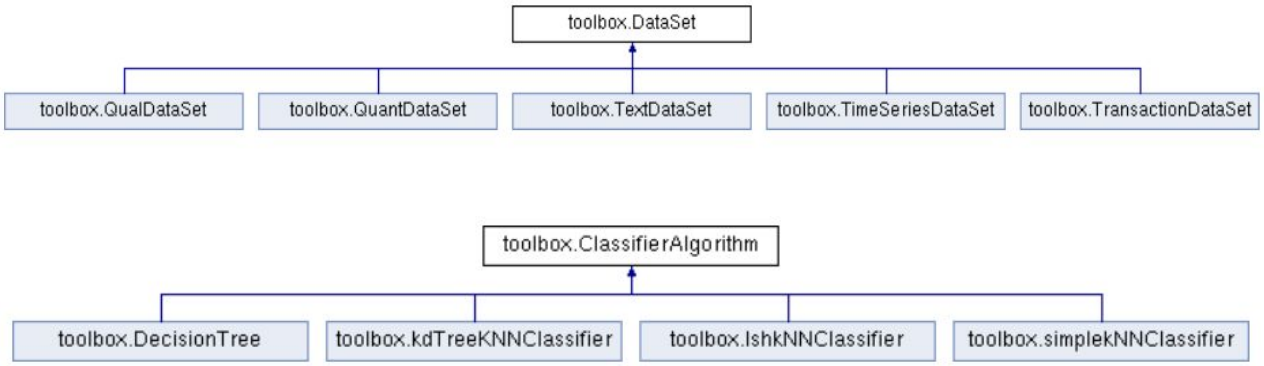
Summary of Object Oriented Toolbox Design

UML :



Summary :

Dataset: Super class contains common functions for all types of dataset, used to store, modify and return datasets.

TimeSeriesDataSet: Subclass inherits from Dataset that is used to store, modify and draw time series datasets. Includes time period selection and time type conversion functions.

TextDataSet: Subclass inherits from Dataset that is used to store, modify and explore text datasets. Includes stemming, lemmatization and stopwords removal to modify text.

QuantDataSet: Subclass inherits from Dataset that is used to store, modify and explore quantitative datasets. Explore the dataset with a line chart and a bar graph.

QualDataSet: Subclass inherits from Dataset that is used to store, modify and explore qualitative datasets. Explore the dataset with a pie chart and a bar graph.

TransactionDataSet: Subclass inherits from Dataset that is used to store, modify and explore transaction datasets. Contains inner class Rule for association rules calculation

ClassifierAlgorithm: Super class contains common functions for all classifiers, used to store training set, store train-test ratio and auto split the dataset to train and test by ratio.

simplekNNClassifier: Subclass inherits from ClassifierAlogrithm that uses k-nearest neighbors algorithm to classify the test set by training set, training label and stored k.

DecisionTree: Subclass inherits from ClassifierAlogrithm and Tree class to build a decision tree with training set and training label, pruning the tree and classifying the test set.

lshkNNClassifier: Subclass inherits from ClassifierAlogrithm using Locality Sensitive Hashing together with k-nearest neighbors algorithm to classify the test set.

kdTreeKNNClassifier: Subclass inherits from ClassifierAlogrithm to classify the test set by building kd Tree, an algorithm which uses a mixture of Decision trees and KNN.

Experiment: Experiment class that used to test compare between different algorithms. It contains functions for cross validation, confusion matrix and ROC curve.

HeterogenousDataSets: HeterogeneousDataSets class to store, load, clean, and explore a list of datasets. Contains functions for select and get type of datasets.

Summary of 3 advanced algorithms including time complexity analysis

Kd-KNN classifier Algorithm

Summary :

KD Tree is one such algorithm which uses a mixture of Decision trees and KNN to calculate the nearest neighbour. Theoretically, at each node we will save 3 things: dimension we split on, value we split on and tightest bounding box which contains all the points within that node. The algorithm could be divided into three parts. First, A Kd-tree class is built to save the information for classification. In the class, not only the distance is calculated, but also the tree is built. Then a KNN function is used to obtain the k-nearest neighbours. The second part is the node part, it saves information of the left child, the right child, the point and the axis. The third part is the large heap. Basically, We built a large list to save the heap and we make modifications on the heap. This class allows the action of adding new elements in, popping the top elements out and exchanging the elements to make adjustments to make sure that the heap is in order.

Actual code for createTree algorithm:

(This algorithm is under the class kdTreeKNNClassifier. A full code can be found in toolbox.)

```
def createTree(self, data,  current_node,axis=0):   # 1 step
    """
    function to create a kd-tree, recursion
    :type data: array-like, samples used to construct a kd-tree or sub_kd-t
    :type axis: int, between 0 and n_dim, dimension used to split data
    :type current_node: Node, the current 'root' node
    :return: None
    """
    if self.size == 0:                  # 2 steps: if, ==
        self.root = current_node        # 1 step
    self.size += 1                      # 1 step
    if data.shape[0] == 1:   # if no more than one sample, then stop iterat
        current_node.point = data[0, :]     # 3 steps: data[0,:], current_no
        current_node.axis = axis            # 2 steps: current_node.axis, =
        return                              # 1 step
    """
    step1: split the points with the median on this axis
    To find the median on target axis, we've got two ways:
    A. simply sort on target axis each time
    B. presort on each axis? but I haven't solved this at present
    """
    temp = data[data[:, axis].argsort()]    # 4 steps: data[:,axis], data[:,
    med = int(len(temp)/2) if len(temp) % 2 == 0 else int((len(temp)-1)/2)
    # find the 'first' med, this means that "<" goto left child, ">" goto r
                                        # 9 steps: if, len(), % , ==, i
```

```
    while med > 0 and temp[med,axis] == temp[med-1, axis]:  # 7 steps: while
        med -= 1                        # 1 step
    current_node.axis = axis            # 2 steps: current_node.axis, =
    current_node.point = temp[med]      # 2 steps: current_node.point, =
    tt = temp[med]                      # 2 steps: temp[med], tt=
    axis = (axis + 1) % self.n_dim      # 4 steps: self.n_dim, %, axis+1
    if temp[:med, :].shape[0] >= 1:     # 4 steps: if, temp[], temp[].sh
        tt = temp[:med, :]              # 2 steps: temp[], tt=
        current_node.left = self.Node()     # 2+5 steps: 2 steps- current_no
        self.createTree(temp[:med, :], current_node.left, axis)  # T(n/2)+2
    if temp[(med+1):, :].shape[0] >= 1:     # 4 steps: if, temp[], temp[].sh
        tt = temp[(med+1):, :]          # 2 steps: temp[], tt=
        current_node.right = self.Node()    # 2+5 steps: 2 steps- current_no
        self.createTree(temp[(med+1):, :], current_node.right, axis)  # T(n/
# ------------------------------------------------------------------------
# Total:
# T(n) = T(n/2)+70 = logn+70
# T(n) is O(n) = logn
# ------------------------------------------------------------------------
```

Total Time Complexity for kdTreeKNNClassifier:

(Details of time complexity calculation please refer to the code.)

---

p: len(self.testDS.shape[0]), n: length of testData

T(n) = (3+p)*logn + 351p + 404

T(n) is O(n) = p*logn

Decision Trees Algorithm

Summary :

Decision Tree is the classification algorithm that is suitable for both continuous and discrete labels. By passing the data_type parameter as “continuous” or “discrete”, buildTree function will build the tree corresponding to the parameter with the corresponding functions. In the tree building procedure, each node created with 9 attributes, including the decided attributes position of the dataset, left and right node, decided value for discrete tree, threshold for continuous tree, probability for each label stored in the node, etc. The decided attribute and its threshold value or decided value for each node is calculated based on the minimal Gini index. After building the tree node by node, the pruning function can help to prune the tree by comparing the NH value stored in a node and the sum of NH value in its left and right nodes.

Actual code for buildTree algorithm:

(This algorithm is under the class DecisionTree. A full code can be found in toolbox.)

```
def buildTree(self,X, y, data_type='continuous',maxDepth=5,currentDepth = 0):
                                                    # 1 step

    Gain_max, thre, d = 1, 0, 0                     # 3 steps: Gain_ma
    attr = ()                                       # 1 step
    X_small, y_small, X_big, y_big = [], [], [], []  # 4 steps: X_small

    ## when there is more than one y label
    if y.size> 1:                                   # 3 steps: y.size,

        if data_type == 'continuous':               # 2 steps: if, ==
            Gain_max, thre, d = self.attribute_based_on_Giniindex(X, y, data_
                                                    ## **-- x: len(inp
                                                    # 46 dx + 22 ldx -

        elif data_type == 'discrete':
            Gain_max, attr, d = self.attribute_based_on_Giniindex(X, y, data_
        if Gain_max >= 0 and len(list(set(y))) > 1 and len(np.unique(X, axis=
                                                    # 13 steps: if, ar

            if data_type == 'continuous':           # 2 steps: if, ==
                X_small, y_small, X_big, y_big = self.devide_group(X, y, thre
                                                    # 18+4 steps: 18 s

            elif data_type == 'discrete':
                X_small, y_small, X_big, y_big = self.devide_group_discrete(X
            currentDepth+=1                         # 2 steps: +=1, cu
            left_branch = self.buildTree(X_small, y_small,data_type,maxDepth,
                                                    # T(n/2)+1 steps,
            right_branch = self.buildTree(X_big, y_big,data_type,maxDepth,cur
                                                    # T(n/2)+1 steps,
```

```
                                                    # **-- l: len(list(set(y))),

            nh = self.NtHt(y)                       # 12l + 18k + 9t + 35 + 1 ste
            max_label = self.maxlabel(y)            # 7 + 1 steps: 7 steps - self
                                                    ## **-- k: len(label_count.ke
            prediction_prob=self.predictScore(y)    # 18k + 9t +20 +1 steps: 18k
            return self.decisionnode(d=d, attr= attr, thre=thre, NH=nh, lb=left_branch,
                                                    # 10 + 1 steps: 10 steps - se
        else:
            nh = self.NtHt(y)
            max_label =self. maxlabel(y)
            prediction_prob=self.predictScore(y)
            return self.decisionnode(results=y[0], NH=nh, max_label=max_label,data_prob=
    else:
        nh = self.NtHt(y)
        max_label = self.maxlabel(y)
        prediction_prob=self.predictScore(y)
        return self.decisionnode(results=y.item(), NH=nh, max_label=max_label,data_prob=

# ------------------------------------------------------------------------------------
# Total:
# x: len(inputted training set), d: len(X_attr), l: len(list(set(y_small))), k: len(labe
# T(x,d,l,k,t) = 2T(n/2)+ 46dx + 22 ldx -22lx -15x + 12 l + 36k + 18t + 148
#              = logn + 46dx + 22 ldx -22lx -15x + 12 l + 36k + 18t + 148
# T(x,d,l,k,t) is O(x,d,l,k,t) = logn + ldx + k + t
# ------------------------------------------------------------------------------------
```

Total Time Complexity for DecisionTree:

(Details of time complexity calculation please refer to the code.)

---

d: len(X_attr), l: len(list(set(y_small))), k: len(label_count.keys()),
t: len(list(total_label)), m: length of inputted training set, n: length of testData

T(d,l,k,t,m,n) = 3logn + 46dm + 22 ldm -22lm + 12 l + 36k + 18t + 6mn + 7m + 7n + 248
T(d,l,k,t,m,n) is O(d,l,k,t,m,n) = logn+ldm+k+t+mn

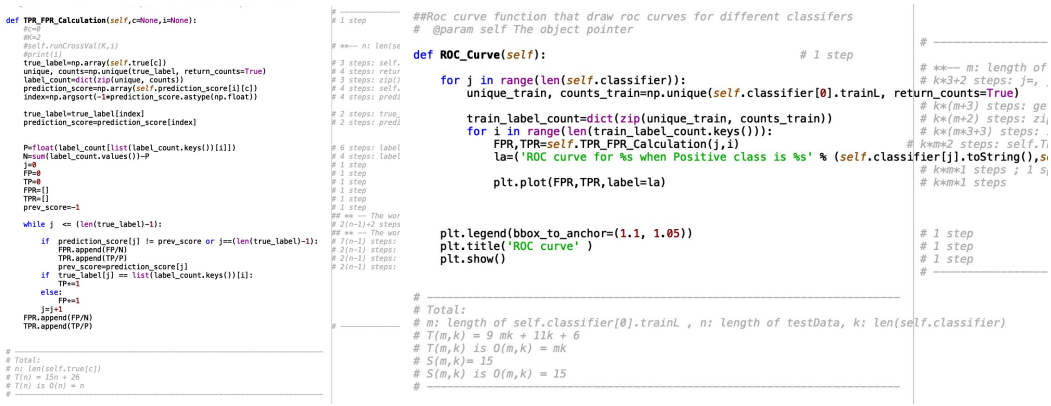
ROC Algorithm

Summary :

ROC is a two-dimensional graph where the true positive rate(TPR) lies on the y axis and false positive rate(FPR) is on the x axis. Before drawing the ROC curve, points representing the FPR and TPR are calculated by using different thresholds. Thresholds are usually scores of positive class for classification. For example, they could be prediction scores for positive classes. The scores are in-order, from the largest to the smallest. (Thresholds will be chosen from +∞ to -∞ according to the ranked scores.) Scores larger than the chosen threshold are classified as positive class, and scores less or equivalent to the chosen threshold are classified as negative class. Then TPR and FPR are calculated and stored. When there are instances with the same scores, there will be no extra point to be stored. Iterations used to calculate true positive rate and false positive rate starts from the first threshold equals the largest score until the last threshold equals the smallest score. For a two-class classification problem, ROC will choose a class as the positive class and the other as the negative class to evaluate the performance of classifiers. For a multiple-class classification problem, one class is chosen as a positive class, and the others are all classified as negative classes.

Actual code for ROC algorithm:

(This algorithm is under the class Experiment. A full code can be found in toolbox.)



```
def TPR_FPR_Calculation(self,c=None,i=None):
    #c=0
    #K=2
    #self.runCrossVal(K,i)
    #print(i)
    true_label=np.array(self.true[c])
    unique, counts=np.unique(true_label, return_counts=True)
    label_count=dict(zip(unique, counts))
    prediction_score=np.array(self.prediction_score[i][c])
    index=np.argsort(-1*prediction_score.astype(np.float))

    true_label=true_label[index]
    prediction_score=prediction_score[index]


    P=float(label_count[list(label_count.keys())[i]])
    N=sum(label_count.values())-P
    j=0
    FP=0
    TP=0
    FPR=[]
    TPR=[]
    prev_score=-1

    while j  <= (len(true_label)-1):

        if  prediction_score[j] != prev_score or j==(len(true_label)-1):
            FPR.append(FP/N)
            TPR.append(TP/P)
            prev_score=prediction_score[j]
        if  true_label[j] == list(label_count.keys())[i]:
            TP+=1
        else:
            FP+=1
        j=j+1
    FPR.append(FP/N)
    TPR.append(TP/P)


# ------------------------------------------------------------------------
# Total:
# n: len(self.true[c])
# T(n) = 15n + 26
# T(n) is O(n) = n
# ------------------------------------------------------------------------
```

```
##Roc curve function that draw roc curves for different classifers
#  @param self The object pointer

def ROC_Curve(self):                                          # 1 step

    for j in range(len(self.classifier)):
        unique_train, counts_train=np.unique(self.classifier[0].trainL, return_counts=True)

        train_label_count=dict(zip(unique_train, counts_train))
        for i in range(len(train_label_count.keys())):
            FPR,TPR=self.TPR_FPR_Calculation(j,i)
            la='ROC curve for %s when Positive class is %s' % (self.classifier[j].toString(),s

            plt.plot(FPR,TPR,label=la)



    plt.legend(bbox_to_anchor=(1.1, 1.05))                    # 1 step
    plt.title('ROC curve' )                                   # 1 step
    plt.show()                                                # 1 step


# ------------------------------------------------------------------------
# Total:
# m: length of self.classifier[0].trainL , n: length of testData, k: len(self.classifier)
# T(m,k) = 9 mk + 11k + 6
# T(m,k) is O(m,k) = mk
# S(m,k)= 15
# S(m,k) is O(m,k) = 15
# ------------------------------------------------------------------------
```

Total Time Complexity for ROC:

(Details of time complexity calculation please refer to the code.)

---

m: length of inputted training set, n: length of testData, k: len(self.classifier)
T(m,n,k) = 9 mk +11k + 15n + 32
T(m,n,k) is O(m,n,k) = mk+n
