k-means, kd-Trees, and Median of Medians

Color Quantization done fast

Stephen Huan[1]

[1]Thomas Jefferson High School for Science and Technology

TJ Vision & Graphics Club, December 2, 2020

Table of Contents

Color Quantization

Reducing the number of colors

Color quantization is the process of reducing the number of colors in an image.
For example, a typical RGB image stores 1 byte per color, so 24 bits over 3 colors.
This has applications in compression, but is most often used for legacy hardware (whose memory is limited, so the number of bits/pixel must be limited).

An image amenable to color quantization

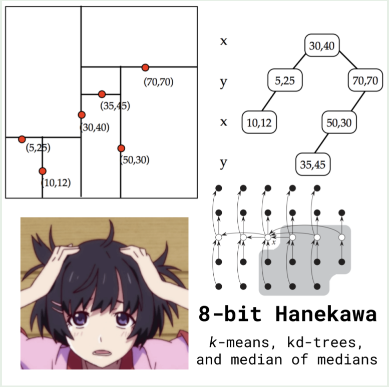


Figure: A summary of this lecture

Techniques

Suppose we want to decompose an image into k distinct colors. One simple approach would be to find the k most frequent colors and use those.
This has a pretty obvious failure mode.

Failure mode for the frequency heuristic

Suppose we have an image with 4 colors: dark red has 50 pixels, light red has 49, dark blue has 48, and light blue has 47. If $k = 2$, then we would choose dark red and light red, which would be problematic for the blues. A better selection would probably be to pick a normal red and a normal blue, at the cost of not showing the dark/light contrast within the colors.

Techniques, continued

We could address this by splitting our color space into color ranges, or using a different algorithm like the *median cut* algorithm, which constructs a kd-tree on the color space. Today, we will discuss applying the k-means clustering technique (an AI/ML lab here at TJ).

Table of Contents

k-means

Suppose we have a set of points and a set of k center points. Define the Euclidean distance between two points as the magnitude of the vector difference, i.e.

$$\texttt{dist}(\vec{u},\vec{v}) = \|\vec{u}-\vec{v}\| = \sqrt{(u_1-v_1)^2+(u_2-v_2)^2+\dots}$$

We want to pick our center points such that they minimize the sum of the distance between each point to its closest center, i.e.

$$\underset{\text{centers}}{\text{argmin}} \underbrace{\sum_{\vec{p}\in\text{points}}}_{\text{sum over all points}} \underbrace{\min_{\vec{c}\in\text{centers}} \texttt{dist}(\vec{p},\vec{c})}_{\text{closest center to }\vec{p}}$$

k-means, continued

This problem is NP-hard, necessitating a greedy algorithm.

k-means algorithm

The standard algorithm alternates between two steps until convergence: Given k initial center points,

1. Assign each point to its closest center
2. Update each center to the centroid of the points assigned to it, where the centroid is the arithmetic mean.

Intuitively, if the center points are fixed, then the best possible assignment is to assign each point to its closest center. In the dual case, if the assignment of points to centers is fixed, then the best possible center position is the mean point, i.e.

$$E[S] = \frac{1}{\|S\|} \sum_{\vec{p} \in S} \vec{p}$$

k-means, continued

"Convergence" is when after the centers are updated, the assignment of points to their closest center is the same as the assignment before the update. (the next center update would be the same).

This finds a local optimum, not necessarily the global optimum.

The number of iterations until convergence is also superpolynomial in the worst case, but in general works quite well in practice.

Our pick for the initial k centers is quite important!

Table of Contents

k-means++

An initialization scheme for *k*-means

Two theoretical problems with *k*-means

1. Running time is superpolynomial with respect to the number of points
2. Approximation can be made arbitrarily bad compared to the optimal clustering

k-means++ fixes the latter problem, it guarantees an $O(\log k)$ approximation bound in expectation (i.e., over expectation the clusters generated by *k*-means++ has a distance of at most $O(\log k)$ times greater than the optimal cluster).
How does it work?

k-means++, continued

k-means++ algorithm

1. Pick the first center point at random
2. From there, pick the next center point by sampling the probability distribution where a point $\vec{p}$ is picked with weighting $\texttt{dist}(\vec{p}, \vec{c})^2$, where $\vec{c}$ is the closest center
3. Repeat until k centers are picked
4. Run k-means as usual

Intuitively, picking centers far away from each other is a good thing, so the weighting favors points that are far away from the existing centers.
Also, it's impossible to have two identical centers, since the distance of a point to itself is 0, so its weighting would be 0.
How do we sample this probability distribution?

Sampling a random variable

Algorithm Sampling a random variable

```
def sample(p: list) -> float:
    """ Samples a value from a random variable. """
    r = random.random()
    i = cmf = 0
    while i < len(p) - 1:
        cmf += p[i]
        if r < cmf:
            break
        i += 1
    return i
```

Sampling a random variable, justification

Claim: This function has the same *cumulative mass function* as the underlying probability distribution.
For a uniform random variable $X \sim [0, 1]$, the probability that X is less than some value x, $p(X \leq x)$, is

$$\int_0^x 1 \; dt = x$$

The chance `sample` outputs an index $\leq j$ is if the sum of the probabilities up to j is greater than the uniform r.v. X, or flipping the inequality, if X is less than the sum.

$$p(\texttt{sample} \leq j) = p(X \leq \sum_{i=0}^{j} p[i]) = \sum_{i=0}^{j} p[i]$$

Sampling a random variable, continued

By definition, this is the cmf of the discrete r.v.
If `sample` has the same cmf as p, then it has the same pmf. If it has the same pmf, then this is "sampling the probability distribution" represented by p by definition!

Step 2 of k-means++, in detail

1. Make a list of the squared distances from a point to its nearest center.
2. Normalize this list into a probability distribution by dividing by its sum: `p = [x/sum(l) for x in l]`
3. Call `sample` to get a index which corresponds to a point.
4. Add this point to the list of centers.

Notes on k-means++

Yes, k-means++ adds an additional k passes over the data compared to picking k points at random.
First, as stated earlier k-means++ bounds the amount of error and will generally produce higher quality clusters.
Second, the better initialization also reduces the number of iterations until convergence for k-means, making it actually faster in practice.

Notes on k-means for color quantization

Let's go back to the original problem, color quantization. How do we apply k-means for color quantization?
If we want to reduce an image to k colors, we simply run k-means, where our points are the RGB values. This finds k colors, and we assign each color in the original image to the closest color in our k colors as usual, except we need to round our k centers to integer pixel values.
Note 1: Euclidean distance is not the best in terms of color difference perception (a smaller Euclidean distance does not necessarily imply that the colors look closer compared to a larger distance). We can change to the Lab color space or change distance measures.

Notes on k-means for color quantization, continued

Note 2: Square roots are expensive, and if we don't ever need the actual distance, we can always compute distance squared. We use `dist` in two ways:

1. To find the closest center point to a given point
2. To weight the probability distribution in k-means++

$f(x) = x^2$ is a *monotonic* function, i.e. if $x < y$ then $f(x) < f(y)$ (if x is nonnegative, and distances are always nonnegative by definition). Therefore, minimizing $f(x)$ is equivalent to minimizing x. The same trick is frequently used in machine learning loss functions.

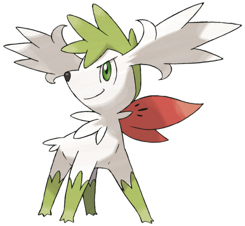
For use case 2, we weight a point by its squared distance. Thus, what I call "`dist`" is in fact

$$\texttt{dist}(\vec{u}, \vec{v}) = \|\vec{u} - \vec{v}\|^2 = (u_1 - v_1)^2 + (u_2 - v_2)^2 + \dots$$

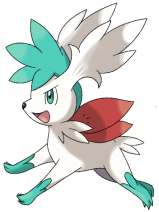
Table of Contents

Pokémon profile picture month

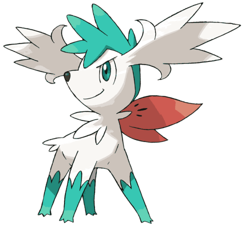
As per TJ tradition, December is for “Pokémon profile pictures”, where people update their Facebook profile pictures (“pfp”’s) to their favorite Pokémon.



(a) The image I like, original Shaymin



(b) The color scheme of a shiny Shaymin



(c) First image with the second’s color

Figure: Color transfer

Suppose I like the pose of the first image, but I want the Shaymin to be shiny. Can I use *k*-means to transfer the color?

k-means for Pokémon pfp month

1. Pick a good value of k. k should be large enough such that the image still looks reasonably good, but small enough for you to be able to modify colors by hand. In this case, $k = 16$.
2. Run k-means as usual
3. Identify which colors are green (to substitute with the shiny colors)
4. Temporarily set a green color to (0, 0, 255), i.e. an indicator color to see where it appears in the image.
5. Identify the corresponding color in the shiny form
6. Make the replacement

k-means for Pokémon pfp month, code

color modification

```
centers, ids, groups = k_means(K, data)
px = [tuple(round(c) for c in center) \
      for center in centers]
# color modifications to make shiny
px[ 5] = ( 55, 199, 179) # main color
px[ 6] = ( 17, 112, 106) # eye color
px[ 8] = ( 97, 210, 182) # tips of head/feet
px[15] = ( 35, 138, 123) # parts in shadow
```

Making k-means faster

Back to theory. If k-means takes I iterations to converge and there are N points, then the computational complexity is $O(NKI)$ as on each iteration, we need to compute the closest center to a point, and the easiest way to do that is to iterate over each center point. There are N points and K center points, so it takes $O(NK)$.
We can speed this up if we can compute the closest center point quicker, which we can do with kd-trees.

Table of Contents

kd-Tree

kd-tree

A kd-tree is essentially a binary search tree (BST) generalized to multiple dimensions. Each node has at most 2 children.

In a BST, to insert a value we compare it against the root's value, if it's less we recur on the left subtree, if greater on the right subtree.
A kd-tree is similar except each node holds a *point*, not a value. This point has D dimensions, and at each layer of the kd-tree we compare against a particular "cutting dimension" d.
We typically cycle through cutting dimensions.

kd-Tree insert

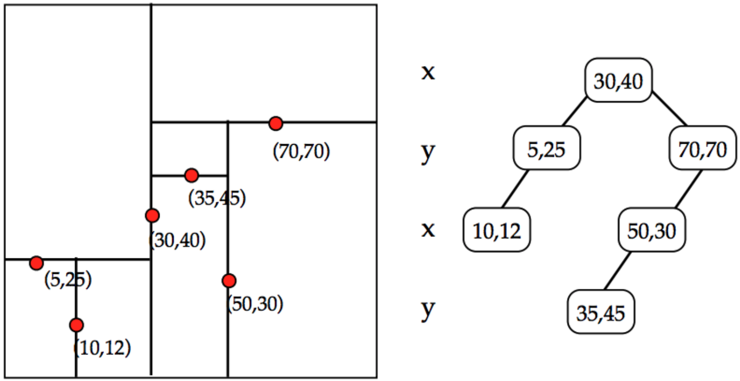


Figure: kd-Tree in 2 dimensions

kd-Tree node

kd-tree node

```
class kdNode:

    def __init__(self, point: tuple=None, cd: int=0):
        self.child = [None, None]
        self.point = point
        self.D = len(point)
        self.cd = cd
```

get children

```
def dir(self, p: tuple) -> int:
    """ Gets the proper left/right child
        depending on the point p. """
    return p[self.cd] >= self.point[self.cd]
```

kd-Tree insert code

Algorithm kd-Tree Insert

```
def __add(self, t, p: tuple, parent=None):
    if t is None:       # found leaf
        t = kdNode(p, (parent.cd + 1) % parent.D)
    elif t.point == p: # ignore duplicates
        return t
    else:              # update pointers
        t.child[t.dir(p)] = \
        self.__add(t.child[t.dir(p)], p, t)
    return t

def add(self, p: tuple) -> None:
    # empty tree, simply change our own point
    if self.point is None:
        self.__init__(p)
    self.__add(self, p)
```

Runtime Analysis

Like a BST, we would expect insert to take $O(\log n)$, since we do a path from root to leaf and in a balanced tree the depth is $\log n$, where n is the number of nodes (equivalent to the number of points).
However, there is a clear degenerate case: if each point increases along every dimension, then the tree becomes a line with height $O(n)$. $1 + 2 + \cdots + n = O(n^2)$, so it might take quadratic time to build a kd-tree in the worst case.
Common BST tricks like AVL trees seem difficult (are rotations even possible if they change the cutting dimension?).
In practice, the points are commonly known ahead of time. Can we guarantee an $O(n \log n)$ build over n points if we know the n points ahead of time?

Table of Contents

Pre-sort algorithm

Observation: Splitting the points perfectly in half between the left and right subtrees is the best possible split. Obviously, we split the points in half based off the *median* value.
To efficiently keep track of the median, we make D copies of the N points. We sort each copy on a different dimension. Finally, we pass this list of lists to the kd-Tree build function, find the median point, split our list of lists into a left and right side, and recursively build the tree. As long as we maintain the sorted invariant, we can compute the median point on any dimension in $O(1)$.

Pre-sort algorithm, code

wrapper over kdNode

```
class kdTreeSort(kdNode):

    def __init__(self, points: list=[]) -> None:
        super().__init__()
        if len(points) > 0:
            D = len(points[0])
            # no need for duplicate points
            self.points = list(set(points))
            # sort points on each dimension
            pointsd = [sorted(self.points,
                              key=lambda p: p[d])
                              for d in range(D)]
            build_tree(self, pointsd)
```

Pre-sort algorithm, code

split

```
def subsplit(pointsd: list, seen: set) -> list:
    """ Only takes the points that are in seen. """
    return [[p for p in points if p in seen]
            for points in pointsd]

def split(pointsd: list, cd: int, p: int) -> tuple:
    """ Splits by the plane x_cd = p[cd]. """
    left, right = set(), set()
    for point in pointsd[0]:
        if point != p:
            # add point with the same value as p
            # at cd to the right side
            (left if point[cd] < p[cd] \
             else right).add(point)
    return subsplit(pointsd, left), \
           subsplit(pointsd, right)
```

Pre-sort algorithm, code

Algorithm Pre-sort algorithm for kd-tree construction

```
def build_tree(t: kdNode, pointsd: list,
               cd: int=0) -> kdNode:
    N, D = len(pointsd[cd]), len(pointsd)
    t.D, t.cd = D, cd
    t.point = pointsd[cd][N//2] # median
    next_cd = (cd + 1) % D
    t.child = [build_tree(kdNode(), l, next_cd) \
               if len(l[0]) > 0 else None
               for l in splitd(pointsd, cd, t.point)]
    return t
```

Runtime Analysis

The running time is dominated by `subsplit`, which must split the D copies of N points. Determining whether a point is in the left or right set is at least an $O(D)$ operation since the hash needs to take into account each value of the point. There are $O(ND)$ checks, so `subsplit` runs in $O(D^2N)$. Over the $\log N$ levels of the tree, the pre-sort algorithm runs in $O(D^2 N \log N)$ (each level of the tree must split N points).

We can improve `subsplit`'s performance by noticing that we don't actually need to copy the points with all their dimensions; we can assign an arbitrary distinct ID to each point and maintain the D lists based off this integer ID. The simplest ID to use is the point's index in the points list, as to go from an ID to a point is just indexing the list. Since we only need the point values when comparing to the cutting dimension, we can find the cutting dimension in $O(1)$ by looking up the point and then indexing the point at that cutting dimension.

Runtime Analysis, continued

This optimization shaves off a D factor, so our running time goes from $O(D^2 N \log N)$ to $O(DN \log N)$ where the running time is dominated by the D initial $O(N \log N)$ sorts.
Recall that we need to do the sorts to find the median efficiently. Can we shave off another D factor if we find the median a different way?
We could maintain a single list of points, and simply sort this list on the cutting dimension at each level. That adds a $\log N$ factor at every level, so the running time is $O(N \log^2 N)$.
We could also just pick a random point to split on. This is equivalent to just calling `add` repeatedly on each point, so it has the same quadratic worst case running time.
Luckily, there is a way to find the median of a list in linear time!

Table of Contents

Order Statistics

The median is a special case of the problem of order statistics.

Order Statistics

The ith order statistic for a list l of n elements is the ith smallest value, i.e. `sorted(l)[i]`.

Special cases

For example, $i = 0$ is the minimum and $i = n - 1$ is the maximum.

The median for a list with an odd number of elements is at $i = \lfloor \frac{n}{2} \rfloor$. When n is even, then the median is ambiguous, with the "upper median" occurring at $\frac{n}{2}$ and "lower median" occurring at $\frac{n}{2} - 1$.
Since $\lfloor \frac{n}{2} \rfloor$ is always a median regardless of the parity of n, for simplicity "median" will always refer to the upper median.

Select Algorithm

We also initially assume that each element is distinct, although we will see what to do if that is not the case.
Suppose we are trying to find the ith order statistic.
The approach will be very similar to quicksort. We pick a pivot value, and split the list into two halves, the left with values less than the pivot and the right with values greater than the pivot (since we assume the elements are distinct, there are no elements equal to the pivot).

Select Algorithm, continued

We look at the size of the left list, which I'll call k.
If $k = i$, then the pivot is greater than i elements, so it is the ith order statistic by definition. Return the pivot.
If $i < k$, then our pivot is too big, so we recur on the left list. We keep the same value of i.
Finally, if $i > k$, then our pivot is not big enough so we recur on the right list. Unlike the left case, we already "beat" k elements so we need to look for the $i - k - 1$th element in the right list, where the 1 comes from the pivot value.
For simplicity, we use additional memory although the algorithm is able to be done in-place.
We also don't technically need a base case, but if the length of the list is 1, there's only one possible element to return.

Select, code

split and pivot selection

```
def split(l: list, x: float) -> tuple:
    """ Splits the list by a value x. """
    left, right = [], []
    for v in l:
        # if the value is equal to the cutoff,
        # add it to the right side
        (left if v < x else right).append(v)
    return left, right

def pivot(l: list) -> float:
    """ Picks a value as a pivot. """
    return l[0]
```

Select, code

Algorithm Select

```
def select(l: list, i: int):
    """ Returns sorted(l)[i]. """
    if len(l) == 1: # base case
        return l[0]
    left, right = split(l, pivot(l))
    k = len(left)
    if i == k: # pivot is the answer
        return right[0]
    # recur on sublist and get rid of pivot
    return select(left, i) if i < k else \
           select(right, i - k - 1)
```

Select, with duplicate elements

If there are duplicates, we could just run the standard select algorithm as usual. It has a problem though: it is sometimes impossible to get a good split if the median value has many duplicates - the many duplicates will carry over, slowing down the algorithm. The way we've implemented it, it could even infinitely recur!

Instead, we partition the list into *three* sublists: one for elements less than the pivot, one for elements equal to the pivot, and one for elements greater than the pivot (left, mid, and right, respectively).

Select, with duplicate elements, continued

Call the length of the left sublist k and the mid sublist m. Everything is exactly the same except instead of seeing whether $i = k$, we can return the pivot if $k \leq i \leq k + m - 1$ as the pivot value takes up more indexes: the first pivot value is greater than k elements, the second is greater than $k + 1$, and so on. (if $m = 1$, this reduces to $i = k$, like in the distinct case).
Also, if we recur on the right sublist we update i to $i - k - m$, since we remove k elements in the left sublist and m elements in the middle list. (if $m = 1$, this reduces to $i - k - 1$ like in the distinct case).

Select with duplicates, code

split modified to deal with duplicates

```
def split(l: list, x: float) -> tuple:
    """ Splits the list by a particular value x. """
    left, mid, right = [], [], []
    for v in l:
        (left if v < x else \
        right if v > x else mid).append(v)
    return left, mid, right
```

Select with duplicates, code

Algorithm Select, modified to deal with duplicate elements

```
def select(l: list, i: int):
    """ Returns sorted(l)[i]. """
    if len(l) == 1: # base case
        return l[0]
    left, mid, right = split(l, pivot(l))
    k, m = len(left), len(mid)
    if k <= i <= k + m - 1: # pivot is the answer
        return mid[0]
    # recur on sublist and get rid of pivot
    return select(left, i) if i < k else \
           select(right, i - k - m)
```

Runtime Analysis

The runtime is analyzed very similarity to quicksort or kd-tree construction. As usual, the selection of the pivot is very important. In the best case, we split the list perfectly in half every iteration, so we take $N + \frac{1}{2}N + \frac{1}{4}N + \ldots = 2N \rightarrow O(N)$. In the worst case, we remove a single element every time, so we take $N + (N-1) + (N-2) + \ldots = \frac{N(N+1)}{2} \rightarrow O(N^2)$. Importantly, as long as we split the list by some constant multiple less than 1, the geometric series will converge into $O(n)$. For example, for a 0.9 split where we remove 10%:

$$N + 0.9N + 0.81N + \ldots = \frac{1}{1-0.9}N = 10N \rightarrow O(N)$$

Is there a heuristic that guarantees a constant multiple?

Table of Contents

Median of Medians

Median of Medians

1. Divide the list into groups of 5, putting the remainder in a group of length $n \mod 5$.
2. Find the median of each group of 5 with any method (including sorting)
3. Find the median of the medians found in step 2
4. Use this median as a pivot in `select`

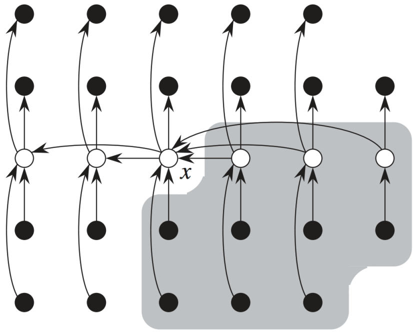
Algorithm Median of medians

```
def median(l: list) -> float:
    """ Returns the median of l, via a sort. """
    return sorted(l)[len(l)//2]

def pivot(l: list) -> float:
    """ Uses the median of medians as a pivot. """
    medians = [median(l[5*i: 5*(i + 1)])
               for i in range(-(-len(l)//5))]
    return select(medians, len(medians)//2)
```

Runtime Analysis

Why does this work? Call the median of medians x.



Half of the groups' medians must be less or equal to x by definition of the median. For each of these groups, the median and the two elements in the group less than the median are also less than or equal to x, contributing 3 elements per group.

Runtime Analysis, continued

There are $\lceil \frac{n}{5} \rceil$ groups in total, but for simplicity we ignore the group with x and the group with less than 5 elements, so the number of elements less than x is at least $3(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2)$ or bounded below by $\frac{3n}{10} - 6$.
Thus, in the worst case we recur on a list of size
$n - (\frac{3n}{10} - 6) = \frac{7n}{10} + 6$
This is basically a constant multiple! Don't get too excited just yet, we added an additional recursive step because `pivot` calls `select` to find the median of a list of size $\lceil \frac{n}{5} \rceil$.

Runtime Analysis, continued

First of all, we can decompose a list into groups of 5 and compute the median of each group in linear time, since a sort of a list of size 5 is $O(1)$ and we perform $\lceil \frac{n}{5} \rceil$ such sorts. Thus, this median decomposition adds no asymptotic overhead to the existing linear time partition step.

If $T(n)$ is the runtime of the algorithm on a list of size n, it fulfills the recurrence relation

$$T(n) = O(n) + T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10} + 6)$$

Is $T(n)$ $O(n)$? According to *Introduction to Algorithms*, yes! We can find the median of a list in linear time.

Table of Contents

Decision Tree

In practice, there's a large constant factor that could be optimized if we could find the median of a list of size 5 quickly. We could model this problem as a decision tree, where each internal node contains a comparison, represented by two indexes to compare in the array, i.e. `a[i]` $>$ `a[j]`. If the comparison is false, we recur on the left node and if the comparison is true, we recur on the right node. We continue until we reach a leaf node, which simply contains the index of the median.

Decision Tree, construction

Suppose we build a decision tree to find the median of a particular list containing n arbitrary and distinct elements.

Claim: If the decision tree works on all $n!$ permutations of this list, then it works on *any* list of n elements, even lists that contain duplicate elements.

Intuitively, we don't care about the actual values in the lists; if the relative comparisons between the indexes are the same then we will take the same path down the tree.

Each set of "relative comparisons" defines an ordering of the list, or a permutation. Thus, all lists have already been "covered" by an corresponding permutation of our particular list.

What about duplicate elements?

I don't care how duplicate values compare to each other, because that just determines the relative order of duplicates, which is arbitrary. I can just pick any permutation that is correct for distinct value comparisons.

Decision Tree, construction example

The simplest particular list to pick is $a = [0, 1, 2, 3, 4]$. Suppose we have a decision tree which correctly identifies the median index for each permutation of a.

Will this give the correct median for $x = [0, 0.3, 0.2, 0.4, 0.1]$? Well, we can define the mapping $0 \to 0$, $0.1 \to 1$, and so on. We know the decision tree works for $[0, 3, 2, 4, 1]$, a permutation of a. Thus, it should work for x.

In general, we can construct this mapping by sorting the list and mapping the value at $x[0]$ to 0, $x[1]$ to 1, and so on. Thus, any list x has a corresponding permutation of a with the same relative comparisons between any pair of indexes.

If x has duplicate entries, then we can still apply the sorting algorithm to generate a mapping. Of course, the corresponding permutation of a *won't* have the same relative comparisons, but this is acceptable because relative comparisons between duplicate elements is arbitrary.

Decision Tree, properties

There are two important properties of this decision tree:

1. The height of the tree, which gives the number of comparisons in the worst case.
2. The expected number of comparisons, assuming each permutation of the list occurs with equal probability.

Of course, we want to minimize both height and expected number of comparisons. How do we actually build such a decision tree?

Decision Tree, construction

Likely NP-hard in general, at least decision trees in a machine learning context greedily maximize information gain at each level instead of trying to globally optimize information gain.

We could just generate every possible decision tree. $N = 5$, how hard could it be? Well, I have no clue because it takes at least 30 minutes, at which point I terminated the program.

We could prune trees by considering *isomorphic* trees, that is, we can take advantage of symmetry (for example, the very first comparison is necessarily symmetric, since there is no difference between any two pair of indexes).

Instead, we'll use the greedy approach, and pick comparisons that split the permutations between the left and right subtrees as evenly as possible (very similar to kd-tree construction).

Decision Tree, application

Once we have a decision tree, we can render it into Python.

```
median
def median(l: list) -> float:
    """ Computes the median of l, if len(l) == 5. """
    a, b, c, d, e = l
    return (((((((c if b < c else b) if b < d else d) if d < e else ((c if b < c else b) if
↪   b < e else e)) if c < e else (((e if b < e else b) if b < c else c) if a < e else
↪   (b if b < c else c))) if a < c else (((b if b < d else (d if a < d else a)) if b <
↪   e else ((d if a < d else a) if d < e else e)) if a < e else (a if a < d else ((e if
↪   d < e else d) if c < e else d))))) if c < d else (((((d if b < d else b) if b < c
↪   else c) if c < e else ((d if b < d else b) if b < e else e)) if d < e else (((e if
↪   b < e else b) if b < d else d) if a < e else ((b if b < d else d) if b < c else
↪   d))) if a < d else (((b if b < c else (c if a < c else a)) if b < e else ((c if a <
↪   c else a) if c < e else e)) if a < e else (a if a < c else (e if c < e else c)))))
↪   if a < b else (((((c if c < e else e) if a < c else a) if a < e else ((e if c < e
↪   else (a if a < c else c)) if b < e else ((a if a < c else c) if b < c else b))) if
↪   a < d else (((d if b < c else (d if b < d else b)) if a < e else (d if b < d else
↪   (b if b < e else e))) if d < e else ((e if b < e else (b if b < d else d)) if c < e
↪   else (c if b < c else (b if b < d else d))))) if c < d else ((((d if c < e else (d
↪   if d < e else e)) if a < d else a) if a < e else ((e if d < e else (a if a < d else
↪   d)) if b < e else ((a if a < d else d) if b < d else b))) if a < c else (((c if b <
↪   c else b) if a < e else (c if b < c else (b if b < e else e))) if c < e else ((e if
↪   b < e else (b if b < c else c)) if d < e else (d if b < d else (b if b < c else
↪   c)))))))
```

Decision Tree, disappointment

First, decision trees use on average 1.5 less comparisons than sorting, and use at most 7 comparisons to find the median:

```
Average value: 6.267, max depth: 7
Python sorted comparisions: 7.775
```

According to timeit, decision trees win over sorting!

```
ternary: 0.266674
   sort: 0.380404
```

The difference is magnified with PyPy, about 26x faster:

```
ternary: 0.002689
   sort: 0.070296
```

In practice, however, decision trees are slower.

Table of Contents

kd-Tree construction

Back to kd-Trees. In order to use the linear time median algorithm derived in the last section, we first generate a list of scalars from our list of points by indexing each point at the current cutting dimension.

We then apply our median algorithm, and obtain a median *value* in linear time. Finally, we iterate through the points again, and pick any point with the median value along the cutting dimension.

Our construction complexity is now $O(n \log n)$, since we avoid the initial D sorts. Note how D does not appear in the complexity. Claim: This complexity for (optimal) kd-tree construction is asymptotically optimal.

Nearest Neighbor

Nearest Neighbor Query

A nearest neighbor query is given a point Q and a set of points P, find the closest point to Q in P.

Suppose we build a kd-tree on P. A kd-tree is helpful in the sense that it gives us spatial data, as it successively partitions a space on hyperplanes defined by points in the tree.
We can take advantage of this by conducing a tree search, with two important modifications:

1. Keep track of the closest point so far, C. Prune subtrees if they can't beat this closest point.
2. Search subtrees in a order that maximizes pruning

Bounding Boxes

Each subtree has a bounding box, or the possible values it could take on each dimension (i.e., a minimum and maximum value for each dimension). If the distance between Q and the closest point in this bounding box is greater than the distance between Q and C, then there's no point to search this subtree.

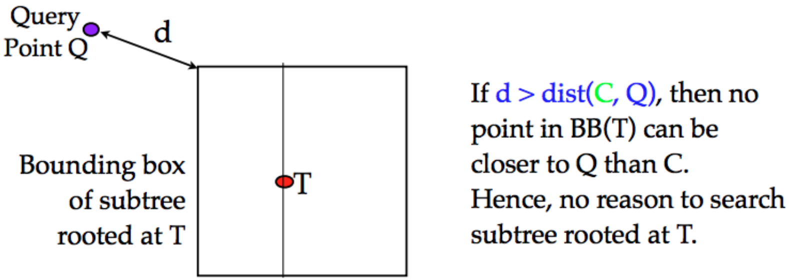

Query Point Q
d
Bounding box of subtree rooted at T
T
If d > dist(C, Q), then no point in BB(T) can be closer to Q than C. Hence, no reason to search subtree rooted at T.

Figure: Justification for pruning subtrees

Bounding Boxes, distance

Question 1: How do we compute the distance between a point and a bounding box? Let the bounding box be a list of tuples, each tuple being the minimum and maximum value on that dimension.

Bounding Box

The bounding box $[(0,5),(-2,3)]$ defines a rectangle in the x-y plane, where x can be between 0 and 5 and y can be between -2 and 3.

Bounding Boxes, distance

As stated previously, the distance between a bounding box and a point Q is the distance between Q and the closest point in the bounding box to Q. We can find this "closest point" by considering each dimension separately, since in Euclidean distance each dimension is independent of the others. Suppose we are on dimension d. We have three cases to consider:

1. $Q[d] < bb[0]$, i.e. the point is left of the bounding box. In this case, we pick $bb[0]$ along this dimension.
2. $bb[0] \leq Q[d] \leq bb[1]$, i.e. the point is in bounding box. We can just use $Q[d]$ since it is contained.
3. $Q[d] > bb[1]$, i.e. the point is right of the bounding box. Similar to the first case, we use $bb[1]$.

Bounding Box distance, code

distance from a bounding box

```
def distbb(p: tuple, bb: list) -> float:
    bbp = tuple(box[0] if x < box[0] else \
                (box[1] if x > box[1] else x)
                 for x, box in zip(p, bb))
    return dist(p, bbp)
```

Bounding Boxes, computation

Question 2: How do we keep track of these bounding boxes? Well, we could say the initial bounding box at the root is completely unbounded, i.e. $(-\infty, \infty)$ on each dimension. When we traverse the left and right subtrees, we must have split on some plane.

Maintaining bounding boxes

Suppose we split on the value 5 along cutting dimension 1. Then for the left subtree we update $bb[1]$ to be $(-\infty, 5)$, and for the right we update $bb[1]$ to be $(5, \infty)$.

Note that these bounds are known *a posteriori*, i.e. generated online during the nearest neighbor search.

Bounding Box maintenance, code

update bounding box

```
def trimbb(bb: list, cd: int, p: int, d: int) -> list:
    if len(bb) == 0: return bb
    bb = list(list(box) for box in bb)
    bb[cd][1 - d] = p[cd]
    return bb
```

Subtree Order

Lastly, we need to determine the order to search the subtrees. It makes sense to first visit the subtree we would visit if we were inserting the point in the kd-tree, i.e. the subtree which would contain the point.

Nearest Neighbor, code

Algorithm Nearest Neighbor Query

```
def __closest(self, t: "kdNode", p: tuple, bb: list) -> tuple:
    # bounding box too far away from point
    if t is None or distbb(p, bb) > self.best_dist:
        return
    # update best point
    d = dist(p, t.point)
    if d < self.best_dist:
        self.best, self.best_dist = t.point, d
    # visit subtrees in order of distance from p
    i, j = t.dir(p), 1 - t.dir(p)
    self.__closest(t.child[i], p, trimbb(bb, t.cd, t.point, i))
    self.__closest(t.child[j], p, trimbb(bb, t.cd, t.point, j))

def closest(self, p: tuple) -> tuple:
    self.best, self.best_dist = None, float("inf")
    bb = [[-float("inf"), float("inf")] for d in range(len(p))]
    self.__closest(self, p, [] if self.tight_bb else bb)
    return self.best
```

Runtime Analysis

In the worst case, we need to traverse the entire tree, $O(n)$. In practice the runtime is closer to

$$O(\underbrace{2^d}_{\text{points in the neighborhood}} + \underbrace{\log n}_{\text{points ``near'' query}})$$

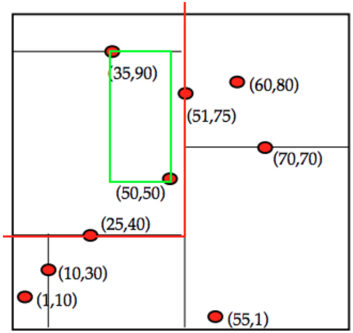
If d is small, this is faster than $O(nd)$ per query with the naive method of searching every point.

Note that we in fact introduce another d factor when we trim the bounding boxes. However, the bounding boxes are the same regardless of the point we're traversing the tree on, since the bounding boxes are a function of the tree which doesn't change.

Bounding Box, tight

We can pre-compute bounding boxes, also taking advantage of “tighter” boxes. The bounding boxes generated by the plane trim method generate boxes that are too big - the real bounds are determined by the extrema of the *points* contained in the subtree, not just the path to the subtree.

Bounding Box, tight



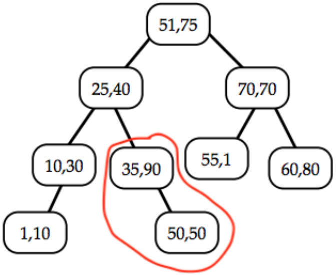


Figure: Difference between the default (in red) and tight bounding boxes (in green). Subtree is circled in red.

Bounding Box, tight, computation

The observation is that if we have a bounding box for both children of a node, we can merge these efficiently since going up the tree means the bounding boxes join together.
We have 3 cases:

1. If we're a leaf, the only point we contain is the leaf's point. This serves as our bounding box (it just contains the point).
2. If we have exactly one child, then we copy its bounding box and add the current node's point as well.
3. If we have two children, we merge their bounding boxes and also add the current node's point.

Bounding Box, tight, code

Algorithm Tight bounding boxes

```
def tighten(self, t: "KdNode"=None) -> None:
    if t is None: t = self # called with None, set to the root
    l, r, t.tight_bb = t.child[0], t.child[1], True
    # recur on children
    if l is not None: self.tighten(l)
    if r is not None: self.tighten(r)
    # leaf node, box is just the singular point
    if l is None and r is None:
        t.bb = [(t.point[d], t.point[d]) for d in range(t.D)]
    # one child, inherit box of child
    elif l is None or r is None:
        t.bb = l.bb if l is not None else r.bb
        # add node's point
        t.bb = [(min(box[0], v), max(box[1], v))
                for box, v in zip(t.bb, t.point)]
    # two children, combine boxes
    else:
        t.bb = [(min(bbl[0], bbr[0], v), max(bbl[1], bbr[1], v))
                for bbl, bbr, v in zip(l.bb, r.bb, t.point)]
```

Bounding Box, tight, runtime analysis

After constructing a kd-tree, we can run `tighten` on the tree to generate bounding boxes for each node.
`tighten` runs in $O(ND)$ since we visit each node in the tree, and at each node we do $O(D)$ operations to do accounting on the bounding box.
We assume that $\log N > D$ or $N > 2^D$, if D is very large or N very small then kd-trees are not a good choice. Thus, `tighten` is dominated by the $O(N \log N)$ initial construction time.
We can run a nearest neighbor search as usual, with the bounding box trimming logic removed.

Nearest Neighbor, code

Algorithm Nearest Neighbor Query with tight bounds

```
def __closest(self, t: "kdNode", p: tuple) -> tuple:
    # bounding box too far away from point
    if t is None or distbb(p, t.bb) > self.best_dist:
        return
    # update best point
    d = dist(p, t.point)
    if d < self.best_dist:
        self.best, self.best_dist = t.point, d
    # visit subtrees in order of distance from p
    i, j = t.dir(p), 1 - t.dir(p)
    self.__closest(t.child[i], p)
    self.__closest(t.child[j], p)

def closest(self, p: tuple) -> tuple:
    self.best, self.best_dist = None, float("inf")
    self.__closest(self, p)
    return self.best
```

Runtime Analysis

First, we save the D factor because we no longer need to trim bounding boxes, each node stores its boxes known *a priori*.
If we determine the bounding boxes based off the points in the tree, then it is a subset of the original bounding box. Thus, the distance between any point and our tight bounding box must be greater than or equal to the distance between the point and the original bounding box. We prune if the distance between the bounding box is greater than the best distance, so our tight bounding box can only prune more since it doesn't change the best distance found so far.
Pruning more means nearest neighbor searches run even faster.

Wrapping it All Up

We finally apply kd-trees to speeding up k-means/k-means++. Simply build a kd-tree on the centers, re-building every time the centers change. Whenever we need to find the closest center to a point, we do a nearest neighbor query with the kd-tree.

Per iteration, our running time is

$$O(\underbrace{K \log K}_{\text{kd-tree construction}} + \underbrace{N \log K}_{N \text{ nearest-neighbor queries}})$$

$N \geq K$ so this is $O(N \log K)$ per iteration, compared to $O(NK)$ for the naive algorithm.

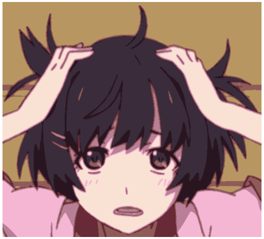
This is overly optimistic.

1. Building a kd-tree naturally adds more memory consumption and overhead
2. If D is very large, kd-trees become impractical
3. kd-tree is *expected* $O(\log K)$, it could be $O(K)$

TODO: Voronoi diagrams?

Application of k-means

(a) Original image


(b) $k = 8$

(c) $k = 256$

Figure: k-means on an image. Can you tell the difference at $k = 256$?

The difference between naive and kd-trees gets larger as k increases. For $k = 8$, kd-trees are roughly 2x slower, and for $k = 256$, kd-trees are roughly 2x faster.

References

1. Wikipedia articles on color quantization, k-means, and k-means++
2. Stanford k-means handout
3. CMU kd-Trees
4. CMU kd-Trees Continued
5. *Introduction to Algorithms*, chapter 9
6. Shiny Shaymin image
7. Regular Shaymin image