Marcus Berger

Scott Carda

Alex Nienhueser

Christine Sorensen

**Data Mining Deliverable**

**Documentation of the Data Used**

**Where We Found the Data**

<http://mlr.cs.umass.edu/ml/datasets/Congressional+Voting+Records>

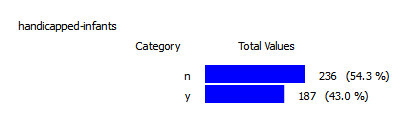
**Data Information**

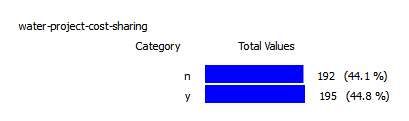
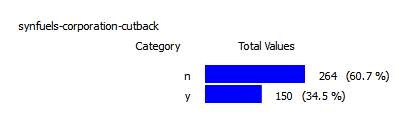
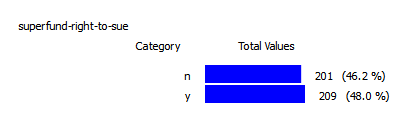
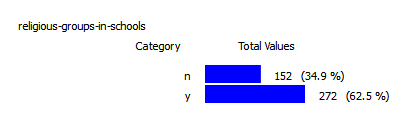
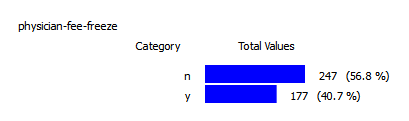
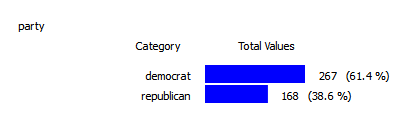
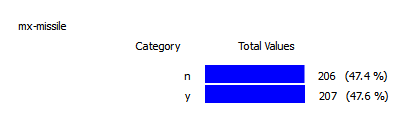
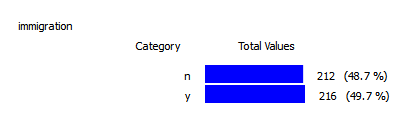
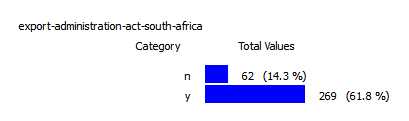
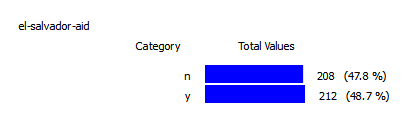
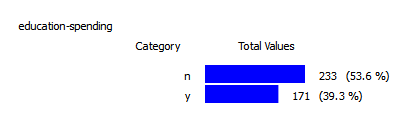
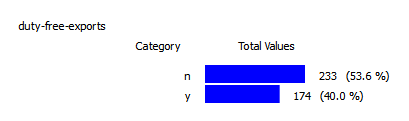
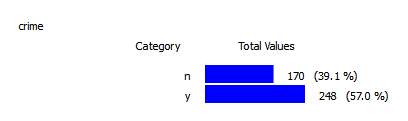
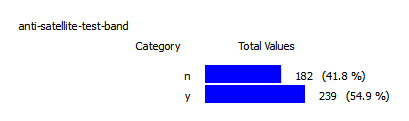
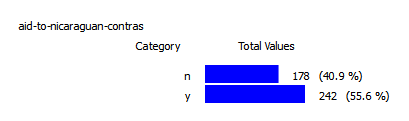
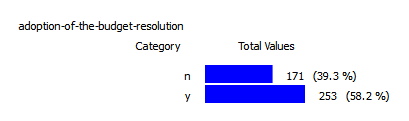
* Number of object: 435 instances
* Number of attributes: 16

1. handicapped-infants: 2 (y,n)   
2. water-project-cost-sharing: 2 (y,n)   
3. adoption-of-the-budget-resolution: 2 (y,n)   
4. physician-fee-freeze: 2 (y,n)   
5. el-salvador-aid: 2 (y,n)   
6. religious-groups-in-schools: 2 (y,n)   
7. anti-satellite-test-ban: 2 (y,n)   
8. aid-to-nicaraguan-contras: 2 (y,n)   
9. mx-missile: 2 (y,n)   
10. immigration: 2 (y,n)   
11. synfuels-corporation-cutback: 2 (y,n)   
12. education-spending: 2 (y,n)   
13. superfund-right-to-sue: 2 (y,n)   
14. crime: 2 (y,n)   
15. duty-free-exports: 2 (y,n)   
16. export-administration-act-south-africa: 2 (y,n)

**Visualization**

|  |  |  |
| --- | --- | --- |
|  | **Republican** | **Democrat** |
| Handicapped Infants | n | y |
| Water Project Cost Sharing | y | n |
| Adoption of the Budget Resolution | n | y |
| Physician Fee Freeze | y | n |
| el Salvador Aid | y | n |
| Religious Groups in Schools | y | n |
| Anti-satellite test Band | n | y |
| Aid to Nicaraguan Contras | n | y |
| Mx-missile | n | y |
| Immigration | n | y |
| Synfuels Corporation Cutback | n | n |
| Education Spending | y | n |
| Superfund Right to Sue | y | n |
| Crime | y | n |
| Duty-free Exports | n | y |
| Export Administration Act South Africa | y | y |





**The Data**

The data file is named data.csv and is provided in the submission.

**The Study**

**Hypothesis**

We hypothesize that we can use a semi-supervised classification algorithm using a hybrid model that combines a classification tree made of the labeled data and using k-means clustering for the unlabeled data to make a classifier.

**How the Study was conducted**

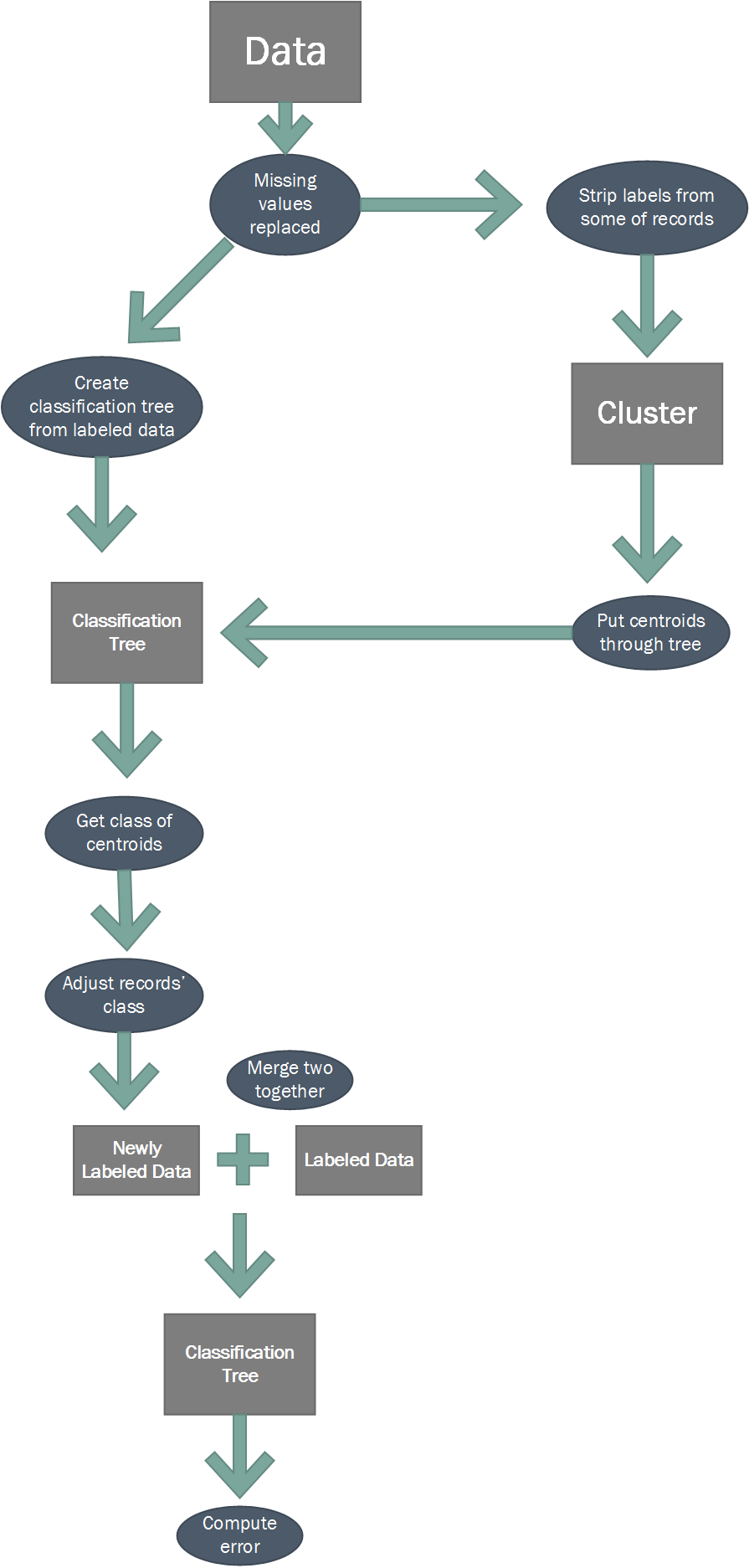
The data was preprocessed by adding labels to the attributes and removing the missing values with a random ‘y’ or a ‘n.’ The preprocessing step also consisted of stripping off the class labels from a random number of records.

Once our program produced a classification model, the model was used to classify the original data. The classes from original data and the classes produces by our model were compared. In each record in the original data and its corresponding record in the new data were equated in their class to see if the class in the new data returned the same. The number of correct and incorrect predictions were kept track of and used to create a confusion matrix. Our group ran the program 25 times and computed the average error to prove how well our classifier was.

**Documentation of the Design Process**

Semi-supervised learning is the creation of a classification model from data that is partially labeled with classes. From our Data Mining class, we know that supervised learning, which is done on data that is labeled, can be done with a decision tree, and that unsupervised learning, which is done on data without labels, can be done using the *k*-means clustering algorithm. Knowing that we could perform learning on part of the data with one method and the other part with another method, we decided to combine the two methods together to create a hybrid algorithm for semi-supervised learning.

The algorithm partitions the data into labeled and unlabeled data, then builds a decision tree using supervised learning on the labeled data. The algorithm then clusters the unlabeled data using the *k*-means algorithm, which is unsupervised learning. The two algorithms are then combined by having the decision tree classify the centroids from the clusters produced by the *k*-means and applying the resulting class to all the data points in the cluster, resulting in all of the data having class labels. Finally the algorithm produces a new decision tree from the fully labeled data, resulting in the semi-supervised learning of the data that was originally partially labeled.



**Running The Code**

The project is run by entering the following command into the command line:

>python SemiSupervised <filename>

<filename>: the name of the csv file that holds the data

SemiSupervised: the name of the directory containing \_\_main\_\_.py

Note: This is the zip file submitted, and could be run as the zip file, in which case this would be “SemiSupervised.zip”

instead of “SemiSupervised”

**The Directory Structure**

SemiSupervised.

| \_\_main\_\_.py

|

\---SemiSupervised

ClassificationTree.py

KMeans.py

Learner.py

Test.py

\_\_init\_\_.py

**The Code**

**\_\_main\_\_.py:**

import sys

import SemiSupervised as ss

def Main():

if len( sys.argv ) != 2:

print("Usage error")

return False

tree = ss.CreateLearner( \*ss.ReadDataFile( sys.argv[1] ) )

attributes, data = ss.GetData(sys.argv[1])

classes = []

for record in data:

record = ss.ConvertToDictionary(record, attributes)

classes.append(tree.Classify(record))

ss.ErrorComp( data, classes )

ss.conf\_matrix( data, classes )

return tree

if \_\_name\_\_ == '\_\_main\_\_':

Main()

**\_\_init\_\_.py:**

from .ClassificationTree import ClassificationTree

from .KMeans import cen\_sel\_random

from .Learner import CreateLearner

from .Learner import ReadDataFile

from .Learner import ConvertToDictionary

from .Learner import ConvertToBinary

from .Test import ErrorComp

from .Test import GetData

from .Test import conf\_matrix

**ClassificationTree.py:**

from collections import Counter

from copy import deepcopy

from math import log

class ClassificationTree(dict):

def \_\_init\_\_(self):

super().\_\_init\_\_()

#need to find a way to prevent setting the

#class's evaluation value outside of class

def TreeGrowth( self, data, class\_attr ):

'''

Returns a classification tree. The data parameter should be a list of

records, each record being a dictionary of the

form {attribute name: value}. The class\_attr is the name of the

attribute that is to be the class. The optional attr\_done parameter

should be a set of attribute names.

'''

#things to stop on:

#done- empty data (ERROR)

#done- data is not well formatted or wrong type (ERROR)

#void- attr\_done is not well formatted or wrong type (ERROR)

#done- class\_attr is not a string (ERROR)

#done- class\_attr is not an attribute (ERROR)

if type(class\_attr) is not str:

print("Error: class is not a string")

return "Error"

elif not data:

print("Error: empty data")

return "Error"

elif type(data) is not list:

print("Error: data is not a list of dictionaries")

return "Error"

elif type(data[0]) is not dict:

print("Error: data is not a list of dictionaries")

return "Error"

#acquire attributes from data

attributes = {key for key in data[0].keys()}

for record in data:

if type(record) is not dict:

print("Error: data is not a list of dictionaries")

return "Error"

elif record.keys() != attributes:

print("Error: all records in data do not share attribute set")

return "Error"

if class\_attr not in attributes:

print("Error: class is not an attribute")

return "Error"

#remove class from the attribute set

attributes -= {class\_attr}

#create classification tree

self.clear()

self.update( self.\_\_PrivateTreeGrowth( data, class\_attr, attributes ) )

#return a copy of tree

return deepcopy(self)

def \_\_PrivateTreeGrowth( self, data, class\_attr, attributes ):

#acquire all possible values of class

class\_values = [record[class\_attr] for record in data]

#base cases

if class\_values.count(class\_values[0]) == len(class\_values):

#data is pure

return class\_values[0]

elif not attributes:

#empty attributes

return Counter(class\_values).most\_common()

else:

#get the attribute that produces the best split

best\_split = ClassificationTree.find\_best\_split( data, attributes,

class\_attr )

root = {best\_split:{}} #initialize the tree to return

#acquire all possible values of best split attribute

values = {record[best\_split] for record in data}

for v in values:

sub\_data = \

[record for record in data if record[best\_split] == v]

sub\_attr = set(attributes) #copy attribute set

sub\_attr -= {best\_split} #remove attribute used to split

#get subtrees

root[best\_split][v] = \

self.\_\_PrivateTreeGrowth( sub\_data, class\_attr, sub\_attr )

#return the classification tree

return root

def Classify( self, record ):

if not self:

print("Error: tree not made yet! Use the TreeGrowth function",

"to create a classification tree.")

return "Error"

tree = deepcopy(dict(self))

while True:

attr = list(tree.keys())[0]

x = tree[attr][record[attr]]

if type(x) is not dict:

return x

else:

tree = x

@staticmethod

def find\_best\_split( data, attributes, class\_attr ):

# get the values to calculate entropy

entropies = {}

num\_records = len( data )

rep\_count = 0;

dem\_count = 0;

attr\_count = {}

attr\_vals = [ 'A=T', 'A=F', 'A=T:+', 'A=T:-', 'A=F:+', 'A=F:-' ]

# create a dict for each attribute to hold their values

# needed to calculate its entropy

for attr in attributes:

# initialize a dict for each attribute

attr\_count[attr] = {}

# initialize the values for each attribute

for val in attr\_vals:

attr\_count[attr][val] = 0.0

# grab the values to calculate the entropies

for record in data:

if record[class\_attr] == '0':

rep\_count = rep\_count + 1.0;

if record[class\_attr] == '1':

dem\_count = dem\_count + 1.0;

for attr in attributes:

if record[attr] == '1':

#increment A=T

attr\_count[attr]['A=T'] += 1.0

if record[class\_attr] == '0':

#increment A=T:-

attr\_count[attr]['A=T:-'] += 1.0

if record[class\_attr] == '1':

#increment A=T:+

attr\_count[attr]['A=T:+'] += 1.0

if record[attr] == '0':

#increment A=F

attr\_count[attr]['A=F'] += 1

if record[class\_attr] == '0':

#increment A=F:-

attr\_count[attr]['A=F:-'] += 1.0

if record[class\_attr] == '1':

#increment A=F:+

attr\_count[attr]['A=F:+'] += 1.0

# calculate each attribute's entropy

Eo\_rep = -1 \* (rep\_count/num\_records) \* \

log((rep\_count/num\_records), 2) if rep\_count != 0 else 0

Eo\_dem = -1 \* (dem\_count/num\_records) \* \

log((dem\_count/num\_records), 2) if dem\_count != 0 else 0

Eorig = Eo\_rep + Eo\_dem

for attr in attributes:

# check if all the data goes one way or the other

if attr\_count[attr]['A=T'] != 0 and attr\_count[attr]['A=F'] != 0:

# Attribute = T

# Attribute = T : positive

t = attr\_count[attr]['A=T:+'] / attr\_count[attr]['A=T']

aTp = -1 \* t \* log(t, 2) if t != 0 else 0

# Attribute = T : negative

t = attr\_count[attr]['A=T:-'] / attr\_count[attr]['A=T']

aTn = -1 \* t \* log(t, 2) if t != 0 else 0

E\_aT = aTp + aTn

# Attribute = F

# Attribute = F : positive

t = attr\_count[attr]['A=F:+'] / attr\_count[attr]['A=F']

aFp = -1 \* t \* log(t, 2) if t != 0 else 0

# Attribute = F : negative

t = attr\_count[attr]['A=F:-'] / attr\_count[attr]['A=F']

aFn = -1 \* t \* log(t, 2) if t != 0 else 0

E\_aF = aFp + aFn

# Calculate the entropy of the attribute

tT = attr\_count[attr]['A=T'] / num\_records

tF = attr\_count[attr]['A=F'] / num\_records

entropies[attr] = Eorig - tT \* E\_aT - tF \* E\_aF

else:

# all of the data when one way, so there is no info gain

entropies[attr] = 0

# determine the best of the entropies

maxEntropyVal = -1

maxEntropyAttr = ''

for attr in entropies:

if entropies[attr] > maxEntropyVal:

maxEntropyVal = entropies[attr]

maxEntropyAttr = attr

# return attribute best to split

return maxEntropyAttr

**KMeans.py:**

import sys

import csv

import random

import math

def MaxDisCentriodSelc(bi\_unlabeled):

points = []

counterFurthest = 0

counterC1 = 0 # tracks the hamming distance from centriod 1

for record in bi\_unlabeled: # add one to record

rand\_val = random.randint(0,3.0)

if rand\_val == 3:

points.append(record)

rand\_1 = random.randint(0,len(points))#Obtain first centriod

c1 = points[rand\_1]

#loop through each character in binary string

#and compute the hamming distance

for point in points:

counterC1 = 0

for i in range(0,len(c1)):

if(c1[i] != point[i]):

counterC1 += 1

if counterC1 > counterFurthest:

furthest = point

counterFurthest = counterC1

finalClusters = KMeans(bi\_unlabeled, c1, furthest)

return finalClusters

def cen\_sel\_random(bi\_unlabeled):

#randomly select the 2 centroid from the unlabeled data

rand\_1 = random.randint(0,len(bi\_unlabeled))

rand\_2 = random.randint(0,len(bi\_unlabeled))

# if the two random centroids happen to be equal re select centroid 2

while rand\_1 == rand\_2:

rand\_centroid\_2 = random.randint(0,len(bi\_unlabeled))

# set the centroid equal to the value in the unlabeled list of records

rand\_centroid\_1 = bi\_unlabeled[rand\_1]

rand\_centroid\_2 = bi\_unlabeled[rand\_2]

finalClusters = KMeans(bi\_unlabeled, rand\_centroid\_1, rand\_centroid\_2)

return finalClusters

def KMeans(bi\_unlabeled, rand\_centroid\_1, rand\_centroid\_2):

att\_num = 16 # number of attributes in the data file

cen\_temp\_1 = '' # temp centroid for compares

cen\_temp\_2 = '' # tracks perv. centroid

c1Calc = [] #list of points associated with centroid 1

c2Calc = [] #list of points associated with centroid 2

counter = 0

#loop until centriods stablize

while rand\_centroid\_1 != cen\_temp\_1 and rand\_centroid\_2 != cen\_temp\_2:

c1Calc = [] #Reset c1Calc and c2Calc

c2Calc = []

#Determine which points reside closest to the centriods

for record in bi\_unlabeled:

closest = mindist(rand\_centroid\_1, rand\_centroid\_2, record)

if (closest == 1):

c1Calc.append(record)

else:

c2Calc.append(record)

#Store the previous values of each centriod

cen\_temp\_1 = rand\_centroid\_1

cen\_temp\_2 = rand\_centroid\_2

rand\_centroid\_1 = ''

rand\_centroid\_2 = ''

counter = 0

#Recalculate centroid 1

for val in range(0,att\_num):

#loop through one attribute and calculate the average

for point in c1Calc:

counter += int(point[val])

counter = counter / len(c1Calc)

#Round up or down

if(counter >= .5):

rand\_centroid\_1 = rand\_centroid\_1 + '1'

else:

rand\_centroid\_1 = rand\_centroid\_1 + '0'

#Recalculate centriod 2

counter = 0

for val in range(0,att\_num):

#loop through one attribute and calculate the average

for point in c2Calc:

counter += int(point[val])

counter = counter / len(c2Calc)

#Round up or down

if(counter >= .5):

rand\_centroid\_2 = rand\_centroid\_2 +'1'

else:

rand\_centroid\_2 = rand\_centroid\_2 + '0'

finalClusters = \

convertOutput(rand\_centroid\_1,rand\_centroid\_2,c1Calc,c2Calc)

return finalClusters

def mindist(c1,c2,point):

#print("c1: ", c1, " c2: ",c2, " point: ", point)

counterC1 = 0 # tracks the hamming distance from centriod 1

counterC2 = 0 # tracks the hamming distance from centroid 2

#loop through each character in binary string

#and compute the hamming distance

for i in range(0,len(c1)):

if(c1[i] != point[i]):

counterC1 += 1

if(c2[i] != point[i]):

counterC2 += 1

if (counterC1 < counterC2):

return 1

else:

return 2

def convertOutput(c1,c2,c1Calc,c2Calc):

final\_c1 = {}

final\_c2 = {}

cen1\_final\_list = []

cen2\_final\_list =[]

cen1\_final\_list.append(c1)

cen2\_final\_list.append(c2)

for i in range(0,len(c1Calc)):

cen1\_final\_list.append(c1Calc[i])

for i in range(0,len(c2Calc)):

cen2\_final\_list.append(c2Calc[i])

return cen1\_final\_list, cen2\_final\_list

**Learner.py:**

from . import ClassificationTree

from . import cen\_sel\_random

import random

def ConvertToBinary( record ):

for i in range( 0, len(record)):

if record[i] == 'y' or record[i] == 'democrat':

record[i] = '1'

elif record[i] == 'n' or record[i] == 'republican':

record[i] = '0'

else:

record[i] = ''

return ''.join(record)

def ConvertToDictionary( binstring, attributes ):

count = len(binstring)

return\_dict = dict()

for i in range(0, count):

return\_dict[attributes[i]] = binstring[i]

return return\_dict

def ReadDataFile( filename ):

labeled = [] #label records in csv file

unlabeled = [] #unlabel records in csv file

with open( filename ) as csvFile:

attributes = csvFile.readline()[:-1].split(',')

for Record in csvFile: # add one to record

record = Record.strip()

if record:

rand\_val = random.randint(0,1) # select either zero or one

#if val is one remove the class from each record

if rand\_val == 1:

new\_record = record.split(',')

new\_record = new\_record[1:]

new\_record = ConvertToBinary(new\_record)

#append new\_record to the unlabeled list

unlabeled.append(new\_record)

else:

new\_record = record.split(',')

new\_record = ConvertToBinary(new\_record)

labeled.append(new\_record)

return attributes, labeled, unlabeled

def CreateLearner( attributes, labeled, unlabeled ):

#convert labeled data into dictionary format

labeled\_dicts = list()

for record in labeled:

labeled\_dicts.append( ConvertToDictionary( record, attributes ) )

#classify labeled data

tree = ClassificationTree()

tree.TreeGrowth( labeled\_dicts, attributes[0] )

#cluster unlabeled data

clusters = cen\_sel\_random(unlabeled)

#classify unlabeled data

centroids = list()

data = list(labeled)

for cluster in clusters:

#get centroids from clusters

centroid = ConvertToDictionary( cluster.pop(0), attributes[1:] )

#classify centroid

clss = tree.Classify( centroid )

#apply class to all records in a cluster

for record in cluster:

data.append( clss + record )

#convert labeled data into dictionary format

data\_dicts = list()

for record in data:

data\_dicts.append( ConvertToDictionary( record, attributes ) )

#regrow tree

tree.TreeGrowth( data\_dicts, attributes[0] )

return tree

**Test.py:**

import sys

from . import ConvertToBinary

def GetData( filename ):

data = [] #label records in csv file

with open( filename ) as csvFile:

attributes = csvFile.readline()[:-1].split(',')

for Record in csvFile: # add one to record

record = Record.strip()

if record:

record = record.split(',')

record = ConvertToBinary(record)

data.append(record)

return attributes, data

def ErrorComp(data, classes):

counter = 0

for i in range(0, len(data)):

if data[i][0] != classes[i]:

counter += 1

print("Percent Error For Run: ", counter/len(data))

def conf\_matrix(data,classes):

correct\_demo = 0

correct\_rep = 0

incorrect\_demo = 0

incorrect\_rep = 0

for i in range(0, len(data)):

if data[i][0]=='1' and classes[i] == '0':

incorrect\_demo += 1

elif data[i][0]=='0' and classes[i] == '1':

incorrect\_rep += 1

elif data[i][0]=='1' and classes[i] == '1':

correct\_demo += 1

elif data[i][0]=='0' and classes[i] == '0':

correct\_rep += 1

# prints the confusion matrix

print(" R D")

print(" ----------------")

print("R | ",

correct\_rep,

" ",

incorrect\_rep)

print("D | ",

incorrect\_demo,

" ",

correct\_demo)

print("Predicted (X) Actual (Y)")

**The Conclusions**

The testing algorithm was run 25 times and produced error rates in the range of 5% to 9.85% with a mean error rate of 7.02%. Our hybrid model for semi-supervised learning achieved a high accuracy rate when tested on the data it was built on. This is to be expected as the model was fit for that data. A more accurate test would have been to train the model on different data than it was tested on. In later installments of the software the data will be divided first into training and test data, and the model will be built using the training data and evaluated using the test data. The results from our tests are a better indication of our ability to merge the classification tree and the clustering algorithm together. That being said, we were able to merge the two models together with relatively little error.

**References**

1. Roach, Christopher. "Building Decision Trees in Python." *- O'Reilly Media*. N.p., 09 Feb. 2006. Web. <http://www.onlamp.com/pub/a/python/2006/02/09/ai\_decision\_trees.html>.
2. Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. "Decision Tree Induciton." *Introduction to Data Mining*. Boston: Pearson Addison Wesley, 2005. N. pag. Print.
3. Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. "K-means." *Introduction to Data Mining*. Boston: Pearson Addison Wesley, 2005. N. pag. Print.