Nearest Neighbour Methods

KNN Classification, KNN Regression and Efficient Distance Computation Using KD Trees

What is it useful for?

- In the absence of a model, then the best thing to do is to look at similar data and choose to be in the same class as them
- We have to calculate the distance between the data points in the sample space and the new data (test data) then assign its class
- If we use Euclidean distances then we have d subtractions and d squarings (d being the dimensionality) and we have to do this for N x N data points which leads to $O(N^2)$ complexity
- So as the number of dimensions increase, the time taken for this method suffers (curse of dimensionality)

K Nearest Neighbours

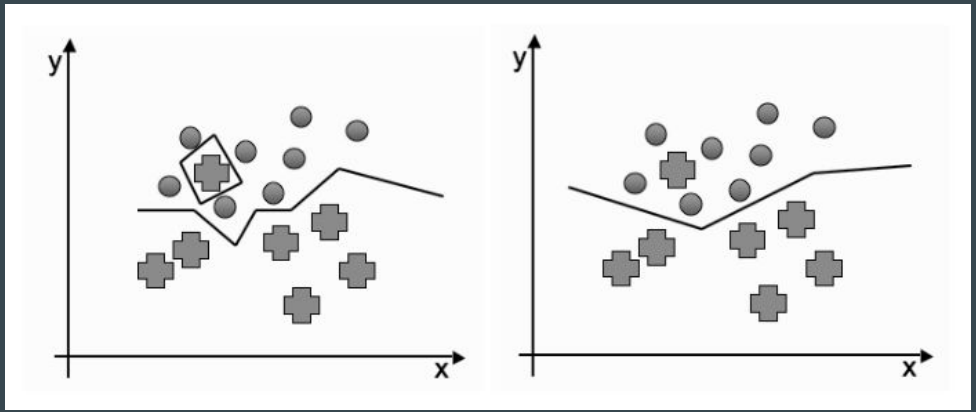
- This algorithm works on the principle of grouping up similar data points
- It is very similar to the K Means Clustering algorithm
- The main difference between KNN and K Means is that KNN is a supervised algorithm which takes into account the feature similarities between data points and assigns it a class based on previous data
- Whereas, K Means is an unsupervised algorithm that is used to group similar data points without any labels
- Other than that they work very similarly
- KNN classifies data using closeness to given examples whereas K Means clusters data without any predefined labels

KNN: The Algorithm

1. Determine the value for k
2. Calculate the distance between the new test data and all the existing training data points
3. Sort the distances in ascending order and get the k nearest neighbours
4. Find the majority class in the k nearest neighbours
5. Assign this class to the new test data

Value of k

- The value of k is not trivial
- If k is too small then the model is very sensitive to noise (variance)
- If k is too large then the accuracy reduces as as points that are too far away are considered (bias)



Left: k = 1 (Overfitting)

Right: k = 2 (Smooth)

KNN Bias Variance Tradeoff

$$E((\mathbf{y} - \hat{f}(\mathbf{x}))^2) = \sigma^2 + \left[f(\mathbf{x}) - \frac{1}{k} \sum_{i=0}^{k} f(\mathbf{x}_i) \right]^2 + \frac{\sigma^2}{k}.$$

The above equation can be simply interpreted as:

- If k is small (few neighbours) then flexibility increases and can represent the data better but it makes mistakes because of noise (High Variance)
- If k increases, the variance decreases but the model becomes less flexible and thus the bias increases

Using KNN for Regression

- The KNN algorithm can be modified slightly and can be used for regression applications
- This can be done by returning the average value of the neighbours to a point as the predicted value
- The most common methods are called kernel smoothers
- They use a kernel that decides how much weight to put on a certain point according to its distance from the input

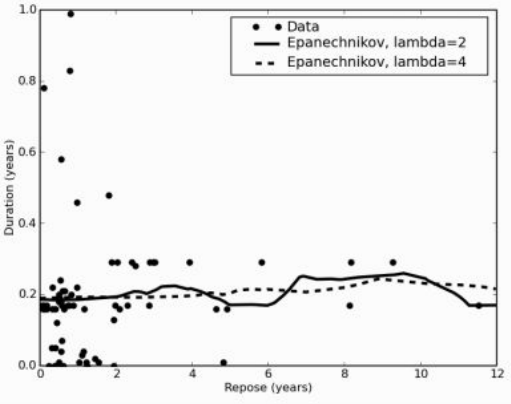
Regression Kernels

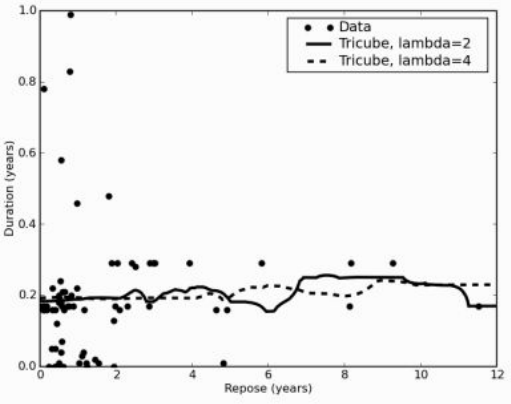
- Epanechnikov Quadratic Kernel:

$$K_{E,\lambda}(x_0, x) = \begin{cases} 0.75\left(1 - (x_0 - x)^2/\lambda^2\right) & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases}$$

- Tricube Kernel:

$$K_{T,\lambda}(x_0, x) = \begin{cases} \left(1 - \left|\frac{x_0 - x}{\lambda}\right|^3\right)^3 & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases}$$





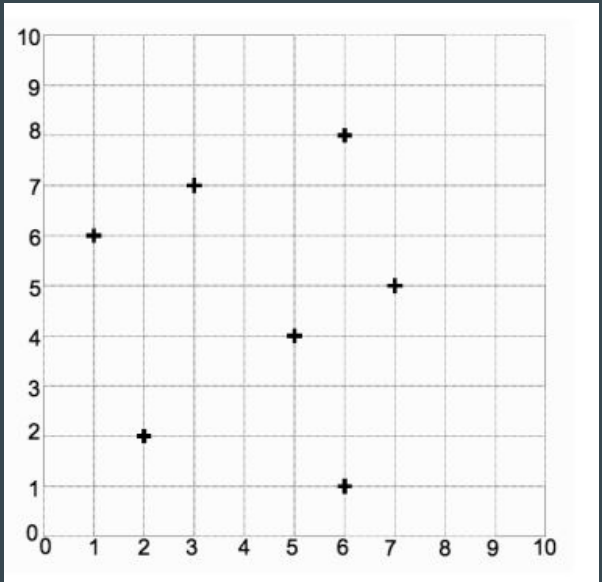
Efficient Distance Computations: The KD-Tree

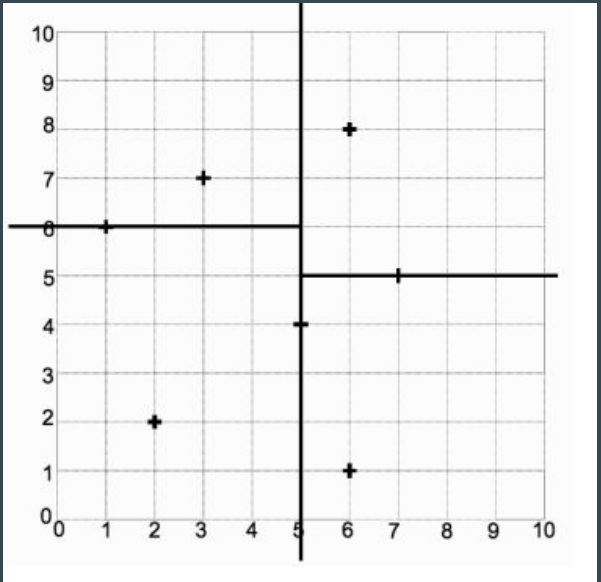
- As discussed before the distance computation is very time consuming
- To combat this we use a Data Structure called the KD Tree
- It reduces the time complexity to O(logN) and space complexity to O(N), the construction of the tree takes $O(Nlog^2N)$
- The tree is very similar to the decision tree where you create a binary tree and choose one dimension at a time to split into two
- Then you place a line through the median value of that coordinate
- The choice of the dimension is made randomly and it changes between horizontal and vertical splits

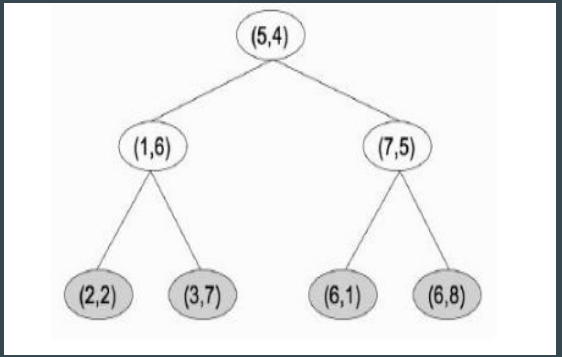
KD Tree Construction: The Algorithm

```
# Pick next axis to split on
whichAxis = np.mod(depth,np.shape(points)[1])

# Find the median point
indices = np.argsort(points[:,whichAxis])
points = points[indices,:]
median = np.ceil(float(np.shape(points)[0]-1)/2)

# Separate the remaining points
goLeft = points[:median,:]
goRight = points[median+1:,:]

# Make a new branching node and recurse
newNode = node()
newNode.point = points[median,:]
newNode.left = makeKDtree(goLeft,depth+1)
newNode.right = makeKDtree(goRight,depth+1)
return newNode
```

KD Tree: Example







Using the KD Tree

- Start at the root then go to each leaf node and check if this point belongs to that region
- Compute this distance and store it
- The advantage of the KD tree is that you only need to check 2 levels: Cousins and Parent's cousins of the current node
- Thus it reduces the amount of computing required by pruning some of the nodes

Distance Metrics

$$L_k(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{d} |x_i - y_i|^k \right)^{\frac{1}{k}}$$

- This metric is called the Minkowski Metric
- With k = 1 we get Manhattan Distance
- With k = 2 we get Euclidean Distance
- k can be changed according to the number of dimensions
- For some cases like images we need to use a metric which is invariant to scaling, rotational and other translational changes
- A common example for an invariant metric with images is the tangent distance