Executive Summary Report

Jacob McIntosh

ANLY 555

5/2/22

The Dataset superclass (***Figure 1***) allows for an individual to read in a CSV file and offers basic dataset operations such as viewing the head and tail of the dataset as well as a generic cleaning method that cleans based on the type of data provided. This data type is provided by a private attribute in the instantiation of some of the subclasses. This allows for the full inheritance of the clean() method for both the QuantDataset and QualDataset subclasses. The TextDataset and TimeSeriesDataset subclasses have overloaded clean() methods that clean the data appropriately according to their respective data type. The explore methods in each of the subclasses provide datatype-specific visualizations of the data loaded into the object.

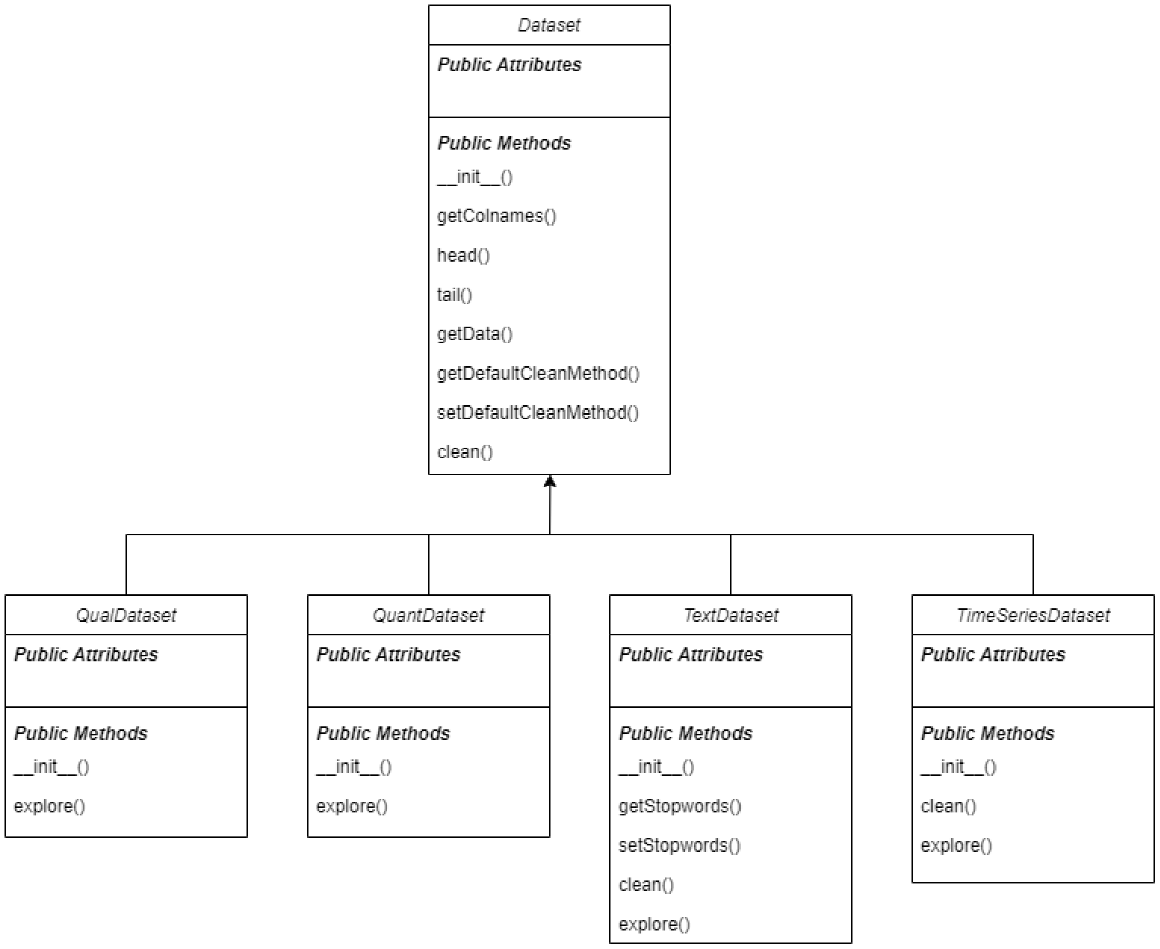

Figure 1: *The class hierarchy diagram for the Dataset class.*

The ClassifierAlgorithm superclass (***Figure 2***) allows for a polymorphic structure of a supervised classification learning algorithm, which for the train() method, simply stores the input data and labels as private attributes. Three subclasses, simpleKNNClassifier, DecisionTreeClassifier, and kdTreeKNNClassifier are provided. The simpleKNNClassifier implements a full inheritance of the base class train() method as all computational steps for this algorithm are performed during the test() method. Each subclass overrides the existing test() method to accommodate for that subclasses individual requirements. Note the use of both inheritance relationships and nested classes. The DecisionTreeClassifier and kdTreeKNNClassifier, which are tree-based methods, inherit attributes and methods from the ClassifierAlgorithm and Tree superclasses. The nested classes allow for an algorithm, specifically the DecisionTreeClassifier and kdTreeKNNClassifier to create multiple instances of a nested class object within itself, without having to instantiate more than one base class object.

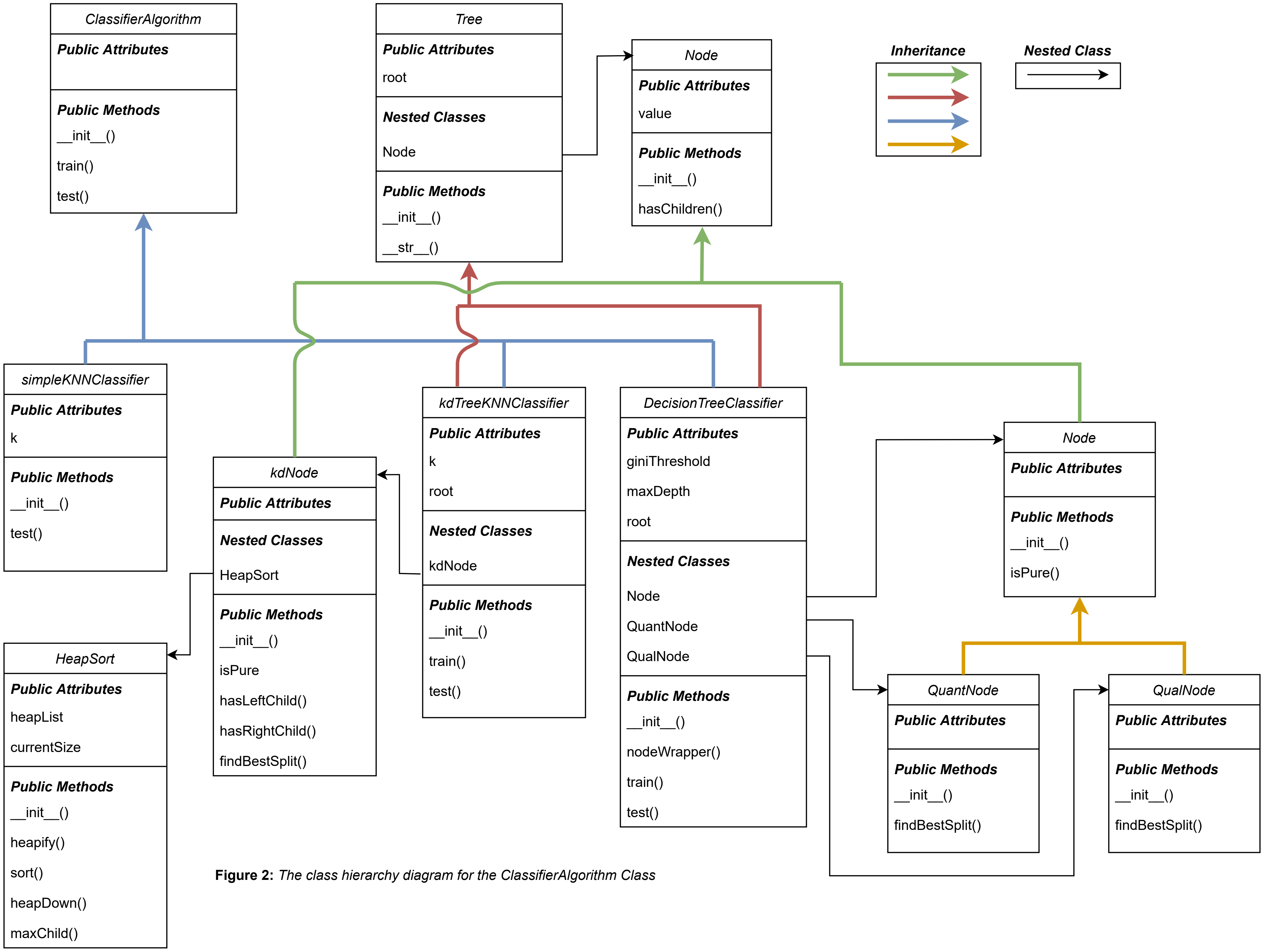

Figure 2: *The class hierarchy diagram for the ClassifierAlgorithm Class*

The Experiment class (***Figure 3***) allows for speedy evaluation of a collection of pre-initialized ClassifierAlgorithm subclasses. The constructor of this class accepts a Dataset subclass object and a list of pre-initialized ClassifierAlgorithm subclasses. The runCrossVal() method will iteratively perform k-fold cross-validation on each of the ClassifierAlgorithm subclasses and save a choice of the class predictions or class probabilities in a private attribute. The score() method displays an aligned table of accuracies for each of the ClassifierAlgorithm subclasses provided for the Experiment Class. The confusionMatrix() method displays a confusion matrix for each of the provided ClassifierAlgorithm subclasses. The ROC() method will plot the ROC curve of each of the provided ClassifierAlgorithm subclasses overlaid in one plot. Again, note the use of the nested class structure with the HeapSort class, which is used to sort an array. This nested class is used in the ROC() method to sort the class probabilities computed by runCrossVal().

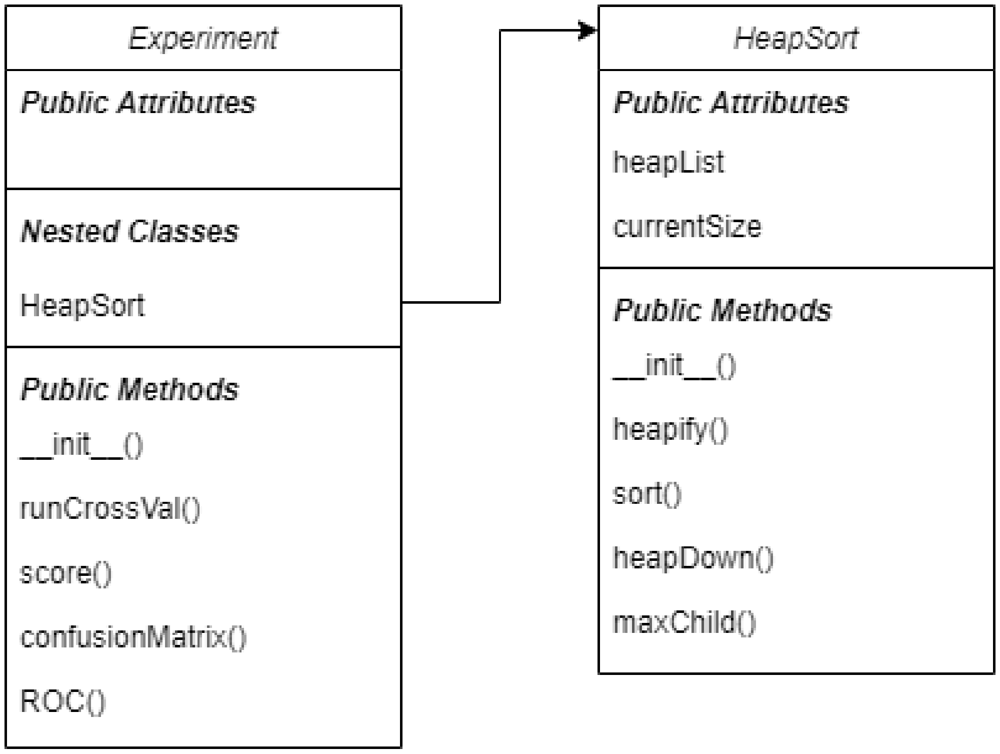

Figure 3: *The class hierarchy diagram for the Experiment Class..*

The simpleKNNClassifier subclass implements a brute force search for the nearest points to an arbitrary test point. Because this algorithm is a brute force search, there is no training algorithm. ***Algorithm 1*** depicts the underlying algorithms to predict a test point's class label. The algorithm iteratively searches through the training data, computing each distance to the test point, and keeping track of the k closest distances. After iteration, the algorithm returns the mode of the class labels of the points that were deemed to be the closest to the test point. This algorithm has a time complexity *O(kn)*. This is because, for each nearest neighbor, it is necessary to iterate through the data and compute the distance to each test point which results in *O(n).* At each distance computed, if the distance is less than the maximum of the currently tracked distances, iterate through the list of tracked distances, find the maximum distance, and replace it with the newly computed distance. Because the list of tracked distances has length k, this results in *O(k)* within the iteration. Therefore, the overall time complexity of this algorithm is *O(kn)*.

Algorithm 1 K Nearest Neighbor Classification using Brute Force Search

```
  Input:    $\mathbb{X}$, training data; k, number of neighbors; p, a test point
  Output:   $\hat{y}$, class prediction of p
$N \leftarrow [\,]$
$i = 0$
while $i < k$ do
    append $\infty$ to $N$
    $i \leftarrow i + 1$
end while
for each $x \in \mathbb{X}$ do
    $d \leftarrow$ distance between p and x
    if $d < max(N)$ then
        pop $max(N)$ from $N$
        append $d$ to $N$
    end if
end for
$\hat{y} = mode$(class labels of $N$)
return $\hat{y}$
```

The decisionTreeClassifier subclass implements a decision tree on the data. ***Algorithm 2*** depicts the logical process for building the tree. The algorithm begins with the root node. If the node is not pure, implying that the points within the node are do not share the same class label, the algorithm will compute a split from a logical operation that divides the points in such a way that each partition of the node's point will be purer than the current node. This partition is used to create two child nodes, where one child contains only points where the logical operation is true, and one child contains only points where the logical operation is false. This process is repeated recursively until all leaf nodes of the decision tree are pure. This algorithm has a time complexity *O(mn^2)*, where m is the number of features in the data. In the worst-case scenario, the decision tree algorithm could recursively build the tree to a depth of n, which results in *O(n).* At each node that isn't pure, the algorithm must compute the best possible split given the data contained within that node. For this, the algorithm must iterate through the up to n unique points, each with m different features, defined on the node, testing each point as a possible split for the node. This results in *O(mn)* for each instance of the iteration*.* Therefore, the overall time complexity of the tree-building algorithm is *O(mn^2)*. However, this is not the best or average case. Generally, the decision tree building process will not amount to building a tree of depth n. Rather, in the best case, it would build a decision tree of depth 1, as the data is completely linearly separable on one single split. This would reduce the time complexity to *O(mn).* It should be expected that the average decision tree falls between these two extremes.

```
Algorithm 2 Decision Tree Building
    Input:    $\mathbb{X}$, training data
    Output:   treeroot, root of decision tree
  $treeroot \leftarrow makeNode(\mathbb{X})$
  BUILDTREE($treeroot$)
  procedure BUILDTREE($node$)
      if $node$ is pure then
          exit
      else
          $node.split = split(node)$
          $leftpoints = [\,]$
          $rightpoints = [\,]$
          for each $element \in node.elements$ do
              if $element \leq node.split$ then
                  append element to leftpoints
              else
                  append element to rightpoints
              end if
          end for
          $node.split = split(node)$
          $node.leftchild = makeNode(leftpoints)$
          $node.rightchild = makeNode(rightpoints)$
          BUILDTREE($node.leftchild$)
          BUILDTREE($node.rightchild$)
      end if
  end procedure
  return $treeroot$
```

The testing algorithm for the decisionTreeClassifier subclass will predict the class of a test point using the built tree. ***Algorithm 3*** shows the logical process for computing this prediction. The algorithm begins with the root node and evaluates the logical expression defined on that node. The algorithm then traverses to one of the children depending on the result of the logical expression. This process is recursively executed until the traversed node is pure. The class label defined by that traversed node is returned as the prediction. The time complexity for this algorithm is *O(n)*. This comes from the worst-case scenario that the tree has depth n, and the algorithm must traverse to the deepest node to retrieve the prediction. This is generally not the case. In an average case, it would be expected that the tree will have a depth less than log$_2$n, resulting in an average or better case time complexity *O(log$_2$n).*

```
Algorithm 3 Decision Tree Classification
    Input:    treeroot, root of decision tree; p, a test point
    Output:   $\hat{y}$, class prediction of p
  $node = $ TRAVERSETREE($treeroot$)
  procedure TRAVERSETREE($node$)
      if $node$ is pure then
          return $node$
      else
          if $p \leq node.split$ then
              TRAVERSETREE($node.leftchild$)
          else
              TRAVERSETREE($node.rightchild$)
          end if
      end if
  end procedure
  $\hat{y} = mode$(class labels of $node.elements$)
  return $\hat{y}$
```

The kdTreeClassifer subclass implements a kdTree to perform an optimized K Nearest Neighbor algorithm. ***Algorithm 4*** depicts the logical process for building the kdTree. First, a root node is created containing all of the data. An axis of the data is selected based on the depth of the current node. The data is sorted on that axis, and the point containing the median value is extracted. The node holds this point and passes the rest of the points to two child nodes. One child node will be given all the points that are less than or equal to the median point on the axis, and the other child will be given all the points greater than the median point on that axis. This process is repeated recursively until each node holds only one point. The time complexity for this algorithm is *O(n^2log_2n).* The algorithm will ultimately have to build n nodes for this tree which results in *O(n).* At each instance of the recursion, the data must be sorted by the axis given by the depth of the tree, resulting in maximum time complexity of *O($nlog_2n$)* for each recursion. Therefore, this algorithm has a time complexity *O(n^2log_2n).*

Algorithm 4 kdTree Building for KNN

```
  Input:    $\mathbb{X}$, training data
  Output:   treeroot, root of kdTree
$dim \leftarrow dimension(\mathbb{X})$
$treeroot \leftarrow makeNode(\mathbb{X})$
BUILDTREE($treeroot, 0$)
procedure BUILDTREE($node, depth$)
    if size($node.elements$) = 1 then
        $node.point = node.elements$
    else
        $axis \leftarrow depth\ modulo\ dim$
        Sort $node.elements$ on axis
        $node.point \leftarrow median(node.elements)$
        $leftpoints = [\,]$
        $rightpoints = [\,]$
        for each $element \in \{node.elements\} - \{node.point\}$ do
            if $element \leq node.point$ then
                $append\ element\ to\ leftpoints$
            else
                $append\ element\ to\ rightpoints$
            end if
        end for
        if size($leftpoints$) > 0 then
            $node.leftchild = makeNode(leftpoints)$
            BUILDTREE($node.leftchild$)
        end if
        if size($rightpoints$) > 0 then
            $node.rightchild = makeNode(rightpoints)$
            BUILDTREE($node.rightchild$)
        end if
    end if
end procedure
return $treeroot$
```

The testing algorithm for the kdTreeClassifer subclass will predict the class of a test point using the built kdTree. ***Algorithm 5*** depicts the logical process behind this algorithm. This algorithm keeps track of the k nearest points to the test point. Beginning with the root node, the algorithm computes the distance from the test point to the median point defined on the node. If the computed distance is less than the largest distance of the k nearest points, then the point and largest distance are replaced with the newly computed distance and point. The algorithm then traverses to one of the child nodes and decides whether the test point is less than or greater than the node's median point on the axis defined by the depth of the node. This algorithm is repeated recursively until the current node has no child nodes. At this point, the algorithm starts to exit the recursion. However, if the distance from the test point to the node's median point on the axis defined by the depth of the node, is less than the maximum of the k nearest points, then the algorithm must traverse through the other child node not previously traversed through. Once the algorithm exits the recursive procedure, the algorithm returns the mode of the class labels of the k points that were deemed to be the closest to the test point. This algorithm has time complexity *O(kn).* In a worst-case scenario, the algorithm is unable to prune any of the branches of the tree and must traverse the entire kdTree to calculate the k nearest points. This results in *O(n)*.

At each distance computed, if the distance is less than the maximum of the currently tracked distances, iterate through the list of tracked distances, find the maximum distance, and replace it with the newly computed distance. Because the list of tracked distances has length k, this results in *O(k)* for within each recursion. Therefore, this algorithm has a time complexity *O(kn).* However, this is the worst-case scenario. The data is rarely distributed in such a way that the algorithm cannot prune branches of the tree. In an average case, the algorithm will only have to visit a number of nodes proportional to $\log_2 n$. Therefore, the average time complexity of this algorithm is *O(k*$\log_2 n$*).*

Algorithm 5 K Nearest Neighbors Classification using kdTree

```
  Input:   treeroot, root of kdTree; k, number of neighbors; p, a test point
  Output:  $\hat{y}$, class prediction of p
$dim \leftarrow dimension(p)$
$N \leftarrow [\,]$
$i = 0$
while $i < k$ do
    $append\ \infty$ to $N$
    $i \leftarrow i + 1$
end while
$depth = 0$
TRAVERSETREE($treeroot, depth$)
procedure TRAVERSETREE($node$, $depth$)
    if $node$ is $NULL$ then
        exit
    end if
    $axis \leftarrow depth\ modulo\ dim$
    $d \leftarrow$distance between $p$ and $node.point$ at axis
    if $d < max(N)$ then
        $pop\ max(N)$ from $N$
        $append\ d$ to $N$
    end if
    if $p \leq node.point$ then
        $bestchild = node.leftchild$
    else
        $bestchild = node.rightchild$
    end if
    TRAVERSETREE($bestchild$, $depth + 1$)
    if $|p - node.point| < max(N)$ on axis then
        TRAVERSETREE($otherchild$, $depth + 1$)
    end if
end procedure
return $treeroot$
```