# Random Decision Trees

Version 1.0

*Project Report*

By Xinwei Li

Dec. 2004

## 1. Introduction

Inductive learning has wide applications in many different fields. Extensive research has been conducted in the past 2 decades to improve the accuracy of the model. There has been significant amount of work in machine learning and data mining to compute “an” (as distinguished from “the”) optimal model. One major interest is on simple hypothesis. According to Occam’s razor, it’s a belief that the simplest hypothesis that fits the data is the best approximation to the optimal solution. But generally it is usually NP-hard to find the simplest hypothesis that fits the data. Thus various heuristics are proposed to search for a closest match. Most of the practical techniques are to use “greedy” to approximate the simplest model. In decision tree learning, typical heuristics include information gain, gini index etc. After the model is completely constructed, it is then further simplified via pruning with different techniques such as MDL-based pruning, reduced error pruning, cost-based pruning etc.

Although no longer NP-hard, most of these techniques still require significant amount of computation as well as storage. Most algorithms require multiple scans of the training data and require data to be held in main memory, which makes the learning inefficient and unscalable for large dataset.

At the same time, it is still questionable if the simplest hypothesis is indeed the closest approximate to the optimal model. Empirical studies have also shown that the simplest tree is not the best. The most evident supporting results are the success of bagging and boosting [3]. Besides improved accuracy, one biggest problem for the family of complicated hypotheses is the learning efficiency.

In this project, I will study another type of complicated trees – random decision trees [1][2], which can provide a much higher efficiency. First an implementation of the algorithms will be given. Both the performance and scalability will be studied on the implementation. Later, some methods will be used to improve the performance of the basic algorithm.

The report is structured as follows. I will present the main idea in random decision trees in Section 2. Some related work will be introduced in Section 3. Section 4 will illustrate the experiments with analysis. Some methods to improve the performance will be shown Section 5. Finally Section 6 will conclude with a discussion about some problems and future work.

## 2. Random Decision Trees

The idea of random decision trees was proposed by Wei Fan in [1][2] in 2003. Although it looks quite different from those classical decision trees, their training and testing share the similar procedure: attribute selection, termination of tree growth, post-pruning and evaluation on new data.

First the structures of N random decision trees are built without the data. For each node of the tree, instead of using any heuristics to select a feature, this method completely randomly chooses a feature from the remaining feature set. A discrete feature can be used only once in a decision path starting from the root of the tree till the current node. Continuous features can be discretized and treated as discrete features. Another approach is to pick a random dividing point (i.e., < and >= the dividing value) each time that this continuous feature is chosen. Since a different dividing point is picked whenever a continuous feature is chosen, it can be used multiple times in a decision path.

After the tree structures are finalized, the dataset is scanned only once to update the statistics of each node in every tree. These statistics are simply to track the number of examples belonging to each class that are "classified" by each node. These statistics are used to compute the posteriori probability.

When there's no significant difference among the child nodes, all these child nodes can be removed and the parent node is made as the leaf node.

To classify an example, the posteriori probabilities output from multiple trees are averaged as the final probability estimate. In order to make a decision, a loss function is required in order to minimize the expected loss when the same example is drawn from the universe repeatedly.

Two parameters must be specified for the random decision trees. One is the tree depth, i.e., when to terminate the growth of the tree. The other is the number of trees. Generally tree depth will be half of the number of features, which will provide the most diversity of the trees. From experiments, it shows that when the number of trees is greater than 10, random decision trees will give a very good result.

For example, we have a training dataset in Table 1

**Table 1 Training Dataset**

| outlook | temperature | humidity | windy | class |
|---|---|---|---|---|
| sunny | 85 | 85 | false | Don't Play |
| sunny | 80 | 90 | true | Don't Play |
| overcast | 83 | 78 | false | Play |
| rain | 70 | 96 | false | Play |
| rain | 68 | 80 | false | Play |
| rain | 65 | 70 | true | Don't Play |
| overcast | 64 | 65 | true | Play |
| sunny | 72 | 95 | false | Don't Play |
| sunny | 69 | 70 | false | Play |
| rain | 75 | 80 | false | Play |
| sunny | 75 | 70 | true | Play |
| overcast | 72 | 90 | true | Play |
| overcast | 81 | 75 | false | Play |
| Rain | 71 | 80 | true | Don't Play |

If we build two random decision trees, they may look like

```
outlook = sunny:
|   temperature <= 71.0 : Play (1.0)
|   temperature > 71.0 : Don't Play (4.0/1.0)
outlook = overcast:
|   windy = true: Play (2.0)
|   windy = false: Play (2.0)
outlook = rain:
|   windy = true: Don't Play (2.0)
|   windy = false: Play (3.0)
```

```
outlook = sunny:
|   humidity <= 87.0 : Play (3.0/1.0)
|   humidity > 87.0 : Don't Play (2.0)
outlook = overcast:
|   humidity <= 92.4 : Play (4.0)
|   humidity > 92.4 : Play (0.0)
outlook = rain:
|   humidity <= 90.6 : Play (4.0/2.0)
|   humidity > 90.6 : Play (1.0)
```

The numbers in the parentheses are the number of training examples that are “classified” into the leaf node by this random tree. Now given a new example (outlook=”sunny” temperature=69, humidity=80), the output of the first tree is 100% “play”. The output of the second tree is 75% “play” and 25% “Don’t play”. The average posterior probability is 87.5% “play” and 12.5% “Don’t play”. Thus the result is “play”.

It may look strange that random decision trees can work at the first look. In [1], the author gives an explanation by error-tolerance property of probabilistic decision making. To minimize a given loss function, in fact, it is not necessary to have the true probability as long as the estimated probability is within a tolerable error range of the true value. For a two-class problem, assume that the true probability for x to be class y is 0.9. An inaccurate estimate of posterior probability as 0.51 will have exactly the optimal prediction under 0-1 loss function.

In [2], the author gives a further analysis of the method. The study shows that the actual reason for random tree’s superior performance is due to its optimal approximation to each example’s true probability to be a member of a given class. On the reliability curve, the probability output by random tree matches the true probability closely throughout the range (between 0 and 1). However, the best single decision tree (one trained with best splitting criteria such as information gain) matches well only at points close to either 0 or 1. In the range around 0.5, single best decision tree either significantly over-estimates or under-estimates.

## 3. Related Work

One similar method for training decision trees is shown in [4], which is called randomized trees. One important distinction of this random tree method from [4] is that, in randomized trees, a feature subset is randomly generated from the complete feature set; from each feature subset, heuristics from conventional decision tree learners will be used to compute the single best tree such as information gain among others. However, in this method, the random tree uses the complete feature set. The structure of the tree is constructed by randomly picking an untested feature and splitting point, in which no heuristic will be adopted to choose a feature at any given step. The choice of feature at each step is completely stochastic.

Another related work is shown in [3], where the effectiveness of bagging and boosting is studies on decision tree learning. Both bagging and boosting manipulate the training data in order to generate different classifiers. Bagging produces replicate training sets by sampling with replacement from the training instances. Boosting uses all instances at each repetition, but maintains a weight for each instance in the training set that reflects its importance; adjusting the weights causes the learner to focus on different instances and so leads to different classifiers. In either case, the multiple classifiers are then combined by voting to form a composite classifier. In these two methods, the training for each tree is same as classical methods. Heuristics will be adopted for feature selection.

## 4. Experimental Results and Analysis

The random decision tree algorithm is contrary to common believes. The biggest concern is on its actual accuracy. First a lot of experiments will be performed using 0-1 loss functions to evaluate its performance.

The first group of experiments is performed on the datasets included in C4.5 source code. These datasets contains training and testing data. The performance will be compared with C4.5 release 8. The single best tree is constructed by running C4.5 with all the default settings. Since random decision tree may introduce large variation in accuracy, each test will be run 10 times. For each test, up to 30 random trees are constructed. The results are shown in Figure 1

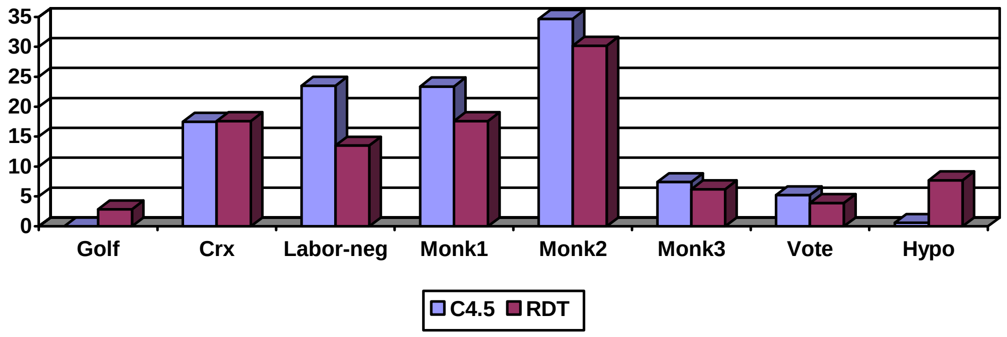

**Figure 1 Performance Evaluation**

In Figure 1, x-axis is the dataset and y-axis is the error rate on testing data. From the comparisons we can see that random decision trees perform better than C4.5 on 5 datasets, while C4.5 performs better on 2 of the datasets. They have the similar performance on one of the datasets. The worse performance on the two datasets may be due to the properties of the datasets. Golf is a very simple dataset with only 14 cases. And there’s no test data available. Here the performance is on training data. C4.5 can always learn well from the training data. But the randomness causes random decision trees sometimes to make errors. Hypo is a dataset with 5 classes and 92.3% of them belong to one big class.

In such a case, random decision trees just assign all the examples into the dominating class.

Another group of experiments is performed to compare random decision trees with C4.5, bagging and boosting implemented in Weka [5]. All the datasets are from UCI Machine Learning Repository [6]. Since there's no splitting of the datasets for training and testing, here I use 10-fold validation to evaluate the performance. The results are shown in Table 1.

**Table 2 10-fold Evaluation**

| | C4.5 | Bagging | Multi Boosting | RDT |
|---|---|---|---|---|
| Balance | 34.52 | 27.2 | 26.72 | 14.2 |
| Ecoli | 15.77 | 15.18 | 16.96 | 31.5 |
| Glass | 34.11 | 26.16 | 20.56 | 32.7 |
| Image | 10.95 | 10 | 9.04 | 12.4 |
| Iris | 4 | 4.67 | 6.67 | 5.3 |
| Yeast | 44.13 | 39.21 | 42.38 | 61.4 |
| Zoo | 7.92 | 5.94 | 3.92 | 2.9 |

From Table 1, we can see that random decision trees perform best on two of the datasets. Boosting performs best on two of the datasets. Bagging performs best on two of the datasets. C4.5 performs best on one of the datasets. But random decision trees perform worst on threes of the datasets. I check the detail about the three datasets and the poor performance may due to the following three reasons:

- Small dataset
- A large number of classes
- Imbalanced distribution of classes

Ecoli is a relatively small dataset with 304 examples. But the number of classes is 8, which is relatively big. Some classes only contain a few examples. In such a case, random decision trees never assign any examples to these classes. Yeast is a relatively large dataset with 1484 examples. But the number of classes is big, which is 10. Similar to ecoli, random decision trees also never assign any examples to small classes. Image is also a relatively small dataset with 210 examples. The number of classes is 7. The distribution of the classes is uniform. In fact, most of the 10-fold test on image gives very good results. But one of them gives a very bad result which makes the overall performance drop down.

The following experiments are performed on the Adult dataset to show the scalability of random decision trees. In the first experiment, the data size of adult is increased by duplicating the dataset. The training time is shown in Figure 2. In the picture, x-axis is the size of the dataset. Y-axis is the running time of training. Here 30 random trees with tree depth 6 will be built. Since every example in the training dataset will only be used once during the training to update the statistics of the leaf nodes, it should be scalable to the data size. From the picture, we can see that the running time increase linearly with the data size.

But since the trees will be held in memory, the memory usage and running time may be sensitive to the depth of the trees. Figure 3 shows the running time of random decision trees with different tree depth on adult dataset. X-axis is the tree depth, while y-axis is the running time. From the picture, we can see the running time increases exponentially with the tree depth. It's the same with the memory usage. When the tree depth is 9, the program terminates by failing to allocate memory for the tree node.

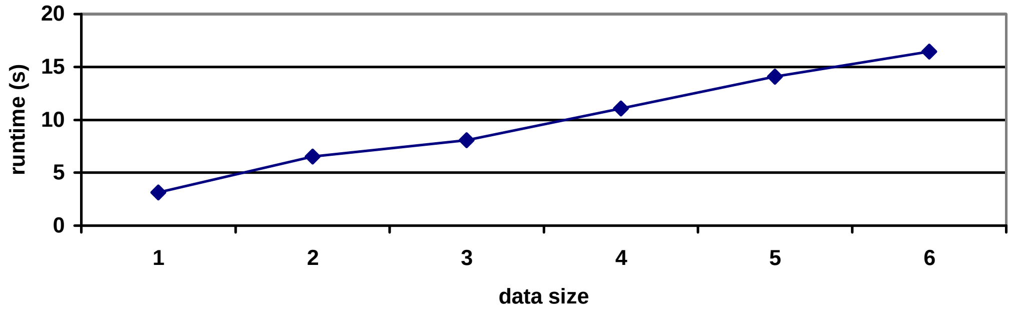

**Figure 2 Scalability on Data Size**

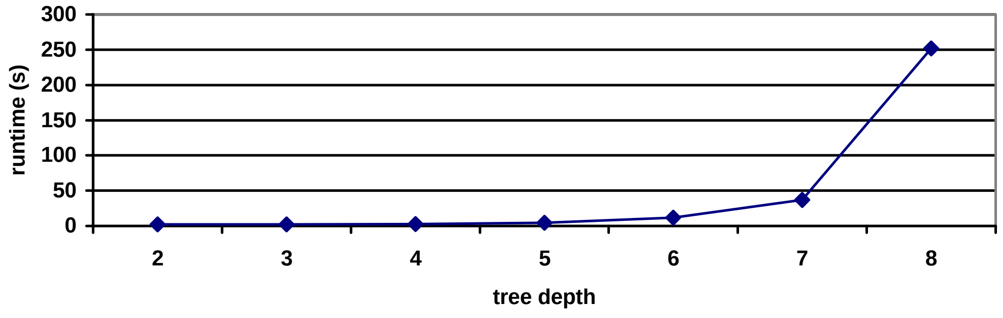

**Figure 3 Scalability on Tree Depth**

## 5. Improvement

From the experiments, we can see that one of the weaknesses about random decision trees is the ability to handle imbalanced dataset with 0-1 loss function. But in most cases, the small class is usually the interesting one. The precision and recall rate for the small class are more important than the overall performance. Instead of using re-sampling to increase the percentage of the small class, a loss function with more cost for misclassifying the small class into the large class may be helpful. The following cost matrix is used on the active chemical compounds dataset.

**Table 3 Cost Matrix**

| | Predict active | Predict not active |
|---|---|---|
| Actual active | 0 | 1 |
| Actual not active | 1 | -40 |

The dataset describes chemical compounds in terms of their structure and activity. Of all the compounds in the data set, only about 2% are active compounds. Under the 0-1 cost function, random decision trees just assign all the examples into the "not active" class and get an overall error rate of 2%. Both the precision and recall for the "active" class are 0.

With the cost matrix in Table 3, the classification will base on the minimal expected cost. Although the overall performance drops down with error rate increasing from 2% to 4.1%, the precision and recall for the "active" class are increased. Precision increases from 0 to 11.7%, while recall increases from 0 to 18%.

Another idea comes from bagging and boosting. For each random decision tree, only a random subset of the training samples will be used to update the statistics of the leaf nodes. After the tree is built, a weight will be assigned to the tree according to its accuracy on the training dataset. This method is applied to the three dataset with worst performance in Table 2. Compared to the original random decision trees, the results are shown in Table 4.

**Table 4 RDT with Bagging and Boosting**

| | Yeast | Image | Glass |
|---|---|---|---|
| Original RDT | 61.4 | 12.4 | 32.7 |
| New RDT | 57.4 | 7.9 | 28.5 |

From the table, we can see that after applying the idea into random decision trees, the performance on these three datasets is improved a little. For the image dataset, the performance is even better than the other three methods in Table 2.

## 6. Conclusions and Future Work

In this project, the algorithm of random decision trees is implemented on Linux based on C4.5 release 8 source code from L. R. Qinlan. Experiments are performed on different datasets with different properties. The performance of random decision trees varies on these datasets. Compared with C4.5, bagging and boosting, random decision trees achieve better results on some datasets and worse results on others. No one is superior in all cases. At the same time random decision trees show a rather good scalability on the training data size.

Some weaknesses of random decision trees are shown in the experiments. One of them is the poor ability to recognize the examples of the small class in an imbalanced dataset. Another problem is the poor scalability on the tree depth, which is proportional to the number of features.

Some methods are used to improve performance. By using cost matrix in imbalanced dataset, the overall accuracy drops down, but the precision and recall of the small class increase a lot. By incorporating bagging and boosting into random decision trees, the accuracy can also be improved a little.

In this implementation, a random threshold is used for continuous feature type when it is selected. In the future, discretization can be used to handle continuous feature type and compare the results with random threshold. The poor scalability on the number of features may be alleviated by avoiding generating empty nodes as mentioned in [2]. More work is needed to improve the performance of random decision trees on imbalanced dataset.

## References:

[1] Fan, W.; Wang, H.; Yu, P.S.; Ma, S.: Is random model better? On its accuracy and efficiency. Data Mining, 2003. ICDM 2003. Third IEEE International Conference on , 19-22 Nov. 2003. pp. 51 – 58

[2] Wei Fan. On the Optimality of Probability Estimation by Random Decision Trees. AAAI 2004: 336-341.

[3] J. R. Quinlan: Boosting, Bagging, and C4.5. AAAI'96, pp 725-730.

[4] Y. Amit and D. Geman. Shape quantization and recognition with randomized trees. Neural Computation, 9(7):1545–1588, 1997.

[5] Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/ml/weka/

[6] UCI Machine Learning Repository. http://www1.ics.uci.edu/~mlearn/MLRepository.html