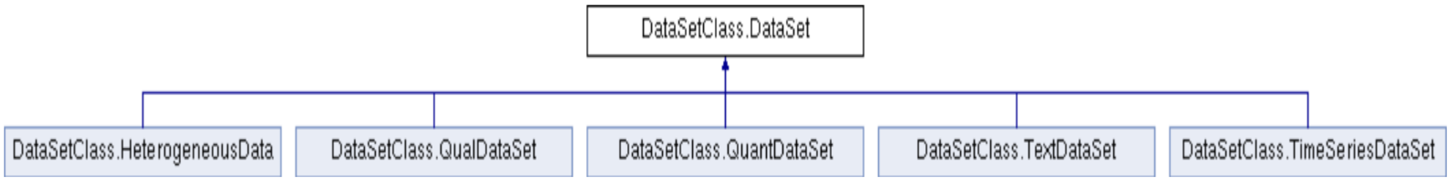
Jake Steinfink
ANLY-555

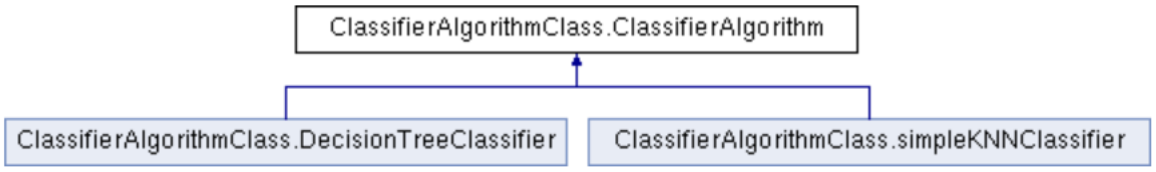
DataScience ToolBox Design

DataSet


DataSetClass.DataSet
DataSetClass.HeterogeneousData
DataSetClass.QualDataSet
DataSetClass.QuantDataSet
DataSetClass.TextDataSet
DataSetClass.TimeSeriesDataSet

The DataSet class is meant to serve as a means of handling all different types of datasets. Irrespective of the type of dataset, this class provides the functionality to load the data into a usable dataframe, clean the data, with appropriate options to fill missing values with the mean, median, or mode, and explore the data through different types of charts suited to each dataset type. The different dataset types with functionality are TimeSeries, Text, Quantitative, and Qualitative. Additionally, this class also provides the ability to perform these operations on multiple different datasets and dataset types at once through the HeterogeneousData subclass.

ClassifierAlgorithm


ClassifierAlgorithmClass.ClassifierAlgorithm
ClassifierAlgorithmClass.DecisionTreeClassifier
ClassifierAlgorithmClass.simpleKNNClassifier

The ClassifierAlgorithm Class provides functionality for a simpleKNN and DecisionTree classifier. These algorithms are meant to be used to predict labels in a test dataset based on a labeled train dataset. The simpleKNN classifier runs in BigO(n) time to make predictions for each test point, while the DecisionTree classifier can make predictions in constant time.

Experiment

The Experiment class is the most results oriented class in the toolkit. It is the class that is used to analyze the effectiveness of algorithms. Within the class there is functionality to run cross validation for each algorithm, score each algorithm with an overall accuracy result, display a confusion matrix, and create a ROC curve to show the tradeoff between false positive rates and true positive rates (the cost of tuning the algorithm to identify all positives).

SimpleKNNClassifier

Summary

The SimpleKNNClassifier algorithm is quite simple. It relies on the idea that rows near each other share similar properties, and in this specific case, share similar labels. As a result, the algorithm's goal is to find the K rows (K is a user defined input) nearest to the test row of interest. Once these are isolated, the labels of these rows are examined and used to make a prediction; the most common label occurring in the k nearest neighbors is used as the predicted label for the test row.

Pseudo Code

Train Method

Inputs- train data and labels

```
1. TrainData, TrainLabels assigned to member attributes
2. Assign unique classes to member attribute
```

Test Method

Inputs – test data and associated labels

Outputs – List of predicted labels, stored as dictionaries

```
1. Predicted_Labels ← [ ]
2. For each row in the test data:
    a. Row ← row
    b. Neighbors ← [ ]
    c. For each row in the train set
        i. Append tuple of the form (train [row] , euclidean distance )
    d. Neighbors ← [Neighbors sorted by euclidean distance, ascending ] [:3]
                    ##Take first k entries
    e. Prediction ← {}
    f. For each  unique neighbor in Neighbors
        i. Prop ← Instances of unique neighbor in Neighbors/Number of neighbors
        ii. Prediction[ unique neighbor ] ← Prop
        iii. Append Prediction to Predicted_Labels
```

Time Complexity for Test Method:

BigO(nlogn) c=m * z (size of the test data), n_0=20

With respect to time complexity, the SimpleKNNClassifier is BigO(nlogn) meaning that the time it takes to run is dependent on the size of the input in a quasi – linear fashion, but it really is more exponential because the coefficient for this particular method is m*z, the number of columns in the dataset * the size of the test data. The time complexity is due to the fact that for every test row, the algorithm makes use of every training row, with all of its columns, to calculate the euclidean distance. All other elements of the algorithm are simple sorting procedures which are completed in nlogn time.

<u>**DecisionTreeClassifier**</u>

<u>*Summary*</u>

The DecisionTreeClassifier creates a decision tree in the train method and predicts test rows in the test method for quantitative data. The structure of the decision tree is such that the train indices at each node are held as the key, the payload is the train labels associated with the indices. Each node also holds an attribute called decision which holds a dictionary containing either the split criteria at the node, or if a terminal node, a dictionary holding the classes present at that node and their associated proportions. Splits are determined by iterating over each column and each value of that column to find the split that produces the lowest gini impurity index score. The split that satisfies this is then stored at the decision attribute of the node and the process is then repeated recursively for both the left and right child nodes until the tree reaches max_depth or min_size limitations.

<u>*Pseudo Code:*</u>

Train Method

Inputs- train data and train labels

Outputs – Decision Tree

```
1. Assign test data and test labels to member attributes
2. Assign unique classes in the data to member attribute
3. Initialize Tree using a Tree class (Supports a tree structure and node attributes) with
   indices as the key and labels as the payload
4. Assign root node
5. Build a tree with user defined max_depth and min_size arguments
     a. Get the best split for the root node
     b. Call a recursive split function that checks to make sure max_depth and min_size
        parameters are not violated.
          i.   Store the split in the decision attribute of the node
          ii.  If not violated, initialize a left child with the indices that are less then the
               split value and then split that node and repeat.
          iii. Do the same for the right child node
     c. When depth and size parameters are violated, replace the decision attribute with a
        dictionary containing the labels and proportions represented in that node
```

Test Method
Inputs – test data and associated labels
Outputs – List of predicted labels, stored as dictionaries

```
1. Predicted_Labels=[]
2. For each row in the test data:
    a. Navigate the tree starting at the root and check if the value in the row at the
       column stored in the node (in the decision attribute) is less than the split value or
       not
    b. If less and the left child node is not a terminal node, recurse
        i. If it is a terminal node, return the dictionary holding class proportions
    c. Repeat step b for right child node if the value is greater
    d. Append terminal node result to Predicted_Labels
```

<u>*Time Complexity for Train Method:*</u>
Big O(n^2) c=m n0=200

With respect to time complexity, the DecisionTreeClassifier is BigO(n^2). It is also dependent on m, the number of columns in the data. The time complexity is n^2 due to the fact that to build the tree, all of the rows and columns in the node must be examined to find the best split. For the root node this is n*m and for all other nodes it sums to n*m. This is because at each level the data points in each node are ½ as many as in the parent node, however because the splitting has to be performed for all nodes, each level ends up iterating through all of the data points again. As a result, building the tree is relatively time consuming, but the payoff is found in the test method.

<u>*Time Complexity for Test Method*</u>*:*
Big O(n, n=test set size): c=8 n_0=3

The time complexity of the test method is linear with respect to the size of the test input. For each test data point, a constant number of operations is performed and the result returned. Thus, predicting a test label using the tree is exceedingly efficient.

<u>**ROC**</u>
<u>*Summary*</u>
The ROC curve algorithm creates a ROC curve for each classifier displaying the performance at all classification thresholds. If it is a 2 class problem, one curve is plotted for each classifier, if a

multi-class problem, then one curve for each label is plotted treating all other classes as negative points. The algorithm works by taking two lists, the actual labels and predicted labels. The lists are sorted in a decreasing order based on the likelihood of the instance being positive. Then, for every distinct likelihood, the number of positive and negative instances are calculated creating a true positive rate and false positive rate pairing for every distinct cutoff. The resulting curve travels from (0,0) to (1,1) with a path dependent on the accuracy of the classifier. The greater the area under the ROC curve (AUC), the better the classifier. A random classifier would create a line of the form y = x.

<u>Pseudo Code</u>

Inputs – List of Likelihoods (probability that the test row belongs to the positive class) and true labels

Outputs – ROC Points in the form of (FPR,TPR)

```
1. Predictions ← List of likelihood scores in decreasing order
2. Labels ← Sorted by same index as Predictions
3. R ← [ ] – will store tuples of ROC points (TPR,FPR)
4. FP,TP ← 0
5. P ← Number of instances of the positive class in the true labels
6. N ← Number of negative instances in the true labels
7. Previous ← any number less than 0
8. For each instance in the Predictions List:
    a. If Predictions[ i ] /= Previous or i == len(Labels) -1
        i. FPR ← FP/N
        ii. TPR ← TP/P
        iii. Append (FPR,TPR) to R
        iv. Previous ← Predictions[ i ]
    b. If Labels[ i ] is positive
        i. TP += 1
    c. Else
        i. FP +=1
```

<u>Time Complexity for ROC Method:</u>

BigO(nlogn)

With respect to time complexity, the ROC method is BigO(n) where n is the number of test labels. Determining the ROC points for every cutoff is done in a constant number of steps, most of which result from getting the information organized into the form necessary to run this algorithm. Another element to consider is the sorting method used to sort the likelihoods in descending order. Many, such as merge sort and quick sort will complete this procedure in logn time.