Faster Real-Time Classification Using Compilation

[Final Report]

Gilbert Bernstein Morgan Dixon Amit Levy

University of Washington
Seattle, WA
{gilbazoid,mdixon,levya}@cs.washington.edu

ABSTRACT

We introduce new methods for optimizing the performance of machine learning classifiers. Building from the property that prediction algorithms *interpret* a model output by a learner, we explore optimizations that can be made during prediction by replacing the interpreter with a compiler. We explore this idea in the context of decision-tree classifiers and use our findings to discuss potential optimizations that could be made in the context of other models. We validate our optimizations on predictors that are used by several classes of applications, including a decision tree classifier used in Prefab, a research project lead by one of the authors.

General Terms

Compilers, Machine Learning

1. INTRODUCTION

Machine Learning has become an extremely effective strategy for solving complex problems [6]. While traditional programs are manually specified, the machine learning pipeline automates some of the difficult components of this process. For example, consider the task of determining whether an image contains a human face. Without machine learning, humans manually program the computer to make this decision, which could be impossible. In supervised machine learning, humans instead program a "learner" and a "predictor," both of which are easier to implement than the manually specified program. To program a learner, humans provide a set of relevant *features* that can be extracted from *training examples* and code that will automatically use these features to make a decision. This data structure is called the *model*. A decision tree learner, for example, is a learner that constructs a decision tree using attributes from training examples. To program a predictor, humans program an *interpreter* that will interpret the learned model to provide labels for unlabeled data. This is much easier to program because determining the structure of the decision-making algorithm has been automated and only requires humans to provide labeled training examples to the learner.

Much of the work in optimizing supervised machine learning pipelines focuses on improving prediction accuracy and speeding up the performance of the learner. This is because the predictors are typically not considered to be algorithmically complex compared to the learning phase of the pipeline and they are not considered to be a bottleneck. For example, the runtime of a decision tree classifier is at worse linear in the number of variables in the input data, while learning a decision tree can be quadratic in the number of variables in the training examples. Is is likely that more time is spent in the learning phase than in the classification phase for many applications. However, consider the class of machine learning applications that require real-time prediction of large amounts of data, such as real-time computer vision, or require classification of extremely large corpora of data, such as web site bounce-rate prediction. For such applications, even relatively small improvements in the performance of the predictor can have a large impact in terms of resource cost or usability to end-users. Additionally, for these types of problems, it is typical that the size of the unlabeled input during the prediction phase is orders of magnitude greater than the number of labeled training examples provided during the learning phase. In this work, we propose to explore the optimization of supervised machine learning classifiers.

Building upon the insight that a predictor is in essence an interpreter, **we hypothesize that reasonable gains in performance can be obtained by compiling the predictor along with the output from the learner**. At a high level, we believe that compiling the predictor and the learned model can provide performance gains for several reasons. First compiling the predictor introduces a potential for specialization and inlining. For example, decision tree classifiers may be compiled to a series of nested if-else statements, where specialized optimizations can be made for each node in the tree. Compilation also enables object fields to be embedded in the program as constants. This reduces the overhead for computing an offset to obtain a field value from an object. Similarly, compilation reduces overhead created by dynamic dispatch. In graphical models, for example, nodes are often objects that implement some interface. Traversing a graph and executing each node's implementation of a function requires many dynamic dispatch operations. This is common in models such as Bayes networks, decision trees, and Hidden Markov models [6].

As an initial exploration of this hypothesis, we will specifically examine decision tree classifiers within two domains, *pixel-based methods for reverse engineering graphical interfaces* and *Weka ID3 classification of diverse machine learning datasets.* We believe decision tree models are ideal for an initial exploration of this hypothesis for several reasons. First, decision trees are frequently used and improving their performance would be beneficial. In bounce-rate prediction task, for example, reducing the prediction time by 5

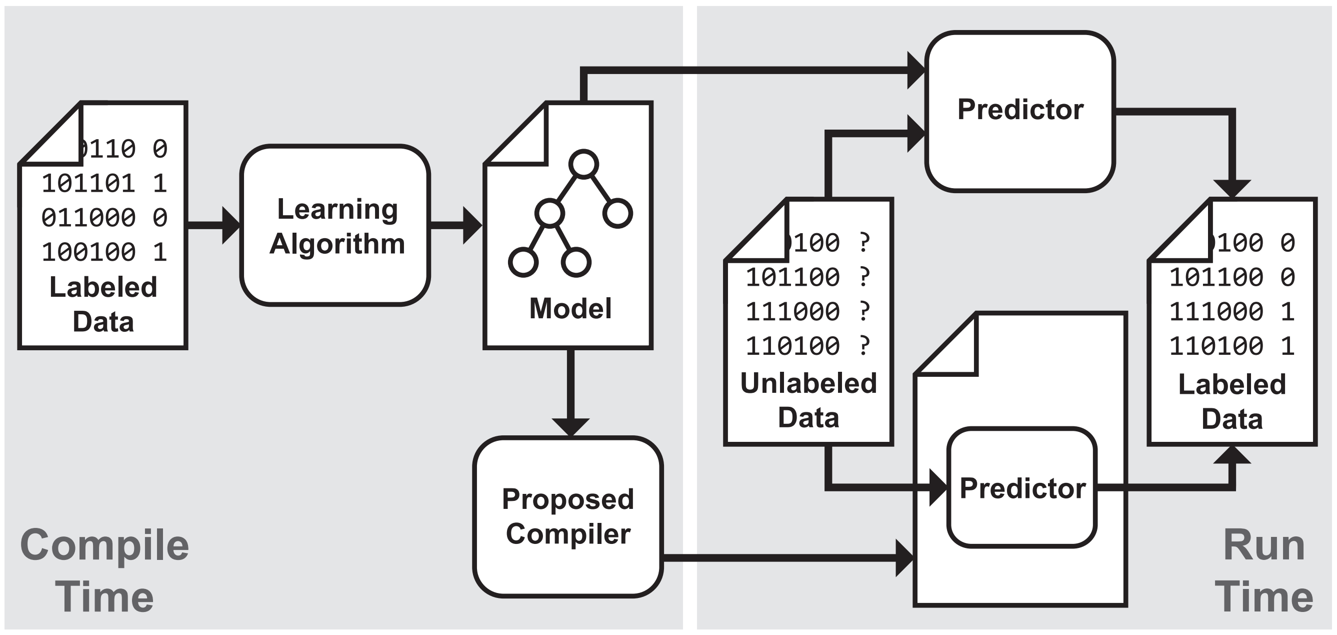

Figure 1: **The Machine Learning Pipeline:** The top path through this diagram depicts the traditional machine learning pipeline. Given labeled data, a model is produced using a learning algorithm. Then this model, along with unlabeled data are fed into a predictor in order to predict labels. We propose modifying the pipeline to take the bottom path. Since the same model is run over and over again on different unlabeled data, we can compile it into a specialized predictor.

The specific contributions of this work are:

1. A novel approach to optimizing the performance of machine learning classifiers through compiliation
2. Performance improvements on decision tree classification by as much as a factor of three.
3. A realistic evaluation of our approach on a realistic research prototype that directly benefits from the performance of decision tree classification.
4. A diverse evaluation of our approach using a standard machine learning library using a popular decision tree model.

2. RELATED WORK

Decision trees are a popular machine learning model. They are conceptually simple, easy to implement and human-interpretable. For instance, Google uses large decision trees (∼7000-12000 nodes) as one step of their speech processing in voicemail transcription and search by voice features[4]. Because of their simplicity, decision trees are also frequently chosen as base models for ensemble methods. These methods train a collection of smaller decision trees to make a joint decision by averaging or voting on their outputs. Ensemble decision trees are widely used, e.g. in computer vision[3] and for predicting advertiser churn[8]. In all three of these examples, decision trees are either run on web-scale data (e.g. every Google phone user's voicemail) or tens of thousands of times per image, with potential real-time constraints for some applications.

Surprisingly, there has been relatively little previous work on accelerating classification speed. Chen[1] considers the problem of compiling a decision tree for speech processing into a finite state transducer. However, he is not concerned with the resulting run-time of the resulting transducer. Mulvaney and Phatak[7] describe a method to merge an ensemble of decision trees into a single decision tree. Rather than accelerate classification time, they aim to produce more human-interpretable models. Lowd and Domingos[5] look at the problem of compiling graphical models (such as Bayesian networks) into arithmetic circuits. They use the size of the resulting circuits as a learning penalty, but also note a dramatic classification time speed up due to their compilation.

3. ARCHITECTURE

Decision trees classify data by traversing a tree, making branching decisions at each node in the tree based on the data in question. Leaf nodes in the decision tree contain labeling information, so data labeling happens as soon as a leaf node is reached. Conceptually, this process is similar to how an abstract syntax tree interpreter executes a program. This insight leads to the intuition that classification with decision trees can be done more efficiently by first converting the tree into sequential code.

We chose decision trees for their simplicity and relatively constant structure. Each inner node of the tree performs a test on the input data and recurses to a child node accordingly. In turn, the child node performs a similar computation if it is an inner node, or returns a label if it is a leaf node. Prefab uses a slight extension to this basic model for traversing the tree, which we discuss in section 4.2. We translate this computation into nested conditional clauses:

```
function classify(String[] data) {
  if (data[0] == "Hello") {
    if (data[3] == null) {
      return INFORMAL_SALUTATION;
    } else if(data[2] == "Mr.") {
      if (data[4] == null) {
        return FORMAL_SALUTATION;
      } else {
        return INTRODUCTION;
      }
    } else {
      return HELLO;
    }
  } else if (data[0] == "Don't") {
    ...
  }
   else {
    return UNCLASSIFIED;
  }
}
```

Figure 2: Structure of generated decision tree code

each node becomes a series of *if-else* statement where each body is either another such conditional (for inner nodes) or simply a return statement (for leaf nodes). Figure 2 gives an example of the code structure our system generates for a simple decision tree.

Our design extends existing decision tree learning frameworks by adding a generic compilation path that generates code to traverse three as well as encapsulate its structure. We design our system to work as follows:

1. A programmer writes code to generate a decision tree from labeled data using an existing machine learning framework, such as Weka. For this step, the programmer can choose any learning algorithm that results in a decision tree (e.g, Id3 or Best First).

2. Instead of classifying unlabeled data directly on the resulting tree, the programmer first passes the tree through our compiler. This steps generates a separate executable encoding the structure and logic of their tree as described above.

3. Finally, the programmer invokes the generated executable, passing in her unlabeled data. Running the compiled executable results in *exactly* the same output as classifying with the original tree.

4. IMPLEMENTATION

We implemented two prototypes of our system: one for the popular Java machine learning framework Weka, and another for Prefab. We chose to implement the prototypes in C# because Prefab is written in C# and compiled to .NET bytecode, and Weka can be easily compiled to .NET bytecode as well using IKVM. Moreover, C# has a convenient interface for generating callable bytecode dynamically, which made implementation and evaluation simpler.

For both of our implementations, we extended the existing decision tree framework by adding a *compile* method to the nodes of the tree. The end-user constructs a tree using normal learning process for that framework. Instead of directly classifying unlabeled data, the user invokes the compiler by calling the *compile* method on the root node of the tree. The output of this method is a dynamic library containing a single static method which takes one unlabeled data instance as an argument and returns a label. To facilitate convenient comparison between the performance of compiled versions of decision trees and the performance of the unmodified frameworks, we directly called the generated library function. However, it is trivial to output this library into a .NET *DLL* and use it from another application entirely.

Traverse at compile time and output MSIL switch statements.

4.1 Weka

Weka is a popular, open-source machine learning framework for Java. It ships with several decision tree learners as well as learners for other classifiers such as rule based classifiers and Bayes models. We chose to modify the Weka's Id3 tree to perform our experiments. Our modifications, however, are not specific to Id3 and could be trivially replicated for similar classifiers. Alternatively, it would be simple to expose the necessary components of these trees (e.g., the successor list) by forcing them to implement a common interface, and writing a single compile method for all of them.

To make implementation of out compiler convenient, we first used IKVM to generate .NET bytecode for the Weka Java source. This allowed us to interface with the Weka trees in C#. We then extended the Weka Id3 tree in a C# class, adding a recursive compile method, and use this subclass in the learning phase to construct an Id3 tree. At is a leaf node of the tree, the output of the method is bytecode removes any arguments from the stack, pushes corresponding label onto the stack and returns. At an inner node, the output is bytecode that retrieves an index to the successor list based on the attribute the node uses to split the data, and switches into code for that successor. Figure 3 lists the code we added to Weka's Id3 nodes.

4.2 Prefab

Prefab is a system for implementing custom user interface behaviors across arbitrary graphical user interfaces. To provide this functionality, Prefab performs online classification of graphical components at the pixel level based on a large corpus of learned GUI components. Each change in the appearance of an interface is analyzed online by Prefab. Before online analysis, Prefab decomposes training examples of widgets into an arrangement of small patterns of pixels. Some of these decomposed parts are used as *features*. A feature stores an exact patch of pixels (exact colors in a spatial arrangement of an exact size). To interpret the structure of an interface from an image, Prefab first searches for the provided features from a library of prototypes. Prefab then examines the surrounding pixels of the features to identify interface elements. In this work, we are concerned with improving the performance of feature detection. Currently in Prefab, a classification is done on each pixel in the changed

```
public void compile ()
{
  // Stack: bottom: [Instance]
  if (this.attribute == null) { // Leaf
    Emit(OpCodes.Pop);
    // Stack: bottom: []
    Emit (OpCodes.Ldc_R4, this.classValue);
    // Stack: bottom: [classValue]
    Emit(OpCodes.Ret);
  } else {
  // Inner node
    Emit(OpCodes.Dup);
    // Stack: bottom: [Instance, Instance]
    Emit(OpCodes.Ldc_I4, this.attribute.index());
    // Stack: bottom: [Instance, Instance, index]
    MethodInfo valueMethod = ...;
    Emit(OpCodes.Callvirt, valueMethod);
    // Stack: bottom: [Instance, result_idx]
    Label[] labels = new Label[this.successors.Length];
    for (int i = 0; i < this.successors.Length; ++i)
    {
      labels[i] = DefineLabel();
    }
    Emit(OpCodes.Switch, labels);
    // Stack: bottom: [Instance]

    for (int i = 0; i < this.successors.Length; ++i)
    {
      MarkLabel(labels[i]);
      this.successors[i].compile();
    }
  }
}
```

Figure 3: The compile method for Weka's Id3 nodes.

region to determine if it contains a feature. Using a relatively small corpus of widgets and with high CPU load, Prefab typically processes a frame change under 50ms. While the current process is adequitely fast for detecting small regions or regions that contain few features, large, rapidly changing portions of an interface (such as scrolling regions) that contain a large number of features suffer. This may take over 100ms. For some applications that require fast, real-time tracking of many interface elements, this could present noticeable lag. Improving the performance of Prefab's feature detection is an important step to providing sufficient performance for future applications. This improvement in performance may also help in providing sufficient performance for larger resolution interfaces. For example, Prefab is currently limited to relatively small windows, but many interesting applications may span multiple large windows or the entire desktop.

Prefab uses a slightly modified model of a decision tree to perform feature detection. When a library of prototypes is created, Prefb chooses a non-transparent hotspot within the patch of pixels defining each feature in the library. Prefab constructs a decision tree for determining whether a pixel in an image is the hotspot of any feature in the tree, as in Figure 4. Each internal node specifies an offset relative to the hotspot, each edge corresponds to the color at that offset, and each leaf corresponds to a feature. Traversing the tree to a leaf tests every pixel in the figure. In addition to branching to a single child at each inner node, Prefab also branches on a "transparent" node. Each transparent node corresponds to a pixel that should be ignored by a feature. Each node has at most one transparent child. This extension departs from our generic implementation for decision trees, but requires only a simple addition to our system. Instead of one set of nested conditionals, for Prefab we output two sets: one for normal branching, and a similarly structured one to follow the transparent path.

Our implementation recursively traverses a given Prefab decision tree. Each inner node computes an offset relative to the current offset being examined in the image. The node then observes the pixel value of the image at this offset and uses this value to potentially branch on a child. Prefab uses a C# implementation of a hash table to efficiently determine if the observed pixel value corresponds to a child or not. We compiled our own simple hash function that directly maps the pixel values of child nodes to indices in a jump table. This allows us to compute a hash value of a pixel and switch on that value. Our hash function simply computes the observed pixel value modulo a prime number. Collisions in a hash table are resolved after the jump instruction has been performed, where the pixel value corresponding to each potential child node is directly evaluated against the observed pixel value in a series of if-else statements. If none of the corresponding children match the observed pixel, the program immediately recurses on the transparent child of the current node. Otherwise, the program recurses on the corresponding child and then recurses on the transparent node. The relative offset and the pixel values to compare against are all hard coded as constant values. Prefab leaf nodes construct an object with fields corresponding to an ID mapping to a feature in the given feature library, the offset of the identified feature occurrence, and it's bounding region. All of

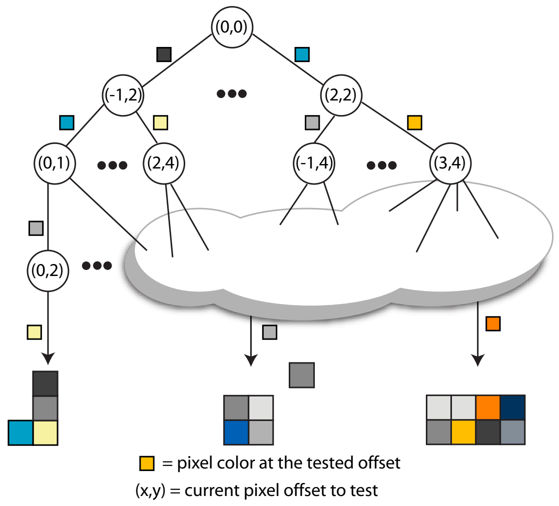
(0,0)
(-1,2)
(2,2)
(0,1)
(2,4)
(-1,4)
(3,4)
(0,2)
= pixel color at the tested offset
(x,y) = current pixel offset to test

Figure 4: Prefab constructs a decision tree that tests whether a pixel is the hotspot of a feature from the prototype library. It uses this tree to scan an image of an interface, detecting all features in a single pass.

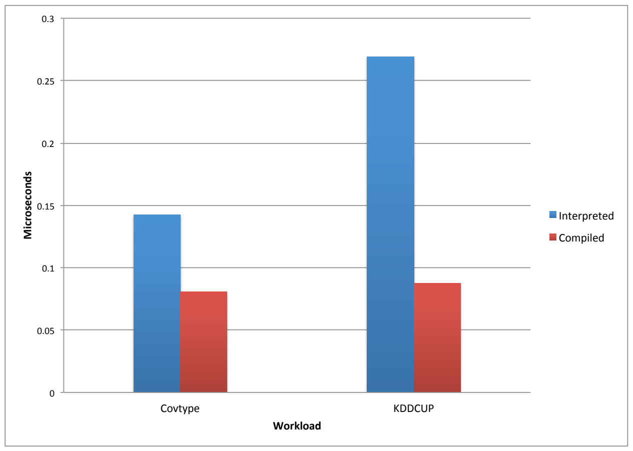

Figure 5: Average classification time for a single data instance in compiled and uncompiled versions of Weka Id3 tree.

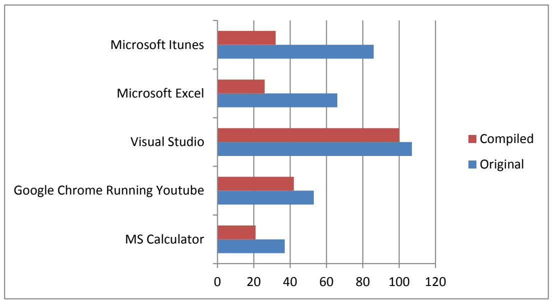

Figure 6: Prefab results.

these parameters are compiled as constants as well. Finally, each node passes a label to the next execution point during our traversal. This allows inner nodes return control to transparent nodes after a child has been traversed and exit the function after transparent children have been traversed.

5. EVALUATION

5.1 Weka

In order to test WEKA, we downloaded two datasets from the UCI machine learning repository[2]: Covertype and KDD Cup 1999. Both datasets were chosen because they are (a) large and (b) come with well defined classification tasks. The Covertype dataset was originally used to predict the type of ground cover on a 30×30 meter plot of land, using only cartographic variables. It contains 581,012 instances. The KDD Cup 1999 consists of 4,000,000 instances of user connection records used to detect system intrusions. The dataset was compiled for a data mining competition.

We evaluated our Id3 decision tree compiler by classifying a large number of test data instances for each workload. We used 11,340 designated instances from the Covertype dataset as a training set and the first 100,000 provided testing instances. We used the first 50,000 entries in the KDD Cup dataset as a training dataset and the subsequent 100,000 entries for testing. All the data was loaded into memory first to avoid incurring the performance cost of streaming data from disk.

To avoid costs associated with the IKVM library, we performed this experiment both in C# and directly in Java for the uncompiled tree. Since Java performance was uniformly better, we used those results for our comparison. We measured the wall time of classifying all the data instances 100 times. The compiled trees performed much better on both workloads. Figure 5 shows the different in classification time between compiled trees and normal Weka. In both cases the compiled trees perform more than twice as fast, and in the KDD Cup case, the compiled tree performed over three times faster.

5.2 Prefab

We evaluated the performance of Prefab's decision tree classifier over a corpus of captured video from a Windows 7 machine. The videos were taken from 5 common Windows 7 applications and vary in resolution from 228 by 322 pixels to 1382 by 1079 pixels in resolution. Each capture contains roughly 2 minutes of footage at a frame rate of 100 frames per second. Only the changed portions of the interface were inspected for features and we only computed the duration for times where some pixels changed between frames. Figure 6 shows a table of each application and the average performances for our system and the original Prefab system. The decision tree model used for this evaluation consists of 297 nodes from a library of 163 features. The features come from a prototype library that was built specifically for these particular applications. The authors of Prefab have noted that one common use of the system may be to construct or download prototypes chosen for specific target applications, so this approach is consistent with their vision. The decision tree has a depth of 5. Prefab decision trees have a wide branching factor and a shallow depth, because Prefab intentionally tries to reduce the height of the tree during construction to minimize the number of pixels to be evaluated during runtime. The applications and window sizes were chosen to be a representative set of applications commonly used on Windows 7 machine on a 23 inch display. Each video contains footage of constant interactions from a user (users were instructed to manipulate the widgets that they identify). The average number of pixels explored per frame was 1152 pixels.

On average our compiled implementation of the tree ran 1.9 times faster per frame. Our goal was to evaluate the performance of a compiled decision tree obtained from a real application running on realistic data. These results show promise in this approach, but there could be several reasons why the performance gains seem quite high. First, the attributes tested in Prefab's decision trees are very simple. They only require an integer comparison. In other common decision trees, the cost of testing an attribute could be more expensive. This cost could potentially mask some of the performance gains introduced by compilation. Additionally, Prefab is implemented in a language that uses a JIT. This could be using hot paths to highly optimize performance. For languages that do not use a JIT, it may be that optimizations would not be made to the compiled code. We believe evaluations of performance in the context of other applications is an interesting area for future work.

The authors of Prefab have been discussing an entirely automated approach to Prefab where larger prototype libraries are used rather than allowing developers choose a prototype library. Additionally, we suspected that as the size of our trees increase, the code size of our compiled trees could be detrimental to performance because of instruction cache misses. We tested a much larger decision tree with 1,977 nodes on the internet explorer and Microsoft Word videos to get a sense of whether this might be true. This in fact seems to be the case. We do not present a detailed analysis here, but the performance was on average a factor of two times worse than the original Prefab tree. We present a more fine-grained evaluation of this problem below.

5.3 Tree Size

Our results evaluating Prefab implies some relationship between the size of the learned tree and the relative performance of compiled versus uncompiled classification. In particular, while for small tree heights a compiled tree performed classification about as fast or faster than the normal Prefab code, compiled trees performed much worse for large heights. We explored this relationship by implementing a random binary tree generator and compiler and feeding it random data.

The tree generator chooses a random number K to *split* on at each node. Inner nodes test if K is a factor of the data (a random integer), and branches left or right accordingly. Each generated tree is perfect – that is, the tree is full and all leaves are at the same depth. We evaluated the performance at different heights by generating a random tree, choosing 100,000 random data instances, measuring the classification

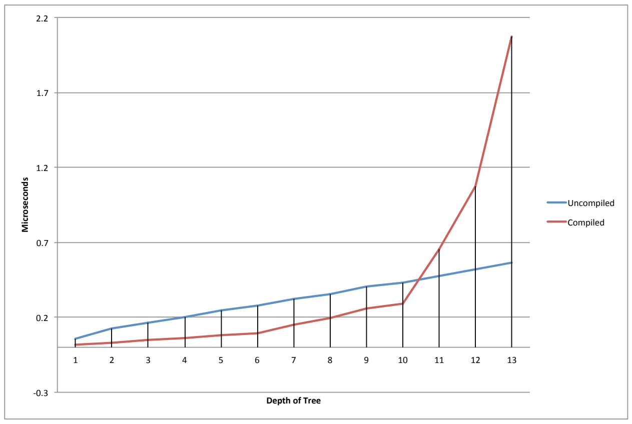

Figure 7: Average classification times for randomly generated decision trees

speed of normally traversing the tree, and finally measuring the performance of passing data to a compiled version of the tree. Figure 7 shows our initial results for trees of height 1 through 13.

As seen in the graph, performance of compiled versions of this simple binary tree is much better for short trees. However, the trend is exponential with the height of the tree, and therefore linear with the number of nodes in the tree. This is surprising since each classification traverses $log(n)$ nodes of the tree, so performance should be linear in the height, and sublinear in the number of nodes. We hypothesized that the .NET JIT compiler is responsible for part of this performance degradation. Since code size of the compiled tree is linear with the number of nodes in the tree, if the JIT kicks in sometime during our evaluation it is likely to begin have a dominating effect on performance when tree sizes are large. To test this we performed a similar experiment, but ran the experiment on the compiled and uncompiled trees twice, only reporting the second trial. The result was a linear growth in performance with respect to the height of the tree. Figrure 8 shows the results of this experiment. We also ran a similar experiment timing both runs together, and averaging the times. The results for this experiment were essentially identical, probably because the the cost of JIT optimizations was amortized over enough runs.

6. DISCUSSION

Our results were mixed. While we observed uniformly faster runtimes from compiling smaller trees, the compiled versions of the larger Prefab trees were up to 2 times slower than their uncompiled counterparts. Consequently, we conclude that compilation is a reliable optimization strategy for relatively small decision trees, but unreliable when applied to relatively large decision trees.

Compilation is necessary to enable a number of optimizations that we benefit from. Fundamentally, these optimizations are a result of "unrolling" the tree, allowing us to specialize the code for individual nodes. First, we can eliminate dynamic dispatches (i.e. on different internal node types) and base case tests (i.e. "Are we at a leaf yet?"). Second, constants can be baked into the instructions now, avoiding

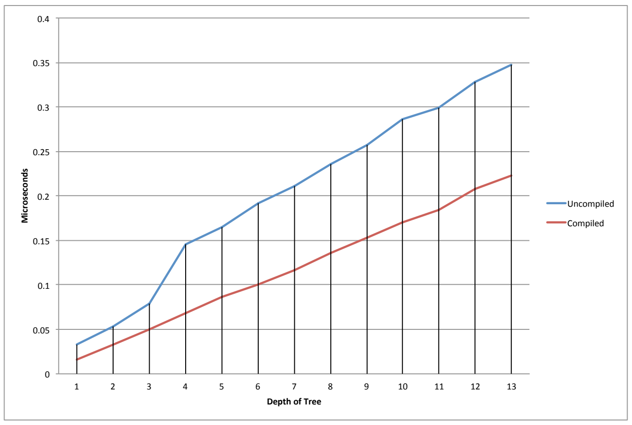

Figure 8: Average classification times for randomly generated decision trees after pre-optimizations

an extra memory load/pointer dereference. Third, now that decision tree is laid out in code, both the JIT compiler and hardware branch prediction can optimize the code based on likely traversals. So long as the tree remains in memory, processed by a single "generic" recursive function, these runtime optimizations will be blocked.

Unfortunately, we do not know precisely why the large Prefab trees have poorer performance. One hint might come from the relative performance between the Visual Studio and Excel benchmarks. On average, Visual Studio contains 1662 features per frame while Excel contains only 37 features per frame. Since each feature corresponds to a unique leaf of the decision tree, this means that the Visual Studio benchmark is exercising a much larger percentage of the decision tree than the Excel benchmark. As a result, we would expect the hot path optimizations in the JIT and instruction caching to be comparatively more effective on the Excel benchmark. This poses an interesting challenge for future investigation: Given a large decision tree, how can we best exploit the reliable performance gains from compiling small decision trees?

7. FUTURE WORK

Although we presented evidence that converting a decision tree into specialized code can yield speedups, we would like to better characterize when a speedup can be expected. To this end it would be good to explore the effect of function size/granularity by partitioning the tree and generating multiple functions instead of one monolithic function. It would also be good to explore the relative benefit of compiling decision trees in a statically rather than dynamically compiled language such as C or C++, especially since optimized implementations are more likely to target those languages. These investigations would help confirm our observations.

Building on this work, it would be interesting to build optimizations into our decision tree compiler. One such optimization might attempt to rebuild a functionally equivalent decision tree using execution-time-cost heuristics rather than the entropy/information gain heuristics that learning algorithms use. These execution-time-cost heuristics could be derived from a set of example inputs (e.g. the training data) that are assumed to be statistically representative. Alternatively, we could also explore the possibility of dynamically rebuilding the tree, guided by runtime profiling.

We also believe that it would be good to look at the benefit of compiling models beyond simple decision trees. While some models (e.g. linear/logistic regression) are unlikely to be accelerated by compilation, others (e.g. Bayesian networks[5]) are very likely to benefit. One particularly interesting avenue is to investigate is the compilation of ensemble models that aggregate the output of multiple models via voting or averaging. Models in the ensemble are very likely to perform redundant computation due to the way in which ensembles are built. This makes ensembles prime targets for common sub-expression elimination.

Finally, our prototype decision tree compilers were written with custom byte code emission. It would be good to investigate appropriate abstractions and intermediate representations. This way, we could integrate our compiler more quickly with new applications and models.

8. CONCLUSION

We have shown that performing classification on a compiled decision tree can provide meaningful performance gains. Moreover, we tackled one of the challenges associated with code explosion. Specifically we found that JIT optimizations are a significant performance drain when method sizes are large. In the future we hope to refine our approach and explore similar approaches for other classification models.

9. ACKNOWLEDGEMENTS

We would like to thank Ben Lerner for his help teaching us how to use the MSIL bytecode emitter, and Mike Ernst for his feedback on draft copies of this paper. We would also like to thank UCI for maintaining their online machine learning repository[2], from which we obtained the Covertype and KDD Cup 1999 datasets.

10. REFERENCES

[1] S. F. Chen. Compiling large-context phonetic decision trees into finite-state transducers. *EUROSPEECH*, 2003.

[2] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[3] D. Hoiem, A. A. Efros, and M. Hebert. Recovering surface layout from an image. *Int. J. Comput. Vision*, 75(1):151–172, 2007.

[4] H. Liao, C. Alberti, M. Bacchiani, and O. Siohan. Decision tree state clustering with word and syllable features. *INTERSPEECH*, 2010.

[5] D. Lowd and P. Domingos. Learning arithmetic circuits, 2008.

[6] T. Michell. *Machine Learning*. McGraw Hill, 1997.

[7] M. R and P. DS. A method to merge ensembles of bagged orboosted forced-split decision trees. 2003.

[8] S. Yoon, J. Koehler, and A. Ghobarah. Prediction of advertiser churn for google adwords. *JSM Proceedings*, 2010.