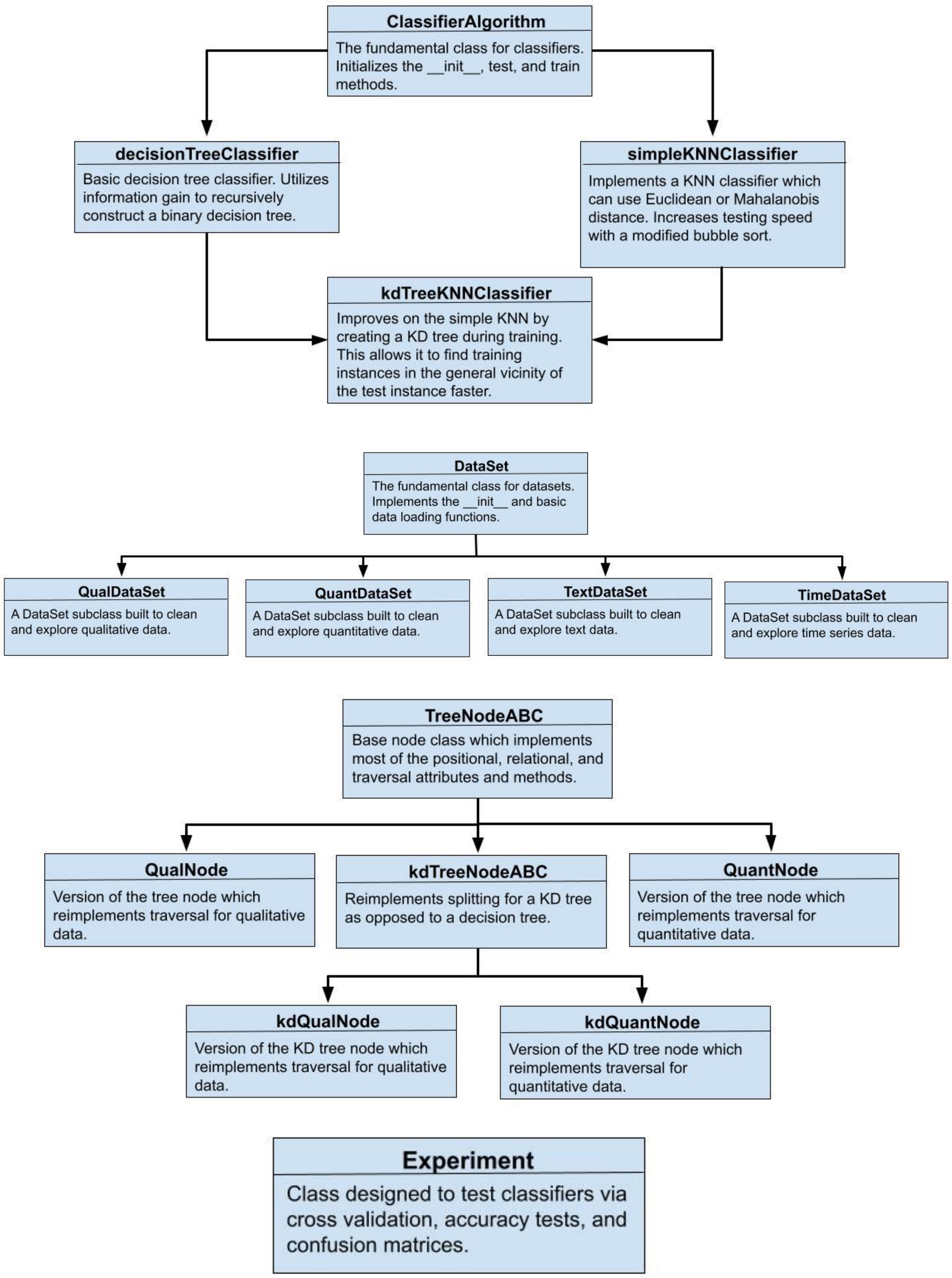
I. Class Hierarchy Diagram


ClassifierAlgorithm
The fundamental class for classifiers. Initializes the __init__, test, and train methods.
decisionTreeClassifier
Basic decision tree classifier. Utilizes information gain to recursively construct a binary decision tree.
simpleKNNClassifier
Implements a KNN classifier which can use Euclidean or Mahalanobis distance. Increases testing speed with a modified bubble sort.
kdTreeKNNClassifier
Improves on the simple KNN by creating a KD tree during training. This allows it to find training instances in the general vicinity of the test instance faster.
DataSet
The fundamental class for datasets. Implements the __init__ and basic data loading functions.
QualDataSet
A DataSet subclass built to clean and explore qualitative data.
QuantDataSet
A DataSet subclass built to clean and explore quantitative data.
TextDataSet
A DataSet subclass built to clean and explore text data.
TimeDataSet
A DataSet subclass built to clean and explore time series data.
TreeNodeABC
Base node class which implements most of the positional, relational, and traversal attributes and methods.
QualNode
Version of the tree node which reimplements traversal for qualitative data.
kdTreeNodeABC
Reimplements splitting for a KD tree as opposed to a decision tree.
QuantNode
Version of the tree node which reimplements traversal for quantitative data.
kdQualNode
Version of the KD tree node which reimplements traversal for qualitative data.
kdQuantNode
Version of the KD tree node which reimplements traversal for quantitative data.
Experiment
Class designed to test classifiers via cross validation, accuracy tests, and confusion matrices.

II. Decision Tree Classifier

The decision tree classifier uses a decision tree structure to predict test data labels. To do this, it inherits form both the base classifier class and a new tree node class. When training, nodes are created recursively, where a best split point is found, and data is subdivided to either side of that split as new nodes. This continues until either the node is pure, or another split would not provide sufficient information gain. Testing then runs each instance, again a recursive process, through each splitting condition and assigns a label based on the leaf node in which the test instance ends up. This algorithm can also be printed and pasted here to produce the training tree.

Below, the pseudo code and time complexity analyses for important functions are included. Note that the time complexity for important lines is denoted by **[]**. Constants are not included here.

Train Method Pseudo Code & Time Complexity:

```
1. Save data and a list of labels as member attributes
2. Initialize a blank root node using the maximum entropy and most common class [n]
3. Determine the best split for this root node [cn]
4. Recreate the root node as either a quantitative or qualitative node [n]
5. Pass to the recursive training function [cn+n]
    a. Check if the node is pure
    b. If not, find the best split
    c. Check if the best split produces enough information gain
    d. If yes, split and repeat steps a-d for each child node
```

O(n) = cn, as this is the time taken to determine the best split. Here, c is the number of columns and n is the number of training instances. This matches with the paper provided in class on decision trees.

Test Method Pseudo Code & Time Complexity:

```
1. Reset the indices of the test data [n]
2. Initialize a list for predicted labels [n]
3. Cycle through each test instance [n]
    a. Recurse through the training tree to find the predicted label [log(n)]
        i. Check if a leaf node
        ii. If so, return the prediction
        iii. If not, check whether to go left or right, then go that way
4. Return the predicted labels
```

O(n) = nlog(n), as the time it takes to traverse down a tree is ~log(n), and we must do this n times. Here, n represents the number of instances in our training dataset.

III. Simple KNN Classifier

The basic KNN classifier uses proximity to training instances to predict test labels. Training for KNN involves simply storing the data. Testing involves comparing each testing instance to each training instance, then finding the k closest neighbors per testing instance. Distance here is calculated using either Euclidean or Mahalanobis distances. The *k* nearest neighbors are found by sorting a list of distances for a given test set using bubble sort, as this can be stopped once the *k* smallest values have been found.

Below, the pseudo code and time complexity analyses for important functions are included. Note that the time complexity for important lines is denoted by **[]**. Constants are not included here.

Test Method Pseudo Code & Time Complexity:

```
1. Reset the indices for the test data [pn]
2. Check whether to use Euclidean or Mahalanobis distances
3. Calculate the distances for both the training and testing data [cn]
4. Create a blank list for predicted labels the size of the test data [pn]
5. Cycle through each test instance [pn]
    a. Create a blank list for distances the size of the training data [n]
    b. Cycle through each training instance [n]
        i. Calculate and save the distance between the current test instance and each training
           instance
    c. Use bubble sort to find the k training instances nearest the test instance [kn]
    d. Find the most common label of these k instances and save it in the list of predicted labels [k]
6. Return the predicted labels
```

$O(n) = n^2$, as we must compare each test instance to each training instance. This would take much longer with a slower sorting algorithm, but luckily bubble sort can store early once *k* neighbors are found. If *k* approaches *n* though, then this algorithm will take a terrible $O(n) = n^3$ to train. Here, *p* represents the percent of points in the test data.

IV. KD-KNN Classifier

Building on the KNN classifier and a modified KD tree node, this classifier has a much faster testing method for larger datasets with low dimensions. To achieve this improved speed, a training method is implemented where each training instance is sorted into a node of KD tree. At each node of the tree, the median value for a particular feature is selected, and all others are split below (left) or above (right) to a new set of nodes down the tree. This is continued until there is only one instance for each node. Finally, during testing instances are passed through the KD tree, much as in a decision tree. All nodes along the path the test instance travels AND one node away from this path are assessed for matches. This helps us locate points in the general area of our test instance much faster than by searching the entire training set.

Below, the pseudo code and time complexity analyses for important functions are included. Note that the time complexity for important lines is denoted by **[]**. Constants are not included here.

Train Method Pseudo Code & Time Complexity:

```
1. Save data and a list of labels as member attributes
2. Find the first dimension to split on and this split's type
3. Find the median instance along this dimension [nlog(n)]
4. Create either a qualitative or quantitative root node based on this split type
5. Save the index, label, split dimension, and split condition as attributes of the root node
6. Conduct the same general process on each child node recursively [$d2^d$nlog(n)]
    a. Stop when there is only one instance left to split
```

O(n) =nlog(n), assuming that n >> d where *d* is the dimensionality of the dataset. Line 6 has $d2^d$ as a coefficient as there will be 2^d nodes per level and some number related to *d* levels. The nlog(n) in the line 6 is due to finding the median instance at each node. As was noted in the accompanying paper, this method struggles as the number of dimensions increases.

Test Method Pseudo Code & Time Complexity:

```
1. Reset the indices for the test data [pn]
2. Initialize an empty list for predicted labels the size of the test data [pn]
3. Check whether to use Euclidean or Mahalanobis distances
4. Calculate the distances for both the training and testing data [n]
5. Cycle through each test instance [pn]
    a. Call the recursive test function to find training instances near the test instance [2log(n)]
    b. Initialize an empty list the size of the test function's path for distances and labels [log(n)]
    c. Cycle through each path instance [log(n)]
        i. Calculate and save the distance between the test and train point
        ii. Save this training point's label
    d. Use bubble sort to find the k closest training points [klog(n)]
    e. Find the most common label of these k closest points [log(n)]
6. Return the predicted labels
```

O(n) = nlog(n), as we must compare each test instance to the training instances near the test instances path traversing the KD tree. Here, *p* represents the percent of points in the test data.