MACHINE LEARNING FOR SELECTING PARALLEL I/O BENCHMARK APPLICATIONS

BY

HONG WEI NG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Emerita Marianne S. Winslett

ABSTRACT

I/O is one of the main performance bottlenecks for many data-intensive scientific applications. Accurate I/O performance benchmarking, which can help us better understand the causes of these bottlenecks and to guide the performance optimization of poor performing applications, is therefore an important problem. We investigate the use of submodular function maximization as a way to select a set of I/O benchmark applications using measures of similarities between applications computed from I/O statistics obtained from the Darshan logs of their jobs. Our optimization problem simultaneously seeks a set of applications that are representative of the applications running on the HPC platform they are chosen from while simultaneously encouraging them to possess diverse I/O behavior between them. We evaluate the quality of the selected applications by training classifiers using features extracted from the jobs of these applications to predict the I/O performance of other jobs that were ran on the platform. Our experiments indicate that the trained classifiers can achieve a fair level of accuracy, thereby lending credence to the feasibility of our optimization approach for selecting I/O benchmark applications.

ACKNOWLEDGMENTS

I am very grateful to my advisor, Professor Marianne S. Winslett, for her help and guidance throughout the course of this project. Over the years, I had many enjoyable discussions with her on research, machine learning, and life in general. I believe the many lessons I picked up from her on research and on how to lead a productive life will benefit me for years to come. Thank you.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 BACKGROUND

Traditionally, the selection of an I/O benchmark is performed by surveying the requirements of users on a platform followed by careful analysis and selection of the benchmark applications from the set of proposed applications by domain experts [1]. The benchmark can then be used, for e.g., to tune the parameters of a I/O benchmarking tool such as IOR to emulate the I/O behavior of these applications so that their I/O performance can be predicted accurately [1].

In this thesis, we investigate the possibility of using machine learning and optimization to find a small set of representative applications to be the benchmark for I/O behavior on a HPC system simply by observing their I/O patterns captured by the Darshan I/O characterization tool [2]. Instead of emulating the I/O workload on a system using software such as IOR or I/O-kernels like Flash-IO [3], we aim to directly predict I/O performance by training classifiers using I/O statistics computed from Darshan logs of the benchmark applications selected by our optimization framework.

Specifically, we frame the problem of selecting a set of I/O benchmarking applications as a submodular function maximization (SFM) problem seeking a small set of applications that are representative of the entire collection of applications run on the HPC system. Simultaneously, we leverage the SFM machinery to encourage selecting applications that have diverse I/O characteristics, for e.g., different parallel I/O strategy, uses different I/O API (POSIX vs MPI-IO) etc. Treating the problem this way allows us to exploit the richness of the SFM framework to define optimization problems that search for solutions with desirable qualities without resorting to heuristics. It also enables us to tap the vast research that has appeared in recent years on modeling problems using SFM and solving them efficiently with good approximation guarantees.

Our experiments on classifying the I/O performance of jobs using classifiers trained on those belonging to our selected benchmark applications suggests that searching for a set of applications that are representative of all the applications that run on a HPC platform while at the same time exhibit diverse I/O patterns amongst themselves can result in fairly accurate classifiers.

1.2 PROBLEMS AND CONTRIBUTIONS

This thesis makes the following contributions:

- We frame the problem of selecting applications to make up a I/O benchmark as a submodular function maximization problem under cardinality constraints. This enables us to adapt ideas from areas such as document summarization [4] and data subset selection [5, 6] to model our problem.
- We borrow ideas from image classification to represent an application as a single vector computed from the set of feature vectors extracted from the Darshan logs of its jobs [7] and we use this representation to compute similarity scores between applications.
- Our experiments demonstrate that applications chosen using our SFM framework can be used to train fairly accurate classifiers for predicting the I/O performance of jobs submitted to the HPC platform the applications are chosen from.

1.3 LIMITATIONS

Our work has several limitations. Firstly, the set of features we work with are highly summarized statistics (see Appendix A) prepared by a domain expert [8]. These features appears to lack fine-grain information related to parallel I/O performance (for e.g., MPI-IO hints), hence it severely limits the performance of classifiers trained on such data. Also, due to the dataset being highly imbalanced — high performing jobs are very rare and majority of jobs have poor I/O performance — it is difficult for a model to learn to accurately predict the I/O performance of those rare high performing jobs.

1.4 RELATED WORK

Existing works on understanding I/O behavior of HPC system rely on tools such as I/O kernels (e.g., FLASH-IO [3]) that mimics target applications' I/O patterns. Other tools such as Skel [9] generates I/O benchmark codes from high-level description of an application's I/O pattern, hence reducing the amount of work needed to hand design them. Applications such as IOR [1] and MADbench2 [10] are I/O benchmark suites that require tuning to emulate I/O behavior across a span of parameter space, architectures, datasets and applications. The key consideration for these various approaches is to figure out the I/O patterns they should be simulating and how to relate those patterns back to actual applications so that their I/O

behavior can be better understood. The choice of I/O patterns to emulate is often decided by through a mix of strategies such as surveying users [1] and analyzing the I/O logs of applications [8] of a HPC. Our approach can be viewed as complementary to these methods in that we formulate the selection of applications in a data-driven optimization framework where the need for human input is minimized.

In the domain of machine learning, submodular function maximization have been successfully applied to the problem of document summarization [4, 11], sensor placements [12] and data subset selection, [5, 6, 13] among many others [14]. Our work is most similar to that of Lin et al. [11] in that we aim to select a subset of items, in this case applications, that are representative of the collection of applications by framing the task as a submodular function maximization problem. At the same time, we wish to minimize redundancy by encouraging the selected applications to possess a wide range of I/O behaviors.

CHAPTER 2: DATASET

The dataset we use for this thesis is derived primarily from the data used by Huong et al. [8, 15] for their study on I/O behavior of several petascale supercomputers. It consists of features extracted from I/O statistics of jobs run on the Mira supercomputer at Argonne Leadership Computing Facility (ALCF) between April 2013 and October 2015 captured by the Darshan I/O characterization tool [2] .

In the following sections, we present further statistics of this dataset and an analysis of the raw features. In Chapter 3, we will describe how these features are used to represent and measure similarities between applications.

2.1 I/O THROUGHPUT OF JOBS

In this thesis, we are particularly interested in the (aggregate) I/O throughput of jobs as it is a direct measure of the I/O performance of applications on a platform. We follow the work of Huong et al. [8] and define the I/O throughput of a job as the ratio of total bytes moved in Darshan-tracked POSIX IO or MPI-IO calls and the longest total time spent in these calls among all the processes of a job. The reason for defining I/O throughput this way, according to Huong et al. [8], is that typically computation does not resume until the slowest process has finished its I/O activities, so this measure is robust to potentially misleading statistics when I/O loads are unevenly distributed across processes.

Next, we discretize the I/O throughputs of the jobs by grouping them into 3 performance intervals (in GB/s): [0, 1.1), [1.1, 26) and [26, ∞). These intervals are obtained by examining the boundaries of the I/O throughputs in the dataset at the 90 and 99.9 percentile. In other words, 90% of jobs in our original dataset have I/O throughputs less than 1.1 GB/s and 99.9% of them have throughputs less than 26 GB/s. See Figure 2.1 for a graphical depiction of this effect in the dataset and Table 2.3 for the actual numerical breakdown.

This imbalance in the I/O performance is one of several long-tail phenomenon noted in Huong et al. [8]. Modeling this kind of data is challenging due to a lack of examples from the minority classes and standard performance metrics not taking the class imbalance into account [16]. We will address these issues using our framework for selecting benchmark applications in Chapter 4 and when setting up our experiments in Chapter 5.

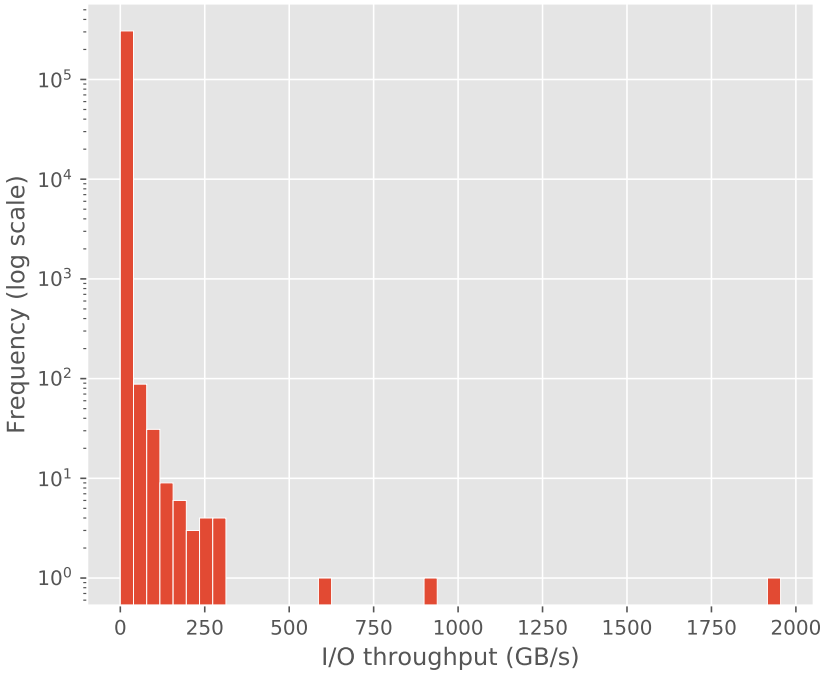

Figure 2.1: Distribution of jobs for various I/O throughput.

2.2 DESCRIPTION OF DATA

Our dataset, denoted $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$, consists of feature vectors $\boldsymbol{x}_i \in \mathbb{R}^d$ computed from Darshan's logs for jobs $1, \ldots, N$ and the corresponding discretized I/O throughput, $y_i \in \{0, 1, 2\}$ (see Section 2.1), associated with that job. A full list and description of these features is given in Appendix A.

2.3 DATA TRANSFORMATION

2.3.1 Data skewness

We know from domain knowledge that the scale of jobs submitted to a HPC platform can vary over a wide range in terms of the number of processes used, amount of data transferred and number of files created etc. [8]. Therefore, we should expect the distributions of features that captures these effects to be skewed. To verify this, we compute the sample skewness for

each feature using the following formula:

$$G_1 = \frac{\sqrt{N(N-1)}}{N-2}\left(\frac{\frac{1}{N}\sum_i^N (x_i - \bar{x})^3}{\left(\frac{1}{N}\sum_i^N (x_i - \bar{x})^2\right)^{\frac{3}{2}}}\right), \tag{2.1}$$

where x_i is the feature's value for the example $\boldsymbol{x}_i$. Table 2.1 shows how these skewness values are distributed across the 44 features. Indeed, for most features, G_1 is large and positive, suggesting that they are extremely right-skewed.

# features	mean	std	min	25%	50%	75%	max
44	141.2	119.9	0.8222	59.17	120	199.6	520.1

Table 2.1: Distribution of skewness of the features

We further show the boxplots of the top 10 most skewed feature distributions ranked according to their G_1 value in Figure 2.2. The boxplots confirm our belief that for the majority of the jobs, these features take on values near 0, and extremely large values for a relatively small number of jobs.

2.3.2 Log Transformation for Correcting Data Skew

We reduce the skewness of the features with $G_1 \geq 2$ by applying the following log transformation to each of those features, denoted x, separately

$$\tilde{x} = \log_{10}(1 + x). \tag{2.2}$$

The log transformation, along with the square root or inverse, is commonly used to reduce the skewness of data so that the distribution of the data becomes closer to that of a Normal distribution, a property implicitly required by many machine learning models [17]. These transformations in turn are specific cases of the more general Box-Cox transformation [18, 17].

2.4 DATASET PREPARATION

We prepare our dataset by first discarding applications that belongs to I/O benchmark suites such as "IOR", "MACSio" etc., by searching for application names that contain the word "benchmark" or those found on a curated list [19]. Note that we do not discard applications whose sole purpose is not for I/O benchmarking but are used to understand

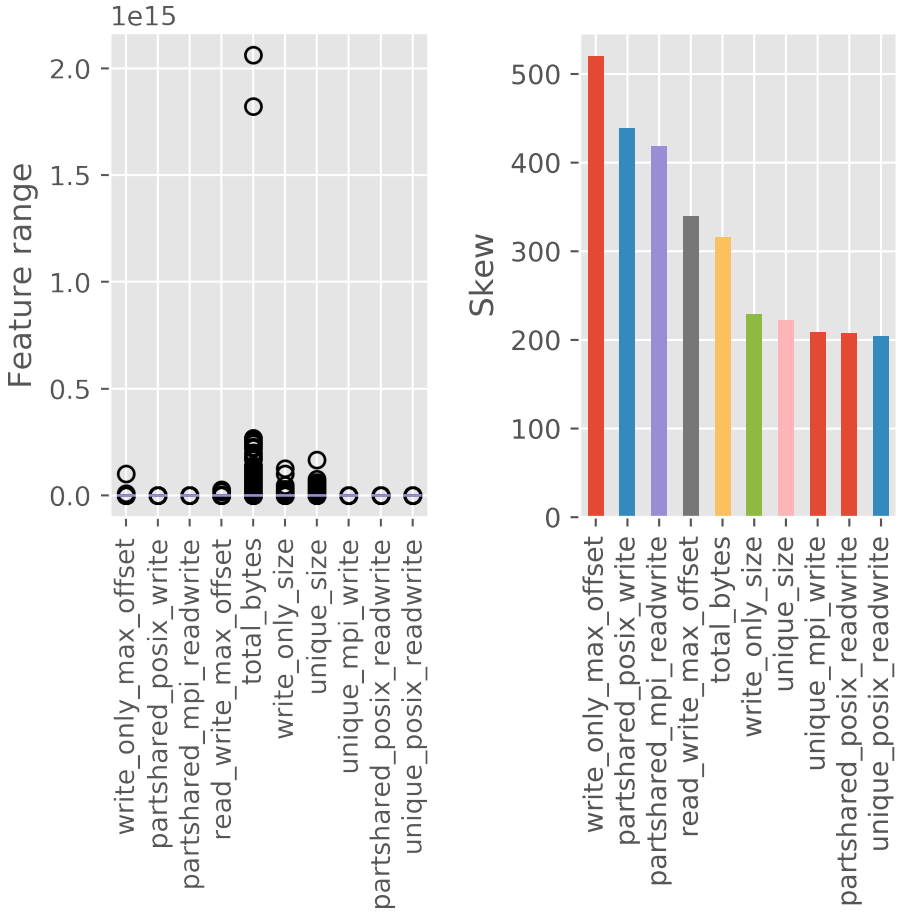

Figure 2.2: Top 10 most skewed features - range and skewness.

a system's I/O behavior due to their importance or ubiquity on various platforms. For e.g., we retain nek5000 even though it is considered an important benchmarking application [19]. The reason for this is that we are searching for applications whose I/O patterns are representative of the diverse workload on the system and we don't want to select or represent these benchmark suites.

After removing the unwanted applications, we split the log-transformed dataset $\mathcal{D}$ into two halves with the same number of jobs based on the time each job started. We then discard applications with fewer than 10 jobs in each half followed by discarding applications in the later half that did not appear in the first half. We denote the first half $\mathcal{D}_{\text{train}}$ as we will use it for selecting our benchmark applications and training our model. The second half is denoted $\mathcal{D}_{\text{test}}$ as we will use it for evaluating our methods. The applications with few jobs are discarded because it is unlikely that we can reliably estimate their feature representation (see Chapter 3) with only so few examples. We remove applications that are not observed in

the first half because we cannot expect these applications to be well represented by those we pick from the first half since they are not observed in that batch of data and therefore it will not be fair to evaluate the performance of our model on those applications.

Pertinent statistics of these $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ are given in Tables 2.2, 2.3, 2.4 and 2.5. In those Table 2.5, "unique" means the job has file(s) that are accessed by only one rank, "shared" means it has file(s) that are accessed by all processes, and part-shared means the some files are accessed by only one process and some by more than one.

	Train Data	Test Data
# Apps	403	90
# Jobs	153624	102894
Earliest Job	05-Apr-2013	13-Dec-2014
Latest Job	13-Dec-2014	21-Oct-2015

Table 2.2: Basic dataset statistics.

	Train data		Test data	
I/O Perf	# Jobs	% Data	# Jobs	% Data
0 - 1.1 GB/s	140182	91.25	91902	89.32
1.1 - 26 GB/s	13374	8.71	10933	10.63
$>$ 26 GB/s	68	0.04	59	0.06

Table 2.3: Number of jobs for each I/O performance level in dataset.

	Train data		Test data	
I/O API	# Jobs	% Data	# Jobs	% Data
POSIX	125085	81.423	59727	58.047
MPI-IO	742	0.483	11343	11.024
POSIX & MPI-IO	27797	18.094	31824	30.929

Table 2.4: Number of jobs for each I/O API type in dataset.

I/O Strategy	Train data # Jobs	Train data % Data	Test data # Jobs	Test data % Data
Unique	115918	75.456	55304	53.749
Part-shared	155	0.101	100	0.097
All-shared	2234	1.454	18143	17.633
Unique & All-shared	32549	21.187	24604	23.912
Unique & Part-shared	1731	1.127	3515	3.416
Part-shared & All-shared	131	0.085	5	0.005
Unique & Part-shared & All shared	906	0.590	1223	1.189

Table 2.5: Number of jobs for each I/O strategy in dataset.

CHAPTER 3: MEASURING SIMILARITIES OF APPLICATIONS

In this chapter, we explore two ways of measuring similarities between applications. They will come in useful in Chapter 4 when we use them in an optimization framework to select benchmark applications based on similarities of applications. Note that we consider similarities only in the narrow context of parallel I/O behavior. Specifically, we consider two applications to be similar if their I/O behavior, as characterized by the I/O statistics captured by Darshan for their jobs, are similar. Details are given in the following.

3.1 REPRESENTATING APPLICATIONS AS SETS OF JOBS

To begin, we consider each application to be represented by the set of feature vector representation of its jobs, $X_i = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{N_i}\}$, where the features are computed from Darshan's logs (see Chapter 2), $i = 1, \ldots, N$ indexes the application, and N_i is the number of jobs observed for the i-th application.

3.2 HAUSDORFF DISTANCE

Given our set representation for applications, a way to measure the distance between a pair of applications X, Y is to use the following symmetric (or two-sided) Hausdorff distance [20],

$$d_{\mathrm{H}}(X, Y) \coloneqq \max\{\sup_{x \in X} \inf_{y \in Y} d(x, y), \sup_{y \in Y} \inf_{x \in X} d(x, y)\}, \tag{3.1}$$

where we choose $d(x, y)$ to be the L_2 metric $||x - y||$. Given the distance between a pair of applications X_i and X_j, we can convert it to a similarity weight $w_{ij} \in [0, 1]$ in several ways. We opt to use a decaying exponential function,

$$w_{ij} = \exp\left(-\frac{d_{\mathrm{H}}(X_i, X_j)}{\sigma}\right), \tag{3.2}$$

and σ is estimated to be the median pairwise distances of every pair of applications. Note that we replace any set with more than 10,000 jobs with 10,000 jobs sampled uniformly at random without replacement from that set to reduce the computational load.

3.3 FISHER VECTORS AND THEIR SIMILARITIES

Another representation of applications that we experiment with is the Fisher vector representation commonly used in image classification [21, 7]. The Fisher vector representation is an extension of the Bag-of-words (BOW) model used in classical Natural Language Processing (NLP). Specifically, given a set of data points, it assumes that they are generated by some generative process. We follow the work of Perronnin et al. [7] and assume this process to be a Gaussian Mixture Model (GMM) with M Gaussians representing a probability density function u_λ with parameters λ. Then, a way to represent a set of jobs of an application $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T\}$ is as the gradient vector [7]:

$$G_\lambda^X = \frac{1}{T} \nabla_\lambda \log u_\lambda(X). \tag{3.3}$$

If we model the GMM as having diagonal covariance matrices, then its parameters are $\lambda = \{\pi_i, \mu_i, \Sigma_i, i = 1, \ldots, M\}$. Let us denote the diagonal of each covariance matrix as the *vector* σ_i and denote the soft assignment of the t-th job of an application, $\boldsymbol{x}_t$ to the i-th Gaussian as:

$$\gamma_t(i) = \frac{\pi_i u_i(\boldsymbol{x}_t)}{\sum_{j=1}^{M} \pi_j u_j(\boldsymbol{x}_t)}. \tag{3.4}$$

Then, the derivatives of the log-likelihood w.r.t. these parameters can be computed as [21, 7]:

$$\mathcal{G}_{\pi,i}^X = \gamma_t(i) - \frac{\exp(\pi_i)}{\sum_{j=1}^{M} \exp(\pi_j)}, \tag{3.5}$$

$$\mathcal{G}_{\mu,i}^X = \frac{1}{T\sqrt{\pi_i}} \sum_{t=1}^{T} \gamma_t(i) \left(\frac{\boldsymbol{x}_t - \mu_i}{\sigma_i} \right), \tag{3.6}$$

$$\mathcal{G}_{\mu,i}^X = \frac{1}{T\sqrt{2\pi_i}} \sum_{t=1}^{T} \gamma_t(i) \left(\frac{(\boldsymbol{x}_t - \mu_i)^2}{\sigma_i^2} - 1 \right), \tag{3.7}$$

where the division of the variances in the vector σ_i is performed elementwise. Finally, the Fisher vector representation of application X is just the concatenation of these gradient vectors. Next, it is recommended to normalize these Fisher vectors by first transforming each of its entry using the function

$$h(z) = \text{sign}(z)|z|^\alpha, \tag{3.8}$$

where α is commonly set to 0.5. This is followed by normalizing the vectors to have unit L_2 norm. Since these vectors have unit norm, the dot product of the Fisher vectors of two

application is just their cosine similarity score. We scale these scores to be in the range [0, 1] by adding 1 and multiplying by 0.5 to become similarity weights w_{ij}. We will use these weights later in Chapter 4 in solving for a set of representative and diverse benchmark applications.

CHAPTER 4: OPTIMIZING SELECTION OF BENCHMARK APPLICATIONS

Submodular functions optimization have been used successfully to achieve state-of-the-art results in a variety of domains such as sensor placement [12], document summarization [11], and product recommendation [22], amongst many others. The common thread among the applications of these domains is the need to choose a subset of objects from a collection such that those chosen are representative of the entire collection in some sense and yet possess diverse characteristics among themselves.

Our work on selecting a set of representative applications to model the I/O performance of a platform is built on top of the theory of submodular function optimization. We therefore begin by giving a short review on submodular functions and their key properties. Following that, we properly formulate the problem of selecting a set of benchmark applications as a submodular function maximization problem that produces solutions with desirable characteristics. Lastly, we list algorithms for solving our submodular function maximization problem along with their theoretical guarantees.

4.1 BACKGROUND ON SUBMODULARITY

We briefly review the theory of submodular functions here. Most of the material are taken from the survey of Krause et al. [14]. The interested reader can refer to Fujishige [23] or Krause et al. [14] for additional details.

A set function is a function $f\colon 2^V \mapsto \mathbb{R}$ that assign each subset $S \subseteq V$ a value $f(S)$. Here V is a finite set often referred to as the *ground set*. For our problem, consider V to be the set of applications from which benchmark applications are to be chosen from.

4.1.1 Basic Definitions

Definition 4.1. A set function $f\colon 2^V \mapsto \mathbb{R}$ is *normalized* if $f(\emptyset) = 0$.

Definition 4.2. For a set function $f\colon 2^V \mapsto \mathbb{R}$, $S \subseteq V$, and $e \in V$, the *discrete derivative* of f at S with respect to e is defined as

$$\Delta_f(e \mid S) = f(S \cup \{e\}) - f(S).$$

We drop the subscript and write $\Delta(e \mid S)$ when there is no ambiguity regarding the function f.

Definition 4.3. A function $f\colon 2^V \mapsto \mathbb{R}$ is *submodular* if and only if $\forall S \subseteq T \subseteq V$ and $\forall e \in V \setminus T$ it holds that

$$\Delta(e \mid S) \geq \Delta(e \mid T).$$

Equivalently, a function $f\colon 2^V \mapsto \mathbb{R}$ is *submodular* if and only if $\forall S, T \subseteq V$,

$$f(S) + f(T) \geq f(S \cap T) + f(S \cup T).$$

The first definition above captures the essence of submodularity — that of *diminishing marginal returns*. This is a phenomenon that occurs in various domains [14] and is a reason why submodular functions are useful for modeling problems of this nature. Intuitively, if $f(S)$ is a measure of the utility of selecting items in S, then what the submodular property implies is that the "benefit" from adding items to a set S diminishes as S gets larger.

Definition 4.4. A function $f\colon 2^V \mapsto \mathbb{R}$ is *supermodular* if and only if $-f$ is submodular. Equivalently, $f\colon 2^V \mapsto \mathbb{R}$ to be supermodular if and only if $\forall S \subseteq T \subseteq V$ and $e \in V \setminus T$,

$$\Delta(e \mid T) \geq \Delta(e \mid S).$$

Yet another way to define a function $f\colon 2^V \mapsto \mathbb{R}$ as supermodular, analogous to the one given in Definition 4.3, is $\forall S \subseteq T \subseteq V$,

$$f(S) + f(T) \leq f(S \cap T) + f(S \cup T).$$

Definition 4.5. A function $f\colon 2^V \mapsto \mathbb{R}$ is *modular* if it is both submodular and supermodular, i.e., $\forall S, T \subseteq V$

$$f(S) + f(T) = f(S \cup T) + f(S \cap T).$$

A modular function is therefore akin to something like a linear function in the sense that a normalized modular function f can always be written in the form $f(S) = \sum_{e \in S} w(e)$ for some weight function $w\colon V \mapsto \mathbb{R}$. Another way to view the modularity property of f is that each item contributes an amount to the value of f independent of the other items in S. In contrast, if f is submodular, then the gain to $f(S)$ by adding an item $e \notin S$ will typically depend on what is currently in S.

Definition 4.6. A set function $f\colon 2^V \mapsto \mathbb{R}$ is *monotone increasing* if and only if $\forall S \subseteq T \subseteq V,\ f(S) \leq f(T)$.

Note that a function f is monotone if and only if all its discrete derivatives are nonnegative. Furthermore, submodular functions that are also monotone, i.e., monotone submodular

functions, have the property that $\forall S \subseteq T \subseteq V$ and $e \in V, \Delta(e \mid S) \geq \Delta(e \mid T)$. Unlike Definition 4.3, we do not require $e \notin T$ here.

4.1.2 Properties of Submodular Functions

We list without proof some useful properties of submodular functions here. These properties allow us to generate a rich set of submodular functions from simpler ones, thereby greatly enhancing the modeling power of the submodular function optimization framework. We will use them to construct relevant submodular functions for our problem of selecting benchmark applications from a set of applications in Section 4.2.

Nonenegative linear combinations. If functions $f_1, \ldots, f_k \colon 2^V \mapsto \mathbb{R}$ are submodular, and $w_1, \ldots, w_k \in \mathbb{R}_{\geq 0}$, then $f(S) = \sum_i w_i f_i(S)$ is also submodular. This is an important property as it allows us to conjure up complex submodular objectives from simpler ones, each with possibly very different characteristics, thereby allowing us to optimize for a far more sophisticated objective than can be done with any single objective alone.

Restriction. If $g \colon 2^V \mapsto \mathbb{R}$ is submodular and $S, P \subseteq V$, then $f(S) = (S \cap P)$ is submodular.

Contraction. if $g \colon 2^V \mapsto \mathbb{R}$ is submodular and $S, P \subseteq V$, then $f(S) = g(S \cup P)$ is submodular.

Reflection. If $g \colon 2^V \mapsto \mathbb{R}$ and $S \subseteq V$, then $\bar{f}(S) = g(V \setminus S)$ is submodular.

Truncation. If $g \colon 2^V \mapsto \mathbb{R}$ is submodular and $S \subseteq V$, then $f(S) = \min\{g(S), c\}$ is also submodular for any constant c.

Concave composition. Given any nondecreasing submodular function $g \colon 2^V \mapsto \mathbb{R}$ and any nondecreasing concave function $h \colon \mathbb{R} \mapsto \mathbb{R}$, the composition $f(S) = h(g(S))$ is submodular. In particular, for any modular function $m \colon 2^V \mapsto \mathbb{R}_{\geq 0}$, $f(S) = h(m(S))$ is submodular.

4.2 PROBLEM FORMULATION

We formulate our problem of choosing a set of benchmark applications for a HPC system (in this case the Mira supercomputer) as a submodular function maximization (SFM) problem.

In doing so, we borrow ideas from the domains of document summarization [11], data subset selection [13, 6, 5], and sensor placement [12], amongst several others that make use of SFM.

First, let V be the ground set containing applications to choose from. Our goal is to select a subset of applications $S \subseteq V$ where $|S| \ll |V|$, such that the applications in S are good representatives of the entire population V in the sense that S can be used to model the diverse I/O behavior of all applications in V well. One can therefore think of S as a "summary" of V, which essentially makes S a benchmark.

Specifically, we formulate the search for a good benchmark, S, as a submodular function maximization problem with cardinality constraint:

Problem 4.1. Find

$$S^* = \underset{S \subseteq V}{\operatorname{argmax}}\, L(S) + \lambda R(S) \text{ subject to: } |S| \leq K \tag{4.1}$$

where $L(S)$ is a submodular function that measures how well S covers the applications in V in terms of their overall similarity, $R(S)$ is a submodular function that encourages diversity in S and $\lambda \geq 0$ trade-off one objective for the other. K is a positive integer given as input for the number of applications to select for the benchmark.

Note that the nonnegative weighted combination of two submodular functions is still a submodular function (see Section 4.1.2). Also, if $L(S)$ and $R(S)$ are both monotone submodular functions, then there is no difference between requiring $|S| \leq K$ and $|S| = K$ as adding more items cannot hurt the performance of the objective function, so K items will always be picked.

A more general variation of **Problem 4.1** with knapsack constraint is:

Problem 4.2. Find

$$S^* = \underset{S \subseteq V}{\operatorname{argmax}}\, L(S) + \lambda R(S) \text{ subject to: } \sum_{i \in S} c_i \leq B. \tag{4.2}$$

The difference between these two problem is that here each item has an associated cost, c_i, and B is a given total budget. This variation is useful if there is a way to associate a cost with each application. For e.g., one might define the c_i to be the run time of the i-th application and B to be the total run time. The problem can then be interpreted as finding the best benchmark that runs within a given time budget B. Note that if all the c_i are 1 and $B = |S|$, then the two problems are identical.

It is possible to consider other more sophisticated constraints when maximizing a sub-

modular function, for e.g., matroid contraints [24]. However, our focus will primarily be on **Problem 4.1**, where we deal only with cardinality constraints, and our choice of set functions will be monotone submodular. The reason is that problem of this nature can be optimized using a simple greedy algorithm to obtain a solution guaranteed to be within $(1 - \frac{1}{e}) \approx 63\%$ of the optimal value. Details will be given in Section 4.5.

4.3 SUBMODULAR COVERAGE AND DIVERSITY FUNCTIONS

We describe a number of possible submodular functions for $L(S)$ and $R(S)$ taken from existing literature [11, 4, 14] here.

4.3.1 Coverage

In the following, let $w_{ij} \geq 0$ be a measure of similarity between the i-th and j-th application in V. They can be computed using any of the similarity measures described in Chapter 3. The nonnegative requirement on w_{ij} is necessary to ensure the following functions are nonnegative monotone submodular as this is a requirement to ensure the optimality guarantees of the greedy optimization algorithm described in Section 4.5.

Facility location. One way to model coverage is to measure the maximum similarity between each application $i \in V$ with any one of the chosen application in $j \in S$. This idea can be represented using the following uncapacitated facility location function [25, 4]:

$$f_{\text{fac}}(S) = \sum_{i \in V} \max_{j \in S} w_{ij}. \tag{4.3}$$

If we set $f_{\text{fac}}(\emptyset) = 0$, and if $w_{ij} \geq 0$ for all i, j, then this function is monotone submodular [26, 14].

Max cut. We can also quantify how well S represents the set V by measuring how similar S and $V \setminus S$ are. This idea is commonly encoded using the graph cut function

$$f_{\text{cut}}(S) = \sum_{i \in V \setminus S} \sum_{j \in S} w_{ij}, \tag{4.4}$$

which is known to be submodular [4].

Coverage. A simple way to measure the representativeness of S is through the following monotone submodular function described in Lin et al. [11]:

$$f_{\text{cov}}(S) = \sum_{i \in V} \sum_{j \in S} w_{ij} \tag{4.5}$$

Saturated coverage. An alternative to f_{cov} that is also submodular and can sometimes give better results [11] is

$$\begin{aligned} f_{\text{sat}}(S) &= \sum_{i \in V} \min\{C_i(S), \alpha C_i(V)\}, \\ C_i(S) &= \sum_{j \in S} w_{ij}, \end{aligned} \tag{4.6}$$

and $\alpha \in [0, 1]$ is a tunable threshold parameter. Note that $C_i(S)$ is submodular, $\alpha C_i(V)$ is actually a constant, so using the properties given in Section 4.1.2, we can easily show that f_{sat} is submodular. The intuition behind this function as given by Lin et al. [11] is to first observe that $C_i(S)$ basically measures how well element i is "covered" by S. Therefore, limiting the maximum coverage of each element to $\alpha C_i(V)$ allows elements which are disimilar to i to be added to S if i is already "sufficiently well covered" by S.

4.3.2 Diversity

We introduce a number of submodular functions for encouraging applications with diverse I/O characteristics to be selected. In particular, we are interested in selecting applications so that their jobs have

1. different performance intervals,
2. uses diverse parallel I/O strategies (for e.g., shared-file approach, single-file approach, or a mix of the two), and
3. different I/O APIs (e.g., POSIX, MPI-IO).

Specifically, for each of the above three categories, we generate a set of partitions that are the powerset of the possible cases for that category. For e.g., since we have three performance intervals (see Chapter 2), the partitions for this case are applications with jobs performing only in exactly one of the three intervals, a mix of any two of the three intervals, and all three intervals. In general, for a category with n possible labels, we generate 2^n partitions containing the applications with jobs that only belong to that partition. Clearly this is only feasible for categories with relatively few labels.

Partition diversity. Suppose we have disjoint sets $P_1, \ldots, P_n$ such that $\bigcup_i P_i = V$ and $\forall i \neq j, P_i \cap P_j = \emptyset$. These sets can be considered groupings of the set of applications V into separate partition based on some properties that they possess, for e.g., those discussed above. Then

$$f_{\text{part}}(S) = \sum_{k}^{n} \sqrt{|P_k \cap S|} \tag{4.7}$$

is a objective function that encourages the selection of applications from various partitions. This function can be easily shown to be monotone submodular using the properties listed in Section 4.1.2.

Partition diversity with reward. We can make f_{part} more general by associating with each item in the partition a nonnegative reward, r_j, as follow:

$$f_{\text{reward}}(S) = \sum_{k}^{n} \sqrt{\sum_{j \in P_k \cap S} r_j}. \tag{4.8}$$

The reward for picking the j-th application, r_j, can either be calibrated by a domain expert, or estimated somehow based on the data. In the latter case, one might, for e.g., estimate r_j using a formula that takes into account the total I/O time of all the jobs for that application, or the number of its jobs observed within a certain time period. These two ways respectively correspond roughly to encouraging the selection of applications that use I/O heavily or are commonly run. For us, we follow Lin and experiment with $r_i = \frac{1}{|V|} \sum_j w_{ij}$ as our reward function. Note that f_{part} is a special case of this function where all items have the same reward $r_j = 1$.

Diversification. In the absence of any domain knowledge on how to construct a suitable diversity-inducing function, one might simply optimize for the selection items to be different using a function such as

$$f_{\text{dissim}}(S) = -\sum_{i,j \in S} w_{ij}. \tag{4.9}$$

Although we can easily show that this function is submodular, it is also clear that it is not monotone and definitely not nonnegative since the similarity weights w_{ij} are nonnegative. An alternative is to get rid of the negative sign by replacing the similarity weights w_{ij} with quantity measuring the *distance* d_{ij} between the two applications. This result in the

diversification function

$$f_{\text{div}}(S) = \sum_{i,j \in S} d_{ij}. \tag{4.10}$$

The problem with this function is that although it is monotone, it is not submodular. In fact, it is *supermodular*. Therefore, using either of these objective functions will cause us to lose the optimality guarantee of the algorithms given in Section 4.5. But we should note that there exists algorithms to optimize non-monotone submodular functions [27, 28] and the special case of a sum of a submodular function and f_{div} [29], but they come with worse optimality guarantees compared to optimizing monotone submodular functions, and in our case, do not really benefit our problem much since the other diversification functions are already adequate as we shall see in our experimental results in Chapter 5. They are listed here for completeness sake.

4.4 SFM WITH MULTIPLE DIVERSITY TERMS

Initial experiments using f_{cut} and f_{cov} did not produce good results. Therefore, even though we list them here, we choose to experiment only with the other submodular functions. In Chapter 5, we run experiments based on **Problem 4.1.** with f_{fac} or f_{sat} as the coverage function L and either f_{part} or f_{reward} as the diversity inducing function R using any of the three category of partitions given in Section 4.3. We further introduce a third problem that combines all three partition functions

Problem 4.3.

$$\begin{aligned} &\underset{S \subseteq V}{\text{maximize}} \; L(S) + \lambda R_{\text{iotype}}(S) + \beta R_{\text{iostrat}}(S) + \gamma R_{\text{ioperf}}(S) \\ &\text{subject to } \; |S| \leq K \end{aligned} \tag{4.11}$$

where again L is either f_{fac} or f_{sat} and the subscripts iotype, iostrat, ioperf refers to the partitions generated based on I/O API type (MPI-IO or POSIX), I/O strategy (single process, shared) and I/O performance respectively. The function R like before can be either f_{part} or f_{reward}. Also, $\lambda, \beta, \gamma \geq 0$ are weights to tradeoff the influence of each term.

Algorithm 4.1 Greedy (Cardinality constraint)

Require: V, K, and f ▷ Ground set, budget, and set function
1: $S \leftarrow \emptyset$
2: **while** $|S| < K$ **do**
3: $v \leftarrow \text{argmax}_{v \in V} f(S \cup \{v\}) - f(S)$
4: $S \leftarrow S \cup \{v\}$
5: $V \leftarrow V \setminus v$
6: **return** S

4.5 ALGORITHMS FOR SFM

Unfortunately, maximizing a submodular function under cardinality constraints in general is NP-hard [30, 14, 27]. However, for the class of nonnegative monotone submodular functions $f\colon 2^V \mapsto \mathbb{R}_{\geq 0}$, the solution returned from the greedy algorithm by Nemhauser et al. [30] given in Algorithm 4.1 is guaranteed to be within $(1 - \frac{1}{e}) \approx 0.63$ of the optimum. Essentially all the greedy algorithm does is pick the item that maximizes the gain in the submodular objective function at each iteration until the desired number of items have been selected.

A variation of the greedy algorithm by Minoux [31] that is sometimes referred to as *lazy greedy* can greatly improve its speed while retaining its optimality guarantees. The pseudocode, slightly modified from [32], is listed in Algorithm 4.2.

Algorithm 4.2 Lazy Greedy (Cardinality constraint)

Require: V, K, and f ▷ Ground set, budget, and set function
1: $S_0 \leftarrow \emptyset; i \leftarrow 0$; Initialize priority queue Q;
2: **for** $v \in V$ **do**
3: INSERT$(Q, (f(v), v, i))$
4: **while** $|S_i| < K$ **do**
5: $(\alpha, v, j) \leftarrow \text{POP}(Q)$
6: **if** $j < i$ **then** ▷ α was computed in an earlier iteration
7: $\alpha \leftarrow \Delta(v \mid S_i)$
8: **if** $i = j$ or $\alpha \geq \text{MAX}(Q)[0]$ **then** ▷ If α is "fresh" or current largest
9: $S_{i+1} \leftarrow S_i \cup \{v\}$
10: $i \leftarrow i + 1$
11: **else**
12: INSERT$(Q, (f(v), v, i)))$
13: **return** S_i

The main difference between the lazy greedy algorithm and the original greedy algorithm is that it avoids evaluating the function f whenever possible as that operation may be costly. This is achieved through the key insight that the marginal gain of a submodular function

is monotonically nonincreasing as the iteration progresses [14]. That is, for a given item e, iterations i and j where $i \leq j$, we always have $\Delta(e \mid S_j) \leq \Delta(e \mid S_i)$. Therefore, if the marginal gain of an item e computed for the current iteration is greater than the largest marginal gain for the remaining items, even if they were computed in previous iterations, e should be selected as those items' marginal gain can only be smaller or equal to their earlier marginal gains. In this way, we avoid evaluating the marginal gains of those items even if they are "outdated". As mentioned earlier, the optimality guarantees of the lazy greedy and greedy are the same — approximately 0.63 of the optimal. In terms of runtime, their worst case performance is the same, but in practice the lazy greedy can perform up to several hundred times faster than the greedy algorithm [12, 14] in some cases.

CHAPTER 5: EXPERIMENTS AND RESULTS

5.1 EXPERIMENTAL SETUP

As a quick reminder, we described in Section 2.4 how our dataset, $\mathcal{D}$, was split into two parts: $\mathcal{D}_{\text{train}}$ for training and $\mathcal{D}_{\text{test}}$ for testing.

5.1.1 Proposed Approach

To evaluate our proposed approach for selecting benchmark applications, we solve problems 4.1 and 4.3 proposed in Chapter 4 for $K \in \{10, 15, 20, 25\}$ benchmark applications using various combinations of the coverage functions f_{fac} and f_{sat}, and the diversity inducing functions f_{part} and f_{reward} proposed in Section 4.3 to get our benchmark applications S_{OPT}. The set of candidate applications to choose from, V, are the available applications in $\mathcal{D}_{\text{train}}$, and the applicable weights of each term (λ, β, γ) in the optimization problem is determined by running randomized search [33] in the range $[0.1, 1000]$.

After solving for S_{OPT} for each of the above configurations, we train various classifiers on the jobs belonging to these applications to predict which of the performance interval the job's aggregate throughput falls in (see Section 2.1). Note that these jobs will typically only be a small subset of the jobs in $\mathcal{D}_{\text{train}}$ as we select at most 25 out of 403 applications, although some applications have more than 10,000 jobs. The best hyperparameters for each classifier is also determined using a randomized search [33] of the hyperparameter space based on the performance of the classifier on the entire set of jobs in $\mathcal{D}_{\text{train}}$. The classifiers we experiment with are Naive Bayes (nb) [34], Logistic Regression (lr) [34], Random Forest (rf) [35] and XGBoost (xg) [36]. Finally, we evaluate the performance of each combination of the above objective functions and classifiers on the held out dataset, $\mathcal{D}_{\text{test}}$.

5.1.2 Baselines

Heaviest I/O Users. One baseline is based on selecting the applications for a benchmark in descending order of their I/O activity. For e.g., to pick $K = 10$ applications, we simply rank the applications in descending order of the total time their jobs spent performing I/O tasks and select those that are the top K heaviest I/O users. This baseline, motivated by the work of Huong et al. [8], is meant to test if our selected benchmark applications can be used to predict the I/O performance of this important class of applications as well as a model

trained directly on them. We refer to this baseline as **top-iotime** in our experiments (see Tables 5.2 and 5.3).

Most Frequently Used. Another baseline we created is based on selecting applications in decreasing order of the number of jobs that belongs to the applications. The reason for having this baseline is to compare how well our proposed approach perform against a model trained on applications whose jobs are likely to make up a significant portion of the training and test sets. We refer to this baseline as **top-jobs** in our experiments.

5.1.3 Handling Imbalanced Data

As mentioned in Section 2.1, the class labels based on the throughput of the jobs are highly imbalanced. In particular, according to Table 2.3, less than 9% of the jobs have aggregate throughput greater than 1.1 GB/s in $\mathcal{D}_{\text{train}}$ and around 0.04% of jobs have throughput greater than 26 GB/s. The corresponding numbers for $\mathcal{D}_{\text{test}}$ are roughly 11% and 0.06 %.

To mitigate this problem, we use the SMOTE algorithm [37] to re-sample the feature vectors of the jobs belonging to the selected applications before training our classifiers so that the minority classes with have the same number of examples as the majority class. This step of rectifying the class imbalance has been shown to significantly improve the performance of classifiers when dealing with this kind of data [16]. Note that SMOTE is **not** applied to the test set, $\mathcal{D}_{\text{test}}$, so the test set is still imbalanced. This is because in an actual deployment environment we will not know the true labels of the test examples for us to re-sample them. Also, configurations that result in selected applications whose jobs as a whole do not have at least 6 jobs in each performance intervals are discarded as we will not be able to run SMOTE on the data successfully. These cases therefore do not show up in the tables of results, Table 5.2 and 5.3.

5.1.4 Evaluation Metrics

As the dataset is extremely imbalanced, the usual performance metric, prediction accuracy, will not give a good indication of the performance of our models [16]. The reason is that a model can simply overfit to the majority class and predict all examples as having throughput less than 1.1 GB/s (the majority class) to achieve an accuracy close to 1. Therefore, to

account for the class imbalance, we use the F1 score as our performance metric

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}},$$
$$\text{precision} = \frac{\text{true positive}}{\text{true positive} + \text{false positive}}$$
$$\text{recall} = \frac{\text{true positive}}{\text{true positive} + \text{false negative}}.$$

Details on this metric can be found in any standard machine learning textbook such as Murphy [38]. Furthermore, since we have a multiclass classification problem rather than a binary classification problem that the F_1 metric is designed for, we evaluate our models on the *unweighted* average of the F_1 scores computed for each label (also known as macro F_1) as well as the average of the F_1 scores *weighted* by the number of true instances for each label. The latter metric takes into account the class imbalance better than the former and is the metric we use for selecting the best configurations for each of our models.

5.2 CONTRIBUTION OF INDIVIDUAL OBJECTIVE TERMS TO OVERALL PERFORMANCE

We experiment with different combinations of submodular coverage and diversity inducing functions for selecting benchmark applications to investigate the effects these functions have on the quality of the benchmark applications, measured in terms of the F1 scores of classifiers trained on the jobs of these applications and tested on $\mathcal{D}_{\text{test}}$. The results are given in Tables 5.2 and 5.3. Labels for the objective functions are given in Table 5.1 and details of the submodular functions are given in Section 4.3.

Objective	Description
iotype-cnt	f_{part} with I/O API type partitions
iostrat-cnt	f_{part} with I/O strategy partitions
ioperf-cnt	f_{part} with I/O performance partitions
fac + iotype-cnt	f_{fac} with iotype-cnt
fac + iostrat-cnt	f_{fac} with iostrat-cnt
fac + ioperf-cnt	f_{fac} with ioperf-cnt
fac + all-cnt	f_{fac} with iotype-cnt, iostrat-cnt, and ioperf-cnt
fac + all-rwd	f_{fac} with f_{reward} in place of f_{part} for iotype-cnt, iostrat-cnt, and ioperf-cnt
sat + all-cnt	f_{sat} with iotype-cnt, iostrat-cnt, and ioperf-cnt
sat + all-rwd	f_{sat} with f_{reward} in place of f_{part} for iotype-cnt, iostrat-cnt, and ioperf-cnt

Table 5.1: Labels of submodular objective functions and the functions they correspond to.

Table 5.2 shows the result obtained from training our four classifiers on 25 applications selected by our SFM framework using similarities computed from the Hausdorff distances between applications and their Fisher vector representation described in Chapter 3. The first thing we note is that the best performing configuration is the one that uses the saturated coverage function with all three versions of parition-based diversity functions along with similarities computed from the Fisher vector representation of the applications. This configuration achieved the highest weighted F1 score of 0.9668 and the second highest unweighted score of 0.6848 among all the configurations and baselines we experimented with. This confirms our intuition that selecting applications that are diverse in the sense that their jobs reflect the wide range of I/O behavior on a platform is a good strategy for building an I/O benchmark.

Furthermore, notice that the configurations that use only the single diversity functions — iotype-cnt, iostrat-cut, and ioperf-cnt — tend to perform worse than the other configurations that also use a coverage function such as f_{fac} or f_{sat}. This suggests that a simple strategy that simply picks applications with different I/O behavior from any single category, as one might try as a heuristic, is unlikely to produce a good benchmark compared to using a coverage function to ensure that the applications are also similar to the remaining applications and casting the net wider to find applications with different I/O characteristics.

Lastly, results from the baseline approaches suggests that selecting applications from among those that are heavy I/O users is fairly helpful for building models for predicting I/O performance. Presumably, they already cover a wide spectrum of I/O behavior. However, as we shall discuss in the following section, the performance of this baseline as the number of applications selected varies is not as consistent as those from our SFM framework that uses all the diversity functions, even though it has the highest unweighted F1 score. The other baseline, top-jobs, in general performs worse than the best model from our SFM framework (fac + all-cnt) and even the other baseline, top-iotime.

The list of 25 applications that make up our best benchmark in Table 5.2 selected by our SFM approach is given in Appendix B. The best performing benchmarks selected by the baseline approaches, top-iotime and top-jobs, are listed in Appendix C and D respectively. Based on a quick inspection of Appendix B, we notice that a few of the prominent applications commonly used for I/O benchmarking [19], for e.g., nek5000 and qb, have been selected by our optimization algorithm. At the same time, the names of the selected applications suggests that few, if any of these applications, are simply different versions of each other. This is a pleasant surprise as our SFM algorithm has no knowledge of parallel I/O other than what was encoded in the Darshan extracted statistics and the various I/O category partitions we used to design our diversity functions. This further supports the usefulness of

framing I/O benchmark selection as a submodular function maximization problem.

Furthermore, we note that the top-jobs baseline with weighted F1 score of 0.9357 and unweighted F1 score of 0.6411 performs much worse than our SFM approach and the top-iotime baseline. As we go through the applications selected by top-jobs in Appendix D, we note that it had selected many applications from the same suite of applications, namely, "qb", "qb-30", "qb-1.52.2-dfpt", "qb-1.57.13" etc. Therefore, this benchmark contains a fair amount of "redundancy" and it lost the opportunity to represent some other applications in place of these applications. This suggests that the quality of the benchmark is unlikely to be ideal if we simply select the benchmark based on criteria that are loosely related to the I/O behavior of the applications. In contrast, our approach that searches for a set of applications that possess diverse I/O behavior is able to find applications that are fairly different and resulted in a model that has better predictive capabilities than this simple baseline.

5.3 EFFECT OF NUMBER OF SELECTED BENCHMARK APPLICATIONS ON PREDICTIVE PERFORMANCE

From Table 5.2, we notice that the XGBoost classifier tends to perform much better than the remaining ones, possibly because it is a nonlinear classifier unlike Naive Bayes and Logistic Regression which are linear classifiers, hence it is able to fit much more complicated models using the data. We therefore experiment just with this classifier as we investigate the effect of the number of selected benchmark applications on the trained model's performance in order to simplify our experiments. Results are given in Table 5.3 and Figures 5.1 and 5.2.

One thing we observe is that unsurprisingly, the model's performance tends to improve when the number of applications selected increases but generally only if the jobs are selected with a combinations of all three diversity functions. One reason for this could be that if there are no forces (such as our diversity functions) in place to encourage applications with different I/O behavior from being selected, then the trained model will just overfit to a small set of I/O behavior, especially for those of the minority classes. This will therefore result in poorer performance when measured using a metric such as the F1 score which is sensitive to class imbalance. In particular, we note that the configurations that select applications based on them having different I/O performance (ioperf-cnt) performed worse as the number of applications selected increased (see Figure 5.1, 5.2).

Also, the baseline top-iotime experienced a sharp drop in performance when the number of applications selected increased from from 10 to 15. This sensitivity of the model to the number of applications selected could be due to the applications that are the 11-th to 15-th most heavy I/O users being very different from the remaining applications in some way. This

Objective	F1 (unweighted)				F1 (weighted)			
Hausdorff	nb	lr	rf	xg	nb	lr	rf	xg
iotype-cnt	0.3839	0.4028	0.4331	0.5870	0.7554	0.7958	0.7949	0.9274
iostrat-cnt	0.3063	0.3520	0.4137	0.5649	0.6661	0.7150	0.7476	0.8877
ioperf-cnt	0.2685	0.3597	0.4123	**0.6452**	0.6449	0.7402	0.7569	0.9306
fac + iotype-cnt	0.4974	0.4256	0.4881	0.5774	0.8823	0.8283	0.8537	0.9369
fac + iostrat-cnt	0.3780	0.4873	0.4716	0.5954	0.7769	0.8668	0.8366	0.9501
fac + ioperf-cnt	0.4273	0.4196	0.4623	0.5772	0.8415	0.8146	0.8470	0.9337
fac + all-cnt	0.4462	0.5051	0.5529	0.6269	0.8293	0.9124	0.9295	**0.9603**
fac + all-rwd	0.4463	0.4833	0.4924	0.5792	0.8510	0.8607	0.8802	0.9501
sat + all-cnt	0.4184	0.5178	0.5917	0.6262	0.8064	0.9247	0.9307	0.9571
sat + all-rwd	0.4819	0.4778	0.5220	0.5865	0.8461	0.8728	0.9157	0.9425
Fisher								
iotype-cnt	0.3843	0.4030	0.4463	0.5933	0.7554	0.7963	0.8069	0.9297
iostrat-cnt	0.3063	0.3520	0.4137	0.5649	0.6661	0.7150	0.7476	0.8877
ioperf-cnt	0.2713	0.3614	0.4174	0.5713	0.6503	0.7423	0.7699	0.8992
fac + iotype-cnt	0.4394	0.4963	0.5349	0.5572	0.8185	0.8832	0.9176	0.9320
fac + iostrat-cnt	0.4847	0.5116	0.5461	0.5763	0.8889	0.8757	0.9150	0.9366
fac + ioperf-cnt	0.3014	0.5120	0.4824	0.5101	0.7555	0.8557	0.8228	0.8618
fac + all-cnt	0.4890	0.5117	0.5876	0.6025	0.8728	0.9091	0.9477	0.9516
fac + all-rwd	0.4038	0.4878	0.5176	0.6058	0.8107	0.8972	0.9114	0.9577
sat + all-cnt	0.3963	0.4988	0.6015	**0.6848**	0.8007	0.9077	0.9449	**0.9668**
sat + all-rwd	0.4047	0.5031	0.5478	0.6061	0.8132	0.8899	0.9369	0.9634
Baseline								
top-jobs	0.1092	0.3993	0.4618	0.6411	0.1180	0.7857	0.8069	0.9357
top-iotime	0.1008	0.4672	0.5078	**0.7374**	0.2288	0.8692	0.8737	**0.9610**

Table 5.2: Effect of different objective terms vs baseline approaches for 25 selected benchmark applications. The classifiers used are Naive Bayes (nb), Logistic Regression (lr), Random Forest (rf) and XGBoost (xg).

Objective	F1 (unweighted)				F1 (weighted)			
Hausdorff	10	15	20	25	10	15	20	25
iotype-cnt	-	-	-	0.5870	-	-	-	0.9274
iostrat-cnt	0.5876	0.5658	0.5827	0.5649	0.8726	0.9121	0.9033	0.8877
ioperf-cnt	0.6343	0.6416	0.6263	**0.6452**	0.9315	0.9309	0.9038	0.9306
fac + iotype-cnt	-	-	0.5443	0.5774	-	-	0.9189	0.9369
fac + iostrat-cnt	-	-	0.6022	0.5954	-	-	0.9571	0.9501
fac + ioperf-cnt	-	-	0.5442	0.5772	-	-	0.9204	0.9337
fac + all-cnt	0.5806	0.5802	0.5912	0.6269	0.9522	0.955	0.9532	**0.9603**
fac + all-rwd	0.4322	0.4987	0.5903	0.5792	0.8134	0.911	0.9487	0.9501
sat + all-cnt	0.5584	0.5703	0.6023	0.6262	0.9268	0.9376	0.954	0.9571
sat + all-rwd	0.4322	0.4922	0.5808	0.5865	0.8134	0.8807	0.9462	0.9425
Fisher								
iotype-cnt	-	-	-	0.5933	-	-	-	0.9297
iostrat-cnt	0.5876	0.5658	0.5827	0.5649	0.8726	0.9121	0.9033	0.8877
ioperf-cnt	0.6343	0.6308	0.6351	0.5713	0.9315	0.9276	0.902	0.8992
fac + iotype-cnt	0.5044	0.5265	0.5595	0.5572	0.8958	0.9155	0.9397	0.9320
fac + iostrat-cnt	-	-	0.5761	0.5763	-	-	0.9357	0.9366
fac + ioperf-cnt	0.496	0.5329	0.6091	0.5101	0.8803	0.8618	0.8856	0.8618
fac + all-cnt	0.5683	0.5716	0.6165	0.6025	0.9255	0.9386	0.935	0.9516
fac + all-rwd	0.576	0.6124	0.5942	0.6058	0.9479	0.9583	0.9502	0.9577
sat + all-cnt	0.5981	0.5802	0.6586	**0.6848**	0.9553	0.9519	0.9562	**0.9668**
sat + all-rwd	0.6053	0.6063	0.5988	0.6061	0.9543	0.9588	0.9614	0.9634
Baseline								
top-jobs	0.6165	0.6489	0.6939	0.6411	0.9286	0.9353	0.9332	0.9357
top-iotime	0.6144	0.3708	0.5831	**0.7374**	0.9248	0.8648	0.9435	**0.9610**

Table 5.3: Effect of number of benchmark applications on classifier (XGBoost) performance.

highlights the problem of selecting applications based on a singular criteria such as I/O usage as applications that have similar I/O behavior in some way (in this case heavy I/O usage) could be very similar among themselves. Therefore, selecting them could introduce unnecessary redundancy in the benchmark and lead to a lost opportunity to represent a wider range of applications. In this respect, our proposed SFM framework with multiple functions to encourage selecting applications that are different intuitively will be less susceptible to such problems.

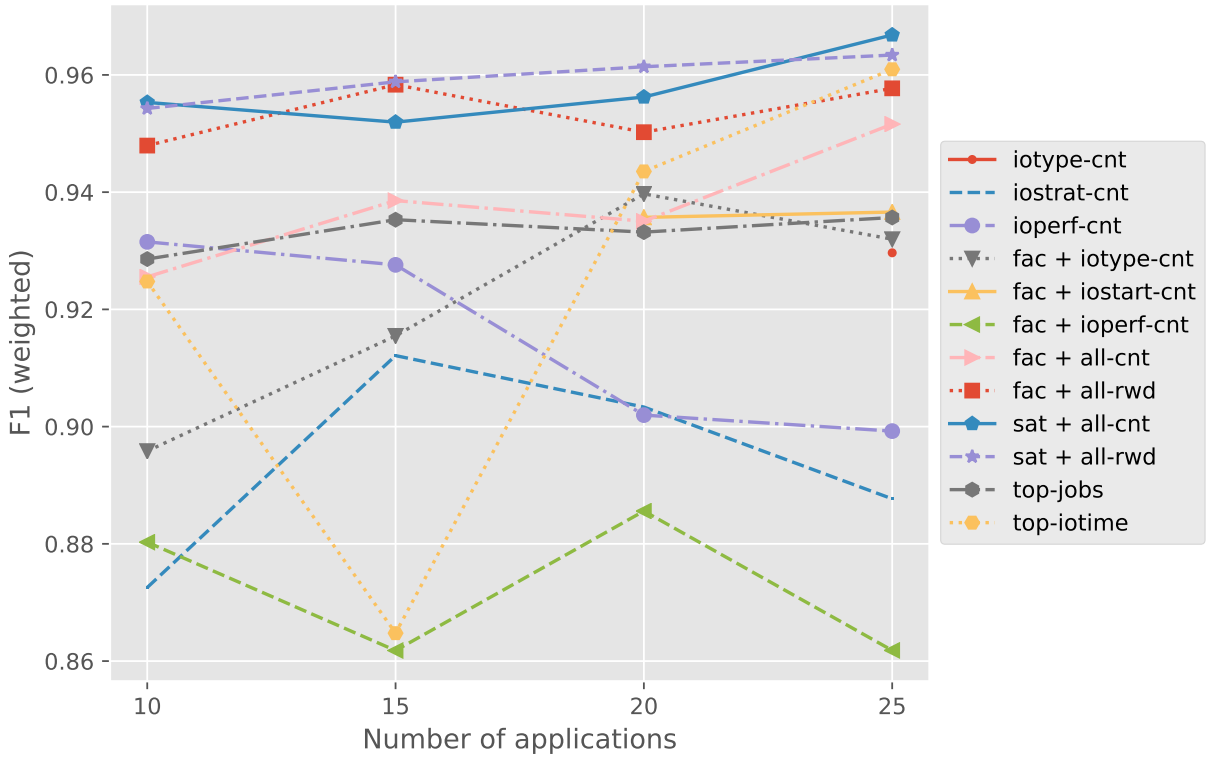

Figure 5.1: Effect of number of benchmark applications selected using Fisher similarities on classifier (XGBoost) performance (weighted F1).

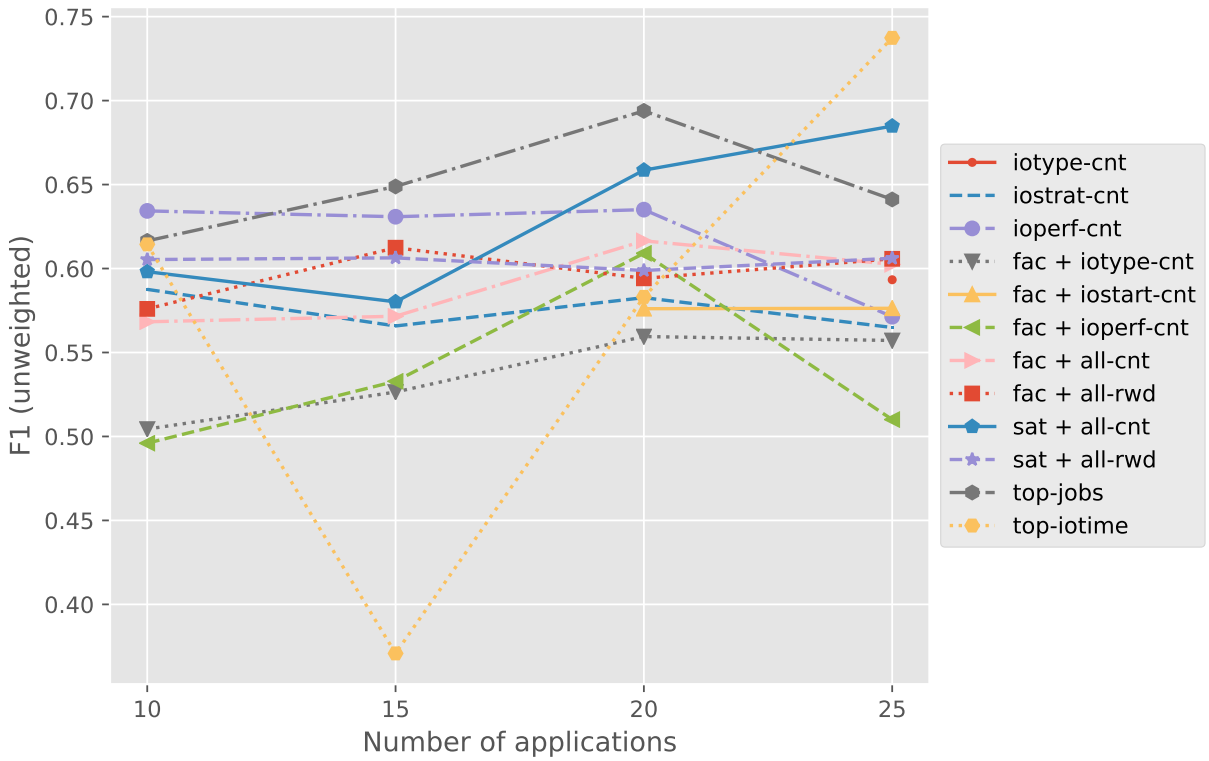

Figure 5.2: Effect of number of benchmark applications selected using Fisher similarities on classifier (XGBoost) performance (unweighted F1).

CHAPTER 6: CONCLUSION

We setup the problem of selecting a good set of applications for a I/O benchmark as a submodular function maximization problem, borrowing ideas from the domain of document summarization and data subset selection. Our optimization algorithm solves for a set of applications that are simultaneously representative of the applications they are chosen from, and also possess diverse I/O behavior as a group. Our experiments show that our algorithm can produce a good I/O benchmark in the sense that classifiers trained on features extracted from Darshan logs of jobs belonging to the applications selected by our algorithm can be used to predict the I/O throughput level of other jobs fairly well.

CHAPTER 7: REFERENCES

[1] H. Shan and J. Shalf, "Using ior to analyze the i/o performance for hpc platforms," 2007.

[2] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–10, 2009.

[3] M. Zingale, "Flash i/o benchmark routine - parallel hdf5," 2018. [Online]. Available: http://www.ucolick.org/~zingale/flash-benchmark-io/

[4] H. Lin, J. Bilmes, and S. Xie, "Graph-based submodular selection for extractive summarization," in *Automatic Speech Recognition & Understanding, 2009. ASRU 2009. IEEE Workshop on*. IEEE, 2009, pp. 381–386.

[5] K. Kirchhoff and J. A. Bilmes, "Submodularity for data selection in machine translation," in *EMNLP*, 2014.

[6] K. Wei, Y. Liu, K. Kirchhoff, C. D. Bartels, and J. A. Bilmes, "Submodular subset selection for large-scale speech training data," *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3311–3315, 2014.

[7] F. Perronnin, J. Sánchez, and T. Mensink, "Improving the fisher kernel for large-scale image classification," in *ECCV*, 2010.

[8] H. Luu, M. Winslett, W. Gropp, R. B. Ross, P. H. Carns, K. Harms, Prabhat, S. Byna, and Y. Yao, "A multiplatform study of i/o behavior on petascale supercomputers," in *HPDC*, 2015.

[9] J. S. Logan, S. Klasky, H. Abbasi, Q. Liu, G. Ostrouchov, M. Parashar, N. Podhorszki, Y. Tian, and M. Wolf, "Understanding i/o performance using i/o skeletal applications," in *Euro-Par*, 2012.

[10] J. Borrill, L. Oliker, J. Shalf, and H. Shan, "Investigation of leading hpc i/o performance using a scientific-application derived benchmark," *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*, pp. 1–12, 2007.

[11] H. Lin and J. A. Bilmes, "A class of submodular functions for document summarization," in *ACL*, 2011.

[12] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. M. VanBriesen, and N. S. Glance, "Cost-effective outbreak detection in networks," in *KDD*, 2007.

[13] H. Lin, "An application of the submodular principal partition to training data subset selection," 2010.

[14] A. Krause and D. Golovin, "Submodular function maximization," in *Tractability*, 2014.

[15] H. Luu, A. Aleyasen, M. Winslett, Y. Mostofi, and K. Peng, "Toward universal analysis and visualization of hpc application input/output behavior."

[16] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, pp. 1263–1284, 2009.

[17] M. Kuhn and K. Johnson, *Applied predictive modeling*. Springer, 2013, vol. 26.

[18] G. E. P. Box, D. R. A. Cox, and R. A. Cox, "An analysis of transformations," 1964.

[19] R. Thakur, "Parallel i/o benchmarks, applications, traces," 2018. [Online]. Available: https://www.mcs.anl.gov/~thakur/pio-benchmarks.html

[20] M. M. Deza and E. Deza, *Encyclopedia of distances*. Springer Science & Business Media, 2012.

[21] J. Sánchez, F. Perronnin, T. Mensink, and J. J. Verbeek, "Image classification with the fisher vector: Theory and practice," *International Journal of Computer Vision*, vol. 105, pp. 222–245, 2013.

[22] A. Ashkan, B. Kveton, S. Berkovsky, and Z. Wen, "Optimal greedy diversity for recommendation," in *IJCAI*, 2015.

[23] S. Fujishige, *Submodular functions and optimization*. Elsevier, 2005, vol. 58.

[24] G. Calinescu, C. Chekuri, M. Pál, and J. Vondrák, "Maximizing a monotone submodular function subject to a matroid constraint," *SIAM J. Comput.*, vol. 40, pp. 1740–1766, 2011.

[25] G. Cornuejols, M. Fisher, and G. L. Nemhauser, "On the uncapacitated location problem," in *Annals of Discrete Mathematics*. Elsevier, 1977, vol. 1, pp. 163–177.

[26] A. M. Frieze, "A cost function property for plant location problems," *Math. Program.*, vol. 7, pp. 245–248, 1974.

[27] U. Feige, V. S. Mirrokni, and J. Vondrák, "Maximizing non-monotone submodular functions," *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*, pp. 461–471, 2007.

[28] J. Lee, V. S. Mirrokni, V. Nagarajan, and M. Sviridenko, "Non-monotone submodular maximization under matroid and knapsack constraints," in *STOC*, 2009.

[29] A. Borodin, H. C. Lee, and Y. Ye, "Max-sum diversification, monotone submodular functions and dynamic updates," *ACM Trans. Algorithms*, vol. 13, pp. 41:1–41:25, 2012.

[30] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions - i," *Math. Program.*, vol. 14, pp. 265–294, 1978.

[31] M. Minoux, "Accelerated greedy algorithms for maximizing submodular set functions," in *Optimization Techniques*, J. Stoer, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 234–243.

[32] J. A. Bilmes, "Lecture notes in submodular functions, optimization, and applications to machine learning, spring quarter, lecture 15," February 2016.

[33] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2012.

[34] T. J. Hastie, R. Tibshirani, and J. H. Friedman, "The elements of statistical learning: data mining, inference, and prediction, 2nd edition," in *Springer series in statistics*, 2009.

[35] L. Breiman, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, 2001.

[36] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *KDD*, 2016.

[37] K. W. Bowyer, N. V. Chawla, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, 2002.

[38] K. P. Murphy, "Machine learning - a probabilistic perspective," in *Adaptive computation and machine learning series*, 2012.

APPENDIX A: DESCRIPTION OF FEATURES

We describe the list of features used to describe a job here.

Table A.1: Description of features.

Features	Description
allshared_mpi_count	# of allshared MPI files.
allshared_mpi_read	Amount of time spent by MPI-IO on reading from files accessed by all processes.
allshared_mpi_readwrite	Amount of time spent by MPI-IO on reading and writing to files accessed by all processes.
allshared_mpi_write	Amount of time spent by MPI-IO on writing to files accessed by all processes.
allshared_posix_count	# of allshared POSIX files.
allshared_posix_read	Amount of time spent by POSIX on reading from files accessed by all processes.
allshared_posix_readwrite	Amount of time spent by POSIX on reading and writing to files accessed by all processes.
allshared_posix_write	Amount of time spent by POSIX on writing to files accessed by all processes.
global_meta	Time spent on meta calls for files opened by all processes.
local_meta	Time spent on meta calls for files opened by 1 process.
nprocs	# of processes.
partshared_mpi_count	# partshared MPI files.
partshared_mpi_read	Amount of time spent by MPI-IO on reading from files accessed by a subset of processes.
partshared_mpi_readwrite	Amount of time spent by MPI-IO on reading and writing to files accessed by a subset of processes.
partshared_mpi_write	Amount of time spent by MPI-IO on writing to files accessed by a subset of processes.
partshared_posix_count	# of partshared POSIX files.

Table A.1 Continued: Description of features.

Features	Description
partshared_posix_read	Amount of time spent by POSIX on reading from files accessed by a subset of processes.
partshared_posix_readwrite	Amount of time spent by POSIX on reading and writing to files accessed by a subset of processes.
partshared_posix_write	Amount of time spent by POSIX on writing to files accessed by a subset of processes.
read_only_count	# of files that are read only.
read_only_max_offset	Maximum offset among files that were read.
read_only_size	Size of read only operation.
read_write_count	# of files that are write only.
read_write_max_offset	Maximum offset among files that were read and written to.
read_write_size	Size of read write operation.
runtime	Runtime of job in seconds.
shared_max_offset	Maximum offset among shared files.
shared_size	Size of I/O operations on files accessed by more than one rank.
meta_io_ratio	Ratio of time spent in meta calls over time spent moving data.
total_bytes	Total bytes transferred by job.
total_max_offset	Sum of maximum offset among files.
unique_max_offset	Maximum offset among files opened by unique process.
unique_mpi_count	# of unique MPI files.
unique_mpi_read	Amount of time spent by MPI-IO on reading from files accessed by only one process.
unique_mpi_readwrite	Amount of time spent by MPI-IO on reading and writing from files accessed by only one process.
unique_mpi_write	Amount of time spent by MPI-IO on writing to files accessed by only one process.

Table A.1 Continued: Description of features.

Features	Description
unique_posix_count	# of unique POSIX files.
unique_posix_read	mount of time spent by POSIX on reading and writing from files accessed by only one process.
unique_posix_readwrite	Amount of time spent by POSIX on reading and writing to files accessed by only one process.
unique_posix_write	Amount of time spent by POSIX on writing to files accessed by only one process.
unique_size	Size of files opened by unique process.
write_only_count	# of write only files.
write_only_max_offset	Maximum offset among files that were written to.
write_only_size	Size of writes.

Table A.1: Description of features.

APPENDIX B: BEST PERFORMING BENCHMARK SELECTED USING SFM

Application	# Jobs
CHANNEL_DNS	97
FlexPepDocking.staticmpi.linuxxlcdebug	11
Solver_CharlesX	54
adios_global	17
agg_sparse.x	11
chris.exe	21
ddt_2d	33
ex2_dif.out	24
ker_GWL_v3_noRho_sta	284
ks_spectrum_hisq-7.7.10-10.qop.xl.dp	256
nek5000	740
openmc-blocking	81
pmergesort.optimized	13
pmergesort.verify	13
qb	3174
qmcapp	137
readInputFile	20
rt-response-time-test.exe	332
sparse-data-writer	28
spinspin	36
test	61
test-PIDX-writer	21
vida.exe	481
x.adlb.12c-11_states-pn	258
xspin1	98

Table B.1: Best performing 25 application benchmark selected using SFM for XGBoost classifier (weighted F1 = 0.9668). SFM weights are: $\lambda = 6.210$, $\beta = 108.263$ and $\gamma = 8.532$. Applications are selected in the order listed.

APPENDIX C: BEST PERFORMING TOP-IOTIME BASELINE BENCHMARK

Application	Total I/O Time (Hours)
vida.exe	103.7047
cp.CL.save.x	78.3444
Solver_CharlesX	74.5181
charles.exe	56.8940
openmc	56.4613
cliff.exe	51.2189
nek5000	50.0532
cp.x	41.3335
ker_GWL_v3_noRho_sta	39.9775
x.mc.4he-pn	38.4781
phParAdapt	34.0489
qb-1.60.4-omp	23.1810
CHANNEL_DNS	19.4462
chris.exe	14.6220
qwalk	13.7667
ddt	13.5540
cesm.exe	13.5439
prepro.exe	11.3841
qb-30	9.8341
aggregate	8.8917
swap	7.6636
shock_tube_3d	6.9638
osiris-2D.e	6.3573
converge-KEVIN	5.9561
qb	5.8520

Table C.1: Best performing 25 application top-iotime baseline benchmark (weighted F1 = 0.961) with the total I/O time of their jobs (in hours). Applications are selected in the order listed.

APPENDIX D: BEST PERFORMING TOP-JOBS BASELINE BENCHMARK

Application	# Jobs
qb-1.52.2-dfpt	114125
qb-1.60.4-omp	11899
x.mc.4he-pn	3202
qb	3174
qb-30	2011
lmp_mira	1914
qb-1.57.13	1369
mpicatnap	1204
nek5000	740
qb-1.60.0	717
pw.x	660
qb-1.57.13-omp	627
cliff.exe	563
moldft	482
vida.exe	481
cp.CL.save.x	425
qb-1e-8	385
rt-response-time-test.exe	332
cp.x	326
ker_GWL_v3_noRho_sta	284
x.adlb.12c-11_states-pn	258
ks_spectrum_hisq-7.7.10-10.qop.xl.dp	256
cp.CL.new.x	251
phParAdapt	249
cesm.exe	233

Table D.1: Best performing 25 application top-jobs baseline benchmark (weighted F1 = 0.9357). Applications are selected in the order listed.