Toward Generalizable Models of I/O Throughput

Mihailo Isakov*, Eliakin del Rosario*, Sandeep Madireddy†,
Prasanna Balaprakash†, Philip Carns†, Robert B. Ross†, Michel A. Kinsy*
* *Adaptive and Secure Computing Systems (ASCS) Laboratory*
Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77843
{mihailo,eliakin.drosario,mkinsy}@tamu.edu
† *Argonne National Laboratory*, Lemont, IL 60439
smadireddy@anl.gov, {pbalapra,carns,rross}@mcs.anl.gov

***Abstract*—Many modern HPC applications do not make good use of the limited available I/O bandwidth. Developing an understanding of the I/O subsystem is a critical first step in order to better utilize an HPC system. While expert insight is indispensable, I/O experts are in rare supply. We seek to automate this effort by developing and interpreting models of I/O throughput. Such interpretations may be useful to both application developers who can use them to improve their codes and to facility operators who can use them to identify larger problems in an HPC system.**

The application of machine learning (ML) to HPC system analysis has been shown to be a promising direction. However, the direct application of ML methods to I/O throughput prediction often leads to brittle models with low extrapolative power. In this work, we set out to understand the reasons why common methods underperform on this specific problem domain, and how to build models that better generalize on unseen data. We show that commonly used cross-validation testing yields sets that are too similar, preventing us from detecting overfitting. We propose a method for generating test sets that encourages training-test set separation. Next we explore limits of I/O throughput prediction and show that we can estimate I/O contention noise by observing repeated runs of an application. Then we show that by using our new test sets, we can better discriminate different architectures of ML models in terms of how well they generalize.

***Index Terms*—High-Performance Computing, I/O Analysis, Machine Learning, Optimization.**

I. Introduction

Many modern high-performance computing (HPC) applications are extremely data-intensive, moving terabytes of data for tasks such as storing and retrieving data as a part of scientific workflows [1] or storing checkpoints to protect the results from possible hardware faults. These actions cause applications to be I/O bound. The I/O subsystem of modern HPC systems is complex, having multiple layers both on a physical (I/O forwarding nodes, network nodes, storage nodes, etc.), and on a logical (application data models, POSIX, etc.) level. This makes debugging I/O problems difficult. Small mistakes in HPC application implementation can reduce the I/O throughput of an application by several orders of magnitude.

This situation motivates us to explore how we can gain greater insight into the workings of I/O systems and into why applications achieve the I/O throughputs they do. Although HPC facilities have experts who can help developers optimize their applications, this is still a time-consuming process. In this work we build and use machine learning (ML) models of I/O throughput in order to better understand I/O throughput and interactions between the applications and the I/O subsystem.

Having an accurate model of an HPC system's I/O capabilities, as well as of the distribution of applications running on the system, is valuable for multiple reasons. These models may be useful in predicting a future job's I/O throughput and act as a sort of early-warning system for wasteful jobs. Alternatively, based on an application and a set of input parameters, an I/O model may be able to tell in advance how long the job will run. By knowing how an application interacts with system resources and other applications, we may better schedule that application. Such an I/O model may tell us if an application is particularly vulnerable to I/O contention or if it will disproportionately impact other jobs running on the system [2].

In this work we focus on another use case: using ML models to gain deeper insight into the system. In this scenario possessing an accurate model of I/O behavior is a goal unto itself. By treating a model as a black box, we can learn how it responds to different stimuli and how to better use the system. This analysis can be beneficial to system owners, who may learn how to better provision and procure new hardware, identify which applications are negatively impacting co-located applications, or determine whether a known optimization exists for a given application. Analyzing I/O models can also be of interest to users who can gain insight into why their application may not be performing as expected, find bottlenecks in their code or in the system, and learn what modifications will yield the largest impact.

In addition, we focus on enabling interpretation of ML models of HPC systems. ML model interpretation is a collection of methods that produces human-interpretable explanations of why a data-driven model has arrived at a certain output or how different input parameters affect a model's outcome. To arrive at sensible interpretations, we seek models that exhibit good extrapolative power, in other words, models that achieve reliable results on unseen data. However, direct application of ML interpretation methods to I/O throughput models does not always yield sensible results. This work was driven by empirical observations that while I/O models can have excellent predictive accuracy on a variety of applications, the interpretations gained from these models are still significantly less useful than the expertise and know-how of system owners

978-0-7381-1046-2/20/$31.00 ©2020 IEEE
DOI 10.1109/ROSS51935.2020.00010

and HPC practitioners. Therefore, we aim to understand the source of this discrepancy in order to build more interpretable models of I/O behavior.

To that end, our contributions are fourfold:

- We show that because of the typical distribution of user jobs in an HPC system, models trained on those jobs do not generalize well to unseen applications. This lack of generalization typically goes unnoticed because of the common methods for creating training and test splits.
- We propose a method for generating test sets that helps diagnose the lack of generalization in ML models. The method clusters the dataset using DBSCAN and holds out a subset of clusters at random.
- We explore the limits of prediction, where by looking at several runs of the same program with the same input we establish the variance of I/O throughput. We use this variance as the lowest possible prediction error achievable, and we show that this "noise floor" highly depends on the type of application.
- We perform a hyperparameter search in an effort to find configurations that work well on this domain. We develop and evaluate ML models that have an improvement in generalization over the baseline.

II. Related Work

Performance analysis and modeling of HPC applications remain a pressing concern because HPC applications often do not make good use of limited I/O resources. One key challenge stems from the fact that the variability of HPC applications has made the task of modeling and predicting their performance a nontrivial undertaking. In fact, several works have tried different strategies to predict or model the performance of these applications. Yang et el. [3] propose an observation-based performance prediction approach where partial execution of iterative programs is observed to predict performance across two applications. While effective, this approach is restricted to the system sizes used for the partial executions such that their model becomes less accurate when reusing partial execution results for a different problem. Madireddy et al. [4] proposed a sensitivity-based modeling approach that leverages application and file system parameters to find partitions in I/O performance data from benchmark application jobs and builds Gaussian process regression models for each partition to predict the I/O performance. In their follow-up work [5], the authors adopted a neural-network-based approach to build global models to predict I/O performance while considering the application and file-system parameters as categorical variables. Armstrong et el. [6], [7] developed a methodology that adopted the "Resource Usage Equations" to characterize application performance by their evolution trends based on system configurations and large dataset usage. Other work such as [8] used a combination of both analytical and experimental methods where the former method is used to capture deterministic factors while the latter produces implicit and dynamic factors.

III. Preliminaries

In this work we model I/O throughput using 89,844 Darshan logs that have an I/O volume larger than 100 MiB. These logs were collected at the Argonne Leadership Computing Facility (ALCF) Theta supercomputer in the period between April 2017 and May 2020. Darshan [9] is an HPC I/O characterization tool that collects I/O access patterns of jobs running on a system. While it supports multiple different APIs such as POSIX, MPI-IO, and STDIO, in this work we focus only on POSIX. Darshan instruments a job and collects aggregate values such as runtime, number of processes, read/write accesses, bytes read or written to shared or unique files, I/O access patterns per each file, and timestamps of first file open and close operations. We perform a significant amount of feature engineering on the collected data to allow easier ingestion by ML algorithms. Our pipeline summarizes each job using 52 features: 13 absolute-values features such as runtime, total data volume, total number of files, bytes, and accesses, as well as 39 relative (percentage) features such as read-to-write ratio, percentage of accesses of certain sizes, and consecutive accesses. An additional feature is I/O throughput, which is used as the prediction target and is not fed to the ML models. In this work we train models that use the 52 input features in order to predict the I/O throughput. In [10] we provide a deeper discussion on feature engineering Darshan logs.

IV. Diagnosing Lack of Generalization

Generalization is important, but it is not always achieved; and without a correct methodology, the lack of generalization may not be detected. To motivate this section, we give a recent example we encountered when applying ML-based prediction and model interpretation on new data from the ALCF Theta system. When using our *Gauge* tool [11]—an I/O throughput prediction and interpretation environment, we observed degraded model accuracy when working on very recent datasets (January 2020 and onward). Since the models were trained only with data in the April 6th, 2017, to January 1st, 2020, range, this degradation points to a lack of generalization capability. However, the errors witnessed were several times larger than the errors seen on our test sets. Since the primary goal of test sets is to estimate real-world errors and whether the model is overfitting, some salient dissimilarities must exist between the test set distribution and real-world data.

We illustrate this problem with a simple experiment. In the top row of Figure 1 we give an example of I/O throughput prediction errors on the data collected in the period from April 2017 to May 2020, with the ML model trained on data collected before 2020. The blue line represents test set errors where the test and training set belong to the same time range. The orange line represents errors on data collected after the model was already trained; in other words, this is "real-world data" and should accurately represent how the model behaves in production. Note that we display mean absolute errors on the y axis, where we report the average absolute ratio between the target and predicted value. For example, a

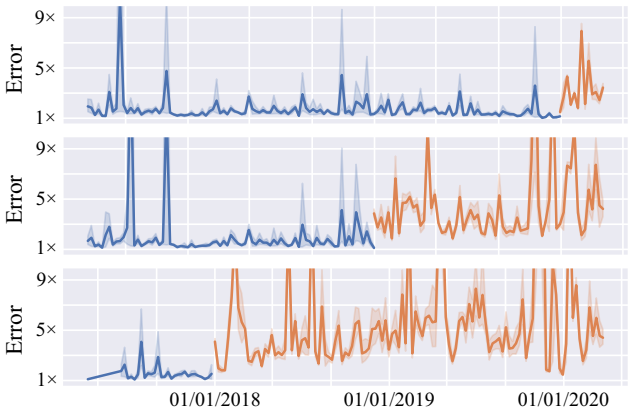


Fig. 1: Mean absolute error (MAE) XGBoost I/O throughput prediction errors as a function of time, averaged per week. Blue lines represent errors on unseen (test) data from the same time range the model was trained on; orange lines represent errors on data collected after the model was already trained.

$2\times$ value specifies that the model on average predicts an I/O throughput either double or half of the real value.

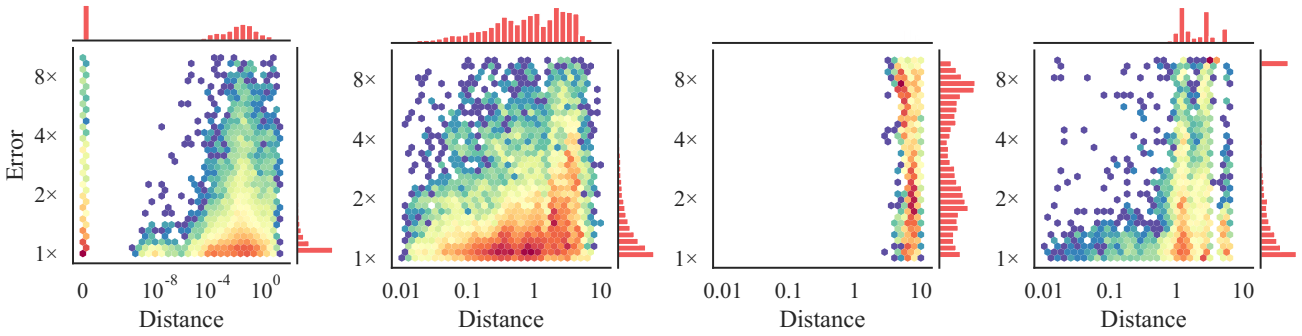
One can observe that the jump in error coincides with the end of the training set, since the ML model that made the predictions was trained on data from April 2017 to January 1st, 2020. What is surprising is that the model performed well on test sets (blue line). The goal of the test set is to inform us of how well we should expect the model to perform in production, yet the random test set and the 2020+ test set predictions have very different error distributions. These results led us to conclude that these models are overfitting, but the test sets were not able to detect this issue. In the rest of the section, we examine the source of the disparity between test set and real-world data.

When creating the test set used in the above example, we used cross-validation, where we randomly split the dataset into training, validation, and test subsets. We trained an XGBoost regressor using the training set, optimized metaparameters using the validation set, and evaluated the performance on the test set. This is widely considered best practice: since the model does not know the validation and test sets, it cannot memorize them as it can with the training set; instead, it has to develop deeper insight in order to successfully achieve good accuracy. Nonetheless, once the model encountered newly collected data, its accuracy was significantly degraded.

Following the I/O throughput formulation from [4], the reason for this change in accuracy might be that (1) the HPC system may have had a change in components, causing jobs to achieve different I/O throughputs or (2) the nature of the applications running on the system changed. We first set out to evaluate whether any changes to the system are the reason for this decreased performance. Without logs on the actual hardware changes to the system, we rely only on Darshan logs. Since these hardware changes can be treated as individual events, we expect that after such an event, the accuracy of the model drops. If a large system change happened around January 2020, that might explain the increase in error. To evaluate this option, we repeated the experiment from the first row of Figure 1 but used a different range for the training/test cutoff. In the second and third rows of Figure 1, we show the performance of a model trained on data from April 2017 to January 2019 and January 2018, respectively. One notices that for all three cutoffs at January 1, 2018, 2019, and 2020, the test error jumps significantly as we cross the new year mark. This leads us to the conclusion that there exist no specific events in the system that drastically change the system's I/O profile and that the models are simply not generalizing well. This conclusion was independently verified with ALCF administrators. This behavior still leaves open the question of why we have excellent performance on the random-generated test set but not on newly collected data.

We hypothesize that this generalization gap between test set performance and real-world performance does not truly exist but that methodologies for creating test sets are inappropriate for the data distribution we are working with. We propose that we are unable to detect overfitting because of the highly dense and clustered nature of our data [10]. When randomly splitting the dataset into training and test sets, often jobs with similar profiles or jobs from the same application would get separated into the two sets. This separation allows our model to simply memorize train-test pairs and achieve good performance. When new data arrives, even though it may stem from the same distribution (e.g., the same applications but run with slightly different configurations), the model can no longer rely on memorization and underperforms. However, because of the combination of the random sampling we use to create the test set and the dense nature of the dataset, this overfitting goes unnoticed until the model is tested in the field. Additionally, since we are unable to detect this behavior on the validation set, our metaparameter optimization step does not improve model accuracy.

To better illustrate this, we ran a simple experiment where we randomly split a dataset into the training and test set with an 80/20 ratio. We predict the I/O throughput of each job in the test set by taking the I/O throughput of that job's nearest neighbor (NN) from the training set. In Figure 2a we show a 2D histogram of these nearest-neighbor pairs of jobs, with Manhattan distances between paired jobs on the x axis and I/O throughput prediction errors on the y axis. The graphs above and to the right of the 2D histogram present the histograms of the marginal distributions of the same data. The cell color represents the number of jobs that have that bin's distance and error value and are logarithmic, with an overwhelming majority of jobs belonging to the red and orange areas. We can draw three conclusions from this graph: (1) as the distances between nearest neighbors grow, so do the prediction errors, (2) almost all jobs in the test set have a nearest neighbor in the training set that has a Manhattan distance less than 1, and (3) a significant number of jobs have "twin" jobs—jobs that have a distance of 0; in other words, these pairs of jobs have identical Darshan input features. We call these pairs "duplicates," since all of these pairs have identical executables



(a) Random test set (b) Climate dataset (c) Cosmology dataset (d) DBSCAN dataset, $\epsilon = 1$

Fig. 2: Test set jobs' distance to their nearest neighbor in the training set vs. difference in their I/O thoughputs.

and represent multiple runs of the same programs, although possibly on different data of the same size and format. Further investigation shows that a large portion of these duplicates are HPC benchmarking jobs.

We conclude that the distance between nearest-neighbor pairs (with one element drawn from the training and another from the test set) is very small. Using Manhattan distances on our 52-dimensional Darshan feature space, the average distance between any two jobs is ≈22, while the average distance between nearest-neighbor pairs is only 0.12. Hence, using a k-nearest-neighbor (KNN) model to predict I/O throughput should provide good results, even though KNN is a nonparametric model and does not generalize at all. Therefore, if we want to test generalization, we need to increase the minimum distance between training and test sets.

V. Robust Test Set Generation

We seek methods for generating test sets that will reveal overfitting and generalization. As discussed, the core issue with randomly splitting the dataset into training and test sets is that the minimum distance between the two sets is very small. Even if a model is overfitting, that is, simply memorizing input-output pairs, because of the training-test distribution similarity, models can have excellent performance on the test set. However, as soon as they are evaluated on new applications or even just new application configurations, they underperform.

We present two methods for separating datasets into training and test sets. The first approach is a per-application method, where we select all jobs of a given application and hold them out of the training set. The second method is a DBSCAN-based method, where we cluster jobs at a certain ϵ value and hold out a single or multiple clusters at random from the test set. We hypothesize that these methods will better estimate robustness to new applications and application configurations.

A. Per-application test set generation

In the first method, we test the generalization of our models by evaluating them on a completely novel application. Since different applications have very different I/O profiles, a single application does not suffice to properly measure model generalization. This can be seen in Figures 2b and 2c. Here the climate application test set often has very small distances to the training set jobs, and using k-nearest-neighbor predictors for I/O throughput can yield good results. On the other hand, the cosmology test set is clearly separated from the rest of the dataset and has very large NN-based I/O throughput prediction errors. Neither of the datasets would provide a good picture of out-of-sample model errors. Therefore, when using the per-application test set generation method, we have to run a number of experiments, one for each of the selected applications. In each run, one of the n most represented applications is held out from the training set and evaluated on, and the results are later aggregated.

B. DBSCAN-based test set generation

A crucial issue with using per-application test set generation is that held-out applications have very different average distances to their nearest neighbors in the training set, so our application-based error is highly sensitive to which and how many applications we choose. Ideally, we would perform a sweep over all applications, holding out every application exactly once, but that is not practical since we have more than 600 different applications recorded in our Darshan logs. Another issue is that many of the applications on average have a very small distance to the training set (e.g., the climate application from Figure 2b). This approach again runs into the same issue as with the randomly sampled test set. Therefore, we seek a method that can *guarantee* training-test set separation, in other word, that the minimum distance between any training-test pair of jobs is greater than some value ϵ. We propose to use DBSCAN clustering for test set generation. DBSCAN is an agglomerative density-based nonparametric clustering algorithm that has been shown to perform well for clustering HPC jobs [10]. DBSCAN works by iteratively clustering together jobs that are within ϵ distance of each other, where ϵ is a user-specified parameter. With each iteration, jobs may form clusters, existing clusters may have new jobs appended to them, and clusters that are closer than ϵ get merged into a single cluster. This process is repeated until no change to the clustering happens during an iteration. Therefore, when DBSCAN converges on a clustering, we

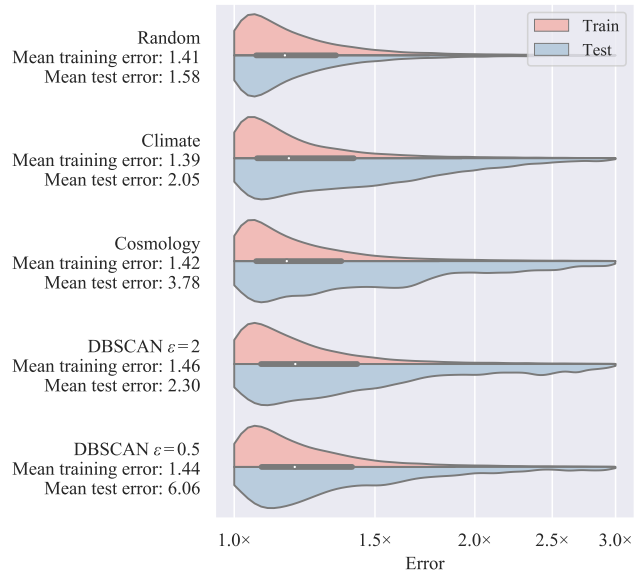


Fig. 3: Distribution of I/O throughput prediction errors on training and test sets generated by different methods.

know that the distance between any pair of clusters is greater than ϵ; otherwise clusters or jobs closer than that would have been combined together. DBSCAN is useful since it does not impose a structure on the clustering (unlike in, e.g., k-Means, where clusters are roughly spherical), and it also does not enforce a fixed number of clusters. For example, for large ϵ values the whole dataset may get merged in a single cluster, while for small values each job may exist in its own cluster. In Figure 2d we show distances and errors of a nearest-neighbor I/O throughput prediction, where a test set consists of randomly selected DBSCAN clusters for $\epsilon = 1$. As we can see, there exists a cutoff on the x axis at the 1-unit mark, with the majority of the data past the distance of 1.

C. Evaluating models on the new test sets

Now that we have test sets that may better diagnose when a model is overfitting, we test how more complex ML models perform on them. In Figure 3, we show the performance of an XGBoost regression model on different test sets. First, note that for the randomly sampled training/test split, the training and test error distributions are almost identical. This is not the case on any of the other datasets, where test set errors are considerably greater than training errors. This is not a negative result. Instead, this underperformance has existed before but went undetected. Now that we have a method to measure the lack of generalization, we can put effort into finding models that are more applicable to the target domain.

D. Test set stability

Another issue with holding out applications or clusters is stability. Our experiments show that predictor performance varies significantly cluster to cluster and that certain groups of jobs are harder to model than others. Hence, we can get very different errors simply by, for example, tuning the random seed. To fix this problem, we use K-fold cross-validation. When this method is applied to the DBSCAN-based test sets, we split the clusters at random into n groups with approximately the same numbers of jobs. Next, we train the predictor n times, each time holding out one of the groups of clusters and training on the rest. This approach guarantees that every cluster and every job belong to some test set once and only once. This modification increases the compute requirements n times but is necessary for reproducibility. With stable measurements of model performance, we run a sweep over the ϵ value and evaluate performance of models on the DBSCAN-based test set. In Figure 4 (top) we show DBSCAN-based test set errors for a range of ϵ values, as well as histograms of cluster sizes at different ϵ values.

On the top left part of the figure, ϵ values are small enough that DBSCAN will cluster each point by itself. Doing so will cause our DBSCAN-based method to behave identically to a randomly sampled test set generation method, hence the flat curve. On the top right, as we increase ϵ, the test set contains progressively larger and larger clusters. For large enough ϵ values, we expect that the majority of the conventional applications get clustered in a single cluster.

That leaves us with the choice of which ϵ value to use in the test sets. There is no right answer here, since we do not know what to expect from future jobs. With larger ϵ values, we will optimize for models that generalize better but possibly perform less than ideal on easy-to-predict jobs. With smaller values, we may arrive at excellent performance on "dense" areas of the space—areas where we have a lot of data, but our models may underperform on novel applications. One option is to select a value so that jobs from a single configuration of a given application get clustered together but that jobs from different configurations end up in different clusters. Using our HPC I/O performance analysis tool Gauge [11] (available at ascslab.org/research/gauge), we can see that for ϵ values in the $[0.5, 1]$ range, clusters typically consist of jobs from a single application, with smooth transitions between the jobs. Therefore, we select $\epsilon = 0.5$ as the value we will use in future experiments. We plan to explore more judicious mechanisms for selecting these values and the trade-offs we make.

VI. Limits of I/O Throughput Prediction

Evaluating an I/O throughput prediction method is difficult without a baseline to compare against. While the field of HPC modeling has explored several approaches for throughput prediction on various datasets and with different goals [4], [5], [12], there does not exist a well-accepted benchmark on which we can test models. In this section we describe an alternative method of evaluating ML models. Instead of comparing performance against weaker baselines, we try to determine the lower bound of the error an I/O throughput model can achieve. To that end, we explore sources of noise that our logging infrastructure does not attempt to measure.

Other than the I/O profile of the job, I/O contention is the single largest source of job performance variability [13]. To get insight into I/O contention, many works had to seek

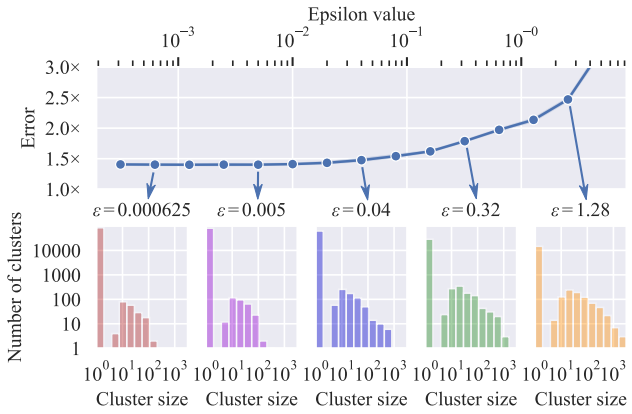


Fig. 4: Top: XGBoost regressor error on DBSCAN-based test sets generated with different ϵ values. Bottom: histograms of cluster sizes for clusterings at different ϵ values.

outside datasets [14] to estimate the state of a whole HPC system at a given time [15]. While Darshan collects hundreds of different features about the behavior of jobs, how they access individual files, and optionally even the behavior of individual ranks (e.g., through Darshan DXT), I/O utilization logs are not currently being collected on Theta. To substitute I/O contention logs, we initially attempted to estimate I/O contention by reconstructing the state of the system using the Darshan logs. Since Darshan profiles only 28% of all the jobs running on Theta, however, we are unable to get a clear picture of I/O subsystem utilization. Therefore, in further text, we assume that we cannot learn I/O contention of the system at a given moment and instead opt to model it as noise. Doing so is problematic since machine learning models are typically much more sensitive to noisy features they are trying to predict than to noisy input features. While a noisy input feature may deteriorate the prediction of that single job, a noisy output feature can impact how the model is trained and therefore deteriorate *all* predictions [16]. That is why we first seek to gain greater intuition about I/O throughput noise, while restricted only to Darshan logs.

A. Duplicate jobs as insight into I/O contention

Several classes of ML models have a property that they can approximate any continuous real function down to some arbitrary precision. While in theory this means that they can achieve perfect accuracy on a training set, this applies only to well-defined functions. In practice, the training dataset may contain multiple datapoints that have the same input features but different target features (e.g., two copies of the same image but with different labels). Since the model has no way to differentiate such datapoints, it cannot achieve 100% accuracy on the training set. While such cases may complicate development of ML models and hurt accuracy, we use these examples to instead evaluate I/O noise.

Duplicate jobs are a key piece of data that allows us to gain insight into I/O contention without actually having I/O subsystem logs. We define duplicate jobs as jobs that have identical Darshan profiles: they have opened the same numbers of files, transmitted the same amounts of data to and from them, have had the same I/O motifs, and so on. They differ, however, in features that are also a function of the system as well of the job: I/O throughput, job runtime, file open and close times, and so on. Note that we do not use these system-sensitive features in our analysis because they are also used by Darshan to estimate I/O throughput [10]. Hence, duplicate jobs look completely the same to our models since the input features are identical. Therefore, given a job, predicting the average value of all of the job's duplicates is guaranteed to give the lowest error *when evaluated on the training set*. Note that this situation does not hold on the test set, since a model may exist that uses information about the distribution of other sets of duplicates to make a better prediction.

We have already witnessed duplicate jobs. In Figure 2a, we see that a large number of training-test pairs of jobs exist that have a distance of 0 (vertical bar on the left). These are duplicate jobs—to our models they are identical, and when duplicate jobs get split over the training and test set, memorization can give good results. Duplicate jobs occupy a surprisingly large portion of our dataset: 21.6%.

We now analyze the I/O throughput noise of duplicate jobs. In Figure 5, we see a scatter plot of I/O throughput and I/O prediction errors for duplicates belonging to the top five applications, in terms of how many duplicates each application has. Each application's jobs are colored in a different color. Since ML models cannot distinguish duplicate jobs, we use a simple mean value predictor that maps a given set of input features to an average I/O throughput. Notice that most jobs lie on diagonal lines of the same color. All jobs on a diagonal are the same type of twin jobs (i.e., they have the same input features); and as they vary in I/O throughput, those differences are linearly reflected in I/O throughput prediction errors, hence a straight line. On the right of the scatter plot we have multiple histograms of I/O throughput prediction errors, sorted by application. Obviously, some types of applications (e.g., the compute-heavy benchmarks) are less sensitive to I/O contention than are others (e.g., the I/O-heavy benchmarks).

B. Prediction errors on duplicate jobs vs. the whole dataset

The reason we are interested in duplicate jobs is that since ML models cannot perform any better than by predicting the average value (on training set data), we can estimate the lower bound on I/O throughput prediction error for duplicate jobs. While derived from duplicates, this bound should apply to other jobs, too. A complicating factor is that this accuracy bound depends on the job. Some jobs are easier to predict than others, likely because they are less sensitive to I/O contention. As an example, take I/O-heavy and compute-heavy benchmarks in Figure 5, and notice the significantly different distributions of prediction errors (blue and orange marginal distributions on the right). We now ask the question of whether we can similarly calculate the amount of noise for the nonduplicate jobs. Since we essentially used nearest-neighbor predictions for duplicate jobs (where the number of neighbors is decided by the number of duplicates for

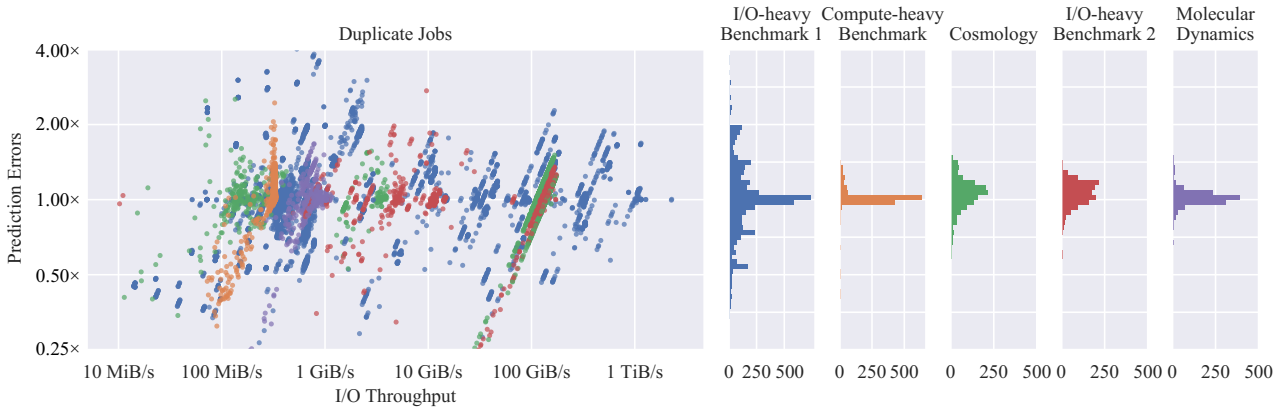


Fig. 5: I/O throughput vs. training set I/O throughput prediction errors for duplicate jobs in the top five most duplicate-abundant applications. Each point in the left graph represents a duplicate job's I/O throughput (x axis) and the error when the job's I/O throughput is predicted as the average I/O throughput of the job's twins (y-axis). Graphs on the right are marginal distribution histograms.

each duplicate job), we use a k-nearest-neighbor predictor to predict I/O throughput. As we are attempting to estimate I/O throughput noise and not model generalization, we use the original randomly sampled training/test set split. In Table I we show the R^2 scores for both the predictions just on duplicate jobs and on the whole dataset using KNN. Note that for $k = 5$, predictions are almost as accurate as that of duplicates.

Type	Duplicates	$k = 1$	$k = 2$	$k = 5$	$k = 10$	$k = 20$
R^2	0.974	0.966	0.972	0.973	0.970	0.967

TABLE I: I/O throughput prediction R^2 scores on duplicate jobs (mean estimates), and on the whole dataset (KNN).

This result is sensible since most jobs have neighbors within an extremely small distance (90% of jobs have a neighbor within a Manhattan distance of 0.2, Figure 2a). This allows us to estimate that no model of I/O throughput can achieve an R^2 score higher than 0.974 on our dataset.

VII. Increasing Prediction Accuracy On Out-Of-Sample HPC Jobs

As a reminder, the goal of this work was to improve our I/O throughput modeling approach so that models would be more robust to new applications and application configurations. With the new test set we have built, and with insight into the upper bound on model accuracy, we attempt to find ML models of I/O throughput that generalize well. An additional goal of this section is to evaluate whether metaparameter optimization works better on the new DBSCAN-based test set and whether metaoptimizing on it yields more generalizable models.

For our tests, we select $\epsilon = 0.5$ as a good compromise between robustness and accuracy, since at that value the clusters typically consist of multiple runs of the same application with the same or a slightly varying configuration.

To evaluate the potential for increasing generalization through metaparameter tuning, we train different XGBoost models with a variety of configurations. We perform a grid search over 5 parameters for a total of 480 experiments: (1) type of test set (DBSCAN or random), (2) number of XGBoost trees, (3) depth of XGBoost trees, (4) XGBoost subsample ratio, which sets the percentage of the (randomly shuffled) dataset each new tree gets exposed to, and (5) XGBoost column sample by tree ratio, which sets the percentage of (randomly shuffled) features each new tree has access to.

We separate experiments into those evaluated on the DBSCAN and randomly sampled test sets. On the left of Figure 6, we see the distribution of R^2 scores for each of the 240 experiments on the randomly sampled test set. The graphs on the right show the heatmaps of best-achieved R^2 scores for the specified configurations of parameters. Notice that on the random test set, more model capacity (through either more trees or more depth) means better accuracy. We also see that models are not very sensitive to the sample ratio parameters, judging by the small variance of the values in the right heatmap. Overall, performing a metaparameter search on this test set may not improve our models, since the R^2 score distribution is very dense and does not respond to metaparameter changes significantly.

In Figure 7, we repeat the same experiment but on the DBSCAN-based test set. Here, the R^2 scores clearly are far more sensitive to metaparameter changes, judging by the larger R^2 variance compared with the previous experiment. We see that here the models are more sensitive to the choice of tree depth but are insensitive to the number of trees. These results may point to the fact that when models are no longer encouraged to overfit, more capacity does not improve accuracy. Interestingly, models are now very sensitive to the choice of subsample and column sample by tree metaparameters. A single configuration of metaparameters exists that gives significantly better results than any other (also highlighted on the histogram as the pink bar). This experiment gives evidence to the conclusion that without the right test set generation methodology, we cannot properly evaluate and compare mod-

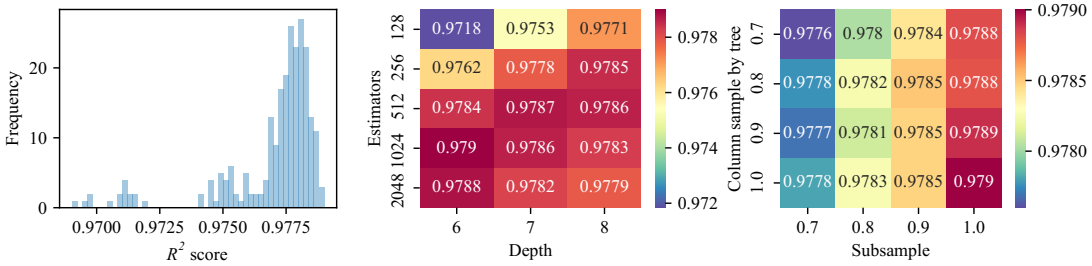


Fig. 6: Randomly sampled test set R^2 scores for a grid search over four parameters. Left graph shows the histogram of R^2 scores; middle and right heatmaps show best achieved scores for specified configurations of parameters.

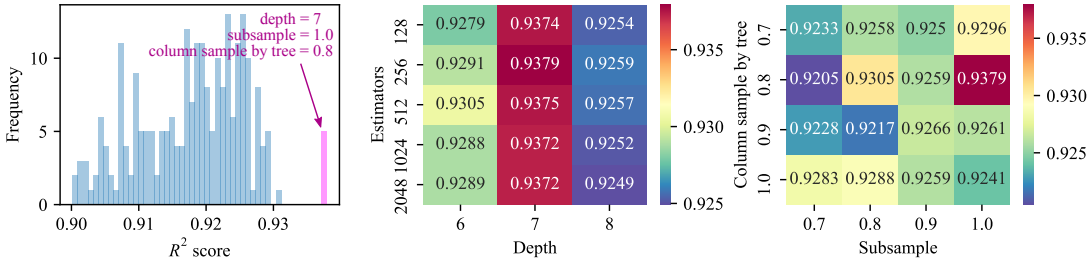


Fig. 7: R^2 distribution and heatmaps for a grid search evaluated on the DBSCAN-based test set.

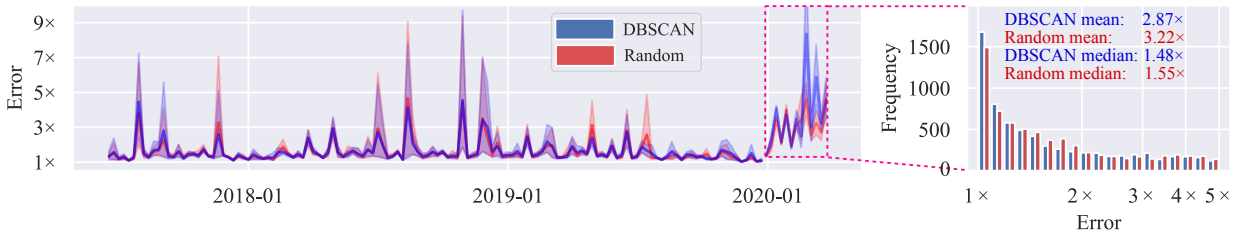


Fig. 8: Evaluation of models with hyperparameters optimized on the randomly sampled test set (red) and DBSCAN test set (blue). Training data is collected in the period from April 2017 to January 2020.

els and so cannot perform a metaparameter search.

We next evaluate how the new models found through a hyperparameter search on the DBSCAN dataset perform in the real world. We repeat the experiment from Figure 1 but this time evaluate two models: one whose hyperparameters were optimized on the randomly sampled test set (red) and one on the DBSCAN-based test set (blue). In Figure 8 (left) we show the errors of the two models both on test data from periods that they were trained on (up to January 2020) and on data collected after the models were trained. On the right, we show the histogram of errors on the new data, where we can see that the model metaoptimized on the DBSCAN test set achieves a 12% lower mean error than does the test set metaoptimized model. We expect that further finetuning and more fine-grained metaparameter searches will reveal models that achieve better results.

VIII. Conclusion

Models of I/O throughput in HPC systems can be used for better scheduling of jobs, for finding the bottlenecks of HPC jobs, and as a proxy for the system that we can analyze and whose dynamics we can explore. However, a large problem in using ML models is their brittleness when deployed in the field. In this work we explore why ML models do not generalize to unseen data as we would expect and why we are unable to detect such lack of generalization. We diagnose this problem and attribute it to the nature of the dataset and the high similarity between multiple runs of the same applications. We develop new methods for generating test sets, allowing us to more accurately measure generalization. Next, we explore the limits of I/O throughput prediction; and by using duplicate jobs (repeated runs of the same applications and with same input configurations), we can estimate the I/O throughput noise. This provides us with an estimate of the upper limit in accuracy a model can achieve. Then, using the new test sets, we are able to run metaparameter searches and arrive at models that have greater extrapolative power than do models evaluated on randomly sampled datasets.

Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

REFERENCES

[1] R. Latham, R. Ross, Q. Koziol, A. Ching, and R. Ananthakrishnan. HPC I/O for computational scientists. [Online]. Available: https://extremecomputingtraining.anl.gov/files/2014/01/hpc-io-all-final.pdf

[2] X. Yang, J. Jenkins, M. Mubarak, R. Ross, and Z. Lan, "Watch Out for the Bully! Job Interference Study on Dragonfly Network," 2016, pp. 750–760.

[3] L. T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 2005, pp. 40–40.

[4] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. M. Wild, "Machine learning based parallel I/O predictive modeling: A case study on Lustre file systems," 2018.

[5] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. Wild, "Modeling I/O performance variability using conditional variational autoencoders," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 109–113.

[6] B. Armstrong and R. Eigenmann, "Performance forecasting: Towards a methodology for characterizing large computational applications," in *Proceedings. 1998 International Conference on Parallel Processing (Cat. No. 98EX205)*. IEEE, 1998, pp. 518–525.

[7] B. Armstrong and R. Eigenmann, "Performance forecasting: Characterization of applications on current and future architectures," *Purdue Univ. School of ECE, High-Performance Computing Lab. Technical report ECE-HPCLab-97202*, 1997.

[8] X. Zhang and Z. Xu, "Multiprocessor scalability predictions through detailed program execution analysis," in *Proceedings of the 9th international conference on Supercomputing*, 1995, pp. 97–106.

[9] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular HPC I/O characterization with Darshan," in *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, 2016, pp. 9–17.

[10] M. Isakov, E. del Rosario, S. Madireddy, P. Balaprakash, P. Carns, R. Ross, and M. Kinsy, "HPC I/O throughput bottleneck analysis with explainable local models," in *SC'20: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020.

[11] E. del Rosario, M. Currier, M. Isakov, S. Madireddy, P. Balaprakash, P. Carns, R. Ross, and M. Kinsy, "Gauge: An interactive data-driven visualization tool for HPC application I/O performance analysis," in *2020 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, 2020.

[12] B. Xie, Y. Huang, J. S. Chase, J. Y. Choi, S. Klasky, J. Lofstead, and S. Oral, "Predicting output performance of a petascale supercomputer," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 181––192.

[13] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 155–164.

[14] C. McCurdy, G. Marin, and J. S. Vetter, "Characterizing the impact of prefetching on scientific application performance," in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds. Cham: Springer International Publishing, 2014, pp. 115–135.

[15] Z. Liu, R. Lewis, R. Kettimuthu, K. Harms, P. Carns, N. Rao, I. Foster, and M. E. Papka, "Characterization and identification of HPC applications at leadership computing facility," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020.

[16] B. Frénay and A. Kabán, "A comprehensive introduction to label noise," in *ESANN*, 2014.