RESEARCH HIGHLIGHTS

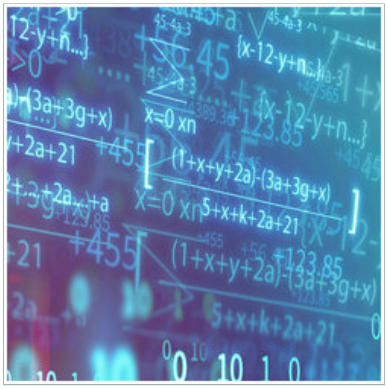
Data-Driven Algorithm Design

By Rishi Gupta, Tim Roughgarden

Communications of the ACM, June 2020, Vol. 63 No. 6, Pages 87-94
10.1145/3394625

Comments

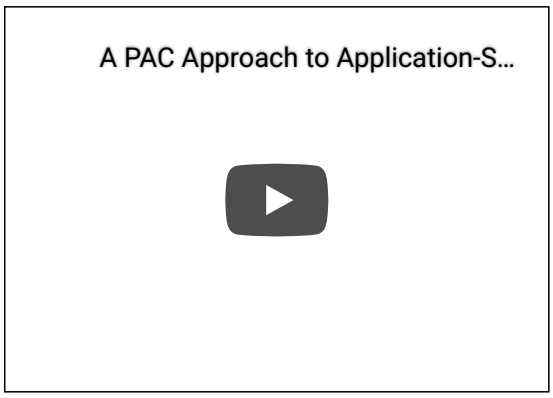
VIEW AS: SHARE:



Credit: Getty Images

The best algorithm for a computational problem generally depends on the "relevant inputs," a concept that depends on the application domain and often defies formal articulation. Although there is a large literature on empirical approaches to selecting the best algorithm for a given application domain, there has been surprisingly little theoretical analysis of the problem.

We model the problem of identifying a good algorithm from data as a statistical learning problem. Our framework captures several state-of-the-art empirical and theoretical approaches to the problem, and our results identify conditions under which these approaches are guaranteed to perform well. We interpret our results in the contexts of learning greedy heuristics, instance feature-based algorithm selection, and parameter tuning in machine learning.



Back to Top

1. Introduction

Rigorously comparing algorithms is hard. Two different algorithms for a computational problem generally have incomparable performance: one algorithm is better on some inputs but worse on the others. How can a theory advocate one of the algorithms over the other? The simplest and most common solution in the theoretical analysis of algorithms is to summarize the performance of an algorithm using a single number, such as its worst-case performance or its average-case performance with respect to an input distribution. This approach effectively advocates using the algorithm with the best summarizing value (e.g., the smallest worst-case running time).

Solving a problem "in practice" generally means identifying an algorithm that works well for most or all instances of interest. When the "instances of interest" are easy to specify formally in advance—say, planar graphs, the traditional analysis approaches often give accurate performance predictions and identify useful algorithms. However, the instances of interest commonly possess domain-specific features that defy formal articulation. Solving a problem in practice can require designing an algorithm that is optimized for the specific application domain, even though the special structure of its instances is not well understood. Although there is a large literature, spanning numerous communities, on empirical approaches to data-driven algorithm design (e.g., Fink[11], Horvitz et al.[14], Huang et al.[15], Hutter et al.[16], Kotthoff et al.[18], Leyton-Brown et al.[20]), there has been surprisingly little theoretical analysis of the problem. One possible explanation is that

worst-case analysis, which is the dominant algorithm analysis paradigm in theoretical computer science, is intentionally application agnostic.

This paper demonstrates that application-specific algorithm selection can be usefully modeled as a statistical learning problem, in the spirit of Haussler.[13] We prove that many classes of algorithms have small pseudo-dimension; in these cases, an approximately best-in-class algorithm can be learned from a modest number of representative benchmark instances. We interpret our results in the contexts of learning greedy heuristics, instance feature-based algorithm selection, and parameter tuning in machine learning.

Dimension notions from statistical learning theory have been used almost exclusively to quantify the complexity of classes of prediction functions (e.g. Anthony and Bartlett[2], Haussler[13]). Our results demonstrate that these concepts are useful and relevant in a much broader algorithmic context while also providing a novel formalization of the oft-mentioned but rarely defined "simplicity" of a family of algorithms.

Back to Top

2. Motivating Scenarios

Our learning framework sheds light on several well-known approaches, spanning disparate application domains, to the problem of learning a good algorithm from data. To motivate and provide interpretations of our results, we describe several of these in detail. There are also strong connections between data-driven algorithm design and *self-improving algorithms*,[1,9] where the goal is to design an algorithm that uses a modest amount of space and, given a sequence of independent samples from an unknown input distribution, converges to the optimal algorithm for that distribution; the full version[12] elaborates on these connections.

◆ 2.1. Example 1: greedy heuristic selection

One of the most common and also most challenging motivations for data-driven algorithm design is presented by computationally difficult optimization problems. When the available computing resources are inadequate to solve such a problem exactly, heuristic algorithms must be used. For most hard problems, our understanding of when different heuristics work well remains primitive. For con-creteness, we describe one recent and high-stakes example of this issue, which also aligns well with our model and results in Section 4.

In 2016–2017, the FCC ran a novel double auction to buy back licenses for spectrum from certain television broad-casters and resell them to telecommunication companies for wireless broadband use.[19] The auction generated roughly $10 billion for the US government. The "reverse" (i.e., buy back) phase of the auction determined which stations to buy out and what to pay them. The auction was tasked with buying out sufficiently many stations, so that the remaining stations (who keep their licenses) can be "repacked" into a small number of channels, leaving a target number of channels free to be repurposed for wireless broadband. To first order, the feasible repackings were determined by interference constraints between stations. Computing a repacking, therefore, resembles familiar hard combinatorial problems such as the independent set and graph coloring problems.

The reverse auction deployed by the FCC used a greedy heuristic to decide which stations would keep their licenses and stay on the air. The chosen heuristic favored stations with high value and discriminated against stations that interfered with a large number of other stations. There are many ways of combining these two criteria and no obvious reason to favor one implementation over another. The specific implementation in the FCC auction was justified through trial-and-error experiments using synthetic instances that were thought to be representative. One interpretation of our results in Section 4 is as a post hoc justification of this approach to algorithm design for sufficiently simple collections of algorithms, such as the family of greedy heuristics considered for this FCC auction.

◆ 2.2. Example 2: parameter tuning in optimization and machine learning

Many "algorithms" used in practice are really meta-algorithms, with a large number of free parameters that need to be instantiated by the user. For instance, implementing even the most basic version of gradient descent requires choosing a step size and error tolerance. For a more extreme example, CPLEX, a widely used commercial linear and integer programming solver, comes with a 221-page parameter reference manual describing 135 parameters.

An analogous problem in machine learning is "hyperparameter optimization," where the goal is to tune the parameters of a learning algorithm, so that it learns (from training data) a model with high accuracy on test data, and in particular a model that does not overfit the training data. A simple example is regularized regression, such as ridge regression, where a single parameter governs the trade-off between the accuracy of the learned model on training data and its "complexity." More sophisticated learning algorithms can have many more parameters.

Figuring out the "right" parameter values is notoriously challenging in practice. The CPLEX manual simply advises that "you may need to experiment with them." In machine learning, parameters are often set by discretizing and then applying exhaustive search (a.k.a. "grid search"), perhaps with random subsampling

("random search").[7] When this is computationally infeasible, variants of gradient descent are often used to explore the parameter space, with no guarantee of convergence to a global optimum.

The results in Section 6 can be interpreted as a sample complexity analysis of grid search for the problem of choosing the step size in gradient descent to minimize the expected number of iterations needed for convergence. We view this as a first step toward reasoning more generally about the problem of learning good parameters for machine learning algorithms.

◆ 2.3. Example 3: empirical performance models for SAT solvers

The examples above already motivate choosing an algorithm for a problem based on characteristics of the application domain. A more ambitious and refined approach is to select an algorithm on a *per-instance* (instead of a per-domain) basis. Although it is impossible to memorize the best algorithm for every possible instance, one might hope to use coarse *features* of a problem instance as a guide to which algorithm is likely to work well.

For example, Xu et al.[24] applied this idea to the satisfiability (SAT) problem. Their algorithm portfolio consisted of seven state-of-the-art SAT solvers with incomparable and widely varying running times across different instances. The authors identified a number of instance features, ranging from simple features such as input size and clause/variable ratio to complex features such as Knuth's estimate of the search tree size[17] and the initial rate of progress of local search.[a] The next step involved building an "empirical performance model" (EPM) for each of the seven algorithms in the portfolio—a mapping from instance feature vectors to running time predictions. They then computed their EPMs using labeled training data and a suitable regression model. With the EPMs in hand, it is clear how to perform per-instance algorithm selection: given an instance, compute its features, use the EPMs to predict the running time of each algorithm in the portfolio, and run the algorithm with the smallest predicted running time. Using these ideas (and several optimizations), their "SATzilla" algorithm won numerous medals at the 2007 SAT Competition. Section 5 outlines how to extend our learning framework to reason about EPMs and feature-based algorithm selection.

Back to Top

3. A PAC Learning Model

This section casts the problem of choosing the best algorithm for a poorly understood application domain as one of learning the optimal algorithm with respect to an unknown instance distribution. Section 3.1 formally defines the basic model, and Section 3.2 reviews the relevant preliminaries from statistical learning theory. Sections 4–6 describe the applications of the model to learning greedy heuristics, instance feature-based algorithm selection, and hyperparameter tuning.

◆ 3.1. The basic model

Our basic model consists of the following ingredients.

1. A fixed computational or optimization problem Π. For example, Π could be the problem of computing a maximum-weight independent set of a graph (Section 4) or minimizing a convex function (Section 6).
2. An unknown distribution D over instances $x \in \Pi$.
3. A set A of algorithms for Π. For example, A could be a finite set of SAT solvers or an infinite family of greedy heuristics.
4. A performance measure cost: $A \times \Pi \to [0, H]$, indicating the performance of a given algorithm on a given instance. Two common choices for cost are the running time of an algorithm and, for optimization problems, the objective function value of the solution produced by an algorithm.

The "application-specific" information is encoded by the unknown input distribution D, and the corresponding "application-specific optimal algorithm" A_D is the algorithm that minimizes or maximizes (as appropriate)

$$\mathbf{E}_{x \in D}[\text{cost}(A,x)]$$

over the algorithms $A \in \mathcal{A}$. The *error* of an algorithm $A \in \mathcal{A}$ for a distribution D is

$$\left|\mathbf{E}_{x \sim D}[\text{cost}(A,x)] - \mathbf{E}_{x \sim D}[\text{cost}(A_D,x)]\right|.$$

In our basic model, the goal is:

Learn the application-specific optimal algorithm from data (i.e., samples from D).

More precisely, the learning algorithm is given m i.i.d. samples $x_1, \ldots, x_m \in \Pi$ from D and (perhaps implicitly) the corresponding performance cost(A, x_i) of each algorithm $A \in \mathcal{A}$ on each input x_i. The learning algorithm uses this information to suggest an algorithm $\hat{A} \in \mathcal{A}$ to use on future inputs drawn from D. We seek learning algorithms that almost always output an algorithm of $\mathcal{A}$ that performs almost as well as the optimal algorithm in $\mathcal{A}$ for D—learning algorithms that are probably approximately correct (PAC).

DEFINITION 3.1 A learning algorithm L (ε, δ)-*learns* the optimal algorithm in $\mathcal{A}$ from m samples if, for every distribution D over Π, with probability at least $1-\delta$ over m samples $x_1, \ldots, x_m \sim D$, L outputs an algorithm $\hat{A} \in \mathcal{A}$ with error at most ε.

◆ 3.2. Pseudo-dimension and uniform convergence

PAC learning an optimal algorithm, in the sense of Definition 3.1, reduces to bounding the "complexity" of the class $\mathcal{A}$ of algorithms. We next review the relevant definitions from statistical learning theory.

Let H denote a set of real-valued functions defined on the set X. A finite subset $S = \{x_1, \ldots, x_m\}$ of X is *(pseudo-)shattered* by H if there exist real-valued *witnesses* $r_1, \ldots, r_m$ such that, for each of the 2^m subsets T of S, there exists a function $h \in H$ such that $h(x_i) > r_i$ if and only if $i \in T$ (for $i = 1, 2, \ldots, m$). The *pseudo-dimension* of H is the cardinality of the largest subset shattered by H (or $+\infty$, if arbitrarily large finite subsets are shattered by H). The pseudo-dimension is a natural extension of the VC dimension from binary-valued to real-valued functions.[b]

To bound the sample complexity of accurately estimating the expectation of all functions in H, with respect to an arbitrary probability distribution D on X, it is enough to bound the pseudo-dimension of H.

THEOREM 3.2 (UNIFORM CONVERGENCE (e.g. Anthony and Bartlett[2])) *Let H be a class of functions with domain X and range in* $[0, H]$, *and suppose H has pseudo-dimension d_H. For every distribution D over X, every* $\varepsilon > 0$, *and every* $\delta \in (0, 1]$, *if*

$$m \geq c\left(\frac{H}{\epsilon}\right)^2\left(d_{\mathcal{H}} + \ln\left(\frac{1}{\delta}\right)\right) \qquad (1)$$

for a suitable constant c (independent of all other parameters), then with probability at least $1-\delta$ *over m samples $x_1, \ldots, x_m \sim D$,*

$$\left|\left(\frac{1}{m}\sum_{i=1}^{m} h(x_i)\right) - \mathbf{E}_{x\sim D}[h(x)]\right| < \epsilon$$

for every $h \in H$.

We can identify each algorithm $A \in \mathcal{A}$ with the real-valued function x COST(A, x). Regarding the class $\mathcal{A}$ of algorithms as a set of real-valued functions defined on Π, we can discuss its pseudo-dimension, as defined above. We need one more definition before we can apply our machinery to learn algorithms from $\mathcal{A}$.

DEFINITION 3.3 (EMPIRICAL RISK MINIMIZATION (ERM)) Fix an optimization problem Π, a performance measure COST, and a set of algorithms $\mathcal{A}$. An algorithm L is an *ERM algorithm* if, given any finite subset S of Π, L returns an algorithm from $\mathcal{A}$ with the best average performance on S.

For example, for any Π, COST, and finite $\mathcal{A}$, there is the trivial ERM algorithm that simply computes the average performance of each algorithm on S by brute force and returns the best one. The next corollary follows easily from Definition 3.1, Theorem 3.2, and Definition 3.3.

COROLLARY 3.4 *Fix parameters* $\varepsilon > 0$, $\delta \in (0, 1]$, *a set of problem instances* Π, *and a performance measure* COST. *Let $\mathcal{A}$ be a set of algorithms that has pseudo-dimension d with respect to* Π *and* COST. *Then, any ERM algorithm* $(2\varepsilon, \delta)$ *learns the optimal algorithm in $\mathcal{A}$ from m samples, where m is defined as in* (1).

The same reasoning applies to learning algorithms beyond ERM. For example, with the same assumptions as in Corollary 3.4, an algorithm from $\mathcal{A}$ with approximately optimal (over $\mathcal{A}$) average performance on a set of samples is also approximately optimal with respect to the true input distribution. This observation is particularly useful when the ERM problem is computationally difficult but can be approximated efficiently.

Corollary 3.4 is only interesting if interesting classes of algorithms $\mathcal{A}$ have small pseudo-dimension. In the simple case where $\mathcal{A}$ is finite, as in our example of an algorithm portfolio for SAT (Section 2.3), the pseudo-dimension of $\mathcal{A}$ is trivially at most $\log_2|\mathcal{A}|$. The following sections demonstrate the much less obvious fact that natural infinite classes of algorithms also have small pseudo-dimension.

Back to Top

4. Learning Greedy Heuristics

The goal of this section is to bound the pseudo-dimension of many classes of greedy heuristics such as, as a special case, the family of heuristics relevant for the FCC double auction described in Section 2.1. Throughout this section, the performance measure cost is the objective function value of the solution produced by a heuristic on an instance, where we assume without loss of generality a maximization objective.

4.1. Definitions and examples

Our general definitions are motivated by greedy heuristics for *NP*-hard problems. For example:

1. *Knapsack*. The input is n items with values $v_1, \ldots, v_n$, sizes $s_1, \ldots, s_n$, and a knapsack capacity C. The goal is to compute a subset $S \subseteq \{1,2, \ldots, n\}$ with maximum total value $\Sigma_{i \in S} v_i$, subject to having total size $\Sigma_{i \in S} S_i$ at most C. Two natural greedy heuristics are to greedily pack items (subject to feasibility) in order of nonincreasing value v_i or in order of nonincreasing density v_i/s_i (or to take the better of the two).

2. *Maximum-Weight Independent Set (MWIS)*. The input is an undirected graph $G = (V, E)$ and a nonnegative weight w_v for each vertex $v \in V$. The goal is to compute an independent set—a subset of mutually nonadjacent vertices—with maximum total weight. Two natural greedy heuristics are to greedily choose vertices (subject to feasibility) in order of nonincreasing weight w_v or nonincreasing density $w_v/(1 + \deg(v))$. (The intuition for the denominator is that choosing v "uses up" $1 + \deg(v)$ vertices—v and all of its now-blocked neighbors.) The latter heuristic also has an adaptive variant, where the degree $\deg(v)$ is computed in the subgraph induced by the vertices not yet blocked from consideration, rather than in the original graph.

In general, we consider *object assignment problems*, where the input is a set of n objects with various attributes, and the feasible solutions consist of assignments of the objects to a finite set R, subject to feasibility constraints. The attributes of an object are represented as an element ξ of an abstract set. For example, in the Knapsack problem, ξ encodes the value and size of an object; in the MWIS problem, ξ encodes the weight and (original or residual) degree of a vertex. In the Knapsack and MWIS problems, $R = \{0, 1\}$, indicating whether or not a given object is selected.

By a *greedy heuristic*, we mean algorithms of the following form (cf., the "priority algorithms" of Borodin et al.[8]):

1. While there remain unassigned objects:
    1. Use a *scoring rule* σ (described below) to compute a score $\sigma(\xi_i)$ for each unassigned object i, as a function of its current attributes ξ_i.
    2. For the unassigned object i with the highest score, use an *assignment rule* to assign i a value from R and, if necessary, update the attributes of the other unassigned objects.[c] For concreteness, assume that ties are always resolved lexicographically.

A *scoring rule* assigns a real number to an object as a function of its attributes. Assignment rules that do not modify objects' attributes yield nonadaptive greedy heuristics, which use only the original attributes of each object (such as v_i or v_i/s_i in the Knapsack problem, for instance). In this case, objects' scores can be computed in advance of the main loop of the greedy heuristic. Assignment rules that modify object attributes yield adaptive greedy heuristics, such as the adaptive MWIS heuristic described above.

In a *single-parameter* family of scoring rules, there is a scoring rule of the form $\sigma(\rho, \xi)$ for each parameter value ρ in some interval $I \subseteq \mathbb{R}$. Moreover, σ is assumed to be continuous in ρ for each fixed value of ξ. Natural examples include Knapsack scoring rules of the form v_i/s_i^ρ and MWIS scoring rules of the form $w_v/(1 + \deg(v))^\rho$ for $\rho \in [0, 1]$ or $\rho \in [0, \infty)$. A single-parameter family of scoring rules is *K-crossing* if, for each distinct pair of attributes ρ, ρ', there are at most k values of ρ for which $\sigma(\rho, \xi) = \sigma(\rho, \xi')$. For example, all of the scoring rules mentioned above are 1-crossing rules.

For an example assignment rule, in the Knapsack and MWIS problems, the rule simply assigns i to "1" if it is feasible to do so, and to "0" otherwise. In the adaptive greedy heuristic for the MWIS problem, whenever the assignment rule assigns "1" to a vertex v, it updates the residual degrees of other unassigned vertices (two hops away) accordingly.

We call an assignment rule *β-bounded* if every object i is guaranteed to take on at most β distinct attribute values. For example, an assignment rule that never modifies an object's attribute is 1-bounded. The assignment rule in the adaptive MWIS algorithm is n-bounded, where n is the number of vertices, as it only modifies the degree of a vertex (which lies in $\{0,1,2, \ldots, n-1\}$).

Coupling a single-parameter family of K-crossing scoring rules with a fixed β-bounded assignment rule yields a *(K, β)-single-parameter family of greedy heuristics*. All of our running examples of greedy heuristics are

(1,1)-single-parameter families, except for the adaptive MWIS heuristic, which is a (1,n)-single-parameter family.

◆ 4.2. Upper bound on pseudo-dimension

We next show that every (K, β)-single-parameter family of greedy heuristics has small pseudo-dimension. This result applies to all of the concrete examples mentioned above; additional examples are described in the full version.[12]

THEOREM 4.1 (PSEUDO-DIMENSION OF GREEDY ALGORITHMS) *If A is a (K, β)-single-parameter family of greedy heuristics for an object assignment problem with n objects, then the pseudo-dimension of A is $O(\log(K\beta n))$.*

In particular, all of our running examples are classes of heuristics with pseudo-dimension $O(\log n)$.

PROOF. Recall from the definitions (Section 3.2) that we need to upper bound the size of every set that is shatterable using the greedy heuristics in A. For us, a set is a fixed set of s inputs (each with n objects) $S = \{x_1, \ldots, x_s\}$. For a potential witness $r_1, \ldots, r_s \in \mathbb{R}$, every algorithm $A \in A$ induces a binary labeling of each sample x_i, according to whether $\text{COST}(A,x_i)$ is strictly more than or at most r_i. We proceed to bound from above the number of distinct binary labelings of S induced by the algorithms of A, for any potential witness.

Consider ranging over algorithms $A \in A$—equivalently, over parameter values $\rho \in I$. The trajectory of a greedy heuristic $A \in A$ is uniquely determined by the outcome of the comparisons between the current scores of the unassigned objects in each iteration of the algorithm. Because the family uses a K-crossing scoring rule, for every pair i,j of distinct objects and possible attributes ξ_i, ξ_j with $\xi_i \neq \xi_j$, there are at most K values of ρ for which there is a tie between the score of i (with attributes ξ_i) and that of j (with attributes ξ_j). Because σ is continuous in ρ for every fixed ξ, the relative order of the score of i (with ξ_i) and j (with ξ_j) remains the same in the open interval between two successive values of ρ at which their scores are tied. The upshot is that we can partition I into at most $K + 1$ intervals, such that the outcome of the comparison between i (with attributes ξ_i) and j (with attributes ξ_j) is constant on each interval. In the case that $\xi_i = \xi_j$, because we break ties between equal scores lexicographically, the outcome of the comparison between $\sigma(\rho, \xi_i)$ and $\sigma(\rho, \xi_j)$ is the same for all $\rho \in I$.

Next, the s instances of S contain a total of sn objects. Each of these objects has some initial attributes. Because the assignment rule is β-bounded, there are at most $sn\beta$ object-attribute pairs (i, ξ_i) that could possibly arise in the execution of any algorithm from A on any instance of S. This implies that, ranging across all algorithms of A on all inputs in S, comparisons are only ever made between at most $(sn\beta)^2$ pairs of object-attribute pairs (i.e., between an object i with current attributes ξ_i and an object j with current attributes ξ_j). We call these the *relevant comparisons*.

For each relevant comparison, we can partition I into at most $K + 1$ subintervals such that the comparison outcome is constant (in ρ) in each subinterval. Intersecting the partitions of all of the at most $(sn\beta)^2$ relevant comparisons splits I into at most $(sn\beta)^2K + 1$ subintervals such that *every* relevant comparison is constant in each subinterval. That is, all of the algorithms of A that correspond to the parameter values ρ in such a subinterval execute identically on every input in S. The number of binary labelings of S induced by algorithms of A is trivially at most the number of such subintervals. Our upper bound $(sn\beta)^2K + 1$ on the number of subintervals exceeds 2^s, the requisite number of labelings to shatter S, only if $s = O(\log(K\beta n))$.

Theorem 4.1 and Corollary 3.4 imply that, if K and β are bounded above by a polynomial in n, then an ERM algorithm (ε, δ)-learns the optimal algorithm in A from only $m = \tilde{O}(\frac{H^2}{\varepsilon^2})$ samples,[d] where H is the largest objective function value of a feasible solution output by an algorithm of A on an instance of Π.[e]

Not all classes of algorithms have such a small pseudo-dimension. Theorem 4.1 thus gives one quantifiable sense in which natural greedy algorithms are "simple algorithms."

REMARK 4.2 (EXTENSIONS) Theorem 4.1 is robust, and its proof is easily modified to accommodate various extensions. For example, consider families of greedy heuristics parameterized by d real-valued parameters $\rho_1, \ldots, \rho_d$. Here, an analog of Theorem 4.1 holds with the crossing number K replaced by a more complicated parameter—essentially, the number of connected components of the co-zero set of the difference of two scoring functions (with ξ, ξ' fixed and variables $\rho_1, \ldots, \rho_d$). This number can often be bounded (by a function exponential in d) in natural cases, for example, using Bézout's theorem.

REMARK 4.3 (NONLIPSCHITZNESS) We noted in Section 3.2 that the pseudo-dimension of a finite set A is always at most $\log_2|A|$. This suggests a simple discretization approach to learning the best algorithm from A: take a finite "ε-net" of A and learn the best algorithm in the finite net. (Indeed, Section 6 uses precisely this approach.) The issue is that without some kind of Lipschitz condition—stating that "nearby" algorithms in A

have approximately the same performance on all instances—there is no reason to believe that the best algorithm in the net is almost as good as the best algorithm from all of A. Two different greedy heuristics—say, two MWIS greedy algorithms with arbitrarily close ρ-values—can have completely different executions on an instance. This lack of a Lipschitz property explains why we take care in Theorem 4.1 to bound the pseudo-dimension of the full infinite set of greedy heuristics.

◆ 4.3. Computational considerations

The proof of Theorem 4.1 also demonstrates the presence of an efficient ERM algorithm: the $O((sn\beta)^2)$ relevant comparisons are easy to identify, the corresponding subintervals induced by each are easy to compute (under mild assumptions on the scoring rule), and brute-force search can be used to pick the best of the resulting $O((sn\beta)^2 K)$ algorithms (with one arbitrary representative from each subinterval). This algorithm runs in polynomial time as long as β and K are polynomial in n, and every algorithm of A runs in polynomial time.

For example, for the family of Knapsack scoring rules described above, implementing this ERM algorithm reduces to comparing the outputs of $O(n^2m^2)$ different greedy heuristics (on each of the m sampled inputs), with $m = O(\log n)$. For the adaptive MWIS heuristics, where $\beta = n$, it is enough to compare the sample performance of $O(n^4m^2)$ different greedy algorithms, with $m = O(\log n)$.

Back to Top

5. Feature-Based Algorithm Selection

The previous section studied the problem of choosing a single algorithm for use in an application domain—of using training data to make an informed commitment to a single algorithm from a class A, which is then used for all future instances. A more refined and ambitious approach is to select an algorithm based on both the previous experience *and the current instance to be solved*. This approach assumes, as in the scenario in Section 2.3, that it is feasible to quickly compute some features of an instance and then to select an algorithm as a function of these features.

Throughout this section, we augment the basic model of Section 3.1 with:

1. A set F of possible instance feature values and a map $f: X \to F$ that computes the features of a given instance.[f]

For example, if X is the set of SAT instances, then $f(x)$ might encode the clause/variable ratio of the instance x, Knuth's estimate of the search tree size,[17] etc.

We focus on the case where A is small enough that it is feasible to learn a separate performance prediction model for each algorithm $A \in A$. (The full version also discusses the case of large A.[12]) This is exactly the approach taken in the motivating example of empirical performance models (EPMs) for SAT described in Section 2.3. We then augment the basic model to include a family of performance predictors.

1. A set P of *performance predictors*, with each $p \in P$ a function from F to R.

The goal is to learn, for each algorithm $A \in A$, among all permitted predictors $p \in P$, the one that minimizes some loss function. Like the performance measure COST, we take this loss function as given. For example, for the squared error loss function, for each $A \in A$, we aim to compute the function that minimizes

$$\mathbf{E}_{x \sim D}[(\text{cost}(A, x) - p_A(f(x)))^2]$$

over $p_A \in P$. For a fixed algorithm A, this is a standard regression problem, with domain F, real-valued labels, and a distribution on $F \times$ R induced by D via $x \to (f(x), \text{COST}(A, x))$. Bounding the sample complexity of this learning problem reduces to bounding the pseudo-dimension of P. For standard choices of P, such bounds are well known. For example, suppose the set P is the class of *linear predictors*, with each $p \in P$ having the form $p(f(x)) = a^T f(x)$ for some coefficient vector $a \in \mathrm{R}^d$. (The EPMs used by Xu et al.[24] are linear predictors.) The pseudo-dimension of P is well known to be d. If all functions in P map all possible inputs to $[0, H]$, then Corollary 3.4 implies a sample complexity bound of $\tilde{O}(\frac{H^4}{\varepsilon^2} d)$ for (ε, δ) learning the predictor with minimum expected square error.

Back to Top

6. Choosing the Step Size in Gradient Descent

For our last example, we give sample complexity results for the problem of choosing the best step size in gradient descent. When gradient descent is used in practice, the step size is generally taken much larger than the upper limits suggested by theoretical guarantees and often converges in many fewer iterations than with the step size suggested by theory. This motivates the problem of learning the best step size from examples. We view this as a first step toward reasoning more generally about the problem of learning good hyperparameters for machine learning algorithms.

6.1. Gradient descent preliminaries

Recall the basic gradient descent algorithm for minimizing a function f given an initial point z_0 over $\mathbb{R}^n$:

1. Initialize $z := z_0$.
2. While $\|\nabla f(z)\|_2 > v$:
    1. $z := z - \rho \cdot \nabla f(z)$.

We take the error tolerance v as given and focus on the more interesting parameter, the step size ρ. Bigger values of ρ have the potential to make more progress in each step but run the risk of overshooting a minimum of f.

We instantiate the basic model (Section 3.1) to study the problem of learning the best step size. There is an unknown distribution D over instances, where an instance $x \in \Pi$ consists of a function f and an initial point z_0. Each algorithm A_ρ of A is the basic gradient descent algorithm above, with some choice ρ of a step size drawn from some fixed interval $[\rho_l, \rho_u] \subset (0, \infty)$. The performance measure $\text{COST}(A, x)$ is the number of iterations (i.e., steps) taken by the algorithm for the instance x. Because gradient descent is translation invariant, we can assume for the sake of analysis that 0 is a minimum of f, with $f(0) = 0$. (We consider only instances that have a global minimum.)

To obtain positive results, we impose three further assumptions on problem instances, analogous to those used in the classical convergence analysis of gradient descent for smooth and strongly convex functions.

A1. Every function f is L-smooth for a known L, meaning that f is everywhere differentiable with $\|\nabla f(z_1) - \nabla f(z_2)\| \le L\|z_1 - z_2\|$ for all z_1 and z_2. (Every norm in this section is the l_2 norm.)

A2. Every function f is m-strongly convex for a known $m \le L$, meaning that it is continuously differentiable with $f(z_2) \ge f(z_1) + \nabla f(z_1)^T(z_2 - z_1) + \frac{m}{2}\|z_2 - z_1\|^2$ for all z_1 and z_2.

A3. The magnitudes of the initial points are bounded, with $\|z_0\| \le Z$ for some known constant $Z > v$.

These assumptions imply a "guaranteed progress toward 0" property: There is a known constant $\gamma \in (0, 1)$ such that $\|z - \rho\nabla f(z)\| \le (1 - \gamma)\|z\|$ for all $\rho \in [\rho_l, \rho_u]$. (The standard analysis of gradient descent implies that $\gamma \ge \rho m$ for every $\rho \le 2/(m + L)$ over this class of functions.) Our assumptions imply that gradient descent halts within $H := \log_{1/(1-\gamma)}(\frac{LZ}{v})$ iterations.

6.2. Technical lemma

The heart of the analysis is the following technical lemma, which bounds how far two gradient descent paths with different step sizes can diverge when initialized at the same point. Define $g(z, \rho) := z - \rho\nabla f(z)$ as the result of taking a single gradient descent step from the point z with the step size ρ, and let $g^j(z, \rho)$ denotes the result of taking j gradient descent steps.

LEMMA 6.1 *With assumptions (A1)–(A3), for every starting point z, iteration number j, and step sizes $\rho \le \eta$,*

$$\|g^j(z, \rho) - g^j(z, \eta)\| \le (\eta - \rho)\frac{c_\rho^j LZ}{\gamma},$$

where $c_\rho = \max\{1, L\rho - 1\} \ge 1$ is a constant that is a function only of L and ρ.

The most important point is that the right-hand side of the inequality in Lemma 6.1 goes to 0 with the step size difference $\eta - \rho$. The proof of Lemma 6.1 can be found in the full version.[12]

Lemma 6.1 easily implies a Lipschitz-type condition on $\text{COST}(A_\rho, x)$ as a function of ρ.

LEMMA 6.2 *For every input x satisfying (A1)–(A3) and step sizes $\rho \le \eta$ with $\eta - \rho \le \frac{\gamma v}{mLZ} c_\rho^{-H}$,*

$$|\text{cost}(A_\rho, x) - \text{cost}(A_\eta, x)| \le \lceil \log_{1/(1-\gamma)}(1 + \frac{L}{m}) \rceil.$$

PROOF. Suppose $\text{COST}(A_\eta, x) \le \text{COST}(A_\rho, x)$; the argument in the other case is similar. Let $j = \text{COST}(A_\eta, x)$, and recall that $j \le H$. By the stopping condition of gradient descent, $\|\nabla g^j(z_0, \eta)\| \le v$. By the m-strong convexity of f, $\|g^j(z_0, \eta)\| \le \frac{v}{m}$. By Lemma 6.1 and the assumed gap between ρ and η,

$$\|g^j(z_0, \rho)\| \le \frac{v}{m} + \gamma\frac{v}{m} = (1 + \gamma)\frac{v}{m}.$$

Set $\tau = \lceil \log_{1/(1-\gamma)}(1+\frac{L}{m}) \rceil$. By the guaranteed progress condition,

$$\| g^{j+\tau}(z_0, \rho) \| \leq \frac{\nu}{L}.$$

By L-smoothness, $\|\nabla g^{j+\tau}(z_0, \rho)\| \leq \upsilon$. This implies that gradient descent with step size ρ requires at most τ more iterations to converge than with step size η.

◆ 6.3. Learning the best step size

We can now apply the discretization approach suggested by Remark 4.3. Let $K = \frac{\gamma^2 \nu}{mLZ} c_{\rho_u}^{-H}$. Let N denotes the set of all integer multiples of K that lie in the interval $[\rho_l, \rho_u]$ of possible step sizes. Note that $|N| \leq \rho_u/K + 1$. We can now state the main result of this section:

THEOREM 6.3 (LEARNABILITY OF STEP SIZE) *There is a learning algorithm that* $(\lceil \log_{1/(1-\gamma)}(1+\frac{L}{m}) \rceil + \epsilon, \delta)$ *learns the optimal algorithm in A using* $m = \tilde{O}(H^3/\varepsilon^2)$ *samples from D.*[g]

PROOF. Because $A_N = \{A_\rho : \rho \in N\}$ is a finite set, its pseudo-dimension is at most $\log_2|N|$. Let L_N denotes the ERM algorithm for this set. Corollary 3.4 implies that L_N (ε, δ)-learns the optimal algorithm in A_N using $m = \tilde{O}(H^2 \log |N| /\varepsilon^2)$ samples. Because $\log_2|N| = \tilde{O}(H)$, m is $\tilde{O}(H^3/\varepsilon^2)$.

Now, Lemma 6.2 implies that for every ρ, there is an $\eta \in N$ such that, for every input distribution D, the difference in expected costs of A_η and A_ρ is at most $\lceil \log_{1/(1-\gamma)}(1+\frac{L}{m}) \rceil$. Thus, L_N $(\lceil \log_{1/(1-\gamma)}(1+\frac{L}{m}) \rceil + \epsilon, \delta)$-learns the optimal algorithm in A using $\tilde{O}(H^3/\varepsilon^2)$ samples.

Back to Top

7. Conclusion

Empirical work on data-driven algorithm design has far outpaced theoretical analysis of the problem, and this paper takes an initial step toward redressing this imbalance. We formulated the problem as one of learning a best-in-class algorithm with respect to an unknown input distribution. Many state-of-the-art empirical approaches to the problem map naturally to our learning framework. This paper demonstrates that many well-studied classes of algorithms have small pseudodimension, and thus, it is possible to learn a near-optimal algorithm from a relatively modest number of representative benchmark instances.

Our work suggests numerous wide-open research directions worthy of further study. For example:

1. Which other classes of algorithms have small pseudo-dimension? There has been recent progress on this question for local search algorithms,[12] clustering and partitioning algorithms,[4] auctions and mechanisms,[5, 6, 21, 22] and mathematical programs.[3]
2. In the *online* version of data-driven algorithm design, instances to a problem arrive one by one. The goal is to choose an algorithm to use at each time step, before the instance at that time step arrives, so that the average performance of the chosen algorithms is close to that of the best fixed algorithm in hindsight. When does such an online learning algorithm exist? The full version of this paper[12] gives positive results for classes of greedy algorithms; these are generalized in Cohen-Addad and Kanade[10] and further in Balcan et al.[3]
3. When is it possible to learn a near-optimal algorithm using only a polynomial amount of computation, ideally with a learning algorithm that is better than brute-force search? Alternatively, are there (conditional) lower bounds stating that brute-force search is necessary for learning? See Roughgardern and Wang[23] for progress on these questions for learning revenue-maximizing auctions.
4. Are there any nontrivial relationships between statistical learning measures of the complexity of an algorithm class and more traditional computational complexity measures?

Back to Top

Acknowledgments

This research was supported in part by NSF awards CCF-1215965 and CCF-1524062.

Back to Top

References

1. Ailon, N., Chazelle, B., Clarkson, K.L., Liu, D., Mulzer, W., Seshadhri, C. Self-improving algorithms. *SIAM J. Comput. 2*, 40 (2011) 350–375.

2. Anthony, M., Bartlett, P.L. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 1999.

3. Balcan, M.-F., Dick, T., Vitercik, E. Dispersion for data-driven algorithm design, online learning, and private optimization. In *Proceedings of the 59th Annual Symposium on Foundations of Computer Science (FOCS)* (2018), 603–614.

4. Balcan, M.-F., Nagarajan, V., Vitercik, E., White, C. Learning-theoretic foundations of algorithm configuration for combinatorial partitioning problems. In *Proceedings of the 30th Conference on Learning Theory (COLT)* (2017), 213–274.

5. Balcan, M.-F., Sandholm, T., Vitercik, E. Sample complexity of automated mechanism design. In *Proceedings of the 29th Annual Conference on Neural Information Processing Systems (NeurIPS)* (2016), 2083–2091.

6. Balcan, M.-F., Sandholm, T., Vitercik, E. A general theory of sample complexity for multi-item profit maximization. In *Proceedings of the 19th ACM Conference on Economics and Computation (EC)* (2018), 73–174.

7. Bergstra, J., Bengio, Y. Random search for hyper-parameter optimization. *J. Mach. Learn. Res. 1*, 13 (2012), 281–305.

8. Borodin, A., Nielsen, M.N., Rackoff, C. (Incremental) priority algorithms. *Algorithmica 4*, 37 (2003), 295–326.

9. Clarkson, K.L., Mulzer, W., Seshadhri, C. Self-improving algorithms for coordinatewise maxima and convex hulls. *SIAM J. Comput. 2*, 43 (2014), 617–653.

10. Cohen-Addad, V., Kanade, V. Online optimization of smoothed piecewise constant functions. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)* (2017), 412–420.

11. Fink, E. How to solve it automatically: Selection among problem solving methods. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems* (1998), 128–136.

12. Gupta, R., Roughgarden, T. A PAC approach to application-specific algorithm selection. *SIAM J. Comput. 3*, 46 (2017), 992–1017.

13. Haussler, D. Decision theoretic generalizations of the PAC model for neural net and other learning applications. *Inform. Comput. 1*, 100 (1992), 78–150.

14. Horvitz, E., Ruan, Y., Gomes, C.P., Kautz, H.A., Selman, B., Chickering, D.M. A Bayesian approach to tackling hard computational problems. In *Proceedings of the Conference in Uncertainty in Artificial Intelligence (UAI)* (2001), 235–244.

15. Huang, L., Jia, J., Yu, B., Chun, B., Maniatis, P., Naik, M. Predicting execution time of computer programs using sparse polynomial regression. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)* (2010), 883–891.

16. Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K. Algorithm runtime prediction: Methods & evaluation. *Artif. Intell.*, 206 (2014), 79–111.

17. Knuth, D.E. Estimating the efficiency of backtrack programs. *Math. Comput.*, 29 (1975), 121–136.

18. Kotthoff, L., Gent, I.P., Miguel, I. An evaluation of machine learning in algorithm selection for search problems. *AI Commun. 3*, 25 (2012), 257–270.

19. Leyton-Brown, K., Milgrom, P., Segal, I. Economics and computer science of a radio spectrum reallocation. *Proc. Natl. Acad. Sci. 28*, 114 (2017), 7202–7209.

20. Leyton-Brown, K., Nudelman, E., Shoham, Y. Empirical hardness models: Methodology and a case study on combinatorial auctions. *J. ACM 4*, 56 (2009).

21. Morgenstern, J., Roughgarden, T. On the pseudo-dimension of nearly optimal auctions. In *Proceedings of the 28th Annual Conference on Neural Information Processing Systems (NeurIPS)* (2015), 136–144.

22. Morgenstern, J., Roughgarden, T. Learning simple auctions. In *Proceedings of the 29th Conference on Learning Theory (COLT)* (2016), 1298–1318.

23. Roughgarden, T., Wang, J.R. Minimizing regret with multiple reserves. In *Proceedings of the 17th ACM Conference on Economics and Computation* (*EC*) (2016), 601–616.

24. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32 (2008), 565–606.

Back to Top

Authors

Rishi Gupta (rishig@cs.stanford.edu), Department of Computer Science, Stanford University, Stanford, CA, USA.

Tim Roughgarden (tr@cs.columbia.edu), Department of Computer Science, Columbia University, New York, USA.

Back to Top

Footnotes

a. It is important, of course, that computing the features of an instance is an easier problem than solving it.

b. The *fat shattering dimension* is another common extension of the VC dimension to real-valued functions. It is a weaker condition, in that the fat shattering dimension of H is always at most the pseudo-dimension of H, and it is sufficient for sample complexity bounds. Most of our arguments give the same upper bounds on both notions, so we present the stronger statements.

c. We assume that there is always at least one choice of assignment that respects the feasibility constraints; this holds for all of our motivating examples.

d. The notation $\tilde{O}(\cdot)$ suppresses logarithmic factors.

e. Alternatively, the dependence of m on H can be removed if learning error εH (rather than ε) can be tolerated—for example, if the optimal objective function value is expected to be proportional to H anyways.

f. Defining a good feature set is a notoriously challenging and important problem, but it is beyond the scope of our model—we take the set F and map f as given.

g. We use the $\tilde{O}(\cdot)$ notation to suppress logarithmic factors in Z, L, m, ρ_u, $\frac{1}{\nu}$, $\frac{1}{\nu'}$ and $\frac{1}{\nu''}$.

A preliminary version of this article was published in the *Proceedings of the 7th Annual Innovations in Theoretical Computer Science Conference*, January 2016. A full version with the title "A PAC Approach to Application-Specific Algorithm Selection" was published in *SIAM Journal on Computing*, June 2017.

No entries found