# Automated Formula Generation and Performance Learning for the FFT

Bryan Singer Manuela Veloso

January, 2000
CMU-CS-00-123

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

A single signal processing algorithm can be represented by many different but mathematically equivalent formulas. When these formulas are implemented in actual code, they often have very different running times. Thus, an important problem is finding a formula that implements the signal processing algorithm as efficiently as possible. In this paper we present three major results toward this goal: (1) Different but mathematically equivalent formulas can be generated automatically in a principled way, (2) Simple features describing formulas can be used to distinguish formulas with significantly different running times, and (3) A function approximator can learn to accurately predict the running time of a formula given a limited set of training data.

This research was sponsored by the DARPA Grant No. DABT63-98-1-0004. The first author, Bryan Singer, is partly supported by a National Science Foundation Graduate Fellowship.

The content of the information in this publication does not necessarily reflect the position or the policy of the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation (NSF), or the US Government, and no official endorsement should be inferred.

**Keywords:** machine learning, signal processing, FFT, performance prediction, mathematical algorithms, application of neural networks, OPAL

# 1 Introduction

Most signal processing algorithms can be represented by a matrix which when multiplied by an input vector produces the desired output vector [4, 5]. A straightforward implementation of the algorithm would be to simply implement the multiplication of the specified matrix and the input vector. However, these matrices often have a particular form that allows them to be factored into a product of sparse, structured matrices. These factorizations allow for faster implementations of signal processing algorithms. Further, these factorizations can be represented by mathematical formulas [1].

A single signal processing algorithm can be represented by many different but mathematically equivalent formulas. When these formulas are implemented in actual code, they often have very different running times. Thus, an important problem is finding a formula that implements the signal processing algorithm as efficiently as possible [3].

This paper presents our preliminary work towards this goal. In particular, this paper contains three major results:

- Different but mathematically equivalent formulas can be generated automatically in a principled way.
- Simple features describing formulas can be used to distinguish formulas with significantly different running times.
- A function approximator can learn to accurately predict the running time of a formula given a limited set of training data.

# 2 Formula Generator

Given that there are many different formulas that represent a single signal processing algorithm, an important problem is determining all the different formulas that represent this algorithm. That is, if we want to find the fastest formula that implements a particular algorithm, then we need to know what the set of formulas that represent the algorithm is.

We have written a formula generator that takes a formula and a set of rewrite rules and produces all mathematically equivalent formulas according to the rewrite rules. This formula generator provides a principled method for generating all mathematically equivalent formulas of some specified formula.

## 2.1 Rewrite Rules

A rewrite rule states how one formula can be "rewritten" as a different but mathematically equivalent formula. Each rewrite rule consists of: (1) a template formula, (2) a result formula, and (3) a set of variables. The template consists of a formula that is to be matched with the current formula or a subexpression of the current formula. The result formula consists of a formula that is mathematically equivalent to the template, and so the result formula can replace the template formula.

Variables may be used in both the template and the result formulas. Two kinds of variables are possible — input and computable variables. Input variables simply match appropriate portions of the input formula and can be used to copy such into the result formula. Thus, input variables allow templates to match many different formulas. For example, an input variable could represent the size of a particular object, as in the following example:

```
(RULE TRANSPOSE-IDENTITY
      (vars n)
      (template (transpose (i n)))
      (result (i n)))
```

which says that whenever we find a transpose of the identity matrix (of any size n), we can replace it simply by the identity matrix (of the same size n).

Computable variables allow values to be computed from input variables that can be used in the result formula. For example, two computable variables could be used to capture a factorization of an integer. In particular, several sets of computable variables can be defined, and for each set of computable variables a function must be given. This function may take as arguments any of the input variables or constants. The function then produces a list of sets of values for the computable variables. Each set of values on this list is used to produce a different result formula, and thus a single rule matching a single formula can actually produce many result formulas.

As an example, consider the rewrite rule:

```
(RULE COOLEY-TUKEY
      (vars n)
      (template (f n))
      (c-vars ((r s) (factors n)))
      (result
       (compose (compose (compose (tensor (f r) (i s))
                                  (t n s))
                         (tensor (i r)(f s)))
                (l n r))))
```

which says that $F_n$ can be replaced by $(F_r \otimes I_s)\, T^n_s\, (I_r \otimes F_s)\, L^n_r$ for any integer factorization $rs$ of $n$ (assuming the function "factors" is appropriately defined).

## 2.2 Formula Search Space

The number of formulas that can be produced by our formula generator can be very large. When given a formula and a set of rewrite rules, the formula generator tries to apply each of the rewrite rules to the given formula and all subexpressions of the formula. Plus if any of these produce a resulting formula, then all of the rewrite rules can be recursively tested on the resulting formula and all of its subexpressions.

Currently our formula generator uses breadth first search. An open research question is how to avoid producing an infinite set of formulas, most of which are useless (e.g., `(transpose (transpose (transpose (f 32))))`).

# 3 Cooley-Tukey

A very important signal processing algorithm is the Fast Fourier Transform (FFT) [5]. One particularly useful factorization of the FFT is the Cooley-Tukey which has the form: $F_{rs} = (F_r \otimes I_s)\, T_s^{rs}\, (I_r \otimes F_s)\, L_r^{rs}$. The key aspect of this factorization is that it splits a large FFT, $F_{rs}$, into two smaller FFT's, $F_r$ and $F_s$. This can be visualized as a tree, as in Figure 1(a). Likewise, a more complicated factorization such as

$$([\,(F_2 \otimes I_4)\, T_4^8\, (I_2 \otimes F_4)\, L_2^8\,] \otimes I_4)\, T_4^{32}\, (I_8 \otimes [\,(F_2 \otimes I_2)\, T_2^4\, (I_2 \otimes F_2)\, L_2^4\,])\, L_8^{32}$$

can be more compactly represented in the split tree shown in Figure 1(b).

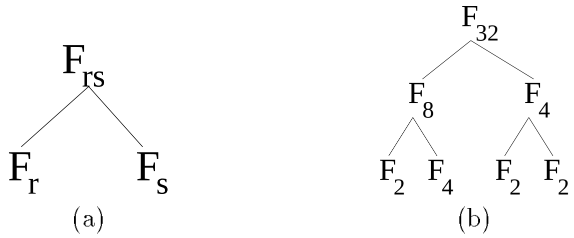


Figure 1: Split Trees: (a) A split tree for $F_{rs}$ and (b) A split tree for $F_{32}$

Much of the data that we used in the experiments that follow involved using the formula generator to produce all possible Cooley-Tukey expansions of a particular sized $F_n$. As an example, $F_{128}$ has 731 different formulas that are produced through applications of Cooley-Tukey. These formulas were then fed to a rather good FFT package [2] to generate running times for each of the formulas.

# 4 Relevant Features for Predicting Running Time

Given that many mathematically equivalent formulas have very different running times when implemented, an important question to ask is what about these formulas determine their running times? Or, equivalently, what are good features for predicting a formula's running time?

To answer these questions, we will begin by introducing several different feature sets that can be used to describe Cooley-Tukey expansions of an FFT. After each of these different feature sets have been described, we will then compare them along two different measures to see how well the features can differentiate formulas with different running times.

## 4.1 Feature Sets

We begin by introducing a simple set of features to describe formulas. In particular, we take advantage of the fact that all of these formulas are produced by repeated applications of Cooley-Tukey to a FFT. Then we successively refine these features in different ways to produce a class of feature sets.

### 4.1.1 Counting Leaf F's

One simple and yet important feature of a Cooley-Tukey expansion of an FFT formula is the number and sizes of the actual FFT's that appear in the formula. These are the $F_n$'s that appear as leaves in the split tree. Specifically, we count the number of $F_2$'s, the number of $F_4$'s, the number of $F_8$'s, and so on that appear in the formula.

For example, the split tree shown in Figure 1(b) would have the features:

- 3 $F_2$'s
- 1 $F_4$'s
- 0 $F_8$'s
- 0 $F_{16}$'s

### 4.1.2 Counting All I's

Considering the previous features and the split tree, one modification of the above features would be to count all of the F's that appear in all of the nodes of the split tree instead of just those in the leaves. If we ignore the root node, this is equivalent to counting the number of I's of different sizes in the actual formula. Recall that the form of the Cooley-Tukey expansion is $F_{rs} = (F_r \otimes I_s)\ T^{rs}_s\ (I_r \otimes F_s)\ L^{rs}_r$. While the $F_r$ and $F_s$ maybe recursively expanded with the Cooley-Tukey, the $I_r$ and $I_s$ are maintained and thus leave a trace of how the split tree was built.

For example, the split tree shown in Figure 1(b) would have the features:

- 3 $I_2$'s
- 2 $I_4$'s
- 1 $I_8$'s
- 0 $I_{16}$'s

### 4.1.3 Counting Leaf F's and All I's

For sufficiently large split trees, it is possible for two different formulas to have the exact same I counts, but to have different leaf F counts. For example, see Figure 2. So, a simple refinement of the previous two feature sets would be to include both. That is, we would count the number of $F_2$'s, the number of $I_2$'s, the number of $F_4$'s, etc. in the formula.

For example, the split tree shown in Figure 1(b) would have the features:

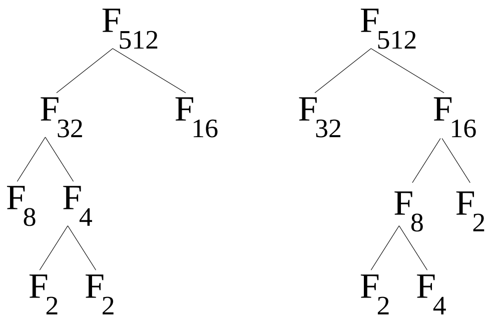


Figure 2: Two split trees with the same I counts but different Leaf F counts

- 3 $F_2$'s and 3 $I_2$'s
- 1 $F_4$'s and 2 $I_4$'s
- 0 $F_8$'s and 1 $I_8$'s
- 0 $F_{16}$'s and 0 $I_{16}$'s

#### 4.1.4 Counting Left and Right Leaf F's

Consider again the first set of features that we introduced which simply counted all of the leaf F's. A different refinement of this would be to separate F's that are right children of their parents in the tree and those that are left children. In particular, we would count the number of left $F_2$'s, the number of right $F_2$', the number of left $F_4$'s, and so on in the formula.

For example, the split tree shown in Figure 1(b) would have the features:

- 1 Right $F_2$'s and 2 Left $F_2$'s
- 1 Right $F_4$'s and 0 Left $F_4$'s
- 0 Right $F_8$'s and 0 Left $F_8$'s
- 0 Right $F_{16}$'s and 0 Left $F_{16}$'s

#### 4.1.5 Counting Left and Right I's

Combining the idea in the previous subsection along with the idea of counting all the nodes in the split tree, produces yet another set of features. In particular we count the number of different sized left and right F's appearing in the tree, excluding the root node. This is equivalent to counting the number of different sized I's on the right or left side of the tensor product within the formula itself.

For example, the split tree shown in Figure 1(b) would have the features:

- 1 Right $I_2$'s and 2 Left $I_2$'s
- 2 Right $I_4$'s and 0 Left $I_4$'s
- 0 Right $I_8$'s and 1 Left $I_8$'s
- 0 Right $I_{16}$'s and 0 Left $I_{16}$'s

#### 4.1.6 Counting Left and Right Leaf F's and All I's

Once again, counting left and right I's can't always distinguish two trees that counting left and right F's can distinguish. Thus, we again combine the two for a large set of features the include all those in the previous two sets.

For example, the split tree shown in Figure 1(b) would have the features:

- 1 Right $F_2$'s and 2 Left $F_2$'s and 1 Right $I_2$'s and 2 Left $I_2$'s
- 1 Right $F_4$'s and 0 Left $F_4$'s and 2 Right $I_4$'s and 0 Left $I_4$'s
- 0 Right $F_8$'s and 0 Left $F_8$'s and 0 Right $I_8$'s and 1 Left $I_8$'s
- 0 Right $F_{16}$'s and 0 Left $F_{16}$'s and 0 Right $I_{16}$'s and 0 Left $I_{16}$'s

### 4.2 Evaluating Features

#### 4.2.1 Number of Partitions

Because several different formulas can have the same set of feature values, the features can be thought of as generating a set of equivalence classes or partitions. Under a set of features, formulas are indistinguishable if they have the same set of feature values, while formulas are distinguishable if they have different feature values.

Thus, a very simple measure of the effectiveness of a set of features is the number of partitions it creates for a set of formulas. Some results are shown in Table 1. As was discussed in Section 3, we used the automatic formula generator to produce all Cooley-Tukey expansions of $F_{16}$, $F_{32}$, $F_{64}$, $F_{128}$, $F_{256}$, and $F_{512}$. The bottom line of the table show the number of different formulas produced. The remaining lines show how many different partitions or equivalence classes are generated by the different features for each set of formulas.

| | $F_{16}$ | $F_{32}$ | $F_{64}$ | $F_{128}$ | $F_{256}$ | $F_{512}$ |
|---|---|---|---|---|---|---|
| Leaf F | 5 | 7 | 11 | 15 | 22 | 30 |
| All I | 7 | 13 | 31 | 68 | 168 | 385 |
| Leaf F and All I | 7 | 13 | 31 | 68 | 168 | 386 |
| Left/Right Leaf F | 11 | 23 | 44 | 81 | 142 | 241 |
| Left/Right All I | 14 | 45 | 149 | 523 | 1832 | 6585 |
| Left/Right Leaf F and All I | 15 | 49 | 170 | 617 | 2262 | 8473 |
| All Formulas | 15 | 51 | 188 | 731 | 2950 | 12235 |

Table 1: Number of Partitions Generated by Different Feature Sets for all of the Cooley-Tukey expansions of Different Sized $F_n$'s

Note that for all the sizes of $F_n$, as we move down through successive refinements the number of partitions generally grows. That is, usually the feature sets towards the bottom of the table split the formulas more than those towards the top of the table. The one except to this is the Left and Right Leaf F features which really is a refinement of the Leaf F features instead of the All I features.

Also note that the final feature set, the Left and Right Leaf F and All I features, are able to almost, but not quite, uniquely identify all the formulas. However, as the size of $F_n$ grows, this feature set is less and less able to uniquely identify formulas.

### 4.2.2 Weighted Average Relative Standard Deviation

While being able to partition a set of formulas into a large set of equivalence classes is important, ultimately we are only concerned that all of the formulas within a partition have roughly the same running time. A good set of features then are ones that can separate formulas with significantly different running times into different partitions so that all formulas within a single partition have roughly the same running time. As a measure of this, we define "weighted average relative standard deviation." For each partition we calculate the standard deviation of the running times of all the formulas that fall into that partition. We then calculate the relative standard deviation for each partition by dividing the standard deviation by the mean. We then take a weighted average over all partitions, weighting each relative standard deviation by the number of formulas in the partition. See Table 2.

- Let $P_k$ be the set of formulas in partition $k$.
- Let $t_i$ be the running time of formula $i$.
- Let $m_k$ be the mean running time of the formulas in $P_k$. Then, $m_k = \frac{1}{|P_k|} \sum_{i \in P_k} t_i$.
- Let $\sigma_k$ be the standard deviation of the running times of the formulas in $P_k$. Then $\sigma_k = \sqrt{\frac{1}{|P_k|} \sum_{i \in P_k} (t_i - m_k)^2}$.
- Let $r_k$ be the relative standard deviation of the running times of the formulas in $P_k$. Then $r_k = \frac{\sigma_k}{m_k}$.
- Then, the Weighted Average Relative Standard Deviation is $\frac{\sum_k |P_k| r_k}{\sum_k |P_k|}$.

Table 2: Calculating Weighted Average Relative Standard Deviation

Some results are shown in Table 3. Once again all of the formulas automatically generated from Cooley-Tukey expansions of various sized $F_n$'s where used. Each formula was timed using the FFT package discussed in Section 3. Not surprisingly, the feature sets with the least and the most number of partitions ("Leaf F" and "Left/Right Leaf F and All I") have the worst (largest) and best (smallest) weighted average relative standard deviations, respectively. However, even though the Left/Right Leaf F feature set had more partitions in some cases than the All I feature set, it consistently had a much worse weighted average relative standard deviation — nearly as bad as that for the Leaf F feature set.

This shows that simply having more partitions does not mean a feature set better distinguishes formulas with different running times.

| | $F_{16}$ | $F_{32}$ | $F_{64}$ | $F_{128}$ | $F_{256}$ | $F_{512}$ |
|---|---|---|---|---|---|---|
| Leaf F | 3.807% | 5.152% | 6.631% | 6.200% | 7.015% | 7.166% |
| All I | 0.905% | 1.303% | 1.437% | 1.744% | 1.772% | 1.764% |
| Leaf F and All I | 0.905% | 1.303% | 1.437% | 1.744% | 1.772% | 1.741% |
| Left/Right Leaf F | 2.739% | 4.152% | 5.955% | 5.705% | 6.648% | 6.856% |
| Left/Right All I | 0.290% | 0.382% | 0.484% | 0.524% | 0.580% | 0.629% |
| Left/Right Leaf F and All I | 0.000% | 0.123% | 0.181% | 0.276% | 0.324% | 0.380% |

Table 3: Weighted Average Relative Standard Deviation of Different Feature Sets for all of the Cooley-Tukey expansions of Different Sized $F_n$'s

# 5 Learning to Predict Running Times

Given that there are many different Cooley-Tukey expansions of large FFT's and that they can have different running times, we would like to find the one with the fastest running time. One simple approach would be to use the formula generator to produce all of the formulas and to time each one on each different machine that we might be interested in. Then the formula with the fastest time can be determined for each machine.

There are two problems with this approach: (1) each formula may take a non-trivial amount of time to run, and (2) there are a very large number of formulas that need to be run. These problems make the approach intractable for FFT's of even fairly modest sizes.

In this section, we present an approach to help solve the first problem. In particular, our approach is as follows:

- Generate a small set of formulas automatically.
- Time each of these formulas.
- Describe the formulas by a set of appropriate features.
- Use this data to learn to quickly and accurately predict the running times of the remaining formulas.

With the features discussed in the previous section and with some training data obtained by timing a few formulas, we can use machine learning techniques to produce a function approximator that can quickly predict the running times of new formulas. Note that this still does not solve the second problem mentioned above: we still must search through a large space of potential formulas. However, we can now obtain a predicted running time much more quickly than we could have obtained an actual running time.

Note that while accurately predicting a formula's running time allows the fastest formula to be determined through exhaustive search over all formulas, it is actually more than necessary. In particular, accurately predicting which of two formulas runs faster would also allow the fastest formula to be determined through exhaustive search over all formulas. Thus, a learning algorithm need not learn the exact running time if it can accurately predict which of two formulas runs faster.

## 5.1 Experimental Setup

We decided to learn to predict running times for the formulas generated by Cooley-Tukey expansions of $F_{128}$. There are 731 such formulas. Timings were obtained in two ways: (1) actual timings through the FFT package discussed in Section 3, and (2) model approximations through the cost model shown in Table 4.

$\text{Cost}(I_m \otimes A) = \text{Cost}(A \otimes I_m) = m{*}\text{cost}(A)$
$\text{Cost}(A * B) = \text{Cost}(A) + \text{Cost}(B)$
$\text{cost}(F_n) = a * n^2 + b * n + c$
$\text{Cost}(T_r^n) = d * n + e$
$\text{Cost}(L_r^n) = 0$
Used $a = b = c = d = e = 1$

Table 4: Simple Cost Model

We used a neural network as the function approximator. For all of the results presented, we used 25 hidden units, a learning rate 0.01 and a momentum of 0.001. These parameters obviously are not highly tuned due to the fact that they were used across several different input feature sets (of varying number of inputs) and across desired output (running time or faster of two formulas).

## 5.2 Results

Results for the model data are shown in Table 5 and results for the real data are shown in Table 6. These tables are broken into several groups of rows, with each row corresponding to a particular set of features that were used. For each of these groups, experiments were run with different sized training and test sets. As a base case, all 731 formulas were used both in training and test in the first row of each group. The 4 following rows in each group correspond to randomly selecting a certain percentage of the formulas for training with the remaining formulas used as a test set. In this latter case, results are averaged over 4 random selections of training sets.

The column marked "Average Percent Error on Predicting Cost" reports the prediction error on the test set. In particular, it is calculated by dividing the absolute different between predicted cost and actual cost by the actual cost, and then averaging over all formulas in the test set:

- Let $c_i$ be the actual running time of formula $i$.
- Let $p_i$ be the predicted running time of formula $i$.
- Then the average percent error on predicting cost is $\frac{\sum_{i \in test-set} \frac{|c_i - p_i|}{p_i}}{|test-set|}$.

The column marked "Percent Mistakes on Predicting Faster of Two" reports the prediction error on a random sampling of pairs of formulas in the test set. In particular, the number of samplings was 100 times the number of formulas in the test set. The percentage was calculated by taking the number of pairs of formulas the network predicted incorrectly which ran faster and dividing it by the total number of pairs of formulas tested.

The "Leaf F and All I" and "Left and Right Leaf F and All I" models yielded the best learning results. These results were quite good with less than 12% error on predicting the faster of two formulas and less than 9% error on predicting the running times. In fact, the error was much less than these in most cases.

Interestingly, the "Left and Right Leaf F and All I" model tended to predict better for the larger training sets while the "Leaf F and All I" model tended to predict better for the smaller training sets. This can be understood from the fact that the "Left and Right Leaf F and All I" model has a lower weighted average relative standard deviation and can thus better distinguish formulas with different running times, but when the training data is small generalization may not occur as easily with this model.

The "Leaf F" and "Left and Right Leaf F" models both perform significantly worse than all of the other models at predicting the faster of two formulas. This is not surprising, given that these two models had much larger weighted average relative standard deviations.

# 6 Conclusions and Future Work

Through the use of rewrite rules and a formula generator, it is possible to automatically generate different but mathematically equivalent formulas in a principled way. These formulas can then be described with various sets of simple features which can reasonably partition the space of formulas into groups with close running times. Finally, a function approximator can learn to accurately predict the running time of a formula given a limited set of training data.

We are currently pursuing several lines of research that build upon the work presented in this paper, including:

- Determining how well a function approximator can interpolate and extrapolate to different size $F_n$'s. The results presented here all were for Cooley-Tukey expansions of $F_{128}$. If a function approximator was presented with Cooley-Tukey expansions of $F_{128}$ and $F_{512}$, could it predict well for Cooley-Tukey expansions of $F_{256}$ or $F_{1024}$?
- Investigating other feature spaces. The features described in this paper certainly are not the only ones that could be chosen.

- Investigating learning across machines and compilers. Could a function approximator learn to predict running times for particular machine or compilers, given some appropriate features of the machine and compiler.
- Investigating other factorizations of the FFT and other signal processing algorithms.
- Finding a solution to the problem that there are an extremely large number of possible formulas representing signal processing algorithms. In particular, it is not feasible to exhaustively generate all possible formulas for large transforms. Instead, we are developing heuristic methods for searching the space of formulas.

## Acknowledgements

We would especially like to thank Jeremy Johnson, José Moura, and Markus Püschel for their many helpful discussions on this research.

## References

[1] L. Auslander, Jeremy R. Johnson, and R. W. Johnson. Automatic implementation of FFT algorithms. Technical Report 96-01, Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA, June 1996.

[2] S. Egner. *Zur Algorithmischen Zerlegungstheorie Linearer Transformationen mit Symmetrie.* PhD thesis, University of Karlsruhe, Germany, 1997.

[3] J. M. F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, and M. M. Veloso. SPIRAL: Portable Library of Optimized Signal Processing Algorithms, 1998. `http://www.ece.cmu.edu/~spiral/`.

[4] K. R. Rao and P. Yip. *Discrete Cosine Transform.* Academic Press, 1990.

[5] R. Tolimieri, M. An, and C. Lu. *Algorithms for Discrete Fourier Transforms and Convolution.* Springer, 2nd edition, 1997.

| Features | Train | Test | Average Percent Error on Predicting Cost | Percent Mistakes on Predicting Faster of Two |
|---|---|---|---|---|
| Leaf F | 100% | 100% | 0.6% | 18.0% |
| | 75% | 25% | 0.8% | 18.3% |
| | 50% | 50% | 0.8% | 18.8% |
| | 25% | 75% | 0.9% | 19.0% |
| | 10% | 90% | 1.4% | 19.4% |
| All I | 100% | 100% | 0.4% | 0.7% |
| | 75% | 25% | 0.7% | 1.0% |
| | 50% | 50% | 0.9% | 1.3% |
| | 25% | 75% | 1.2% | 3.1% |
| | 10% | 90% | 2.0% | 8.1% |
| Leaf F and All I | 100% | 100% | 0.1% | 0.0% |
| | 75% | 25% | 0.2% | 0.1% |
| | 50% | 50% | 0.3% | 0.2% |
| | 25% | 75% | 0.6% | 1.4% |
| | 10% | 90% | 1.2% | 3.8% |
| Left/Right Leaf F | 100% | 100% | 0.5% | 15.9% |
| | 75% | 25% | 0.8% | 17.3% |
| | 50% | 50% | 0.8% | 17.3% |
| | 25% | 75% | 0.9% | 17.1% |
| | 10% | 90% | 1.5% | 19.0% |
| Left/Right All I | 100% | 100% | 0.2% | 0.6% |
| | 75% | 25% | 0.8% | 1.6% |
| | 50% | 50% | 1.0% | 2.3% |
| | 25% | 75% | 1.8% | 6.0% |
| | 10% | 90% | 3.7% | 12.8% |
| Left/Right Leaf F and All I | 100% | 100% | 0.1% | 0.2% |
| | 75% | 25% | 0.5% | 0.5% |
| | 50% | 50% | 0.6% | 1.2% |
| | 25% | 75% | 1.2% | 2.6% |
| | 10% | 90% | 2.9% | 8.4% |

Table 5: Prediction Accuracy for Model Data

| Features | Train | Test | Average Percent Error on Predicting Cost | Percent Mistakes on Predicting Faster of Two |
|---|---|---|---|---|
| Leaf F | 100% | 100% | 10.1% | 31.3% |
| | 75% | 25% | 10.0% | 31.2% |
| | 50% | 50% | 9.2% | 32.1% |
| | 25% | 75% | 8.1% | 31.2% |
| | 10% | 90% | 9.2% | 32.3% |
| All I | 100% | 100% | 2.1% | 8.3% |
| | 75% | 25% | 2.2% | 8.0% |
| | 50% | 50% | 2.5% | 8.7% |
| | 25% | 75% | 4.7% | 13.5% |
| | 10% | 90% | 11.6% | 19.7% |
| Leaf F and All I | 100% | 100% | 1.8% | 7.8% |
| | 75% | 25% | 1.9% | 7.8% |
| | 50% | 50% | 2.0% | 8.2% |
| | 25% | 75% | 2.3% | 8.7% |
| | 10% | 90% | 3.9% | 9.7% |
| Left/Right Leaf F | 100% | 100% | 6.0% | 28.7% |
| | 75% | 25% | 6.8% | 30.3% |
| | 50% | 50% | 6.5% | 29.5% |
| | 25% | 75% | 6.9% | 29.6% |
| | 10% | 90% | 8.6% | 33.2% |
| Left/Right All I | 100% | 100% | 1.1% | 6.0% |
| | 75% | 25% | 2.7% | 8.1% |
| | 50% | 50% | 4.1% | 9.1% |
| | 25% | 75% | 9.7% | 14.1% |
| | 10% | 90% | 16.3% | 21.6% |
| Left/Right Leaf F and All I | 100% | 100% | 0.7% | 4.3% |
| | 75% | 25% | 1.6% | 4.7% |
| | 50% | 50% | 1.6% | 5.3% |
| | 25% | 75% | 3.2% | 6.5% |
| | 10% | 90% | 8.1% | 11.7% |

Table 6: Prediction Accuracy for Real Data