Tufts

Class #08: Non-Parametric Learning: Clustering with Neighbors

Machine Learning (COMP 135): M. Allen, 30 Sept. 19

Parametric Learning Methods

- So far, the regression/classification methods we have seen are all parametric
- Each method assumes that there is some fixed set of weights that is to be learned:

1. Linear weights in linear/logistic regression/classification:

$$w_0 + w_1\,x_1 + w_2\,x_2 + \cdots + w_n\,x_n$$

2. Non-linear weights in polynomial regression

$$w_0 + w_1\,x_1 + w_2\,x_1^2 + \cdots + w_{2n-1}\,x_n + w_{2n}\,x_n^2$$

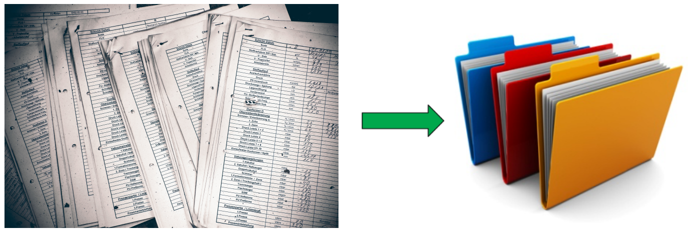
Non-Parametric Learning Methods

- In a parametric process, once the learning is done, the weight parameters are saved, and we are effectively done
  - We can ***throw out*** training data, and just record weights
- Not every problem has this feature: in some, learning is always continuing, and is based on all examples so far
- These non-parametric methods have no fixed set of weights (or other numbers) to memorize
  - As data continues to come in, we continue to adjust the model, in the middle of our classification task

Clustering Problems

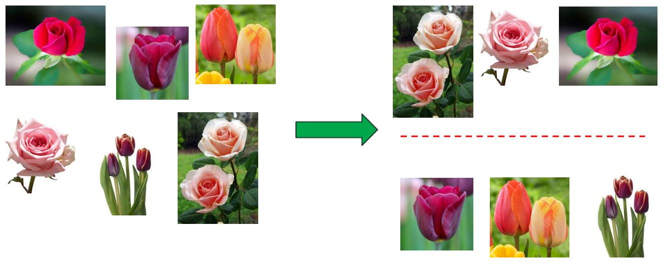
- The other techniques we have seen are all supervised
  - We have training data for which we ***know*** the appropriate output, and minimize some loss function based on this
- In many cases, we want to work with unlabeled data
  - We want to take data and group them together
  - Data should end up in a group with other "similar" data
  - We want to find clusters, without knowing the correct answer ahead of time
- This requires us to give precise meaning to similarity
  - We also need efficient ways to do the grouping

Examples of Clustering



- One use of clustering might be for document processing
  - We have many, many different documents in a database
  - We want to organize them into groups based on subject matter
  - We don't know what the different subjects are ahead of time

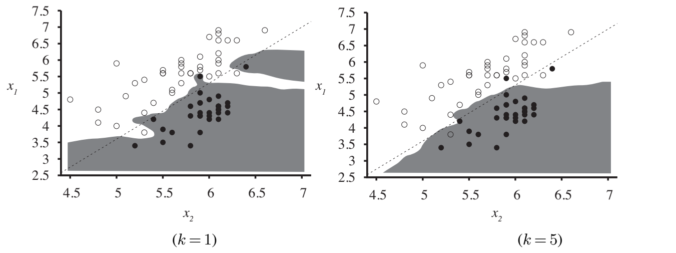
Examples of Clustering



- Another use of clustering might be for image processing
  - We have many, many different images of flowers
  - We want to organize them into groups of specific flowers
  - We don't know what the different flowers are ahead of time

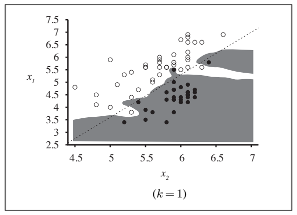
Nearest Neighbor Models

Image source: Russel & Norvig, *AI: A Modern Approach* (Prentice Hal, 2010)



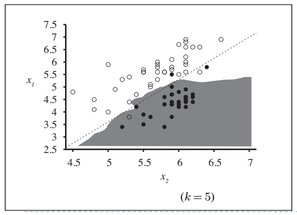
- One idea: classify inputs so any input x has ***same class*** as its k ***nearest neighbors***
- Depending upon the size of k, we may get over-fitting
- We can do techniques similar to those for linear regression to figure out what the best value of k might be

Nearest Neighbor Models



Here, the classes are set so that each point is in the same set as its ***nearest single neighbor***.

This is very likely to be ***over-fitting*** on data. New inputs will often end up in the wrong class in the future due to the odd contours of the boundary between classes.



Here, the classes for each point in the problem space is set to have the same class as the ***majority of its nearest 5 neighbors***.

Using more neighbors likely to ***under-fit*** data.

Obviously, if we use value $k = N$ (the total number of examples), then we just lump everything into a single category.

Measuring Distance between Neighbors

- Suppose we have two inputs, each with n features

$$\mathbf{x}_i = \langle x_{i,1},\ x_{i,2},\ \ldots,\ x_{i,n} \rangle$$

$$\mathbf{x}_j = \langle x_{j,1},\ x_{j,2},\ \ldots,\ x_{j,n} \rangle$$

- Each can be regarded as a point in an n-dimensional space
- We can measure the distance between those points using the Minkowski distance (aka L^p norm):

$$L^p(\mathbf{x}_i, \mathbf{x}_j) = (\sum_{k=1}^{n} |x_{i,k} - x_{j,k}|^p)^{1/p}$$

- This works best if we ***normalize*** each dimension $x_{i,n}$

Geometrical Distance Metrics

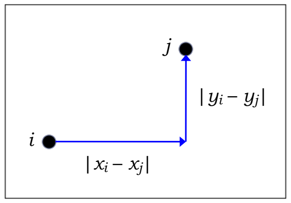
$$L^p(\mathbf{x}_i, \mathbf{x}_j) = (\sum_{k=1}^{n} |x_{i,k} - x_{j,k}|^p)^{1/p}$$

- The Minkowski distance extends certain intuitive distance numbers to different numbers of dimensions, n
  - Depending upon the power, p, we get different measures
- When $p = 1$, this is the ***Manhattan Distance***:

$$L^1(\mathbf{x}_i, \mathbf{x}_j) = \sum_{k=1}^{n} |x_{i,k} - x_{j,k}|$$

- In a two dimensional (x, y) space this is:

$$|x_i - x_j| + |y_i - y_j|$$



Geometrical Distance Metrics

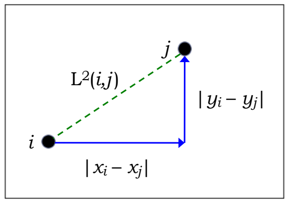
$$L^p(\mathbf{x}_i, \mathbf{x}_j) = (\sum_{k=1}^{n} |x_{i,k} - x_{j,k}|^p)^{1/p}$$

- Minkowski distance extends certain intuitive distance numbers to different numbers of dimensions, n
  - Depending upon the power, p, we get different measures
- When $p = 2$, this is the ***Euclidean Distance***:

$$L^2(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{k=1}^{n} |x_{i,k} - x_{j,k}|^2}$$

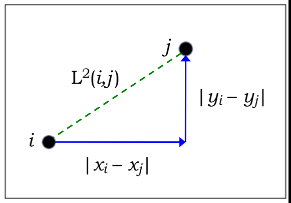
- In a two dimensional (x,y) space this is:

$$\sqrt{|x_i - x_j|^2 + |y_i - y_j|^2}$$



Comparative Distance Metrics

$$\sqrt{|x_i - x_j|^2 + |y_i - y_j|^2}$$



- The Euclidean distance can be extended to 3 or more dimensions
- Computationally, we can actually make our lives somewhat easier…
- To do clustering, we only care about relative distances
  - Doesn't matter what the ***actual distance*** is
  - We really only care about which things are ***closest*** together
  - This means we can ***skip*** the square root computation entirely

Finding Nearest Neighbors

- Naïve implementations of these measures can be problematic
- For n dimensions each comparison of two points requires $O(n)$ operations, which is ***often*** reasonable
- ***However***, the classifications work best when we have large amounts of data, relative to the number of dimensions
  - Ideally, we have $O(2^n)$ input points
  - Much smaller numbers tend to lead to poor classifications, due to large numbers of ***outliers***
- ***However***, if we simply compare ***all pairs*** of points, we have $O(|X|^2)$ such operations, where $|X|$ is full size of data-set
  - This can be much too cumbersome for large data-sets

KD-Trees: Efficient Neighbor Calculation

```
function BUILD-TREE($X$, $d$) returns a tree
    inputs: $X = \{\mathbf{x}_1 \ldots, \mathbf{x}_m\}$, a set of $n$-dimensional data-points, and depth $d$
    local variables: $S \geq 1$, a pre-set size limit for sets
    if $|X| \leq S$ :
        return : $Node(X)$, a tree-node containing all elements of $X$
    else :
        $\delta \leftarrow (d \bmod n) + 1$   (the dimension for splitting inputs)
        $m_\delta \leftarrow$ the median for dimension $\delta$ in $X$
        $X^- \subseteq X \leftarrow$ the set of all data-points $\mathbf{x}_i \leq m_\delta$ for dimension $\delta$
        $X^+ \subseteq X \leftarrow$ the set of all data-points $\mathbf{x}_j > m_\delta$ for dimension $\delta$
        $\mathbf{N}^\circ \leftarrow Node(m_\delta)$, a tree-node containing median-value $m_\delta$
        $\mathbf{N}^\circ_{left} \leftarrow$ BUILD-TREE($X^-$, $d+1$)
        $\mathbf{N}^\circ_{right} \leftarrow$ BUILD-TREE($X^+$, $d+1$)
        return : $\mathbf{N}^\circ$
```

- We can build a data-structure to search for nearest neighbors efficiently
- A recursive algorithm, called on original data set, X:
  BUILD-TREE(X, 0)

K-D Trees: Efficient Neighbor Calculation

```
function BUILD-TREE($X$, $d$) returns a tree
    inputs: $X = \{\mathbf{x}_1 \ldots, \mathbf{x}_m\}$, a set of $n$-dimensional data-points, and depth $d$
    local variables: $S \geq 1$, a pre-set size limit for sets
    if $|X| \leq S$ :
        return : $Node(X)$, a tree-node containing all elements of $X$
    else :
        $\delta \leftarrow (d \bmod n) + 1$   (the dimension for splitting inputs)
        $m_\delta \leftarrow$ the median for dimension $\delta$ in $X$
        $X^- \subseteq X \leftarrow$ the set of all data-points $\mathbf{x}_i \leq m_\delta$ for dimension $\delta$
        $X^+ \subseteq X \leftarrow$ the set of all data-points $\mathbf{x}_j > m_\delta$ for dimension $\delta$
        $\mathbf{N}^\circ \leftarrow Node(m_\delta)$, a tree-node containing median-value $m_\delta$
        $\mathbf{N}^\circ_{left} \leftarrow$ BUILD-TREE($X^-$, $d+1$)
        $\mathbf{N}^\circ_{right} \leftarrow$ BUILD-TREE($X^+$, $d+1$)
        return : $\mathbf{N}^\circ$
```

Each time we go deeper down the tree, we cycle to the next data feature

- Input parameter d sets the feature we use to divide data into separate subsets
- We cycle through these features: $x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_{n-1} \rightarrow x_n \rightarrow x_1 \rightarrow \cdots$

K-D Trees: Efficient Neighbor Calculation

```
function BUILD-TREE($X$, $d$) returns a tree
    inputs: $X = \{\mathbf{x}_1 \ldots, \mathbf{x}_m\}$, a set of $n$-dimensional data-points, and depth $d$
    local variables: $S \geq 1$, a pre-set size limit for sets
    if $|X| \leq S$ :
        return : $Node(X)$, a tree-node containing all elements of $X$
    else :
        $\delta \leftarrow (d \bmod n) + 1$   (the dimension for splitting inputs)
        $m_\delta \leftarrow$ the median for dimension $\delta$ in $X$
        $X^- \subseteq X \leftarrow$ the set of all data-points $\mathbf{x}_i \leq m_\delta$ for dimension $\delta$
        $X^+ \subseteq X \leftarrow$ the set of all data-points $\mathbf{x}_j > m_\delta$ for dimension $\delta$
        $\mathbf{N}^\circ \leftarrow Node(m_\delta)$, a tree-node containing median-value $m_\delta$
        $\mathbf{N}^\circ_{left} \leftarrow$ BUILD-TREE($X^-$, $d+1$)
        $\mathbf{N}^\circ_{right} \leftarrow$ BUILD-TREE($X^+$, $d+1$)
        return : $\mathbf{N}^\circ$
```

Data divides along the median value of the chosen feature

- Once a feature is chosen, we find the median value for that feature, and divide all data in two at that median point

K-D Trees: Efficient Neighbor Calculation

```
function BUILD-TREE$(X, d)$ returns a tree
    inputs: $X = \{\mathbf{x}_1 \ldots, \mathbf{x}_m\}$, a set of $n$-dimensional data-points, and depth $d$
    local variables: $S \geq 1$, a pre-set size limit for sets

    if $|X| \leq S$ :
        return : $Node(X)$, a tree-node containing all elements of $X$
    else :
        $\delta \leftarrow (d \bmod n) + 1$   (the dimension for splitting inputs)
        $m_\delta \leftarrow$ the median for dimension $\delta$ in $X$
        $X^- \subseteq X \leftarrow$ the set of all data-points $\mathbf{x}_i \leq m_\delta$ for dimension $\delta$
        $X^+ \subseteq X \leftarrow$ the set of all data-points $\mathbf{x}_j > m_\delta$ for dimension $\delta$

        $\mathbf{N}^\circ \leftarrow Node(m_\delta)$, a tree-node containing median-value $m_\delta$
        $\mathbf{N}^\circ_{left} \leftarrow$ BUILD-TREE$(X^-, d+1)$
        $\mathbf{N}^\circ_{right} \leftarrow$ BUILD-TREE$(X^+, d+1)$

        return : $\mathbf{N}^\circ$
```

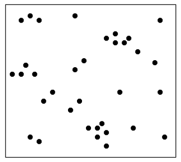
Recursion ends when data subsets are small enough

Recursive calls build a binary tree, branch by branch

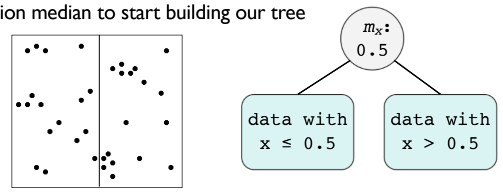
- Recursively builds a binary tree, with sub-tree roots each containing a median value
- Recursion terminates whenever we hit a pre-determined minimum data-set size

A 2-Dimensional Example

- We start with a set of 2-dimensional data-points, $p_i = (x_i, y_i)$

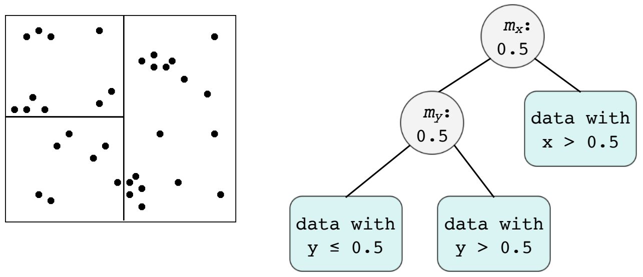


- Split along x-dimension median to start building our tree



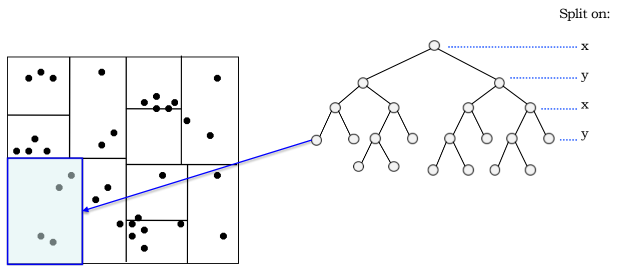
A 2-Dimensional Example

- We then split each group along y-dimension median separately



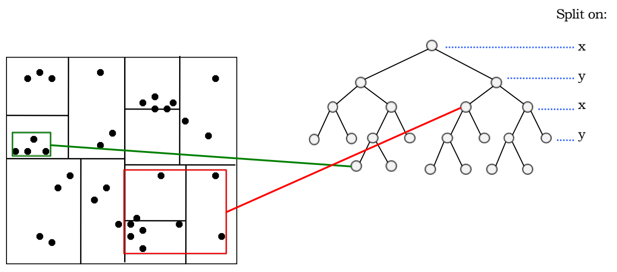
- We repeat on the other branch, and continue in each case, ***splitting again*** on dimensions $x, y, x, y, x, y, \ldots$

A 2-Dimensional Example



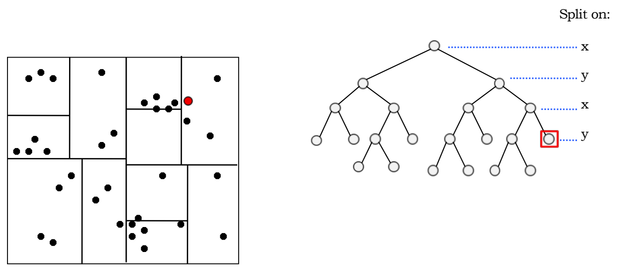
- We stop when we have small enough subsets, each of which is stored in and represented by a leaf-node of our tree
- Interior nodes store median values

A 2-Dimensional Example



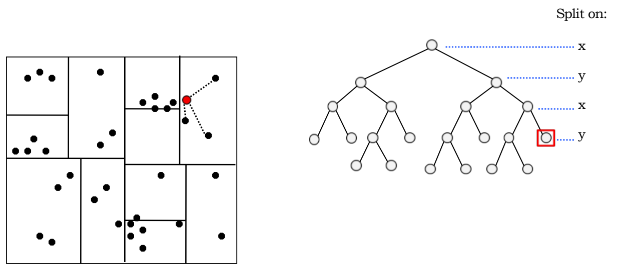
- Interior nodes store median values
- Each node (leaf ***or*** interior) also stores information about the least (tightest) bounding box of all points below it in its sub-tree

Querying the Tree for the Nearest Neighbor



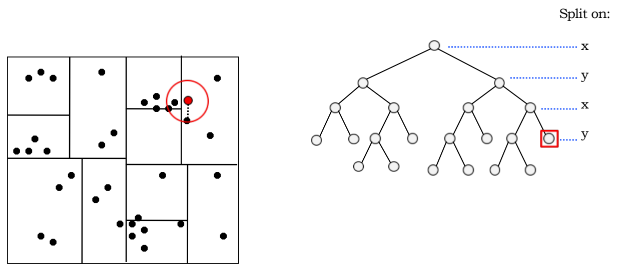
- Suppose we want to find the nearest neighbor of a new data-point (red)
- We start by isolating what sub-set it belongs to, following branches according to the median values (like a binary search tree)

Querying the Tree for the Nearest Neighbor



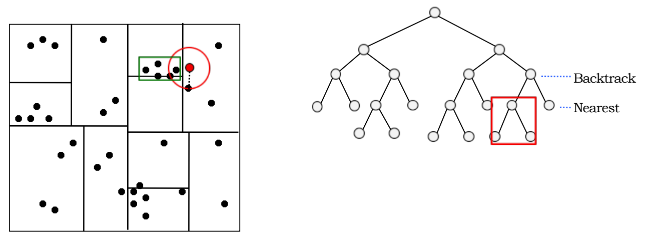
- Once we have found the proper subset, we measure all distances within it
- The closest neighbor ***may be*** in this set, but it ***may not***

Querying the Tree for the Nearest Neighbor



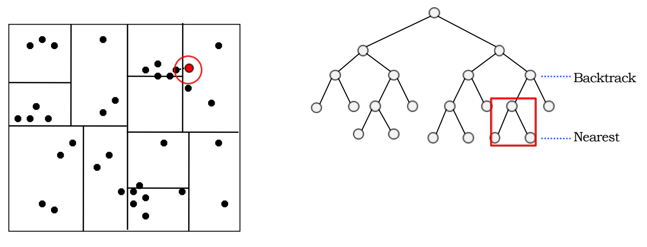
- We need to check any data-point that ***could be closer*** to our new point
- The tree helps us here, as we can do some ***pruning*** as we go backwards up the tree towards the root

Querying the Tree for the Nearest Neighbor



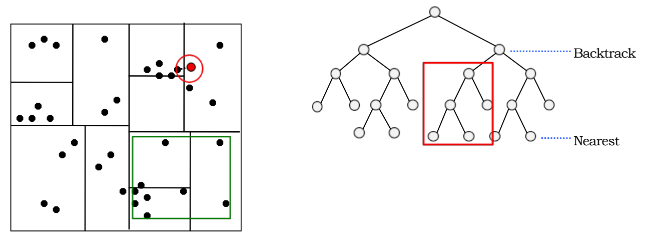
- As we back-track up the tree, we check any branch where the stored bounding box intersects our current bounds

Querying the Tree for the Nearest Neighbor



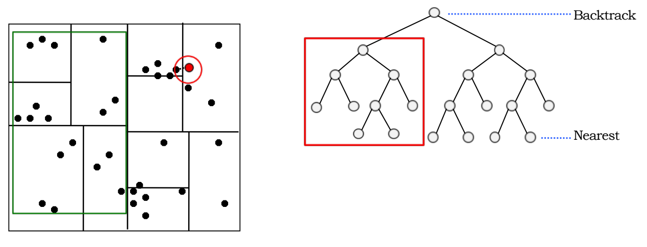
- When this happens, we compute distances to all required nodes, and update distance measure if necessary

Querying the Tree for the Nearest Neighbor



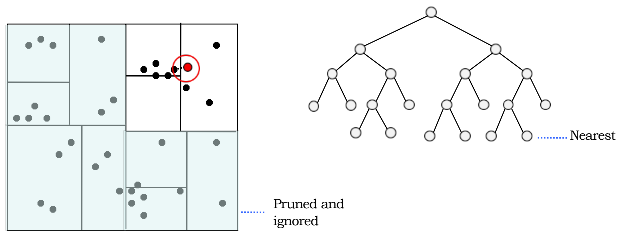
- When we see a sub-tree with a bounding box that ***does not*** intersect our current bound, we can ***ignore it***, saving time overall

Querying the Tree for the Nearest Neighbor



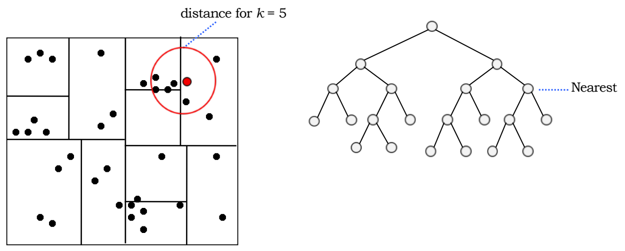
- When we see a sub-tree with a bounding box that ***does not*** intersect our current bound, we can ***ignore it***, saving time overall
- Once we are at the root, we have found the overall nearest neighbor

Querying the Tree for the Nearest Neighbor



- The data-structure may allow us to prune off large numbers of nodes, restricting those that we need to measure distance from new point
- Although it is possible that we still have to do $O(N)$ comparisons, under many distributions of data-points, we get $O(\log N)$, significantly speeding up our algorithm for classification

K-D Trees for k-Nearest Neighbors



- If what we want is not the single nearest neighbor point, but some set of k such points (for better classification), the ***exact same approach*** can be used
- Works the same way, but the distance measure is set to use the full set of neighbors (i.e., distance to ***farthest one*** of the k nearest)

Uses of Nearest Neighbors

- Once we have found the k-nearest neighbors of a point, we can use this information:

1. ***In and of itself***: sometimes we just want to know what those nearest neighbors actually are (items that are similar to a given piece of data)
2. ***For additional classification purposes***: we want to find the nearest neighbors in a set of *already-classified* data, and then use those neighbors to classify new data
3. ***For regression purposes***: we want to find the nearest neighbors in a set of points for which we *already know* a functional (scalar) output, and then use those outputs to generate the output for some new data

This Week

- Nearest Neighbors: Clustering and Regression
- Readings:
  - Linked from class website schedule page.
- Homework 02: due Wednesday, 02 October, 9:00 AM
- Office Hours: 237 Halligan, Tuesday, 11:00 AM – 1:00 PM
  - TA hours can be found on class website as well