CS5820 AI - Homework 2

Cody Hanson
University of Colorado Colorado Springs
1420 Austin Bluffs Pkwy
Colorado Springs, Colorado USA 80918
chanson@uccs.edu

Abstract

In this work, a 'decision tree' classifier was implemented and tested. Large datasets that included continuous and missing values were used for classifying discrete target classes. A hybrid of the ID3 and C4.5 algorithms was chosen to be implemented using the Python programming language. Statistics for precision, recall, and f-measure were collected over a number of test runs.

Introduction

For this assignment I have implemented a decision tree classifier that has characteristics of ID3 and C4.5. I have chosen the Python programming language, mainly for its ease of use and expressiveness. In this paper, noteable aspects of the design and implementation of the decision tree classifier will be discussed, as well as the datasets users, and the experimental results. Finally, a future work section mentions areas for improvement. Application code and output are included in the appendices.

Datasets

To test my decision tree implementation, I used two datasets. One which we will refer to as 'AD', which classifies images on the internet as either 'ad' or 'nonad' (as in advertisement). This dataset is from Nicholas Kushmerick, contains 3279 instances, and each instance has 1558 attributes. AD can be found here: http://archive.ics.uci.edu/ml/datasets/Internet+Advertisements.

The other dataset is 'ADULT', which contains demographic information about people around the world, and is used to classify them based on income (greater than 50K dollars, or less than or equal to 50K dollars). There are 48842 instances, and each instance contains 14 attributes. The dataset was donated by Ronny Kohavi and Barry Becker, and can be found here: http://archive.ics.uci.edu/ml/datasets/Adult.

Both datasets contain continuous attributes, as well as a number of missing values.

Decision Tree Creation

The algorithm's that I have chosen to model my inducer on are ID3 and C4.5. These are designed to handle classification, for discrete target classes. In the case of C4.5, it also has techniques to handle missing and continuous values. A key factor in my decision was ease of implementation.

My main routine is called 'inducetree' which is a recursive function that computes the information gain of the set of cases it has, splits on the highest gain and then recurses to the subtrees.

For the minimum set size base case, (the size of partition under which we choose the most common value for the leaf node), I chose 30. This was somewhat arbitrary, and it would be interesting to see how this constant affects precision and recall.

Missing Values

Often data has missing or incomplete attributes. C4.5 was in part created to improve ID3's ability to handle these missing values. By convention, a missing value is denoted by a '?' symbol.

There are three considerations for handling missing attributes. The first is how to handle them while computing information gain for an attribute. I have handled this by following the C4.5 method of simply not including them in the computations. The next consideration is determining how to split a node on an attribute that has cases with missing values. The final consideration is how to classify a new instance that has a missing value.

I have implemented part of a missing value strategy. I was able to easily exclude missing values from information gain computations. Then, when splitting a node on an attribute, for cases that have an unknown value for the split attribute, I duplicate that case and add it to each outgoing edge, with the missing value replaced by the value for that particular partition. I have not implemented any weighting of these cases, however, and this would be a good improvement to make.

Continuous Valued Attributes

In order to handle attributes with continuous values, I take a simple approach by splitting the value on the mean, into a 'high' and 'low'. One assumption that this approach makes is that the training data set are randomly distributed. The implementation approach that I took was to preprocess the training data and rewrite the continuous valued attributes before the data is sent to the decision tree inducing routine.

Then, when applying rules to new examples that needed to match a continuous value, I had to implement special code which would parse a rule like '¡=1000' into an operation ('¡=') and a value ('1000'), and compare it to the value of the attribute for the case. Regular expressions proved helpful for this operation.

One downside of this simple approach is that it doesn't take into account continuous valued attributes that have more than two 'regions'. For instance, if 'Age' is split on '¿45' and '¡=45', this could potentially not take into account that the '¡=45' range significantly changes '¡=18', or vice versa for the upper ranges.

Pruning

C4.5 uses a postpruning method, which prunes back a fully grown tree. This is in contrast to a pre-pruning method which has to determine when to stop growing the tree to begin with. The first approach I took to postpruning was to look for subtrees which all had the same class in their leaf nodes, and then to replace the entire subtree with the leaf node value. This technique has the benefit of reducing the size and complexity of the tree, and the rules generated from the tree, but it doesn't necessarily address the issue of overfitting like an error-based pruning method would. Given more time, I would have attempted to implement error based pruning

Generating Rules

Since a tree can be sometimes difficult to process when it becomes large, it can be convenient to have an alternate view of the classifier data. A common approach is to convert each path from root to leaf in the tree into a rule, which is essentially a series of AND clauses where each term is a node in the decision tree. I implemented a routine which traverses a tree in a depth first manner, compiling a list of rules as it goes.

Classifying New Data

To simplify implementation, I decided to use the rules list data structure to classify new cases rather than traverse a tree. While this approach makes the code simpler, it has the downside of not taking advantage of the properties of traversing the decision tree. Many more comparisons must be made for each case, and many rules which will not apply to a case will need to be checked every time.

Results

Each dataset was randomly divided into training and test sets for 10 runs, 2/3 of the cases for training, and 1/3 for the test set. Values for precision, recall and f-measure were collected for each run and averaged to produce a final score for how the classifier performed for a given dataset. Results given are averages over all runs, to 4 significant figures.

Ten Run Average for Decision Tree Performance

Dataset	Precision	Recall	F-Measure
ADULT	0.8439	0.8513	0.8476
AD	0.8672	0.5949	0.7039

We can see from the F-Measure value, that my algorithm performed better on the ADULT dataset, mostly because of the higher average Recall. Precision for both datasets was mostly the same, although a bit higher for AD.

Future Work

The performance for large datasets could be improved by parallellizeing some of the workload across multiple threads or machines. All code is currently single threaded for simplicity. There are also cases in the implementation where certain values are computed more often than they need to be, and could instead be memoized to increase performance.

The algorithms as implemented also make heavy use of recursion. For large datasets, I found that this can be problematic due to machine limits on recursion depth. In order to handle larger sets, I had to configure my Python environment to allow more than the default 1000 recursive call limit. By converting the code to using standard iteration and loops, this could be mitigated, although at the likely cost of code clarity and brevity.

One other thing that I mentioned earlier is the constant for minimum set size. While I chose 30, it would be interesting to see how the success of the algorithm changes as this value increases or decreases.

Appendix A - Implementation Files

C45.py

```
#! /usr/bin/env python

#This module implements the C4.5 Algorithm for a decision tree classifier
from collections import defaultdict
import math
import re
from numpy import mean

MINIMUM_SET_SIZE = 30

#super quick implementation of a tree.
Tree = lambda: defaultdict(Tree)


#recursive function to induce a decision tree from a set of training case data.
def induce_tree(schema, s):
    #first check for base cases
    if len(schema) is 0 or set_is_small(s):
        #if schema length is zero, it means that there is only one attribute left.
        #and we can't parition anymore.
        #print "Small set base case for {0} {1}".format(schema, s)
        #set of cases is below the minimum size, take the most common class as a leaf node
        return most_common_class(s)
    elif are_in_same_class(s):
        #base case! return a leaf node.
        #print "All same class base case for {0} {1}".format(schema, s)
        return get_class(s[0])
    else:
        #find the information gain of each attribute
        #and split on the node that has the most information gain
        gains = find_information_gains(s)
        split_node_index = max_gain_index(gains)
        #print "Splitting on attr {0} index:{1}".format(schema[split_node_index],
            split_node_index)
        tree = Tree()
        partitions = partition_on_attribute(split_node_index, s)
        for key in partitions.keys():
            tree[schema[split_node_index]][key] = \
                induce_tree(strip_schema_by_index(split_node_index, schema),
                    strip_partition_by_index(split_node_index, partitions[key]))
        return tree


#removes the attribute at 'index' from all tuples. returns list of stripped tuples.
def strip_partition_by_index(index,s):
    stripped = []
    for a_tuple in s:
        tmp = list(a_tuple)
        del tmp[index]
        stripped.append(tuple(tmp))
    return stripped


def strip_schema_by_index(index, schema):
    tmp = list(schema)
    del tmp[index]
    return tuple(tmp)


def find_information_gains(s):
    num_attributes = len(s[0]) − 1 #don't want to include the class
    gains = []
    for i in range(0, num_attributes):
        probabilities = compute_probabilities(i, s)
        gains.append(information_gain(probabilities))
    return gains


#for a given set of probabilities, computes the information gain
def information_gain(probabilities):
    ig = 0
    for p in probabilities:
        ig += - p * math.log(p, 2)
    return ig



def attribute_is_continuous(index, cases):
    continuous = True
    #match int, decimal or unknown value
    continuous_pattern = re.compile('^([\d\.]+|\?)$')
    vals = defaultdict(int)
    for case in cases:
        if len(case) < index:
            pass
       m = re.match(continuous_pattern, case[index])
        if m is None:
            continuous = False
            break
        else:
            vals[m.group(0)] += 1

    #special code to handle the case where there is just '0' and '1'
    if len(vals) == 2 and '0' in vals.keys() and '1' in vals.keys():
        #just a binary column! not continuous
        continuous = False

    return continuous


#this function takes a list of training cases,
#and for each continuous attribute, discretizes it into ranges.
#this then returns the processed set.
#this function should be ran before the training set is passed into induce_tree
def preprocess_continuous_attrs(schema, cases):
    print 'Preprocessing␣data␣to␣get␣rid␣of␣continuous␣attributes...'
    #iterate over each attribute.
    modified_cases = cases
    for i in range(0, len(modified_cases[0]) - 1):
        #print 'Processing {0}'.format(schema[i])
        if not attribute_is_continuous(i, modified_cases):
            #don't need to process this attribute
            continue

        values = [float(case[i]) for case in modified_cases if case[i] is not '?']
        avg_val = mean(values)
        for j in range(0, len(modified_cases)):
            if modified_cases[j][i] == '?':
                #missing attribute, for this case leave it as is.
                continue
           tmp = list(modified_cases[j])
            number_to_compare = float(modified_cases[j][i])
            if number_to_compare <= avg_val:
                tmp[i] = '<=' + str(avg_val)
            else:
                tmp[i] = '>' + str(avg_val)
            modified_cases[j] = tuple(tmp)
    return modified_cases


def compute_probabilities(index, cases):
    probabilities = []
    total = 0.0
    attr_values = defaultdict(int)

    for case in cases:
        #skip missing values
        if case[index] is '?':
            continue
        total += 1.0
        attr_values[case[index]] += 1
    for attr_name in attr_values.keys():
        probabilities.append(attr_values[attr_name]/total)
    return probabilities


#returns the index of the highest gain value
def max_gain_index(gains):
    max_index = 0
    max_val = 0
    for i in range(0,len(gains)):
        if gains[i] > max_val:
            max_val = gains[i]
            max_index = i
    return max_index


#partitions the set based on values of the attribute at index 'split_index'.
#a partition will have the same values of the attribute at index
def partition_on_attribute(split_index, s):
    partitions = defaultdict(list)
    missing = []
    for case in s:
        if case[split_index] == '?':
            missing.append(case)
        else:
            partitions[case[split_index]].append(case)

    #now process the missing cases
    #each case has a missing value for this attribute.
    #we put it in each partition, with the attribute set to the
    #value for that partition
    for case in missing:
        tmp = list(case) #so we can modify the tuple
        for key in partitions.keys():
            tmp[split_index] = key
            updated_case = tuple(tmp)
            partitions[key].append(updated_case)

    return partitions


#in a case tuple, the classification is the last element
def get_class(case):
    return case[-1]


#checks to see if all cases are in the same class.
def are_in_same_class(cases):
    classes = {}
    for case in cases:
        classes[get_class(case)] = True
    return len(classes) == 1


#checks to see if the set of cases is below some minimum size
def set_is_small(cases):
    return len(cases) < MINIMUM_SET_SIZE


#returns the most common class for the set of cases
def most_common_class(cases):
    classes = defaultdict(int)
    for case in cases:
        classes[get_class(case)] += 1
    most_common = sorted(classes, key=classes.get)[-1]
    return most_common


#https://en.wikipedia.org/wiki/Autovivification
def convert_nested_dd(dd):
    '''Converts a nested defaultdict back into a native dictionary.
    '''
    return {k:convert_nested_dd(v) for k,v in dd.items()} if isinstance(dd, defaultdict) else
         dd


def prune(tree):
    #fist check to see if tree is a simple leaf
    if type(tree) is not dict:
        return tree

    #if all leaf nodes are the same, prune subtree
    leaf_value = all_leaves_same(tree)
    if leaf_value:
        return leaf_value
    else:
        #prune all subtrees
        for k,v in tree.iteritems():
            tree[k] = prune(v)
        leaf_value2 = all_leaves_same(tree)
        if leaf_value2 is None:
            return tree #can't be further pruned
        else:
            return leaf_value2 #remove this whole subtree


#tree has all leaves that are the same.
#if so, return the value of the leaves. otherwise None
def all_leaves_same(tree):
    values = defaultdict(int)
    for k,v in tree.iteritems():
        if (type(v) is dict):
            #nested.
            val = all_leaves_same(v)
            if val is None:
                return None
            else:
                values[val] = 1
        else:
            values[v] = 1
    if len(values) == 1:
        return values.keys()[0] #should only be one key, the leaf value.
    else:
        #not all leaves in the subtree were the same.
        return None
```

rules.py

```
#Module which has functions that can format a decision tree as a set of rules
#as well as print rules in an easy to read human format


#this does a depth first traversal of the tree, to build up a list of rules.
#a rule is basically an AND of each state going down a path in the tree,
#culminating in a 'leaf' node which contains the classification for that rule
def build_rules_list(schema, tree, prefix):
    lst = []
    for k, v in tree.iteritems():
        if type(v) is dict:
            #recurse
            for k2, v2 in v.iteritems():
                if type(v2) is dict:
                    #clone the prefix so we can modify it
                    new_prefix = list(prefix)
                    new_prefix.append(tuple([k,k2]))
                    #merge lists
                    lst = lst + build_rules_list(schema, v2, new_prefix)
                else:
                    lst.append(prefix + [tuple([k, k2]), v2])
    return lst


#rules is a list of decision tree rules, which themselves are tuples
#that are of the form:
#the ordering of rules is not important,
#since each item in the rule is combined with the name of the object in the schema.
#Schema: (Attribute 1, Atribute 2, ...)
#Rule: ({attribute_1:'attr1 va'}, {attribute_2:'attr2 val'}, ..., CLASSIFICATION)
def print_rules_list(rules):
    pass
```

classify.py

```
import re

continuous_val = re.compile('^(>|<=)([\d\.]+)$')

#given a decision tree and an example, return the determined classification for the unknown
    example.

def is_continuous(rule_clause):
    m = re.match(continuous_val, rule_clause[1])
    if m is None:
        return None
    comparison = m.group(1)
    value = m.group(2)
    return comparison, value


def rule_classify(schema, rules, case):
    rule_counter = 0
    rule_count = len(rules)
    for rule in rules:
        #print "trying rule {0} of {1}".format(rule_counter,rule_count)
        rule_counter += 1
        rule_match = True
        for i in range(0, len(rule) - 1):
            idx = schema.index(rule[i][0]) #index of this attribute
            #we need to check if this rule is a continuous value.
            #and handle it differently than if it was discreet.
            cv = is_continuous(rule[i])
            if cv is not None:
                #we need to do the compare a special way
                if cv[0] == '>' and case[idx] <= cv[1]:
                    #this rule doesn't match
                    rule_match = False
                    break
                elif cv[0] == '<=' and case[idx] > cv[1]:
                    #this rule doesn't match
                    rule_match = False
                    break
            #or test if the case's attribute matches the value in the rule (non continuous
                vals).
            elif case[idx] != rule[i][1]:
                #this rule doesn't match
                rule_match = False
                break
        if rule_match:
            #found a match, return the classification deemed by the rule
            #print "Found a match!"
            return rule[-1]
    return None
```

run.py - Main Program

```
#! /usr/bin/env python

#python libs
import sys
import json
import random
from numpy import mean

#my libs
import C45
import rules
import classify

#some datasets require deep recursion. change Python's default recursion depth.
sys.setrecursionlimit(10000)

def randomly_divide(data):
    training_set = []
    test_set = []
    for case in data:
        if random.randint(1, 3) in [1, 2]:
            training_set.append(case)
        else:
            test_set.append(case)
    return training_set, test_set


def test_run(positive_classification, schema, data):
    print "Randomly dividing Data into training and test..."
    training_set, test_set = randomly_divide(data)
    print "processing training data to partition continuous attributes..."
    processed_training_tuples = C45.preprocess_continuous_attrs(schema, training_set)
    print "inducing decision tree..."
    tree = C45.induce_tree(schema, processed_training_tuples)
    tree = C45.convert_nested_dd(tree)
    print "Post pruning tree..."
    pruned_tree = C45.prune(tree)

    #this code snippet can dump tree json to a file
    with open('tree.json', 'w') as outfile:
        json.dump(pruned_tree, outfile)

    print "Generating rules list..."
    rules_list = rules.build_rules_list(schema, pruned_tree, [])
    print "Using rules to classify test data..."
    print "Number of Rules:{0}".format(len(rules_list))
    #we are computing stats based on the criteria of 'correctly identifying <=50K cases'
    #this is to allow us to compute precision and recall, and f-measure
    #correctly guessed <=50k
    true_positive_count = 0
    #incorrectly guessed <=50k
    false_positive_count = 0
    #correctly guessed  >50k
    true_negative_count = 0
    #incorrectly guessed < 50k
    false_negative_count = 0
    total_pos = 0
    total_neg = 0
    total_count = 0
    num_tests = len(test_set)
    for case in test_set:
        total_count += 1
        #print "Classifying {0} of {1} test cases.".format(total_count, num_tests)
        case_classification = case[-1]
        result = classify.rule_classify(schema, rules_list, case)
        if result == case_classification:
            if result == positive_classification:
                true_positive_count += 1
                total_pos += 1
            else:
                true_negative_count += 1
                total_neg += 1
        else:
            if result == positive_classification:
                false_positive_count += 1
                total_neg += 1
            else:
                false_negative_count += 1
                total_pos += 1

    precision, recall, fmeasure = compute_stats(total_count,
                                                total_pos,
                                                total_neg,
                                                true_positive_count,
                                                true_negative_count,
                                                false_positive_count,
                                                false_negative_count)
    print "Precision: {0}, Recall:{1}, F-measure:{2}".format(precision, recall, fmeasure)
    return precision, recall, fmeasure


def compute_stats(total, totalpos, totalneg, tpos, tneg, fpos, fneg):
    precision = tpos / float(tpos + fpos)
    recall = tpos / float(totalpos)
    fmeasure = 2 * (precision * recall) / (precision + recall)
    return precision, recall, fmeasure


##############################
```

```
#MAIN
##############################

seed = random.randint(1000, 10000)
random.seed(seed)
print "Using random seed:{0}".format(seed)

#command line args
schema_file = sys.argv[1]
dataset_file = sys.argv[2]

schema = tuple([raw_line.strip().split(', ') for raw_line in open(schema_file, 'r')][0])
positive_classification = schema[-1] #last item.
#now remove the last item from the schema list.
schema = schema[0:len(schema) - 1]

#read in the dataset, stripping newlines from the end. split each record on a comma, into a
    tuple.
data_tuples = []
for line in [raw_line.strip() for raw_line in open(dataset_file, 'r')]:
    tokens = line.split(',')
    for i in range(0, len(tokens)):
        tokens[i] = tokens[i].lstrip()
    data_tuples.append(tuple(tokens))

#now data_tuples contains all of our cases, which we will divide randomly into 3 different
    segments:
#2/3 for training, 1/3 for testing. We'll do this 10 times and compute statistics.

precisions = []
recalls = []
fmeasures = []
for run in range(1, 10):
    print "
        ------------------------------------------------------------------------------------
        "
    print "Beginning Test run: {0}".format(run)
    results = []
    p, r, f = test_run(positive_classification, schema, data_tuples)
    precisions.append(p)
    recalls.append(r)
    fmeasures.append(f)

print "
    ------------------------------------------------------------------------------------"
print "Average precision: {0}".format(mean(precisions))
print "Average recall: {0}".format(mean(recalls))
print "Average fmeasure: {0}".format(mean(fmeasures))
```

Appendix B - Test run output

Adult Dataset

```
Using random seed:6336
--------------------------------------------------------------------------------
Beginning Test run: 1
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:2260
Precision: 0.84619019492, Recall:0.852230814991, F-measure:0.849199762893
--------------------------------------------------------------------------------
Beginning Test run: 2
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:1923
Precision: 0.83580495907, Recall:0.855910581946, F-measure:0.845738295318
--------------------------------------------------------------------------------
Beginning Test run: 3
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:1743
Precision: 0.843280977312, Recall:0.861933642526, F-measure:0.852505292872
--------------------------------------------------------------------------------
Beginning Test run: 4
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:2160
Precision: 0.849020082265, Recall:0.838771363691, F-measure:0.843864606505
--------------------------------------------------------------------------------
Beginning Test run: 5
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:2130
Precision: 0.850282825851, Recall:0.852745926373, F-measure:0.851512594914
--------------------------------------------------------------------------------
Beginning Test run: 6
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
```

```
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:1930
Precision: 0.842402826855, Recall:0.854276158624, F-measure:0.848297948049
------------------------------------------------------------------------------
Beginning Test run: 7
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:2322
Precision: 0.844537313433, Recall:0.842625685013, F-measure:0.843580416244
------------------------------------------------------------------------------
Beginning Test run: 8
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:2540
Precision: 0.849400538292, Recall:0.841881896447, F-measure:0.845624505207
------------------------------------------------------------------------------
Beginning Test run: 9
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:2015
Precision: 0.834566161794, Recall:0.861471336808, F-measure:0.847805343511
------------------------------------------------------------------------------
Average precision: 0.843942875533
Average recall: 0.851316378491
Average fmeasure: 0.847569862835
```

Ad Dataset

```
Using random seed:6856
------------------------------------------------------------------------------
Beginning Test run: 1
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:475
Precision: 0.92380952381, Recall:0.573964497041, F-measure:0.70802919708
------------------------------------------------------------------------------
Beginning Test run: 2
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
```

```
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:367
Precision: 0.936842105263, Recall:0.574193548387, F-measure:0.712
```

```
Beginning Test run: 3
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:331
Precision: 0.891089108911, Recall:0.584415584416, F-measure:0.705882352941
```

```
Beginning Test run: 4
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:471
Precision: 0.664, Recall:0.601449275362, F-measure:0.631178707224
```

```
Beginning Test run: 5
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:457
Precision: 0.88785046729, Recall:0.633333333333, F-measure:0.739299610895
```

```
Beginning Test run: 6
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:394
Precision: 0.798076923077, Recall:0.549668874172, F-measure:0.650980392157
```

```
Beginning Test run: 7
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:394
Precision: 0.858407079646, Recall:0.591463414634, F-measure:0.70036101083
```

```
Beginning Test run: 8
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
```

```
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:381
Precision: 0.938053097345, Recall:0.619883040936, F-measure:0.746478873239
--------------------------------------------------------------------------------
Beginning Test run: 9
Randomly dividing Data into training and test...
processing training data to partition continuous attributes...
Preprocessing data to get rid of continuous attributes...
inducing decision tree...
Post pruning tree...
Generating rules list...
Using rules to classify test data...
Number of Rules:351
Precision: 0.906779661017, Recall:0.625730994152, F-measure:0.740484429066
--------------------------------------------------------------------------------
Average precision: 0.867211996262
Average recall: 0.594900284715
Average fmeasure: 0.703854952604
```