On the Use of ML for Blackbox System Performance Prediction

Silvery Fu, *UC Berkeley;* Saurabh Gupta and Radhika Mittal, *UIUC;*
Sylvia Ratnasamy, *UC Berkeley*

https://www.usenix.org/conference/nsdi21/presentation/fu

This paper is included in the
Proceedings of the 18th USENIX Symposium on
Networked Systems Design and Implementation.

April 12–14, 2021

978-1-939133-21-2

Open access to the Proceedings of the
18th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

On the Use of ML for Blackbox System Performance Prediction

Silvery Fu
UC Berkeley

Saurabh Gupta
UIUC

Radhika Mittal
UIUC

Sylvia Ratnasamy
UC Berkeley

Abstract

There is a growing body of work that reports positive results from applying ML-based performance prediction to a particular application or use-case (*e.g.,* server configuration, capacity planning). Yet, a critical question remains unanswered: does ML make prediction simpler (*i.e.,* allowing us to treat systems as blackboxes) and general (*i.e.,* across a range of applications and use-cases)? After all, the potential for simplicity and generality is a key part of what makes ML-based prediction so attractive compared to the traditional approach of relying on handcrafted and specialized performance models. In this paper, we attempt to answer this broader question. We develop a methodology for systematically diagnosing whether, when, and why ML does (not) work for performance prediction, and identify steps to improve predictability.

We apply our methodology to test 6 ML models in predicting the performance of 13 real-world applications. We find that 12 out of our 13 applications exhibit inherent variability in performance that fundamentally limits prediction accuracy. Our findings motivate the need for system-level modifications and/or ML-level extensions that can improve predictability, showing how ML fails to be an easy-to-use predictor. On implementing and evaluating these changes, we find that while they do improve the overall prediction accuracy, prediction error remains high for multiple realistic scenarios, showing how ML fails as a general predictor. Hence our answer is clear: ML is not a general and easy-to-use hammer for system performance prediction.

1 Introduction

Performance prediction has long been a difficult problem, traditionally tackled using handcrafted performance models tailored to a specific application [26, 27, 43, 44, 53, 56, 61, 62] or use-case[1] [23, 30, 55, 58]. However, this approach is tedious, doesn't generalize, and is increasingly difficult given the growing complexity of modern systems. Ideally, one would want a predictor that is **accurate, general, and easy to use**. By *general*, we mean an approach that applies to a broad range of applications and a broad range of use-cases; by *easy to use*, we mean an approach that can be applied without requiring detailed knowledge of the application internals or use-case.

[1] Throughout this paper, we use the term *use-case* to refer to an application of prediction such as scheduling (*e.g.,* where/when to run jobs), configuration (*e.g.,*, how many workers or how much memory to use), or capacity planning (*e.g.,* determining what server configurations to purchase).

Given the success of machine learning (ML) in many domains, it is natural to think that ML might offer a solution to this challenge: *i.e.,* that an ML model can learn the relationship between a system's externally observable features and its resultant performance *while treating the application as a black-box*. The ability to treat the application as a black-box and remain agnostic to the prediction's use-case would enable a predictor that meets our goals of generality and ease-of-use.

But does ML deliver on this promise? Several recent efforts have applied ML to predict or optimize performance [22, 37, 47, 57, 60]; these report positive results and hence one might assume that the answer to our question is "yes." However, as we discuss in §9, these effort focus on specific applications, models, or use-cases and hence do not shed light on our question of broad generality and ease-of-use. In this paper, we take a first step towards filling this gap, empirically evaluating whether ML-based prediction can simultaneously offer high accuracy, generality, and ease-of-use.

The first step in such an undertaking is to define a methodology for evaluation. As we shall show, evaluating the accuracy of a model's prediction is subtle particularly when prediction fails because in such cases we need to understand the cause of failure: was it a poor choice of model? was the model poorly tuned? or was the application's performance somehow fundamentally not predictable? In other words, we need a methodology that allows us to both, evaluate the accuracy of a predictor and *attribute* errors in prediction.

This observation led us to define a methodology for systematic evaluation and analysis of ML-based predictors that, as we detail in §2, provides us with two bounds: a lower bound on the prediction error that *any* model can hope to achieve, and a more realistic bound that is based on the best prediction made by the ML models that we consider. We apply our methodology to evaluate 6 different ML models (*e.g.,* k-nearest neighbors, random forest, neural networks) in predicting performance for 13 real-world applications (*e.g.,* Tensorflow, Spark, nginx) under a range of test scenarios (*e.g.,* predicting the impact of dataset size).

Our first key finding is that *irreducible* prediction errors are common (§4). In particular, we find that the majority of our applications exhibit a high degree of performance variability that cannot be captured by any black-box parameters and that manifests even in best-case scenarios (*e.g.,* running an identical configuration of the application, on identical hardware, with no contention for resources). E.g., we show that, even

across identical runs, the performance of the JVM Garbage Collector (GC) varies between two modes depending on the precise timing of GC events (§5). Because of this variability, our lower bound on prediction error is non-trivial: we show that no application consistently achieves a lower bound on prediction error that is <10% and many applications fare far worse; *e.g.,* the lower bound on prediction error in memcached is >40% in ∼20% of our prediction scenarios. Borrowing the terminology of the ML community [34], we say that the prediction error that results from this variability is "irreducible" as it stems from behavior that cannot be modeled or controlled, and hence cannot be learned. Irreducible error fundamentally limits the accuracy that *any* ML model can achieve.

We further find that, while the root cause of irreducible error varies across applications, a common theme is that they stem from design decisions that were made to optimize performance, efficiency or resilience but in the process led to a fundamental trade-off between predictability and these other design goals (§5).

These findings suggest a clear negative result for our goal of an ML-based predictor that is both accurate and general. So, where do we go from here? A natural follow-on is to ask whether we can usefully relax our goals of generality and ease-of-use - *i.e.,* make some assumptions about application design or prediction use-cases that would improve predictability.

The second part of our paper explores this question. We do so from two different vantage points: that of the application developer and that of the operator who is using predictions for some operational task.

Our *developer-centric* exploration asks: if developers expose knobs that give operators the *option* to disable design features that lead to irreducible errors, how much would this improve the accuracy of ML predictors?

Our *operator-centric* exploration asks: if we assume operators can accommodate some notion of uncertainty in how they use the predictions, then could ML meet our goals for a useful predictor?

We note that both of the above represent a non-trivial compromise on our original goal and, perhaps more importantly, in neither case do we address the question of *how* developers/-operators would make such changes nor the impact that such changes might have on other design goals such as efficiency, resilience, etc. Instead we are merely asking whether making these changes would improve predictability.

Our findings on this front are mixed. We find that, in both cases, our relaxed assumptions do significantly improve predictability in our best-case scenarios, but we continue to see prediction fail in a non-trivial fraction of our more realistic tests. E.g., in our best-case setup, the lower bound on prediction error is now <6% in >90% of our test cases but, in our more realistic tests, 3 (out of 13) applications see error rates >30% in ∼10% of test cases.

While not the clean result that one might have hoped for, they reinforce that ML-based performance prediction is best applied with a scalpel rather than a hatchet. In this sense, our study mirrors the extensive literature in applied machine learning that explores the trade offs between black- *vs.* gray-box learning. While this trade off has been studied (and debated!) in many other domains [35, 38, 54], to our knowledge we are the first to do so for performance prediction in systems.

Thus unlike many NSDI papers, our contribution lies not in the design and implementation of a particular system but instead in triaging and critically examining the role of ML for managing system performance. Specifically:

- We provide the first broad evaluation of the generality of ML-based prediction, showing that blackbox prediction is often fundamentally limited and expose why this is the case.
- In light of these limitations, we propose and empirically evaluate two complementary approaches aimed at broadening the applicability of ML-based prediction and show that these approaches alleviate but don't eliminate the above limitations.

We view our study as a first step and recognize that it must be extended to more applications and models before we can draw final conclusions on the generality of ML-based performance predictors. We hope that our methodology and results provide the foundation for such future work.

2 Methodology

Our methodology is based on two tests and two predictors: the Best-Case (BC) and Beyond Best-Case (BBC) tests, plus the Oracle and Best-of-Models predictor. In what follows, we first define our metrics and parameters followed by these tests and predictors.

2.1 Metrics and Parameters

We test prediction accuracy by comparing an ML model's prediction to the true performance measured in our experiments. We measure quality of predictions using the root mean square relative error (rMSRE) which is computed as:

$$rMSRE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{Y_i - f(X_i)}{Y_i}\right)^2}, \quad (1)$$

where n is the number of points in the test set for a given prediction scenario and (X_i, Y_i) is a test sample where Y_i is the true measured performance and f is the function learned by the ML model that, given a test feature of value X_i, predicts the performance as $f(X_i)$. rMSRE is a common metric used in regression analysis [34] and in prior work on performance prediction [60]. Note that rMSRE measures the true error in predicted performance - *i.e.,* comparing predicted to actual performance. It is possible that a predictor with high rMSRE is still "good enough" for a particular use-case[2] and, indeed, it is common in papers that focus on specific use-cases to evaluate prediction in terms of the benefit to their use-case. However, given our goal of generality and remaining agnostic

[2]For example: say that a predictor must pick between two configurations c_1 and c_2 where c_1 leads to a JCT of 10s and c_2 to 100s, then even a predictor with an rMSRE of 50% would successfully pick the right configuration.

to the specifics of a use-case, we believe rMSRE is the correct metric for a *predictor* and one can separately study what prediction accuracy is required for a target *use-case*.

We consider the following three classes of parameters that impact an application's performance[3]:
(i) **Application-level input** parameters capture inputs that the application acts on; *e.g.,* the records being sorted, or the images being classified. We consider both the size of these inputs and (when noted) the actual values of these inputs.
(ii) **Application-level configuration** parameters capture the knobs that the application exposes to tune its behavior – *e.g.,* the degree of parallelism, buffer sizes, etc.
(iii) **Infrastructure** parameters capture the computational resources on which the application runs – *e.g.,* CPU speed, memory size, whether resources are shared vs. dedicated.

These parameters correspond to what-if questions that are likely to be of practical relevance: *e.g.,* predicting how performance scales with input data size, under increasing parallelism (executors), or with more infrastructure resources. We note that the above parameters, when used as features for our ML models, capture the application as a black-box, together with the infrastructure it runs on. We also note that these are static parameters known prior to running the application, in contrast to runtime metrics and counters such as a task's CPU or cache utilization. While runtime counters are widely used for monitoring performance, relying on them for prediction limits our potential use-cases to scenarios where prediction is invoked after the application is *already running* rather than at the time of planning, placement, or scheduling. Moreover, it is unclear how one might *use* a predictor based on runtime counters to manage performance: *e.g.,* even if an operator can predict that a CPU utilization value of C leads to a desired performance target, she must still know how to set system parameters in order to achieve the desired CPU utilization. In other words, runtime counters are *measures* not parameters.[4] Hence, in this paper, we assume the ML models cannot use runtime counters as features for prediction.

2.2 The Best-Case (BC) Test

Our BC test is designed to give a predictor the "best chance" at making accurate predictions. It does so by making very strong assumptions on both the systems and ML front, as we describe below.

(1) ML front: simplifying the predictive task. The BC test makes two assumptions that greatly simplify the prediction task. First, in all the data given to the model (training and test), only a single parameter is being varied and that parameter is the *only* feature on which the model is trained. In other words: say we ask an ML model M to predict an application's performance for a particular configuration c_i that is defined by k parameters: i.e., $c_i =< p_1 = X_1, p_2 = X_2, p_3 = X_3, \ldots, p_k = X_k >$, where the p_i are parameters and X_i are parameter values. In the BC test, we give M a training data set in which only one parameter - say p_2 – is varied and all other parameters are set to the test value; *i.e.,* all training data will come from runs where p_2 is varied while $p_1 = X_1, p_3 = X_3, \ldots, p_k = X_k$. We call this our **one-feature-at-a-time** assumption.

Second, the model's training data always includes datapoints from the scenario it is being asked to predict. I.e., continuing with the above example, when predicting the application's performance for an input configuration $<p_1 = X_1, p_2 = X_2, p_3 = X_3, \ldots, p_k = X_k>$, not only do we enforce the one-feature-at-a-time assumption, we also require that the training set for the task include datapoints with $p_2 = X2$. Hence, the training set contains data points from an identical configuration to the one the model is being asked to predict! For example, if we ask the model to predict the time it takes to sort a dataset of size 5GB, the training set already includes times for sorting the same 5GB dataset. We call this the **seen-configuration** assumption since it ensures that, during training, the ML model has already "seen" the configuration it is being asked to predict.

We emphasize the extreme simplification due to the above assumptions: the complexity of prediction has been reduced from understanding the impact of k features to just one feature (*e.g.,* p_2), the training data is deliberately selected to cleanly highlight the impact of this feature, *and* the training set includes data from test configurations that are *identical* to what the model is being asked to predict!

(2) Systems front: best-case assumptions. Given our assumptions so far, the only reason prediction might be non-trivial is if we see variable performance across repeated runs of the application even with a fixed configuration of parameters. That software systems may exhibit variable performance is well recognized in the systems community with two commonly cited reasons for this: (i) contention for resources that an application might experience when it shares physical infrastructure with other applications/tenants [33,40,49,52] and (ii) variability that arises when the processing time depends on the values of data inputs [29] Our assumptions on the systems side aim to remove the likelihood of such variability. To avoid variability due to contention, we run our workloads on *dedicated* EC2 instances [1], and only run a single experiment at a time on a given server. We call this our **no-contention** requirement. We are still left with the possibility of contention within the datacenter network.Our workloads are not network-heavy and, with one exception (nginx, discussed later) none of our workloads appear to be impacted by contention with other apps over network bandwidth and hence we optimistically conclude that network contention is unlikely to have affected predictability for our workloads. We still cannot *entirely* avoid contention, however – *e.g.,* some of our apps use shared cloud

[3] We define these precisely in the context of each application in §3.

[4] I.e., one would need to predict how a parameter impacts the runtime metric in addition to how the runtime metric impacts performance. This might be appropriate for certain use-cases that include long-running jobs, *e.g.,* [61], and where we cannot directly predict how parameters impact performance but, for now, we focus on understanding whether the more general and direct/simple approach works.

services such as the S3 and DNS. Nonetheless, we believe the scenario we construct is far more conservative than what is commonly used in production and hence we view it as a pragmatic approximation of the best case while still running on real-world deployment environments like EC2.

Our second assumption is that, for a given input dataset size, the application's input data is identical across all experiments. I.e., repeated runs of a test configuration act on the same input data. E.g., all training/test data for an application that sorts N records, will involve exactly the same N records. We call this our **identical-inputs** assumption.

The sheer simplicity of our prediction tasks should be immediately apparent:we're essentially asking a model to predict performance for a test configuration that is identical to that seen in training. Our expectation was that, under these conditions, a model should be able to predict performance with a very high level of accuracy - *e.g.,* with error rates well under 5-10% – and if a model cannot accurately predict performance under the above conditions then it is unlikely to be a useful predictor under more realistic conditions.

2.3 The Beyond Best-Case (BBC) Test

In the BBC test, we systematically relax each of the constraints/assumptions imposed in the BC test to study prediction accuracy under more realistic scenarios:

(i) Relaxing the seen-configuration assumption. For this, we perform a leave-one-out analysis in which all data samples corresponding to the configuration on which a model is tested are withheld from the training set. More precisely, for a performance dataset with N configurations $\{c_1, c_2, \ldots, c_N\}$, when testing a datapoint with configuration c_i, we use a model trained on a dataset that consists of the $N-1$ configurations *other* than c_i; *i.e.,* our training set excludes all training datapoints for configuration c_i. We perform this N-fold leave-one-out analysis for each of our prediction scenarios.

Note that predictions are now harder as models must learn the *trend* in performance as a function of the parameter being varied. Such predictions are useful in answering *what-if* questions of the form: *"what performance can we expect if we increase the number of workers to 10?"*

(ii) Relaxing the one-feature-at-a-time assumption. We relax our constraint of varying only one parameter at a time and instead enumerate the configuration space generated by simultaneously varying all the features in question and then sample from this space to collect training and test data on which we rerun our predictions. We describe the details of our approach inline when presenting our results.

(iii) Relaxing the no-contention assumption. For this, we repeat the experiments used to collect training and test data but this time do not run our experiments on dedicated EC2 instances. We present additional detail on our experimental setup in the following section.

(iv) Relaxing the identical-inputs assumption. For this, we generate a different input dataset for each datapoint in the training and test set. Section 3 provides additional detail on our input datasets in the context of each test application.

We studied the impact of relaxing each of the above assumptions individually and then all together. In this paper, we present a subset of our results as relevant.

2.4 Predictors

As mentioned, our evaluation considers two predictors.

The Best-of-Models (BoM) predictor. Recall that in order to obtain a broad view of ML-based performance prediction, we consider a range of ML models (§3). For any given prediction test, we compute the rMSRE for each ML model, and define the *best-of-models error (BoM-err)* as the minimum rMSRE across all the models we consider. Thus BoM-err tells us how well *some* ML model can predict system performance. However, if BoM-err is high, we still cannot tell whether this is because of a poor choice or tuning of ML models, or whether performance prediction was inherently hard. For this, what we would like to have is a lower bound on the error rate we can expect from *any* ML model. We achieve this through our *Oracle Predictor*.

The Oracle predictor looks at all the data points in the *test* set that share *the same feature values* as the prediction task, and returns a prediction that will minimize the metric in Eqn. 1 for all these data points. We obtain this by differentiating the expression for the metric in Eqn. 1 with respect to the prediction and finding the global optima for each *unique feature value*, , as below.

$$f_{\text{oracle}}(X) = \left(\sum_{i=1}^{n} \frac{\delta(X_i,X)}{Y_i}\right) / \left(\sum_{i=1}^{n} \frac{\delta(X_i,X)}{Y_i^2}\right), \quad (2)$$

$$\delta(a,b) = 1 \text{ if } a \text{ is equal to } b, \text{ and 0 otherwise.} \quad (3)$$

Note that our features are discrete entities (number of workers, size of dataset, type of instance) and thus the oracle is well-defined. We use O-err to denote the error rate obtained by this Oracle. If there is no variance at all in these data points, the Oracle will achieve zero error. Simply put, O-err quantifies the impact of variance in performance – across multiple runs of the same application and under *identical* configurations – on prediction accuracy.

Clearly, our Oracle predictor is not usable in practice since it is allowed to "peek" at both the test data and the error function; nonetheless O-err is helpful in attribution. Specifically, it gives us a lower bound on the prediction error that *any* ML model could achieve and, in this sense, sheds light on whether performance prediction is at all feasible, *i.e.,* a high O-err perhaps suggests that predicting system performance is "impossible." In addition, a small gap between O-err and BoM-err confirms that our model is well tuned (§3).

3 Test Setup

Our experimental setup comprises two main stages: application profiling and model training. In the profiling stage, an application is run under different configurations to generate a

raw dataset. In the training stage, the dataset first undergoes pre-processing (featurization, normalization, and outlier removal), and is then randomly split into two disjoint datasets: the *training* set is used to train models and the *test* set is reserved for model evaluation. The training set is also used for hyper-parameter tuning of ML models. In what follows, we discuss key aspects of each stage. This is not an exhaustive description of our experimental setup – all datasets [6] and tooling [20] from our experiments are publicly available.

3.1 Application Profiling

Applications. Table 1 enumerates the applications used in our study. Our selection reflects multiple considerations including the application's relevance in production environments, diversity (spanning web services, timeseries database, microservices, data analytics, and model serving), and ease of instrumentation. See Appendix A for additional details.

Parameter Values and Performance Metrics. We select values for our three broad classes of parameters as follows: (i) *application-level input parameters.* We experiment with varying the size of these inputs on a scale of 1 to 10, with scale 1 being the default input size in the workload generator; (ii) *app-level configuration parameters.* We experiment with varying the number of worker nodes between 1 and 10. (iii) *infrastructure parameters.* We experiment with 13 different EC2 instance types: from the 150+ instance types offered on EC2 we select the latest generation of the three common instance families (c5, m5 and r5) with four different scales (large, xlarge, 2xlarge, 4xlarge) plus a c4.4xlarge instance for a total of 13 instance types (see [2] for details). This selection matches the instances used in prior work [57, 60].

As listed in Table 1, we use different performance metrics (*e.g.,* job completion time, request throughput) depending on the application. We run each parameter setting 10 times recording the resulting performance. This constitutes our raw dataset. Appendix A and our code repository [20] include additional details on the setup.

3.2 Model Training

We select six ML algorithms: k-nearest neighbors, random forest regression, linear regression, linear support vector machine (SVM) regression, kernelized SVM regression, and neural networks. We select these as they are commonly used in practice and, taken together, represent well the various families of ML techniques: parametric and non-parametric models, linear and non-linear models, discrete and continuous [34]. This diversity is in contrast to prior work that focuses on a single ML technique [22, 37, 47, 57, 60].We follow the standard machine learning practice of k-fold cross-validation (k=3) for setting hyper-parameters. Folds were carefully picked such that they represented the data well and included points for each feature value present in the training set. We searched for regularization parameters (all models); kernels (RBF, polynomial, and sigmoid) and their parameters for SVMs; number and depth of trees for random forests; number of neighbors for nearest neighbors; and number of hidden layers (varying from zero to four), size of layers, activation functions (ReLU, tanh), and learning rates for neural networks. The neural networks we used, MLPs with different non-linearity, are the standard models for treating data of the form in this paper (as opposed to RNN/LSTM/CNNs).

As is standard in machine learning, we pre-process our dataset before training our models. We convert numeric features to have zero mean and unit standard deviation (by subtracting the mean and dividing by the standard deviation, computed per feature channel across the entire training set). We map all categorical features to their numerical index. We also throw out outlier data-points (that have any feature value or performance metric beyond the 99$^{\text{th}}$ percentile) from the training set. We use `scikit-learn`, a widely used machine learning tool-set for data preparation and model training [46].

We then train each model to predict the impact of a particular parameter p on performance for a given app A. For this, we select data points corresponding to runs of A in which *all* parameters other than p are fixed. We do a 50:50 random split of the resultant dataset into a training and test set.

4 Results: Existing Applications and Models

We start with the results from our BC test. Recall that each prediction task involves predicting the performance of a given application for a given configuration of: (i) application input size, (ii) number of workers, and (iii) instance types, while subject to the constraints and assumptions presented in §2.2.

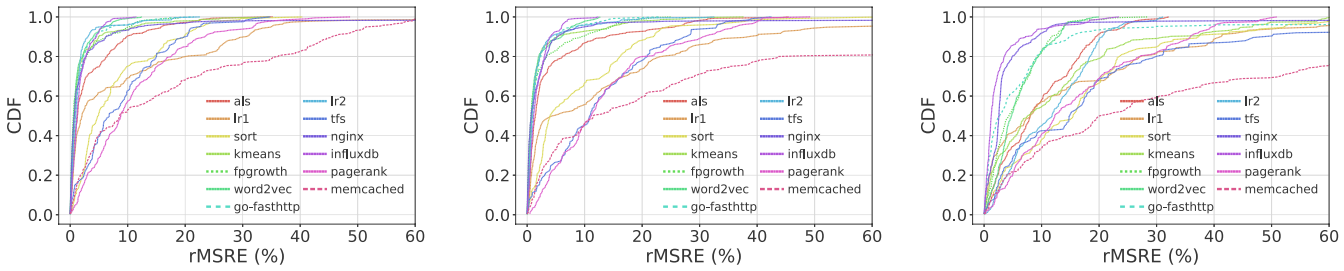
We start by looking at the results for our BC test. Fig.1a plots the cumulative distribution function (CDF) of the O-err for each application across all predictions tasks while Fig.1b shows the same for the BoM-err.[5] Recall that O-err captures the irreducible error inherent to an application (i.e., no ML model could achieve an error rate lower than the O-err) while the BoM-err captures the lowest error rate achieved by some ML model. Given the extreme simplicity of our best-case prediction task, we expected a very high degree of accuracy with an O-err of (say) well under 5% error. To our surprise, our results did not match this expectation. For example: in 5 of our 13 applications, O-err is >15% for at least 20% of prediction tasks; for LR1 O-err is >30% for 10% of the prediction tasks; for memcached O-err exceeds 40% for approximately 20% of prediction tasks; in fact, no application enjoys an O-err of <10% in all prediction scenarios.

The high O-err that we see in our (extremely generous) best-case scenario tells us that many applications suffer from non-trivial irreducible error which fundamentally limits our ability to achieve high prediction accuracy under the general black-box conditions that we desired. Given that we fail even at the BC test, we focus next on understanding the sources of this irreducible error. But first, a few additional observations:

[5] We show a detailed breakdown of error based on the feature being predicted in Appendix C.

Framework	Application/Description	Input Workload	Input Parameter	App. Config. Parameter	Infra. Parameter	Metric
Memcached [12]	Disributed in-memory k-v store	Mutilate [13]	value size	# servers	inst. type	mean query lat.
Nginx [9]	Web server, LB, Reverse Proxy	Wrk2 [5]	req. rate	# servers	inst. type	median req. lat.
Influxdb [15]	Open source time series database	Inch [10]	# points per timeseries	# servers	inst. type	mean query lat.
Go-fasthttp [7]	Fast HTTP package for Go	wrk2 [5]	# conn.	# servers	inst. type	median req. lat.
Spark [3]	*TeraSort:* sorting records	TeraGen	# records	# executors	inst. type	JCT
	PageRank: graph computation	GraphX SynthBenchmark [8]	# vertices			
	LR1: logistic regression	MLLib examples	# examples			
	LR2: logistic regression	Databricks Perf Test [16]				
	KMeans: clustering					
	Word2vec: feature extraction					
	FPGrowth: data mining					
	ALS: recommendation					
TensorFlow [17], Kubernetes [11]	*TFS:* Tensorflow model serving	Resnet examples [18]	# conn.	# servers	inst. type	requests/sec

Table 1: Applications, input workload, parameters, and metrics used in our study.



(a) CDF of O-err in the BC test

(b) CDF of BoM-err in the BC test

(c) CDF of BoM-err in the BBC test

Figure 1: O-err and BoM-err for existing applications and ML models.

(i) As one might expect, prediction accuracy deteriorates as we move from the BC to the BBC test. As a sample result, Figure 1c shows the CDF of the BoM-err for our BBC test when relaxing our seen-configuration assumption: for 6 of our 13 applications (*vs.* 3 in the BC test), the 80%ile BoM-err exceeds 20% (for 1 app, it exceeds 60% error!)
(ii) As the CDFs suggest (and as we validated on individual data points), BoM-err on our BC test tracks O-err very closely, confirming that the higher-than-expected errors arise from irreducible causes, as opposed to poor ML engineering.
(iii) The error rates vary across applications even when they use the same framework (*e.g.,* Spark-based sort vs. LR2). This reinforces the importance of considering a range of applications when evaluating ML for performance prediction.

5 Tackling Irreducible Error

The high irreducible errors that we saw in the previous section tells us that, even for a fixed configuration of parameters, there are times when an application's performance was so variable that *no* predictor could predict performance with high accuracy. While it is well recognized that software systems can experience variable performance due to various runtime factors, we were surprised to see the extent of this variability even in our best-case scenario. Recall that, in our best-case scenario, we are essentially just repeatedly running an identical software stack, on identical hardware, with identical data inputs, and without contention on our servers. What can cause performance to vary significantly across runs? In particular, we were interested in understanding whether the sources of variability could be captured by features that are known prior to runtime. If so, we could hope to achieve higher prediction accuracy by simply adding more features to our ML models.

Unfortunately, we find that this variability stems from design choices – often optimizations – the effect of which could *not* have been captured by system parameters known prior to running the application. In this section, we briefly summarize our findings (§5.1) and then discuss their implications (§5.2).

5.1 Root-Cause Analysis

Table 2 summarizes the findings from our root cause analysis. Each entry summarizes the root cause we discovered, the applications it impacted, the trade-off that eliminating this root cause would impose, the manner in which we modified applications to eliminate/mask their impact, and the burden associated with undertaking such analysis. We stress that the modifications we make are ad-hoc hacks intended only to verify that we correctly identified our root-causes; as we discuss later, we do *not* view them as the desirable long-term fix.

Finally, we report the person hours it took to identify and validate these root causes. We recognize that any such estimate depends on many nebulous factors such as our familiarity with the codebase and experience in performance analysis. We present it merely as anecdotal evidence that analyzing performance in modern systems is non-trivial and would benefit from better developer support as we discuss in §8.

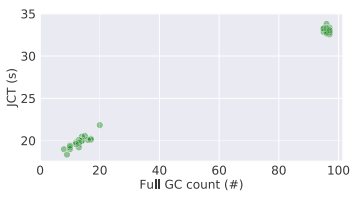
To give the reader a tangible sense of these root causes, we first present a longer description of the first two root causes, followed by a very brief summary of the remaining ones. A detailed description – with experimental validation – is in Appendix B, while the following section (§6) evaluates the impact of eliminating these root causes on O-err and BoM-err.

5.1.1 Spark's "Worker Readiness" Optimization

We found that the relatively high O-err in Spark's Terasort application stems from a dynamic sizing optimization in Spark. By default, Spark launches an application once at least 80% of its target worker nodes are ready, and the application partitions the input dataset based on the number of workers ready at this time. This optimization ensures resilience to failure and stragglers, but leads to variable parallelism and hence JCTs. This variability leads to irreducible errors and cannot be captured by any input parameters/features as the exact degree of parallelism is affected by small differences in worker launch times which cannot be predicted prior to runtime.

Disabling this optimization lowered Terasort's average O-err from 12.6% to 2.6% in our predictions that involve varying the instance type. While this optimization also affected other Spark apps (*e.g.,* PageRank), its impact there was small.

5.1.2 Adaptive Garbage Collection in the JVM



Our analysis showed that LR1's error stems from an optimization in the Java Virtual Machine's (JVM) garbage collector (GC). Specifically, we found a positive correlation between the number of "full" GC events (explained below) and JCT, leading to the bimodal behavior in the figure above.

As a performance optimization, the JVM GC divides the memory heap into two regions – young and old – and typically tries to constrain garbage collection to just the young region. However, in situations where the memory heap is consistently low on free space, the JVM GC runs a "full" collection that operates over the entire heap space (young and old regions). To determine whether a full GC is needed, the JVM maintains a *promotion estimate* metric. Our analysis revealed that due to minor differences in the timing of memory allocation and GC events, the promotion estimate ends up just above the threshold (triggering a full GC) in some runs and just below it in others, leading to the bimodal behavior (see Appendix B.2). Once again, since the exact mode being triggered depends on the detailed timing of runtime events, this effect could not have been captured by any input parameters/features known prior to actually running the job.

We verify our analysis by rerunning our experiments with an extra 50MB of memory which ensures that the promotion estimate remains below the threshold in all runs. This eliminates the high-mode runs, reducing average O-err by 9× from 18.2% to 2.1% in our predictions that involve varying the number of workers.

5.1.3 Other Root Causes

We briefly mention the remaining entries in Table 2; detailed tracing for each is in Appendix 9.

HTTP Redirection and DNS Caching in S3. We found that multiple applications built on Amazon's S3 storage service suffered variable performance that arose from the DNS-based resolution of S3 object names; some name resolution requests experienced HTTP redirects while others didn't depending on the detailed timing of when DNS updates were propagated.

Imperfect Load-Balancing. We observed high irreducible errors when predicting the request throughput of a Tensorflow Serving (TFS) cluster. We run TFS within a Kubernetes cluster and found that this error stems from the randomized load-balancing policy that Kubernetes employs, which leads to an inherent imbalance in the load at each server; when running the cluster at high utilization this imbalance led to some servers being overloaded and the *variability in this imbalance* leads to variations in the overall request throughput.

Non-deterministic Scheduling. We found that independent Spark tasks were being scheduled in different orders across different runs leading to different JCTs. This variability arose because the data structure used to track runnable tasks (Scala's `HashSet`) offers no guarantees on the order of iteration through the set members. Switching to a different data structure eliminated this variability.

Variability in Cloud APIs. Our last two cases of high O-err stemmed from variability across repeated invocations of cloud APIs: memcached was affected by variability in how worker instances were placed (which affects node-to-node latency) while nginx suffered from variability in the default network bandwidth associated with particular instances types. Again, with little visibility into (or control over) cloud API implementations, the impact of this variability could not be predicted by input parameters/features prior to runtime.

5.2 Implications and Next Steps

At a high level, many of the irreducible errors we encountered may be attributed to two common design techniques: the use of randomization (*e.g.,* in load-balancing, scheduling) and the use of system optimizations in which a new mode of behavior is triggered by a threshold parameter (*e.g.,* a promotion estimate, timeouts, a threshold on available workers). Some of these design choices can be altered with little loss to design goals such as performance or resilience (*e.g.,* remov-

Root Cause	Applications Impacted	Trade-off	Modification	Effort to diagnose
Spark's "start when 80% of workers are ready" optimization	Terasort	Decreased resilience to stragglers and worker failure	Disable optimization	5 person days
Multi-mode optimization in JVM Garbage Collector	LR1	Slower garbage collection	Avoid triggering, or disable, optimization	39 person days
Non-determinism in Spark sched.	PageRank	None	Use deterministic data structure	14 person days
HTTP redirects and DNS caching in S3's name resolution	KMeans, LR2, FPGrowth, ALS	Decreased flexibility[6] (OR slower name resolutions)	Client-side caching of HTTP redirects (OR always redirect)	10 person days
Imperfect load-balancing at high load	TensorFlow serving	Load imbalance when each server has different numbers of workers	Modified load-balancing policy to always favor local workers	7 person days
Variability in implementations of Cloud APIs (EC2)	memcached, Nginx	Cloud APIs expose more information (less flexibility)	Use AWS placement APIs / include inter-node RTTs as ML feature	5 person days

Table 2: Root causes of the irreducible errors we observed in our test applications. Person hours were calculated using the timestamps in our debugging logs and covers the entire process of reproducing observed behavior, adding instrumentation, processing logs, modifying the system to eliminate suspected causes, running tests for validation, etc.

ing the non-determinism in Spark's scheduler). However, for many others, eliminating them would come at some loss in performance/efficiency (*e.g.,* DNS caching improves scalability, partitioning regions offers faster GC). Moreover, because of the recent emphasis on extracting performance, modern systems now make extensive use of such optimizations.

Given the lower bound set by the O-err (as discussed in §2.4) and the underlying root causes, no amount of model modifications or feature engineering would have resulted in significantly better prediction accuracy. So, where do we go from here? We set out to test an ambitious hypothesis: that ML could serve as a general and easy-to-use predictor of system performance. Unfortunately, we found that most of the applications we studied suffered poor prediction accuracy on a non-trivial number of test cases even under extremely favorable test conditions. Moreover, the design choices that led to low accuracy cannot be easily forsaken without impacting other goals such as performance.

Thus a natural follow-on is to ask whether we can usefully relax our goal of generality - *i.e.,* make some assumptions about application design or prediction use-cases that would still allow us to leverage ML-based prediction in a (mostly) general and easy-to-use manner. As mentioned in §1, the remainder of this paper explores this question from the vantage point of application developers (§6) vs. operators (§7).

6 Results: Modified Applications

We now examine the impact of removing the above root-causes on performance predictability. If doing so improves prediction accuracy, then we can envision a workflow where the application developer identifies the root causes of irreducible errors and makes them configurable. For example, the JVM garbage collector (GC) designer can expose a knob to turn off the optimization for reducing full GCs. This, in turn, would enable system operators to disable such techniques depending on their desired trade-off between predictability and other design goals such as performance.

We emphasize that our goal here is not to provide a mechanism for identifying and eliminating sources of irreducible errors. Developing a systematic approach for this is an interesting problem but beyond the scope of this paper. Instead, we focus on the consequences of doing so: *if* these sources are identified and eliminated, to what extent would it improve performance predictability? Further note that, in line with this objective, we focus on evaluating the resultant impact on performance *predictability*, and not on performance itself.[7]

We now (re)evaluate predictability for our BC (§6.1) and BBC (§6.2) tests after applying the "fixes" to remove the root causes of irreducible errors as discussed in §5.

6.1 BC test results

Fig. 2a shows the CDF of the O-err for our modified applications across all prediction tasks. As expected, removing the sources of irreducible errors results in a dramatic reduction in O-err. All applications now have O-err well within 10% for at least 90% of their prediction tasks. In other words, 90%ile O-err is $< 10\%$ for all 13 applications, and in fact, only two applications have O-err $> 6\%$.

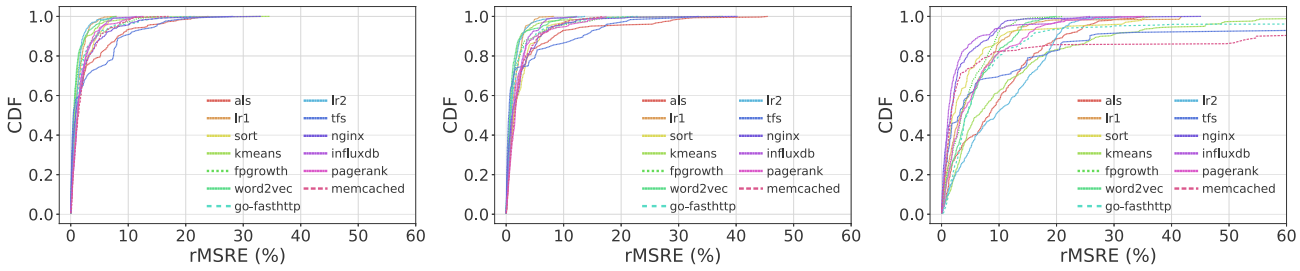
Fig. 2b shows the corresponding CDF for BoM-err. Again, only two applications have 90%ile BoM-err $> 6\%$, of which only one (TFS) has 90%ile BoM-err $> 10\%$. Further observe that the trends for BoM-err closely track those of O-err above, highlighting that ML comes close to achieving the lower bound on prediction errors for our simple BC predictions.

6.2 BBC test results

With the promising BC test results, we next turned our attention to BBC tests after application modifications. We systematically relaxed the assumptions of our BC test (as described in §2), and present a subset of results for brevity.

Fig. 2c presents the CDF for BoM-err after we relax the seen-configuration assumption by performing a leave-one-out test as described in §2. Note that, since the test dataset remains unchanged, O-err is the same as in our BC prediction above. Comparing Figures 2b and 2c, we see that the BoM-

[7] In fact, many of our "fixes" would actually improve performance (*e.g.,* allocating more memory, caching the correct server address) and hence including such results would be misleading!



(a) CDF of O-err in the BC test

(b) CDF of BoM-err in the BC test

(c) CDF of BoM-err in the BBC test

Figure 2: O-err and BoM-err in the BC and BBC test after modifying applications to remove sources of irreducible errors. Note the sharp drop in performance prediction errors, as compared to unmodified applications in Fig. 1.

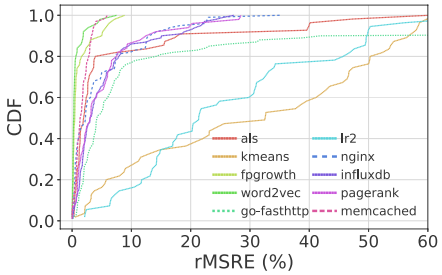


Figure 3: CDF of BoM-err in the BBC leave-one-out test where input data content is not identical, for predictions across varying input data scale. Relaxing the identical-inputs assumption reduces the prediction accuracy for multiple applications.

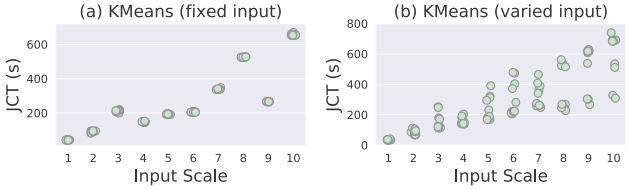


Figure 4: JCTs for KMeans with fixed vs. varied inputs for each value of input scale. KMeans JCT is sensitive to the input dataset.

err is significantly higher for the BBC leave-one-out test than for the BC test. 10 out of the 13 applications exhibit 90%ile BoM-err $>$ 10%. Some degradation in prediction accuracy is expected due to the increased difficulty of the prediction tasks, which now require the ML models to learn the *trend* in performance. What is surprising is that multiple applications experience exceptionally high BoM-err for a significant fraction of prediction tasks. For example, the 90%ile BoM-err of memcached exceeds 60%, while that of KMeans and TFS exceeds 25%. As can be seen, the corresponding 95%ile BoM-err is even higher. We dig deeper into the reasons behind these high errors in §6.3.

While we observe these high prediction errors for only a subset of applications and prediction scenarios, these results emphasize that we cannot simply rely on ML as a general predictor that works across all apps and prediction scenarios.

We next present results from tests in which we relaxed the identical-inputs assumption in addition to relaxing the seen-configuration assumption. In other words, we perform a leave-one-out analysis, but now generate a different input dataset for each datapoint in the training and test set. We do so for 10 applications by varying the random seed in the workload generator, keeping the distribution underlying the input data unchanged. Fig. 3 shows the corresponding CDF for BoM-err across prediction tasks where we vary input scale. We observe that, in general, relaxing the identical-inputs assumption further reduces the prediction accuracy for multiple applications.[8] The applications that are most notably impacted are KMeans and LR2. This is because the performance of these applications is sensitive to the input data. For example, as we show in §6.3, the number of iterations for KMeans, and therefore its JCT, depends on the actual values/content of the input data. This results in a multi-modal behaviour for a given input scale, and thus, high prediction errors. As expected, the corresponding O-err (not shown for brevity) is also high.

Finally, we relaxed our no-contention assumption by repeating the above experiments on non-dedicated (or "shared") instances and found that doing so did not produce statistically meaningful differences in our results. In particular, moving to shared instances resulted in $<$3% degradation in BoM-err across all scenarios (detailed results in Appendix D.2).

6.3 Deep Dive on high BBC prediction errors

Our BBC leave-one-out tests required the ML models to learn the underlying *trend* in performance as a function of the parameter being varied. We observed that in many cases (with high BoM-err), this trend is inherently hard to predict because it changes too fast (i.e. the underlying function has a high Lipschitz constant). This causes our ML models to generalize poorly [24, 42]. We highlight this phenomena for three applications here, and discuss some of the others in Appendix D.4.

(i) KMeans has a high average BoM-err of 40.4% and 30.4% for prediction across varying input-scale with identical inputs and non-identical inputs respectively. We observed that the

[8] Note that Fig. 3 captures a subset of prediction scenarios (i.e. across varied input scale). This is in contrast to Fig. 2c that captures a wider range of prediction scenarios, which explains any observed deviation from this general trend. Appendix D.3 shows the results for our leave-one-out test with only the predictions across varied input scale, for a more direct comparison.

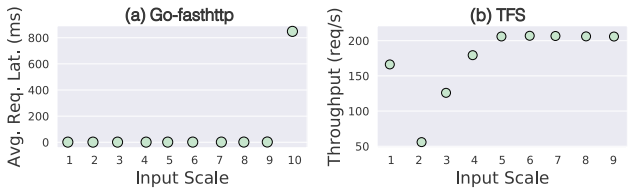


Figure 5: Average request latency of go-fasthttp and TFS at varying input scales. These functions vary too fast, and are hence inherently hard to predict.

performance of KMeans is sensitive to the content of its input dataset. At larger input sizes, Kmeans exhibits multi-modal behavior; specifically, the number of iterations to converge (and hence, the JCT) depends on the input data content (this effect is shown in Fig. 4(b)). This multi-modality impacts O-err and BoM-err in prediction tasks where input content is varied. With identical inputs, this multi-modality impacts the underlying trend as the input scale is varied, making it (surprisingly, even more) difficult to learn. The JCT for an input scale factor of 9 in Fig. 4(a) is an example of this behavior.

*(ii) Go-fasthttp has an exceptionally high average BoM-err of 128.6% for predictions across varying input-scale.*The mean query latency increases dramatically at the highest input scale because the system behavior changes under such high load due to queuing (Fig. 5(a)). This sudden change makes prediction on that value inherently difficult. The BoM-err for go-fasthttp reduces to 4.9% if the test datapoints for the highest input scale are removed from consideration.

(iii) TFS has a higher BoM-err of 52.7% for prediction across varying input-scale. This is again due to the underlying function being difficult to learn, as shown in Fig. 5(b). We are still investigating the root-cause behind this.

6.4 Summary

Application modifications to eliminate sources of irreducible errors do produce a notable increase in prediction accuracy, suggesting that a workflow where application developers provide knobs that give operators the option of disabling these error sources could be a promising direction moving forward. However, there are important concerns that cannot be neglected: (1) From the viewpoint of predictability, as we move to more realistic BBC scenarios, prediction errors do remain high for certain applications due to the underlying trend being difficult to learn (as illustrated in §6.3). Eliminating irreducible errors via application modification is not sufficient in these scenarios. (2) From the viewpoint of generality and ease-of-use, identifying the root causes of errors and making them configurable imposes a non-trivial burden on application developers; the same is true of asking system operators to reason about the trade-offs between predictability and other goals such as performance. In other words, such changes do weaken the black-box nature of performance prediction that we originally hoped ML-based predictors could provide.

7 Probabilistic Predictions

It may not always be possible to identify and eliminate sources of irreducible errors as required by the previous section. For instance, it might be too time-consuming to do so, or the system may be closed-source and not amenable to modifications. Or, operators may not want to compromise on the benefits of the relevant system optimizations (*e.g.,* experiencing slower garbage collection by disabling the GC optimization). Therefore, we now explore an approach that allows operators to *embrace*, rather than eliminate, performance variability.

Our empirical observations in §5 reveal that the optimizations causing irreducible error often lead to bimodal/multi-modal performance distributions. This is the key insight that drives our approach, leading us to hypothesize that a way forward could be to extend ML models to predict not just one performance value, but a probability distribution from which we derive k possible values, with the goal that the true value is one of the k predictions, for a low k.

This immediately raises the question of whether such top-k probabilistic predictions would even be useful to an operator? We believe they can be, depending on the use case. For instance, the operator can use the worst among the k predictions when provisioning to meet SLOs; or they can use the average expectations across the k predictions to pick an initial number of workers in a system that anyway dynamically adapts this number over time; or they can use the probability distribution to compare which system configuration will perform better in expectation when purchasing new servers. Note that, as in §6, this approach also comes with a trade-off – using the worst of k predictions may lead to over-provisioning, or using the expected average may lead to sub-optimal choices.

Our goal here is not to design such use cases of probabilistic predictions, or reason about these trade-offs. We instead focus on the following questions: *assuming* operators could make use of top-k probabilistic predictions: (i) how do we extend ML models to enable top-k predictions, and (ii) is there a small enough value of k that results in accurate top-k predictions? If not, exploring use cases of top-k predictions would be moot.

We thus proceed with discussing how we extended two of our models to predict probabilistic outputs (§7.1), and our prediction results (§7.2).

7.1 Extending ML models

We extend random forest and neural network to predict performance as a probability distribution across k possible outputs. We chose these models as they were most natural to extend.

Probabilistic Random Forests (Prob. RF). Instead of using the average JCT of the training points at the leaf node as the prediction (as is done in conventional decision trees), we use the *distribution* of the JCT data points at the leaf node, modeled using a Gaussian Mixture Model (GMM) [34] with k components. We train this Prob. RF as before, still picking splits that minimize the variance of JCT in child nodes.

Mixture Density Networks (MDNs). We adopt MDNs [25],

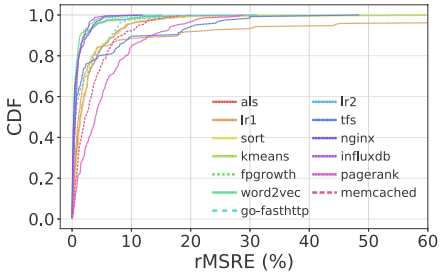


Figure 6: CDF of BoM-err in the BC test where models with prob. outputs are used (k=3). Compare to Fig. 1b.

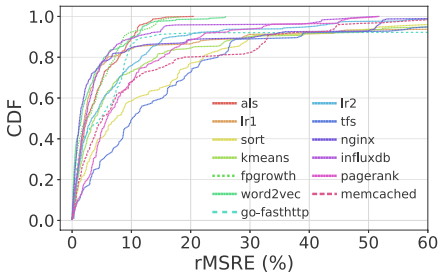


Figure 7: CDF of BoM-err in the BBC leave-one-out test where the models with probabilistic outputs are used (k=3). Compare to Fig 1c.

and modify our neural network to predict parameters for a Gaussian Mixture Model with k components (mean and variance for each component and mixing coefficients). We use negative log-likelihood of the data under the predicted GMM, as the loss function to train the MDN.

We implement Prob. RF based on random forest module in `scikit-learn`, and MDN in TensorFlow [21].

7.2 Top-k Prediction Results

To evaluate predictability with the probabilistic models above, we obtain k predictions from the models as the k different means of the k component GMM, and report the *top-k rMSRE* score, i.e. the rMSRE score of the best prediction among the k predictions made by the model. Such a top-k rMSRE shares the same interpretation as the rMSRE score – in fact, rMSRE scores reported so far can be thought of as top-1 rMSRE scores. In our evaluation, we observed a sharp drop in error rates as we move from a top-1 to top-2 measure and that further improvement plateaus off for $k > 3$. For brevity, we present a subset of our results for $k = 3$.

BC Test Fig.6 shows the top-3 BoM-err under the BC predictions. We see a significant decrease in BoM-err compared to our top-1 prediction presented earlier in Fig.1b. The 90%ile BoM-err is less than 10% for all but two applications. Note that the test data set remains unchanged by our use of probabilistic models, and hence O-err remains as high as in Fig.1a. The reduced BoM-err with top-k predictions shows the promise of this approach in embracing inherent variability.

BBC Test Fig.7 presents the top-3 BoM-err under the BBC test, where we relax the seen-configuration assumption by conducting leave-one-out predictions. While there is an improvement over models that make a single prediction (as in Fig 1c), we note that our multi-modal predictions don't improve performance in cases where the underlying trend is hard to predict for a reason other than multi-modality (*e.g.,* TFS and go-fasthttp).

7.3 Summary

Our findings along this direction are similar to those in §6. While it is possible to reduce prediction errors by extending our ML models to predict top-k performance values, two concerns with regard to generality and ease-of-use remain: (1) Even with top-k predictors, we continue to see scenarios with high error rates when we consider our more realistic BBC tests because the underlying performance trend is difficult to learn. Thus achieving a fully general predictor remains out of reach. (2) The use of top-k predictors complicates the process of applying performance prediction (which compromises ease-of-use) and may lead to sub-optimal decisions; in fact, how to best use such predictions and reason about the resulting trade-offs remains an open question.

8 Takeaways and Next Steps

We set out to evaluate whether ML offers a simpler, more general approach to performance prediction. We showed that: **(1)** Taken "out of the box", many of the applications we studied exhibit a surprisingly high degree of irreducible error, which fundamentally limits the accuracy that *any* ML-based predictor can achieve, and **(2)** We *can* significantly improve accuracy if we relax our goals (accepting the trade offs) and modify applications and/or modify how we use predictions. But even with these relaxations we still see a non-trivial number of predictions with high error rates! E.g., apps where $\sim$10% of the BBC tests have BoM-err $>$ 30-40%.

While ML fails to meet our goal of generality, we did find several scenarios where ML-based prediction *was* effective, showing that we must apply ML in a more nuanced manner by first identifying whether/when ML-based prediction is effective. Our methodology provides a **blueprint** for this, as summarized in Figure 8. Concretely, say that operators want to assess and improve the predictability of a target application. The first step is to run our BC test with the target workloads. If O-err is low, they can continue to the BBC test and check BoM-err. Otherwise, they have two options. The first is to disable any root-causes of variability if possible, rebuild the application, and re-run the BC test. If disabling root-causes is not possible or not desirable, operators can choose to use the top-k predictions. They can also combine both options, reconfiguring the application and using probabilistic predictions.

Even for this more nuanced approach, many open questions remain: (i) do our findings hold beyond the 13 apps and 6 models studied? (ii) how do we design systems to more easily identify sources of irreducible error? (iii) how can developers and operators more easily reason about trade offs between predictability and other design goals? Etc.

Our work provides empirical evidence motivating the above

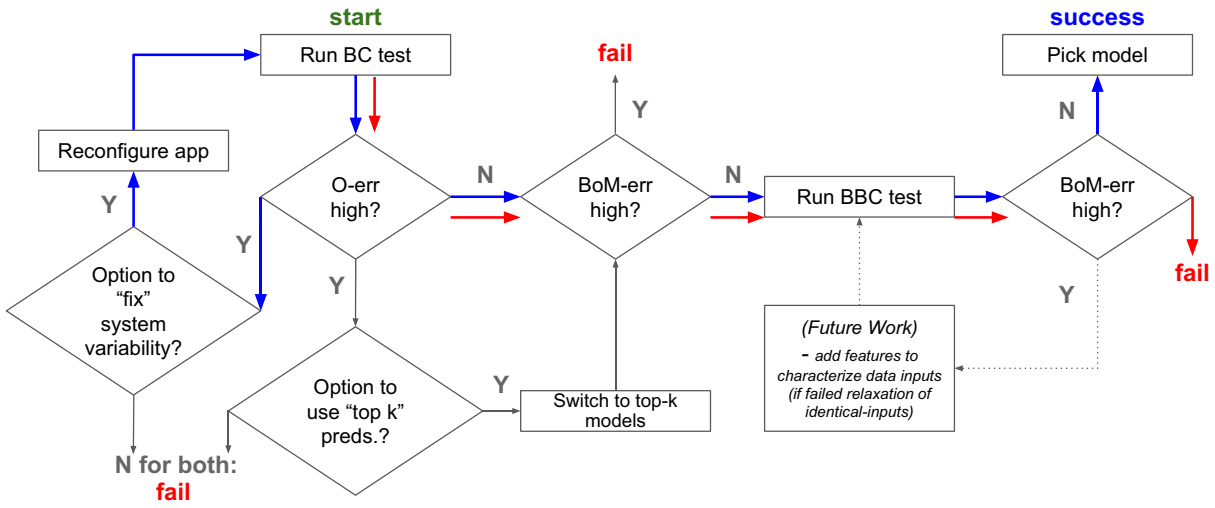


Figure 8: Methodology blueprint. As examples from our findings: Terasort's path (shown in blue) started out with only 73% of prediction tests having an O-err <10% in our BC test but once we eliminated its root-cause of irreducible error (the worker-readiness optimization), 99+% of test cases saw an O-err <10% and the BoM-err even in our BBC test was <10% for ∼90% of test cases leading to what we would deem a successful outcome. By contrast, KMeans (shown by the red path) started with a good O-err in our BC test (>90% of test cases had O-err <10%) but ultimately failed when we relaxed our identical-inputs assumption where point >60% of prediction tests had O-err >20%!

questions and a blueprint for how to approach and evaluate them. Overall, we remain cautiously skeptical about the role of ML in predicting system performance. We note that a common thread in the above questions is the need to evaluate predictors in a manner that is systematic and *consistent across studies*. We call on the community to adopt and extend our methodology as the foundation for such evaluation.

9 Related Work

Prior work has explored using ML based performance prediction for tuning and optimizing system configuration. Ernest [57] uses domain expert knowledge to build an analytical model for Spark performance, that is based on transformations and combinations of different features (such as number of cloud instances and input dataset scale), and trains the parameters of this model using ML.

Similar data-driven, gray-box modeling approaches have been applied to predicting and tuning performance for deep learning workloads and scientific computing [47, 48]. Paris [60] is a black-box performance modeling tool for selecting the best instance type by training a Random Forest model for each instance, and profiling unseen test applications on a small subset of instances to feed as input to the model.

Selecta [37] makes innovative use of collaborative filtering to predict performance and select the best-performing storage configuration for data analytics applications. CherryPick [22] explores black-box optimization (Bayesian Optimization) for a guided search towards the optimal cloud configurations without accurately predicting performance. Google's Vizier [31] leverages similar black-box optimization and makes it an internal application service for various workloads. Each of the above focus on answering a specific question with a specific ML technique. Our goal is to understand how ML can be more broadly applied to predicting performance across a range of systems and predictive tasks. We hope that our results, particularly as they relate to our methodology and irreducible error, can be applied to many of the contexts explored in prior work. Monotasks [44] proposes a radically new design for Spark aiming at assisting performance diagnostics and prediction; it does not explore the role of ML for performance prediction.

Similar to our proposal in §7.1, several recent papers recognize that performance is perhaps better represented as a probability distribution. [51] proposes modeling the performance of individual functions/methods as probability distributions and presents automatic instrumentation to capture such distributions. [45] shows how to design cluster schedulers that take as input, distributions derived from historical performance. Our proposal adds a new dimension to this general approach: we show how to extend ML models to generate probabilistic performance predictions.

Google Wide Profiler [36, 50] explores the use of performance counter information collected on an always-on profiling infrastructure in datacenters to provide performance insights and drive job scheduling decisions. As mentioned in §2, our work differs in that we focus on static parameters to enable use-cases where predictions happen before runtime.

Performance variability has been widely reported in contexts from hardware-induced variability [41], to stragglers in batch analytics [59], variability in VM network performance [52], and tail request latencies in microservices [29]. Our work can be thought as complementary: we studied a wide range of applications, report variability even under best-case scenarios, focus on the impact of variability to ML-based performance prediction, and propose systematic approaches to cope with variability.

References

[1] Amazon EC2 dedicated instances. https://aws.amazon.com/ec2/pricing/dedicated-instances/.

[2] Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types/.

[3] Apache spark. https://spark.apache.org/.

[4] Compute optimized instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compute-optimized-instances.html.

[5] A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[6] Datasets used in this paper. https://s3.console.aws.amazon.com/s3/buckets/perfd-data.

[7] Fast HTTP package for Go. https://github.com/valyala/fasthttp.

[8] Graphx synthbenchmark. https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/graphx/SynthBenchmark.scala/.

[9] High performance load balancer, web server, & reverse proxy. https://www.nginx.com/.

[10] An influxdb benchmarking tool. https://github.com/influxdata/inch.

[11] Kubernetes: Production-grade container orchestration. https://kubernetes.io/.

[12] memcached - a distributed memory object caching system. https://memcached.org/.

[13] Mutilate: high-performance memcached load generator. https://github.com/leverich/mutilate.

[14] Placement groups. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

[15] Scalable datastore for metrics, events, and real-time analytics. https://github.com/influxdata/influxdb.

[16] Spark performance tests. https://github.com/databricks/spark-sql-perf.

[17] Tensorflow serving. https://github.com/tensorflow/serving.

[18] Tensorflow serving on kubernetes. https://www.tensorflow.org/tfx/serving/serving_kubernetes.

[19] Testing the performance of nginx and nginx plus web servers. https://tinyurl.com/yccm2uf6.

[20] Tooling and setup used in this paper. https://github.com/perfd/perfd.

[21] Martın Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*, 1(2), 2015.

[22] Omid Alipourfard et al. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. USENIX NSDI*, 2017.

[23] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. In *Proc. ACM PPoPP*, pages 213–223, 1991.

[24] Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. In *Advances in Neural Information Processing Systems*, pages 6240–6249, 2017.

[25] Christopher M Bishop. Mixture density networks. 1994.

[26] Jianhua Cao, Mikael Andersson, Christian Nyberg, and Maria Kihl. Web server performance modeling using an m/g/1/k* ps queue. In *10th International Conference on Telecommunications, 2003. ICT 2003.*, volume 2, pages 1501–1506. IEEE, 2003.

[27] Harshal S Chhaya and Sanjay Gupta. Performance modeling of asynchronous data transfer methods of ieee 802.11 mac protocol. *Wireless networks*, 3(3):217–234, 1997.

[28] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. Capacity and trainability in recurrent neural networks. In *ICLR*, 2017.

[29] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[30] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proc. ACM ASPLOS*, 2014.

[31] Daniel Golovin et al. Google vizier: A service for black-box optimization. In *Proc. ACM KDD*, 2017.

[32] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016.

[33] Xinlei Han, Raymond Schooley, Delvin Mackenzie, Olaf David, and Wes J Lloyd. Characterizing public cloud resource contention to support virtual machine co-residency prediction. In *Proc. IEEE IC2E*, 2020.

[34] Trevor Hastie, Robert Tibshirani, Jerome Friedman, and James Franklin. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):83–85, 2005.

[35] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[36] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proc. ACM ISCA*, 2015.

[37] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: heterogeneous cloud storage configuration for data analytics. In *Proc. USENIX ATC*, 2018.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[39] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proc. ACM EuroSys*, 2014.

[40] Amiya K Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. Mitigating interference in cloud services by middleware reconfiguration. In *Proc. of the 15th International Middleware Conference*, 2014.

[41] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *Proc. USENIX OSDI)*, 2018.

[42] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[43] Matthias Nicola and Matthias Jarke. Performance modeling of distributed and replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 2000.

[44] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proc. ACM SOSP*, 2017.

[45] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A Kozuch, and Gregory R Ganger. 3sigma: distribution-based cluster scheduling for runtime uncertainty. In *Proc. ACM EuroSys*, 2018.

[46] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[47] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proc. ACM EuroSys*, 2018.

[48] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *Proc. USENIX NSDI*, 2019.

[49] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *Proc. IEEE Cloud*, 2010.

[50] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, 2010.

[51] Daniele Rogora, Antonio Carzaniga, Amer Diwan, Matthias Hauswirth, and Robert Soulé. Analyzing system performance with probabilistic performance annotations. In *Proc. ACM EuroSys*, 2020.

[52] Ryan Shea, Feng Wang, Haiyang Wang, and Jiangchuan Liu. A deep investigation into network performance in virtual machine based cloud environments. In *Proc. IEEE INFOCOM*, 2014.

[53] Harish Sukhwani, José M Martínez, Xiaolin Chang, Kishor S Trivedi, and Andy Rindos. Performance modeling of pbft consensus process for permissioned blockchain network (hyperledger fabric). In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 253–255. IEEE, 2017.

[54] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[55] Omesh Tickoo, Ravi Iyer, Ramesh Illikkal, and Don Newell. Modeling virtual machine performance: challenges and approaches. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):55–60, 2010.

[56] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proc. ACM SIGMOD*, 2017.

[57] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *Proc. USENIX NSDI*, 2011.

[58] Yangyang Wu and Ming Zhao. Performance modeling of virtual machine live migration. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 492–499. IEEE, 2011.

[59] Neeraja J Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proc. ACM SoCC*, 2014.

[60] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. Selecting the best vm across multiple public clouds: a data-driven performance modeling approach. In *Proc. ACM SoCC*, 2017.

[61] Ji Zhang et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proc. ACM SIGMOD*, 2019.

[62] Xiaolan Zhang, Giovanni Neglia, Jim Kurose, and Don Towsley. Performance modeling of epidemic routing. *Computer Networks*, 51(10):2867–2891, 2007.

Appendix

A Test Applications: Additional Detail

We include **Memcached**, a popular in-memory key-value store. We use the mutilate [13, 39] memcached load generator which generates realistic memcached request streams and records fine-grained latency measurements. We vary the key size (corresponding to the varying input scale prediction), number of memcached servers (varying number of servers), and the server instance type, while keeping the other workload parameters in mutilate as fixed values[9]. We report results on predicting the average query latency.

We include **Influxdb**, a widely used timeseries database. We use its official benchmarking tool Inch as the client. Inch sends write queries to Influxdb and reports the time it takes to complete each write query. We vary the number of points per timeseries written per query (varying input scale prediction), the number of Influxdb servers, and the server instance type. We report results on predicting the query latency.

We include **nginx**. We use wrk2, an http benchmarking tool used by nginx's official benchmark reports [19]. It sends http requests and reports fine-grain latency and throughput measurements. We vary the per-client request rate, number of worker processes (each worker process is a single-threaded process; the maximum number of worker processes we vary to is equal to the number of cores of the instance), and the server instance type. We report results on predicting the median request latency.

We include **go-fasthttp**, a high-performance HTTP package for building REST services. We use the wrk2 tool to generate HTTP loads again. We vary the request rate, number of server instances, and the server instance type. We report results on predicting the median request latency.

We include multiple **Spark-based applications** spanning diverse forms of computation: sorting, graph computation, classification (two different implementations), data mining, recommendation etc.

We include **Tensorflow Serving (TFS)**, a high performance model serving system, which we orchestrate using Kubernetes (k8s). Our TFS setup resembles that of other (e.g., Web) serving systems in which a front-end load balancer (we use EC2 ELB) spreads client requests across a backend of (model) serving instances.

B Understanding Irreducible Errors – Details

B.1 Spark's "Worker-readiness" Optimization

Details included in the main text.

B.2 Multi-mode Optimization in JVM GC

We elaborate on the details of how the JVM adapative GC leads to variable application performance in the context of our Spark Logistic Regression (LR1) workload.

[9] See https://github.com/perfd/perfd/blob/master/apps/memcached/manifests/perf_predict/suite_1/keySize.yaml for an example setup.

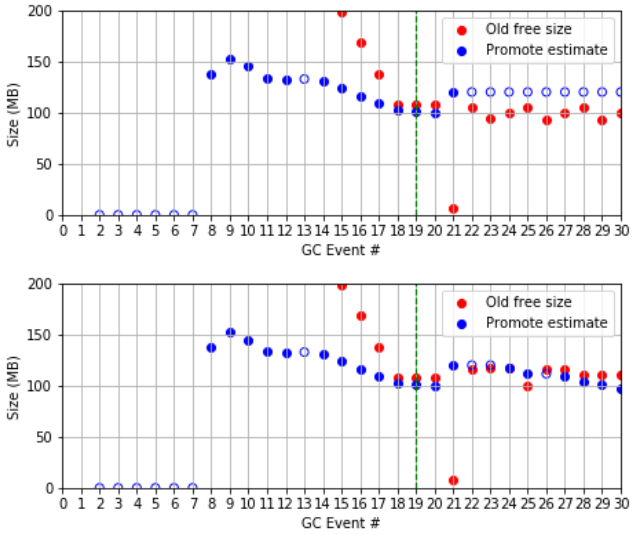


Figure 9: High-mode (top) and Low-mode (bottom) trajectories for promotion estimate and free space in old region during the first 30 garbage collections. Hollow blue dots depict major/full GCs and solid blue dots depict minor GCs.

§5.1.2 revealed a positive correlation between the number of *full* GCs (explained below) and JCT with a distinct bimodal behaviour. We now describe how this bimodality arises. For this, we first explain relevant aspects of the GC mechanism in Java Virtual Machine (JVM) that Spark uses.

JVM divides the Java memory heap into two regions – a *young* region to allocate new objects, and an *old* region to accommodate 'old' objects that have survived multiple GC rounds. It supports three different types of GCs over these regions: (i) *minor GC* on the young region, (ii) *major GC* on the old region, and (iii) *full GC* over the entire heap space (both young and old regions), with the surviving objects residing in the old region.

In the face of heap space shortage, JVM first runs a minor GC on the young region, deleting the unused objects and promoting the old objects (that have survived multiple GC rounds) to the old region. A minor GC triggers a major GC if the old region has too little free space to hold the promoted objects. If a minor GC constantly triggers a major GC, the garbage collector can save time by skipping the minor GC (and the ensuing major GC), and directly running full GC over the entire space. JVM implements this adaptive skipping of minor GC as a performance optimization. To estimate whether a minor GC would trigger a major GC, it maintains a *promotion estimate*, calculated as the moving average of the number of promoted objects after each round of minor GC. If the promotion estimate is higher than the amount of free space in the old region, JVM GC skips the minor GC and runs a full GC directly.

We now show how this adaptive skipping behavior impacts performance predictability. We randomly pick one sample

each from the slower "high-mode" runs (whose JCTs sit at the top-right corner of the figure in §5.1.2) and the faster "low-mode" runs (whose JCTs sits in the bottom-left corner of the figure in §5.1.2). The top and bottom plots in Fig.9 show the promotion estimate values and the amount of free space in the old region observed for the first 30 GC events in the two sample runs respectively.

Fig.9(top) reveals that beyond GC event #21, the promotion estimate in the high-mode run remains greater than the amount of free space in the old region (even after event 30, beyond what the plot shows). This results in consecutive full GCs (shown as hollow blue dots). Note that since no objects get promoted in a full GC, the JVM does not update the promotion estimate.

Fig.9(bottom) shows how the low-mode run escapes from such a fate: the promotion estimate is higher than the amount of free old space only for a few GC events, and stays lower than that beyond GC event #26. Consequently, the low mode run sees more minor GCs than full GCs (shown as solid blue dots). Since a full GC, which scans objects across a larger memory space, is significantly more time consuming than a minor GC, the perpetuation of full GCs has a large impact JCT (witnessed in the scatter-plot in §5.1.2).

Fig.10 shows the corresponding time-series of GC events along with events that mark the start and finish of individual tasks (for clarity, we only show the time range of 20-26s, which captures the regime where the two runs start deviating from one another). Careful observation of Fig.10 reveals that the GC behaviour of the two runs begins to diverge around time 24s, when two of tasks finish slightly earlier in the high mode run than in the low mode run. We checked that around this time, the two modes see a difference of about 10MB in the amount of free old space (which is rather small compared to the total heap size of 1GB).

Such subtle differences cannot be determined without knowing the runtime state of the garbage collection and the Spark application.

B.3 Non-determinism in the Spark Scheduler

Our analysis revealed that PageRank's high O-err stems from non-determinism in the Spark scheduler. Specifically, in a job with multiple stages, a stage A may be *independent* of stage B in the sense that A's tasks can be scheduled to run whether or not B's tasks have completed. Our traces showed that independent stages were being scheduled in different orders in different runs leading to different JCTs. This arises because of how the Spark scheduler tracks dependencies between stages. Spark's scheduler maintains a graph that captures the dependencies between stages; a stage v's parents in the graph are those stages that can only be scheduled after v is complete. Spark's scheduler uses the Scala `HashSet` data structure to track the parents of a stage. When the search propagates to the parents, the order of visiting these parents can be inconsistent across runs because `HashSet` makes no guarantees on the order of iteration through the set members.

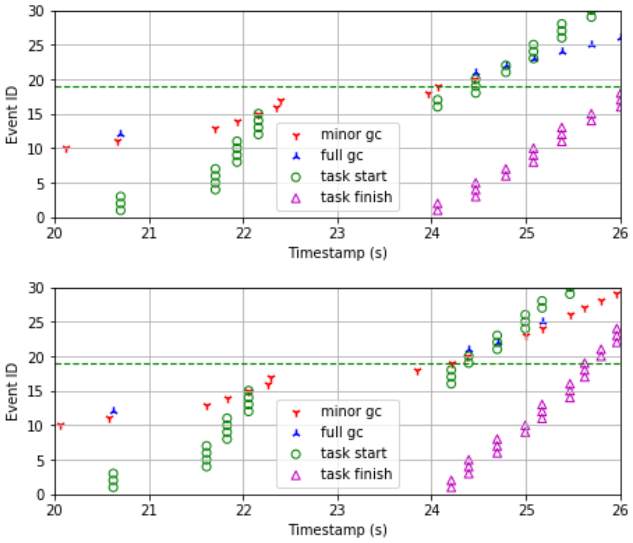


Figure 10: High-mode (top) and Low-mode (bottom) trajectories for the run-time events. Note that unlike Fig.9, the x-axis unit here is the wall time.

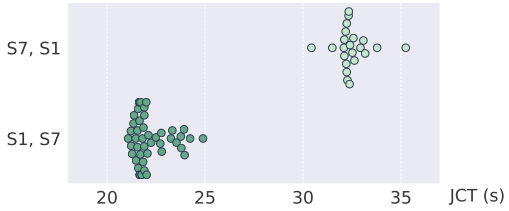


Figure 11: Bimodal performance that results from scheduling independent stages in different orders.

Fig.11 shows the effect of this non-determinism. Stages 1 and 7 are two independent stages and the figure plots the JCT for runs where stage 7 is scheduled before stage 1 (dots on the bottom-left) and vice versa (dots on the top-right). We see that the difference in scheduling order corresponds to bimodality in the JCTs. We emphasize that this bimodality could not be predicted prior to runtime as it depends on the runtime state of the HashSet structure rather than any static input parameters.

Setup	Naive-err	BoM-err	O-err
Baseline	44.7%	19.2%	18.2%
+Sched. mod.	40.7%	5.7%	3.2%

Table 3: Prediction error before vs. after.

To verify our analysis, we modify the Spark scheduler such that the order of scheduled stages is consistent across runs with identical configurations (replacing HashSet with a LinkedHashSet). Table 3 shows that our modification (the "+Sched. mod." row) reduces the O-err by more than 5.7x bringing the error to under 4%.

B.4 HTTP Redirection and DNS Caching in S3

Multiple applications in our study – KMeans, LR2, FPGrowth, and ALS – experienced irreducible errors due to name resolution in Amazon's S3 storage service. We explain this effect in the context of the KMeans application.

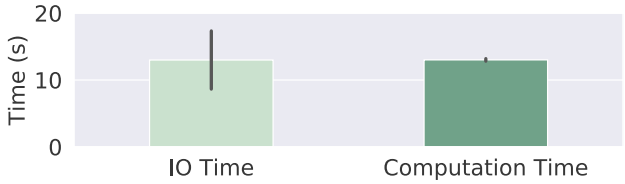


Figure 12: Breakdowns of Spark KMeans job completion time as time spent on cloud storage IO (writing the results to S3) and on computation for 100 experiment runs.

When triaging the overall JCT, we often found that, on average, the computation time was on par with the S3 IO time but that the standard deviation for the latter was 24× times higher than the former (Fig.12). Analysis of the application's runtime logs for KMeans revealed that the *variance* in performance stems from I/O to Amazon's S3 storage service. This is shown in Fig.12 which breaks down the overall JCT into computation time and S3 IO time. We see that the computation time has little variance while the S3 IO time has substantial variance (between 17.5s and 9s).

Further instrumentation revealed a correlation between the time spent on S3 IO and the number of HTTP redirection events, and that the latter varied across multiple runs of identical test configurations.

S3 is a distributed storage service in which objects are spread across multiple datacenters. When a new object "bucket" is created, its DNS entry points to a default datacenter location. If a user creates a bucket in a datacenter other than the default one, then a request to that bucket is first sent to the default server, which responds with an HTTP redirect containing a new URL that resolves to the datacenter where the bucket is located.

Such redirection continues until the DNS entry for the bucket is correctly updated.[10] The S3 buckets in our experiments were created in a different datacenter from the default one, leading to such HTTP redirects.

Fig.13 illustrates the above behavior during one of our experiments. We plot two values over time: (a) the number of distinct S3 servers that our application connected to, and, (b) the time spent on S3 IO. We observe three distinct phases to S3 performance, demarcated by green lines in Fig.13.

Interestingly, we observe a significant intermediate period where the two values oscillate. We found that this oscillation arises because AWS load-balances DNS requests across a pool of DNS servers and the servers in this pool converge to the new DNS entry at different times. In summary, we see

[10] AWS states that DNS entries can take up to 24 hours to be fully propagated; we observed average delays of approx 4 hours. AWS also recommends that clients not cache the redirect URL as its validity is only temporary.

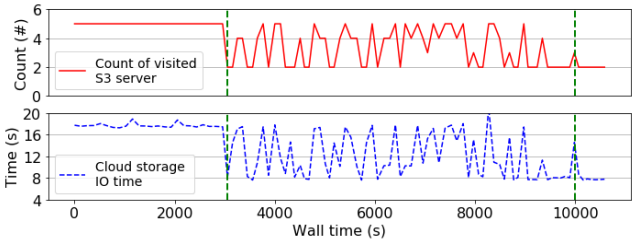


Figure 13: The number of S3 servers visited during the cloud storage IO (top) and the total IO time (bottom). This is plotted over wall-clock time during multiple rounds of experiments.

variable performance even for identical test configurations due to the distributed and eventually-consistent nature of object name resolution in S3. To validate our analysis, we repeated our experiments after modifying the KMeans application to cache the correct bucket location after the first redirection event (yes, ignoring AWS' recommendation). Table 4 shows that this modification dramatically reduces prediction error to an O-err of 1.0% and a BoM-err of 1.1%.

Setup	Naive-err	BoM-err	O-err
Baseline	22.7%	16.0%	15.2%
Correct bucket loc.	5.6%	1.1%	1.0%

Table 4: Prediction error before vs. after.

One might ask whether it is possible to predict the impact of AWS's DNS service on applications. An in-depth exploration of this question is beyond the scope of this paper. However, we note that doing so appears impractical, if not infeasible, since we would have to know how an application's lifetime overlaps with the DNS timers not just for a single server but for an entire pool of servers, as well as precisely knowing how DNS requests are load-balanced across this pool.

B.5 Imperfect load-balancing at high load

We observed high irreducible errors when predicting the request throughput provided by a TFS cluster under increasing numbers of workers (servers). Our analysis revealed that this high error stems from how client requests are load-balanced across TFS servers. We run TFS servers within a Kubernetes cluster and hence incoming client requests undergo two levels of load balancing. First, the AWS Elastic Load Balancer round-robins incoming client connections across the k8s nodes. Next, each k8s node load balances incoming RPCs across the TFS instances.

We found that the irreducible errors in TFS stem from the second stage of load-balancing. The load-balancing within k8s is based on selecting a TFS server at random, leading to some inherent imbalance in the load at each server. When running the TFS cluster at high utilization (as our experiments do), this imbalance means that some servers are already running at capacity while others are running below the maximum request rate they can sustain. The *variability in this imbalance*

leads to variations in the overall request throughput; e.g., in repeated runs of one identical test setup, we saw aggregate throughput vary between 80-140 req/sec.

Setup	Naive-err	BoM-err	O-err
Baseline tput.	123.5%	26.7%	11.8%
-Rnd LB tput.	163.2%	7.6%	6.7%

Table 5: Prediction error before vs. after.

To verify this effect, we reconfigure the k8s load-balancer to always direct client RPCs to the local TFS server instance. In effect, this disables the k8s load-balancer which is acceptable for our test purposes. Table 5 shows that this modification substantially reduces the prediction error. We found a similar effect when predicting request latency and also found that incorporating heavy-hitter clients exacerbated the error due to randomized load-balancing.

B.6 Variability in Implementations of Cloud APIs

We observed high O-err in the memcached and ngnix applications, both stemming from variability in how the cluster is configured and the limited control/visibility that default cloud APIs provide for this process.

In the case of memcached, the variability stemmed from how our worker instances were being placed within the cloud infrastructure. Our experiments used EC2's default VM placement which offers no guarantee on the proximity between our allocated instances and hence our node-to-node latency varied across runs. This in turn led to variable memcached read latencies, as we found memcached read times are dominated by the network latency between the client and server in our setup. We found that switching to an API that consistently places a set of instances close together reduced the O-err from 36.9% to 2.4% (varying input scale) [14].

For nginx, the high O-err stemmed from variability in the default network bandwidth associated with smaller instance types. Specifically, with EC2's "c5" instances types, those smaller than c5.9xlarge are by default assigned "up to" 10Gbps network bandwidth while c5.9xlarge instances are assigned exactly 10Gbps. For smaller instance types we found that EC2 *occasionally* throttles network bandwidth and the degree of throttling varies across runs (see Appendix B.6). The response time in nginx is also dominated by the network latency between client and server and hence this variability in throttling leads to unpredictable request latencies. Repeating our scale-out and input-scaling tests with a c5.9xlarge instance (instead of our previous default of c5.xlarge for the client and c5.4xlarge for the server) reduced the O-err from 15.6% to 2.0% under varying number of workers.

Fig.20 depicts the varying available bandwidth across different runs of the same configuration (specifically, the number of worker processes in varying number of worker process scenario). We measured the node-to-node bandwidth using iperf between the instance running the client (wrk2) and the instance running the nginx right before the experiments were run. The results show that despite the configurations (instance types, number of worker processes) remaining the same, the available bandwidth can vary. We conjecture this is a result of the bandwidth allocation mechanism AWS employs, which throttles bandwidth usage based on an estimation of the average network utilization of the instances [4].

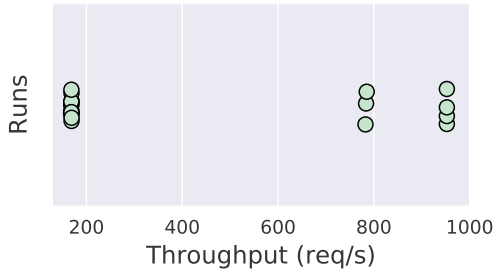


Figure 14: Available bandwidth (measured with `iperf`) between client and server nodes across different runs of the same configuration.

Note that it is arguable whether the above sources of error are "irreducible". On the one hand, it might be possible to augment cloud APIs or incorporate the parameters of cloud APIs as a feature in our ML models, however, doing so is likely to come at a loss in flexibility and efficiency for cloud operators. E.g., one might envisage placement APIs that guarantee *consistent* proximity between instances, but the achievable proximity is likely to vary depending on the number and type of instances (we will see an example of this in §**??**). Similarly, if the AWS throttling that we observed is determined by current network load,[11] then it is not clear how one might capture the impact of throttling prior to runtime. We leave the question of how cloud providers might augment their APIs to aid performance predictability as a topic for future work.

C Best-Case Analysis - More Details

Fig.15 and Fig.16 describe the per-app average prediction error rates for best-case experiments.

D Beyond Best-Case Analysis - More Details

D.1 Leave-one-out - More Details

Fig.17 and Fig.18 describe the per-app average prediction error rates for leave-one-out experiments.

D.2 Dedicated Instance vs. Shared Instance

Fig.20 shows the impact of changing dedicated instance type to shared instance type for the Memcached and Spark sorting applications. We see that overall the relaxation does not have a statistically meaningful impact on the predictability of these two applications.

[11] AWS documentation does not elaborate on the exact conditions under which they implement throttling [4]

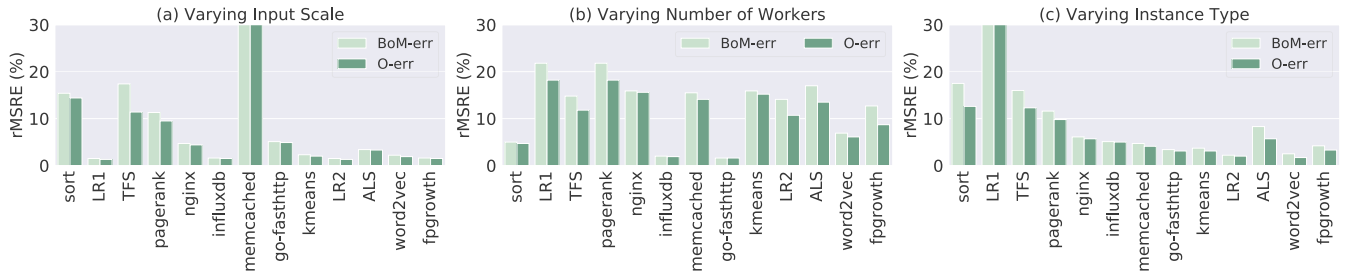


Figure 15: Best-case prediction errors for different applications under (a) changing input-scale, (b) scaling-out the number of workers, and (c) scaling-up the instances; before system modifications.

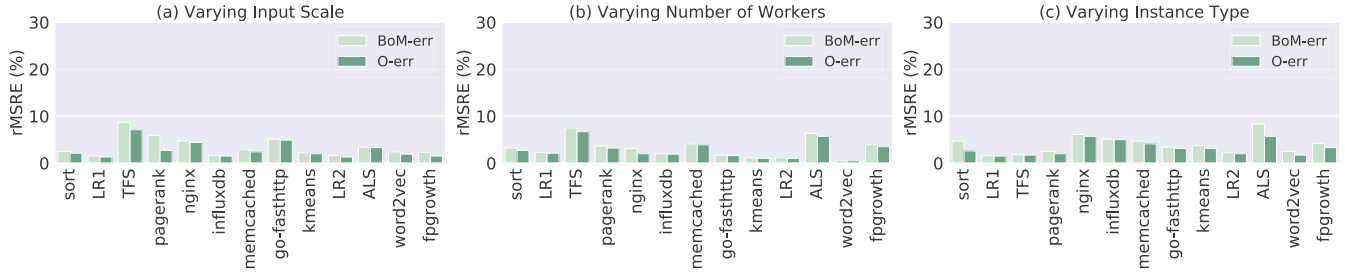


Figure 16: Best-case prediction errors for different applications under (a) changing input-scale, (b) scaling-out the number of workers, and (c) scaling-up the instances; after system modifications.

D.3 Leave-one-out Test with only Varied Input Scale

Fig.21 shows the CDFs of BoM-err in the BBC leave-one-out test, for only the predicitons across varying input data scale.

D.4 High BoM-err in BBC after App. Modifications - Other Applications

(i) TFS has a higher BoM-err of 52.7% for prediction across varying input-scale. This proved to be because the trend in TFS throughput under varying input scale is inherently hard to predict. The underlying function – shown in Fig.5(a) – has a high Lipschitz constant (*i.e.,* it changes too fast), which causes the trained model to have poor generalization [24, 42]. We have not yet been able to determine the root cause for this odd trend in TFS.

(ii) Memcached has a high BoM-err of 50.9% for prediction across scaling up instance types. Recall from §5 that memcached performance is sensitive to the node-to-node latency and hence we modified our experiments to use AWS's cluster placement group API that ensures nodes in a cluster are consistently placed close to each other. This avoids variability in placement across multiple runs of an identical test configuration. However, what we discovered is that while AWS's placement API is consistent in the proximity at which it places instances of a particular type, this proximity may vary across *different* instance types, making it hard for our ML models to learn a trend across instance types (as is needed for our leave-one-out analysis).

E Comparing ML Models

So far, we focused on the error of the best-performing ML model for a given task. We now compare across our six ML models. We do so in the context of our leave-one-out analysis since it is both more challenging and realistic. Table 6(top) reports the error rate that each model achieves for each of the three prediction scenarios, averaged across all applications. We normalize each of these error rates by the lowest average error for the corresponding prediction scenario. Therefore, a normalized value of 1.0 indicates that the corresponding model performed the best (on average) for the corresponding prediction scenario. Table 6(bottom) similarly reports the normalized error rates for each model and each application, but now averaged across all prediction tasks.

It is important to note that our analysis uses only out-of-the-box versions of each ML model and applies the same hyper-parameter tuning approach to each of them. This is in contrast to many prior studies (in systems contexts and beyond) in which a particular model is often specialized and carefully tuned to achieve high accuracy for a given prediction task [28, 32, 60]. In this sense, our results can be viewed as a "fair" comparison of models while at the same time we acknowledge that the performance of any individual model/prediction could probably be improved through careful tuning. We view our approach as establishing a useful baseline and conjecture that there is value to a low-effort ML predictor, especially given the rapid pace of evolution in modern software services.

We draw the following conclusions from our results:

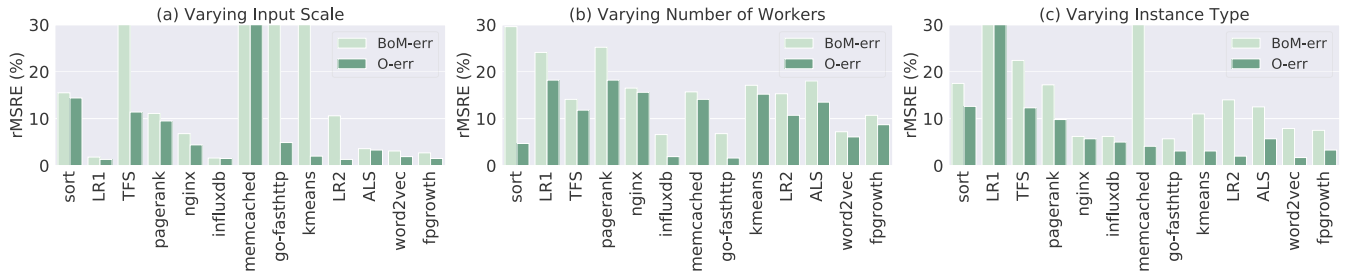


Figure 17: Leave-one-out prediction errors for different applications under (a) changing input-scale, (b) scaling-out the number of workers, and (c) scaling-up the instances; before system modifications.

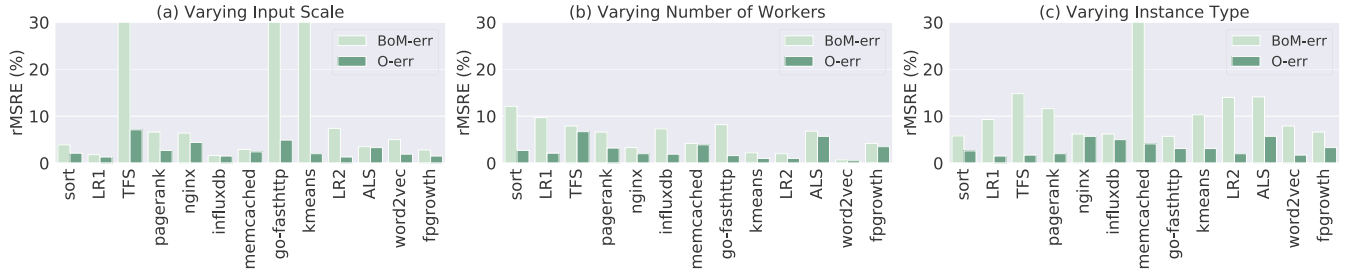


Figure 18: Leave-one-out prediction errors for different applications under (a) changing input-scale, (b) scaling-out the number of workers, and (c) scaling-up the instances; after system modifications.

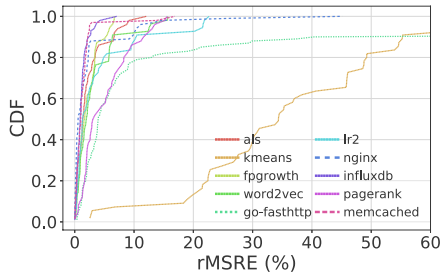


Figure 19: CDF of BoM-err in the BBC leave-one-out test where input data content is identical, for predictions across varying input data scale

(i) The key takeaway from our results is that there is no clear winner: no ML model performs the best across all prediction scenarios and across all applications. This validates the merits of studying a range of ML models and applications so that we can understand how to best apply or combine various models.

(ii) There is no clear loser either: each model performs the best for at least one prediction scenario or application.

(iii) Even though there is no clear winner, neural network often results in the best performance or has error rates close to the best performing model. There were a few exceptions where neural network resulted in 2× higher error rates than the lowest error. We found that using a different hyper-parameter tuning methodology reduced the neural network error in these cases. This confirms the common wisdom that neural networks, while powerful, can be difficult to train and tune.

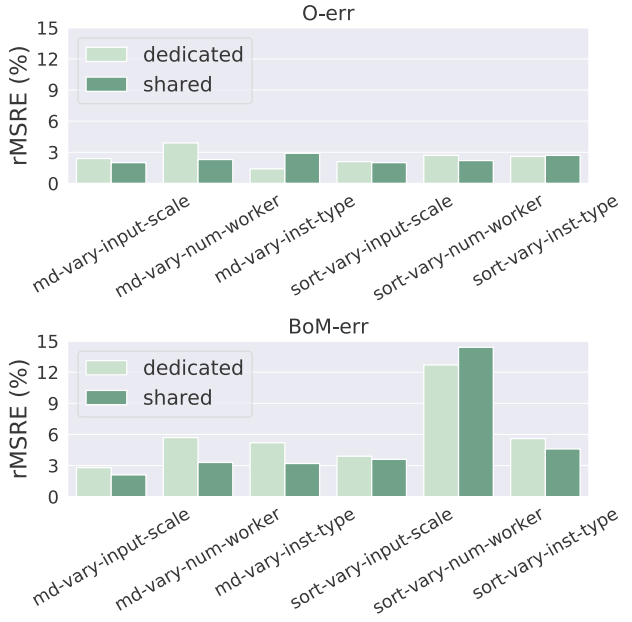


Figure 20: Memcached (md) and Spark sorting (sort) prediction accuracy in the three prediction scenarios with dedicated instances and shared instances.

(iv) For cases where neural network performed the best, there was often at least one other simpler model performed as well.

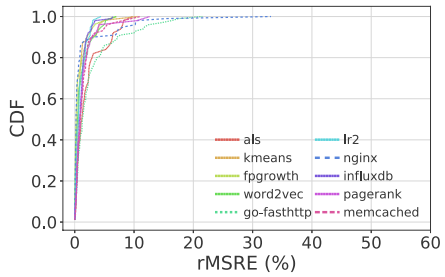


Figure 21: CDF of O-err in the BBC leave-one-out test where input data content is not identical, for predictions across varying input data scale

Prediction Type	Linear Regression	kNN	Random Forest	SVM	SVM kernelized	Neural Network
Input scale	**1.0**	3.5	3.5	**1.0**	2.4	**1.0**
# workers	2.0	2.0	2.1	1.6	1.4	**1.0**
Inst. type	4.0	1.2	**1.0**	1.4	1.4	1.6

Application	Linear Regression	kNN	Random Forest	SVM	SVM kernelized	Neural Network
sort	4.8	2.9	3.3	2.2	1.8	**1.0**
LR1	1.3	1.6	2.0	1.2	1.1	**1.0**
TFS	4.3	2.4	2.2	1.3	**1.0**	1.1
pagerank	2.5	1.7	1.8	1.8	3.4	**1.0**
nginx	6.6	2.6	1.1	1.1	**1.0**	1.7
influxdb	1.2	2.9	2.9	**1.0**	1.2	2.6
memcached	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	1.4
go-fasthttp	1.5	1.1	1.2	1.5	**1.0**	1.1
kmeans	1.3	1.4	**1.0**	1.3	1.6	1.2
LR2	2.3	2.6	2.2	2.4	**1.0**	1.9
word2vec	1.2	5.0	5.1	**1.0**	4.1	2.1
fpgrowth	6.0	2.8	2.7	**1.0**	2.0	1.1
ALS	1.7	1.2	1.1	1.3	2.5	**1.0**

Table 6: Comparison across ML models. The top table reports the error rates for each type of prediction scenario and ML model, averaged across all applications, and normalized by the lowest average error for that scenario. The bottom table similarly reports normalized error rates for each application, averaged across different prediction scenarios.