TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 06/11 pp415–425
Volume 21, Number 4, August 2016

Taiga: Performance Optimization of the C4.5 Decision Tree Construction Algorithm

Yi Yang and Wenguang Chen*

Abstract: Classification is an important machine learning problem, and decision tree construction algorithms are an important class of solutions to this problem. RainForest is a scalable way to implement decision tree construction algorithms. It consists of several algorithms, of which the best one is a hybrid between a traditional recursive implementation and an iterative implementation which uses more memory but involves less write operations. We propose an optimized algorithm inspired by RainForest. By using a more sophisticated switching criterion between the two algorithms, we are able to get a performance gain even when all statistical information fits in memory. Evaluations show that our method can achieve a performance boost of 2.8 times in average than the traditional recursive implementation.

Key words: C4.5; RainForest; decision trees; machine learning; performance optimization

1 Introduction

Decision tree learning is an important class of supervised learning algorithms in the areas of applied statistics, machine learning, and data mining[1,2].

In the most typical scenario, a decision tree classifier is used to deal with *classification problems*. Decision tree learning algorithms are those which are used to construct a decision tree classifier. The terminology in this paper is defined as follows:

- The input is a given data set, called the *training set*. The training set consists of many *cases*.
- Each of the *cases* has a number of *attributes*. The attributes can be categorical (discrete) or continuous. Each case belongs to one of a number of *classes*.
- The output is a *classifier*, which is able to predict the class for any new case.

A decision tree is a rooted tree structure. Each of the non-leaf nodes in a decision tree is a decision node, which represents a criterion to divide the cases into two or more subtrees. Each of the leaf nodes represents a cluster of similar cases, which are assigned to a single class. A decision tree classifier predicts the class for any unseen case, by going down from the root node to a leaf, using the criteria in the nodes to decide which branch to go to.

Many decision tree algorithms were proposed and widely adopted. Among them, the ID3 algorithm and its generalizations[3-5], its successors C4.5[6] and C5.0[7], all designed by Quinlan, and the CART[8] algorithm, CHAID[9], FACT[10], and Sprint[11] are frequently employed.

At first, most decision tree construction algorithms are implemented in an recursive way. While this approach is easy to implement, it does have some performance concerns. Works are also done to propose different frameworks, like RainForest[12], that does not modify the tree-construction logic, but changes the execution order to an iterative fashion. However, these algorithms, even if they change execution order, rarely

• Yi Yang and Wenguang Chen are with Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: ahyangyi@gmail.com.
• Wenguang Chen is also with Technology Innovation Center at Yinzhou, Yangtze Delta Region Institute of Tsinghua University, Yinzhou 315100, China. E-mail: cwg@tsinghua.edu.cn.
∗ To whom correspondence should be addressed.
Manuscript received: 2015-06-04; accepted: 2015-07-11

take architectural elements like cache and prefetching into consideration.

In this paper, we propose Taiga, a cooler variant of RainForest. Building upon a hybrid of the recursive and the iterative approaches, Taiga manages to obtain higher performance while generating the same output as vanilla C4.5.

The rest of this paper is organized in the following way: Section 2 will list other works in optimizing the performance of decision tree construction algorithms. Section 3 will describe the implementation details of the C4.5 algorithm. Section 4 will describe the RainForest framework, which inspired the Taiga algorithm described in Section 5. The performance of Taiga is evaluated in Section 6. Finally, we conclude this paper in Section 7, and discuss possible future directions in Section 8.

2 Related Work

RainForest[12] is a framework designed to accelerate decision tree construction algorithms, including C4.5. RainForest introduces the important concept of AVC-set, which stands for the set of numbers of occurrences of **A**ttribute-**V**alue **C**lass pairs. By isolating the steps of counting AVC-sets and constructing decision trees, RainForest allows one to implement different strategies including recursive, iterative, and hybrid approaches, with their own trade-offs.

The same authors also proposed BOAT[13], a technique that offers even better performance than RainForest. BOAT achieves this performance by speculatively constructing a decision tree from a number of decision trees from subsets of training data, that has a good chance of being similar to the "real" decision tree.

SPIES[14] is a parallel extension of RainForest. With the additional insight that calculating AVC-sets is essentially reduction operations, SPIES allows one to implement various decision tree construction algorithms for parallel execution.

The EC4.5 algorithm[15] achieved a performance gain of up to 5 times over C4.5. It does so by proposing three different ways to handle *continuous* attributes, one of them building upon the RainForest algorithm. Then it discusses when to choose which algorithm to achieve the optimal performance.

Various parallel algorithms have also been proposed. ScalParC[16] proposed the parallel hashing paradigm to help with the handling of continuous attributes in a parallel way.

3 C4.5 Tree-Construction Algorithm

3.1 Definition

Designed by Quinlan in 1993, C4.5 is one of the most popular decision tree algorithms. It supports both discrete and continuous attributes, dealing with unknown attribute values, and many other optional features. For the following discussion, we define the terminology as follows: For a case t in the training set, its attributes are denoted as $t_1, t_2, \ldots, t_n$. Further, each case belongs to a class, denoted by t.class. The set of all possible classes is Classes.

C4.5 constructs a decision tree with a divide-and-conquer approach. To construct the root node of the decision tree, C4.5 considers all possible tests. They are:

- A test in the form $t_i = a$, where t_i is a discrete attribute, with one outcome for each possible value a.
- A test in the form $t_i \leqslant r$ for a continuous attribute t_i and a real number r, with two possible outcomes `true` and `false`.

C4.5 enumerates all the possible tests and chooses the "best" one. By default, C4.5 uses the *information gain ratio* to determine how good a test is:

$$\text{gainRatio}(T) = \frac{\text{gain}(T)}{\text{baseInfo}(T)},$$

where $\text{gain}(T)$ is the information gain of the test, and $\text{baseInfo}(T)$ is the amount of information of the split caused by the test. There is also an option to just use the information gain instead.

For a discrete attribute, the information gain of the corresponding test is:

$$\text{gain} = \text{info}(T) - \sum_{i=1}^{s} \frac{|T_i|}{|T|} \times \text{info}(T_i),$$

where $T_1, \ldots, T_s$ are subsets of T corresponding to cases with different known values, for that attribute, and

$$\text{info}(T) = -\sum_{c \in \text{Classes}} \frac{|\{t | t \in T \wedge \text{t.class} = c\}|}{|T|} \times \log_2 \left(\frac{|\{t | t \in T \wedge \text{t.class} = c\}|}{|T|} \right).$$

On the other hand, $\text{baseInfo}(T)$ is calculated from the following formula:

$$\text{baseInfo}(T) = -\sum_{i=1}^{s} \frac{|T_i|}{|T|} \times \log_2 \left(\frac{|T_i|}{|T|} \right).$$

3.2 Algorithm

C4.5 employs a divide-and-conquer approach, as shown in Algorithm 1.

3.3 Data structure

The vanilla C4.5 code stores the training set using a continuous region of memory. Its layout can be seen in Fig. 1a. To support the divide-and-conquer strategy, C4.5 creates an array of pointers pointing to each of the training set cases, as shown in Fig. 1b.

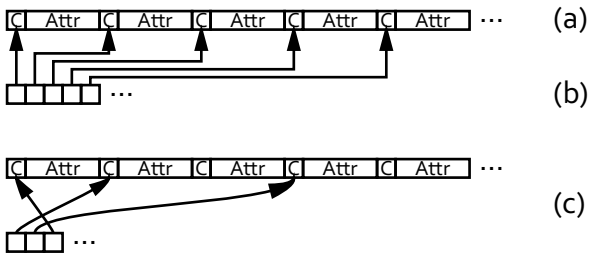
After C4.5 chooses a test for a certain node, and before it moves on to one of its children nodes, it scans through the pointer array and moves the cases relevant to the new node together. This operation involves a number of pointer swapping. After processing a few nodes, the pointers might be in an apparently random order, as seen in Fig. 1c.

4 RainForest Framework

RainForest[13] was proposed to make decision tree construction more scalable.

Algorithm 1: Top-down approach of C4.5

```
   Data: A training set T
   Result: A decision tree
1  for every attribute A do
2  |  Calculate the Information Gain Ratio for using A to
   |  splitting T;
3  end
4  if ∄A s.t. gainRatio (Split(T; A)) > threshold then
5  |  return A degenerated tree with only one node
6  end
7  Construct a root node with the selected test;
8  for every subtree do
9  |  Move all cases belonging in the subtree to a
   |  continuous memory area;
10 |  Recursively call C4.5 to construct the subtrees,
   |  using the subset of training cases as its training set;
11 end
```



(a) The *class* and *attributes* for each case.
(b) A pointer array pointing to all available cases.
(c) After a divide-and-conquer iteration.

Fig. 1 The C4.5 memory layout.

4.1 AVC-set

AVC-sets are a core concept proposed in RainForest. The AVC-set of an attribute a of a specific node n is defined as the counts of cases with distinct attribute values of a and distinct classes. The AVC-group of a node n is the set of all AVC-sets of it.

By defining the AVC-sets, RainForest manages to separate scalability issues from decision tree quality issues. This is because of an important observation: with only the AVC-group of a certain node, there is enough information for most decision tree building algorithms to decide whether that node should be a leaf or a non-leaf decision node, and if the latter, which test should be used at that node.

RainForest proposed a number of algorithms, differing in how many AVC-groups one is able to fit in the memory.

4.2 Algorithms

The first algorithm proposed by RainForest is the RF-Write algorithm. It is essentially the generalized form of Algorithm 1. However, since RainForest assumes the data is stored in disks, the step 9 will incur quite some time cost.

The second algorithm proposed by RainForest is the RF-Read algorithm. Instead of constructing the decision tree with a top-down recursive approach, RF-Read constructs the tree *iteratively* by calculating the AVC-groups of all nodes in the same level of the decision tree simultaneously, as shown in Algorithm 2.

However, there is the possibility that AVC-groups of all nodes in the same level of the decision tree won't fit in memory. This could be solved by using multiple

Algorithm 2: The RF-Read algorithm

```
   Data: A training set T
   Result: A decision tree
1  frontier ← A root node covering T;
2  while frontier≠∅ do
3  |  Initializef;
4  |  for t ∈ T do
5  |  |  for j do
6  |  |  |  f_ijkl ← f_ijkl +1 where i is the id of the
   |  |  |  node of t belongs to, k=t_j, and
   |  |  |  l=t:class;
7  |  |  end
8  |  end
9  |  Choose a test for each node in frontier;
10 |  frontier ← {children of frontier};
11 end
```

passes to construct one level of nodes, each pass calculating a subset of the AVC-groups. Since RF-Read does not rearrange the training cases, one has to read a case before determining if this case corresponds to a node whose AVC-group is currently being calculated. Therefore, in the case AVC-groups don't fit in memory, RF-Read will perform poorly in term of its running time.

Based on the above observations, a third algorithm is proposed. This algorithm is called RF-Hybrid, which is a mixed strategy that combines RF-Read and RF-Write. Basically, RF-Hybrid is RF-Read as long as the AVC-Groups of all nodes in the current level fit in memory. When this no longer holds, RF-Hybrid switches to RF-Write. Evaluations show that RF-Hybrid is generally the best algorithm among those proposed in RainForest, which is intuitive since it combines the best part of the other two algorithms.

Finally, a fourth algorithm, RF-Vertical, is proposed. However, it is designed for situations that even one AVC-group cannot be fitted in memory. We won't go on details of this algorithm, since it is irrelevant to our assumption.

5 Taiga Algorithm

Focusing on the different assumption that the database is stored in memory, however, our version of problem is slightly different from RainForest's. On one hand, RF-Read is often runnable from start to end, due to the much larger memory size of modern systems. On the other hand, the high running time overhead caused by more disk writing associated with RF-Write is alleviated, because now we are only writing to memory. Therefore, there is a need to reevaluate the strategies in this new context.

5.1 Taiga-Recursive

We include a recursive strategy Taiga-Recursive as a baseline algorithm. It is essentially what the vanilla C4.5 algorithm does, with extra timing code added. However, this algorithm will become the basis of modifications that Taiga-Hybrid and Taiga-Transposed apply on. Therefore we had better give it its own name.

5.2 Taiga-Iterative

Taiga-Iterative is a modified version of C4.5 inspired by RF-Read. Since the most significant difference between Taiga and RainForest is the assumption regarding the memory size, the most important difference between Taiga-Iterative and RF-Read is naturally the data structure.

A new data structure f is introduced in Taiga. Logically, f is a four-dimensional array where

$$f_{ijkl} = \left|\{t|t \in T_i \wedge t_j = k \wedge \text{t.class} = l\}\right|,$$

where T_i is the i-th node in the current level of the decision tree.

However, since the number of possible values of different attributes may differ greatly, in the implementation we have to pack all the values of f_{ijkl} into a continuous memory region with the help of some smaller auxiliary data structure. Thus, the size of f is

$$4 \times \text{Number of Classes} \times \text{Number of Branches} \times \sum_i (\text{Number of possible values for attribute } i).$$

5.3 Taiga-Hybrid

The Taiga-Iterative algorithm performs much better than the original algorithm when the number of nodes in each level is small. For many applications, this is desirable since small trees are often preferable. For example, the decision trees used in bagging and boosting are generally small[17].

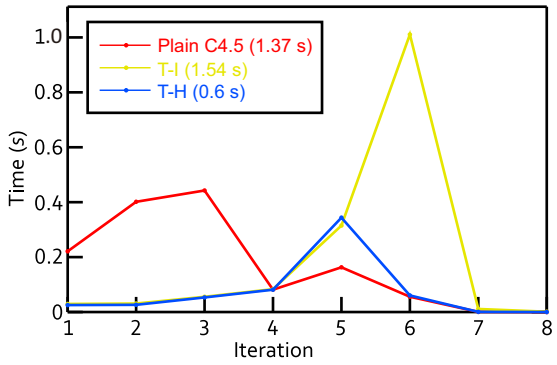
However, if, for some reason, the user decides she wants to generate a decision tree with large depth, Taiga-Iterative could get into trouble when dealing with deep levels. Figure 2 shows how the performance of Taiga-Iterative degrades as the tree grows deeper.

The solution to this problem is a *hybrid* approach, similar to RF-hybrid. Taiga-Iterative is run until a certain criterion is satisfied, then Taiga-Recursive is used. The discussion about which criterion works best is deterred to Section 5.6.

Intuitively, this hybrid algorithm combines the best part of both strategies. However, as shown in Fig. 2, the hybrid algorithm itself introduces some runtime overhead. This overhead is partly due to the time spent to reshuffle the pointer array, and partly due to the colder cache after switching to the vanilla algorithm. The overhead is generally small though.

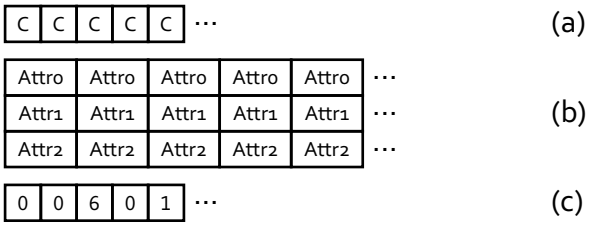
5.4 Transposing source data

The algorithm Taiga-Iterative works best if all AVC-groups of the current level of the decision tree could be fit into the cache. Therefore, we want to reduce the working set to improve performance. The idea is to rearrange the training data so that we only read the data about one certain attribute per time. Thus we propose a new data structure shown in Fig. 3.



Different versions of C4.5 running the training set `poker-hand-test` on Machine **A**.

Fig. 2 Per-iteration performance comparison.



(a) Classes are stored separately in an array;
(b) Attribute values are stored continuously for each attribute;
(c) We use the same method to store which sample belongs to which node.

Fig. 3 Transposed data structure.

However, this change would also increase the total number of read operations, since the array storing which sample belongs to which node has to be read multiple times during processing one level.

Applying this change to Taiga-Recursive, Taiga-Iterative, and Taiga-Hybrid yields Taiga-Recursive-Transposed, Taiga-Iterative-Transposed, and Taiga-Hybrid-Transposed, respectively.

5.5 Choosing switching criterion

From Fig. 2, we can clearly see that the performance of Taiga-Iterative may suffer even if all the AVC-sets fit in memory. Therefore, whether the AVC-sets fit in memory is not an adequate switching criterion for us.

To figure out a good switching criterion, we use the running time data obtained from the experiments described in Section 6.

We enumerate all combinations as the left part of the switching criterion, where the parameters shown in Table 1 are either absent, multiplied to the numerator or multiplied to the denominator. Limiting the formulas enumerated in this manner has a few advantages. First, it ensures the resulting formulas are at least somewhat explainable. Second, it ensures the simplicity of resulting formulas, reducing the probability of "overfitting" it. Finally, it ensures the enumeration could be done in a reasonable time budget.

For each combination, we find the best value of k for each machine separately. A score is calculated as the sum of logarithm of expected speedups over vanilla C4.5 in all test cases (as described in Section 6.2).

The top formulas are shown in Table 2 and Table 3.

5.6 Interpreting switching criterion

One can find that neighboring formulas in Table 2 and Table 3 often differ only by how they handle *Class*. It can be seen that Class does not affect the results significantly. This might be due to the small variation in Class in our test cases. Many training sets have only two classes. Others have only up to 23 classes, which offers little variation when compared to other parameters that varies more greatly.

There are two main factors that affect the time usage of Taiga-Iterative and Taiga-Recursive. The first factor is the size of an AVG-group. When the size of an AVG-group becomes too large, the cache miss rate will be huge, and the time consumed in zeroing all the AVC-groups and postprocessing the AVC-groups will be taken more compared to processing the training set. The second factor is the proportion of active samples. If most samples belong to branches that are already finished, that is, branches that do not grow to the current level, Taiga-Iterative would be less effective because it still needs to read the whole training set.

Consider the formula of the size of an AVC-group,

$$\text{AttrSum} \times \text{Class},$$

that of AVC-groups of all nodes in one level,

Table 1 Parameters in consideration.

Name	Formula	Depend on level?
Attr	$(\lvert\text{Attributes}\rvert + 1)$	No
Class	$(\lvert\text{Classes}\rvert + 1)$	No
AttrSum	$\sum_{i \in \{\text{Attributes}\}} (\lvert i \rvert + 1)$	No
Sample	$(\lvert\text{Samples}\rvert)$	No
ActiveSample	$\lvert\{s \lvert s \in \{\text{Samples}\} \text{ and } s \text{ belongs to some node in the current level}\}\rvert$	Yes
Branch	Number of nodes in the current level	Yes

Table 2 Best switching criteria for Taiga-Hybrid.

Rank	Formula	k (For three machines respectively)	Score
1	$\frac{\text{Attr}\times\text{Sample}\times\text{ActiveSample}}{\text{AttrSum}\times\text{Branch}} < k$	5832, 5832, 5832	13.4
2	$\frac{\text{Attr}\times\text{Sample}\times\text{ActiveSample}}{\text{AttrSum}\times\text{Class}\times\text{Branch}} < k$	715.5, 715.5, 1510	13.0
3	$\frac{\text{Sample}\times\text{ActiveSample}}{\text{AttrSum}\times\text{Branch}} < k$	507.9, 507.9, 507.9	12.6
4	$\frac{\text{Attr}\times\text{Class}\times\text{Sample}\times\text{ActiveSample}}{\text{AttrSum}\times\text{Branch}} < k$	61301, 61301, 61301	12.5
5	$\frac{\text{Sample}\times\text{ActiveSample}}{\text{AttrSum}\times\text{Class}\times\text{Branch}} < k$	116.2, 116.2, 72.87	12.2

Note: The three machines are described in Section 6.1. Changing machine seldom affects the k value.

Table 3 Best switching criteria for Taiga-Transposed-Hybrid.

Rank	Formula	k (For three machines respectively)	Score
1	$\frac{\text{Attr}\times\text{Class}\times\text{Sample}\times\text{ActiveSample}}{\text{Branch}} < k$	4.78×10^8, 4.78×10^8, 4.78×10^8	16.2
2	$\frac{\text{Attr}\times\text{Sample}\times\text{ActiveSample}}{\text{Branch}} < k$	1.22×10^8, 1.22×10^8, 1.22×10^8	16.1
3	$\frac{\text{Attr}\times\text{ActiveSample}}{\text{Class}\times\text{Branch}} < k$	4.31×10^7, 4.31×10^7, 4.31×10^7	15.4
4	$\frac{\text{Attr}\times\text{Sample}\times\text{ActiveSample}}{\text{AttrSum}\times\text{Branch}} < k$	6603, 6603, 6603	15.3
5	$\frac{\text{Attr}\times\text{Class}\times\text{Sample}}{\text{Branch}} < k$	1.79×10^4, 1.79×10^4, 1.79×10^4	15.2

$$\text{AttrSum}\times\text{Class}\times\text{Branch},$$

and the proportion of the active samples,

$$\frac{\text{ActiveSample}}{\text{Sample}}.$$

By observation, we can find that most of the top formulas point out that when Branch grows too large, the total size of AVC-groups will also grow and this makes Taiga-Recursive more competitive. On the other hand, when ActiveSample is large, Taiga-Iterative becomes more attractive.

A probable explanation about why Attr and Sample correlate with the relative performance of Taiga-Iterative is that they correlate the training set size, and a large training set wants Taiga-Iterative more, because it reads the training set in fewer times, and does not need to rewrite the data back after rearranging it.

For the sake of simplicity, further evaluation in Section 6 will only use the first formula in both situations.

5.7 Handling unknown attribute values

C4.5 has built-in support for unknown attribute values in the training set. When choosing a test for a specific node, an unknown attribute value is considered to conform to the same probabilistic distribution as the training cases belonging to that node. If an attribute with unknown values in the training cases is chosen, those training cases will be considered to belong to all children nodes simultaneously. However, these training cases are considered to be "split" to the children nodes, according to the distribution of known values for the same attribute in that node.

The concept of *weight* is introduced to help handling this. In the beginning all cases have weight 1. As cases are split for unknown attribute values, they get fractional weights. When calculating values such as information and case count, a weight sum is used instead of plain summation.

Conceptually, C4.5 splits one training case with an unknown attribute into many, each with the same attribute assigned to some fixed value and having a fractional weight. This splitting is implicit during the recursive implementation.

However, in Taiga, we have to *explicitly* do the splitting: whenever we choose a test that tests an attribute some cases do not have, we split all these cases into multiple copies with fractional weights. This means the number of training cases stored in memory may grow during Taiga in case of unknown attributes. The time complexity of Taiga in this situation remains the same as vanilla C4.5. However, the space complexity of Taiga can be exponentially larger than vanilla in the worst case. Fortunately, the hybrid algorithm with the proposed switching criterion is

adequate in avoiding the worst case, by switching to the old algorithm when the space complexity seems to grow unreasonably.

This approach is different from RainForest as well. RainForest simply avoids this problem by not recording which training case belongs to which node. Instead, RainForest utilizes the finished part of the decision tree to locate the node. Therefore, in case of an unknown value, RainForest will simply allow the same implicit splitting to happen.

6 Evaluation

We implemented Taiga on top of the official C4.5 Release 8 source code.

6.1 Training sets

At C4.5's time, training data were tiny compared to our current standards. For certain considerations such as the ability to measure performance on continuously controllable variables, RainForest relied on the data generator introduced in Ref. [18], which was also used in many previous works such as Refs. [19, 20]. However, the data generator has the following problem:

(1) What it generates is not real data. Therefore, it is hard to argue that it is representative of real life situations.

(2) Its AVC-set sizes are dependent on the training set size. Therefore, no matter how one switches the f-functions and data sizes, the generated training sets still cannot cover many practical situations.

Since the considerations that mandated the usage of only generated data no longer hold, we can use real training data in our research work. Also, since our assumption is that the training set can be fitted in memory, we do not really need the largest training set possible. Instead, we just pick those training data set that's adequately large enough. We use training sets available from the UCI Machine Learning Repository website (http:// archive.ics.uci.edu/ml/). We pick training sets on the following criteria:

- The training set constitutes a *classification* problem.
- The training set should be able to fit in memory.
- The training set should be large enough so that its running time can be reliably measured.
- The training set should be reasonable for a decision tree approach. A training set with ten million attributes is probably better solved in other methods than a decision tree, so we exclude such data sets.

Furthermore, we also use the data generator introduced in Ref. [18]. To avoid overweighting the data generator's data in our analysis however, we only include one instance of generated data. In our experiments, test cases generated by the ten algorithms exhibit similar performances.

The description and characteristics of the training sets are shown in Table 4.

We measure the number of branches and the number of active cases in each level of the decision tree. They are shown in Fig. 4.

6.2 Hardware

We run both vanilla C4.5 and Taiga on a few different machines. Their specifications are shown in Table 5.

6.3 Speedup

We measure the running time of the tree building process.

The time spent for loading the data from the disk, pruning the tree, and outputting are excluded. The pruning stage is generally accepted to be orthogonal to other stages. It is also less time-consuming than the tree building stage[12,21]. Therefore, to improve the quality and accuracy of our reports, we exclude these stages from our performance measurement.

To reduce the effect of system performance fluctuation, we report the minimal time across twenty runs. The results are shown in Fig. 5. More detailed per-level running time is shown in Fig. 6.

From the results we can see that Taiga-Hybrid is generally a safe choice in that it always offers a

Table 4 The training sets.

Name	Attibutes	Classes	Samples
`adult`*	14	2	32 561
`census`	41	2	3651
`connect-4`	42	2	67 557
`covtype`	12	7	15 972
`donation`	9	2	5 749 131
`ijcnn1`	13	2	49 990
`kddcup`	41	23	4 898 431
`mnist`	270	10	60 000
`poker-hand`	10	10	25 010
`poker-hand-test`†	10	10	1 000 000
`skin`	3	2	245 057

Notes: *, Attribute `fnlwgt` is ignored because C4.5 does not support setting weight by default.
†, The "test" data set of `poker-hand`.

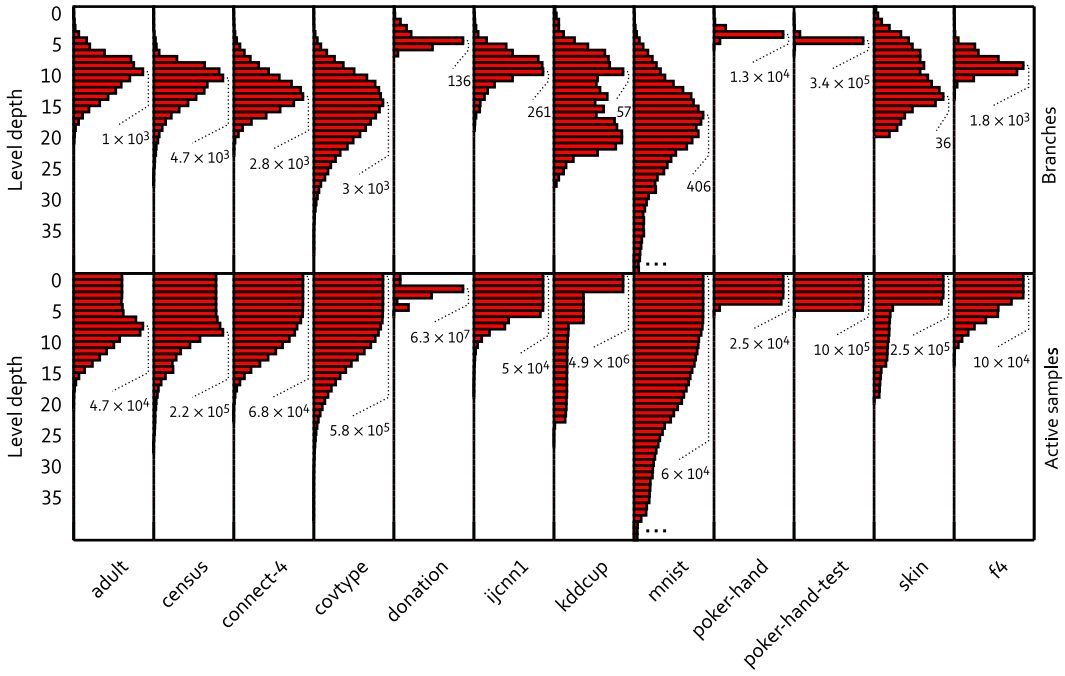


Fig. 4 The shapes of decision trees. The *x*-axis for each training set corresponds to the number of branches and active samples in each level of the decision trees. As the number of branches and active samples varies greatly from training set to training set, each graph is scaled individually. The number of active samples occasionally increases as the level goes deeper due to how unknown values are handled.

Table 5 The hardware specification.

Name	CPU	Microarchitecture	Cache
A	Xeon E7-4870	Westmere EX	35 MiB
B	Xeon E5-2697	Haswell	30 MiB
C	A10-7850K	Steamroller	20 MiB

near-optimal performance, and avoids possible out-of-memory issues.

Transposed versions of Taiga generally perform worse due to the increased amount of memory reads. However they bring benefit to mnist by reducing the size of the working set greatly, and they optimize poker-hand and poker-hand-test by switching with better timing.

7 Conclusion

The hardwares are ever progressing and changing. It is often worthy to revisit old ideas to see their application in new assumptions.

This work is different from RainForest in the following aspects:

(1) *The assumptions are different*. The two works made different basic assumptions.

(2) *The data structures are different*. To reduce the amount of calculation, Taiga introduces an extra array to record which sample belongs to which node. Taiga-Transposed further modifies the memory layout of training data.

(3) *The algorithm switching criteria are different*. RF-Hybrid switches to RF-Write as soon as the AVC-groups can't be fit in the memory. In contrast, Taiga-Hybrid uses a more sophisticated criterion to switch between Taiga-Recursive and Taiga-Iterative.

8 Future Work

Taiga handles continuous attribute values with the same nave algorithm as used by RainForest, that is, when calculating the AVC-sets, one has to count the occurrences of all pairs of distinct attribute values and class labels. This means even if we do not alter the other parts of the algorithm at all, Taiga will run slower than plain C4.5 when both the number of distinct attribute values and the number of class labels are large. However, there are certainly better ways to handle continuous attribute values, such as using the *partial AVC Group* proposed in SPIES[15], or using the optimization to RainForest proposed in EC4.5[16]. It should be straightforward to integrate this work into Taiga.

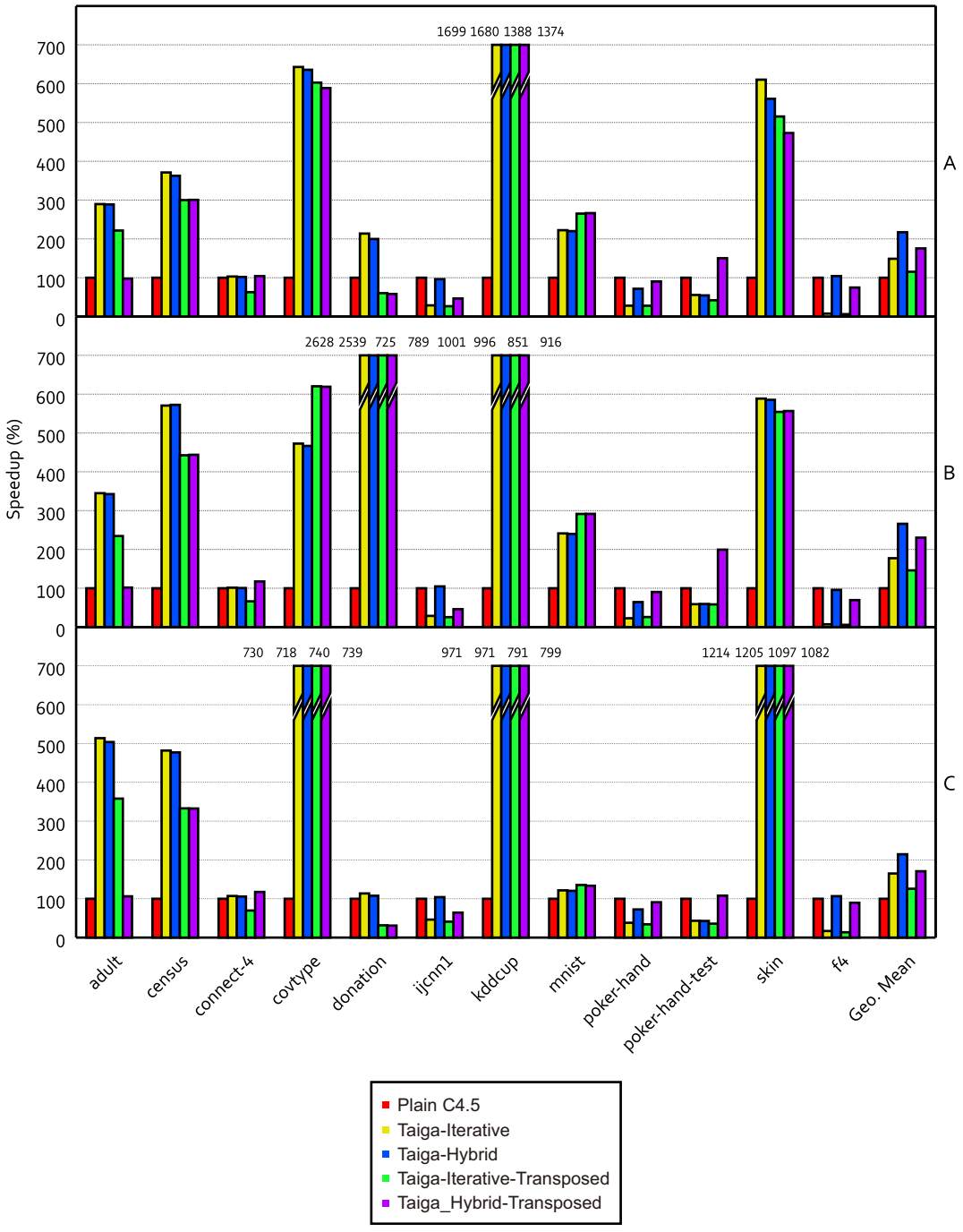


Fig. 5 Comparison of running time of different algorithms on three machines (shown in Table 5).

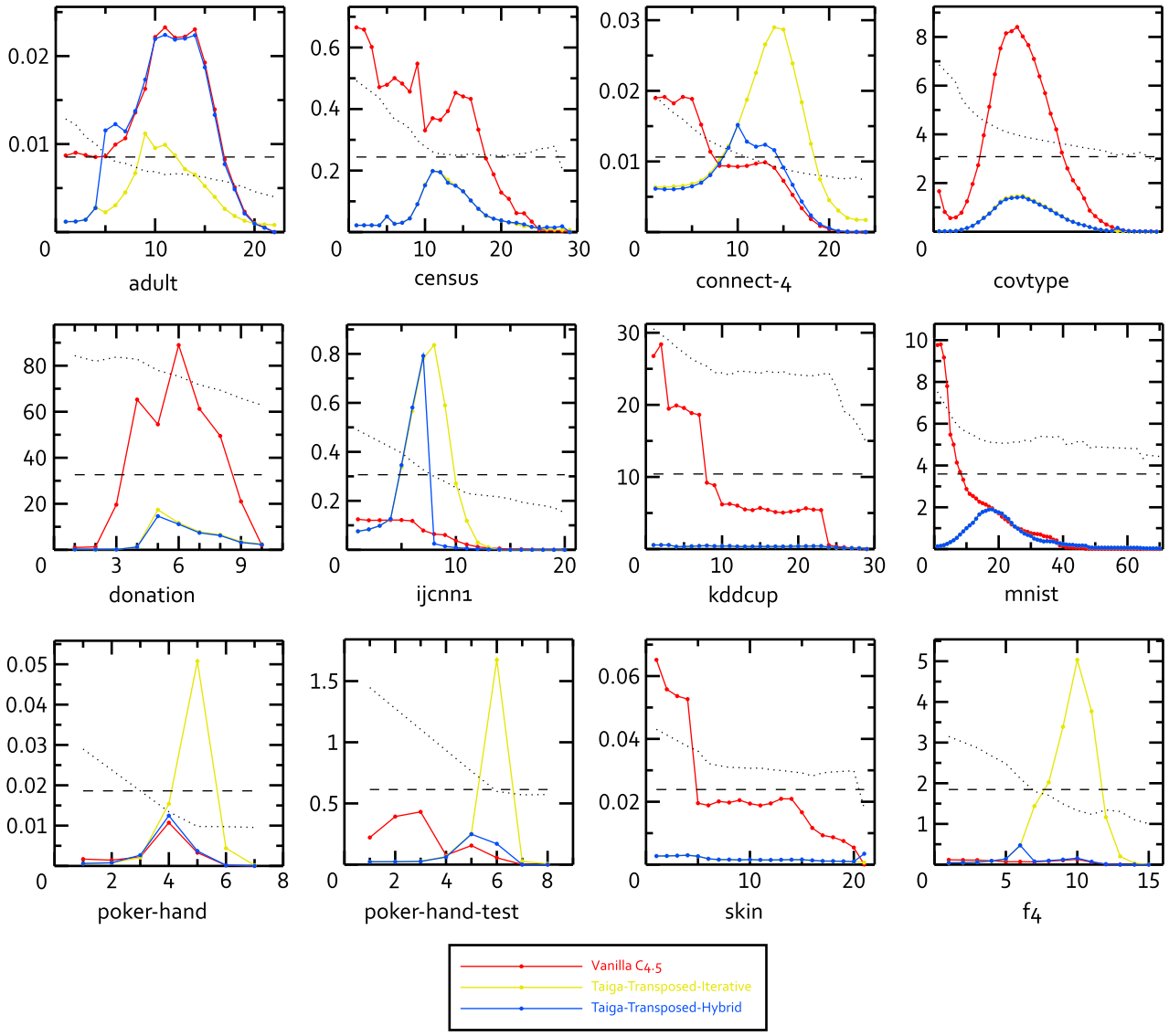
One of the other important ideas for optimizing decision tree algorithms is parallelization. While Taiga removes the opportunities to exploit the inherit parallelism in the divide-and-conquer paradigm, it allows more straightforward and balanced parallel implementation by simply dividing all the relevant cases evenly. Therefore, we expect good speedup when combining this technique with parallel algorithms.

References

[1] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann, 2001.

[2] S. R. Safavian and D. Landgrebe, A survey of decision tree classifier methodology, *IEEE Transactions on Systems, Man and Cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[3] J. Cheng, U. M. Fayyad, K. B. Irani, and Z. Qian, Improved decision trees: A generalized version of ID3, in *Proc. Fifth*



X-axis: the level of decision trees; *Y*-axis: the running time spent in seconds.

Fig. 6 Comparison of running time on machine A. The dotted lines and dasdhed lines show the left and right sides of the switching criterion respectively.

Int. Conf. Machine Learning, 1988, pp. 100–107

[4] J. R. Quinlan, Learning efficient classification procedures and their application to chess end games, in *Machine Learning*. Springer, 1983, pp. 463–482

[5] J. R. Quinlan, Induction of decision trees, *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[6] J. R. Quinlan, *C4.5: Programs for Machine Learning*, vol. 1. Morgan Kaufmann, 1993.

[7] J. R. Quinlan, Data mining tools see5 and C5.0, http://www.rulequest.com/see5-info.html, 2004.

[8] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*. CRC Press, 1984.

[9] G. V. Kass, An exploratory technique for investigating large quantities of categorical data, *Applied Statistics*, vol. 29, no. 2, pp. 119–127, 1980.

[10] W.-Y. Loh and N. Vanichsetakul, Tree-structured classification via generalized discriminant analysis, *Journal of the American Statistical Association*, vol. 83, no. 403, pp. 715–725, 1988.

[11] M. Mehta, J. Rissanen, and R. Agrawal, Mdl-based decision tree pruning, *KDD*, vol. 21, pp. 216–221, 1995.

[12] J. Gehrke, R. Ramakrishnan, and V. Ganti, Rainforest-a framework for fast decision tree construction of large datasets, *VLDB*, vol. 98, pp. 416–427, 1998.

[13] J. Gehrke, V. Ganti, R. Ramakrishnan, and W.-Y. Loh, Boat–optimistic decision tree construction, *ACM SIGMOD Record*, vol. 28, pp. 169–180, 1999.

[14] R. Jin and G. Agrawal, Communication and memory efficient parallel decision tree construction, *SDM*, pp. 119–129, 2003.

[15] S. Ruggieri, Efficient C4.5, *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, no. 2, pp. 438–444, 2002.

[16] M. V. Joshi, G. Karypis, and V. Kumar, Scalparc: A new scalable and efficient parallel classification algorithm for mining large datasets, in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing 1998*, 1998, pp. 573–579

[17] J. R. Quinlan, Bagging, boosting, and C4.5, *AAAI/IAAI,* vol. 1, pp. 725–730, 1996.

[18] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami, An interval classifier for database mining applications, in *Proc. of the VLDB Conference*, 1992, pp. 560–573.

[19] R. Agrawal, T. Imielinski, and A. Swami, Database mining: A performance perspective, *Knowledge and Data Engineering, IEEE Transactions on*, vol. 5, no. 6, pp. 914–925, 1993.

[20] J. Shafer, R. Agrawal, and M. Mehta, Sprint: A scalable parallel classifier for data mining, in *Proc. 1996 Int. Conf. Very Large Data Bases*, 1996, pp. 544–555.

[21] M. Mehta, R. Agrawal, and J. Rissanen, Sliq: A fast scalable classifier for data mining, in *Advances in Database Technology EDBT'96*, 1996, pp. 18–32

Wenguang Chen received the BS and PhD degrees in computer science from Tsinghua University in 1995 and 2000, respectively. He was the CTO of Opportunity International Inc. from 2000-2002. Since January 2003, he joined Tsinghua Univeristy. He is now a professor in Department of Computer Science and Technology, Tsinghua University. His research interest is in parallel and distributed computing, programming model, and mobile cloud computing.

Yi Yang received the BS degree from Tsinghua University in 2012, and is now a master student. His research interests include program optimization and program analysis.