Boosting as a Metaphor for Algorithm Design

Kevin Leyton-Brown, Eugene Nudelman,
Galen Andrew, Jim McFadden, and Yoav Shoham
{kevinlb;eugnud;galand;jmcf;shoham}@cs.stanford.edu

Stanford University, Stanford CA 94305

Abstract. Hard computational problems are often solvable by multiple algorithms that each perform well on different problem instances. We describe techniques for building an algorithm portfolio that can outperform its constituent algorithms, just as the aggregate classifiers learned by boosting outperform the classifiers of which they are composed. We also provide a method for generating test distributions to focus future algorithm design work on problems that are hard for an existing portfolio. We demonstrate the effectiveness of our techniques on the combinatorial auction winner determination problem, showing that our portfolio outperforms the state-of-the-art algorithm by a factor of three.[1]

1 Introduction

Although some algorithms are better than others on average, there is rarely a best algorithm for a given problem. Instead, it is often the case that different algorithms perform well on different problem instances. Not surprisingly, this phenomenon is most pronounced among algorithms for solving $\mathcal{NP}$-Hard problems, because runtimes for these algorithms are often highly variable from instance to instance. When algorithms exhibit high runtime variance, one is faced with the problem of deciding which algorithm to use; in 1976 Rice dubbed this the "algorithm selection problem" [13]. In the nearly three decades that have followed, the issue of algorithm selection has failed to receive widespread study, though of course some excellent work does exist. By far, the most common approach to algorithm selection has been to measure different algorithms' performance on a given problem distribution, and then to use only the algorithm having the lowest average runtime. This approach, to which we refer as "winner-take-all", has driven recent advances in algorithm design and refinement, but has resulted in the neglect of many algorithms that, while uncompetitive on average, offer excellent performance on particular problem instances. Our consideration of the algorithm selection literature, and our dissatisfaction with the winner-take-all approach, has led us to ask the following two questions. First, what general techniques can we use to perform per-instance (rather than per-distribution) algorithm selection? Second, once we have rejected the notion of winner-take-all algorithm evaluation, how ought novel algorithms to be evaluated? Taking the idea of boosting from machine learning as our guiding metaphor, we strive to answer both questions.

[1] This work has previously been published as a two-page extended abstract [9].

1.1 The Boosting Metaphor

Boosting is a machine learning paradigm due to Schapire [17] and widely studied since. Although this paper does not make use of any technical results from the boosting literature, it takes its inspiration from the boosting philosophy. Stated simply, boosting is based on two insights:

1. Poor classifiers can be combined to form an accurate ensemble when the classifiers' areas of effectiveness are sufficiently uncorrelated.
2. New classifiers should be trained on problems on which the current aggregate classifier performs poorly.

In this paper, we argue that algorithm design should be informed by two analogous ideas:

1. Algorithms with high average running times can be combined to form an algorithm portfolio with low average running time when the algorithms' easy inputs are sufficiently uncorrelated.
2. New algorithm design should focus on problems on which the current algorithm portfolio performs poorly.

Of course the analogy to boosting is imperfect; we discuss differences in section 5.

1.2 Case Study: Combinatorial Auctions (Weighted Set Packing)

To discuss the effectiveness of an algorithm design methodology, it is necessary to perform a case study. We chose to consider the combinatorial auction winner determination problem (WDP), and made use of runtime prediction techniques and runtime data from our previous work [10]. However, it must be emphasized that none of the techniques we propose here are particular to this problem domain. The full version of this paper will also consider other domains; in particular, we have had some positive initial results building portfolios for SAT.

Combinatorial auctions provide a general framework for allocation problems among self-interested agents by allowing bids for bundles of goods. WDP is a weighted set packing problem (SPP): the goal is to choose a non-conflicting subset of bids maximizing the seller's revenue. SPP is $\mathcal{NP}$-Complete, and also inapproximable within a constant factor (cf. [15]). Let n be the number of goods, and m be the number of bids. A bid is a pair $< S_i, p_i >$, where $S_i \subseteq \{1, \ldots, n\}$ is the set of goods requested by bid i, and p_i is that bid's price offer. WDP can be formulated as the following integer program:

$$\begin{aligned} \text{maximize:} \quad & \sum_{i=1}^{m} x_i p_i & \\ \text{subject to:} \quad & \sum_{i \mid g \in S_i} x_i \leq 1 & \forall g \\ & x_i \in \{0, 1\} & \forall i \end{aligned}$$

We consider three algorithms for solving WDP: ILOG's CPLEX package;GL (Gonen-Lehmann) [5], a simple branch-and-bound algorithm with CPLEX's LP solver as its heuristic; and CASS [2], a more complex branch-and-bound algorithm with a non-LP heuristic. Unfortunately, we were unable to get access to CABOB [16], another widely-cited WDP algorithm.

1.3 Overview

In the next three sections we give general methods for our boosting analogy in algorithm design. In section 2 we present a methodology for constructing algorithm portfolios and show some results from our case study. We go on in section 3 to offer practical extensions to our methodology, including techniques for avoiding the computation of costly features, trading off between accuracy on hard and easy instances, and building models when runtime data is capped at some maximum running time. In section 4 we consider the empirical evaluation of portfolios, and describe a method for using a learned model of runtime to generate a test distribution that will be hard for a portfolio. Similar techniques can be used to generate instances that score highly on a given "realism" metric. Finally, section 5 discusses our design choices and compares them to the choices made in related work.

2 Algorithm Portfolios

Our previous work [10] demonstrated that statistical regression can be used to learn surprisingly accurate algorithm-specific models of the empirical hardness of given distributions of problem instances. In short, the method proposed in that work is:

1. Use domain knowledge to select features of problem instances that might be indicative of runtime.
2. Generate a set of problem instances from the given distribution, and collect runtime data for the algorithm on each instance.
3. Use regression to learn a real-valued function of the features that predicts runtime.

Given this existing technique for predicting runtime, we now propose building portfolios of multiple algorithms as follows:

1. Train a model for each algorithm, as described above.
2. Given an instance:
   (a) Compute feature values
   (b) Predict each algorithm's running time using runtime models
   (c) Run the algorithm predicted to be fastest

This technique is powerful, but deceptively simple. For discussion and comparison with other approaches in the literature, please see section 5.1. As we will demonstrate in our case study, such portfolios can dramatically outperform the algorithms of which they are composed.

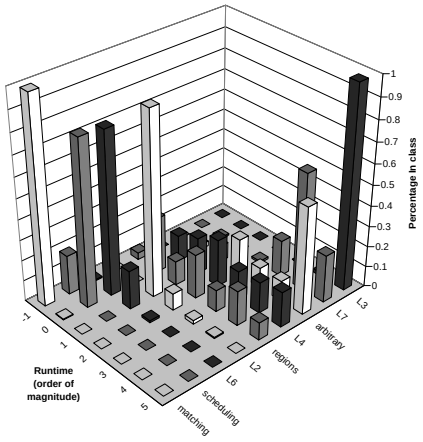

Fig. 1. Gross Hardness for CPLEX (from [10])

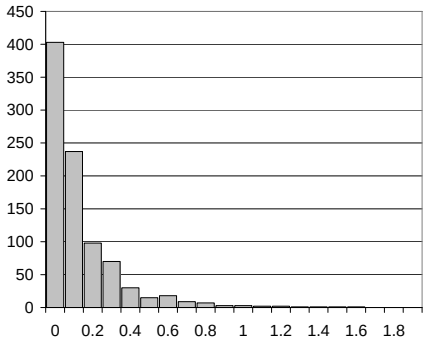

Fig. 2. Mean Absolute Error (from [10])

2.1 Case Study: Experimental Setup

We performed our case study using data collected in our past work [10], which we recap briefly in this section. All of our results focused on problems of a fixed size: numbers of goods and non-dominated bids were held constant to 256 and 1000 respectively.[2] Our instance distribution involved making a uniform choice between nine of the distributions from the Combinatorial Auction Test Suite (CATS) [11], and randomly choosing parameters for each instance. The complete dataset was composed of about 4500 instances. For each instance we collected runtime data for CPLEX 7.1, and computed 25 features that fall roughly into four categories:

1. Norms of the linear programming slack vector (integrality of the LP relaxation of the IP)
2. Deviations of prices
3. Node statistics of the Bid-Good bipartite graph
4. Various statistics of the Bid graph (effectively, the problem's constraint graph)

All data was collected on 550 MHz Pentium Xeon machines running Linux 2.2; over 3 years of CPU time was spent gathering this data. Fig. 1 shows a 3D histogram of the distribution of hard instances across our dataset. Observe that CPLEX's runtime varied by seven orders of magnitude even though the number of goods and bids was held constant. Also, there is considerable variation within most of the distributions.

Using quadratic regression, we were able to build very accurate models of the logarithm of runtime. Fig. 2 shows a histogram of the mean absolute error in predicting the log of CPLEX's runtime observed on test set instances. Since our methodology relies

[2] In a separate research effort, we are in the process of extending the work from [10] to models of variable problem size; when these models become available it will be possible to extend the techniques presented in this paper without any modification.

on machine learning, we split the data into training, validation, and test sets. We report our portfolio runtimes only on the test set that was never used to train or evaluate models. An error of 1 in predicting the log means that runtime was mispredicted by a factor of 10, or roughly that an instance was misclassified by one of the bins in Fig. 1; observe that nearly all prediction errors are less than 1.

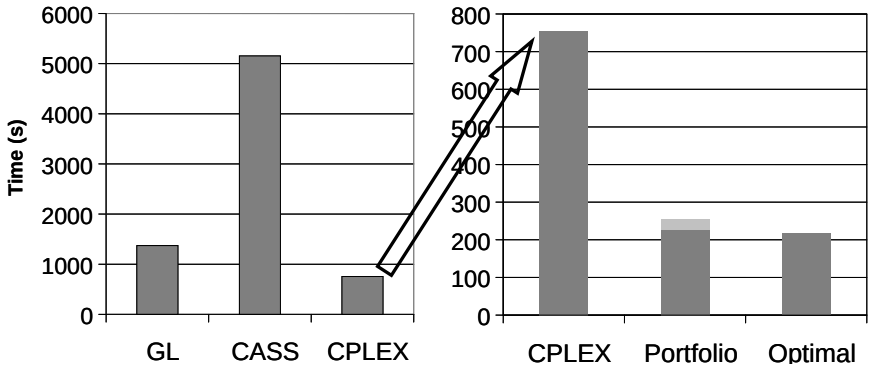

Fig. 3. Algorithm and Portfolio Runtimes

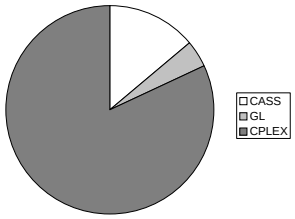

Fig. 4. Optimal

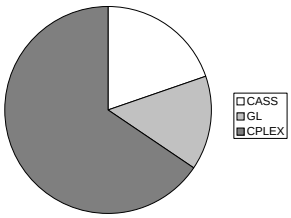

Fig. 5. Selected

2.2 Case Study Results

We now turn to new results. First, we used the methodolody described in section 2.1 to build regression models for two new algorithms (GL and CASS). Fig. 3 compares the average runtimes of our three algorithms (CPLEX, CASS, GL) to that of the portfolio[3]. Note that CPLEX would be chosen under winner-take-all algorithm selection. The "optimal" bar shows the performance of an ideal portfolio where algorithm selection is performed perfectly and with no overhead. The portfolio bar shows the time taken to compute features (light portion) and the time taken to run the selected algorithm (dark portion). Despite the fact that CASS and GL are much slower than CPLEX on average, the portfolio outperforms CPLEX by roughly a factor of 3. Moreover, neglecting the

[3] Note the change of scale on the graph, and the repeated CPLEX bar

cost of computing features, our portfolio's selections take only 5% longer to run than the optimal selections.

Figs. 4 and 5 show the frequency with which each algorithm is selected in the ideal portfolio and in our portfolio. They illustrate the quality of our algorithm selection and the relative value of the three algorithms. Observe that our portfolio does not always make the right choice (in particular, it selects GL much more often than it should). However, most of the mistakes made by our models occur when both algorithms have very similar running times; these mistakes are not very costly, explaining why our portfolio's choices have a running time so close to optimal.

These results show that our portfolio methodology can work very well even with a small number of algorithms, and when one algorithm's average performance is considerably better than the others'. We suspect that our techniques could work even better in other settings.

3 Extending our Portfolio Methodology

Once it has been demonstrated that algorithm portfolios can offer significant speedups over winner-take-all algorithm selection, it is worthwhile to consider modifications to the methodology that make it more useful in practice. Specifically, we describe methods for reducing the amount of time spent computing features, transforming the response variable, and capping runs of some or all algorithms.

3.1 Smart Feature Computation

Feature values must be computed before the portfolio can choose an algorithm to run. We expect that portfolios will be most useful when they combine several exponential-time algorithms having high runtime variance, and that fast polynomial-time features should be sufficient for most models. Nevertheless, on some instances the computation of individual features may take substantially longer than one or even all algorithms would take to run. In such cases it would be desirable to perform algorithm selection without spending as much time computing features, even at the expense of some accuracy in choosing the fastest algorithm. In order to achieve this, we partition the features into sets ordered by time complexity, $S_1, \ldots, S_l$, with $i > j$ implying that each feature in S_i takes significantly longer to compute than each feature in S_j.[4] The portfolio can start by computing the easiest features, and iteratively compute the next set only if the expected benefit to selection exceeds the cost of computation. More precisely:

1. For each set S_j learn or provide a model $c(S_j)$ that estimates time required to compute it. Often, this could be a simple average time scaled by input size.
2. Divide the training examples into two sets. Using the first set, train models $M_1^i \ldots M_l^i$, with M_k^i predicting algorithm i's runtime using features in $\bigcup_{j=1}^{k} S_j$. Note that M_l^i is the same as the model for algorithm i in our basic portfolio methodology. Let M_k be a portfolio which selects $\text{argmin}_i\, M_k^i$.

[4] We assume here that features will have low runtime variance. We have found this assumption to hold in our case study. If feature runtime variance makes it difficult to partition the features into time complexity sets, smart feature computation is more difficult.

3. Using the second training set, learn models $D_1 \ldots D_{l-1}$, with D_k predicting the difference in runtime between the algorithms selected by M_k and M_{k+1} based on S_k. The second set must be used to avoid training the difference models on data to which the runtime models were fit.

Given an instance x, the portfolio now works as follows:

4. For $j = 1$ to l
   (a) Compute features in S_j
   (b) If $D_j[x] > c(S_{j+1})[x]$, continue.
   (c) Otherwise, return with the algorithm predicted to be fastest according to M_j.

3.2 Transforming the Response Variable

Average runtime is an obvious measure of portfolio performance if one's goal is to minimize computation time over a large number of instances. Since our models minimize root mean squared error, they appropriately penalize 20 seconds of error equally on instances that take 1 second or 10 hours to run. However, another reasonable goal may be to perform well on every instance regardless of its hardness; in this case, relative error is more appropriate. Let r_i^p and r_i^* be the portfolio's runtime and the optimal runtime respectively on instance i, and n be the number of instances. One measure that gives an insight into the portfolio's relative error is *percent optimal*:

$$\sum_{i|r_i^p=r_i^*} \frac{1}{n}.$$

Another measure of relative error is *average percent suboptimal*:

$$\frac{1}{n} \sum_i \frac{r_i^p - r_i^*}{r_i^*}.$$

Taking a logarithm of runtime is a simple way to equalize the importance of relative error on easy and hard instances. Thus, models that predict a log of runtime help to improve the average percent suboptimal, albeit at some expense in terms of the portfolio's average runtime. In Figure 6 (overleaf) we show three different functions; linear (identity) and log are the extreme values; clearly, many functions can fall in between. The functions are normalized by their maximum value, since this does not affect regression, but allows us to better visualize their effect. In our case study (section 3.4) we found that the cube root function was particularly effective.

3.3 Capping Runs

The methodology of section 2 requires gathering runtime data for every algorithm on every problem instance in the training set. While the time cost of this step is fundamentally unavoidable for our approach, gathering perfect data for every instance can take an unreasonably long time. For example, if algorithm a_1 is usually much slower than a_2 but in some cases dramatically outperforms a_2, a perfect model of a_1's runtime on hard

instances may not be needed to discriminate between the two algorithms. The process of gathering data can be made much easier by capping the runtime of each algorithm at some maximum and recording these runs as having terminated at the captime. This approach is safe if the captime is chosen so that it is (almost) always significantly greater than the minimum of the algorithms' runtimes; if not, it might still be preferable to sacrifice some predictive accuracy for dramatically reduced model-building time. Note that if any algorithm is capped, it can be dangerous (particularly without a log transformation) to gather data for any other algorithm without capping at the same time, because the portfolio could inappropriately select the algorithm with the smaller captime.

3.4 Case Study Results

Fig. 7 shows the performance of the smart feature computation discussed in section 3.1, with the upper part of the bar indicating the time spent computing features. Compared to computing all features, we reduce overhead by almost half with nearly no cost in running time.

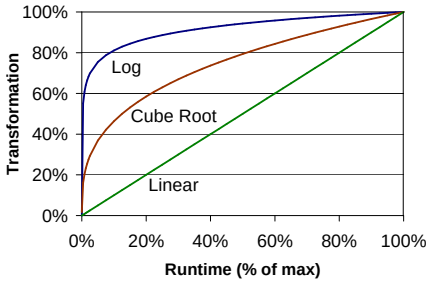

Fig. 6. Transformation F'ns (Normalized)

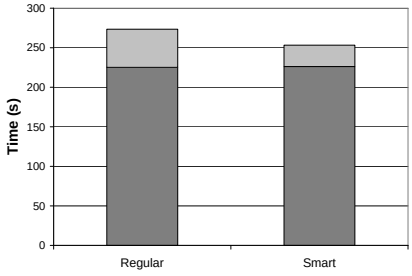

Fig. 7. Smart Features

	Average Runtime	% Optimal	Average % Suboptimal
(Optimal)	216.4 s	100	0
Log	236.5 s	97	9
Cuberoot	225.6 s	89	17
Linear	225.1 s	81	1284

Table 1. Portfolio Results

Table 1 shows the effect of our response variable transformations on average runtime, percent optimal and average percent suboptimal. The first row has results that would be obtained by a perfect portfolio. As discussed in section 3.2, the linear (identity) transformation yields the best average runtime, while the log function leads to better algorithm selection. We tried several transformation functions between linear and

log. Here we only show the best, cube root: it has nearly the same average runtime performance as linear, but also made choices nearly as accurately as log.

4 Focused Algorithm Design

Once we have decided to select among existing algorithms using a portfolio approach, it is necessary to reexamine the way we design and evaluate algorithms. Since the purpose of designing new algorithms is to reduce the time that it will take to solve problems, designers should aim to produce new algorithms that complement an existing portfolio. First, it is essential to choose a distribution D that reflects the problems that will be encountered in practice. Given a portfolio, the greatest opportunity for improvement is on instances that are hard for that portfolio, common in D, or both. More precisely, the importance of a region of problem space is proportional to the amount of time the current portfolio spends working on instances in that region. This is analogous to the principle from boosting that new classifiers should be trained on instances that are hard for the existing ensemble, in the proportion that they occur in the original training set.

4.1 Inducing Hard Distributions

Let H_f be a model of portfolio runtime based on instance features, constructed as the minimum of the models that constitute the portfolio. By normalizing, we can reinterpret this model as a density function h_f. By the argument above, we should generate instances from the product of this distribution and our original distribution, D. However, it is problematic to sample from $D \cdot h_f$: D may be non-analytic (an instance generator), while h_f depends on features and so can only be evaluated after an instance has been created.

One way to sample from $D \cdot h_f$ is rejection sampling [1]: generate problems from D and keep them with probability proportional to h_f. This method works best when another distribution is available to guide the sampling process toward hard instances. Test distributions usually have some tunable parameters $\overrightarrow{p}$, and although the hardness of instances generated with the same parameter values can vary widely, $\overrightarrow{p}$ will often be somewhat predictive of hardness. We can generate instances from $D \cdot h_f$ in the following way:[5]

1. Create a hardness model H_p with features $\overrightarrow{p}$, and normalize it to create a pdf, h_p.
2. Generate a large number of instances from $D \cdot h_p$.
3. Construct a distribution over instances by assigning each instance s probability proportional to $\frac{H_f(s)}{h_p(s)}$, and select an instance by sampling from this distribution.

Observe that if h_p turns out to be helpful, hard instances from $D \cdot h_f$ will be encountered quickly. Even in the worst case where h_p directs the search away from hard

[5] In true rejection sampling step 2 would generate a single instance that would be then accepted or rejected in step 3. Our technique approximates this process, but doesn't require us to normalize H_f and allows us to output an instance after generating a constant number of samples.

instances, observe that we still sample from the correct distribution because the weights are divided by $h_p(s)$.

In practice, D may be factored as $D_g \cdot D_{p_i}$, where D_g is a distribution over otherwise unrelated instance generators with different parameter spaces, and D_{p_i} is a distribution over the parameters of the chosen instance generator i. In this case it is difficult to learn a single H_p. A good solution is to factor h_p as $h_g \cdot h_{p_i}$, where h_g is a hardness model using only the choice of instance generator as a feature, and h_{p_i} is a hardness model in instance generator i's parameter space. Likewise, instead of using a single feature-space hardness model H_f, we can train a separate model for each generator $H_{f,i}$ and normalize each to a pdf $h_{f,i}$.[6] The goal is now to generate instances from the distribution $D_g \cdot D_{p_i} \cdot h_{f,i}$, which can be done as follows:

1. For every instance generator i, create a hardness model H_{p_i} with features $\overrightarrow{p_i}$, and normalize it to create a pdf, h_{p_i}.
2. Construct a distribution over instance generators h_g, where the probability of each generator i is proportional to the average hardness of instances generated by i.
3. Generate a large number of instances from $(D_g \cdot h_g) \cdot (D_{p_i} \cdot h_{p_i})$
   (a) select a generator i by sampling from $D_g \cdot h_g$
   (b) select parameters for the generator by sampling from $D_{p_i} \cdot h_{p_i}$
   (c) run generator i with the chosen parameters to generate an instance.
4. Construct a distribution over instances by assigning each instance s from generator i probability proportional to $\frac{H_{f,i}(s)}{h_g(s) \cdot h_{p_i}(s)}$, and select an instance by sampling from this distribution.

4.2 Inducing Realistic Distributions

It is important for our portfolio methodology that we begin with a "realistic" D: that is, a distribution accurately reflecting the sorts of problems expected to occur in practice. Care must always be taken to construct a generator or set of generators producing instances that are representative of problems from the target domain. Sometimes, it is possible to construct a function R_f that scores the realism of a generated instance based on features of that instance; such a function can encode additional information about the nature of realistic data that cannot easily be expressed in a generator. If a function R_f is provided, we can construct D from a parameterized set of instance generators by using R_f in place of H_f above and learning r_p in the same way we learned h_p. This can allow us to make informed choices when setting the parameters of instance generators, and also to discard less realistic data after it has been generated. Note that when inducing hard distributions a hardness model had to be used because it was infeasible to score each sample by actual portfolio runtime. In the case of inducing realistic distributions this is no longer a problem, because the realism function *can* be evaluated on each sample. Therefore, our rejection sampling technique is guaranteed to generate instances with increased average realism scores. The use of parameter-space models r_p can still improve performance by reducing the number of samples needed for obtaining good results.

[6] However, the case study results presented in figs. 8–10 use hardness models H_f trained on the whole dataset rather than using models trained on individual distributions. Learning new models would probably yield even better results.

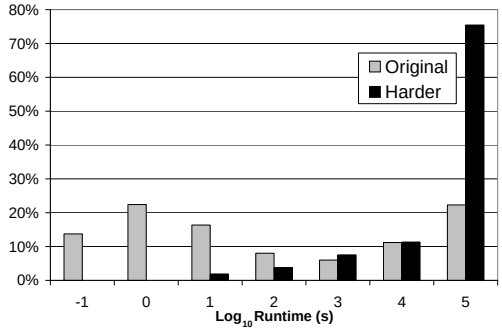

Fig. 8. Inducing Harder Distributions

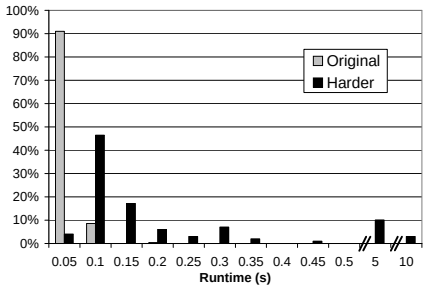

Fig. 9. Matching

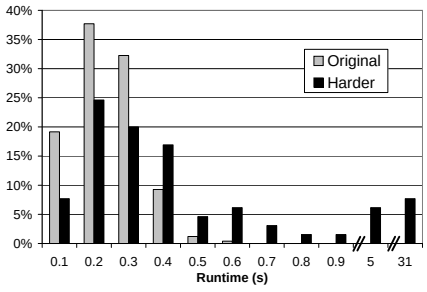

Fig. 10. Scheduling

4.3 Case Study Results

Due to the wide spread of runtimes in our composite distribution D (7 orders of magnitude) and the high accuracy of our model h_f [10], it is quite easy for our technique to generate harder instances. These results are presented in fig. 8. Because our runtime data was capped, there is no way to know if the hardest instances in the new distribution are harder than the hardest instances in the original distribution; note, however, that very few easy instances are generated. Instances in the induced distribution came predominantly from the CATS "arbitrary" distribution, with most of the rest from "L3".

To demonstrate that our technique also works in more challenging settings, we sought a different distribution with small runtime variance. As it happens, there has been ongoing discussion in the WDP literature about whether those CATS distributions [11] that are relatively easy could be configured to be harder (see e.g., [4, 16]). We consider two easy distributions with low variance from CATS, *matching* and *scheduling*, and show that they indeed can be made harder than originally proposed. Figures 9 and 10 show the histograms of the runtimes of the ideal portfolio before and after our technique was applied. In fact, for these two distributions we generated instances that were (respectively) 100 and 50 times harder than anything we had previously seen! Moreover, the *average* runtime for the new distributions was greater than the observed *maximum* running time on the original distribution.

5 Discussion and Related Work

Although it is helpful, our analogy to boosting is clearly not perfect. One key difference lies in the way components are aggregated: classifiers can be combined through majority voting, whereas the whole point of algorithm selection is to run only a single algorithm. We instead advocate the use of learned models of runtime as the basis for algorithm selection, which leads to another important difference. It is not enough for the easy problems of multiple algorithms to be uncorrelated; the models must also be accurate enough to reliably recommend against the slower algorithms on these uncorrelated instances. Finally, while it is impossible to improve on correctly classifying an instance, it is almost always possible to solve a problem instance more quickly. Thus improvement is possible on easy instances as well as on hard instances; the analogy to boosting holds in the sense that focusing on hard regions of the problem space increases the potential gain in terms of reduced average portfolio runtimes.

5.1 Algorithm Selection

It has long been understood that algorithm performance can vary substantially across different classes of problems. Rice [13] was the first to formalize algorithm selection as a computational problem, framing it in terms of function approximation. Broadly, he identified the goal of selecting a mapping $S(x)$ from the space of instances to the space of algorithms, to maximize some performance measure perf$(S(x), x)$. Rice offered few concrete techniques, but all subsequent work on algorithm selection can be seen as falling into his framework. We explain our choice of methodology by relating it to other approaches for algorithm selection that have been proposed in the literature.

Parallel Execution One tempting alternative to portfolios that select a single algorithm is the parallel execution of all available algorithms. While it is often true that additional processors are readily available, it is also often the case that these processors can be put to uses besides running different algorithms in parallel, such as parallelizing a single search algorithm or solving multiple problem instances at the same time. Meaningful comparisons of running time between parallel and non-parallel portfolios require that computational resources be fixed, with parallel execution modelled as ideal (no-overhead) task swapping on a single processor. Let $t^*(x)$ be the time it takes to run the algorithm that is fastest on instance x, and let n be the number of algorithms. A portfolio that executes all algorithms in parallel on instance x will always take time $nt^*(x)$. On the data from our case study such parallel execution has roughly the same average runtime as winner-take-all algorithm selection (we have three algorithms and CPLEX is three times slower than the optimal portfolio), while our techniques do much better, achieving running times of roughly $1.05t^*(x)$.

In some domains, parallel execution *can* be a very effective technique. Gomes and Selman [3] proposed such an approach for incomplete SAT algorithms, using the term *portfolio* to describe a set of algorithms run in parallel. In this domain runtime depends heavily on variables such as random seed, making runtime difficult to predict; thus parallel execution is likely to outperform a portfolio that chooses a single algorithm.

In such cases it is possible to extend our methodology to allow for parallel execution. We can add one or more new algorithms to our portfolio, with algorithm i standing as a placeholder for the parallel execution of k_i of the original algorithms; in the training data i would be given a running time of k_i times the minimum of its constituents. This approach would allow portfolios to choose to task-swap sets of algorithms in parts of the feature space where the minimums of individual algorithms' runtimes are much smaller than their means, but to choose single algorithms in other parts of the feature space. Our use of the term "portfolio" may thus be seen as an extension of the term coined by Gomes and Selman, referring to a set of algorithms and a strategy for selecting a subset (perhaps one) for parallel execution.

Classification Since algorithm selection is fundamentally discriminative—it entails choosing the algorithm that will exhibit minimal runtime—classification is an obvious approach to consider. Any standard classification algorithm (e.g., a decision tree) could be used to learn which algorithm to choose given features of the instance and labelled training examples. The problem is that such classification algorithms use the wrong error metric: they penalize misclassifications equally regardless of their cost. We want to minimize a portfolio's average runtime, not its accuracy in choosing the optimal algorithm. Thus we should penalize misclassifications more when the difference between the runtimes of the chosen and fastest algorithms is large than when it is small. This is just what happens when our decision criterion is to select the smallest prediction among a set of regression models that were fit to minimize root mean squared error.

A second classification approach entails dividing running times into two or more bins, predicting the bin that contains the algorithm's runtime, and then choosing the best algorithm. For example, Horvitz et. al. [6, 14] used classification to predict runtime of CSP and SAT solvers with inherently high runtime variance (heavy tails). Despite its similarity to our portfolio methodology, this approach suffers from the use of a classification algorithm to predict runtime. First, the learning algorithm does not use an error function that penalizes large misclassifications (off by more than one bin) more heavily than small misclassifications (off by one bin). Second, this approach is unable to discriminate between algorithms when multiple predictions fall into the same bin. Finally, since runtime is a continuous variable, class boundaries are artificial. Instances with runtimes lying very close to a boundary are likely to be misclassified even by a very accurate model, making accurate models harder to learn.

Markov Decision Processes Perhaps most related to our paper is work by Lagoudakis and Littman ([7, 8]). They worked within the MDP framework, and concentrated on recursive algorithms (e.g. sorting, SAT), sequentially solving the algorithm selection problem on each subproblem. This work demonstrates encouraging results; however, its generality is limited by several factors. First, the use of algorithm selection at each stage of a recursive algorithm can require extensive recoding, and may simply be impossible with 'black-box' commercial or proprietary algorithms, which are often among the most competitive. Second, solving the algorithm selection problem recursively requires that the value functions be very inexpensive to compute; in our case study we found that more computationally expensive features were required for accurate predictions of run-

time. Finally, these techniques can be undermined by non-Markovian algorithms, such as those using clause learning, taboo lists or other forms of dynamic programming.

Of course, our approach can also be described in an MDP framework, with each action (choice of algorithm) leading to a terminal state, and reward equal to the negative of runtime. Optimal policy selection is trivial given a good value function; thus the key to success is good value estimation. Our approach emphasizes making the value functions—that is, models of runtime—explicit, since this provides the best defense against good but fragile policies. We do not describe our models as MDPs because the framework is redundant in the absence of sequential decision-making.

Different Regression Approaches Lobjois and Lemaître [12] select among several simple branch-and-bound algorithms based on a prediction of running time. This work is similar in spirit to our own; however, their prediction is based on a single feature and works only on a particular class of branch-and-bound algorithms.

Since our goal is to discriminate among algorithms, it might seem more appropriate to learn models of pairwise differences between algorithm runtimes, rather than models of absolute runtimes. For linear regression (and the forms of nonlinear regression used in our work) it is easy to show that the two approaches are mathematically equivalent.

5.2 Inducing Hard Distributions

It is widely recognized that the choice of test distribution is important for algorithm development. In the absence of general techniques for generating instances that are both realistic and hard, the development of new distributions has usually been performed manually. An excellent example of such work is Selman et. al. ([18]), which describes a method of generating SAT instances near the phase transition threshold, which are known to be hard for most SAT solvers.

6 Conclusions

Just as boosting allows weak classifiers to work together effectively, algorithms can be combined into portfolios to build a whole greater than the sum of its parts. First, we have described how to build such portfolios. Our techniques can be elaborated to reduce the cost of computing features, to reduce the time spent gathering training data by capping runs, and to strike the right balance between the penalties for mispredicting easy and hard instances. Second, we argued that algorithm design should focus on problem instances upon which a portfolio of existing algorithms spends most of its time. We have provided techniques for inducing such distributions, and also for refining distributions to emphasize instances that have high scores on a given 'realism' function. We performed a case study on combinatorial auctions, and showed that a portfolio composed of CPLEX and two older—and generally *much* slower—algorithms outperformed CPLEX alone by about a factor of 3. In future work, we aim to perform case studies of our methodology on other hard problems; our first effort in this direction is a portfolio of 10 algorithms which we have entered in the 2003 SAT competition.

Acknowledgments

Thanks to Ryan Porter, Carla Gomes and Bart Selman for helpful discussions. This work was supported by DARPA grant F30602-00-2-0598, the Intelligent Information Systems Institute at Cornell, and a Stanford Graduate Fellowship.

References

1. A. Doucet, N. de Freitas, and N. Gordon(ed.). *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, 2001.
2. Y. Fujishima, K. Leyton-Brown, and Y. Shoham. Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In *IJCAI*, 1999.
3. C. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.
4. R. Gonen and D. Lehmann. Optimal solutions for multi-unit combinatorial auctions: Branch and bound heuristics. In *ACM Conference on Electronic Commerce*, 2000.
5. R. Gonen and D. Lehmann. Linear programming helps solving large multi-unit combinatorial auctions. Technical Report TR-2001-8, Leibniz Center for Research in Computer Science, April 2001.
6. E. Horvitz, Y. Ruan, C. Gomes, H. Kautz, B. Selman, and M. Chickering. A Bayesian approach to tackling hard computational problems. In *UAI*, 2001.
7. M. Lagoudakis and M. Littman. Algorithm selection using reinforcement learning. In *ICML*, 2000.
8. M. Lagoudakis and M. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. In *LICS/SAT*, 2001.
9. K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. A portfolio approach to algorithm selection. In *IJCAI*, 2003.
10. K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *CP*, 2002.
11. K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *ACM EC*, 2000.
12. L. Lobjois and M. Lemaître. Branch and bound algorithm selection by performance prediction. In *AAAI*, 1998.
13. J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.
14. Y. Ruan, E. Horvitz, and H. Kautz. Restart policies with dependence among runs: A dynamic programming approach. In *CP*, 2002.
15. T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *IJCAI*, 1999.
16. T. Sandholm, S. Suri, A. Gilpin, and D. Levine. CABOB: A fast optimal algorithm for combinatorial auctions. In *IJCAI*, 2001.
17. R. Schapire. The strength of weak learnability. *Machine Learning*, 5:197–227, 1990.
18. B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.