The Interdisciplinary Center, Hertzlia

Machine Learning

Winter 2011

Problem Set 1

Due: Nov. 20 2011

Send your answers (word document, source files, text outputs) in a single .rar file to: iassayag@post.idc.ac.il

Question 1 (10 points):

Draw decision trees to represent the following Boolean functions:

a) (A ^ ~B) V C
b) A NAND B
c) (A NAND B) NAND (A NAND B)
d) (A NAND B) NAND (C NAND D)

Notes:

^	stands for the Boolean "and" function
~	stands for the Boolean "not" operator
V	stands for the Boolean "and" function

Question 2 (30 points):

The following table contains some example of people tested for Hepatitis, and their symptoms:

Example	**Malaise**	**Fever**	**Fatigue**	**Enlarged Liver**	**Hepatitis**
1	0	0	0	0	0
2	0	0	1	0	0
3	1	1	0	1	0
4	1	0	0	1	1
5	0	1	1	0	1
6	0	0	1	1	1
7	0	0	0	1	1
8	1	1	0	0	1
A	0	1	1	1	?
B	1	0	0	0	?
C	1	0	1	0	?

a) Considering only examples 1 to 8, what is the entropy of this problem?

b) Which attribute would ID3 choose at the root of the decision tree? Show all computations of information gain for each attribute.

c) Build the ID3 decision tree to classify this problem.

d) Classify cases A, B and C using the decision tree you built.

Question 3 (60 points):

Random Forests have been introduced in 2001 by Leo Breiman, and is one of the best general-purpose Machine Learning algorithms to this day. Random Forests consist of a (typically large) set of random classification trees. Classification is performed by feeding the instance to classify to each tree, and then performing a vote between the different trees to reach a final classification. The original tree used in Random Forests chooses, at each node, to split on the feature with the highest Information Gain (like C4.5), from a randomly selected subset of features. Continuous features are handled much like the C4.5 algorithm.

> *A few words about Leo Breiman - he was one of the most prominent researchers in Machine Learning, and passed away in 2005 after having published an innumerable amount of papers. His contribution consists, among others, of some of the most important algorithms in Machine Learning, such as: CART, Bagging, and Random Forests, just to name a few. You can find more information about his work in a paper written by Adele Cutler, who also worked with him on Random Forests.*

In this exercise you will write a simplified version of a Random Tree. At each node, you will randomly choose a feature to split on. Continuous attributes will also be handled using a random algorithm, by randomly selecting a split value based on all the values found in the training set (and proportional to their distribution).

You are given a database with features which are thought to be indicative of liver disorders that might arise from excessive alcohol consumption. Our purpose is to learn from this database how to identify, using the 6 given features, whether or not the individual suffers from liver disorder.
The first 5 features are the results of blood tests, the 6th feature is the number of beers drunk per day.
The last field in the database indicates the label (1,0), which stands for whether or not the person suffers from liver disorder.

You will write a simplified Random Tree as described above, load the data, train the tree, and test your results.

What should you do?

a) Create a class named *Instance* with the following properties:

```
/**
 * Represents an instance which consists of a set of Features and a Label.
 * Features and Labels are all of type double.
 * A missing Label is indicated by the value -1.
 */
public class Instance
{
        /**
         * Loads the instances from the given CSV file.
         * The first line represents the names of the Features.
         * Each line thereafter represents a separate instance.
         * Each field in each line represents a Feature,
         * except the last field which is the Label.
         * Following is an example:
         *
         * Feature1,Feature2,Feature3,Label
         * 1,2,3,0
         * 2,4,6,1
         * 3,6,9,-1
         *
         * In this example we have:
         * 3 instances
         * 3 Features
         * 2 classes (0, 1)
         * The last instance has an empty label
         */
        public static List<Instance> loadInstances(String path)    {      }
        /**
         * Returns the Feature at the given index.
         */
        public double getFeatureValue(int index)      {      }

        /**
         * Returns the name of the Feature at the given index.
         */
        public String getFeatureName(int index){      }

        /**
         * Returns the Label. A missing Label is indicated by the value -1.
         */
        public double getLabelValue()    {      }

        /**
         * Returns the number of Features in the instance
         * (the Label is not considered a Feature).
         */
        public int numFeatures()  {      }
}
```

b) Create a class named *RandomDT* with the following properties:

```
/**
 * Assumes that all Features are continuous.
 * Missing values are considered as a separate possible feature value.
 * Serves only for classification, not regression.
 * At each node, the Feature used is selected randomly.
 * The splitting value used is also selected randomly.
 */
public class RandomDT
{
        /**
         * Creates a new instance of a Random Decision Tree.
         * Configuration options (if applicable), should be added here.
         */
        public RandomDT()   {      }
        /**
         * Trains the Decision Tree using the received set of Instance objects.
         */
        public void train(List<Instance> trainSet)    {      }
        /**
         * Returns the classification of the given instance.
         * If the Decision Tree has not been trained, either throw an exception
         * or return "-1", which stands for an empty Label value.
         */
        public double classify(Instance instance)     {      }
```

```
            /**
             * Returns a tree-structure string representation of the internal
             * Decision Tree that has been learned by train().
             */
            public void toString()    {      }
      }
```

c) Randomly split the liver-disorders.csv file into 2 disjoint sets:
   a. Roughly 2/3 of the instances should be used for training
   b. The remaining instances should be used for testing

d) Repeat the previous step (randomly splitting the database, training, testing) 100 times, and calculate the following statistics:
   a. Average, Median, Minimum, Maximum and Standard Deviation of the overall accuracy
   b. Average, Median, Minimum, Maximum and Standard Deviation of the Positive Predictive Value (i.e. the percentage of cases that were classified “1” and were indeed 1, or P(class is 1 | classification is 1)   ).
   c. Average, Median, Minimum, Maximum and Standard Deviation of the Negative Predictive Value (i.e. the percentage of cases that were classified “0” and were indeed 0, or P(class is 0 | classification is 0)   ).

e) Choose a single execution of your tree, and produce:
   a. The string representation of the tree by printing the output of the toString() method.
      Following is an example of such a printout:

```
gammagt <= 20
|   sgpt <= 19: 2 (60.0/19.0)
|   sgpt > 19: 1 (80.0/20.0)
gammagt > 20
|   drinks <= 5
|   |   drinks <= 3
|   |   |   alkphos <= 65: 2 (42.0/6.0)
|   |   |   alkphos > 65
|   |   |   |   sgot <= 24: 1 (30.0/11.0)
|   |   |   |   sgot > 24: 2 (23.0/5.0)
|   |   drinks > 3: 2 (41.0/3.0)
|   drinks > 5
|   |   sgpt <= 35: 2 (34.0/10.0)
|   |   sgpt > 35: 1 (35.0/12.0)
```

The output of toString() does NOT have to be *exactly* like that, but this should be the general idea. Just make sure it generates a humanly understandable tree structure.

b. A Confusion Matrix, which shows the number of actual instances vs. predictions. Following is an example matrix:

	Classified as 1	Classified as 0
Actual class is 1	89	56
Actual class is 0	40	160

f) The Prior Probability of the class is the list of probabilities of each of the class values to appear in the data. In other words, the prior probability of the class being "1" corresponds to the recurrence of class "1" in the whole dataset, or to the chance that some randomly drawn instance would be labeled with class 1. In some cases, the prior probability is inferred by making assumptions about the domain, but in many cases (like ours) the prior probability is simply calculated by assuming that the data we have consist of the whole set of available data for this problem, and thus the prior probability of the class is immediately inferred from the data by counting the occurrences of each of the class values. Prior Probability, like any other kind of probability, is measured in values in the range [0,1]. Also, the sum of prior probabilities for all possible class values is always 1.

a. What were the Prior Probabilities of each class value (over the whole database) ?

b. If you had no way of learning, but you knew only the Prior Probabilities – how would you classify a given instance?

c. How would you qualify the results of your tree vs. the alternative guessing you just proposed? Did you reach a result better than that? If so – why?

g) **Bonus question (10 points):** One way of improving the tree could be to use the Information Gain for splitting continuous attributes, instead of randomly selecting a value. Add an option to your tree (configurable via the ctor.) to use Information Gain instead of Random splitting.

a. Did you reach any improvement? Explain why?

Implementation notes

a) You may write it either in Java or C#.
b) All source files and output files should be included in your handout. Provide a short description of how to build and run your application. EVERY FILE NEEDED for compilation and execution should be included in your handout!
c) Focus on the algorithm, this is not a SW engineering course, but points will be deducted for very bad coding, the code must be easily understandable!
d) You do not *have* to follow this exact interface. If you want to alter it – feel free to do so, just make sure it makes sense and document it properly.
e) Whenever in doubt – follow the KISS principle (http://en.wikipedia.org/wiki/KISS_principle)