Appears in Proceedings of the First International Conference on Autonomic Computing (ICAC-04).
New York, NY. May 2004.

File classification in self-* storage systems

Michael Mesnier, Eno Thereska, Gregory R. Ganger
Carnegie Mellon University

Daniel Ellard, Margo Seltzer
Harvard University

Abstract

To tune and manage themselves, file and storage systems must understand key properties (e.g., access pattern, lifetime, size) of their various files. This paper describes how systems can automatically learn to classify the properties of files (e.g., read-only access pattern, short-lived, small in size) and predict the properties of new files, as they are created, by exploiting the strong associations between a file's properties and the names and attributes assigned to it. These associations exist, strongly but differently, in each of four real NFS environments studied. Decision tree classifiers can automatically identify and model such associations, providing prediction accuracies that often exceed 90%. Such predictions can be used to select storage policies (e.g., disk allocation schemes and replication factors) for individual files. Further, changes in associations can expose information about applications, helping autonomic system components distinguish growth from fundamental change.

1. Introduction

Self-* [1] infrastructures require a much better understanding of the applications they serve than is currently made available. They need information about user and application behavior to select and configure internal policies, reconfigure upon the addition of new users and applications, tune the system, diagnose performance dips, and in general, "be self-*." Current applications expose little or no information explicitly, making automation of decision-making and planning extremely difficult.

Despite the need, one cannot expect application writers, users, or system administrators to provide the necessary tuning information. Exposing hints has proved to be too tedious and error prone for programmers, despite years of research in the software engineering and operating systems communities. Moreover, the same complexity issues that push for self-* infrastructures preclude requiring users and administrators to generate such information. The information must be learned by the infrastructure.

This paper explores techniques for one sub-problem of this large space: helping file and storage systems automatically classify and predict the properties of files (e.g., access pattern, lifespan, and size). We build on our observation that file properties are often strongly associated with file names and attributes (e.g., owner, creation time, and permissions) [5], meaning that users and application writers are *implicitly* providing hints simply by organizing their data. These associations can be used to predict a file's properties as early as when it is created. The key, of course, is to extract and verify these associations automatically.

These implicit hints, or associations, can be used to select policies and parameter settings for each file as it is created. As one example, popular read-mostly files might be replicated several times to provide opportunities for load spreading [12] and disk service time reduction [29]. As another, large, sequentially-accessed files might be erasure-coded to more efficiently survive failures [26]. Modeling associations may also provide value on a larger scale. In particular, self-* components (or system administrators) can gain intuition into how a system is being used (e.g., which files are write-only), and changes in association over time offer a glimpse into changes in the applications being served. For example, the emergence of new associations may suggest that a substantive, long-term change in the workload has occurred, rather than a transient one. Proactively differentiating these cases may allow for a more efficient provisioning and reallocation of resources.

This paper shows that decision tree models are an accurate, efficient, and adaptable mechanism for automatically capturing the associations between file attributes and properties in a self-* system. Specifically, we use month-long traces from four real NFS environments to show that these associations can be exploited to learn classification *rules* about files (e.g., that a file ending in *.log* is likely to be write-only). These rules are used to predict file properties, often with 90% or better accuracy. The compression ratio of files to rules can be greater than 4000:1 (i.e., one classification rule for every 4000 files in a file system). In addition, we show that the decision tree models can adapt to changing workloads.

[1] Pronounced 'self-star,' this term is a play on the UNIX shell wild character '*,' encompassing self-managing, self-configuring, self-tuning, etc.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 motivates the use of decision trees for file classification. Section 4 overviews the Attribute-based Learning Environment (ABLE), our prototype file classification system. Section 5 compares training strategies for building trees and evaluates their longevity (i.e., accuracy over time). Section 6 identifies some design tradeoffs involved with a tree-based classification system. Section 7 summarizes the contributions of this paper and outlines some future work.

2. Related work

Given the importance, cost, and complexity of storage, self-* storage systems are a target for many researchers [7, 10, 27]. Storage is often responsible for 40–60% of the capital costs and 60–80% of the total cost of ownership [15] of a data center. In fact, Gartner and others have estimated the task at one administrator per 1–10 terabytes [8]. As a result, many research groups are targeting aspects of this general problem space. Our work contributes one tool needed to realize the self-* storage vision: the ability to predict file properties.

Predictive models for file caching and on-disk layout have been heavily researched. Intra-file access pattern classification learns the access pattern of blocks within a file for on-disk layout and prefetching [16, 21]. Inter-file classification identifies relationships between files for whole-file caching and prefetching [1, 14]. Additionally, per-file usage statistics have been used to optimize the layout of blocks on disk [24]. Our work complements these by learning to predict how a file will be used as soon as it is created, rather than having to first observe it in use for a period of time. This also results in building models for classes of files, rather than for each individual file. All files within a class can share the same model.

At the storage level, I/O workload characterization is a long-standing problem [6], and a difficult one, because many independent, per-application file streams are merged together and are often indistinguishable. Our work provides storage systems with additional insight, by exposing and predicting the collection of individual file patterns. In addition to making predictions at the file system level (e.g., in a file server), storage devices that understand file attributes (e.g., object-based storage [19]) will be able to make attribute-based predictions inside the storage device.

Machine learning tools developed by the AI community [3] have matured to the point where they can be efficiently integrated into real systems. The following section motivates the use of decision trees in the context of our problem.

3. Classifying files in self-* systems

A file's properties determine the best management policies for the file. Today, system administrators are responsible for selecting and configuring these policies and, because of the complexity of file and storage systems, policies are often assigned at the file system and storage level, not on individual files. Administrators therefore use rules-of-thumb for policy selection, often in anticipation of a certain workload. There are two problems with this. First, setting global policies often results in few files getting managed optimally, as a generic choice is made. Second, workloads are complex and variable, often preventing effective human configuration. However, if one could distinguish between different types of files, or *classes*, policies could be set on each individual class (e.g., store the write-only class in a volume optimized for writes).

Specifying the set of classes and policies will remain the responsibility of a system designer. Classes are chosen to distinguish the file properties that are relevant for policy assignment within an application. As examples, access patterns may be read-only or write-only; file sizes may be large or small; and lifetimes may be short-lived or long-lived. However, automatically determining the classes of newly created files and assigning polices (depending on the class) can become the responsibility of a self-* storage system.

Ideally, the classes of a file should be determinable at file creation time. If this is the case, then the appropriate policies can be assigned up-front, and resources allocated accordingly, thus reducing the need for per-file monitoring and expensive file system reorganization. However, users and applications almost never disclose the classes of a file when it is created; a system must learn them.

Fortunately, files have descriptive attributes (e.g., names, user and group IDs, permissions) available when the file is created. These attributes, in certain combinations, often indicate (implicitly) the classes of a file. The trick is to determine which combinations determine which class. For example, the file owner, application, and time the file is created may be the determining factors in whether or not a file will belong to the class of write-only files. Stated differently, because attributes are statistically associated with the properties of a file [5], they can be used to predict the classes to which a file will belong.

To make predictions, a self-* system must therefore learn to automatically classify files. In our case, the *training set* is a large collection, or sample, of files. There are two common ways to obtain a sample: from traces or from a running file system. For each file in the sample, we record its attributes and its classes (which have already been determined, by an offline analysis of the file's evolution). To simplify the problem, each classification takes on a binary value: '+' if the file is member of a class and '-' otherwise.

File class	Example policy
File size is zero	Allocate with directory
0 <size≤16KB	Use RAID1 for availability
File lifespan ≤ 1sec	Store in NVRAM
File is write-only	Store in an LFS partition
File is read-only	Aggressively replicate

Table 1. Classes we want to predict.

Attributes currently used in our models include the file's name, owner and group identifiers, the file type (directory, file, or symbolic link), permissions (read, write, execute), and the creation time (specified as morning, afternoon, evening, or night). The classes include information about the access pattern, lifetime and size properties. Specifically, we want to predict whether or not a file will belong to any of the classes shown in Table 1. These classes are representative of the kinds of classes a system administrator or application may specify, and that a self-* storage system will need to learn in order to automate policy assignment.

From our training set, we want a *classifier* (or model) to organize the files into *clusters*. A cluster contains files with similar attributes that are also from the same class. For example, the write-only class may be composed of two clusters: files ending in *.log* and files with their read permissions turned off.

Because not all attributes are relevant in determining cluster membership, the largest job of a classifier is separating out the irrelevant attributes. The classifier must therefore learn *rules* to determine what cluster a file is in, and these rules will be based on the most relevant attributes. These same rules are then used to make predictions on new files outside of the training set, sometimes referred to as a *test set*. These will be new files, actually created in the system. Using the clustering rules, the class of a new file is predicted to be the same as the class of other files in its cluster. This is often referred to as a nearest neighbor problem, because a file is predicted to be similar to its neighbors, where neighborhoods are defined by files with similar (relevant) attributes.

Depending on the number and cardinalities of the attributes used, thousands of clustering rules may be discovered from a single day of traces. For example, suppose we want to classify files based on two attributes: the group ID and user ID. If there are 3 groups and 10 users, then a model can have at most 30 rules (clusters), one for each combination of group and user. Moreover, some of the attributes are filename components [5] (e.g., the file extension), of which there are comparatively an unlimited number. Maintaining a statistic for each attribute combination should be avoided. We therefore need an efficient way to capture only those attribute combinations that matter and to create rules that generalize well to unseen examples.

3.1. Selecting the right model

Our file classification problem translates into these primary requirements that a model needs to satisfy:

- **Handling of mixed-type attributes.** File attributes can take on categorical, discrete or continuous values.
- **Handling of combinative associations.** A file's properties may depend on combinations of attributes. For example, the expected file size may depend on both the file name and creation time.

In addition, the following secondary requirements are desirable, since we plan to use the model in a real system:

- **Scalable.** The model must be able to make predictions quickly, even if this means a longer training time. In addition, making the prediction must be inexpensive in terms of computational and storage requirements.
- **Dynamic.** The model must efficiently adapt to new workloads. Furthermore, by monitoring the addition/deletion of rules, autonomic components can more easily track changes.
- **Cost-sensitive.** The model should be able to reduce the overall cost of mispredictions, by taking application-specific cost functions into consideration during training.
- **Interpretable.** Rules should be human-readable. Administrators (or self-* components) may be curious how the system is being used, and they may have additional rules and hints that they could contribute.

Many algorithms have been developed in the machine learning community. We chose to use decision trees. Trees handle mixed attributes easily, and combinations of related attributes (i.e., AND and OR) are naturally captured. Moreover, trees require minimal storage, quickly make predictions, are easy to interpret, and can learn incrementally over time [25].

We also considered algorithms such as Nearest Neighbor, Naive Bayes, and Bayesian belief networks. These models also handle mixed data types well and can learn incrementally from new samples. However, each failed to satisfy one or more of the other requirements. In particular, nearest neighbor algorithms favor training time over prediction time. Naive Bayes assumes that attributes are conditionally independent of one another, and thus violates our requirement to detect combinations of attributes. Bayesian belief networks model attribute dependence in networks, but require additional techniques to determine the structure of the network. Further, none of these other models are easy

to interpret. More complex models, such as neural nets, were not considered because of their slow training time and inability to efficiently handle categorical attributes.

4. ABLE

The Attribute-based Learning Environment (ABLE) is our prototype system for classifying files with decision trees. ABLE currently obtains file samples from NFS traces. However, the longer-term goal for this work is to obtain samples and deploy predictors within a running self-* storage system. ABLE is a crucial step toward this goal.

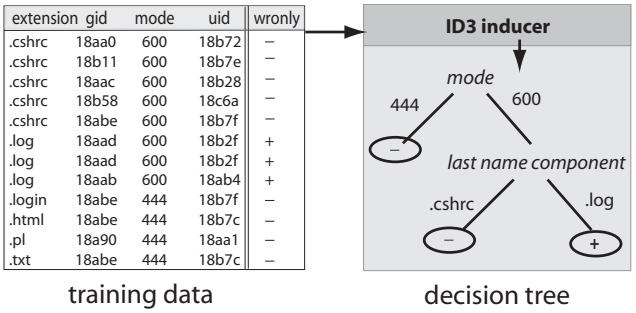
ABLE consists of three basic steps:

1. **Obtain training data from NFS traces.** For each file, ABLE records its attributes (names, UID, GID, type, permissions (UNIX mode), and creation time) and classes shown in Table 1.
2. **Induce a decision tree.** For each file class, ABLE trains a decision tree to classify each file in the training data. The result of this step is a set of predictive models, one for each class, that can be used to make predictions on newly created files.
3. **Make predictions.** We use the models to predict the classes of files in a new set of NFS traces, and then check whether the predictions are accurate.

Figure 1 illustrates the ID3 algorithm [20] ABLE uses to induce a tree from sample data. In general, a decision tree learning algorithm recursively splits the samples into clusters, where each leaf node is a cluster. The goal is to create clusters whose files have similar attributes and the same classification. The *purity* of a leaf node (the percentage of files correctly classified) is the metric used to build the tree. Attributes that produce the purest clusters are placed further up in the tree, and are ranked using a chi-squared test for association [18]. The chi-squared test determines which attributes are most associated with a given class. By splitting on the most relevant attributes, a tree can quickly find the naturally occurring clusters of files and ignore the irrelevant attributes.

The tree is thus built top-down, until either all attributes are used or all leaf nodes have reached a specified level of purity. We elaborate on the tree building process and relative strength of the attribute associations in previous work [5]. After a tree is induced, we determine the class of a file by querying the tree. The values of the file's attributes will determine its path, and ultimately direct the query to a given leaf node. The classification of the file is that of its leaf node ('+' or '-'), and is simply the most common classification of all files in that cluster.

To obtain an estimate of the true error of the model, we classify each file in the training set according to the tree and



ID3 Algorithm

1. Select attribute A as node to split on (based on relative ranking).
2. Split samples according to values they take on attribute A.
3. If leaf nodes are "pure", done.
4. Else, if attributes remaining, goto 1.

Figure 1. Example data and tree.

then compare this classification to the actual class of the file, as observed in the annotated NFS trace. To obtain the most realistic estimate of the true error, we calculate the training error using 10-fold cross-validation [13]. We can then use this estimate to automatically determine whether or not the model is suitable enough for deployment in a self-* system. However, the real measure of a model's effectiveness is its performance on unseen samples.

To demonstrate that our findings are not confined to a single workload, system, or set of users, we train and test our models on three different NFS traces from Harvard: DEAS03, EECS03, and CAMPUS [4], and one trace from Carnegie Mellon: LAB.

- **DEAS03** (February) captures a mix of research and development, administrative, and email traffic from Harvard's Division of Engineering and Applied Sciences.
- **EECS03** (February) captures a canonical engineering workstation workload from Harvard's Electrical Engineering and Computer Science department.
- **CAMPUS** (10/15/2001 to 10/28/2001) traces one of 14 file systems that hold home directories for the Harvard College and Harvard Graduate School of Arts and Sciences. The trace is almost entirely email.
- **LAB** (3/10/2003 to 3/23/2003) traces the main server used for storage research in the Parallel Data Laboratory at Carnegie Mellon University.

The single-day prediction accuracies for the properties shown in Table 1 are detailed elsewhere [5] and summarized here. For each trace, models are trained on a Monday and tested on a Tuesday. The baseline for comparison is a MODE predictor, that simply predicts the most common classification for a property. For example, if most files on Monday are read-only, then MODE will predict read-only

for every file on Tuesday. MODE is therefore the equivalent of a learning algorithm that places all files into a single cluster, as opposed to a decision tree which has one cluster per leaf node, distinguished by their common attributes.

In nearly all cases, our previous work shows that ABLE much more accurately predicts the classes of files, relative to MODE. The prediction accuracies for the size and lifetime classes are quite accurate (90-100%), especially when compared to MODE which is often no better than random guessing (i.e., 50%). The write-only and read-only predictions show more variability (70-90%), but again are much better than MODE. There are a few cases, such as for the CAMPUS trace, where the workload is so skewed that MODE performs about the same as ABLE. Barring these, the benefits of clustering files by their attributes is clear.

Although these results are encouraging, to be of practical use in a self-* system, we must also be able to answer questions related to the longevity of the model. In particular, we must have robust prediction accuracies over time and retraining strategies for when prediction accuracies deteriorate. We expect that a model's rule set will converge to a small working set over time and that changes in the rule set can be used to help describe changes in the workload. This will contribute to techniques for automatically adapting to changing workloads.

5. Model longevity

Storage systems experience a variety of change in their workload: users come and go, new applications are installed, and different seasons exist (e.g., tax season in an accounting firm). Self-* storage systems must learn to adapt to these changes accurately and efficiently.

Adapting to changing workloads, in our case, requires discovering new associations between attributes and classes; that is, discovering or changing the clustering of files. Recall that the leaf nodes of the decision tree are the clusters to which we refer, and a file's cluster determines its class.

We distinguish two kinds of change: additive and contradictory. Additive changes result in new rules that are added to a given tree, and are most commonly seen when a tree is still learning the workload of a system or when the workload changes. Contradictory changes produce rules in opposition to an already existing set of rules and signify a fundamental change in user or application behavior. However, we found no significant occurrence of contradictory change in the traces. We do find contradictory rules, but most occur with rules that hover around 50% confidence. For example, there are many cases where about half of the files in a cluster are read-only and half are not. The classification rule for this cluster can therefore easily flip given a few positive or negative training examples. In addition, changes can either be transitory (lasting only a few days), or long-term.

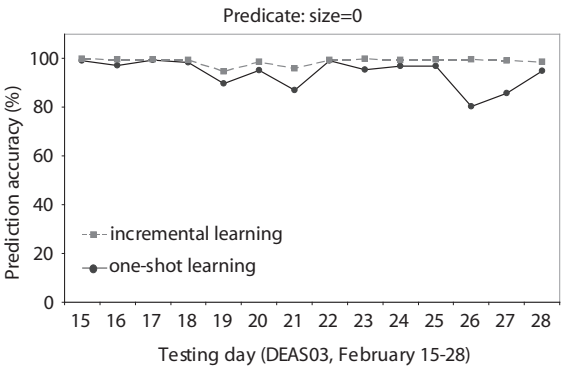


Figure 2. By adding rules dynamically, an incremental learner can often reduce the prediction errors seen by a one-shot learner.

Additive change is quite common in our NFS traces. User activity varies by day, and consequently so does the workload. This can be seen by the popularity of certain files, or the manner in which they are accessed. As such, a learning model that is built from a single day of activity may not be representative of the true workload. In other words, it may take more than a day of traffic to train a decision tree.

This section shows that the decision tree classifiers use for our problem efficiently adapt to new workloads and continue to produce accurate predictions over the entire duration of our traces (two weeks to one month). For comparison, we present the results of two different training strategies: incremental and one-shot. Incremental learners cumulatively train on all days. One-shot learners train on only the first day of traffic in each of the traces.

5.1. Benefits of incremental learning

Unlike one-shot learning, which trains a model from a fixed sample, incremental learning dynamically refines a model with new samples as they become available [25]. The benefits are that it is unnecessary to determine a fixed training window and that the model naturally adjusts to slow changes over time. To quantify these benefits, we compare the accuracies of these two training techniques on the entire trace period for each of our traces, and illustrate specific examples from DEAS03, our most diverse trace.

Figure 2 shows an example using the *size=0* class. The graph illustrates how an incremental learner is, in general, more stable than a one-shot learner. Although a one-shot learner may predict well at times, there are often cases (e.g., Feb. 26-27) where missing rules result in a significant degradation in prediction accuracy. By refining its rule set, the incremental learner avoids the errors completely.

	DEAS03		EECS03		CAMPUS		LAB	
Predicate	ONCE	INCR	ONCE	INCR	ONCE	INCR	ONCE	INCR
size=0	94%	95%	94%	97%	99%	99%	95%	96%
0<size≤16K	90%	91%	81%	88%	99%	99%	**84%**	**89%**
lifespan≤1s	83%	85%	87%	94%	76%	78%	91%	93%
write-only	**75%**	**85%**	68%	79%	84%	89%	83%	84%
read-only	62%	67%	**52%**	**69%**	85%	86%	72%	74%

Table 2. Monthly accuracies for incremental (INCR) and one-shot (ONCE) learning.

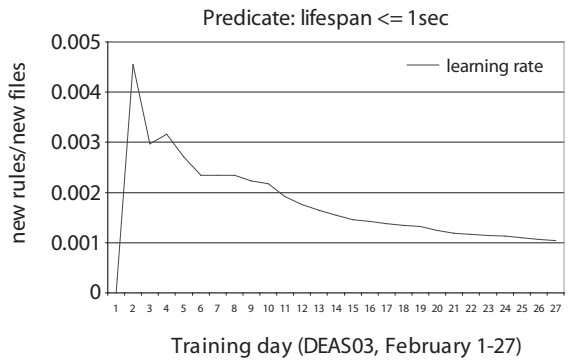


Figure 3. The learning rate shows that most of the learning occurs in the first few days of training, and that the model eventually converges to a working set of rules.

5.2. Learning rate

We define the *learning rate* of a model as the ratio of new rules to files on a given day. Models still learning the workload of a system will thus experience higher learning rates than models that already have a well-established rule set. We can use this knowledge to determine when a model has passed the initial learning phase. It also tells us when there has been a large additive change in the rule set.

Figure 3 shows the learning rate for the lifespan predicate on DEAS03. Although the learning rate approaches a non-zero limit, the rule set does converge. The non-zero limit is the result of transitory changes in the rule set over time. In this case, these are due to uniquely named application files (usually for lock-files that include a random string in the name) that are being created at a relatively constant rate throughout the month. Because tree creation uses filename components [5], a unique filename component (e.g., the component *unique* in the filename *foo.unique.lock*) may get its own rule. Analysis indicates that these transitory rules have no impact on prediction accuracy, and can be lazily pruned from the tree based on simple LRU-style, frequency-of-use metrics.

5.3. Discussion

In general, we find that incremental learners perform better overall than the one-shot learners. Table 2 compares the one-shot and incremental learners over all of our traces and for each of the classes. In most cases the incremental learners performed slightly better, indicating that most of the activity seen on the first day of the training period was representative of the entire trace. There are, however, cases where the one-shot learner does poorly, resulting in lower average prediction accuracy. This behavior is particularly evident when comparing the accuracies of the write-only and read-only classes. In many cases, the incremental learner improved average accuracy by over 10%.

Furthermore, the rule sets for our incremental learners are relatively small. In this particular example, the month of DEAS03 activity resulted in about 1 million new files, for which we created 1000 new rules, giving us a 1000:1 *compression ratio*. Table 3 shows the compression ratios, which are the inverse of the training rate, over all traces and classes. These ratios show the total number of files in each trace to the total number of rules, at the end of the trace period. The size and lifespan predictions were made on newly created files and the write-only and read-only predictions on files that were written or read. The ratios are relative to the file counts shown in the table.

The relatively small set of rules means that a system administrator or a self-* system component can sort the rule set by frequency of use and obtain some insight into how the system is being used. For example, imagine a system that was being overwhelmed by application lock files. Trace-based diagnosis of such a problem would be tedious and incomplete, at best. Ideally, an administrator would like to set a rule in a self-* system to automatically discover the class of files exhibiting lock-file behavior. This class could then be optimized for. This example is actually a reality on the DEAS03 and EECS03 traces. The class is defined as any file that lives for less than a second and contains zero bytes of data. At create time, we can predict if a new file will exhibit these properties with 95% accuracy.

	DEAS03	EECS03	CAMPUS	LAB
created files	987K	575K	441K	116K
read/written	742K	711K	106K	140K
size=0	1134:1	725:1	4327:1	571:1
0<size≤16K	806:1	529:1	4012:1	282:1
lifespan≤1s	952:1	739:1	434:1	397:1
write-only	529:1	532:1	291:1	302:1
read-only	455:1	458:1	439:1	309:1

Table 3. File-to-rule compression ratios.

6. Modeling trade-offs

File classification for self-* systems has several trade-offs. For example, we can trade model efficiency for accuracy, and model accuracy for precision.

6.1. Trading efficiency for accuracy

File systems such as NTFS [22] allow applications to store attributes with a file, and research is being conducted to allow users and applications to tag files with additional attributes for improved search capabilities [9, 23]. Although these additional attributes may provide a wealth of information, building classification trees with too many attributes can result in over-fitting of the training data (i.e., discovering false associations) if there are too many attributes and too few observations. Furthermore, additional attributes can lead to more rules and smaller compression ratios. In our NFS traces, the attributes of a file are limited to the UNIX file attributes. However, even with such a small attribute space, we begin to see the effects of over-fitting.

In general, by withholding attributes from a model, the efficiency of model generation improves, thereby allowing for faster learning rates, but at the potential cost of accuracy. To test this, we create a modified version of ABLE that only uses filenames, and ignores the other attributes (GID, UID, permissions, creation time). The resulting tree, NABLE (name-based ABLE), is compared against ABLE for accuracy. ABLE does find false associations, albeit a small number (less than 2% difference in prediction accuracy), in many of the classes we are predicting. These false associations are eliminated with more training data. In this particular test, 10 extra days of training were required to completely erase the effects of over-fitting.

Over-fitting due to large attribute spaces is always a problem. Fortunately there are solutions [2, 11], most of which involve some form of attribute filtering as a pre-processing step to building a tree.

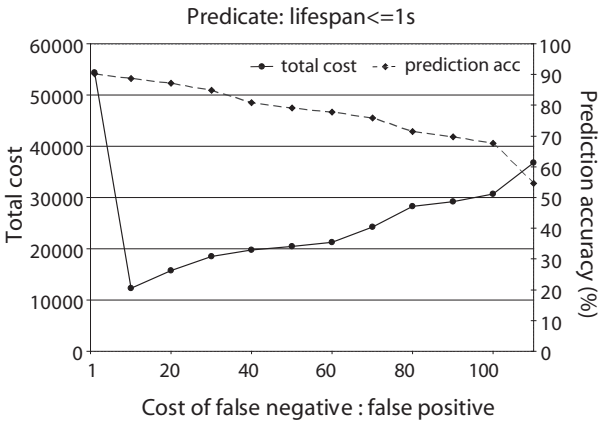


Figure 4. Total cost vs. learning bias.

6.2. Trading accuracy for precision

Predictive models can have false positives (incorrectly predicting that a new file belongs to a class) and false negatives. A more *accurate* model maximizes the number of correct predictions without considering the relative costs of these error types. A more *precise* model considers their costs so as to minimize an application cost function. However, introducing such a learning *bias* through cost functions reduces the model's overall accuracy. This is a common trade-off for classifiers, and decision trees can be made more or less precise as desired.

For example, consider building a learning model to predict the lifetime of a file. If a file will live for less than a second, we may choose to store it in NVRAM. Otherwise, the file is stored on disk through several I/Os. In this case, false negatives (a missed NVRAM opportunity) may be preferred, thus arguing for a more accurate model. On the other hand, when classifying files to be archived, we want to avoid false positives (prematurely sending a file to tape), thereby arguing for a model with greater precision.

By using cost functions, decision trees can be built from samples with different proportions of positive and negative examples. By weighting samples in this manner you penalize the tree for making one of the two types of errors (false positive or false negative) and create biased trees that try to avoid the particular error [28].

Figure 4 shows the total cost of mispredictions, relative to the cost of false negative for one day of the DEAS03 trace. The cost on the y-axis is in relative units, and the x-axis shows the relative cost of a false negative to a false positive. The optimal bias is the x-value that produces the lowest point in the graph. Figure 4 illustrates how a self-* system can adjust the cost and improve precision at the expense of overall accuracy. To be practical in a self-* system, system goals will need to be translated into appropriate cost

functions. In addition, techniques for estimating class probabilities can be used to allow an application to take more subtle actions, depending on the confidence of the prediction [17]. These are areas of future work.

7. Conclusion and Future Work

Truly self-* systems will automatically set policies, adapt to changes, and diagnose performance dips. This paper develops one technology that will help file and storage systems work towards these goals. ABLE uses decision trees to automatically learn how to classify files and predict the properties of new files, as they are created, based on the strong associations between a file's properties and its names and attributes. These trees provide prediction accuracies in the 90% range, across the four real NFS environments studied. They also efficiently capture the associations and adapt well over time via incremental learning.

In continuing work, we plan to integrate ABLE into a prototype self-* storage system being built at Carnegie Mellon University [7]. Model induction will happen regularly within each server in the system, and the models will be used to guide such policy decisions as encoding schemes (e.g., RAID5/erasure coding vs. mirroring), replication factors, disk layout, prefetching, and load balancing. Further, changes in the models may identify substantial changes in the set of applications being served, and can therefore be used to self-diagnose workload changes and automatically reconfigure a running system.

References

[1] A. Amer, D. Long, J.-F. Paris, and R. Burns. File access prediction with adjustable accuracy. International Performance Conference on Computers and Communication. IEEE, 2002.

[2] R. Caruana and D. Freitag. Greedy attribute selection. International Conference on Machine Learning, pages 28–36, 1994.

[3] T. Dietterich. Machine learning. *ACM Computing Surveys*, **28**(4):3–3.

[4] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of an email and research workload. Conference on File and Storage Technologies, pages 203–217. USENIX Association, 2003.

[5] D. Ellard, M. Mesnier, E. Thereska, G. R. Ganger, and M. Seltzer. *Attribute-based prediction of file properties*. TR 14-03. Harvard University, December 2003.

[6] G. R. Ganger. Generating Representative Synthetic Workloads: An Unsolved Problem. Proceedings of the Computer Management Group (CMG) Conference, pages 1263–1269, 1995.

[7] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. *Self-* Storage: brick-based storage with automated administration*. Technical Report CMU-CS-03-178. Carnegie Mellon University, August 2003.

[8] Gartner Group. Total Cost of Storage Ownership — A User-oriented Approach.

[9] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole Jr. Semantic file systems. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **25**(5):16–25, 13–16 October 1991.

[10] IBM. Autonomic Storage. http://www.almaden.com.

[11] G. H. John, R. Kohavi, and K. Pfleger. Irrelevant features and the subset selection problem. International Conference on Machine Learning, pages 121–129, 1994.

[12] A. J. Klosterman and G. Ganger. *Cuckoo: layered clustering for NFS*. Technical Report CMU–CS–02–183. Carnegie Mellon University, October 2002.

[13] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. Fourteenth international joint conference on artificial intelligence, pages 1137–1143, 1995.

[14] T. M. Kroeger and D. D. E. Long. The case for efficient file access pattern modeling. Hot Topics in Operating Systems, pages 14–19, 1999.

[15] E. Lamb. Hardware spending sputters. *Red Herring*, pages 32–33, June, 2001.

[16] T. M. Madhyastha and D. A. Reed. Input/output access pattern classification using hidden Markov models. Workshop on Input/Output in Parallel and Distributed Systems, pages 57–67. ACM Press, December 1997.

[17] D. D. Margineantu and T. G. Dietterich. Improved Class Probability Estimates from Decision Tree Models. Research note, Department of Computer Science, Oregon State Univ.

[18] J. T. McClave, F. H. Dietrich II, and T. Sincich. *Statistics*. Prentice Hall, 1997.

[19] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based Storage. *Communications Magazine*, **41**(8):84–90. IEEE, August 2003.

[20] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[21] J. Oly and D. A. Reed. Markov model prediction of I/O requests for scientific applications. ACM International Conference on Supercomputing. ACM, 2002.

[22] D. A. Solomon and M. E. Russinovich. *Inside Microsoft Windows 2000, Third Edition*. Microsoft Press, 2000.

[23] C. A. N. Soules and G. R. Ganger. Why can't I find my files? New methods for automating attribute assignment. Hot Topics in Operating Systems, pages 115–120. USENIX Association, 2003.

[24] C. Staelin and H. Garcia-Molina. Smart filesystems. Winter USENIX Technical Conference, pages 45–51, 21–25 January 1991.

[25] P. E. Utgoff, N. C. Berkman, and J. A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, **29**(1):5–44, 1997.

[26] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: a quantitative approach. First International Workshop on Peer-to-Peer Systems (IPTPS 2002), 2002.

[27] J. Wilkes. Data services - from data to containers.

[28] I. H. Witten and E. Frank. *Data mining: practical machine learning tools and techniques with Java implementations*. Morgan Kaufmann, 2000.

[29] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. Symposium on Operating Systems Design and Implementation, pages 243–258. USENIX Association, 2000.