, our data will again reduce down to size $N$.

We will now discuss the time complexity involved with computing:

- Sampling our data is $O(N)$ because we need to access every data element to sample. On an expected basis, the size of our sample will be $\frac{N}{M}$
- Constructing our top tree strictly depends on how quickly we deplete our data when we perform a recursive call. We know that outside the recursive call we have $O(|S|^2) = O((\frac{N}{M})^2)$ to perform regardless. On an expected basis.
- Letting $D =$ the depth of the top tree, classifying all of our $N$ data will take $O(NPD)$ as argued earlier, because we will need to compare each data point against a boundary condition, which will be $O(P)$, and we will need to do that at most $D$ times for each of our $N$ data.

- Once we have our data on each machine, each of expected size $UM$, we will need to compute another spill tree on each machine, again at least $O((UM)^2)$.
- Assuming we use the same threshold $U$ for our top and bottom tree, the leaves of each bottom tree will be size $O(U)$. Again, the number of leaves depends on our choice of $\tau$, $\rho$, and $U$, as well as our data shape. However, each leaf will compute a naive K nearest neighbor, which is $O(PU^2)$.

We notice from the above complexity analysis that the total time complexity requires us to know how many leaves we have in the top tree and the bottom trees. Our hope is that the number of leaves is such that our total time complexity is such that it is less than our naive $O(N^2P)$ method of computing K nearest neighbors. Regardless, if our data cannot fit on a single machine, we cannot hope to compute K nearest neighbor anyway without a distributed environment.

We notice that one of the advantages to our algorithm is that the runtime is heavily modifiable. Some of our user parameters such as $M$ and $U$ are tuned according to memory constraints or dataset size, but we can adjust $\tau$ and $\rho$ to bring down the runtime of our algorithm at the expense of accuracy, or increase the accuracy at the expense of the runtime [6]. Once you fit an initial model it is convenient to be able to optimize your accuracy, subject to your personal time constraints. Consider our top tree, we know that when $\tau$ is 0 the model becomes metric, and that as $\tau$ increases the overlap will become greater and greater between pairs of children nodes. It is therefore important to keep $\tau$ relatively low in the top tree, where a large amount of overlap will result in a greater compute time in the top tree, as well as many more computations down the line when we are using our bottom trees [6]. In the bottom trees, however, it is less important to keep $\tau$ low, as multiple machines are making their spill tree computations in parallel, and we can afford to allow more overlap, as long as we set an appropriate value for $\rho$ to prevent the tree depth from getting out of hand. Overall, the user defined parameters give our model a large degree of customization, and allow it to become optimized for the problem at hand.

## 5 Conclusion

We have seen two methods of distributing K nearest neighbors. The direct approach exhibits rather large communication costs as we need to duplicate our data multiple times ($O(NM)$) to implement it. It does, however, have reduced complexity; linear in the number of machines that we have.

Our second approach is to approximate our K nearest neighbors with hybrid spill trees. The exact time complexity and communication cost is highly dependent on your data as well as some of your model parameters (namely your resampling parameters). Further emperical analysis will need to be completed to flesh out how dependent these features are on runtime and communication cost. It should be noted, however, that despite these uncertainties you may still have to implement your problem in this manner because of your data size. If your data is too big to fit on one machine, there is little you can do to avoid high communication costs when trying to implement K nearest neighbors on a distributed framework.

## 6 References

[1]: Richard Duda, Peter Hart, David Stork Duda. *"Pattern Classification"* p.18-30, 2001.

[2]: Saravanan Thirumuruganathan. *"A Detailed Introduction to K-Nearest Neighbor (KNN) Algorithm"*, 2010.

[3]: Ali Dashti. *"Efficient Computation Of K-Nearest Neighbor Graphs For Large High-Dimensional Data Sets On GPU Clusters"*, 2013.

[4]: Wei Dong, Moses Charikar, Kai Li. *"Efcient K-Nearest Neighbor Graph Construction for Generic Similarity Measures"*, 2011.

[5]: Ting Liu, Andrew W. Moore, Alexander Gray, Ke Yang. *"An Investigation of Practical Approximate Nearest Neighbor Algorithms"*, 2004.

[6]: Ting Liu, Charles Rosenberg, Henry Rowley. *"Clustering Billions of Images with Large Scale Nearest Neighbor Search"*, 2007.