SIAM J. COMPUT.
Vol. 46, No. 3, pp. 992–1017

A PAC APPROACH TO APPLICATION-SPECIFIC ALGORITHM SELECTION*

RISHI GUPTA† AND TIM ROUGHGARDEN†

Abstract. 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. While there is a large body of literature on empirical approaches to selecting the best algorithm for a given application domain, there has been surprisingly little theoretical analysis of the problem. This paper adapts concepts from statistical and online learning theory to reason about application-specific algorithm selection. Our models capture several state-of-the-art empirical and theoretical approaches to the problem, ranging from self-improving algorithms to empirical performance models, and our results identify conditions under which these approaches are guaranteed to perform well. We present one framework that models algorithm selection as a statistical learning problem, and our work here shows that dimension notions from statistical learning theory, historically used to measure the complexity of classes of binary- and real-valued functions, are relevant in a much broader algorithmic context. We also study the online version of the algorithm selection problem, and give possibility and impossibility results for the existence of no-regret learning algorithms.

Key words. algorithm selection, parameter tuning, PAC learning, online learning, meta-algorithms

AMS subject classification. 68Q32

DOI. 10.1137/15M1050276

1. Introduction.

Rigorously comparing algorithms is hard. The most basic reason for this is that 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, instances of interest commonly possess domain-specific features that defy formal articulation. Solving a problem in practice can require selecting an algorithm that is optimized for the specific application domain, even though the special structure of its instances is not well understood. While there is a large literature, spanning numerous communities, on empirical approaches to algorithm selection (e.g. [13, 18, 16, 17, 22, 24]), there has been surprisingly little theoretical analysis of the problem.

*Received by the editors November 30, 2015; accepted for publication (in revised form) January 25, 2017; published electronically June 13, 2017. A preliminary version of this paper appeared in the Proceedings of the 7th Innovations in Theoretical Computer Science Conference, January 2016.
http://www.siam.org/journals/sicomp/46-3/M105027.html
Funding: This research was supported in part by NSF awards CCF-1215965 and CCF-1524062.
†Dept. of Computer Science, Stanford University, Stanford, CA 94305 (rishig@cs.stanford.edu, tim@cs.stanford.edu).

One possible explanation is that worst-case analysis, which is the dominant algorithm analysis paradigm in theoretical computer science, is deliberately application-agnostic.

This paper demonstrates that application-specific algorithm selection can be usefully modeled as a learning problem. Our models are straightforward to understand, but also expressive enough to capture several existing approaches in the theoretical computer science and AI communities, ranging from the design and analysis of self-improving algorithms [1] to the application of empirical performance models [18].

We present one framework that models algorithm selection as a statistical learning problem in the spirit of Haussler [15]. We prove that many useful families of algorithms, including broad classes of greedy and local search heuristics, have small pseudo-dimension and hence low generalization error. Previously, the pseudo-dimension (and the Vapnik–Chervonenk is (VC) dimension, fat shattering dimension, etc.) has been used almost exclusively to quantify the complexity of classes of prediction functions (e.g., [15, 2]).[1] Our results demonstrate that this concept is useful and relevant in a much broader algorithmic context. It also offers a novel approach to formalizing the oft-mentioned but rarely defined "simplicity" of a family of algorithms.

We also study regret-minimization in the online version of the algorithm selection problem. We show that the "non-Lipschitz" behavior of natural algorithm classes precludes learning algorithms that have no regret in the worst case, and prove positive results under smoothed analysis-type assumptions.

Paper organization. Section 2 outlines a number of concrete problems that motivate the present work, ranging from greedy heuristics to SAT (satisfiability problem) solvers, and from self-improving algorithms to parameter tuning. The reader interested solely in the technical development can skip this section with little loss. Section 3 models the task of determining the best application-specific algorithm as a PAC (probably approximately correct) learning problem, and brings the machinery of statistical learning theory to bear on a wide class of problems, including greedy heuristic selection, sorting, and gradient descent step size selection. A time-limited reader can glean the gist of our contributions from subsection 3.1–3.3.3. Section 4 considers the problem of learning an application-specific algorithm online, with the goal of minimizing regret. Subsection 4.2 and 4.3 present negative and positive results for worst-case and smoothed instances, respectively. Section 5 concludes with a number of open research directions.

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.

2.1. Example #1: Greedy heuristic selection. One of the most common and also most challenging motivations for algorithm selection 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 concreteness, we describe one current and high-stakes example

[1] A few exceptions: Srebro and Ben-David [32] use the pseudo-dimension to study the problem of learning a good kernel for use in a support vector machine, Long [26] parameterizes the performance of the randomized rounding of packing and covering linear programs by the pseudo-dimension of a set derived from the constraint matrix, and Mohri and Medina [28] and Morgenstern and Roughgarden [29] use dimension notions from learning theory to bound the sample complexity of learning approximately revenue-maximizing truthful auctions.

of this issue, which also aligns well with our model and results in subsection 3.3. The computing and operations research literature has many similar examples.

The Federal Communications Commission (FCC) is currently (in 2016) running a novel double auction to buy back licenses for spectrum from certain television broadcasters and resell them to telecommunication companies for wireless broadband use. The auction is expected to generate over 20 billion dollars for the US government [7]. The "reverse" (i.e., buyback) phase of the auction must determine which stations to buy out (and what to pay them). The auction is 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 are determined by interference constraints between stations. Computing a repacking therefore resembles familiar hard combinatorial problems like the independent set and graph coloring problems. The reverse auction uses a greedy heuristic to compute the order in which stations are removed from the reverse auction (removal means the station keeps its license) [27]. The chosen heuristic favors stations with high value, and discriminates against stations that interfere with a large number of other stations.[2] There are many ways of combining these two criteria, and no obvious reason to favor one specific implementation over another. The specific implementation in the FCC auction has been justified through trial-and-error experiments using synthetic instances that are thought to be representative [27]. One interpretation of our results in subsection 3.3 is as a post hoc justification of this exhaustive approach for sufficiently simple classes of algorithms, including the greedy heuristics considered for this FCC auction.

2.2. Example #2: Self-improving algorithms. The area of *self-improving algorithms* was initiated by Ailon et al. [1], who considered sorting and clustering problems. Subsequent work [11, 9, 10] studied several problems in low-dimensional geometry, including the maxima and convex hull problems. For a given problem, the goal is to design an algorithm that, given a sequence of i.i.d. samples from an unknown distribution over instances, converges to the optimal algorithm for that distribution. In addition, the algorithm should use only a small amount of auxiliary space. For example, for sorting independently distributed array entries, the algorithm by Ailon et al. [1] solves each instance (on n numbers) in $O(n \log n)$ time, uses space $O(n^{1+c})$ (where $c > 0$ is an arbitrarily small constant), and after a polynomial number of samples has expected running time within a constant factor of that of an information-theoretically optimal algorithm for the unknown input distribution. Subsection 3.4 reinterprets self-improving algorithms via our general framework.

2.3. Example #3: 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 in the most basic version of gradient descent requires choosing a step size and error tolerance. For a more extreme version, CPLEX, a widely-used commercial linear and integer programming solver, comes with a 221-page parameter reference manual describing 135 parameters [34].

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

[2] Analogously, greedy heuristics for the maximum-weight independent set problem favor vertices with higher weights and with lower degrees [30]. Greedy heuristics for welfare maximization in combinatorial auctions prefer bidders with higher values and smaller demanded bundles [23].

(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 brute-force search (a.k.a. "grid search"), perhaps with random subsampling ("random search") [4]. When this is computationally infeasible, variants of gradient descent are often used to explore the parameter space, with no guarantee of converging to a global optimum.

The results in subsection 3.6 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.

2.4. Example #4: Empirical performance models for SAT algorithms. The examples above already motivate selecting 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. While it's 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. [33] 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 like input size and clause/variable ratio, to complex features like Knuth's estimate of the search tree size [21] and the rate of progress of local search probes.[3] 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.[4] Subsection 3.5 outlines how to extend our PAC learning framework to reason about EPMs and feature-based algorithm selection.

3. PAC learning an application-specific algorithm. This section casts the problem of selecting the best algorithm for a poorly understood application domain as one of learning the optimal algorithm with respect to an unknown instance distribution. Subsection 3.1 formally defines the basic model, subsection 3.2 reviews relevant preliminaries from statistical learning theory, subsection 3.3 bounds the

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

[4] See Xu et al. [35] for details on the latest generation of their solver.

pseudo-dimension of many classes of greedy and local search heuristics, subsection 3.4 re-interprets the theory of self-improving algorithms via our framework, subsection 3.5 extends the basic model to capture empirical performance models and feature-based algorithm selection, and subsection 3.6 studies step size selection in gradient descent.

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 computing a maximum-weight independent set of a graph (subsection 2.1), or sorting n elements (subsection 2.2).
2. An unknown distribution $\mathcal{D}$ over instances $x \in \Pi$.
3. A set $\mathcal{A}$ of algorithms for Π; see subsections 3.3 and 3.4 for concrete examples.
4. A performance measure $\textsc{cost} : \mathcal{A} \times \Pi \to [0, H]$ indicating the performance of a given algorithm on a given instance. Two common choices for $\textsc{cost}$ 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 $\mathcal{D}$, and the corresponding "application-specific optimal algorithm" $A_{\mathcal{D}}$ is the algorithm that minimizes or maximizes (as appropriate) $\mathbf{E}_{x \in \mathcal{D}}[\textsc{cost}(A, x)]$ over $A \in \mathcal{A}$. The *error* of an algorithm $A \in \mathcal{A}$ for a distribution $\mathcal{D}$ is

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

In our basic model, the goal is

> *Learn the application-specific optimal algorithm from data (i.e., samples from $\mathcal{D}$).*

More precisely, the learning algorithm is given m i.i.d. samples $x_1, \ldots, x_m \in \Pi$ from $\mathcal{D}$, and (perhaps implicitly) the corresponding performance $\textsc{cost}(A, x_i)$ 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 $\mathcal{D}$. 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 $\mathcal{D}$.

Definition 3.1. *A learning algorithm L (ϵ, δ)-learns the optimal algorithm in $\mathcal{A}$ from m samples if, for every distribution $\mathcal{D}$ over Π, with probability at least $1 - \delta$ over m samples $x_1, \ldots, x_m \sim \mathcal{D}$, 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 $\mathcal{H}$ 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 $\mathcal{H}$ 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 $h \in \mathcal{H}$ such that $h(x_i) > r_i$ if and only if $i \in T$ (for $i = 1, 2, \ldots, m$). The *pseudo-dimension* of $\mathcal{H}$ is the cardinality of the largest subset shattered by $\mathcal{H}$ (or $+\infty$, if arbitrarily large finite subsets are shattered by $\mathcal{H}$). The pseudo-dimension is a natural extension of the VC dimension from binary-valued to real-valued functions.[5]

[5] 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 $\mathcal{H}$ is always at most the pseudo-dimension of $\mathcal{H}$, that is still sufficient for sample complexity bounds. Most of our arguments give the same upper bounds on pseudo-dimension and fat shattering dimension, so we present the stronger statements.

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

THEOREM 3.2 (Uniform convergence (e.g. [2])). *Let $\mathcal{H}$ be a class of functions with domain X and range in $[0, H]$, and suppose $\mathcal{H}$ has pseudo-dimension $d_{\mathcal{H}}$. For every distribution $\mathcal{D}$ over X, every $\epsilon > 0$, and every $\delta \in (0, 1]$, if*

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

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

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

for every $h \in \mathcal{H}$.

We can identify each algorithm $A \in \mathcal{A}$ with the real-valued function $x \mapsto \text{COST}(A, x)$. 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 (arbitrary) 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 $\epsilon > 0$, $\delta \in (0, 1]$, 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 $(2\epsilon, \delta)$-learns the optimal algorithm in $\mathcal{A}$ from m samples, where m is defined as in* (1).

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 (subsection 2.4), 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.

Remark 3.5 (Computational efficiency). The present work focuses on the sample complexity rather than the computational aspects of learning, so outside of a few remarks we won't say much about the existence or efficiency of ERM in our examples. A priori, an infinite class of algorithms may not admit any ERM algorithm at all, though all of the examples in this paper do have ERM algorithms under mild assumptions.

3.3. Application: Greedy heuristics and extensions. The goal of this section is to bound the pseudo-dimension of many classes of greedy heuristics including, as a special case, the family of heuristics relevant for the FCC double auction described in subsection 2.1. It will be evident that analogous computations are possible

for many other classes of heuristics, and we provide several extensions in subsection 3.3.4 to illustrate this point. 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.

3.3.1. Definitions and examples. Our general definitions are motivated by greedy heuristics for (NP-hard) problems like the following; the reader will have no difficulty coming up with additional natural examples.

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 $\sum_{i \in S} v_i$, subject to having total size $\sum_{i \in S} s_i$ 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; see subsection 3.3.4).
2. *Maximum-weight independent set (MWIS).* The input is an undirected graph $G = (V, E)$ and a non-negative weight w_v for each vertex $v \in V$. The goal is to compute the independent set—a subset of mutually non-adjacent 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 neighbors.) The latter heuristic also has a (superior) 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.[6]
3. *Machine scheduling.* This is a family of optimization problems, where n jobs with various attributes (processing time, weight, deadline, etc.) need to be assigned to m machines, perhaps subject to some constraints (precedence constraints, deadlines, etc.), to optimize some objective (makespan, weighted sum of completion times, number of late jobs, etc.). A typical greedy heuristic for such a problem considers jobs in some order according to a score derived from the job parameters (e.g., weight divided by processing time), subject to feasibility, and always assigns the current job to the machine that currently has the lightest load (again, subject to feasibility).

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. In machine scheduling problems, R could be $\{1, 2, \ldots, m\}$, indicating the machine to which a job is assigned, or a richer set that also keeps track of the job ordering on each machine.

By a *greedy heuristic*, we mean algorithms of the following form (cf. the "priority algorithms" of Borodin et al. [5]):

1. While there remain unassigned objects,
   (a) Use a *scoring rule* σ (see below) to compute a score $\sigma(\xi_i)$ for each unassigned object i, as a function of its current attributes ξ_i.

[6] An equivalent description is, whenever a vertex v is added to the independent set, delete v and its neighbors from the graph, and recurse on the remaining graph.

(b) 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.[7] 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 (like 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 κ-*crossing* if, for each distinct pair of attributes ξ, ξ', there are at most κ 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. A typical machine scheduling assignment rule assigns the current job to the machine with the lightest load. 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 attributes is 1-bounded. The assignment rule in the adaptive MWIS algorithm is n-bounded, since it only modifies the degree of a vertex (which lies in $\{0, 1, 2 \ldots, n-1\}$).

Coupling a single-parameter family of κ-crossing scoring rules with a fixed β-bounded assignment rule yields a (κ, 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.

3.3.2. Upper bound on pseudo-dimension. We next show that every (κ, β)-single-parameter family of greedy heuristics has small pseudo-dimension. This result applies to all of the concrete examples mentioned above, and it is easy to come up with other examples (for the problems already discussed, and for additional problems).

THEOREM 3.6 (Pseudo-dimension of greedy algorithms). *If $\mathcal{A}$ denotes a (κ, β)-single-parameter family of greedy heuristics for an object assignment problem with n objects, then the pseudo-dimension of $\mathcal{A}$ is $O(\log(\kappa\beta n))$.*

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

Proof. Recall from the definitions (subsection 3.2) that we need to upper bound the size of every set that is shatterable using the greedy heuristics in $\mathcal{A}$. 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 $A \in \mathcal{A}$ induces a binary labeling of each sample x_i,

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

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 labellings of S induced by the algorithms of $\mathcal{A}$, for any potential witness.

Consider ranging over algorithms $A \in \mathcal{A}$; equivalently, over parameter values $\rho \in I$. The trajectory of a greedy heuristic $A \in \mathcal{A}$ is uniquely determined by the outcome of the comparisons between the current scores of the unassigned objects in each iteration of the algorithm. Since the family uses a κ-crossing scoring rule, for every pair i, j of distinct objects and possible attributes ξ_i, ξ_j, there are at most κ 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). Since σ is continuous in ρ, 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 $\kappa + 1$ 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.[8]

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 $\mathcal{A}$ on any instance of S. This implies that, ranging across all algorithms of $\mathcal{A}$ 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 $\kappa + 1$ 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 $(sn\beta)^2\kappa + 1$ subintervals such that *every* relevant comparison is constant in each subinterval. That is, all of the algorithms of $\mathcal{A}$ that correspond to the parameter values ρ in such a subinterval execute identically on every input in S. The number of binary labellings of S induced by algorithms of $\mathcal{A}$ is trivially at most the number of such subintervals. Our upper bound $(sn\beta)^2\kappa + 1$ on the number of subintervals exceeds 2^s, the requisite number of labellings to shatter S, only if $s = O(\log(\kappa\beta n))$. ∎

Theorem 3.6 and Corollary 3.4 imply that, if κ and β are bounded above by a polynomial in n, then an ERM algorithm (ϵ, δ)-learns the optimal algorithm in $\mathcal{A}$ from only $m = \tilde{O}(\frac{H^2}{\epsilon^2})$ samples,[9] where H is the largest objective function value of a feasible solution output by an algorithm of $\mathcal{A}$ on an instance of Π.[10]

We note that Theorem 3.6 gives a quantifiable sense in which natural greedy algorithms are indeed "simple algorithms." Not all classes of algorithms have such a small pseudo-dimension; see also the next section for further discussion.[11]

[8] This argument assumes that $\xi_i \neq \xi_j$. If $\xi_i = \xi_j$, then because we break ties between equal scores lexicographically, the outcome of the comparison between $\sigma(\xi_i)$ and $\sigma(\xi_j)$ is in fact constant on the entire interval I of parameter values.

[9] The notation $\tilde{O}(\cdot)$ suppresses logarithmic factors.

[10] 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 anyway.

[11] When the performance measure COST is solution quality, as in this section, one cannot identify "simplicity" with "low pseudo-dimension" without caveats: strictly speaking, the set $\mathcal{A}$ containing only the optimal algorithm for the problem has pseudo-dimension 1. When the problem Π is NP-hard and $\mathcal{A}$ consists only of polynomial-time algorithms (and assuming $P \neq NP$), the pseudo-dimension is a potentially relevant complexity measure for the heuristics in $\mathcal{A}$.

Remark 3.7 (Non-Lipschitzness). We noted in subsection 3.2 that the pseudo-dimension of a finite set $\mathcal{A}$ is always at most $\log_2 |\mathcal{A}|$. This suggests a simple discretization approach to learning the best algorithm from $\mathcal{A}$: take a finite "ϵ-net" of $\mathcal{A}$ and learn the best algorithm in the finite net. (Indeed, subsection 3.6 uses precisely this approach.) The issue is that without some kind of Lipschitz condition—stating that "nearby" algorithms in $\mathcal{A}$ have approximately the same performance on all instances—there's no reason to believe that the best algorithm in the net is almost as good as the best algorithm from all of $\mathcal{A}$. Two different greedy heuristics—two MWIS greedy algorithms with arbitrarily close ρ-values, say—can have completely different executions on an instance. This lack of a Lipschitz property explains why we take care in Theorem 3.6 to bound the pseudo-dimension of the full infinite set of greedy heuristics.[12]

3.3.3. Computational considerations. The proof of Theorem 3.6 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 $O((sn\beta)^2\kappa)$ algorithms (an arbitrary one from each subinterval). This algorithm runs in polynomial time as long as β and κ are polynomial in n, and every algorithm of $\mathcal{A}$ 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 $O(n^2m)$ 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 $O(n^4m)$ different greedy algorithms, with $m = O(\log n)$.

3.3.4. Extensions: Multiple algorithms, multiple parameters, and local search. Theorem 3.6 is robust and its proof is easily modified to accommodate various extensions. For a first example, consider algorithms than run q different members of a single-parameter greedy heuristic family and return the best of the q feasible solutions obtained.[13] Extending the proof of Theorem 3.6 yields a pseudo-dimension bound of $O(q \log(\kappa\beta n))$ for the class of all such algorithms.

For a second example, consider families of greedy heuristics parameterized by d real-valued parameters $\rho_1, \ldots, \rho_d$. Here, an analog of Theorem 3.6 holds with the crossing number κ 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 (see, e.g., [14]).

For a final extension, we sketch how to adapt the definitions and results of this section from greedy to local search heuristics. The input is again an object assignment problem (see subsection 3.3.1), along with an initial feasible solution (i.e., an assignment of objects to R, subject to feasibility constraints). By a *k-swap local search heuristic*, we mean algorithms of the following form:

1. Start with arbitrary feasible solution.

[12] The ϵ-net approach has the potential to work for greedy algorithms that choose the next object using a softmax-type rule, rather than deterministically as the unassigned object with the highest score.

[13] For example, the classical $\frac{1}{2}$-approximation for Knapsack has this form (with $q = 2$).

2. While the current solution is not locally optimal,
   (a) Use a *scoring rule* σ to compute a score $\sigma(\{\xi_i : i \in K\})$ for each set of objects K of size k, where ξ_i is the current attribute of object i.
   (b) For the set K with the highest score, use an *assignment rule* to reassign each $i \in K$ to a value from R. If necessary, update the attributes of the appropriate objects. (Again, assume that ties are resolved lexicographically.)

We assume that the assignment rule maintains feasibility, so that we have a feasible assignment at the end of each execution of the loop. We also assume that the scoring and assignment rules ensure that the algorithm terminates, e.g., via the existence of a global objective function that decreases at every iteration (or by incorporating timeouts).

A canonical example of a k-swap local search heuristic is the k-OPT heuristic for the traveling salesman problem (TSP)[14] (see, e.g., [20]). We can view TSP as an object assignment problem, where the objects are edges and $R = \{0, 1\}$; the feasibility constraint is that the edges assigned to 1 should form a tour. Recall that a local move in k-OPT consists of swapping out k edges from the current tour and swapping in k edges to obtain a new tour. (So in our terminology, k-OPT is a $2k$-swap local search heuristic.) Another well-known example is the local search algorithms for the p-median problem studied in Arya et al. [3], which are parameterized by the number of medians that can be removed and added in each local move. Analogous local search algorithms make sense for the MWIS problem as well.

Scoring and assignment rules are now defined on subsets of k objects, rather than individual objects. A single-parameter family of scoring rules is now called *κ-crossing* if, for every subset K of at most k objects and each distinct pair of attribute sets ξ_K and ξ'_K, there are at most κ values of ρ for which $\sigma(\rho, \xi_K) = \sigma(\rho, \xi'_K)$. An assignment rule is now *β-bounded* if for every subset K of at most k objects, ranging over all possible trajectories of the local search heuristic, the attribute set of K takes on at most β distinct values. For example, in MWIS, suppose we allow two vertices u, v to be removed and two vertices y, z to be added in a single local move, and we use the single-parameter scoring rule family

$$\sigma_\rho(u, v, y, z) = \frac{w_u}{(1 + \deg(u))^\rho} + \frac{w_v}{(1 + \deg(v))^\rho} - \frac{w_y}{(1 + \deg(y))^\rho} - \frac{w_z}{(1 + \deg(z))^\rho}.$$

Here $\deg(v)$ could refer to the degree of vertex v in original graph, to the number of neighbors of v that do not have any neighbors other than v in the current independent set, etc. In any case, since a generalized Dirichlet polynomial with t terms has at most $t - 1$ zeroes (see, e.g., [19, Corollary 3.2]), this is a 3-crossing family. The natural assignment rule is n^4-bounded.[15]

By replacing the number n of objects by the number $O(n^k)$ of subsets of at most k objects in the proof of Theorem 3.6, we obtain the following.

THEOREM 3.8 (Pseudo-dimension of local search algorithms). *If $\mathcal{A}$ denotes a (κ, β)-single-parameter family of k-swap local search heuristics for an object assignment problem with n objects, then the pseudo-dimension of $\mathcal{A}$ is $O(k \log(\kappa\beta n))$.*

[14] Given a complete undirected graph with a cost c_{uv} for each edge (u, v), compute a tour (visiting each vertex exactly once) that minimizes the sum of the edge costs.

[15] In general, arbitrary local search algorithms can be made β-bounded through timeouts: if such an algorithm always halts within T iterations, then the corresponding assignment rule is T-bounded.

3.4. Application: Self-improving algorithms revisited. We next give a new interpretation of the self-improving sorting algorithm of Ailon et al. [1]. Namely, we show that the main result in [1] effectively identifies a set of sorting algorithms that simultaneously has low representation error (for independently distributed array elements) and small pseudo-dimension (and hence low generalization error). Other constructions of self-improving algorithms [1, 11, 9, 10] can be likewise reinterpreted. In contrast to subsection 3.3, here our performance measure COST is related to the running time of an algorithm A on an input x, which we want to minimize, rather than the objective function value of the output, which we wanted to maximize.

Consider the problem of sorting n real numbers in the comparison model. By a *bucket-based sorting algorithm*, we mean an algorithm A for which there are "bucket boundaries" $b_1 < b_2 < \cdots < b_\ell$ such that A first distributes the n input elements into their rightful buckets, and then sorts each bucket separately, concatenating the results. The degrees of freedom when defining such an algorithm are (i) the choice of the bucket boundaries; (ii) the method used to distribute input elements to the buckets; and (iii) the method used to sort each bucket. The performance measure COST is the number of comparisons used by the algorithm.[16]

The key steps in the analysis in [1] can be reinterpreted as proving that this set of bucket-based sorting algorithms has low representation error, in the following sense.

THEOREM 3.9 ([1, Theorem 2.1]). *Suppose that each array element a_i is drawn independently from a distribution $\mathcal{D}_i$. Then there exists a bucket-based sorting algorithm with expected running time at most a constant factor times that of the optimal sorting algorithm for $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n$.*

The proof in [1] establishes Theorem 3.9 even when the number ℓ of buckets is only n, each bucket is sorted using InsertionSort, and each element a_i is distributed independently to its rightful bucket using a search tree stored in $O(n^c)$ bits, where $c > 0$ is an arbitrary constant (and the running time depends on $\frac{1}{c}$).[17] Let $\mathcal{A}_c$ denote the set of all such bucket-based sorting algorithms.

Theorem 3.9 reduces the task of learning a near-optimal sorting algorithm to the problem of (ϵ, δ)-learning the optimal algorithm from $\mathcal{A}_c$. Corollary 3.4 reduces this learning problem to bounding the pseudo-dimension of $\mathcal{A}_c$. We next prove such a bound, which effectively says that bucket-based sorting algorithms are "relatively simple" algorithms.[18]

THEOREM 3.10 (Pseudo-dimension of bucket-based sorting algorithms). *The pseudo-dimension of $\mathcal{A}_c$ is $O(n^{1+c})$.*

Proof. Recall from the definitions (subsection 3.2) that we need to upper bound the size of every set that is shatterable using the bucket-based sorting algorithms in $\mathcal{A}_c$. For us, a set is a fixed set of s inputs (i.e., arrays of length n), $S = x_1, \ldots, x_s$. For a potential witness $r_1, \ldots, r_s \in \mathbb{R}$, every algorithm $A \in \mathcal{A}_c$ 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.

[16] Devroye [12] studies similar families of sorting algorithms, with the goal of characterizing the expected running time as a function of the input distribution.

[17] For small c, each search tree T_i is so small that some searches will go unresolved; such unsuccessful searches are handled by a standard binary search over the buckets.

[18] Not all sorting algorithms are simple in the sense of having polynomial pseudo-dimension. For example, the space lower bound in [1, Lemma 2.1] can be adapted to show that no class of sorting algorithms with polynomial pseudo-dimension (or fat shattering dimension) has low representation error in the sense of Theorem 3.9 for general distributions over sorting instances, where the array entries need not be independent.

We proceed to bound from above the number of distinct binary labelings of S induced by the algorithms of $\mathcal{A}_c$, for any potential witness.

By definition, an algorithm from $\mathcal{A}_c$ is fully specified by (i) a choice of n bucket boundaries $b_1 < \cdots < b_n$; and (ii) for each $i = 1, 2, \ldots, n$, a choice of a search tree T_i of size at most $O(n^c)$ for placing x_i in the correct bucket. Call two algorithms $A, A' \in \mathcal{A}_c$ *equivalent* if their sets of bucket boundaries $b_1, \ldots, b_n$ and $b'_1, \ldots, b'_n$ induce the same partition of the sn array elements of the inputs in S; that is, if $x_{ij} < b_k$ if and only if $x_{ij} < b'_k$ (for all i, j, k). The number of equivalence classes of this equivalence relation is at most $\binom{sn+n}{n} \leq (sn+n)^n$. Within an equivalence class, two algorithms that use structurally identical search trees will have identical performance on all s of the samples. Since the search trees of every algorithm of $\mathcal{A}_c$ are described by at most $O(n^{1+c})$ bits, ranging over the algorithms of a single equivalence class generates at most $2^{O(n^{1+c})}$ distinct binary labelings of the s sample inputs. Ranging over all algorithms thus generates at most $(sn+n)^n 2^{O(n^{1+c})}$ labelings. This exceeds 2^s, the requisite number of labellings to shatter S, only if $s = O(n^{1+c})$. ∎

Theorem 3.10 and Corollary 3.4 imply that $m = \tilde{O}(\frac{H^2}{\epsilon^2} n^{1+c})$ samples are enough to (ϵ, δ)-learn the optimal algorithm in $\mathcal{A}_c$.[19] The results implicit in [1] are similarly for relative error. Since the minimum running time is $\Omega(n)$, taking $\epsilon = \Theta(n)$ is enough to ensure that all running times are correctly estimated up to a constant factor. We can also assume that the maximum running time H of any algorithm in $\mathcal{A}_c$ is $O(n \log n)$, since if an algorithm exceeds this bound, we can abort and safely run MergeSort instead. Hence we obtain a sample complexity bound of $\tilde{O}(n^{1+c})$.

Remark 3.11 (Comparison to [1]). The sample complexity bound implicit in [1] for learning a near-optimal sorting algorithm is $\tilde{O}(n^c)$, a linear factor better than the $\tilde{O}(n^{1+c})$ bound implied by Theorem 3.10. There is good reason for this: the pseudo-dimension bound of Theorem 3.10 implies that an even harder problem has sample complexity $\tilde{O}(n^{1+c})$, namely that of learning a near-optimal bucket-based sorting algorithm with respect to an *arbitrary* distribution over inputs, *even with correlated array elements*.[20] The bound of $\tilde{O}(n^c)$ in [1] applies only to the problem of learning a near-optimal bucket-based sorting algorithm for an unknown input distribution with independent array entries; the savings comes from the fact that all n near-optimal search trees $T_1, \ldots, T_n$ can be learned in parallel.

3.5. Application: Feature-based algorithm selection. Previous sections studied the problem of selecting a single algorithm for use in an application domain, i.e., using training data to make an informed commitment to a single algorithm from a class $\mathcal{A}$, which is then used on all future instances. A more refined and ambitious approach is to select an algorithm based both on previous experience *and on the current instance to be solved*. This approach assumes, as in the scenario in subsection 2.4, 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 subsection 3.1 with

5. A set $\mathcal{F}$ of possible instance feature values, and a map $f : X \to \mathcal{F}$ that computes the features of a given instance.[21]

[19] We again use $\tilde{O}(\cdot)$ to suppress logarithmic factors.

[20] When array elements are not independent, however, Theorem 3.9 fails and the best bucket-based sorting algorithm might be more than a constant factor worse than the optimal sorting algorithm.

[21] 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 $\mathcal{F}$ and map f as given.

For instance, 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 [21], and so on.

When the set $\mathcal{F}$ of possible instance feature values is finite, the guarantees for the basic model extend easily with a linear (in $|\mathcal{F}|$) degradation in the pseudo-dimension.[22] To explain, we add an additional ingredient to the model.

6. A set $\mathcal{G}$ of *algorithm selection maps*, with each $g \in \mathcal{G}$ a function from $\mathcal{F}$ to $\mathcal{A}$.

An algorithm selection map recommends an algorithm as a function of the features of an instance.

We can view an algorithm selection map g as a real-valued function defined on the instance space X, with $g(x)$ defined as $\text{COST}(g(f(x)), x)$. That is, $g(x)$ is the running time on x of the algorithm $g(f(x))$ advocated by g, given that x has features $f(x)$. The basic model studied earlier is the special case where $\mathcal{G}$ is the set of constant functions, which are in correspondence with the algorithms of $\mathcal{A}$.

Corollary 3.4 reduces bounding the sample complexity of (ϵ, δ)-learning the best algorithm selection map of $\mathcal{G}$ to bounding the pseudo-dimension of the set of real-valued functions induced by $\mathcal{G}$. When $\mathcal{G}$ is finite, there is a trivial upper bound of $\log_2 |\mathcal{G}|$. The pseudo-dimension is also small whenever $\mathcal{F}$ is small and the set $\mathcal{A}$ of algorithms has small pseudo-dimension.[23]

PROPOSITION 3.12 (Pseudo-dimension of algorithm selection maps). *If $\mathcal{G}$ is a set of algorithm selection maps from a finite set $\mathcal{F}$ to a set $\mathcal{A}$ of algorithms with pseudo-dimension d, then $\mathcal{G}$ has pseudo-dimension at most $|\mathcal{F}|d$.*

Proof. A set of inputs of size $|\mathcal{F}|d + 1$ is shattered only if there is a shattered set of inputs with identical features of size $d + 1$. ∎

Now suppose $\mathcal{F}$ is very large (or infinite). We focus on the case where $\mathcal{A}$ is small enough that it is feasible to learn a separate performance prediction model for each algorithm $A \in \mathcal{A}$ (though see Remark 3.15). This is exactly the approach taken in the motivating example of empirical performance models (EPMs) for SAT described in subsection 2.4. In this case, we augment the basic model to include a family of performance predictors.

7. A set $\mathcal{P}$ of *performance predictors*, with each $p \in \mathcal{P}$ a function from $\mathcal{F}$ to $\mathbb{R}$.

Performance predictors play the same role as the EPMs used in [33].

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

$$\mathbf{E}_{x \sim \mathcal{D}}\left[(\text{COST}(A, x) - p(f(x)))^2\right]$$

over $p \in \mathcal{P}$.[24] For a fixed algorithm A, this is a standard regression problem, with domain $\mathcal{F}$, real-valued labels, and a distribution on $\mathcal{F} \times \mathbb{R}$ induced by $\mathcal{D}$ via $x \mapsto$

[22] For example, [33] first predicts whether or not a given SAT instance is satisfiable or not, and then uses a "conditional" empirical performance model to choose a SAT solver. This can be viewed as an example with $|\mathcal{F}| = 2$, corresponding to the feature values "looks satisfiable" and "looks unsatisfiable."

[23] When $\mathcal{G}$ is the set of all maps from $\mathcal{F}$ to $\mathcal{A}$ and every feature value of $\mathcal{F}$ appears with approximately the same probability, one can alternatively just separately learn the best algorithm for each feature value.

[24] Note that the expected loss incurred by the best predictor depends on the choices of the predictor set $\mathcal{P}$, the feature set $\mathcal{F}$, and map f. Again, these choices are outside our model.

$(f(x), \text{COST}(A, x))$. Bounding the sample complexity of this learning problem reduces to bounding the pseudo-dimension of $\mathcal{P}$. For standard choices of $\mathcal{P}$, such bounds are well known. For example, suppose the set $\mathcal{P}$ is the class of *linear predictors*, with each $p \in \mathcal{P}$ having the form $p(f(x)) = a^T f(x)$ for some coefficient vector $a \in \mathbb{R}^d$.[25]

PROPOSITION 3.13 (Pseudo-dimension of linear predictors). *If $\mathcal{F}$ contains real-valued d-dimensional features and $\mathcal{P}$ is the set of linear predictors, then the pseudo-dimension of $\mathcal{P}$ is at most d.*

If all functions in $\mathcal{P}$ map all possible φ to $[0, H]$, then Proposition 3.13 and Corollary 3.4 imply a sample complexity bound of $\tilde{O}(\frac{H^4}{\epsilon^2}d)$ for (ϵ, δ)-learning the predictor with minimum expected square error. Similar results hold, with worse dependence on d, if $\mathcal{P}$ is a set of low-degree polynomials [2].

For another example, suppose $\mathcal{P}_\ell$ is the set of regression trees with at most ℓ nodes, where each internal node performs an inequality test on a coordinate of the feature vector φ (and leaves are labeled with performance estimates).[26] This class also has low pseudo-dimension, and hence the problem of learning a near-optimal predictor has correspondingly small sample complexity.

PROPOSITION 3.14 (Pseudo-dimension of regression trees). *Suppose $\mathcal{F}$ contains real-valued d-dimensional features and let $\mathcal{P}_\ell$ be the set of regression trees with at most ℓ nodes, where each node performs an inequality test on one of the features. Then, the pseudo-dimension of $\mathcal{P}_\ell$ is $O(\ell \log(\ell d))$.*

Remark 3.15 (Extension to large $\mathcal{A}$). We can also extend our approach to scenarios with a large or infinite set $\mathcal{A}$ of possible algorithms. This extension is relevant to state-of-the-art empirical approaches to the auto-tuning of algorithms with many parameters, such as mathematical programming solvers [18]; see also the discussion in subsection 2.3. (Instantiating all of the parameters yields a fixed algorithm; ranging over all possible parameter values yields the set $\mathcal{A}$.) Analogous to our formalism for accommodating a large number of possible features, we now assume that there is a set $\mathcal{F}'$ of possible "algorithm feature values" and a mapping f' that computes the features of a given algorithm. A performance predictor is now a map from $\mathcal{F} \times \mathcal{F}'$ to $\mathbb{R}$, taking as input the features of an algorithm A and of an instance x, and returning as output an estimate of A's performance on x. If $\mathcal{P}$ is the set of linear predictors, for example, then by Proposition 3.13 its pseudo-dimension is $d + d'$, where d and d' denote the dimensions of $\mathcal{F}$ and $\mathcal{F}'$, respectively.

3.6. Application: Choosing the step size in gradient descent. For our last PAC 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 step size from examples. We view this as a baby step towards reasoning more generally about the problem of learning good parameters for machine learning algorithms. Unlike the applications we've seen

[25] A linear model might sound unreasonably simple for the task of predicting the running time of an algorithm, but significant complexity can be included in the feature map $f(x)$. For example, each coordinate of $f(x)$ could be a nonlinear combination of several "basic features" of x. Indeed, linear models often exhibit surprisingly good empirical performance, given a judicious choice of a feature set [24].

[26] Regression trees, and random forests thereof, have emerged as a popular class of predictors in empirical work on application-specific algorithm selection [18].

so far, the parameter space here satisfies a Lipschitz-like condition, and we can follow the discretization approach suggested in Remark 3.7.

3.6.1. 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 $\|\nabla f(z)\|_2 > \nu$
   (a) $z := z - \rho \cdot \nabla f(z)$.

We take the error tolerance ν 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 (subsection 3.1) to study the problem of learning the best step size. There is an unknown distribution $\mathcal{D}$ over instances, where an instance $x \in \Pi$ consists of a function f and an initial point z_0. Each algorithm A_ρ of $\mathcal{A}$ is the basic gradient descent algorithm above, with some choice ρ of a step size drawn from some fixed interval $[\rho_\ell, \rho_u] \subset (0, \infty)$. The performance measure $\textsc{cost}(A, x)$ is the number of iterations (i.e., steps) taken by the algorithm for the instance x.

To obtain positive results, we need to restrict the allowable functions f (see Appendix A). First, we assume that every function f is convex and L-smooth for a known L. A function f is *L-smooth* if it is everywhere differentiable, and $\|\nabla f(z_1) - \nabla f(z_2)\| \leq L\|z_1 - z_2\|$ for all z_1 and z_2 (all norms in this section are the ℓ_2 norm). Since gradient descent is translation invariant, and f is convex, we can assume for convenience that the (uniquely attained) minimum value of f is 0, with $f(0) = 0$.

Second, we assume that the magnitudes of the initial points are bounded, with $\|z_0\| \leq Z$ for some known constant $Z > \nu$.

Third, we assume that there is a known constant $c \in (0, 1)$ such that $\|z - \rho\,\nabla f(z)\| \leq (1 - c)\|z\|$ for all $\rho \in [\rho_\ell, \rho_u]$. In other words, the norm of any point z—equivalently, the distance to the global minimum—decreases by some minimum factor after each gradient descent step. We refer to this as the *guaranteed progress* condition. This is satisfied (for instance) by L-smooth, m-strongly convex functions,[27] which is a well studied regime (see, e.g., [6]). The standard analysis of gradient descent implies that $c \geq \rho m$ for $\rho \leq 2/(m + L)$ over this class of functions.

Under these restrictions, we will be able to compute a nearly optimal ρ given a reasonable number of samples from $\mathcal{D}$.

Other notation. All norms in this section are ℓ_2-norms. Unless otherwise stated, ρ means ρ restricted to $[\rho_\ell, \rho_u]$, and z means z such that $\|z\| \leq Z$. We let $g(z, \rho) := z - \rho\nabla f(z)$ be the result of taking a single gradient descent step, and $g^j(z, \rho)$ be the result of taking j gradient descent steps.

Typical textbook treatments of gradient descent assume $\rho < 2/L$ or $\rho \leq 2/(m + L)$, which give various convergence and running time guarantees. The learning results of this section apply for any ρ, but this natural threshold will still appear in our analysis and results. Let $D(\rho) := \max\{1, L\rho - 1\}$ denote how far ρ is from $2/L$.

By the guaranteed progress condition, $\|g^j(z, \rho)\| \leq (1-c)^j\|z\|$, and so by L-smoothness,

$$\left\|\nabla f\left(g^j(z, \rho)\right)\right\| \leq (1-c)^j L\|z\|.$$

[27] A (continuously differentiable) function f is *m-strongly convex* if $f(y) \geq f(w) + \nabla f(w)^T(y - w) + \frac{m}{2}\|y - w\|^2$ for all $w, y \in \mathbb{R}^n$. The usual notion of convexity is the same as 0-strong convexity. Note that the definition of L-smooth implies $m \leq L$.

Since $\|z_0\| \leq Z$, and we stop once the gradient is $\leq \nu$, $\textsc{cost}(A_\rho, x) \leq \log(\nu/LZ)/\log(1-c)$ for all ρ and x. Let $H = \log(\nu/LZ)/\log(1-c)$.

3.6.2. A Lipschitz-like bound on $\textsc{cost}(A_\rho, x)$ as a function of ρ. This will be the bulk of the argument. Our first lemma shows that for fixed ρ, the gradient descent step g is a Lipschitz function of z, even when ρ is larger than $2/L$. One might hope that the guaranteed progress condition would be enough to show that (say) g is a contraction, but the Lipschitzness of g actually comes from the L-smoothness. (It is not too hard to come up with nonsmooth functions that make guaranteed progress, and where g is arbitrarily non-Lipschitz.)

LEMMA 3.16. $\|g(w,\rho) - g(y,\rho)\| \leq D(\rho)\|w - y\|$.

Proof. For notational simplicity, let $\alpha = \|w - y\|$ and $\beta = \|\nabla f(w) - \nabla f(y)\|$. Now,

$$\begin{aligned}\|g(w,\rho) - g(y,\rho)\|^2 &= \|(w-y) - \rho(\nabla f(w) - \nabla f(y))\|^2 \\ &= \alpha^2 + \rho^2\beta^2 - 2\rho\langle\alpha,\beta\rangle \\ &\leq \alpha^2 + \rho^2\beta^2 - 2\rho\beta^2/L \\ &= \alpha^2 + \beta^2\rho(\rho - 2/L).\end{aligned}$$

The only inequality above is a restatement of a property of L-smooth functions called the co-coercivity of the gradient, namely that $\langle\alpha,\beta\rangle \geq \beta^2/L$.

Now, if $\rho \leq 2/L$, then $\rho(\rho - 2/L) \leq 0$, and we're done. Otherwise, L-smoothness implies $\beta \leq L\alpha$, so the above is at most $\alpha^2(1 + L\rho(L\rho - 2))$, which is the desired result. ∎

The next lemma bounds how far two gradient descent paths can drift from each other, if they start at the same point. The main thing to note is that the right-hand side goes to 0 as η becomes close to ρ.

LEMMA 3.17. *For any z, j, and $\rho \leq \eta$,*

$$\left\|g^j(z,\rho) - g^j(z,\eta)\right\| \leq (\eta - \rho)\frac{D(\rho)^j LZ}{c}.$$

Proof. We first bound $\|g(w,\rho) - g(y,\eta)\|$, for any w and y. We have

$$\begin{aligned}g(w,\rho) - g(y,\eta) &= [w - \rho\nabla f(w)] - [y - \eta\nabla f(y)] \\ &= g(w,\rho) - [g(y,\rho) - (\eta - \rho)\nabla f(y)]\end{aligned}$$

by definition of g. The triangle inequality and Lemma 3.16 then give

$$\begin{aligned}\|g(w,\rho) - g(y,\eta)\| &= \|g(w,\rho) - g(y,\rho) + (\eta - \rho)\nabla f(y)\| \\ &\leq D(\rho)\,\|w - y\| + (\eta - \rho)\|\nabla f(y)\|.\end{aligned}$$

Plugging in $w = g^j(z,\rho)$ and $y = g^j(z,\eta)$, we have

$$\left\|g^{j+1}(z,\rho) - g^{j+1}(z,\eta)\right\| \leq D(\rho)\,\left\|g^j(z,\rho) - g^j(z,\eta)\right\| + (\eta - \rho)\left\|\nabla f(g^j(z,\eta))\right\|$$

for all j.

Now,

$$\left\|\nabla f(g^j(z,\eta))\right\| \leq L\left\|g^j(z,\eta)\right\| \leq L\|z\|(1-c)^j \leq LZ(1-c)^j,$$

where the first inequality is from L-smoothness, and the second is from the guaranteed progress condition. Letting $r_j = \|g^j(z,\rho) - g^j(z,\eta)\|$, we now have the simple recurrence $r_0 = 0$, and $r_{j+1} \le D(\rho)\, r_j + (\eta - \rho)LZ(1-c)^j$. One can check via induction that

$$r_{j+1} \le D(\rho)^j(\eta-\rho)LZ\sum_{i=0}^{j}(1-c)^i D(\rho)^{-i}$$

for all j. Recall that $D(\rho) \ge 1$. Rounding $D(\rho)^{-i}$ up to 1 and doing the summation gives the desired result. ∎

Finally, we show that $\textsc{cost}(A_\rho, x)$ is essentially Lipschitz in ρ. The "essentially" is necessary, since COST is integer-valued.

LEMMA 3.18. $|\textsc{cost}(A_\rho,x) - \textsc{cost}(A_\eta,x)| \le 1$ *for all* x, ρ, *and* η *with* $0 \le \eta - \rho \le \frac{\nu c^2}{LZ} D(\rho)^{-H}$.

Proof. Assume that $\textsc{cost}(A_\eta,x) \le \textsc{cost}(A_\rho,x)$; the argument in the other case is similar. Let $j = \textsc{cost}(A_\eta,x)$, and recall that $j \le H$. By Lemma 3.17, $\|g^j(x,\rho) - g^j(x,\eta)\| \le \nu c$. Hence, by the triangle inequality,

$$\left\|g^j(x,\rho)\right\| \le \nu c + \left\|g^j(x,\eta)\right\| \le \nu c + \nu.$$

Now, by the guaranteed progress condition, $\|w\| - \|g(w,p)\| \ge c\|w\|$ for all w. Since we only run a gradient descent step on w if $\|w\| > \nu$, each step of gradient descent run by any algorithm in $\mathcal{A}$ drops the magnitude of w by at least νc.

Setting $w = g^j(x,\rho)$, we see that either $\|g^j(x,\rho)\| \le \nu$, and $\textsc{cost}(A_\rho,x) = j$, or that $\|g^{j+1}(x,\rho)\| \le (\nu c + \nu) - \nu c = \nu$, and $\textsc{cost}(A_\rho,x) = j+1$, as desired. ∎

3.6.3. Learning the best step size. We can now apply the discretization approach suggested by Remark 3.7. Let $K = \frac{\nu c^2}{LZ} D(\rho_u)^{-H}$. Note that since D is an increasing function, K is less than or equal to the $\frac{\nu c^2}{LZ} D(\rho)^{-H}$ of Lemma 3.18 for every ρ. Let N be a minimal K-net, such as all integer multiples of K that lie in $[\rho_\ell, \rho_u]$. Note that $|N| \le \rho_u/K + 1$.

We tie everything together in the theorem below.[28]

THEOREM 3.19 (Learnability of step size in gradient descent). *There is a learning algorithm that* $(1+\epsilon,\delta)$*-learns the optimal algorithm in* $\mathcal{A}$ *using* $m = \tilde{O}(H^3/\epsilon^2)$ *samples from* $\mathcal{D}$.[29]

Proof. The pseudo-dimension of $\mathcal{A}_N = \{A_\rho : \rho \in N\}$ is at most $\log|N|$, since $\mathcal{A}_N$ is a finite set. Since $\mathcal{A}_N$ is finite, it also trivially admits an ERM algorithm L_N, and Corollary 3.4 implies that L_N (ϵ,δ)-learns the optimal algorithm in $\mathcal{A}_N$ using $m = \tilde{O}(H^2 \log|N|/\epsilon^2)$ samples.

Now, Lemma 3.18 implies that for every ρ, there is a $\eta \in N$ such that, for every distribution $\mathcal{D}$, the difference in expected costs of A_η and A_ρ is at most 1. Thus L_N $(1+\epsilon,\delta)$-learns the optimal algorithm in $\mathcal{A}$ using $m = \tilde{O}(H^2\epsilon^{-2}\log|N|)$ samples.

Since $\log|N| = \tilde{O}(H)$, we get the desired result. ∎

[28] Alternatively, this guarantee can be phrased in terms of the fat-shattering dimension (see, e.g., [2]). In particular, $\mathcal{A}$ has 1.001 fat-shattering dimension at most $\log|N| = \tilde{O}(H)$.

[29] We use $\tilde{O}(\cdot)$ to suppress logarithmic factors in $Z/\nu, c, L$, and ρ_u.

4. Online learning of application-specific algorithms. This section studies the problem of learning the best application-specific algorithm *online*, with instances arriving one by one.[30] The goal is choose an algorithm at each time step, before seeing the next instance, so that the average performance is close to that of the best fixed algorithm in hindsight. This contrasts with the statistical (or "batch") learning setup used in section 3, where the goal was to identify a single algorithm from a batch of training instances that generalizes well to future instances from the same distribution. For many of the motivating examples in section 2, both the statistical and online learning approaches are relevant. The distribution-free online learning formalism of this section may be particularly appropriate when instances cannot be modeled as i.i.d. draws from an unknown distribution.

4.1. The online learning model. Our online learning model shares with the basic model of subsection 3.1 a computational or optimization problem Π (e.g., MWIS), a set $\mathcal{A}$ of algorithms for Π (e.g., a single-parameter family of greedy heuristics), and a performance measure $\text{COST} : \mathcal{A} \times \Pi \to [0,1]$ (e.g., the total weight of the returned solution).[31] Rather than modeling the specifics of an application domain via an unknown distribution $\mathcal{D}$ over instances, however, we use an unknown instance *sequence* $x_1, \ldots, x_T$.[32]

A learning algorithm now outputs a sequence $A_1, \ldots, A_T$ of algorithms, rather than a single algorithm. Each algorithm A_i is chosen (perhaps probabilistically) with knowledge only of the previous instances $x_1, \ldots, x_{i-1}$. The standard goal in online learning is to choose $A_1, \ldots, A_T$ to minimize the worst-case (over $x_1, \ldots, x_T$) *regret*, defined as the average performance loss relative to the best algorithm $A \in \mathcal{A}$ in hindsight:[33]

$$\frac{1}{T}\left(\sup_{A \in \mathcal{A}} \sum_{t=1}^{T} \text{COST}(A, x_i) - \sum_{t=1}^{T} \text{COST}(A_i, x_i)\right). \tag{2}$$

A *no-regret* learning algorithm has expected (over its coin tosses) regret $o(1)$, as $T \to \infty$, for every instance sequence. The design and analysis of no-regret online learning algorithms is a mature field (see, e.g., [8]). For example, many no-regret online learning algorithms are known for the case of a finite set $|\mathcal{A}|$ (such as the "multiplicative weights" algorithm).

4.2. An impossibility result for worst-case instances. This section proves an impossibility result for no-regret online learning algorithms for the problem of application-specific algorithm selection. We show this for the running example in subsection 3.3: maximum-weight independent set (MWIS) heuristics[34] that, for some parameter $\rho \in [0,1]$, process the vertices in order of nonincreasing value of $w_v/(1 + \deg(v))^\rho$. Let $\mathcal{A}$ denote the set of all such MWIS algorithms. Since $\mathcal{A}$ is an infinite set, standard no-regret results (for a finite number of actions) do not immediately

[30] The online model is obviously relevant when training data arrives over time. Also, even with offline data sets that are very large, it can be computationally necessary to process training data in a one-pass, online fashion.

[31] One could also have COST take values in $[0, H]$ rather than $[0,1]$, to parallel the PAC setting; we set $H = 1$ here since the dependence on H will not be interesting.

[32] For simplicity, we assume that the time horizon T is known. This assumption can be removed by standard doubling techniques (e.g., [8]).

[33] Without loss of generality, we assume COST corresponds to a maximization objective.

[34] Subsection 3.3 defined adaptive and nonadaptive versions of the MWIS heuristic. All of the results in section 4 apply to both, so we usually won't distinguish between them.

apply. In online learning, infinite sets of options are normally controlled through a Lipschitz condition, stating that "nearby" actions always yield approximately the same performance; our set $\mathcal{A}$ does not possess such a Lipschitz property (recall Remark 3.7). The next section shows that these issues are not mere technicalities: there is enough complexity in the set $\mathcal{A}$ of MWIS heuristics to preclude a no-regret learning algorithm.

4.2.1. A hard example for MWIS. We show a distribution over sequences of MWIS instances for which every (possibly randomized) algorithm has expected regret $1 - o_n(1)$. Here and for the rest of this section, by $o_n(1)$ we mean a function that is independent of T and tends to 0 as the number of vertices n tends to infinity. Recall that $\text{COST}(A_\rho, x)$ is the total weight of the returned independent set, and we are trying to maximize this quantity. The key construction is the following:

LEMMA 4.1. *For any constants $0 < r < s < 1$, there exists a MWIS instance x on at most n vertices such that $\text{COST}(A_\rho, x) = 1$ when $\rho \in (r, s)$, and $\text{COST}(A_\rho, x) = o_n(1)$ when $\rho < r$ or $\rho > s$.*

Proof. Let A, B, and C be three sets of vertices of sizes $m^2 - 2, m^3 - 1$, and $m^2 + m + 1$ respectively, such that their sum $m^3 + 2m^2 + m$ is between $n/2$ and n. Let (A, B) be a complete bipartite graph. Let (B, C) also be a bipartite graph, with each vertex of B connected to exactly one vertex of C, and each vertex of C connected to exactly $m - 1$ vertices of B. See Figure 1.

Now, set the weight of every vertex in A, B, and C to tm^r, t, and tm^{-s}, respectively, for $t = (m^3 - 1)^{-1}$. Table 1 summarizes some straightforward calculations. We now calculate the cost of A_ρ on this instance.

If $\rho < r$, the algorithm A_ρ first chooses a vertex in A, which immediately removes all of B, leaving at most A and C in the independent set. The total weight of A and C is $o_n(1)$, so $\text{COST}(A_\rho)$ is $o_n(1)$.

If $\rho > s$, the algorithm first chooses a vertex in C, which removes a small chunk of B. In the nonadaptive setting, A_ρ simply continues choosing vertices of C until B is gone. In the adaptive setting, the degrees of the remaining elements of B never

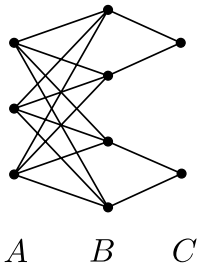


FIG. 1. *A rough depiction of the MWIS example from Lemma* 4.1.

TABLE 1
Details and simple calculations for the vertex sets comprising the MWIS example from Lemma 4.1.

	size	weight	deg	weight/(deg+1)$^\rho$	size × weight
A	$m^2 - 2$	tm^r	$m^3 - 1$	$tm^{r-3\rho}$	$o_n(1)$
B	$m^3 - 1$	t	$m^2 - 1$	$tm^{-2\rho}$	1
C	$m^2 + m + 1$	tm^{-s}	$m - 1$	$tm^{-s-\rho}$	$o_n(1)$

change, but the degrees of A decrease as we pick more and more elements of C. We eventually pick a vertex of A, which immediately removes the rest of B. In either case, the returned independent set has no elements from B, and hence has COST $o_n(1)$.

If $\rho \in (r, s)$, the algorithm first picks a vertex of B, immediately removing all of A, and one element of C. The remaining graph comprises $m - 2$ isolated vertices of B (which get added to the independent set), and $m^2 + m$ stars with centers in C and leaves in B. It is easy to see that both the adaptive and the nonadaptive versions of the heuristic return exactly B. ∎

We are now ready to state the main result of this section.

THEOREM 4.2 (Impossibility of worst-case online learning). *There is a distribution on MWIS input sequences over which every algorithm has expected regret* $1-o_n(1)$.

Proof. Let $t_j = (r_j, s_j)$ be a distribution over sequences of nested intervals with $s_j - r_j = n^{-j}$, $t_0 = (0, 1)$, and with t_j chosen uniformly at random from within t_{j-1}. Let x_j be an MWIS instance on up to n vertices such that $\text{COST}(A_\rho, x) = 1$ for $\rho \in (r_j, s_j)$, and $\text{COST}(A_\rho, x) = o_n(1)$ for $\rho < r_j$ and $\rho > s_j$ (Lemma 4.1).

The adversary presents the instances $x_1, x_2, \ldots, x_T$, in that order. For every $\rho \in t_T$, $\text{COST}(A_\rho, x_j) = 1$ for all j. However, at every step t, no algorithm can have a better than $1/n$ chance of picking a ρ_t for which $\text{COST}(A_{\rho_t}, x_t) = \Theta_n(1)$, even given $x_1, x_2, \ldots, x_{t-1}$ and full knowledge of how the sequence is generated. ∎

4.3. A smoothed analysis. Despite the negative result above, we can show a "low-regret" learning algorithm for MWIS under a slight restriction on how the instances x_t are chosen. By low-regret we mean that the regret can be made polynomially small as a function of the number of vertices n. This is not the same as the no-regret condition, which requires regret tending to 0 as $T \to \infty$. Nevertheless, inverse polynomially small regret $\text{poly}(n^{-1})$ is a huge improvement over the constant regret incurred in the worst-case lower bound (Theorem 4.2).

We take the approach suggested by smoothed analysis [31]. Fix a parameter $\sigma \in (0, 1)$. We allow each MWIS instance x_t to have an arbitrary graph on n vertices, but we replace each vertex weight w_v with a probability distribution $\Delta_{t,v}$ with density at most σ^{-1} (pointwise) and support in $[0, 1]$. A simple example of such a distribution with $\sigma = 0.1$ is the uniform distribution on $[0.6, 0.65] \cup [0.82, 0.87]$. To instantiate the instance x_t, we draw each vertex weight from its distribution $\Delta_{t,v}$. We call such an instance a *σ-smooth MWIS instance*.

For small σ, this is quite a weak restriction. As $\sigma \to 0$ we return to the worst-case setting, and Theorem 4.2 can be extended to the case of σ exponentially small in n. Here, we think of σ as bounded below by an (arbitrarily small) inverse polynomial function of n. One example of such a smoothing is to start with an arbitrary MWIS instance, keep the first $O(\log n)$ bits of every weight, and set the remaining lower-order bits at random.

The main result of this section is a polynomial-time low-regret learning algorithm for sequences of σ-smooth MWIS instances. Our strategy is to take a finite net $N \subset [0, 1]$ such that, for every algorithm A_ρ and smoothed instance x_t, with high probability over x_t the performance of A_ρ is identical to that of some algorithm in $\{A_\eta : \eta \in N\}$. We can then use any off-the-shelf no-regret algorithm to output a sequence of algorithms from the finite set $\{A_\eta : \eta \in N\}$, and show the desired regret bound.

4.3.1. A low-regret algorithm for σ-smooth MWIS. We start with some definitions. For a fixed x, let $\tau'(x)$ be the set of *transition points*, namely,[35]

$$\tau'(x) := \{\rho : A_{\rho-\omega}(x) \neq A_{\rho+\omega}(x) \text{ for arbitrarily small } \omega\}.$$

It is easy to see $\tau'(x) \subset \tau(x)$, where

$$\tau(x) := \{\rho : w_{v_1}/k_1^\rho = w_{v_2}/k_2^\rho \text{ for some } v_1, v_2, k_1, k_2 \in [n];\ k_1, k_2 \geq 2\}.$$

With probability 1, the vertex weights w_v are all distinct and non-zero, so we can rewrite τ as

$$\tau(x) := \{\rho(v_1, v_2, k_1, k_2) \ : \ v_1, v_2, k_1, k_2 \in [n];\ k_1, k_2 \geq 2;\ k_1 \neq k_2\},$$

where

$$\rho(v_1, v_2, k_1, k_2) = \frac{\ln(w_{v_1}) - \ln(w_{v_2})}{\ln(k_1) - \ln(k_2)} \tag{3}$$

and ln is the natural logarithm function. The main technical task is to show that no two elements of $\tau(x_1) \cup \cdots \cup \tau(x_m)$ are within q of each other, for a sufficiently large q and sufficiently large m, and with high enough probability over the randomness in the weights of the x_t's.

We first make a few straightforward computations. The following brings the noise into log space.

LEMMA 4.3. *If X is a random variable over $(0, 1]$ with density at most δ, then $\ln(X)$ also has density at most δ.*

Proof. Let $Y = \ln(X)$, let $f(x)$ be the density of X at x, and let $g(y)$ be the density of Y at y. Note that $X = e^Y$, and let $v(y) = e^y$. Then $g(y) = f(v(y)) \cdot v'(y) \leq f(v(y)) \leq \delta$ for all y. ∎

Since $|\ln(k_1) - \ln(k_2)| \leq \ln n$, Lemma 4.3 and our definition of σ-smoothness implies the following.

COROLLARY 4.4. *For every σ-smooth MWIS instance x, and every $v_1, v_2, k_1, k_2 \in [n]$, $k_1, k_2 \geq 2$, $k_1 \neq k_2$, the density of $\rho(v_1, v_2, k_1, k_2)$ is bounded by $\sigma^{-1} \ln n$.*

We now show that it is unlikely that two distinct elements of $\tau(x_1) \cup \cdots \cup \tau(x_m)$ are very close to each other.

LEMMA 4.5. *Let $x_1, \ldots, x_m$ be σ-smooth MWIS instances. The probability that no two distinct elements of $\tau(x_1) \cup \cdots \cup \tau(x_m)$ are within q of each other is at least $1 - 4q\sigma^{-1}m^2n^8 \ln n$.*

Proof. Fix instances x and x', and choices of (v_1, v_2, k_1, k_2) and (v_1', v_2', k_1', k_2'). Denote by ρ and ρ' the corresponding random variables, defined as in (3). We compute the probability that $|\rho - \rho'| \leq q$ under various scenarios, over the randomness in the vertex weights. We can ignore the case where $x = x'$, $v_1 = v_1'$, $v_2 = v_2'$, and $k_1/k_2 = k_1'/k_2'$, since then $\rho = \rho'$ with probability 1. We consider three other cases.

Case 1. Suppose $x \neq x'$, and/or $\{v_1, v_2\}$ and $\{v_1', v_2'\}$ don't intersect. In this case, ρ and ρ' are independent random variables. Hence the maximum density of $\rho - \rho'$ is at

[35] The corner cases $\rho = 0$ and $\rho = 1$ require straightforward but wordy special handling in this statement and in several others in this section. We omit these details to keep the argument free of clutter.

most the maximum density of ρ, which is $\sigma^{-1} \ln n$ by Corollary 4.4. The probability that $|\rho - \rho'| \leq q$ is hence at most $2q \cdot \sigma^{-1} \ln n$.

Case 2. Suppose $x = x'$, and $\{v_1, v_2\}$ and $\{v_1', v_2'\}$ share exactly one element, say $v_2 = v_2'$. Then $\rho - \rho'$ has the form $X - Y$, where $X = \frac{\ln(w_{v_1})}{\ln(k_1) - \ln(k_2)}$ and X and Y are independent. Since the maximum density of X is at most $\sigma^{-1} \ln n$ (by Lemma 4.3), the probability that $|\rho - \rho'| \leq q$ is again at most $2q \cdot \sigma^{-1} \ln n$.

Case 3. Suppose $x = x'$ and $\{v_1, v_2\} = \{v_1', v_2'\}$. In this case, $k_1/k_2 \neq k_1'/k_2'$. Then

$$\begin{aligned} |\rho - \rho'| &= \left| \big(\ln(w_{v_1}) - \ln(w_{v_2})\big) \left(\frac{1}{\ln(k_1) - \ln(k_2)} - \frac{1}{\ln(k_1') - \ln(k_2')} \right) \right| \\ &\geq \frac{|\ln(w_{v_1}) - \ln(w_{v_2})|}{n^2}. \end{aligned}$$

Since w_{v_1} and w_{v_2} are independent, the maximum density of the right-hand side is at most $\sigma^{-1} n^2$, and hence the probability that $|\rho - \rho'| \leq q$ is at most $2q \cdot \sigma^{-1} n^2$.

We now upper bound the number of tuple pairs that can appear in each case above. Each set $\tau(x_i)$ has at most n^4 elements, so there are at most $m^2 n^8$ pairs in Cases 1 and 2. There are at most n^4 choices of (k_1, k_2, k_1', k_2') for each (x, v_1, v_2) in Case 3, for a total of at most mn^6 pairs. The theorem now follows from the union bound. ∎

Lastly, we formally state the existence of no-regret algorithms for the case of finite $|\mathcal{A}|$.

FACT 4.6 (E.g. [25]). *For a finite set of algorithms $\mathcal{A}$, there exists a randomized online learning algorithm L^* that, for every $m > 0$, has expected regret at most $O(\sqrt{(\log |\mathcal{A}|)/m})$ after seeing m instances. If the time cost of evaluating $\textsc{cost}(A, x)$ is bounded by B, then this algorithm runs in $O(B|\mathcal{A}|)$ time per instance.*

We can now state our main theorem.

THEOREM 4.7 (Online learning of smooth MWIS). *There is an online learning algorithm for σ-smooth MWIS that runs in time* $\text{poly}(n, \sigma^{-1})$ *and has expected regret at most* $\text{poly}(n^{-1})$ *(as $T \to \infty$).*

Proof. Fix a sufficiently large constant $d > 0$ and consider the first m instances of our sequence, $x_1, \ldots, x_m$, with $m = n^d \ln(\sigma^{-1})$. Let $q = 1/(n^d \cdot 4\sigma^{-1} m^2 n^8 \ln n)$. Let E_q be the event that every two distinct elements of $\tau(x_1) \cup \cdots \cup \tau(x_m)$ are at least q away from each other. By Lemma 4.5, E_q holds with probability at least $1 - 1/n^d$ over the randomness in the vertex weights.

Now, let $\mathcal{A}_N = \{A_i : i \in \{0, q, 2q, \ldots, \lfloor 1/q \rfloor q, 1\}\}$ be a "q-net." Our desired algorithm L is simply the algorithm L^* from Fact 4.6, applied to $\mathcal{A}_N$. We now analyze its expected regret.

If E_q does hold, then for every algorithm $A \in \mathcal{A}$, there is an algorithm $A' \in \mathcal{A}_N$ such that $\textsc{cost}(A, x_t) = \textsc{cost}(A', x_t)$ for $x_1, \ldots, x_m$. In other words, the best algorithm of $\mathcal{A}_N$ is no worse than the best algorithm from all of $\mathcal{A}$, and in this case the expected regret of L is simply that of L^*. By Fact 4.6 and our choice of m, the expected regret (over the coin flips made by L^*) is at most inverse polynomial in n.

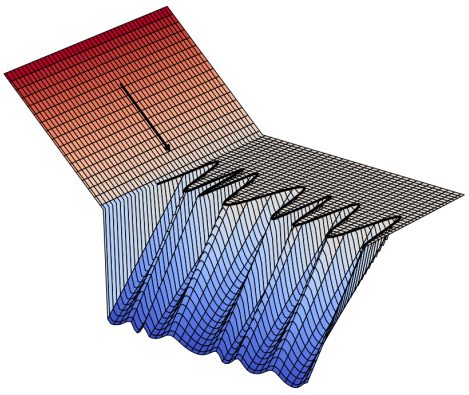
If E_q does not hold, our regret is at most 1, since $\textsc{cost}$ is between 0 and 1. Averaging over the cases where E_q does and does not hold (with probabilities $1 - 1/n^d$ and $1/n^d$), the expected regret of the learning algorithm L (over the randomness in L^* and in the instances) is at most inverse polynomial in n. ∎

5. Conclusions and future directions. Empirical work on application-specific algorithm selection has far outpaced theoretical analysis of the problem, and this paper takes an initial step towards redressing this imbalance. We formulated the problem as one of learning the best algorithm or algorithm sequence from a class with respect to an unknown input distribution or input sequence. Many state-of-the-art empirical approaches to algorithm selection map naturally to instances of our learning frameworks. This paper demonstrates that many well-studied classes of algorithms have small pseudo-dimension, and thus it is possible to learn a near-optimal algorithm from a relatively modest amount of data. While worst-case guarantees for no-regret online learning algorithms are impossible, good online learning algorithms exist in a natural smoothed model.

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

1. Which computational problems admit a class of algorithms that simultaneously has low representation error and small pseudo-dimension (as in subsection 3.4)?
2. Which algorithm classes can be learned online, in either a worst-case or a smoothed model?
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?[36]
4. Are there any nontrivial relationships between statistical learning measures of the complexity of an algorithm class and more traditional computational complexity measures?
5. How should instance features be chosen to minimize the representation error of the induced family of algorithm selection maps (cf. subsection 3.5)?

Appendix A. A bad example for gradient descent. We depict a family $\mathcal{F}$ of real-valued functions (defined on the plane $\mathbb{R}^2$) for which the class $\mathcal{A}$ of gradient descent algorithms from subsection 3.6 has infinite pseudo-dimension. We parameterize each function $f_I \in \mathcal{F}$ in the class by a finite subset $I \subset [0,1]$. The "aerial view" of f_I is as follows:



[36] Recall the discussion in subsection 2.3: even in practice, the state of the art for application-specific algorithm selection often boils down to brute-force search.

The "squiggle" $s(I)$ intersects the relevant axis at exactly I (to be concrete, let $s(I)$ be the monic polynomial with roots at I). We fix the initial point z_0 to be at the tail of the arrow for all instances, and fix ρ_ℓ and ρ_u so that the first step of gradient descent takes z_0 from the upper incline into the middle of the lower flat or wavy areas. Let x_I be the instance corresponding to f_I with starting point z_0. If for a certain ρ and I, $g(z_0, \rho)$ lands in the flat area, gradient descent stops immediately and $\text{COST}(A_\rho, x_I) = 1$. If $g(z_0, \rho)$ instead lands in the sloped, wavy area, $\text{COST}(A_\rho, x_I) \gg 1$.

It should be clear that $\mathcal{F}$ can shatter any finite subset of (ρ_ℓ, ρ_u), and hence has infinite pseudo-dimension. One can also make slight modifications to ensure that all the functions in $\mathcal{F}$ are continuously differentiable and L-smooth.

Acknowledgments. We are grateful for the many helpful comments provided by the anonymous SICOMP and ITCS reviewers.

REFERENCES

[1] N. Ailon, B. Chazelle, S. Comandur, and D. Liu, *Self-improving algorithms*, in Proceedings of the Symposium on Discrete Algorithms (SODA), 2006, pp. 261–270.
[2] M. Anthony and P. L. Bartlett, *Neural Network Learning: Theoretical Foundations*, Cambridge University Press, Cambridge, 1999.
[3] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit, *Local search heuristics for k-median and facility location problems*, SIAM J. Comput., 33 (2004), pp. 544–562.
[4] J. Bergstra and Y. Bengio, *Random search for hyper-parameter optimization*, J. Mach. Learn. Res., 13 (2012), pp. 281–305.
[5] A. Borodin, M. N. Nielsen, and C. Rackoff, *(Incremental) priority algorithms*, Algorithmica, 37 (2003), pp. 295–326.
[6] S. Boyd and L. Vandenberghe, *Convex optimization*, Cambridge University Press, Cambridge, 2004.
[7] *The Budget and Economic Outlook*: 2015 to 2025, U. S. Congressional Budget Office, 2014.
[8] N. Cesa-Bianchi and G. Lugosi, *Prediction, Learning, and Games*, Cambridge University Press, Cambridge, 2006.
[9] K. L. Clarkson, W. Mulzer, and C. Seshadhri, *Self-improving algorithms for convex hulls*, in Proceedings of the Symposium on Discrete Algorithms (SODA), 2010, pp. 1546–1565.
[10] K. L. Clarkson, W. Mulzer, and C. Seshadhri, *Self-improving algorithms for coordinate-wise maxima*, in Proceedings of the Symposium on Computational Geometry (SoCG), 2012, pp. 277–286.
[11] K. L. Clarkson and C. Seshadhri, *Self-improving algorithms for Delaunay triangulations*, in Proceedings of the Symposium on Computational Geometry (SoCG), 2008, pp. 148–155.
[12] L. Devroye, *Lectures Notes on Bucket Algorithms*, Birkhäuser, Basel, 1986.
[13] E. Fink, *How to solve it automatically: Selection among problem solving methods*, in Proceedings of the International Conference on Artificial Intelligence Planning Systems, 1998, pp. 128–136.
[14] A. Gathmann, *Lectures Notes on Algebraic Geometry*, TU Kaiserslautern, 2014.
[15] D. Haussler, *Decision theoretic generalizations of the PAC model for neural net and other learning applications*, Inf. Comput., 100 (1992), pp. 78–150.
[16] E. Horvitz, Y. Ruan, C. P. Gomes, H. A. Kautz, B. Selman, and D. M. Chickering, *A Bayesian approach to tackling hard computational problems*, in Proceedings of the Conference in Uncertainty in Artificial Intelligence (UAI), 2001, pp. 235–244.
[17] L. Huang, J. Jia, B. Yu, B. Chun, P. Maniatis, and M. Naik, *Predicting execution time of computer programs using sparse polynomial regression*, in Proceedings of Advances in Neural Information Processing Systems (NIPS), 2010, pp. 883–891.
[18] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, *Algorithm runtime prediction: Methods & evaluation*, Artif. Intell., 206 (2014), pp. 79–111.
[19] G. J. O. Jameson, *Counting zeros of generalized polynomials: Descartes rule of signs and Laguerres extensions*, Math. Gaz., 90 (2006), pp. 223–234.
[20] D. S. Johnson and L. A. McGeoch, *The traveling salesman problem: A case study in local optimization*, in Local Search in Combinatorial Optimization, E. Aarts and

J. K. Lenstra, eds., Wiley, New York, 1997, pp. 215–310. Reprinted by Princeton University Press, Princeton, 2003.

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

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

[23] D. Lehmann, L. I. O'Callaghan, and Y. Shoham, *Truth revelation in approximately efficient combinatorial auctions*, J. ACM, 49 (2002), pp. 577–602.

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

[25] N. Littlestone and M. K. Warmuth, *The weighted majority algorithm*, Inf. Comput., 108 (1994), pp. 212–261.

[26] P. M. Long, *Using the pseudo-dimension to analyze approximation algorithms for integer programming*, in Proceedings of the International Workshop on Algorithms and Data Structures (WADS), 2001, pp. 26–37.

[27] P. Milgrom and I. Segal, *Deferred-acceptance auctions and radio spectrum reallocation*, in Proceedings of the Fifteenth ACM Conference on Economics and Computation, Palo Alto, CA, Association for Computing Machinery, 2014, pp. 185–186.

[28] M. Mohri and A. M. Medina, *Learning theory and algorithms for revenue optimization in second price auctions with reserve*, in Proceedings of the International Conference on Machine Learning (ICML), 2014, pp. 262–270.

[29] J. Morgenstern and T. Roughgarden, *The pseudo-dimension of near-optimal auctions*, in Proceedings of Advances in Neural Information Processing Systems, 2015, pp. 136–144.

[30] S. Sakai, M. Togasaki, and K. Yamazaki, *A note on greedy algorithms for the maximum weighted independent set problem*, Discrete Appl. Math., 126 (2003), pp. 313–322.

[31] D. A. Spielman and S. Teng, *Smoothed analysis: an attempt to explain the behavior of algorithms in practice*, Commun. ACM, 52 (2009), pp. 76–84.

[32] N. Srebro and S. Ben-David, *Learning bounds for support vector machines with learned kernels*, in Proceedings of the 19th Annual Conference on Learning Theory, 2006, pp. 169–183.

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

[34] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, *Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming*, in Proceedings of the RCRA Workshop on Combinatorial Explosion at the International Joint Conference on Artificial Intelligence (IJCAI), 2011, pp. 16–30.

[35] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, *SATzilla* 2012: *Improved algorithm selection based on cost-sensitive classification models*, in Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT), 2012.