K Dimensional Tree

Why it is used

K Dimensional Trees or KD-Trees in short are used to avoid the brute force approach of Naïve K Nearest neighbors (KNN). Having to check every single in large and highly dimensional dataset would simply take too long. A data set of N training points and M testing points would have to check N x M (Too much time is wasted).

Fortunately, there are better approaches KD Trees function almost identically to Binary search trees in the ways there are traversed. Except here instead of a single value showing the splitting direction there are K values (dimensions) taking turn as splitting targets. In a K = 2 tree, each value coordinate of the point will become the target from one level of the tree to the next. For a tree *T* with 4 nodes R = (5, 4); Left_C = (4, 5); Right_C = (12, 6) and Right_C_RChild = (2, 8). Where R is the *root node*. Inserting a point insert = (7, 7) would find itself on the right sub-tree of Right_C_RChild. Since X of root is less than X insert, we will turn right. One the second level second dimension (Y) now becomes the target and since since Y of insert is greater than Y of Right_C we will keep going right. Now being on the third level and having explored all dimension we reset the target to the initial dimension I.e. (X) and with X of insert being greater than X of Right_C_RChild we will again go right. A successful insertion of the of the node has occurred, the process will be repeated N times until remains.

The functions in ***myMN_NB*** class:

Training Phase

Fitting the model

The ***fit*** function takes in two parameters a training table of data and its associated training labels. It computes mean and standard deviation of all testing point. These cell arrays and then used to perform Z-score standardization on the data which will be of use later. Z-score standardization is a crucial step as in some feature might have higher values than other (perhaps they were recorded using different units). This constitutes a problem as some important features maybe drowned out as a result, fortunately Z-Score standardization takes care of such issues.

It also builds a tree using the *BuildTree* function and parsing the model a root node containing all the standardized unsliced data and other features such as current dimension index etc.

Building a tree

The ***BuildTree*** function allows us to like the name says build an actual tree with all of our data. However, there is an issue. Since each node will contain multiple points, how do we know which one to take and make our split?

Simple we find the median value of the dataset. The median is the middle value of a set of number. But is that really all that is involved? The median must change as we descent the tree but according to what criteria?

As mentioned in the introduction as the depth change, the splitting dimension (target row/feature) also changes. The median will hence be calculated at each level for a different feature until the Nth feature, only then with the dimension with reset to its starting point. Everything less than the median will go to the left and everything equal or greater to it, to the right. Being a recursive function, some conditions are required for unwinding. In our case the stopping condition is when no more data can be split (all data become either less or greater than median).

This operation will be performed as many times as necessary until no more splits are available and a root with links to all descendants will be returned.

Classification Phase

Predicting a label

Predict

The ***predict*** function predicts the labels for every testing data point. On each iteration the data point is standardized using Z-score standardization. The **predict_one** function is then called for the current data point and receives the standardized data point and the model.

Predict_one

The **predict_one** function, predicts the label for a single test point. To do so the ***descend_tree*** method is called. This is where all the important computation is done. It takes a node, the test example, the best distance (initially 1), the model, a struct array of node neighbors (initially empty) and the number of visited nodes so far on unwinding the recursion (initially 0).

Once a list of best K possible nodes has been returned ordered based on best distance. The top node (first) node in the list is picked as it will have the shortest distance.

The winning node's class cell is then call and indexed with the index of the best distance point in that cell. And a prediction is made.

Descending the tree

The ***descend_tree*** function descends the tree alternating dimensions at each depth level. Going left and right depending on the result of the comparison between the median value of the node and the value at the same dimension as that of which produced the median value of said node. Once a leaf is reached, we can now compute the Euclidian distance from the current test

point to all points in the leaf node. The leaf will then be parsed to the ***memorize_leaves*** function where it is added to a list of other leaves which is sorted based on based distance. A ***possible*** match has been found for out test point. However, it is not enough.

Each node can be seen as a box, having its own domain. The test point could at the edge of the domain in leaf X, and have a closer point to it (Living on the cost and far from the town center, means we are closer to the ocean than we are from the town center; even though we are living in city X or Y). If nothing is done about such cases some accuracy in prediction may be lost.

Finding a better match

On unwinding the recursion, the ***find_better_match*** function is called. As the name implies it will find a better match based on a few conditions. To find a better match, the absolute value of the distance between median of the node currently unwinding and the testing point at this node's dimensional plane must be less than the best distance recorded so far. This is computed by the ***find_Best_distance*** function which returns a string flag, If the flag says ***"visit"*** we investigate the opposite of the last visited node's parent (*currently unwinding node*) if and only if said node has more than one child. If yes, the opposite child of the current unwinding node is parsed to ***find_better_match*** until new a leaf is reach. It is important to note that some implementations only visit at most 2H nodes, where H is the height of the tree.

Memorizing leaves

The ***memorize_leaves*** function returns an array of node structures sorted based on the best distance of each nodes.

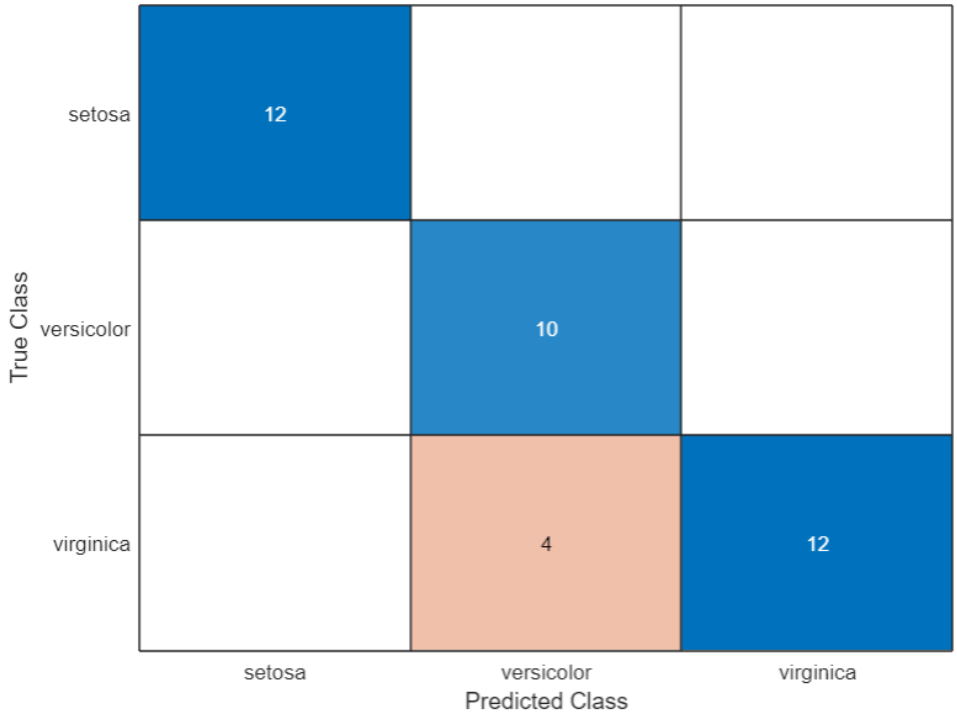
find_Best_distance

This function returns a string flag either "visit" or "skip", depending on whether the distance of the test point to the current unwinding node is less or greater than the best distance.

Final comparison between KNN and KDTree on identical data set of 38 flowers

```
classifying example example 38/38

confusion_mat2 =

    11     0     1
     0    10     0
     0     6    10


order =

  3×1 categorical array

     setosa
     versicolor
     virginica
```

Fig1: above KD-Tree had a classification accuracy of 81.58%



KNN had with brute force had an accuracy of 89.47%