Executive Summary Report

Group Member:

Ying Liu: yl1206

Guiming Xu: gx26

Class: ANLY 555

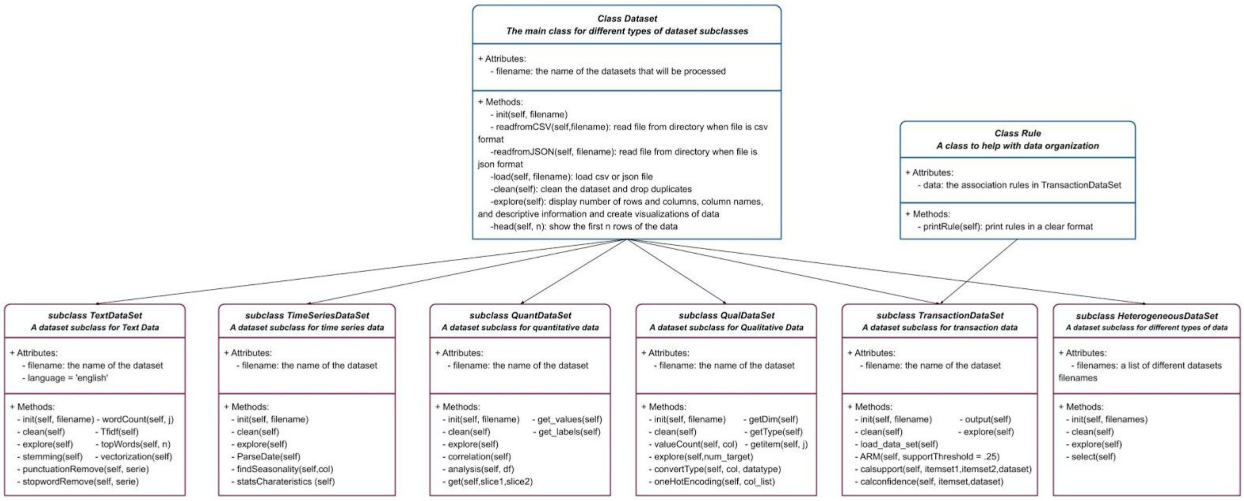
Professor: Jeremy Bolton

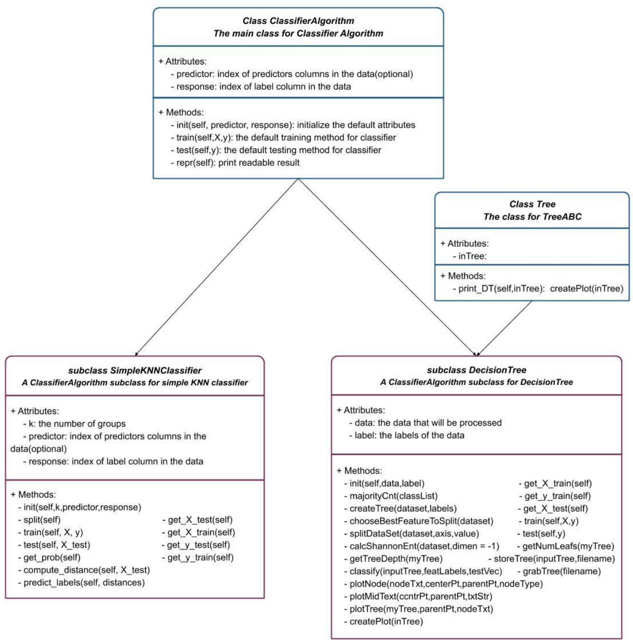
Date: 11/27/2020

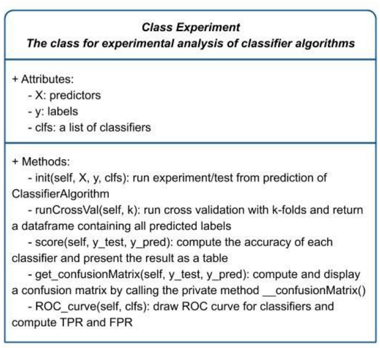
Contents

Part One: Summary of Object Oriented Toolbox Design

Class Hierarchy Diagram:







Introduction of each class:

DataSet:

This class is to store, clean, and explore the dataset inputted. This class is able to deal with Qualitative Dataset, Time Series Dataset, Text Dataset, Quantitative Dataset, Transaction Dataset and this class also can deal with a list of datasets.

ClassifierAlgorithm:

This class is mainly to help users to do classifier analysis, and there are two methods under this class, which are Decision Tree and Simple KNN. The Decision Tree method uses C4.5 and Shannon entropy to classifier, and Simple KNN just uses K-Nearest-Neighbor.

Experiment:

After we use the classifier algorithm, we can use Experiment class to do experimental verification. For example, runCrossVal method can run cross validation with k-folds and return a dataframe containing all predicted labels. Score and confusion matrix methods can help us compare the accuracy of each classifier. Finally the ROC curve method can draw ROC curves for classifiers and compute TPR and FPR. All in all, this class can help us check whether a model works well or not.

Tree:

This class is an implementation of a decision tree, and it is able to create trees depending on datasets.

Rule:

This class is an implementation of Transaction, and it is able to create the whole rules under given datasets.

Part Two: Summary of 3 advanced algorithm

(1) Decision Tree

Part One: (Explanation → Code with Time Complexity)

Explanation: These functions are preset functions, and can help find the best split variable values, and calculate the threshold.

Function Explanation & Time Complexity:

calcShannonEnt: calculate the shannon entropy:
T(n) = O(n^3*log(n))
Time Complexity: There are two for loops in this function, and there is an if statement to judge keys in the dictionary.

splitDataSet: Split dataset by the value input
T(n) = O(n^2)
Time complexity: There is a for loop in the function, and we have a statement to filter the features, and append the filter feature into the new list.

chooseBestFeatureToSplit: Split the by the feature depending on shannon entropy
T(n) = O(n^3*log(n))
Time Complexity: There is one for loop in the function, and one more nested for loop in the function and one more if statement, in the nested for loop, we will use splitDataSet to split the input data.

majorityCnt: Vote
T(n) = O(n)
Time Complexity: to create a dictionary

Code:

```
def calcShannonEnt(dataset,dimen = -1):
    '''
    to calculate the shannon entropy

    dataset: the input dataset

    dimen: rule to split dataset


    return value: shannon entropy
    '''

############### O(n) = n^2*log(n) #################
############### S(n) = 6(6*6 + 6*6*3*3) = 2160 ######

    numEntries = len(dataset) # get the dimension
    labelCnt = {}
    for currentLabel in dataset:
        if currentLabel[dimen] not in labelCnt.keys():
            labelCnt[currentLabel[dimen]] = 0
        labelCnt[currentLabel[dimen]] += 1
    shannonEnt = 0.0
    for key in labelCnt: # calculate the shannon entropy
        prob = float(labelCnt[key])/numEntries
        shannonEnt -= prob * math.log(prob,2)
    return shannonEnt

def splitDataSet(dataset,axis,value):
    '''
    to split dataset by the rule

    dataset: matrix

    axis: value in the each row

    value: rule to split
    '''

    ############### O(n) = n^2 #################
    ############### S(n) = 36  #################

    retDataSet = []
    for featVec in dataset:
        if featVec[axis] == value: # filter the features
            reducedFeatVec = featVec[:axis]
            reducedFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec)
    return retDataSet
```

```python
def chooseBestFeatureToSplit(dataset):
    '''
    split by the least shannon entropy

    dataset: matrix
    '''

    ############### O(n) = n^2*log(n) + n^2*n*log(n)*n = n^3*log(n) ##########
    ############### S(n) = 4 + 6*408*4*2 = 19588                    ##########

    numFeatures = len(dataset[0]) - 1
    baseEntropy = calcShannonEnt(dataset,dimen = -1)
    bestInfoGain = 0.0
    bestFeature = -1
    for i in range(numFeatures):
        featList = [examples[i] for examples in dataset]
        uniqueVals = set(featList)
        newEntropy = 0.0
        for value in uniqueVals:
            subDataSet = splitDataSet(dataset,i,value)
            prob = len(subDataSet)/float(len(dataset))
            newEntropy += prob*calcShannonEnt(subDataSet,dimen = -1)
        infoGain = baseEntropy - newEntropy
        if infoGain > bestInfoGain:
            bestInfoGain = infoGain
            bestFeature = i
    return bestFeature

def majorityCnt(classList):
    '''
    Determination of the final label by majority vote
    '''

    ############### O(n) = n           #######
    ############### S(n) = 408*2 = 916 #######

    classCnt = {}
    for vote in classList:
        if vote not in classCnt.keys():
            classCnt[vote] = 0
        classCnt[vote] += 1

    # classCount.iteritems() decomposing the classCount dictionary into a tuple list.
    # operator.itemgetter(1) sorting the tuple in the order of the second element.

    sortedClassCnt = sorted(classCnt.items(), key = operator.itemgetter(1),reverse=True)
    return sortedClassCnt[0][0]
```

Part Two: (Explanation → Code with Time Complexity)

Explanation: After we use functions shown above, we can use the splitting method to create a tree, and get the information of this tree like number leaf, number of depth.

Function Explanation & Time Complexity:

createTree: create decision tree by functions above

T(n) = O(n^3*log(n))

Time complexity: There are two if statements and one for loop, moreover, we also use chooseBestFeatures function to get the best feature to split, and after each selection, we will delete the best feature in the list.

getNumLeafs: get number of leafs of decision tree

T(n) = O(n)

Time Complexity: use for loop to get number

getTreeDepth: get the depth of tree

T(n) = O(n)

Time complexity: use for loop to get the depth

Code:

```
def createTree(dataset,labels):
    '''
    create the decision tree

    dataset: dataset in matrix format

    labels: the whole labels except the target
    '''

    ############### O(n) = n^3*log(n) + n^2 + n^2 = n^3*log(n)     #################
    ############### S(n) = 408 + 408 + 408 + 19588 + 408 + 4 + 4988 = 21224 #######

    classList = [example[-1] for example in dataset]
    if classList.count(classList[0]) == len(classList):
        return classList[0]
    if len(dataset[0]) == 1:
        return majorityCnt(classList)
    bestFeat = chooseBestFeatureToSplit(dataset)
    bestFeatLabel = labels[bestFeat]
    myTree = {bestFeatLabel:{}}
    del(labels[bestFeat])
    featValues = [example[bestFeat] for example in dataset]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        subLabels = labels[:]
        myTree[bestFeatLabel][value] = createTree(splitDataSet(dataset,bestFeat,value),subLabels
    return myTree
```

```python
def getNumLeafs(myTree):
    '''
    to get the number of leafs
    '''

    ############### O(n) = n     #################
    ############### S(n) = 408*4 = 1632 #########

    numLeafs = 0
    firstStr = list(myTree.keys())[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':
            numLeafs += getNumLeafs(secondDict[key])
        else:
            numLeafs += 1
    return numLeafs

def getTreeDepth(myTree):
    '''
    get the dimension of the decision tree

    myTree: decision tree in dictionary format

    return value: the dimension
    '''

    ############### O(n) = n     #################
    ############### S(n) = 408*4 = 1632 #########

    maxDepth = 0
    firstStr = list(myTree.keys())[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':
            thisDepth = 1 + getTreeDepth(secondDict[key])
        else:
            thisDepth = 1
        if thisDepth > maxDepth:
            maxDepth = thisDepth
    return maxDepth

def classify(inputTree,featLabels,testVec):
    '''
    decision classifier

    inputTree: Enter a decision tree classifier that has been trained with a structure of dictio
    featlabels: A list of feature labels, the members of which are strings that represent each m
    testVector: A test vector without a classification label, but whose members represent the me
    '''

    ############### O(n) = n^3          ###########
    ############### S(n) = 408*4 = 1632 ###########

    valueOfFeat = inputTree
    while True:
        if isinstance(valueOfFeat, dict):
            firstStr = list(valueOfFeat.keys())[0]
            secondDict = valueOfFeat[firstStr]
            featIndex = featLabels.index(firstStr)
            key = testVec[featIndex]
            try:
                valueOfFeat = secondDict[key]
            except KeyError:
                classLabel = 6
                break
        else:
            classLabel = valueOfFeat
            break
    return classLabel
```

Part Three: (Explanation → Code with Time Complexity)

Explanation: Then, we can use matplotlib to plot the tree like hierarchy diagram, and we can also plot the tree in another format (http://mshang.ca/syntree/)

Code:

```
def plotNode(nodeTxt,centerPt,parentPt,nodeType):
    '''
    plot one node
    '''
    createPlot.axl.annotate(nodeTxt,xy=parentPt,xycoords='axes fraction',
    xytext=centerPt,textcoords='axes fraction',
    va='center',ha='center',bbox=nodeType,arrowprops=arrow_args)

def plotMidText(ccntrPt,parentPt,txtStr):
    xMid = (parentPt[0]-ccntrPt[0])/2.0 + ccntrPt[0]
    yMid = (parentPt[1]-ccntrPt[1])/2.0 + ccntrPt[1]
    createPlot.axl.text(xMid,yMid,txtStr)

def plotTree(myTree,parentPt,nodeTxt):
    '''
    plot the tree
    '''
    numLeafs = getNumLeafs(myTree)
    depth = getTreeDepth(myTree)
    firstStr = list(myTree.keys())[0]
    cntrPt = (plotTree.xoff + (1.0 + float(numLeafs))/2.0/plotTree.totalW,plotTree.yoff)
    plotMidText(cntrPt,parentPt,nodeTxt)
    plotNode(firstStr,cntrPt,parentPt,decisionNode)
    secondDict = myTree[firstStr]
    plotTree.yoff = plotTree.yoff - 1.0/plotTree.totalD
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == "dict":
            plotTree(secondDict[key],cntrPt,str(key))
        else:
            plotTree.xoff = plotTree.xoff + 1.0/plotTree.totalW
            plotNode(secondDict[key],(plotTree.xoff,plotTree.yoff),cntrPt,leafNode)
            plotMidText((plotTree.xoff,plotTree.yoff),cntrPt,str(key))
    plotTree.yoff = plotTree.yoff + 1.0/plotTree.totalD

def createPlot(inTree):
    '''
    to plot the tree in dictionary format
    '''
    fig = plt.figure(1,facecolor='white')
    fig.clf()
    axprops = dict(xticks=[],yticks=[])
    createPlot.axl = plt.subplot(111,frameon=False)
    plotTree.totalW = float(getNumLeafs(inTree))
    plotTree.totalD = float(getTreeDepth(inTree))
    plotTree.xoff = -0.5/plotTree.totalW
    plotTree.yoff = 1.0
    plotTree(inTree,(0.5,1.0),'')
    plt.show()
```

Part Four: (Explanation → Code with Time Complexity)

Explanation: Finally, we need to build Tree class firstly, and then create subclass DecisionTree, and this class is inherited from both Classifier Algorithm class and Tree class. In the DecisionTree subclass, we are able to train the model, and use the trained model to test on the test data set.

Function Explanation & Time Complexity:

Train: get a decision tree model based on the dataset

T(n) = O(n^3*log(n))

Time Complexity: Inherited from Tree class, and has the same time complexity as createTree

Test: test the trained model on a given test set

T(n) = O(n^4*log(n))

Time Complexity: Inherited from Tree class, and has the same time complexity as classify function with one more for loop.

Code:

```
class Tree(object):
    '''
    The class for TreeABC
    '''
    def print_DT(self,inTree):
        createPlot(inTree)

class DecisionTree(Tree, ClassifierAlgorithm):
    '''
    Inheritate from Tree and ClassifierAlgorithm
    '''
    def __init__(self,data,label):
        self.myData = data
        self.label = label
        # print(label)
        # print(self.label)
        self.predict_result = []
        self.train_data = data[:len(data)//2] # use half of dataset to train
        self.test_data = data[len(data)//2:] # use other half of dataset to check

    def get_X_train(self):
        return self.train_data

    def get_y_train(self):
        return self.label

    def get_X_test(self):
        return self.test_data

    def train(self,X,y):
        '''
        to train the decision tree model
        '''
        ############### O(n) = n^3*log(n)    #################
        self.dt = createTree(X,y)
        print(self.dt)
    def test(self,y):
        '''
        test on the X_test and then plot the tree
        '''
        ############### O(n) = n^4 * log(n)                           #################
        self.true_result = []
        for spline_data in y:
            predict_res = classify(self.dt,['citric acid', 'residual sugar', 'density', 'pH', 'sulphates',
            true_res = spline_data[-1]
            self.predict_result.append(predict_res)
            self.true_result.append(true_res)
        print("Test done!")
        print('Prediction: ')
```

(2) Apriori Algorithm

Part One: (data preparation)

Explanation: __init__ , load_data_set, and clean methods help me prepare the transaction data. Specifically, __init__ method inherited from the main class Dataset can read and load the data as a dataframe. After loading, load_data_set method will convert the dataframe into transaction data. Then, the clean method cleans the transaction data by dropping NA values.

Code:

```
class TransactionDataSet(DataSet):
    """ A dataset subclass for transaction data """

############### O(n) = n^2 #################

    def __init__(self, filename):
        """
        Create a new transaction dataset instance

        filename          the name of the dataset
        """
        # inherit from class Dataset
        super().__init__(filename) # read and load the csv data   T(n) = 3


    def load_data_set(self):
        """
        Transfer the data frame to transaction data
        """
        transactions = []          # T(n) = 1
        for i in range(0, 19):     # T(n) = 20*5 = 100
            transactions.append([str(self.df.values[i,j]) for j in range(0, 4)])

        return transactions

    def clean(self):
        """
        Clean the transaction data
        """
        transactions = self.load_data_set() # T(n) = 101
        cleanTransactions = [[i for i in nested if i != 'nan'] for nested in transactions] # T(

        return cleanTransactions
```

Time Complexity: (line by line step count shown in the picture)

__init__ method inherited from the Dataset has 3 steps.

Load_data_set method has 101 (n+1) steps.

In the worst case, the clean method has 10101 (n^2+n+1) steps. Therefore,

T(n) = 3 + 101 + 10101 = 10205

= n^2 + n + 1 + n + 1 + 3

= n^2 + 2n + 5

= **O(n^2)**

Part Two: (Apriori Algorithm)

Explanation: explore, __ARM__, calsupport, and calconfidence methods achieve the apriori algorithm. Specifically, the explore method calls the private method __ARM__() that performs association rule mining by apriori algorithm. __ARM__() first creates the 1-item candidate by measuring the support of each item in the dataset and then creates a list to store the candidate whose support is greater than supportThreshold. Next, create the 2-item candidate by using the list from the previous step, check whether subsets in itemset are frequent or not, and remove infrequent itemset from the list. List 2 is generated as list 1 by comparing the support of candidates with supportThreshold and storing those whose support is greater than the threshold. The same for the 3-item candidates. Finally, generate association rules with List 1,2,3 and compute support and confidence for each rule. The calsupport and calconfidence functions used by __ARM__() work to compute the support and confidence for all Rules.

Code:

```
def explore(self, supportThreshold = .25):
    """
    supportThreshold: the minimum support
    ------------
    Compute the Support, Confidence, and Lift for all Rules above supportThreshold by callin
    """

    return self.__ARM__(supportThreshold = .25)

def __ARM__(self, supportThreshold = .25):
    """
    supportThreshold: the minimum support
    ------------
    Perform association rule mining.
    This is a priavte method and will be called by explore().
    """
    dataset = self.clean()    # T(n) = 201

    # create the 1-item candidate by measuring the support_count of each item in the dataset
    C1 = list()               # T(n) = 1
    for i in dataset:         # T(n) = n^2 = 100^2
        for j in i:
            C1.append(j)
    C1 = set(C1)

    # create item_set L1 by comparing C1 support_count with supportThreshold
    L1 = list()               # T(n) = 1
    L1_tem = list()           # T(n) = 1
    for i in C1:              # T(n) = n^2 = 100^2
        num = 0
        for j in dataset:
            if(i in j):
                num += 1
        #print("L1",i,'supportThreshold = ',num)
        if(num>=supportThreshold):
            L1.append(i)
        else:
            L1_tem.append([i])
```

```python
# create the 2-item candidate by using Item_set L1 from the previous step
C2 = list()                 # T(n) = 1
for i in L1:                # T(n) = n^2 = 100^2
    for j in L1:
        if(i<j):
            C2.append(sorted([i,j]))

# check all subsets in itemset are frequent if not, remove respective itemset from the l
C2_tem = C2.copy()          # T(n) = 100
for i in C2_tem:            # T(n) = 100
    for n in L1_tem:
        if(set(n).issubset(set(i))):
            print(i)
            C2.remove(i)

# create item_set L2 by comparing candidate_set C2 with supportThreshold
L2 = list()                 # T(n) = 1
L2_tem = list()             # T(n) = 1
for i in C2:                # T(n) = n^2 = 100^2
    num1 = 0
    for j in dataset:
        if(set(i).issubset(set(j))):
            num1 +=1
    if(num1>=supportThreshold):
        #print('L2:',i,'supportThreshold =',num1)
        L2.append(i)
    else:
        L2_tem.append(i)

# create the 3-item candidate by using Item_set L2 from the previous step
C3 = list()                 # T(n) = 1
for i in L2:                # T(n) = n^2 = 100^2
    for j in L2:
        length = len(i)
        set1 = set(i[0:length-1])
        set2 = set(j[0:length-1])
        result = list(set1.difference(set2))
        if(result==[] and list(set(i).difference(set(j)))!=[]):
            C3_temp = set(i).union(set(j))
            C3.append(sorted(list(C3_temp)))
C3_df = pd.DataFrame(C3)
C3 = C3_df.drop_duplicates().values.tolist()

# check all subsets in itemset are frequent if not, remove respective itemset from the l
C3_tem = C3.copy()          # T(n) = 100
for i in C3_tem:            # T(n) = n^2 = 100^2
    for j in L2_tem:
        if(set(j).issubset(set(i))):
            C3.remove(i)

# create item_set L3 by comparing candidate_set C3 with supportThreshold
L3 = list()                 # T(n) = 1
for i in C3:                # T(n) = n^2 = 100^2
    num = 0
    for j in dataset:
        if set(i).issubset(j):
            num += 1
    #print('L3',i,'supportThreshold = ',num)
    if num>=supportThreshold:
        L3.append(i)

# create the association rules, the rules will be a list.
# compute support and confidence
result = list()             # T(n) = 1
for i in L2:                # T(n) = n^2 = 100^2
    for j in L3:
        if set(i).issubset(set(j)):
            support = self.calsupport(j,i,dataset)
            confidence = self.calconfidence(j,dataset)
            sublist = list(set(j)-set(i))
            result.append([i,sublist,support,confidence,support/confidence])

return result
```

```python
def calsupport(self, itemset1,itemset2,dataset):
    """
    itemset1: the list for items whose support is larger than supportThreshold.
              Here we use item_set L3.
    itemset2: the list for items whose support is larger than supportThreshold.
              Here we use item_set L2.
    dataset: a list of Transactions, every transaction is also a list, which contains severa
    ------------
    Calculate the support
    """
    num1 = 0                # T(n) = 1
    num2 = 0                # T(n) = 1
    for i in dataset:       # T(n) = n
        if set(itemset1).issubset(i):
            num1 += 1
    for i in dataset:       # T(n) = n
        if set(itemset2).issubset(i):
            num2 += 1
    return num1/num2


def calconfidence(self, itemset,dataset):
    """
    itemset: the list for items whose support is larger than supportThreshold.
             Here we use item_set L3.
    dataset: a list of Transactions, every transaction is also a list, which contains severa
    ------------
    Calculate the confidence
    """
    num1 = 0                 # T(n) = 1
    length = len(dataset)    # T(n) = 2
    for i in dataset:        # T(n) = n
        if set(itemset).issubset(i):
            num1 += 1
    return num1/length
```

Time Complexity: (line by line step count shown in the picture)

The explore method calls the __ARM__ method with 1 step.

In the worst case, __ARM__ method has 80510 (8n^2+5n+10) steps.

The calsupport and calconfidence methods have 202 (2n+2) and 103 (n+3) steps respectively.

Therefore,

T(n) = 1 + 80510 + 202 + 103 = 80816

= 8n^2 + 5n + 10 + 2n + 2 + n + 3 + 1

= 8n^2 + 8n + 16

= **O(n^2)**

Part Three: (output)

Explanation: class Rule and output method help with data organization. Specifically, Rule class has a member method printRule, which was designed for transaction dataset and can print the top 10 rules along with support, confidence, lift, these three measures to the console. The output method will call the Rule class and display the result.

Code:

```
    def output(self):
        """
        Call the Rule class.
        Display the top 10 rules along with these three measures to the console.
        """
        data = self.explore()  # T(n) = 1
        r = Rule(data)         # T(n) = 32

        return r.printRule()


class Rule:
    """ A class to help with data organization"""

    def __init__(self, data):
        """
        data: the association rules in TransactionDataSet
        ------------
        Run rule to help with data organization
        """
        self.data = data       # T(n) = 1

    def printRule(self):
        aim = ['support', 'confidence', 'lift']   # T(n) = 1
        for i in range(len(aim)):                 # T(n) = 3*10 = 30
            result = sorted(self.data, key = lambda x : x[i+2], reverse = True)[:10]
            print("Top 10 rules for", aim[i], ":")
            for j in result:
                print(j[0],'==>',j[1],'Support : ',j[2],"Confidence : ",j[3],'Lift : ',j[4])
            print("=========================================================\n")
```

Time Complexity: (line by line step count shown in the picture)

The output method has 33 (n+3) steps.

Class Rule has 32 (n+2) steps. Therefore,

$T(n) = 33 + 32 = 65$

$= n + 3 + n + 2$

$= 2n + 5$

$= \mathbf{O(n)}$

(3) ROC function

Explanation: ROC_curve method is under the class Experiment. It can be used to draw ROC curves for each classifier. Specifically, it will first loop through the classifiers list to get test and predicted labels for each classifier, calculate the probability of labels, and iterate through all random thresholds to determine the fraction of true positives and false positives found at the threshold. Then, it will loop and calculate each label's TPR and FPR and save them to a list. Finally, compute the average of TPR and FPR for all labels and draw ROC curves. During the process, countX function will be used to count how many specific labels are in a list.

Code:

```
def ROC_curve(self, clfs):
    '''
    Draw ROC curve for classifier
    compute TPR and FPR
    '''
    for c in clfs:
        #if c == simpleKNN:
    ############## S(n) = 361 ##############
    ############## T(n) = O(n^2)##############

        c.split()
        X_train = c.get_X_train().values           # T(n) = 1
        y_train = c.get_y_train().quality.values   # T(n) = 1
        X_test = c.get_X_test().values             # T(n) = 1
        y_test = c.get_y_test().quality.values     # T(n) = 1
        c.train(X_train, y_train)                  # T(n) = 2
        y_pred, p = c.test(X_test)                 # T(n) = 3n+6
        y_pred = y_pred.tolist()                   # T(n) = 1
        prob = c.get_prob()                        # T(n) = n^2
        # Iterate thresholds from 0.0, 0.01, ... 1.0
        thresholds = np.arange(0.0, 1.01, .01)              #####space:101, T(n) = 1
        # iterate through all thresholds and determine fraction of true positives
        # and false positives found at this threshold
        labelclass = [5,6,7]                                #####space:3, T(n) = 1
        fpr_lst =[]                                # T(n) = 1
        tpr_lst = []                               # T(n) = 1
```

```
            for j in range(len(labelclass)):           # T(n) = 3*101

                fpr = []
                tpr = []
                P = countX(y_pred, labelclass[j])
                N = len(y_pred) - P

                for thresh in thresholds:
                    FP=0
                    TP=0
                    # loop and calculate each label's TPR and FPR (one versus all)
                    for i in range(len(prob)):
                        if (prob[i] > thresh):
                            if y_pred[i] == labelclass[j] and y_test[i] == labelclass[j]:
                                TP = TP + 1
                            if y_pred[i] == labelclass[j] and y_test[i] != labelclass[j]:
                                FP = FP + 1
                    fpr.append(FP/float(N))
                    tpr.append(TP/float(P))
                # save to lst
                fpr_lst.append(fpr)
                tpr_lst.append(tpr)

            # compute average of TPR and FPR for all labels
            fpr_lst_df = pd.DataFrame(fpr_lst)          # T(n) = 1
            #fpr_avg = list(fpr_lst_df.mean (axis=0))
            tpr_lst_df = pd.DataFrame(tpr_lst)          # T(n) = 1
            #tpr_avg = list(tpr_lst_df.mean (axis=0))
            plt.title('ROC Curve for KNN')
            plt.plot(list(fpr_lst_df.loc[0]), list(tpr_lst_df.loc[0]),linestyle='solid')
            plt.plot(list(fpr_lst_df.loc[1]), list(tpr_lst_df.loc[1]),linestyle='solid')
            plt.plot(list(fpr_lst_df.loc[2]), list(tpr_lst_df.loc[2]),linestyle='solid')
            plt.legend(['label 5 vs (6,7)', 'label 6 vs (5,7)', 'label 7 vs (5,6)'], loc='upper right')
            plt.xlabel("FPR")
            plt.ylabel("TPR")
            plt.show()


def countX(lst, x):
    '''
    count how many of specific labels in a lst
    '''
    count = 0                    # T(n) = 1
    for ele in lst:              # T(n) = 3n
        if (ele == x):
            count = count + 1
    return count
```

Time Complexity: (line by line step count shown in the picture)

The ROC_curve method has 168010 ($n^2+3n+322$) steps.

The countX function has 1225 ($3n+1$) steps. Therefore,

$T(n) = n^2 + 3n + 322 + 3n+1$

$= n^2 + 6n + 323$

$=$ **$O(n^2)$**