ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Volume 1, Issue 8, October 2013

Boosting As a Metaphor for Algorithm Design

Kannan Subramanian

Department of MCA, Bharath Institute of Science and Technology, Chennai, TamilNadu, India

ABSTRACT: 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 algorithms for solving NP-hard problems, when run times are highly variable from instance to instance. When algorithms exhibit high run time variance, one is faced with the problem of deciding which algorithm is to be used.

MAKING ALGORITHM PORTFOLIOS PRACTICAL

We have demonstrated that algorithm portfolios can offer significant speedups over winner-take-all algorithm selection. It is thus worth while to consider modifications to the methodology that make it more useful in practice.

I. TRANSFORMING THE RESPONSE VARIABLE

Average run time 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 s most appropriate. Let rpi 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. Another measure of relative error is average percent suboptimal.

Taking a logarithm of runtime is a simple way to equalize the importance of relative error in easy and hard instances. Thus, models that predict a log of runtime helps to improve the average percent suboptimal, albeit at some expense in terms of the portfolio's average runtime. Other transformations achieve different tradeoffs. The functions are normalized by their maximum value, since this does not affect regression, but allows us to better visualize their effect.

II. SMART FEATURE COMPUTATION

Feature value must be computed before the portfolio can choose an algorithm to run. We expect that portfolio's will be most useful when they combine several exponential time algorithms having high run time 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 of time in 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. 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:

- For each set Sj learn or provide a model c(Sj) that estimates time required to compute it. Often, this could be a average time scaled by input size.
- Divide the training examples into two sets. Using the first set, train models M1:::Ml, with Mi k predicting algorithm I's runtime using features in Sk j=1 Sj. Note that Mi l is the same as the model for algorithm I in our basic portfolio methodology. Let Mk be a portfolio which selects argminiMi k.
- Using the second training set learn models D1:::Dl i1, with Dk predicting the difference in runtime between the algorithms selected by Mk and Mk+1 based on data to which the runtime models were fit.
- For j=1 to l.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Volume 1, Issue 8, October 2013

1. Compute features in Sj.
2. If Dj[x]>c(Sj+1)[x], continue.
3. Otherwise, return with an algorithm predicted to be fastest according to Mj.

III. CAPPING RUNS

The methodology 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 a1 is usually much slower than a2 . But in some cases dramatically outperforms a2, a perfect model of a1's runtime on hard instances may not be needed for discrimination. The process of gathering data can be made much easier by the capping the run time of each algorithm and recording these runs as having terminated at the captime. This is safe if the captime is chosen that it is always significantly greater than the minimum of the algorithm's runtime; 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 to gather data for another algorithm without capping at the same time, because the portfolio could inappropriately select the algorithm with the smaller captime.

IV. CASE STUDY RESULTS

The first row has results that would be a perfect portfolio. We tried several transformation functions between linear and log. Here we only show the best, cube root: it has nearly the average runtime performance as linear, but also made choices nearly as accurate as log. Notice that the three models sown here are not equally accurate on our dataset. The effect on the transformations is to shift model accuracy to achieve different tradeoff's. That fact that all of these models achieve good portfolio results with respect to model accuracy. When using smart feature computation the average time spent on computing features s almost in half without any significant effect on the actual algorithm's running time. This result becomes quite significant for easy instances.

V. 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 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 Hf 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 Hf. 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 to that region. This is analogous to the principal 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 Dhf is problematic, since D may not be non-analytic while hf depends on features and so only be evaluated after an instance can be created. One way to handle this is rejection sampling : 1.generate problems from D and keep them with probability proportional to hf. This method works best when another distribution is available to guide the sampling process towards hard instances. Test distributions usually have some tunable parameters and although the hardness of instances generated with the same parameter values can vary widely. We can generate instances from D hf in the following way:

1. Create a hardness model Hp with features and normalize it to create a pdf, hp.
2. Generate a large number of instances from D hp.
3. Construct a distribution over instances by assigning each instance s probability proportional to Hf(s).

Note, that if Hp s helpful, hard instances from Dhf will be encountered quickly. Even in the first case where hp directs the search away from hard instances, we'll sample from the correct distribution, since the weights are divided by hp(s).

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Volume 1, Issue 8, October 2013

Since our runtimes are capped, the induced distribution doesn't generate any instances that are orders of magnitude harder than previous instances.

REFERENCES

1. A.Doucet, N. de Freitas and N. Gordon, "Sequential Monte Caarlo Methods in Practice", Springer Verlag, 2001.
2. J.R Rice, "The algorithm selection problem. Advances in Computers", 15:65-118.1976.
3. E. Horvitz Y. Ruan, C.Gomes, "A Bayesian approach to tackling hard computational problems", UAI 2001.