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 [⋆⋆]

1 Introduction

Although some algorithms are better than others on average, there is rarely a best algorithm for a given problem. Instead, different algorithms often perform well on different problem instances. Not surprisingly, this phenomenon is most pronounced among algorithms for solving $\mathcal{NP}$-hard problems, when runtimes are 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" [8]. More recent work on this problem includes [5, 4].

Our previous work [7] demonstrates that statistical regression can be used to learn surprisingly accurate models of an algorithm's runtime. In a recent extended abstract [6] we discussed the use of these runtime models for algorithm selection, and also for the automated tuning of instance generators. This companion paper extends these ideas, describing new techniques for making algorithm portfolios more practical and for making benchmarks harder. As in [7, 6] we evaluate our techniques in a case study on the combinatorial auction winner determination problem (WDP)—an $\mathcal{NP}$-hard combinatorial optimization problem formally equivalent to weighted set packing. We consider three algorithms for solving WDP: ILOG's CPLEX package; GL (Gonen-Lehmann) [3], 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.

What does it mean to see boosting as a metaphor for algorithm design? Boosting is a machine learning paradigm [9] 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. 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* spends most of its time.

[⋆⋆] 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.

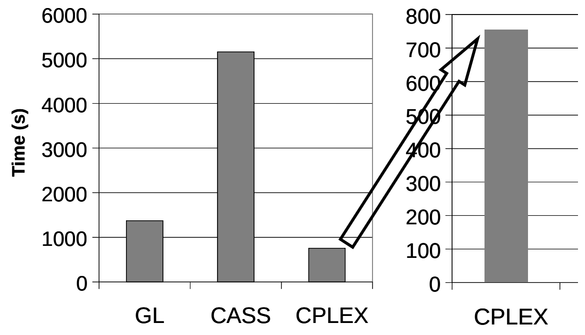

Fig. 1. Algorithm and Portfolio Runtimes

2 Making Algorithm Portfolios Practical

We have demonstrated that algorithm portfolios can offer significant speedups over winner-take-all algorithm selection (see Fig. 1, reproduced from [6]). It is thus worthwhile to consider modifications to the methodology that make it more useful in practice.

2.1 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*: $\frac{1}{n}\#\{i|r_i^p = r_i^*\}$. 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. Other transformations achieve different tradeoffs. In Figure 2 (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 2.4) we found that the cube root function was particularly effective.

2.2 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—if an instance is easy for all algorithms, we can tolerate a much greater prediction error. 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. 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 $\text{argmin}_i\, M_k^i$.
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.

2.3 Capping Runs

The methodology of [7] 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. When 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 for discrimination. The process of gathering data can be made much easier by capping the runtime of each algorithm and recording these runs as having terminated at the captime. This 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 one algorithm is capped, it can be dangerous (particularly without a log transformation) to gather data for another algorithm without capping at the same time, because the portfolio could inappropriately select the algorithm with the smaller captime.

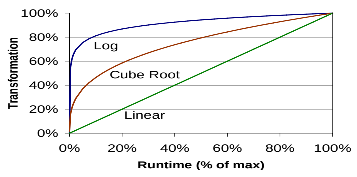

Fig. 2. Transformation

	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

Fig. 3. Portfolio Results

2.4 Case Study Results

Table 3 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 2.1, 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. Notice that the three models shown here are not equally accurate on our dataset (they are non-linear transformations of each other). The effect of the transformations is to shift model accuracy to achieve different tradeoffs. That fact that all of these models achieve good portfolio performance illustrates the robustness of our portfolio results with respect to model accuracy.

When using smart feature computation described in section 2.2, the average time spent computing features is cut almost in half, without any significant effect on the actual algorithms' running time (the graph is omitted due to the lack of space). This result becomes quite significant for easy instances.

3 Inducing Hard Distributions

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 of new algorithms should aim to complement an existing portfolio. First, it is essential to choose a distribution D that reflects the problems that will be encountered in practice. 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. 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 (formally, importance is measured by $D \cdot h_f$). 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.

Sampling from $D \cdot h_f$ is problematic, since 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 handle this is rejection sampling [1]: generate problems from D and keep them with probability proportional to h_f. (In fact, the technique described

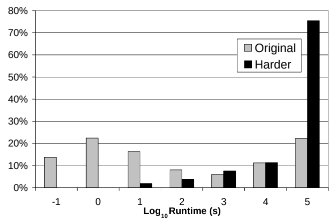

Fig. 4. Inducing Harder Distributions

below is approximate rejection sampling, which saves us from having to normalize H_f and always outputs an instance after a constant number of samples.) 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:

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.

Note, that if h_p is 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 instances, we'll still sample from the correct distribution, since the weights are divided by $h_p(s)$.

Figure 4 shows the results of applying this procedure to our dataset. Since our runtimes were capped, the induced distribution doesn't generate any instances that are orders of magnitude harder than previous instances. In [6] we showed that this can also be achieved, making extremely easy distributions between 50 and 100 times harder.

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. R. Gonen and D. Lehmann. Linear programming helps solving large multi-unit combinatorial auctions, April 2001. TR-2001-8, Leibniz Center for Research in Computer Science.
4. E. Horvitz, Y. Ruan, C. Gomes, H. Kautz, B. Selman, and M. Chickering. A Bayesian approach to tackling hard computational problems. In *UAI*, 2001.
5. M. Lagoudakis and M. Littman. Algorithm selection using reinforcement learning. In *ICML*, 2000.
6. K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. A portfolio approach to algorithm selection. In *IJCAI*, 2003.
7. K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *CP*, 2002.
8. J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.
9. R. Schapire. The strength of weak learnability. *Machine Learning*, 5:197–227, 1990.