12-2021

Intelligent Resource Prediction for HPC and Scientific Workflows

Benjamin Shealy
btsheal@clemson.edu

Intelligent Resource Prediction for HPC and Scientific Workflows

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

---

by
Benjamin Thomas Sherman Shealy
December 2021

Accepted by:
Dr. Melissa Smith, Committee Chair
Dr. Jon Calhoun
Dr. Frank Feltus
Dr. Adam Hoover

Abstract

Scientific workflows and high-performance computing (HPC) platforms are critically important to modern scientific research. In order to perform scientific experiments at scale, domain scientists must have knowledge and expertise in software and hardware systems that are highly complex and rapidly evolving. While computational expertise will be essential for domain scientists going forward, any tools or practices that reduce this burden for domain scientists will greatly increase the rate of scientific discoveries. One challenge that exists for domain scientists today is knowing the resource usage patterns of an application for the purpose of resource provisioning. A tool that accurately estimates these resource requirements would benefit HPC users in many ways, by reducing job failures and queue times on traditional HPC platforms and reducing costs on cloud computing platforms. To that end, we present Tesseract, a semi-automated tool that predicts resource usage for any application on any computing platform, from historical data, with minimal input from the user. We employ Tesseract to predict runtime, memory usage, and disk usage for a diverse set of scientific workflows, and in particular we show how these resource estimates can prevent under-provisioning. Finally, we leverage this core prediction capability to develop solutions for the related challenges of anomaly detection, cross-platform runtime prediction, and cost prediction.

Dedication

This dissertation is dedicated to four very important women: my mom (Joy), my grandma (Deb), my aunt (Meg), and my wife (Andrena). These women have challenged me and supported me each in their own way at different points in my life. While the men in my life taught be how to be a man, these women taught me how to be a human. I certainly would not have had the courage to pursue a doctorate, and finish it, without them.

Acknowledgments

This dissertation proposal would not have been possible without the tremendous support of my friends, colleagues, and mentors at Clemson University.

First and foremost, I would like to thank Dr. Melissa Smith for training me, both as a teacher and academic advisor. Dr. Smith was willing to take me on as her student, which has since led to a fruitful research partnership and many great memories. I am immensely grateful for her wisdom and encouragement, and the numerous opportunities that she afforded to me. I would like to thank Dr. Alex Feltus, who was like a second advisor to me. Dr. Feltus inspired me to go beyond my comfort zone, both in research and in life. Furthermore, Dr. Feltus trusted me with many of his lab's resources while giving me space to be creative, and I hope that I have benefited his lab as much as it benefited me. The combined mentorship of Dr. Smith and Dr. Feltus has been a truly wonderful and special experience. I would also like to thank Dr. Jon Calhoun and Dr. Adam Hoover for serving on my doctoral committee and instructing me in a number of courses throughout my time at Clemson. This dissertation was greatly improved by their comments and insights.

I owe a great debt to my community of friends and colleagues from the labs of Dr. Smith and Dr. Feltus: Colin, Courtney, Jesse, Eddie, Sufeng, Ankit, Lin, Afshin, Iris, Cole, Benafsh, Reed, Gao, Alli, Ashwin, and many others not listed here. These individuals brought so much joy and inspiration to my time at Clemson.

Finally, I would like to thank the Holcombe Department of Electrical and Computer Engineering for providing the Graduate Assistance in Areas of National Need (GAANN) Fellowship, which allowed me to pursue my research freely, and encouraged me to develop my abilities as a mentor. This work was supported by National Science Foundation Award #1659300 "CC*Data: National Cyberinfrastructure for Scientific Data Analysis at Scale (SciDAS)". Clemson University is acknowledged for generous allotment of compute time on the Palmetto cluster.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Artificial intelligence (AI) is a technological dream of humanity with a long history in both science and fiction. And yet, AI has manifested itself somewhat differently than expected. Whereas the common image of AI is a robot with human-level intelligence, modern AI exists primarily as software tools for humans. Whereas the greatest fear of AI is that it might outsmart, enslave, and destroy humans, the actual danger of modern AI is that humans might use it to outsmart, enslave, and destroy each other. Here are some prominent examples of modern AI systems:

- A vehicle equipped with AI can monitor the driver and alert them if they fall asleep, or watch out for potential collisions, or perform a parallel park [46]
- A medical imaging tool equipped with AI can help a radiologist identify tumors by extracting the images that are most likely to contain tumors [64]
- Recommendation systems based on AI models can curate content from a virtually endless collection of websites, images, videos, music, and products based on user preferences [15]

To summarize, AI technology in its current form is human-centered and data-driven. It is enabled by the fact that humans generate data with every action they take, and it is designed to aid humans in their everyday lives. It is with this paradigm that we consider the potential for AI in scientific research – how can AI be used to aid scientists and engineers in their work?

In the past two decades, scientific research (as well as many other domains) has been transformed by the emergence of three phenomena – machine learning, big data, and high-performance

computing (HPC). Recent advances in machine learning were enabled by (1) the increased availability of computational resources, which reside primarily in HPC and cloud platforms, and (2) the massive amount of training data that is generated by devices, ranging from smartphones to DNA sequencers, that become cheaper and more commonplace each year. As a result, the HPC platform has itself become a scientific instrument that complements traditional lab equipment, as scientists create increasingly complex workflows to extract insights from large datasets. These computational experiments require a great deal of computational expertise, especially as experiments become large, and domain scientists are struggling to close the gaps they have in this kind of knowledge. Thus the *usability* of data-intensive scientific workflows is a major bottleneck to scientific progress today, and while computational expertise will continue to become a necessary skill for domain scientists, anything that simplifies the process of science experiments at scale will ultimately increase the rate of scientific discovery.

One of the greatest challenges with data-intensive computing is knowing the amount of resources that are required, such as CPU-hours, GPU-hours, memory, storage, and I/O bandwidth. Understanding the resource usage patterns of an application is critical when using large-scale computing platforms. Users must request the resources that they need for an experiment, and there are pitfalls to both under-provisioning and over-provisioning. On shared HPC platforms such as university clusters, over-provisioning may increase the time that the job is waiting in the queue, and under-provisioning may cause the job to fail. On cloud platforms such as Amazon Web Services (AWS) and Google Cloud Platform (GCP), there are no queue times or walltime limits because resources are highly available, but there are significant financial risks related to incorrectly provisioning and budgeting for cloud resources. Thus the lack of knowledge about resource usage patterns is a hindrance on HPC platforms and a major setback on cloud platforms. These challenges are multiplied for scientific workflows with multiple steps, where each step may have very different resource requirements. Workflow managers such as Nextflow [14] greatly reduce the burden of executing scientific workflows, but they do not assist users in resource provisioning. An accurate resource prediction tool would confer significant benefits to users simply by addressing the aforementioned problems. Many studies have found that accurate runtime estimates do lead to shorter queue times for users [9, 36, 20, 19, 61]. Having an estimate of runtime would allow the user to better plan future experiments and to tell when a job has been running for too long and should be cancelled. Furthermore, resource estimates could be combined with the cost information of a cloud platform

to estimate the end-to-end cost of running an experiment in the cloud, which would be extremely valuable when applying for a grant.

Resource prediction has received moderate research attention with promising results, but few studies have translated into usable tools for real users. We believe this gap between research and application is largely due to the complexity and heterogeneity of modern computing platforms, as well as the understandable lack of computational expertise among domain scientists. It is very difficult to develop applications that work and perform consistently across many different platforms, and harder still to develop tools to understand the behavior of such applications. The biggest institutions and the biggest software projects can bring on the technical expertise that is required to do this kind of work, but the majority of scientists must rely on themselves and their peers to perform their computational experiments successfully. To that end, we present Tesseract, a tool that provides intelligent resource prediction for any application, on any computing platform, and can be used by experts and non-experts alike. In this dissertation, we demonstrate the use of Tesseract with a number of real scientific workflows on three different computing platforms, and we focus specifically on how Tesseract helps to prevent under-provisioning.

The remainder of this dissertation is organized as follows. Chapter 2 provides background information on machine learning and scientific workflows, as well as a review of current research on resource prediction. Chapter 3 introduces Tesseract, the resource prediction tool, and describes the research and experiments that were done to validate Tesseract. Chapter 4 provides results and related discussion for the experiments described in Chapter 3. Finally, Chapter 5 concludes and describes future directions that are beyond the scope of this work.

Chapter 2

Background and Related Work

In this chapter, we will discuss several topics which provide background for this work. We will discuss machine learning concepts, including the structure and role of datasets, machine learning models, training, and evaluation. We will also discuss scientific workflows, including the general definition of a workflow, workflow management systems, and the emergence of scientific workflows in particular. Finally, we will review the existing literature on resource prediction methods for scientific workflows and HPC applications, including analytical modeling, empirical modeling, and other related efforts.

2.1 Machine Learning

Machine learning is a subset of artificial intelligence (AI) which has experience significant success and growth in the past two decades. While artificial intelligence pertains generally to the ability of machines to perform tasks that are considered "intelligent," machine learning specifies a framework in which a computer algorithm uses data to learn how to perform a task (see also "statistical learning theory"). More formally, a machine learns from some data D to perform some task T if its performance at T, as measured by some performance metric P, improves with D [40]. This definition encapsulates each of the key elements of a machine learning system. That is, to create a machine learning system in practice, we define the task T we would like to perform, we develop a model (the machine), we train the model using a dataset D to perform the task T, and then we evaluate our model using a performance metric P. Figure 2.1 provides a visual description of this

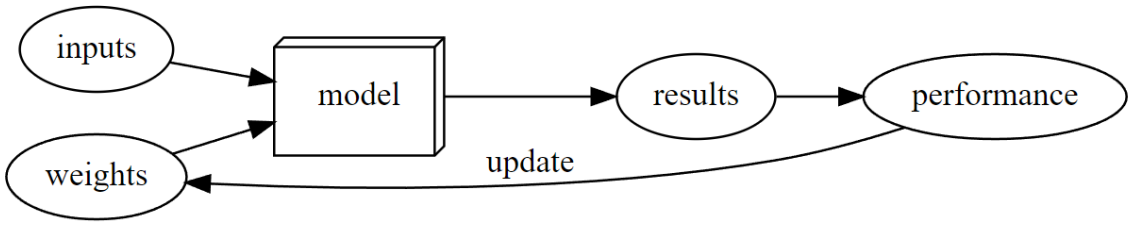


Figure 2.1: High-level diagram of a machine learning system, based on Arthur Samuel's original description and taken from the fastai book [29].

process. In this section, we will discuss each of these elements and how they play an important role in a machine learning system.

AI applications have a tendency to be excluded from the definition of AI over time as they become well established; for example, optical character recognition and any predictive system based on linear models. In contrast, applications such as autonomous driving, natural language understanding, and reinforcement learning agents for video games, are all considered to require AI capabilities. These applications fall under the broader long-term goal of artificial general intelligence (AGI), in which a machine is able to learn any task that a human can. While AGI represents the common view of AI as a distant technological dream, there are already many applications today which use AI, such as the applications used above. Even though optical character recognition and linear regression are generally not considered to be AI, and indeed they existed long before the modern prevalence of machine learning, they are based on the same principles as the most advanced AI systems of today. In fact, linear regression and logistic regression can be understood as two of the simplest illustrations of a machine learning system, as we will soon see.

2.1.1 Models

The *model* is the representation of the "machine" in machine learning. In mathematical terms, any machine learning model can be defined as follows:

$$y = f(x; \theta)$$

In other words, a machine learning model is a function f which maps some input x to an output y. The values x and y are determined by the dataset and the desired task. The distinguishing aspect

of a machine learning model is that it also takes as input a set of parameters θ, which allows the model to be adjusted to suit the desired task. Examples of commonly-used machine learning models include decision trees, k-means clustering, k-nearest neighbors, linear regression, neural networks, principal component analysis, and support vector machines.

2.1.2 Datasets

Data is the foundation of any machine learning task; a machine learning model cannot learn without data. Datasets contain the inputs and outputs that are used to train and evaluate machine learning models. Datasets are typically classified as either structured or unstructured. Structured data is essentially tabular data, which can be organized into a spreadsheet or relational database, while unstructured data is everything else.

An example of a structured dataset is the Iris dataset, which contains 150 samples from three different species of Iris flowers, where each sample consists of four measurements of an individual flower. This dataset can be represented as a table with 150 rows and 5 columns, and it is often used as an example of a classification task – predict the species of an Iris flower from the four measurements.

An example of an unstructured dataset is the ImageNet database, which contains more than 14 million images across more than 200,000 categories. This database is one of the most popular benchmarks for computer vision research, but it is not very usable in its raw form. Instead, it is typical to extract a subset of images for a small set of categories and resize them to a uniform size so that they can be easily fed into a machine learning model.

These examples highlight the fact that it is generally much easier to train a machine learning algorithm with structured data. Unstructured data, in some cases, can be processed using more advanced machine learning techniques, however the more common approach is to transform the unstructured data into a structured format via feature extraction. Thus the choice of feature extraction techniques tend to be critically important in designing a machine learning system.

2.1.3 Training

Training is the process of tuning the parameter set θ of a model to learn a mapping $f : x \to y$ which best fits the dataset. The majority of machine learning models are trained using a *loss*

function, a function that quantifies the amount of error (or loss) in the model's predictions, and an *optimizer*, an algorithm that iteratively updates the parameter set θ in order to minimize the loss function. In essence, the loss function defines *what* the goal is, and the optimizer defines *how* to achieve the goal. From this perspective it is clear that the desired task or objective should translate directly to the loss function, since the loss function directly determines what the model will learn to do. Some examples of loss functions include mean squared error, mean absolute error, and cross-entropy. Some examples of optimizers include stochastic gradient descent (SGD), Adagrad, and Adam.

When the dataset has labels, the training process is *supervised*, which means that the loss function can compare the model's predictions to the labels (or the *ground truth*) in order to quantify the error. When the dataset does not have labels, the training process is *unsupervised*, which means that the loss function has no ground truth with which to evaluate the model's predictions and must instead use other metrics. Supervised learning problems are generally divided into *regression* problems and *classification* problems. Regression problems involve predicting a continuous variable, while classification problems involve predicting a discrete variable. Some examples of a supervised learning problem include predicting the price of a house (regression) or deciding whether or not an email is spam (classification). An example of an unsupervised learning problem is identifying clusters in a dataset of size measurements in order to determine the most optimal sizes for a T-shirt or other article of clothing.

Given a labeled dataset, it is possible to use supervised or unsupervised training. For example, a logistic regression model could be trained to classify the labeled data, or a k-means clustering model could be used to cluster the data, while using the labels after the fact to validate the clustering results. The choice depends on the relative usefulness of the labels; while labels provide a tangible performance metric, they can also introduce bias if they are unreliable, in which case it may be better to simply allow the data to "speak for itself." Additionally, any regression problem can be transformed into a classification problem, and vice versa, simply by modifying the output variable and the training objective accordingly. For example, the aforementioned regression problem for housing prices could be transformed into a classification problem by defining a set of price ranges and classifying which price range or "bin" a house falls into.

Prior to training, the dataset is split into two subsets, the training set and test set. The test set is withheld during training as it will be used to evaluate the trained model. The training

process itself is an iterative process of using the model to make predictions on the training data, using the loss function to compute the error, and then using the optimizer to update the model in order to minimize the error.

The training process described here does not hold for all machine learning algorithms. In reality, training manifests in slightly different ways for different algorithms. For some algorithms, such as principal component analysis, the model parameters can be computed directly rather than iteratively. Similarly, for k-nearest neighbors there is really no training process because the model simply uses the training data to make predictions. Our description of training is most accurate for neural networks, however we believe that this description forms a good baseline from which to understand other machine learning models.

2.1.4 Evaluation

Evaluation is the process of measuring how well a trained model performs on unseen data. During this phase, the test set which was withheld during training is fed into the model, the model's predictions are compared to the ground truth, and an evaluation metric is used to give the model a score. Some common metrics include accuracy (for classification models) and relative error (for regression models). Additionally, any loss function can also be used as an evaluation metric.

The purpose of the evaluation phase is to determine whether the trained model has learned a general pattern from the training data. For example, if a model was trained on a subset of points on a curve, we can evaluate whether the model learned the entire curve by giving it inputs that it has not seen before and observing whether the output matches the ground truth. The ability of a model to learn a general pattern from relatively few examples is called *generalization*, and it is a primary goal of any machine learning system. Therefore it is of critical importance that the test set is not used in any way to train the model, so that it can be used as unseen data during evaluation.

2.1.5 An Example: Logistic Regression

One of the simplest illustrations of a machine learning algorithm is logistic regression (see Figure 2.2). The logistic regression model predicts an output y from an input x by taking a weighted sum of the inputs:

$$y = \sigma(w^T x + b), \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$

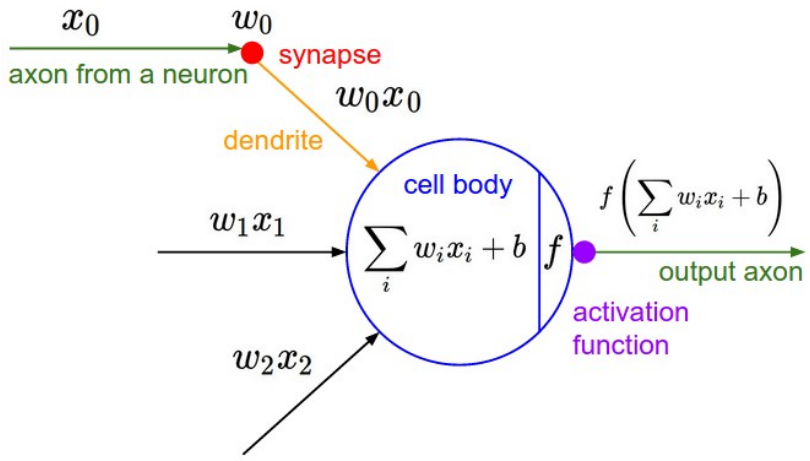
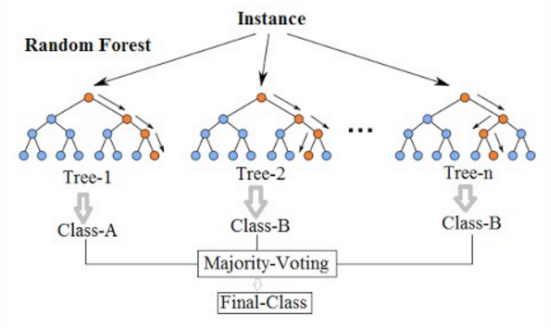
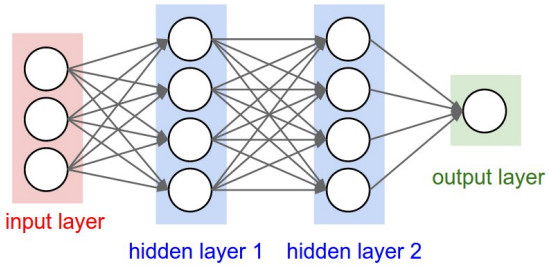


Figure 2.2: Depiction of a logistic regression model, which is also equivalent to a single artificial neuron, taken from Stanford's CS231n course [21].



(a) Random Forest [32]



(b) Neural Network [21]

Figure 2.3: Diagrams of two widely-used machine learning models, random forest and neural network.

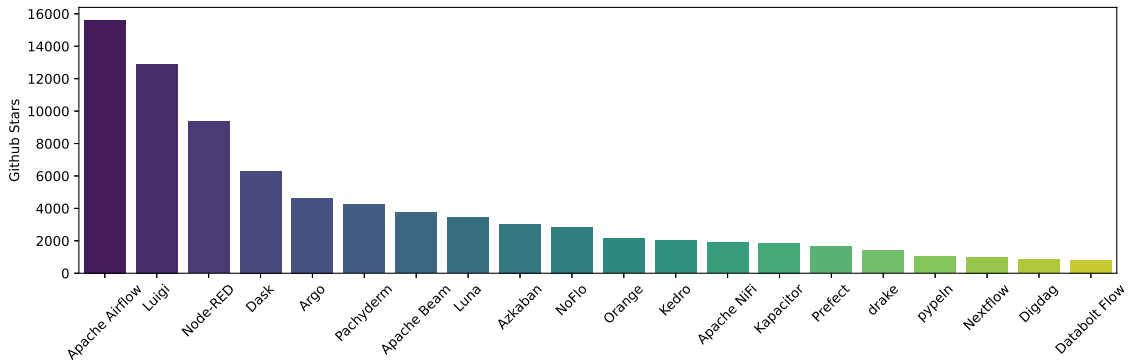


Figure 2.4: Plot of the 20 most popular workflow managers in terms of Github Stars (collected January 2020).

In this model, the parameter set θ consists of the weights w and biases b. Since this model is differentiable, it can be trained by gradient descent or any other gradient-based optimization algorithms. In fact, the globally optimum parameter set for logistic regression can be computed directly for a given dataset, but the same is not true for most other machine learning algorithm. Despite the name, logistic regression is actually a classification model. In fact, logistic regression may be understood as the classification analog to linear regression:

$$y = w^T x + b$$

Logistic regression is also equivalent to a single artificial neuron with sigmoid activation. Similarly, linear regression is equivalent to an artificial neuron with linear activation.

2.2 Scientific Workflows

A *workflow* is a set of tasks arranged as a directed acyclic graph (also called the *task graph*), in which an edge represents a dependency of one task by another. A *workflow management system* (WMS) is an application that executes and monitors workflows. In other words, a WMS maps a task graph to an execution environment, which could be a desktop or an HPC cluster or a cloud platform, for example.

The history of workflow management systems is complex because workflows can be found in many fields, including:

- Business processes (purchase orders, safety operations, etc)
- Build automation (Make)
- Package management (apt, yum, etc)
- Continuous integration, deployment, and delivery (CI/CD/CD)
- Modeling computer hardware (Petri nets)
- Data science, machine learning (Dask, TensorFlow)
- Computational sciences (bioinformatics, materials science, neuroscience, physics, etc)

Many different workflow management systems have emerged within each domain, to suit that domain. As a result, there are countless workflow managers to choose from, and each provides a different set of features and capabilities (Figure 2.4). For example, many basic command-line tools such as `apt` and `make` can be considered workflow managers, because they execute tasks with dependencies: in the case of `apt`, installing software packages; in the case of `make`, compiling and linking code into applications and libraries. It is possible to implement other types of computational workflows in a Makefile, and in fact there are many workflow managers inspired by `make`, including drake, Snakemake, and Biomake.

In this work we will focus on the domain of scientific workflows, and in particular we will focus on the Nextflow workflow manager [14]. Nextflow provides broad support for many key elements of scientific workflows; it is language-agnostic, portable across many execution environments, and highly scalable, and it supports many other useful features such as caching, containerization, and pipeline sharing for reproducibility. All of the scientific workflows used in this work are implemented as Nextflow pipelines. More importantly, Nextflow automatically collects an execution trace of each task that is executed in a workflow run, which makes it much easier to integrate resource prediction with workflow execution.

Within the realm of bioinformatics, two of the most commonly-used workflow managers are Pegasus [13] and Galaxy [23]. Pegasus enables large experiments to be run at scale on grid computing platforms such as the Open Science Grid. The Galaxy Project is a platform for running bioinformatics workflows and publishing results. These two workflow managers lie on opposite ends of a trade-off between control and usability. Pegasus is highly scalable and provides a fine-grained

level of control, but it is very complicated to write Pegasus workflows, especially for domain scientists. Galaxy is designed for domain scientists as it provides a graphical interface and turnkey workflows for the most common types of analyses, but it lacks scalability and control. We have found that Nextflow eliminates this trade-off by providing an easy way to write workflows while also providing a full suite of features that can be gradually incorporated into a workflow to meet more advanced computational needs.

2.3 Resource Prediction Methods

We define *resource prediction* as the task of estimating the usage of some resource by an application before it is run. The resources which we are primarily interested in are runtime, memory usage, and disk usage. A closely related task is *performance prediction* or *performance modeling*, which refers to measuring either how much or how well an application uses a resource. Resource prediction and performance prediction have been subjects of research interest for many years and for many different reasons. The studies described in this section each approached resource prediction with slightly different goals and constraints, and as such they each contribute different kernels of knowledge and insight to this work.

2.3.1 Analytical Modeling

Arguably the most basic approach to performance prediction is to consider the algorithmic complexity. For example, multiplying an $m \times n$ matrix by an $n \times p$ matrix requires $mnp + m(n-1)p$ floating-point operations and $mn + np + mp$ memory elements, assuming a naive implementation. We could use these formulas, in combination with knowledge of the underlying hardware capabilities, to develop models of runtime and memory performance for any application and architecture. This approach is known as *analytical modeling* and it broadly characterizes most of the early approaches to performance prediction, especially in the domain of high-performance computing. Many studies [17, 22, 34, 8, 33] developed models of particular applications and architectures in an attempt to capture all of the intricacies of these systems. Analytical modeling can yield very accurate predictions, but a new model must be developed for every new application or architecture, each of which requires expert knowledge. As a result, analytical modeling is not suitable for non-expert users such as domain scientists who want to predict the resource usage of their applications. Additionally,

many of these efforts relied on simulations to validate a given performance model, which can make these models computationally expensive to develop.

2.3.2 The Convolution Approach

A more portable prediction framework was developed by Snavely *et al.* [59, 58] and Carrington *et al.* [7, 6]. Their approach, which they refer to as the "convolution" approach, is to convolve an application signature (determined by profiling tools) with a machine signature (determined by micro-benchmarks) to predict runtime. In particular, these authors focused on starting with a simple model and only adding complexity as necessary. This framework was a significant improvement, as it removed the dependence on analytical models and still achieved an acceptably low relative error (defined by the authors as 20% or less). However, the convolutional approach still requires expert knowledge in the use of profiling tools and machine benchmarks, and it depends on detailed application traces which are expensive to acquire. Additionally, the convolution approach makes some assumptions about the factors that determine application performance based on the particular HPC applications that were used in the aforementioned studies; as such, this framework may not transfer well or may require significant adjustments as computing platforms become more complex. For example, the convolution approach does not incorporate any information about accelerators such as GPUs and FPGAs because such accelerators were not prevalent in HPC platforms when this approach was developed. To become usable for modeling GPU applications, the application signatures and machine signatures used in the convolution approach would need to be augmented with appropriate GPU profiling tools and GPU benchmarks. In this sense, the convolution approach is still somewhat dependent on and limited by the particular applications and architectures for which it was designed. So while this approach significantly reduces the amount of expert knowledge required compared to previous work, it still requires too much expertise to be reasonably accessible to non-expert users.

2.3.3 Machine Learning Approaches

Many studies have explored the use of machine learning models to predict resource usage. Ipek *et al.* [31], one of the earliest studies of its kind, trained a multilayer perceptron (MLP) to predict the runtime of SMG2000 with 5-7% test error. In particular, the authors found that allocating

an extra core to handle OS-related tasks and modifying the training process to minimize relative error instead of absolute error mitigated the effect of noise and greatly reduced the test error. Li *et al.* [36] modified an existing scheduling algorithm to use runtime estimates generated by a neural network for a number of real and synthetic R applications. The authors observed similarly low error (2-6%) as well as a significant improvement in the throughput of the scheduler (20-40%). Matsunaga *et al.* [39] trained a number of machine learning algorithms to predict the runtime, memory usage, and disk usage of two bioinformatics applications, BLAST and RAxML. They evaluate several classical algorithms as well as a custom algorithm called PQR2, a type of regression tree that can use any regression model at each leaf node. In addition to application-specific input features, the authors use simple benchmarks to measure CPU speed, memory speed, and disk I/O speed. They visualize the impact of these platform characteristics on the overall trend in runtime for their selected applications. Rodrigues *et al.* [52] used similar methods to develop an online memory usage prediction tool on an IBM POWER8 cluster. Interestingly, these authors included several textual features, such as user ID, working directory, and the executed command, as categorical inputs, however they did not specify how much these features contributed to prediction accuracy. Additionally, these authors used a database of jobs executed on their cluster over a period of time, rather than focusing on a few specific applications. Pallipuram *et al.* [47] proposed a regression-based framework for predicting the runtime of distributed-GPU synchronous iterative algorithms (SIAs). The authors combined FLOP and byte counts for each application with performance metrics obtained from micro-benchmarks to predict runtime, similar to the convolution approach. The regression-based framework consisted of several linear regression models designed specifically to model SIAs by treating the computation and communication phases separately. The authors claim that their approach simplifies performance prediction by removing any dependence on details of the GPU microarchitecture. This framework was developed to predict the performance of an application on different architectures and thereby recommend the best architecture to focus on. While we are tackling a slightly different problem and using different techniques, our methodology inherits the name Tesseract from the work done by these authors, and in that way is a successor of it.

While all of the examples mentioned so far have focused on individual applications, many studies have also attempted to predict resource usage of entire workflows. Nadeem *et al.* [44] proposed a local learning framework that predicts total workflow runtime based on the past runtime of similar workflows. The authors defined a custom distance measure for this purpose which considers

attributes such as workflow structure and the execution environment. More recent work by Nadeem *et al.* [43] included similar experiments at a larger scale with an ensemble of three machine learning models. Pietri *et al.* [50] developed an algorithm to estimate the makespan (total runtime) of a workflow based on the structure of the task graph and the runtime of individual tasks. Their work assumes that task runtime can already be predicted accurately. They were able to estimate makespan with under 20% error on three scientific workflows that were executed on AWS. Da Silva *et al.* [12] profiled several Pegasus workflows with the Kickstart profiling tool, explored the relationship between input parameters and the profiling results, and developed a model that uses density-based clustering and regression trees to identify correlations between input parameters in order to predict runtime, memory usage, and disk usage. Additionally, the authors integrated their prediction model into an online prediction tool which continuously estimates the resource usage of tasks in a workflow, updating its predictions with the real usage patterns of tasks as they finish. They provide a clear methodology towards automatic resource prediction; we aim to improve upon their work by using a simpler prediction model, using confidence intervals to address under-provisioning, and incorporating environment-related input features in order to predict runtime across different platforms.

Many other examples of empirical approaches to resource prediction exist in the literature. Miu *et al.* [41] trained a number of different machine learning models to predict the training time of the C4.5 decision tree builder. Monge *et al.* [42] predicted the runtime of several gene expression analysis workflows with an ensemble of M5P regression trees, trained using a combination of data-related and environment-related input features. Fan *et al.* [16] predicted job runtimes on an HPC platform with an additional objective of minimizing underestimation. Pham *et al.* [49] and Hilman *et al.* [28] separately predicted task runtimes for several cloud-based scientific workflows using a machine learning model that is continuously trained on input features and profiling data. Wyatt *et al.* [70] predicted runtime and I/O usage by converting job scripts into image-like "character maps" and feeding them into a convolutional neural network (CNN). A more in-depth review of resource prediction efforts is given by Amiri *et al.* [3].

Empirical approaches to resource prediction are powerful because they do not require any knowledge about the internal details of the application or workflow. In many cases, only those features that are readily available and easy to obtain, such as input parameters and input data characteristics, are used. Runtime predictions can be improved further by including performance metrics based on simple benchmarks (as shown by Matsunaga *et al.*). The main drawback of empir-

ical models is that training data is expensive to acquire. Machine learning models typically need at least $\mathcal{O}(100)$ samples, which means that the application-under-test must be run many times in order to train a sufficiently accurate model. The high cost of acquiring training data can be mitigated by using historical job information, as in the study by Rodrigues *et al.*, since those jobs would have been run anyway. Thus, machine learning is a promising approach to resource prediction that is generic and requires minimal computational expertise.

2.3.4 Related Problems

There are several research problems related to the core task of predicting resource usage, such as anomaly detection, cross-platform runtime prediction, and cost prediction. Here we describe some of the previous work in these topics as they relate to this dissertation.

Uncertainty Quantification. In many supervised learning problems it is very helpful to have some measure of a model's uncertainty. In the context of a regression problem such as resource prediction, an uncertainty measure would be a confidence interval around the predicted value. Uncertainty quantification is a large and diverse field because of its relevance to many problems in science and engineering, so here we will only describe a few relevant techniques. A gradient boosting model can be trained with a quantile loss, which allows it to estimate a particular quantile of the target distribution. A confidence interval can then be obtained by training two models on the desired upper and lower bounds (e.g. 5-th and 95-th percentiles). For random forests, the jackknife and infinitesimal jackknife [66] can be used to obtain a bias-adjusted variance estimate based on the predictions of the individual decision trees in the random forest. Gal *et al.* [18] demonstrated that a neural network with dropout [60] also approximates a deep Gaussian process and thereby provides an estimate of model uncertainty. A confidence interval can be obtained by performing inference multiple times on the same input, with dropout enabled during inference, and taking the mean and variance of the predictions. Lastly, there also exist generic approaches such as the framework proposed by Lei *et al.* [35], which uses conformal inference to construct prediction intervals (i.e. without a point prediction) from an arbitrary regression model.

Anomaly Detection. Anomaly detection is the problem of identifying unusual, or anomalous, outcomes. In the context of scientific workflows, anomalies can occur in the input (e.g. bad input configuration or input data), the execution (e.g. unusually short or long runtime, unusually low or high memory usage), or the output (unusually small or large, or otherwise invalid, output

data). An anomaly detection system would be able to identify these anomalies when they occur and alert a user for further review. Mandal *et al.* [37] developed an end-to-end anomaly detection framework for scientific workflows, which integrated performance modeling, workflow and infrastructure monitoring, and a suite of different anomaly detection techniques tailored to each aspect of the performance data. Tuncer *et al.* [65] developed a similar system, which used basic machine learning models to detect and classify different types of anomalies based on system-level performance metrics. A related but interesting use case in industry is LandingLens [2], a machine learning operations (MLOps) platform developed by Landing AI for visual inspection problems in manufacturing. Whereas much of AI research has focused on fine-tuning models on fixed datasets (i.e. benchmarks), LandingLens is built around improving a dataset while using a fixed model architecture. This approach, known as "data-centric AI", is one way to distinguish between an incorrect model prediction and a data anomaly.

Cross-Platform Runtime Prediction. One challenge that persists with every approach to runtime prediction is predicting runtime across different platforms. Earlier studies such as Snavely *et al.* [59] sought to address this problem in order to predict the runtime of applications on platforms that had not been built yet. Yang *et al.* [71] developed an approach in which they combined full executions on a reference platform with partial executions on a target platform, and used the relative performance between the two platforms to infer runtime on the target platform. Their approach was designed specifically for bulk-synchronous parallel (BSP) applications, and does not generalize easily to other types of applications. Matsunaga *et al.* [39] incorporated performance metrics as input features to a runtime prediction model, in order to predict runtime on a heterogeneous computing platform that contains nodes with different hardware resources. This approach works for any application and is easily extended to predict runtime across different platforms. Marathe *et al.* [38] developed an approach based on transfer learning to predict runtime of large application runs. They collected performance data for many small runs and a few large runs, trained two separate neural networks on these two "domains", and used the feature vectors from each network to train a third network that can perform inference over both domains.

Cost Prediction. In academic research there are primarily two ways to perform large computational experiments. Researchers typically have free access to an institutional cluster that is shared among many researchers. When a computational experiment is too large for this cluster, or when the cluster is highly congested by many users, researchers can instead run their experiments

in the cloud. Cloud computing platforms such as Amazon Web Services (AWS) and Google Cloud Platform (GCP) provide virtually limitless capacity, but they are not free to use and the user interface is much more complex. These factors in combination mean that experiments in the cloud can become very expensive if they are not set up properly. Understanding the resource requirements of an experiment is critical to performing experiments successfully in the cloud. In the ideal scenario, researchers would use their institutional cluster to run and debug small and medium-sized experiments, and then use a cloud platform to run very large experiments. Resource estimates could be used to predict the cost of an experiment, which could in turn be used to justify grant proposals for experiments that are likely to drive significant science discoveries. Wolski *et al.* [69] developed a time-series analysis tool to predict an appropriate bid price for spot instances on AWS. This tool allowed the authors to obtain the reliability of reserved instances at a greatly reduced cost. Rosa *et al.* [53] predicted runtime and cost of two gene analysis workflows in the cloud using linear regression and a metaheuristic technique called GRASP. They evaluated a variety of scenarios by either minimizing cost, maximizing performance, or minimizing runtime. Few studies aside from these two have explored cost prediction of scientific workflows in the cloud.

2.4 Summary

Machine learning techniques have been applied to many fields within engineering and science for their ability to learn from and make predictions on data. At the same time, HPC platforms have become critical instruments for scientific research and discovery, on par with traditional lab instruments. However, these computing platforms are only as effective as the scientists who use them. Given that domain scientists are not computational experts, there is a great need for intelligent tools to minimize the amount of software engineering and decision-making that must be done by scientists in order to run their computational experiments. In particular, knowing the resource usage of an application is extremely helpful both for minimizing queue times and job failures on HPC platforms and estimating the cost of experiments on cloud platforms. Resource prediction has been explored from many perspectives over the last few decades, and many approaches have been proposed. Our work aims to synthesize the best aspects of these prior studies into a methodology for resource prediction that is reliably accurate, generic across application and computing platform, and easy to use for domain scientists.

Chapter 3

Research Design and Experiments

In this chapter we describe our research design and the experiments that we performed. First, we describe the scientific workflows that were used to develop and evaluate our resource prediction tool. Second, we introduce the resource prediction tool, which we call Tesseract. We describe the design of Tesseract, the relevant constraints, and how it would be used by a domain scientist. Lastly, we describe the experiments that were performed to demonstrate and validate the various capabilities of Tesseract, including resource prediction, anomaly detection, cross-platform runtime prediction, and cost prediction.

3.1 Workflow Suite

In this section we describe the scientific workflows that were used in this dissertation. We selected five scientific workflows that are used frequently by fellow researchers at our institution. These workflows played an important role in the development of Tesseract, including which input features to include, how to construct the trace datasets, and which models and preprocessing steps to use. Using these workflows for development and evaluation will ensure that Tesseract is easy to use for domain scientists, such as our peers, who do not have deep expertise in performance engineering. Our hope is that our peers will be able to benefit immediately from this work in their own research.

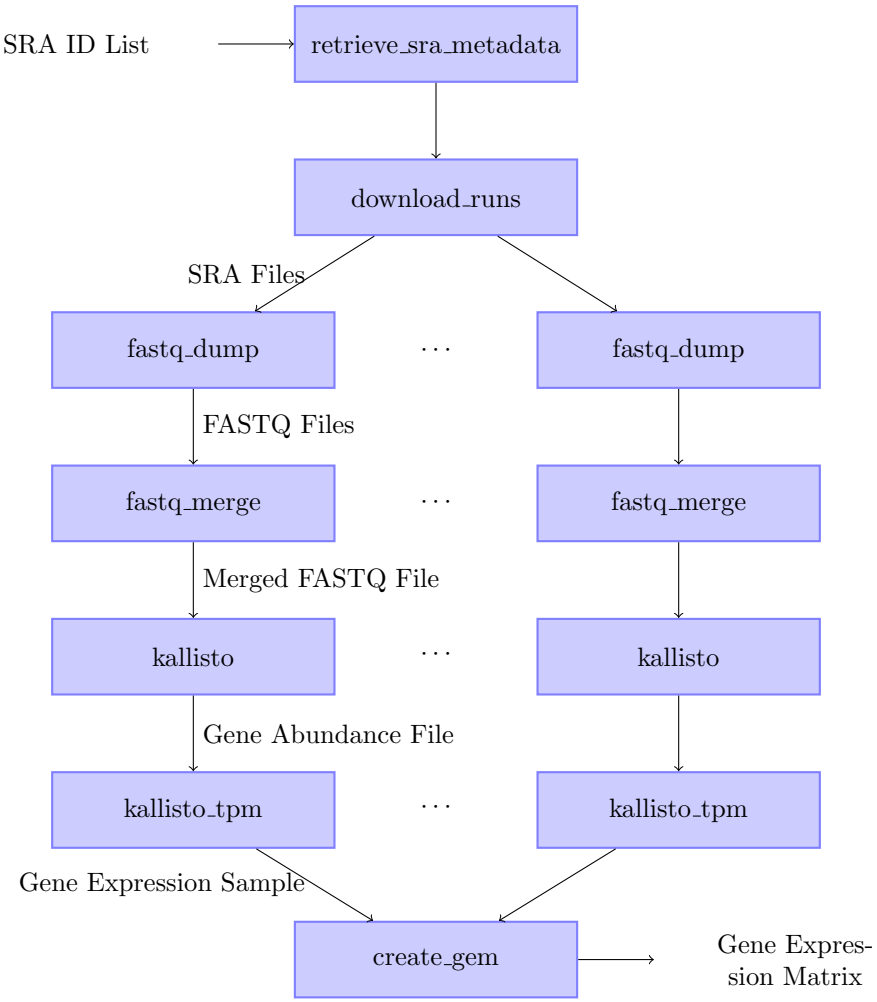


Figure 3.1: GEMmaker workflow diagram

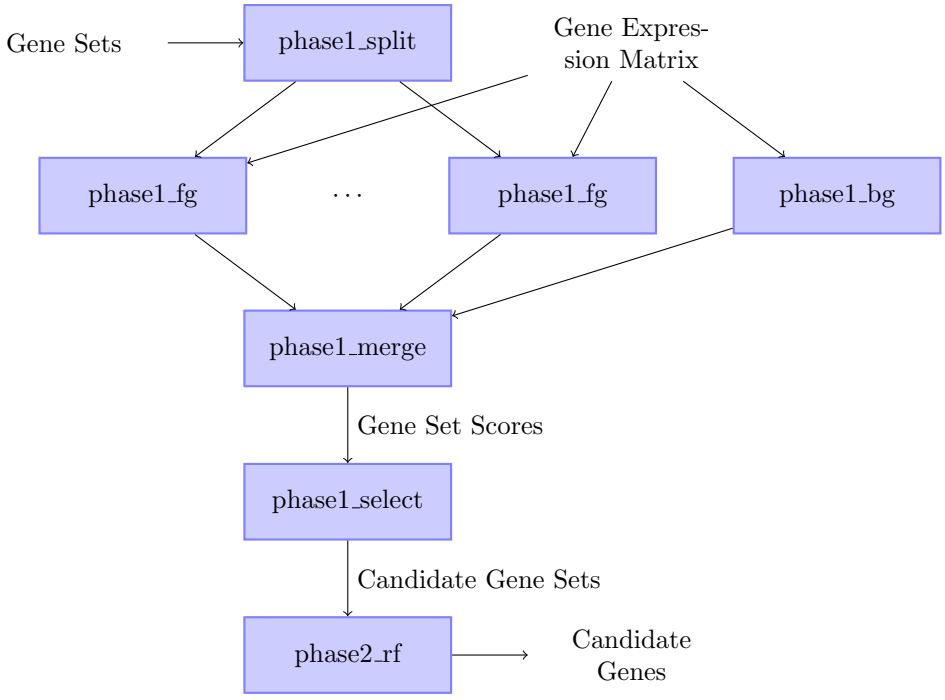


Figure 3.2: Gene Oracle workflow diagram

3.1.1 GEMmaker

GEMmaker is a bioinformatics pipeline for constructing gene expression matrices (GEMs) from Illumina RNA-seq data. Although many such RNA-seq pipelines already exist, GEMmaker stands out in its ability to scale [27]. GEMmaker can process up to thousands of RNA-seq samples in parallel, and it maintains a minimal storage footprint by automatically removing intermediate data. In its largest recorded experiment, GEMmaker processed 14,441 Arabidopsis datasets, consisting of nearly 27,000 individual runs, in a single run on Clemson University's Palmetto cluster, which took 40 days and consumed 358,732 CPU-hours. Results from GEMmaker are typically used in differential gene expression (DGE), such as DESeq2 or TSPG, and gene co-expression network (GCN) analyses, such as KINC.

3.1.2 Gene Oracle

Gene Oracle is a pipeline for identifying biologically significant biomarkers, or "candidate genes," from a high-dimensional gene expression matrix (GEM) [63]. Gene Oracle uses machine learning models to evaluate the "classification potential" of user-defined gene sets, and for those sets

Figure 3.3: HemeLB workflow diagram

which perform the best it uses machine learning to select the most salient genes within each set. Gene Oracle can use any standard classifier provided by scikit-learn to evaluate gene sets, and it is highly parallel since each gene set can be processed independently.

3.1.3 HemeLB

HemeLB is a high-performance Lattice Boltzmann code for sparse complex geometries, typically used to study vascular flow. HemeLB is a distributed CPU application, and it has been shown to scale up to nearly 50,000 cores [25, 24]. More recently, HemeLB was extended to also use GPUs, and it was shown to be scalable up to at least 32 GPUs [56]. HemeLB is a traditional bulk synchronous parallel application, which we have wrapped in a single-step Nextflow pipeline for the purpose of this study, in order to demonstrate the use of Tesseract with traditional HPC applications.

3.1.4 KINC

Knowledge Independent Network Construction (KINC) is a bioinformatics pipeline for constructing gene co-expression networks (GCNs) [55, 5]. KINC employs a master/worker distributed approach to divide the work among many processes. Additionally, each worker can independently perform work using the CPU or a GPU if one is available. The master assigns work dynamically such that each worker receives work blocks as fast as it can process them. In this case, the only communication is that of inputs and results between the master and each worker, and all of the disk I/O occurs on the master. Alternatively, KINC can use a chunk-and-merge approach in which many independent tasks are submitted with a static work distribution, each task produces a "chunk file" of the intermediate results, and a final merge process reads all of the chunk files and produces the final output files. This approach does not require MPI and may be advantageous on democratized computing platforms because it consists of many small tasks instead of one large task; however it may suffer from workload imbalance since it does not use a dynamic work distribution scheme.

KINC was one of the primary motivating examples for Tesseract, as shown by Figure 3.5.

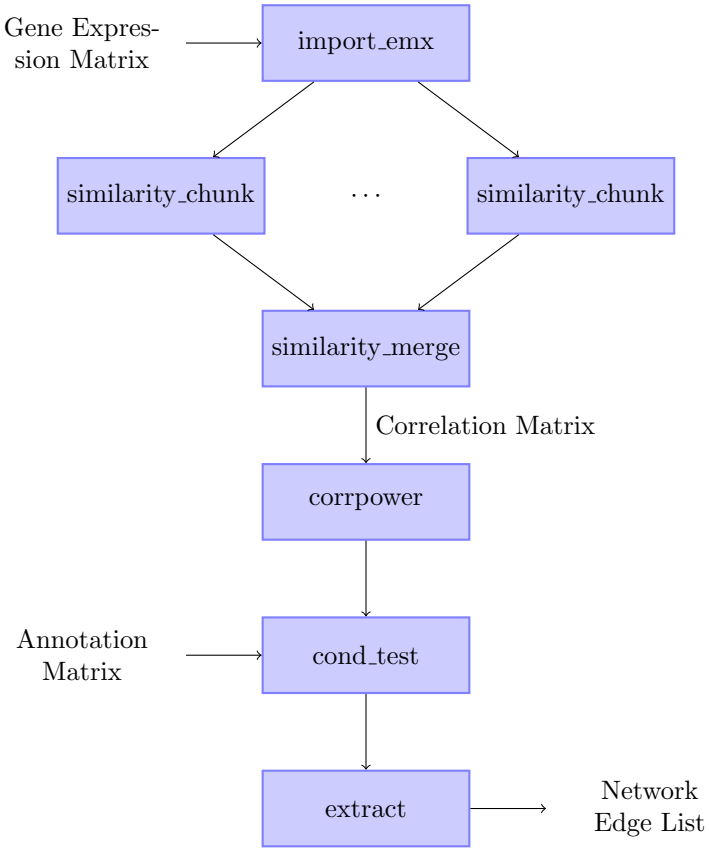


Figure 3.4: KINC workflow diagram

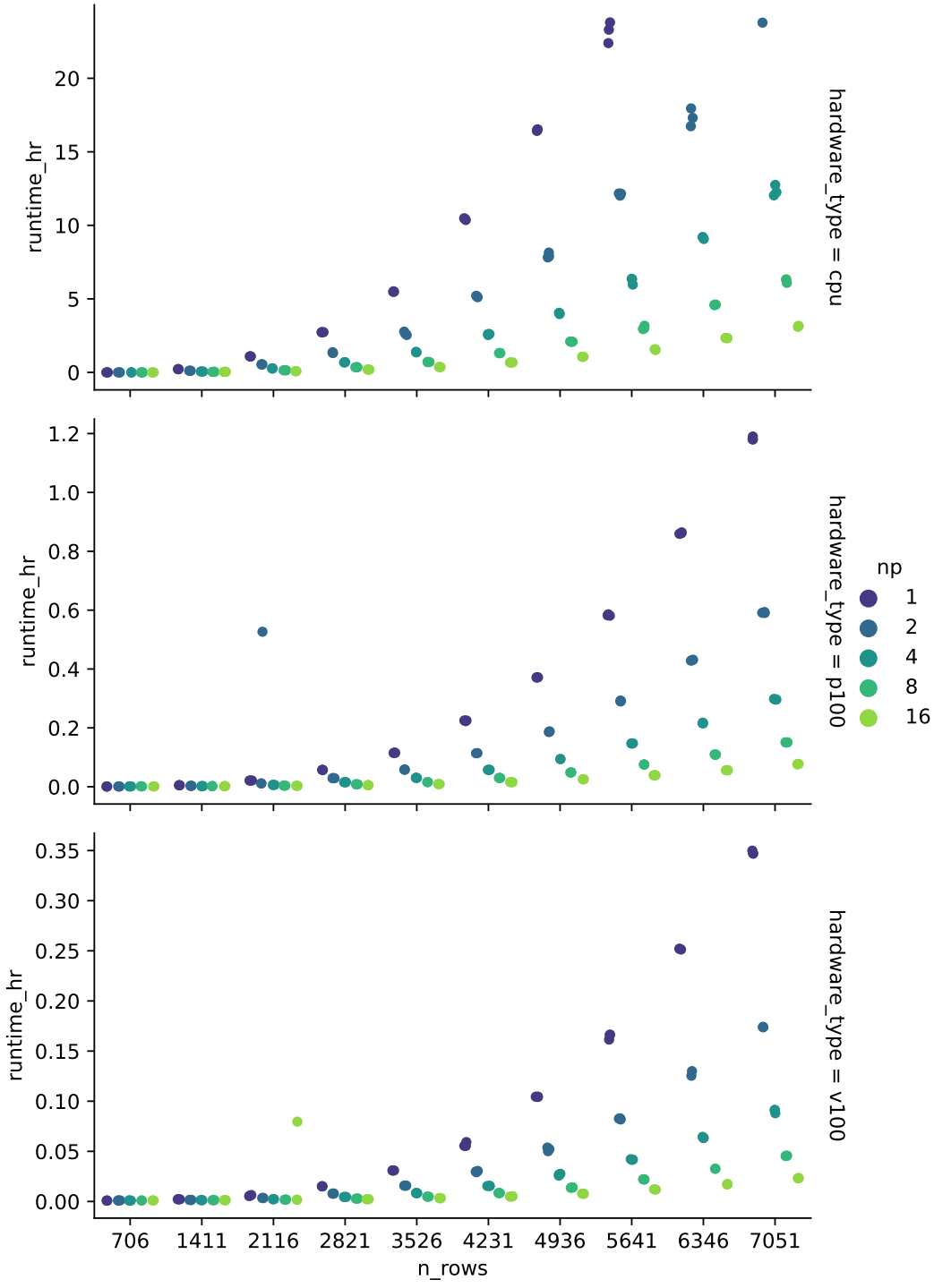


Figure 3.5: Runtime of KINC over a range of input data and processes.

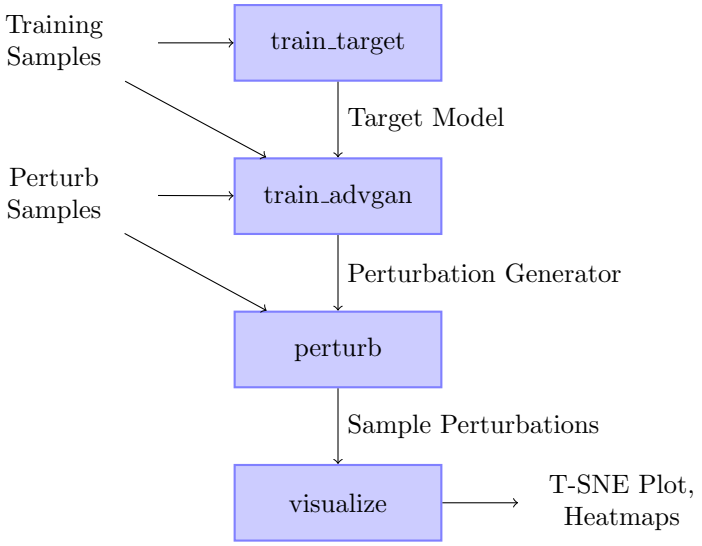


Figure 3.6: TSPG workflow diagram

The runtime of KINC can vary greatly depending on the size of the input data, the number of processes, and the hardware type (CPU or GPU). A single KINC run can range anywhere from a few minutes to a few days. There is no single walltime allocation that would be appropriate for all of these cases; any single value would either be insufficient in the largest cases or unnecessarily large for the smallest cases. Therefore, a model is needed to determine the required walltime from the relevant inputs.

3.1.5 TSPG

Transcriptome State Perturbation Generator (TSPG) is a deep learning pipeline for identifying genes with transitions between biological conditions [62]. TSPG uses a generative-adversarial network (GAN) to generate realistic gene expression perturbations between a source and target condition, such as normal kidney tissue to cancerous kidney tissue. The genes most strongly perturbed in the transition are used in downstream analyses, such as identifying therapeutic targets for a cancer condition. TSPG consists of several steps, including training a target model, training the GAN, and generating and visualizing the final perturbations. While a single TSPG model is trained with a fixed set of genes and a fixed target class, multiple TSPG models can be trained in parallel to cover different combinations of gene sets and target classes.

3.2 Computing Environments

In this section we describe the computing environments that were used in our experiments.

3.2.1 Palmetto Cluster

The Palmetto cluster at Clemson University is a condominium cluster with over 20 hardware phases, ranging from older CPU-only nodes to newer nodes equipped with GPU nodes and a high-speed interconnect. The GPU nodes include NVIDIA K40, P100, and V100 GPUs. When running a Nextflow pipeline on Palmetto, each task is submitted as a job script to Palmetto's PBS scheduler.

3.2.2 Nautilus

Nautilus is a Kubernetes cluster that is maintained as part of the Pacific Research Platform (PRP) [57]. The physical nodes in Nautilus are distributed across the United States, as well as a few nodes internationally, with the majority of nodes located in California. Nautilus contains a variety of hardware resources, including many nodes equipped with desktop-grade GPUs. In contrast to conventional university clusters and cloud platforms, Nautilus is an example of a "democratized" or "potluck" computing platform, because each user can contribute nodes to the cluster from anywhere around the world. When running a Nextflow pipeline on Nautilus, each task is submitted as a pod to the Kubernetes scheduler.

3.2.3 Google Cloud Platform

Google Cloud Platform (GCP) is a public cloud based on the same infrastructure that is used to run Google end-user products, such as Google Search, Gmail, Google Drive, and YouTube. Like other public clouds, GCP includes services for compute, storage, networking, big data, and machine learning. In particular, Google Compute Engine provides virtual machines (VMs) with a variety of CPU types and GPU types, including NVIDIA P100 and V100 GPUs. Nextflow pipelines can be run on GCP either through Google Kubernetes Engine or directly through the Cloud Life Sciences API. We use the latter approach in this dissertation as it requires fewer steps to configure the environment.

3.3 Tesseract: Intelligent Resource Prediction

3.3.1 Motivation

Scientific workflows, such as the ones described above, are often created by domain scientists who need to perform complex multi-step analyses on large datasets. However, domain scientists generally do not have the expertise required to optimize performance or diagnose failures quickly. While they can usually fix problems with enough persistence, such engineering tasks are a significant distraction from their research. One of the greatest challenges with using scientific workflows is knowing the resource requirements of each task in the workflow. Understanding the resource usage patterns of a workflow is critical when using large-scale computing platforms. Users must request the resources that they need for an experiment, and there are pitfalls to both under-provisioning and over-provisioning. On shared HPC platforms such as university clusters, over-provisioning may increase the time that the job must wait in the queue, and under-provisioning may cause the job to fail and have to be restarted. On cloud platforms, there are no queue times or walltime limits because resources are highly available[1], but there are significant financial risks related to incorrectly provisioning other resources such as GPUs, memory, and storage. Thus the lack of knowledge about resource requirements is a hindrance on HPC clusters and a major setback on cloud platforms. To this end, we have developed Tesseract, a semi-automated tool for intelligent resource prediction.

3.3.2 Design Considerations

Tesseract draws inspiration from a number of previous studies, including Ipek *et al.* [31], Matsunaga *et al.* [39], and Da Silva *et al.* [12]. The overall approach of Tesseract is to collect the resource usage data from past runs of a workflow and then train a machine learning model to predict the resource usage of future runs. This approach is open-ended enough that it may be able to capture the many sources of variation that contribute to application performance on a heterogeneous computing platform. In the same way that workflows like KINC are used to model biological systems, Tesseract views scientific workflows as "computational systems", composed of the numerous hardware and software layers that facilitate their use. While the individual components are carefully designed, the system as a whole is complex and emergent, and should be modeled as

[1] Preemptible VMs, which are preferred for their significantly lower cost, have a maximum runtime of 24 hours on GCP, so there is effectively a walltime limit in this case

such. Here we describe the Tesseract in more detail by answering a number of pertinent design questions.

Which resources do we need to predict? While Tesseract can target any resource metric that is collected by Nextflow, we are particularly interested in runtime, memory usage, and disk usage, as these resources are the most pertinent to the problem of resource provisioning. Runtime refers to the duration of a task. Memory usage refers to the maximum amount of memory used at one time throughout the duration of a task. Disk usage refers to the total amount of output data written to disk storage by a task. In the Nextflow execution trace, these resources correspond to `realtime`, `peak_rss`, and `write_bytes`. Tesseract predicts the resource usage of individual tasks rather than entire workflow runs, as resource provisioning typically occurs at the level of tasks.

The number of CPUs is treated as an input rather than an output because it is almost always provided as an input. Applications are generally designed to work with however many CPUs they are given, whereas memory and disk usage are usually determined by the problem size and other input parameters. In other words, if fewer CPUs are provided, then the application will do the same work but will take longer; if an application requires more memory or disk space than is available, generally the application has no other option than to terminate. We can derive metrics such as CPU-hours from runtime as long as we have the number of CPUs as an input. The same reasoning can be applied to GPUs, FPGAs, and any other such accelerators, which is pertinent to this work since we consider several GPU-enabled applications. Similarly, we do not consider CPU/GPU utilization in this study because utilization is not a resource that can be allocated (although some platforms do support fractional CPU allocations). However, utilization metrics would be useful for comparing the performance of different implementations of an application, and in fact Tesseract can be easily applied to this task.

What features can we use as inputs? Previous studies have addressed this question in many different ways. To help guide our exploration, we have devised a set of categories inspired by Guo [26]. We say that a workflow task has three possible sources of inputs:

- **Code**: Compile-time and run-time application parameters
- **Data**: Input data characteristics; file size, dimensions, sparsity
- **Environment**: Attributes of the underlying hardware and operating system; kernel settings, environment variables, benchmark metrics

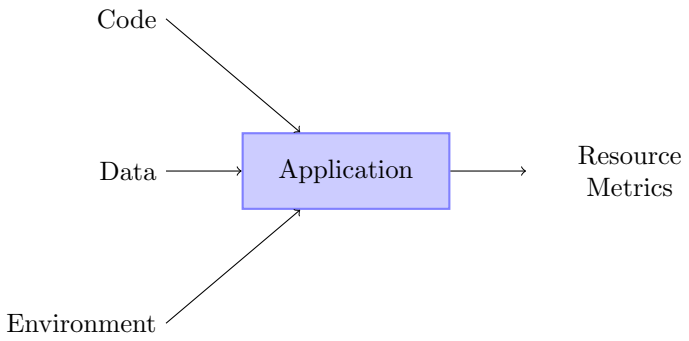


Figure 3.7: Breakdown of input features from code, data, and environment

These three categories form a basis for understanding how to select the right input features for a prediction model. For example, one could include specific command-line parameters (code), input data size (data), and performance metrics for the underlying hardware (environment). The user may include as many features as needed from each category to meet the needs of each application. Because every application is unique, Tesseract requires the user to manually define the input features for each workflow process. As a result, the input features should be easy to obtain. Using more fine-grained information such as profiling traces would require additional tools and instrumentation, all of which are unfamiliar to non-expert users. Input parameters and input data characteristics can be obtained from the task script with few modifications, and previous studies have shown that simple input features can achieve sufficiently accurate predictions.

How accurate does the prediction model need to be? Some studies define 20% relative error or less as acceptable [6]. Relative error can be assessed using mean absolute percentage error (MAPE):

$$\text{MAPE } (\%) = 100 \times \sum_{i=1}^{n} \left| \frac{y_{true,i} - y_{pred,i}}{y_{true,i}} \right|$$

While we find this threshold reasonable, this metric does not adequately describe the effectiveness of a resource prediction model because the costs of underestimation and overestimation are different. An under-provisioned task will fail and have to be re-run, whereas an over-provisioned task may have some negative effects (i.e. longer queue time, wasted resources) depending on the situation, but will still complete. A good resource prediction model should avoid under-provisioning entirely, even at the expense of some over-provisioning, so long as it is not extreme [2]. We address this issue by

[2]Under-provisioning is not so much an issue for applications that use checkpointing, however many applications, including our entire test suite, do not use checkpointing.

using regression models that provide a confidence interval around each point prediction, and using the upper bound as the resource request. We evaluate these intervals using the coverage probability (CP), which is the percentage of intervals that contain the true target value. Given a sufficiently large number of predictions, a 95% confidence interval should provide a coverage probability of at least 95%. This metric essentially measures a model's ability to avoid both under-provisioning and extreme over-provisioning. For aesthetic consistency with prediction error, we use coverage error, which we define as CE (%) = 100 − CP. Achieving at least 95% coverage is equivalent to achieving 5% coverage error or less.

How many training samples are needed? The amount of training data is a serious consideration because training data is expensive to acquire, which is a primary drawback of data-driven resource prediction. Tesseract should be able to achieve acceptably low error with as few training samples as possible, in order to minimize the cost of acquiring training data. More importantly, however, Tesseract should be able to learn from the runs that users have already performed as part of their normal work. That way, not only will the prediction model reflect the most recent usage patterns, but it will not require any more runs than would have been performed anyway.

The design principles of Tesseract are summarized as follows:

- Predict runtime, memory usage, and disk usage of individual workflow tasks;
- Use input features that are easy to acquire, such as input parameters, input data characteristics, and basic performance metrics;
- Achieve less than 20% relative error on point predictions;
- Achieve less than 5% coverage error on 95% confidence intervals;
- Use trace data from historical runs as training data.

3.3.3 Implementation and Usage

Here we describe how Tesseract is implemented, and how a domain scientist would use Tesseract with their scientific workflow. The entire usage cycle can be summarized as follows (also illustrated in Figure 3.8):

1. User defines input features for each process in a Nextflow pipeline

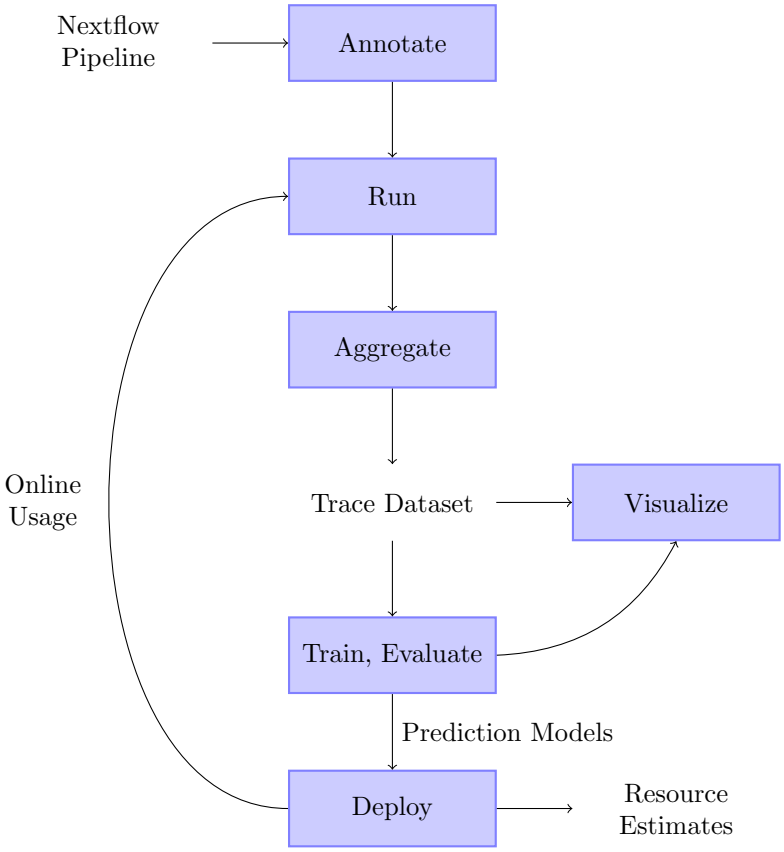


Figure 3.8: Workflow diagram of Tesseract

```
echo "#TRACE dataset=${dataset}"
echo "#TRACE hardware_type=${params.similarity_hardware_type}"
echo "#TRACE chunks=${params.similarity_chunks}"
echo "#TRACE threads=${params.similarity_threads}"

kinc settings set cuda ${params.similarity_hardware_type == "cpu" ? "none" : "0"}
kinc settings set opencl none
kinc settings set threads ${params.similarity_threads}
kinc settings set logging off

kinc chunkrun ${index} ${params.similarity_chunks} similarity \
    --input ${emx_file} \
    --clusmethod ${params.similarity_clusmethod} \
    --corrmethod ${params.similarity_corrmethod}
```

Listing 3.1: Process script for KINC / similarity_chunk with trace directives

2. User runs the pipeline many times as part of their normal work
3. Tesseract combines input features from execution logs with the execution traces to produce a trace dataset for each workflow process
4. User selects the desired input features and prediction targets for each workflow process
5. Tesseract trains a model for each prediction target
6. Prediction models are used to set resource requests in future workflow runs

Tesseract can predict resource usage for any application or workflow that is implemented as a Nextflow pipeline. Tesseract depends on Nextflow to collect the input features and resource metrics that comprise the training data. Any standalone application can be easily wrapped into a single-step Nextflow pipeline, and any workflow can be refactored into a Nextflow pipeline, although difficulty may vary. Tesseract itself is a collection of Python scripts that use a number of standard libraries for machine learning, including Numpy, Pandas, scikit-learn [48], Tensorflow [1], and Keras [10], as well as matplotlib [30] and seaborn [68] for visualizations.

Given a Nextflow pipeline, the user must first define the input features for each workflow process using "trace directives". A trace directive is a print statement that prints the `#TRACE` prefix followed by a key-value pair, which denotes the name and value of an input feature. The key will become a column in the trace dataset, and the value will be saved for each task. Trace directives are evaluated and printed to the execution log during task execution. An example of a process script with trace directives is shown in Listing 3.1. Since the user may not yet know which inputs will be

most important for prediction, it is better to be inclusive rather than exclusive at this stage. Input features can always be discarded later on, but if an input feature is not included as a trace directive, it will be difficult or impossible to recover it later on without rerunning the workflow. In general, trace directives consist of input parameters and simple input data characteristics such as file size, number of lines, or number of rows and columns. Only those input features that vary between runs are useful for prediction.

Once a pipeline has been annotated with input features, it must be run many times in order to generate enough training data. These runs should ideally be a part of the user's normal work, or they can be generated for the specific purpose of acquiring training data. In either case, the runs should be representative of real experiments and should span a range of input conditions. Nextflow will produce an execution trace for each task which contains the desired resource metrics. After enough experiments have been run, Tesseract aggregates the input features and resource metrics from all runs and produces a trace dataset for each workflow process.

Tesseract creates a prediction model for each resource metric, for each process in the workflow. We refer to a particular workflow / process / resource metric as a "prediction target". A workflow with multiple processes and multiple resource metrics will produce many prediction targets. However, we have found that a simple heuristic can be used instead of a regression model in many cases. For example, many processes take only a few minutes to execute, or use only a few megabytes of memory, or produce only a few kilobytes of output data (i.e. log files). For these targets, the maximum target value rounded up is sufficient for the purpose of resource provisioning. Therefore, Tesseract only trains a model for prediction targets with standard deviation greater than 0.1 hr (in the case of runtime) or 0.1 GB (in the case of memory and disk usage); otherwise, it uses the maximum target value rounded up. Tesseract can use any prediction model that implements the scikit-learn Estimator API, but also provides models that can be used out-of-the-box. These models are described in the next section.

In the final step, the user employs Tesseract to provide resource estimates for future runs. Currently, this step is done by querying Tesseract and manually updating the corresponding resource settings in the Nextflow pipeline. In the future, we would like to integrate Tesseract with Nextflow such that resource estimates are automatically queried and applied when a task is launched.

3.4 Experiments

3.4.1 Resource Prediction

We defined input features for each process, for each workflow in our test suite. We generated trace datasets for each workflow by running the workflow many times with a range of parameters and input data, based on the typical usage of these workflows by our peers. More information about the trace datasets is provided in Appendix A. We used Tesseract to predict runtime, memory usage, and disk usage for each workflow in our test suite. This setup produced over 120 prediction targets, 16 of which were selected by Tesseract for model training based on the criteria described in the previous section. The selected targets span all five workflows and all three resource types. For each selected prediction target, we trained a neural network and a random forest. The neural network was configured with hidden layer sizes $(128, 128, 128)$, ReLU activation, MAE loss, and the Adam optimizer. Additionally, the neural network used L2 regularization and dropout in order to facilitate the use of confidence intervals. The neural network was implemented in TensorFlow. We used the random forest regressor in scikit-learn with 100 decision trees and MAE criterion. Preliminary experiments revealed that these two models consistently outperformed other regression models and neural network architectures. We evaluated each model by performing 5-fold cross validation. Since each fold is used once as the test set, we combined the predictions of each fold in order to compare predicted and actual values for the entire dataset. We evaluated these predictions using MAPE, as described in the previous section.

We extended the models described above to provide a 95% confidence interval around each point prediction. For the neural network, we enabled dropout during inference and took the mean and variance of multiple repeated predictions, as demonstrated by Gal *et al.* [18]. For the random forest, we used the jackknife [66] to obtain a bias-adjusted variance estimate based on the predictions of the individual decision trees in the random forest. The `forestci` package [51] integrates this approach seamlessly with the random forest regressor in scikit-learn. For both models, the 95% confidence interval is equivalent to the point prediction (mean) +/- two standard deviations. We evaluated the confidence intervals using coverage error as described in the previous section.

We also investigated the effect of training set size on prediction error and coverage error, in order to determine the minimum number of training samples needed to achieve acceptable levels in these metrics. For each prediction target, we trained and evaluated a neural network for each

train/test split ranging from 10/90 to 90/10. We performed three independent trials for each split.

3.4.2 Anomaly Detection

Several approaches to anomaly detection were described by Mandal *et al.* [37]. We decided to use auto-regression, as it integrates seamlessly with our existing methodology. In auto-regression, a prediction model is trained on "normal" (non-anomalous) data, and then if the model makes an incorrect prediction during inference, the corresponding input sample is marked as an anomaly. For example, suppose a model is trained to predict the runtime of a workflow process, and during inference the model predicts that a particular task will run for 12 hours, but the task ultimately runs for 18 hours. The model would then identify the task as an anomaly and an external system could alert the user to review the task details for potential errors. The model could go one step further by flagging the task while it's running, if its runtime exceeds the predicted runtime by some amount. Similar strategies can be implemented for memory and disk usage.

The challenge with auto-regression is determining the appropriate decision threshold. In many cases there are inherent sources of noise (e.g. network and disk congestion) that lead to noise in the training data and irreducible error in a trained model. Continuing the previous example, a task that runs 6 hours longer than predicted is anomalous only if the normal variation in the runtime of that process is significantly less than 6 hours. In other words, we need a way to separate *aleatoric* (irreducible) uncertainty from *epistemic* (reducible) uncertainty. This problem is addressed by equipping the prediction models with confidence intervals, as described in the previous subsection. When a task is completed, the actual resource usage is compared to the predicted resource usage, and an anomaly score is assigned to the task based on the difference. Any task with an anomaly score greater than 0.997 (corresponding to three standard deviations or more) is marked as an anomaly. We use this approach to identify anomalies in the core resource prediction experiments, and we evaluate the results qualitatively.

3.4.3 Cross-Platform Runtime Prediction

Given a dataset of many workflow runs from a source platform and a few small runs from a target platform, we would like to predict the runtime of large runs on the target platform. This capability is crucial for estimating the cost of large-scale cloud experiments *a priori*, which we

Name	Description	Benchmark
cpu_flops	CPU FLOP rate	Matrix multiplication (double precision)
cpu_mem_bw	CPU memory bandwidth	STREAM Triad (double precision)
disk_read	Disk read bandwidth	Read 1 GB file from disk
disk_write	Disk write bandwidth	Write 1 GB file to disk
gpu_flops	GPU FLOP rate	Matrix multiplication (double precision)
gpu_mem_bw	GPU memory bandwidth	STREAM Triad (double precision)

Table 3.1: Performance metrics collected by Minibench

describe in the next sub-section.

In order to train a model to predict runtime across different "platforms", whether they are completely different systems or different nodes within a heterogeneous system, the training data must include some set of features that can distinguish the two platforms. One approach is to define a categorical variable with a category for each platform (for example, the "hardware_type" input feature defined in the HemeLB workflow). This approach can be effective but it is inflexible to the addition of new platforms. A more robust approach is to define a set of performance metrics that can provide a "fingerprint" of each platform. Basic performance metrics such as FLOP rate, memory bandwidth, and disk bandwidth have proven to be a good starting point [39]. To this end, we have developed Minibench, a lightweight benchmarking tool that can be run in any Linux environment. Minibench collects a standard set of performance metrics, described in Table 3.1, but it can be extended to collect other metrics that might be relevant to specific use cases.

The procedure for using Minibench with Tesseract is as follows:

1. Define the set of "node types" for a given platform (e.g. each phase in Palmetto).
2. Run Minibench, which produces a table of performance metrics for each node type.
3. Define an input feature in each workflow process that records the node type (e.g. the phase on which a task is executed).
4. Augment the trace data with Minibench metrics via join operation.

Minibench is itself a Nextflow pipeline, which means that it can capture system-level variation that might not be captured in a more controlled benchmarking environment. Some examples of system-level variation include (1) a networked file system whose performance is affected by overall system usage, and (2) a "node type" in Minibench that consists of nodes with the same GPU model but

different CPU models (see Nautilus benchmark in Appendix C). In these situations, it is necessary to run Minibench several times for each node type and encode the distribution of each metric rather than a single result. For the purpose of this dissertation, we take the mean of three independent trials for each metric. As an aside, one could instead provide several percentiles (e.g. the 5-th, 50-th, and 95-th percentile) for each metric, in cases where the mean is insufficient. More information about Minibench, including results for each platform that was used in this dissertation, is provided in Appendix C.

For this experiment and the following cost prediction experiment, we focused on a single prediction target, the runtime of the KINC / similarity_chunk process, as the resource usage of KINC was one of the primary motivating examples of this dissertation. Additionally, we focused on the neural network regressor as described in the core resource prediction experiments. We performed many small and large KINC runs on the Palmetto cluster, as well as a smaller number of small and large runs on the the Nautilus cluster. Notably, the KINC runs on Palmetto consisted of runs on both P100 and V100 GPUs. The trace data was augmented with performance metrics collected by Minibench on each platform. More information about the trace datasets is provided in Appendix A. We then performed two evaluations. In the first evaluation, we treated the P100 and V100 runs on Palmetto as two different platforms: we trained a neural network regressor on the P100 runs and some V100 runs, and evaluated the model on the large V100 runs. In the second evaluation, we trained a neural network regressor on the Palmetto runs and some Nautilus runs, and evaluated the model on the large Nautilus runs.

This experiment evaluates the ability of Tesseract to predict runtime on one platform with training data from a different platform and performance metrics from both platforms. As such, the prediction model is given the appropriate performance metrics as input features during evaluation.

3.4.4 Cost Prediction

Cost estimates are critical to performing large science experiments on cloud platforms in a cost-effective way, as described in the previous chapter. Many platforms provide an interface for estimating the cost of long-running deployments, but these interfaces are not amenable to estimating the cost of many heterogeneous tasks. Fortunately, the cost of a workflow can be easily derived from the resource usage of the workflow tasks and the prices of individual cloud resources, which cloud platforms also provide.

In this experiment, we predicted the cost of several large KINC runs on GCP, using training data from Palmetto. We repeated the Palmetto/Nautilus runtime prediction experiment, re-using the trace data from Palmetto and replicating the Nautilus runs on GCP. We trained and evaluated a neural network regressor, in the same manner as before, to predict both runtime and cost of the large runs on GCP. We developed a cost model based on the GCP pricing documentation [11], which computes the hourly cost of a task based on the resource requirements (CPUs, memory, disk, GPUs). The predicted cost of a task, then, is the product of the hourly cost and predicted runtime of the task. More information about this cost model is provided in Appendix D. Finally, as a side note, we estimate the cost of the Palmetto runs as if they were run on GCP, and we discuss the trade-offs that can be ascertained from the resulting resource usage and cost estimates.

3.5 Summary

In this chapter, we described Tesseract and the experiments that were performed to validate it. Our experiments were performed in two phases. In the first phase, we developed resource prediction models for a suite of scientific workflows with diverse resource requirements. In the second phase, we developed additional capabilities that leverage resource predictions to perform more advanced tasks. In all of these efforts we used a methodology that is easy to replicate for domain scientists, as we intend for the results of our work to benefit domain scientists first and foremost. In the next chapter, we present and discuss our results.

Chapter 4

Preliminary Results

In this chapter, we describe the results for each of the experiments laid out in the previous chapter, demonstrating Tesseract's capabilities in resource prediction, anomaly detection, cross-platform runtime prediction, and cost prediction. We discuss both the limitations and implications of our findings, and we discuss how Tesseract addresses the needs of domain scientists who use data-intensive computing.

4.1 Resource Prediction

Figure 4.1 shows a random selection of prediction targets that were not selected for model training. Nearly all of the "excluded" targets had resource usage below 1 hr or 1 GB.

The results of the resource prediction experiments are summarized in Figure 4.2, which shows the prediction error and coverage error achieved by each model for each prediction target. An effective model should achieve less than 20% MAPE, or, failing that, achieve less than 5% coverage error. While many of the models did not achieve low prediction error, every model except for three achieved sufficiently low coverage error, and for every prediction target there is at least one model with low coverage error. In other words, even when prediction error is high, such as in cases where the training data is sparse or noisy, the prediction model can still protect against under-provisioning by providing sufficiently large confidence intervals. These results demonstrate that Tesseract can predict resource usage across a diverse set of workflows, using only basic input features, for the purpose of provisioning resources for tasks.

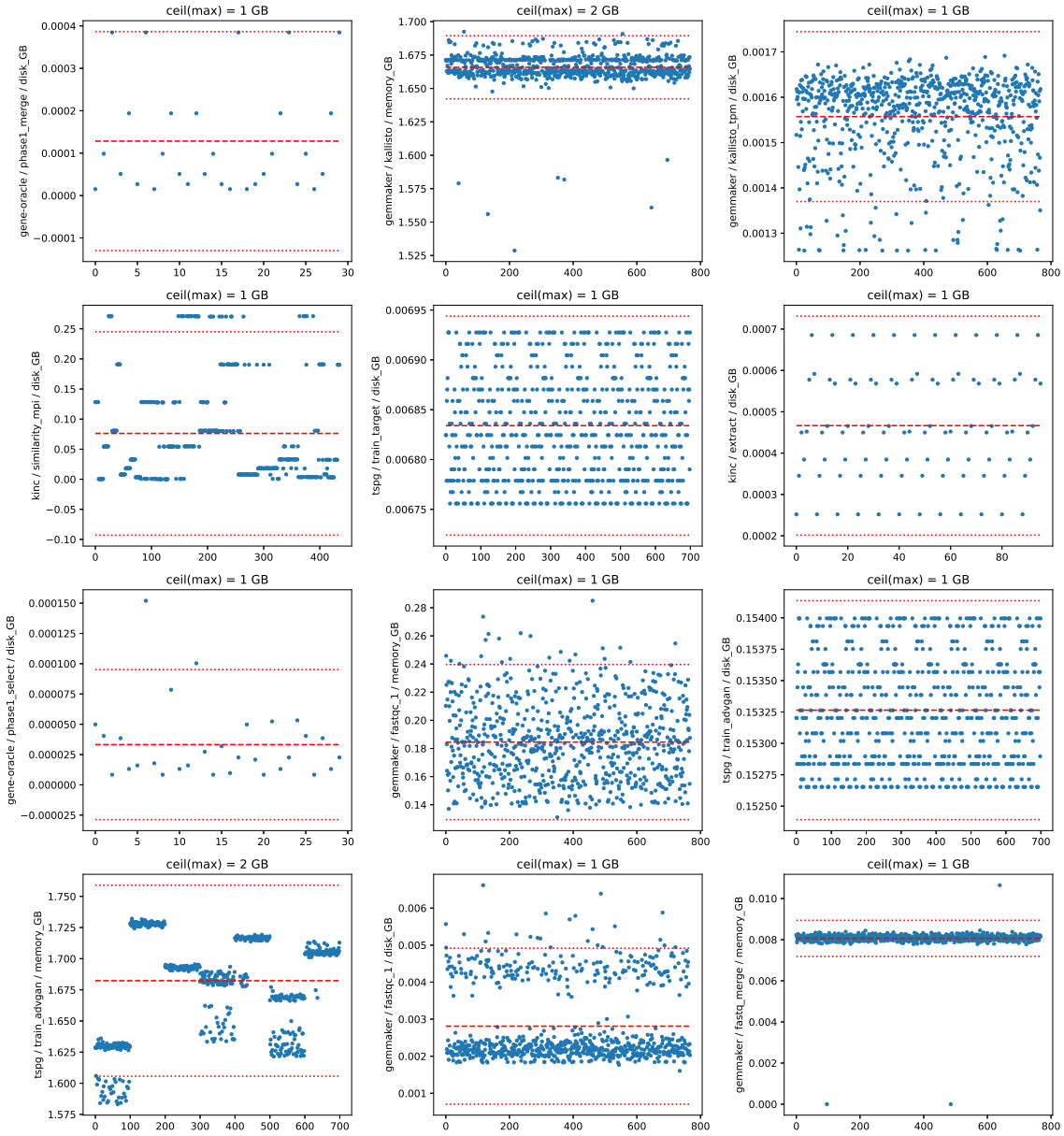


Figure 4.1: Resource usage plots for several prediction targets that were not selected for model training. The mean +/- two standard deviations are denoted by the dashed and dotted red lines. The recommended resource request, defined as the maximum target value rounded up to the next hour or GB, is given in the title of each plot.

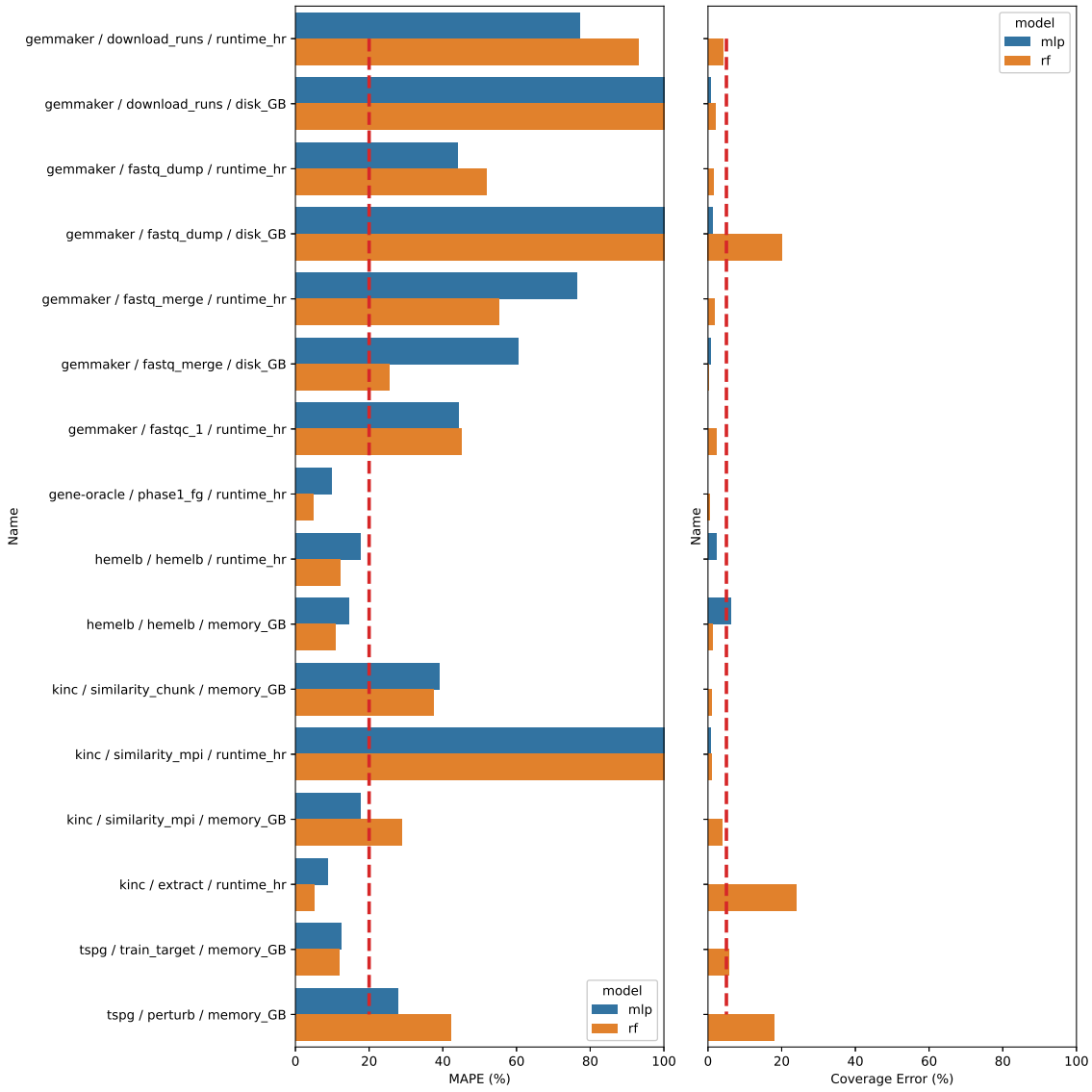
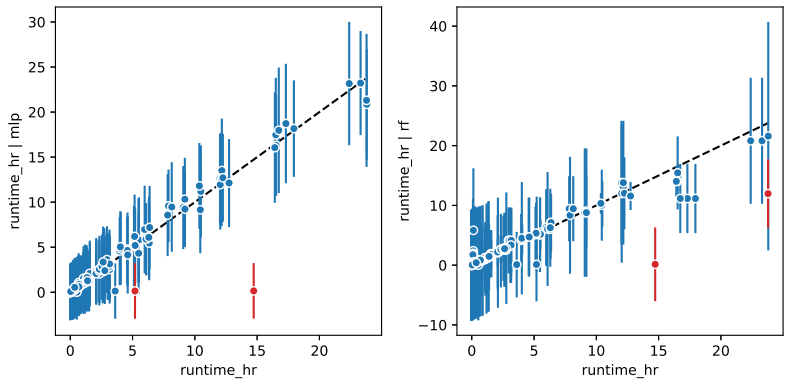
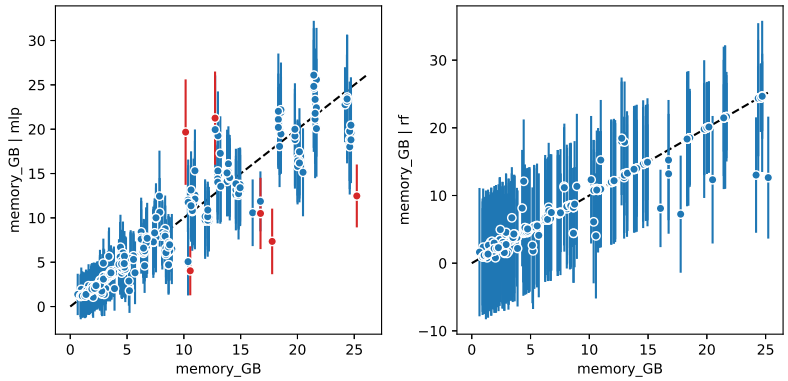


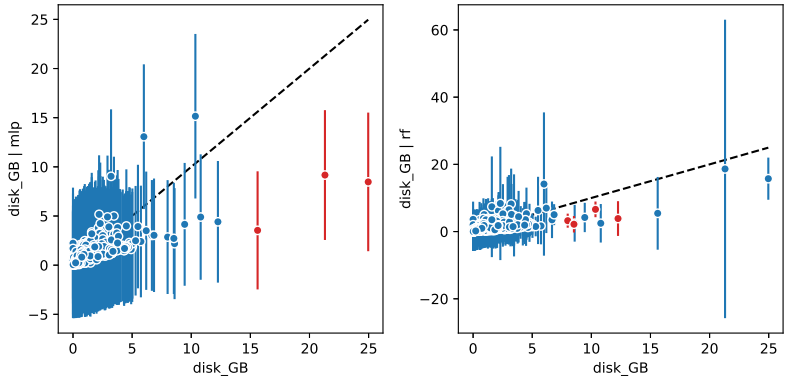
Figure 4.2: Summary of resource prediction results for the selected prediction targets. The left-hand panel shows mean absolute percentage error (MAPE), with 20% MAPE denoted by the dashed red line. The right-hand panel shows coverage error of the prediction intervals, with 5% coverage error (95% coverage probability) denoted by the dashed red line.



(a) KINC / similarity_mpi runtime



(b) HemeLB memory usage



(c) GEMmaker / download_runs disk usage

Figure 4.3: Expected vs predicted target values for three prediction targets, one example for each of runtime, memory usage, and disk usage. In each case, the black dashed line denotes equality, and each point and vertical bar is a point prediction with corresponding 95% confidence interval. Runs that were marked anomalies are colored red.

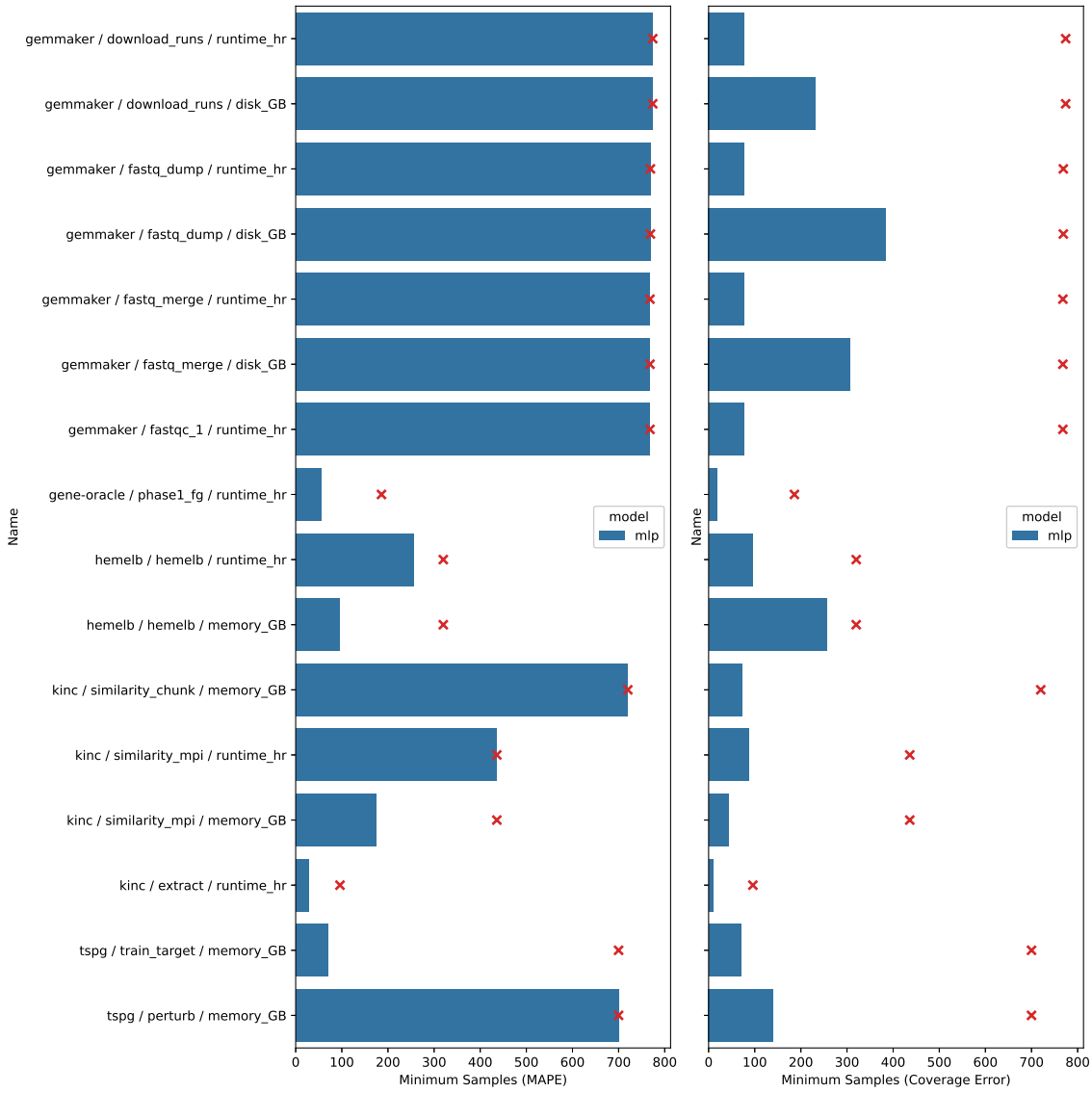


Figure 4.4: Summary of training set size results for the selected prediction targets. The left-hand panel shows the minimum required samples to achieve 20% MAPE or less, and the right-hand panel shows the minimum required samples to achieve 5% coverage error or less. In both cases, the red crosses denote the total number of samples for the prediction target.

Figure 4.3 provides more detailed results for three prediction targets, one for each of runtime, memory usage, and disk usage. These scatter plots are merely a visual sample in lieu of displaying detailed results for all 16 prediction targets. In general, these plots are useful because they provide context for the error metrics at the level of individual predictions. The full results for each prediction target are provided in Appendix B.

The neural network and random forest are evenly matched overall, with each model providing better performance over the other in different cases. In many cases, one model achieves better coverage than the other by having larger confidence intervals, at the cost of greater overestimation. In the future, it would be useful to train these models in such a way that strikes a balance between coverage and interval size. In practice, the user can use one or the other across the board, or have Tesseract select the best model in each case.

The discrepancy between prediction error and coverage can be explored by considering the prediction targets for GEMmaker. All of these targets exhibited high prediction error but low coverage error, and they offer insights into the potential sources of prediction error. One example is the download_runs process, which downloads files from a remote database. There is inherent uncertainty in the runtime of this process due to variation in network throughput. Another example is the fastq_dump process, which decompresses a compressed file. The disk usage of this process can not be fully determined from the compressed file size because some files can be compressed more than others. In both cases, there is a lower bound on the prediction error due to inherent uncertainty, and while additional features such as networking metrics or compression ratios might reduce this uncertainty, confidence intervals provide a way to manage it in the absence of complete information. In other words, confidence intervals give users the flexibility to collect as many input features as they need to achieve sufficiently small confidence intervals. A model with fewer input features will generally have larger confidence intervals and require more conservative resource requests, but the user can collect more input features at their own pace, and in the meantime the confidence intervals will still be more useful than blind estimates.

The model errors in these experiments tended to be underestimates at the high end and evenly distributed at the low end. As a result, using the point predictions alone would lead to poor outcomes because the cost of underestimation (task failure) is much greater than the cost of overestimation. The first issue can be explained by the imbalance between low-usage and high-usage runs in nearly every dataset; a model trained with relatively few high-usage runs will perform poorly

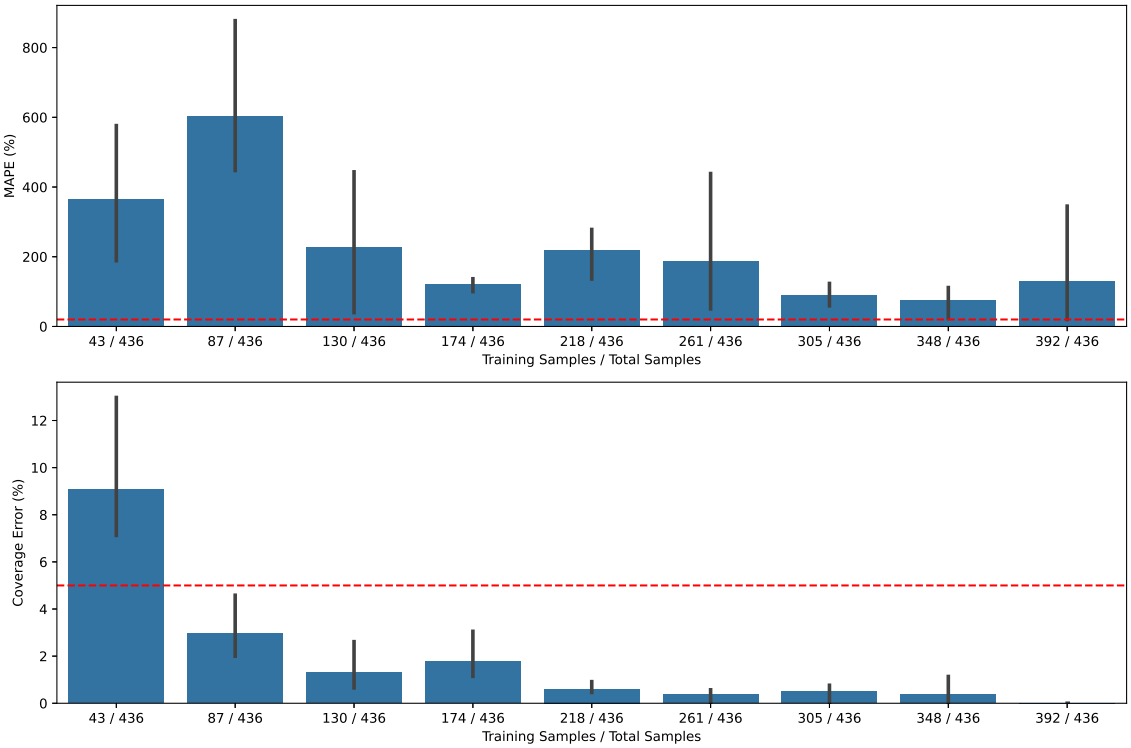


Figure 4.5: Effect of training set size on prediction error and coverage error for KINC / similarity_mpi runtime.

on unseen high-usage runs, even though the cost of underestimating these runs is much greater. Methods for addressing data imbalance include transforming the target variable and resampling the training data, however we were not able to reduce underestimation with these methods. The second issue can be explained by the fact that the models were trained with a symmetric loss function. An obvious alternative is to use an asymmetric loss function that penalizes underestimation more than overestimation, thereby biasing the model to overestimate. Again, this approach did not reduce underestimation in our case, and in fact it only increased overall prediction error. Instead, we found that confidence intervals were able to reduce underestimation in both cases, simply by using the upper bound as the resource request. In other words, confidence intervals reduce underestimation in lieu of other methods such as resampling and asymmetric loss functions, which would require more fine-tuning. There are a few cases in which high-usage runs are still underestimated, such as GEMmaker / download_runs disk usage, and in these cases the only solution may be to acquire more medium- and high-usage runs for the training data.

The results of the training set size experiments are summarized in Figure 4.4, which shows the minimum number of training samples required to achieve 20% prediction error and 5% coverage error for each prediction target. Additionally, detailed results for KINC / similarity_mpi are shown in Figure 4.5, and the same results for all prediction targets are provided in Appendix B. These results allow us to determine how many runs of a workflow are necessary to produce a reliable prediction model. The same prediction targets that did not achieve low prediction error with an 80/20 split in the core experiments, namely all of the GEMmaker targets, also cannot achieve low prediction error in these experiments, hence their "minimum required samples" is equal to the total sample size. On the other hand, every prediction target can achieve 95% coverage with only a fraction of the respective dataset, and a small fraction in many cases.

The minimum required training samples for a given process will depend on factors such as the complexity of the resource usage and the number of relevant input features. The trace datasets in this dissertation were generated by varying many different inputs, and in that sense are intended to provide an exhaustive view of the resource usage of each workflow. In reality, not all inputs will need to be varied. For example, while the runtime of KINC / similarity_mpi depends on input data size, number of parallel processes, and hardware type, a typical user might run KINC on a variety of datasets but with an otherwise constant configuration (e.g. four chunks with V100 GPUs). This scenario has two key implications: (1) fewer training samples will be required because fewer inputs are being varied, (2) the training data will likely be imbalanced because it is not being generated in a controlled manner. Mitigating imbalance in small datasets has proven difficult, but there are many strategies, such as using a weighted loss function and other re-sampling strategies, that have not yet been tried, and warrant investigation in future work.

4.2 Anomaly Detection

The anomaly detection experiments were performed in tandem with the core resource prediction experiments. Predicted anomalies are highlighted in the marginal scatter plots in Appendix B. We focus on the results for KINC / similarity_mpi runtime (Figure 12), as it contains the clearest example of anomalies in the training data. The scatter plots show that there are a few high-runtime tasks that were predicted to have very low runtime, but the marginal plot of runtime versus hardware_type shows that these runs took significantly longer than similar runs, that is, they are clearly

anomalies. Figure 12 shows that both models were able to correctly identify these runs as anomalies, using the anomaly score and threshold defined in the previous chapter.

Similar examples can be found for the other prediction targets, however not all marked anomalies appear to be true anomalies. For example, several high-usage runs for GEMmaker / download_runs disk usage (Figure 2) are marked anomalies by the random forest, when in reality these runs were probably just high-usage runs that were under-represented in the training data. Examples such as these highlight some of the limitations of the anomaly score. In some cases, it is difficult to distinguish between data anomalies and model errors without broader context about the application. Some kinds of anomalies cannot be caught; for example, if a task has unusually high runtime and the predicted runtime is also used as the requested walltime, then the task will fail before the model can assess whether it is an anomaly. On the other hand, there are no walltime limits in a cloud environment, so the anomaly score is very useful in this situation for alerting the user to potential cost overruns.

Ultimately, the anomaly detection mechanism is only intended to identify potential anomalies for the user to review. Regardless of whether a task is marked an anomaly or fails outright, the user must always assess the context of the anomaly (inputs, execution logs, which resource was anomalous, etc) and decide whether the task or the model is at fault. In an online training scenario, the resource usage data may then be kept for training or discarded based on the user's decision. This cycle of anomaly detection and re-training both acts as a guided debugging tool and allows the model to improve over time. The appropriate anomaly threshold will depend on the user's tolerance for false positives and false negatives; even so, the 0.997 threshold is a sensible default, and the anomaly score itself is an improvement because it is based on the variance of the model prediction.

4.3 Cross-Platform Runtime Prediction

The results of the Palmetto P100/V100 experiment are given in Figure 4.6, following the same visual semantics as the core resource prediction experiments. In this experiment, the P100 and V100 runs are ordered by dataset size, and each model is trained on a certain proportion of P100 and V100 runs. For example, a (0.75, 0.25) model is trained on the first 75% of P100 runs and the first 25% of V100 runs, and then evaluated on the remaining V100 runs. Varying these two parameters yields nine models ranging from (0.25, 0.25) to (1.00, 0.75), allowing us to observe the

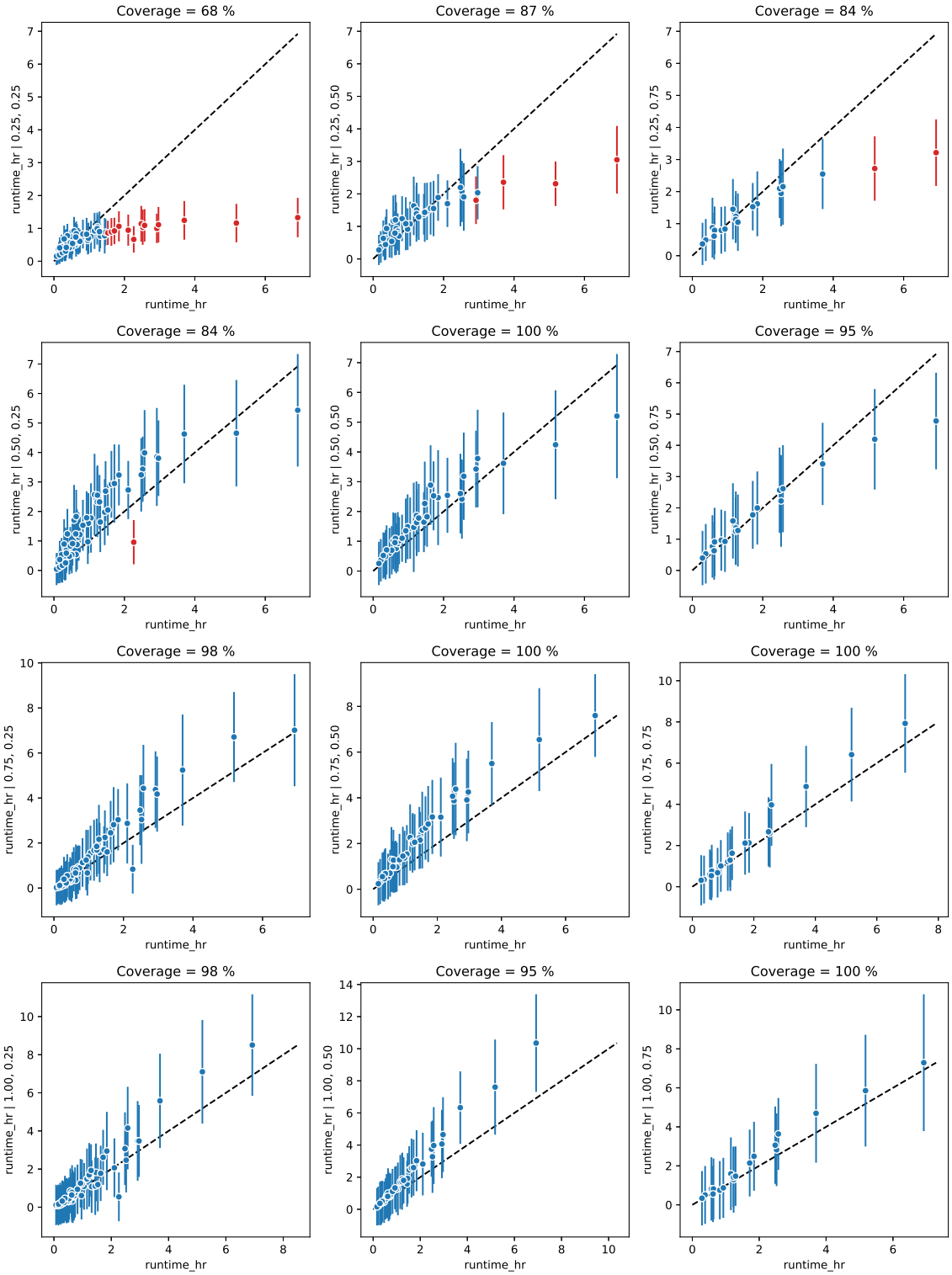


Figure 4.6: Expected vs predicted runtime for each model in the Palmetto P100/V100 experiment.

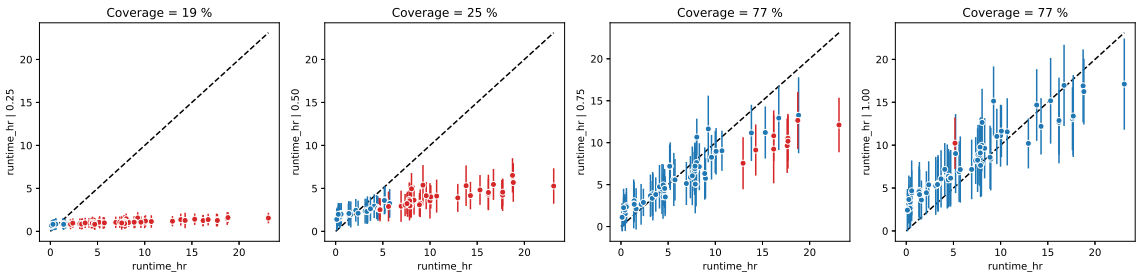
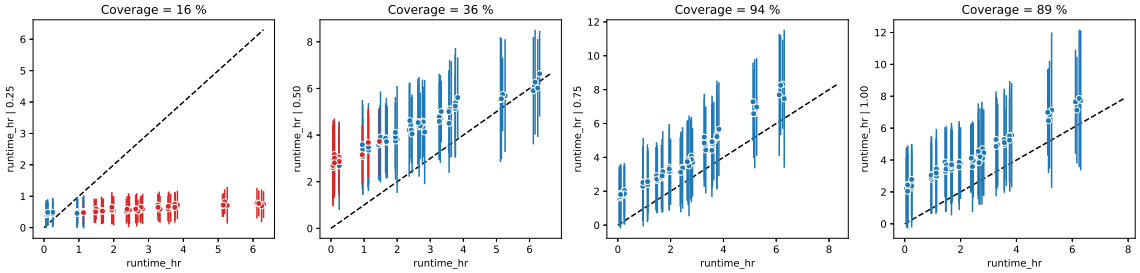


Figure 4.7: Expected vs predicted runtime for each model in the Palmetto/Nautilus experiment.

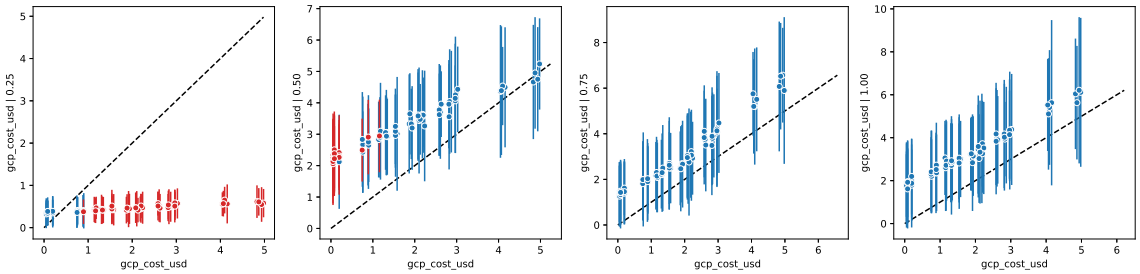
effect of including larger runs in the training data. The models trained with only the first 25% of P100 runs do not generalize well to larger runs, even with more V100 runs. However, several models with 50% or more of the P100 runs achieve high coverage, including models that did not use the largest 25% of P100 runs, which shows a limited ability to generalize from smaller runs to larger runs.

The results of the Palmetto/Nautilus experiment are given in Figure 4.7, following the same visual semantics as the core resource prediction experiments. In this experiment, the Nautilus runs consisted only of a few small runs and several large runs, so only the proportion of Palmetto runs was varied. Only the model that was trained on 100% of the Palmetto runs was able to reliably predict the runtime of the large Nautilus runs. Unsurprisingly, it is much more difficult to extrapolate runtime from Palmetto to Nautilus than to extrapolate between different phases of Palmetto. Prediction accuracy can be improved with more training data, as the figure shows, and potentially with additional Minibench performance metrics that provide a better representation of the source and target platforms. Investigating additional performance metrics is an important subject for future research.

As an aside, it is worth exploring why certain models in each experiment tended to either overestimate or underestimate the runtime. In the first experiment, the models tended to overestimate when the training data consisted of more P100 runs. Since the V100 runs are significantly shorter than the P100 runs, the training set in this situation consists of much higher runtimes than the test set, which causes the model to be biased towards overestimation. Similarly, the models in the second experiment were trained on P100 and V100 runs, but evaluated on Nautilus runs that used lower-performing GPUs such as a GTX 1080 or TITAN Xp. As a result, these models tended to underestimate the runtime. These observations suggest that, in cases where the source and tar-



(a) Expected vs predicted runtime



(b) Expected vs predicted cost

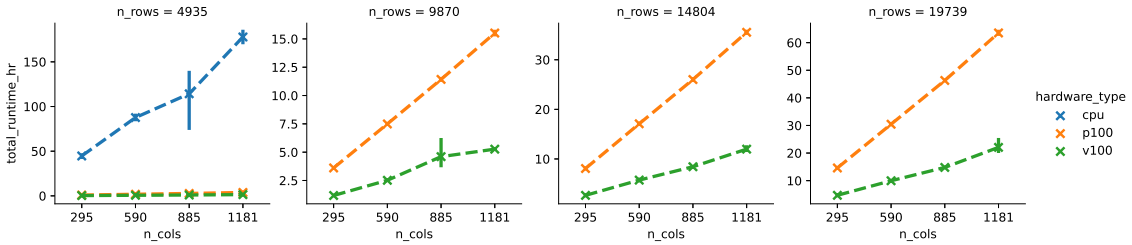
Figure 4.8: Expected vs predicted target values for each model in the Palmetto/Google experiment.

get platforms are very different, the model will be biased towards the source platform because the training data will include more examples from the source platform. Addressing this data imbalance is challenging when there are only a few training samples from the target platform, however as mentioned before, there are many strategies for mitigating data imbalance in the training data.

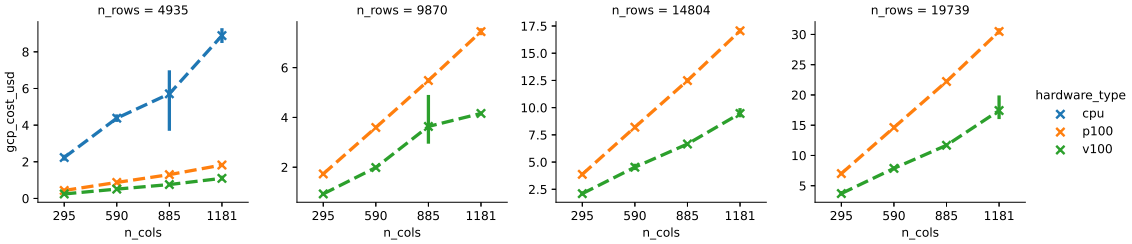
4.4 Cost Prediction

The results of the Palmetto/Google experiment are shown in Figure 4.8, following the same visual semantics as the core resource prediction experiments. This experiment was identical to the Palmetto/Nautilus experiment; the runs that were performed on Nautilus were also performed on GCP. In this case, the model is able to achieve better coverage with less training data. It is not surprising that the GCP runs are easier to predict than the Nautilus runs, because GCP is both more homogeneous than Nautilus and more similar to Palmetto. These results suggest that Palmetto is an ideal proxy for predicting resource usage and cost of workflows on GCP, due to hardware similarities such as P100 and V100 GPUs.

We used the runtime data from the Palmetto runs to estimate the cost of running the



(a) Total runtime



(b) Estimated GCP cost

Figure 4.9: Total runtime and estimated GCP cost of several Palmetto KINC runs. Error bars denote standard error across different values of the "chunks" parameter.

same experiments on GCP. The total runtime and cost of these runs are shown in Figure 4.9, respectively. The CPU runs were only collected for the smallest datasets and were not used in any other experiments. These plots clarify the comparative cost of different node types; in particular, they show that V100 GPUs are are both faster and more cost efficient than CPUs and P100 GPUs, even for small datasets. Additionally, the small error bars indicate that the "chunks" parameter, which determines the number of parallel tasks, has very little effect on cost. Using more chunks decreases runtime if the tasks are actually executed in parallel, but the cost is the same because the KINC / similarity_chunk process is embarrassingly parallel. This approach can be used to determine the most cost-effective way to run workflows in the cloud without running anything in the cloud beforehand. The source and target node types should be similar when performing a comparative analysis; this example depends on the fact that both Palmetto and GCP have P100 and V100 GPUs. Furthermore, if an accurate cost estimate is also desired, then the full cost prediction analysis should be used so that the runtimes are extrapolated to the target platform. Based on the Palmetto/Google experiment, this example is likely a slight overestimate of the true cost.

4.5 Summary

In this chapter, we have demonstrated several prediction capabilities with Tesseract. Fundamentally, Tesseract is a data collection and data analysis tool, and our experiments only highlight a set of prediction problems that are most relevant to scientific workflows and resource provisioning. We developed a generic approach to resource prediction, based on standard machine learning models and confidence intervals, that was effective for a diverse collection of scientific workflows. We investigated the impact of training set size and developed a basic anomaly detection mechanism, and we discussed their relevance to online training and prediction. Finally, we developed an approach for the more challenging problem of predicting the resource usage – and ultimately the cost – of running a workflow on a target platform with training data from a source platform. We also discussed challenges, such as data imbalance, that require further investigation. In all of these experiments, we have demonstrated capabilities that are effective across applications and platforms and that address key challenges with resource provisioning from the perspective of a domain scientist. In the final chapter, we summarize the work as a whole, and we discuss potential research directions for the future.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Many prominent applications of AI focus on maximizing capability and capacity, that is, employing bigger models, more data, more computational resources, and more technical expertise to achieve as little error as possible. A complementary but similarly difficult class of problems, is to employ small models with minimal training data to achieve acceptably low error, using a methodology that requires minimal technical expertise. These constraints are shared by many small problems that scientists and other non-expert users face when using data-intensive computing systems. This dissertation focused on one such problem – predicting resource usage of scientific workflows. We laid out the history of resource prediction and motivated the need for tools that are generic and easy to use. We proposed Tesseract, a semi-automated tool for predicting resource usage of any application on any platform. We described the design, implementation, and usage of Tesseract, including workflow annotation, performance data collection, and model training. We demonstrated the core resource prediction capability of Tesseract on a diverse set of scientific workflow. Finally, we leveraged this core capability to demonstrate the use of Tesseract for anomaly detection, cross-platform runtime prediction, and cost prediction. The source code for Tesseract is available at `https://github.com/bentsherman/tesseract` under the MIT license.

5.2 Future Work

There are a number of research directions and software engineering efforts that may be derived from this work. The most straightforward efforts include investigating different applications, such as those from other science and engineering fields, and investigating other resource metrics, particularly networking metrics. Other efforts have been mentioned throughout this dissertation, such as exploring performance metrics in more detail and addressing imbalance in trace datasets. Tesseract may be applicable to the problem of predicting and characterizing different types of failures (e.g. incorrect task configuration, corrupt input data, network reliability, hardware failures). Similarly, predicting the queue time of submitted tasks is useful for estimating the makespan (total runtime) of a workflow run on some computing platforms. The feasibility of any such capability is a function of acquiring the appropriate inputs and outputs, and there is a rich literature of studies aimed at solving each of these problems. Finally, an important goal for making Tesseract highly usable is the implementation of online training and prediction. Tesseract should ideally operate as part of a service that automatically collects resource usage data from workflow runs, trains and retrains models, and provides resource estimates to workflow runs in real time. In this scenario, given an annotated pipeline, the only step that requires user input is labeling mispredictions as anomalies or model errors. We have developed a web service called Nextflow-API [54] which includes some of these capabilities, and there are other emerging services around Nextflow, such as Nextflow Tower, that are working towards similar goals. We are hopeful that these kinds of tools and services will reduce the friction of data-intensive computing and enable scientists to make further advances across many science domains.

Appendices

Appendix A: Trace Datasets

This appendix describes the trace datasets that were generated for this dissertation. Table 2 lists the input features for each workflow process associated with the selected prediction targets, as well as the number of samples in each trace dataset. Unless specified otherwise, all workflow runs were performed on the Palmetto cluster.

GEMmaker. GEMmaker downloads sequence read archive (SRA) data from the NCBI SRA repository [45] and produces a gene expression matrix (GEM). To generate trace data for GEMmaker, we processed 1,000 Arabidopsis datasets that were selected randomly NCBI SRA. These runs vary in size, complexity, and origin, as they were produced by many different labs. From these runs, 768 samples were processed successfully, while the remaining runs failed at different stages in the pipeline. Nearly all of these failures were caused by missing or invalid sample metadata from NCBI.

Gene Oracle. Gene Oracle takes as input a GEM and a list of gene sets. To generate trace data for Gene Oracle, we used the kidney GEM from the unified GTEx-TCGA data [67]. We generated six gene set lists with $\{32, 64, 128, 256, 512, 1024\}$ gene sets, respectively. Each gene set contained anywhere from 5 to 20 genes, all drawn from the unified GEM itself. We then ran Gene Oracle with the kidney GEM and each gene set list, as well as with $\{1, 2, 4, 8, 16\}$ parallel processes, for a total of 30 workflow runs.

HemeLB. To generate trace data for HemeLB, we used six unstructured cerebral aneurysm (CA) geometries from the AneuriskWeb repository [4], as well as a simple cylinder geometry. These geometries vary in site count and sparsity. Each geometry was processed by HemeLB multiple times under a variety of input conditions, as well as three independent trials for each unique run, for a total of 324 runs, of which 320 were successful.

KINC. Several trace datasets were generated for KINC throughout the duration of this

Name	Genes	Samples
bile-duct	18,764	40
bladder	19,427	390
breast	19,738	1,181
cervix	19,316	272
colon	19,096	665
esophagus	19,629	853
head-neck	19,132	502
kidney	19,216	929
liver	18,764	458
lung	19,648	1,415
prostate	19,046	580
rectum	19,096	97
salivary	19,132	55
stomach	19,969	605
thyroid	19,239	812
uterus	19,316	293
yeast	7,050	188

Table 1: Datasets used to generate KINC runs

study. For the core resource prediction experiments, we used a yeast GEM to generate a range of subset GEMs. For the MPI execution mode (KINC / similarity_mpi), 10 subset GEMs were generated. Each GEM was processed multiple times under a variety of input conditions, as well as three independent trials for each unique run, for a total of 450 workflow runs, of which 436 were successful. For the chunk-and-merge execution mode (KINC / similarity_chunk), 8 subset GEMs were generated. Each GEM was processed multiple times under a variety of input conditions, for a total of 96 workflow runs. For the cross-platform and cost prediction experiments, we used the breast GEM from the unified GTEx-TCGA data [67] to generate a range of 16 subset GEMs. Each GEM was processed multiple times under a variety of input conditions, using the chunk-and-merge execution mode, for a total of 194 workflow runs. These runs were used as the training data. Each GEM from the unified GTEx-TCGA data was processed both on Nautilus and Google Cloud Platform (GCP), and these runs were used as the evaluation data for the cross-platform and cost prediction experiments, respectively. Table 1 lists all of the GEMs that were used to generate KINC runs.

TSPG. TSPG takes as input a GEM of training data, a GEM of perturb data, and a list of gene sets. To generate trace data for TSPG, we used the thyroid GEM from the unified GTEx-TCGA data [67]. We generated nine train/perturb splits ranging from 10/90 to 90/10, as well as a

Process	Input Features	No. Samples
GEMmaker / download_runs	n_remote_run_ids, n_spots	774
GEMmaker / fastq_dump	sra_bytes	769
GEMmaker / fastq_merge	fastq_lines	768
GEMmaker / fastqc_1	fastq_lines	768
Gene Oracle / phase1_fg	chunks, gmt_lines, gmt_genes	186
HemeLB	n_sites, hardware_type, np	320
KINC / similarity_chunk	n_rows, n_cols, hardware_type, chunks	720
KINC / similarity_mpi	n_rows, n_cols, hardware_type, np	436
KINC / extract	n_rows, n_cols, ccm_bytes, cmx_bytes	96
TSPG / train_target	n_genes, n_train_samples	700
TSPG / perturb	n_genes, n_train_samples, n_perturb_samples	700

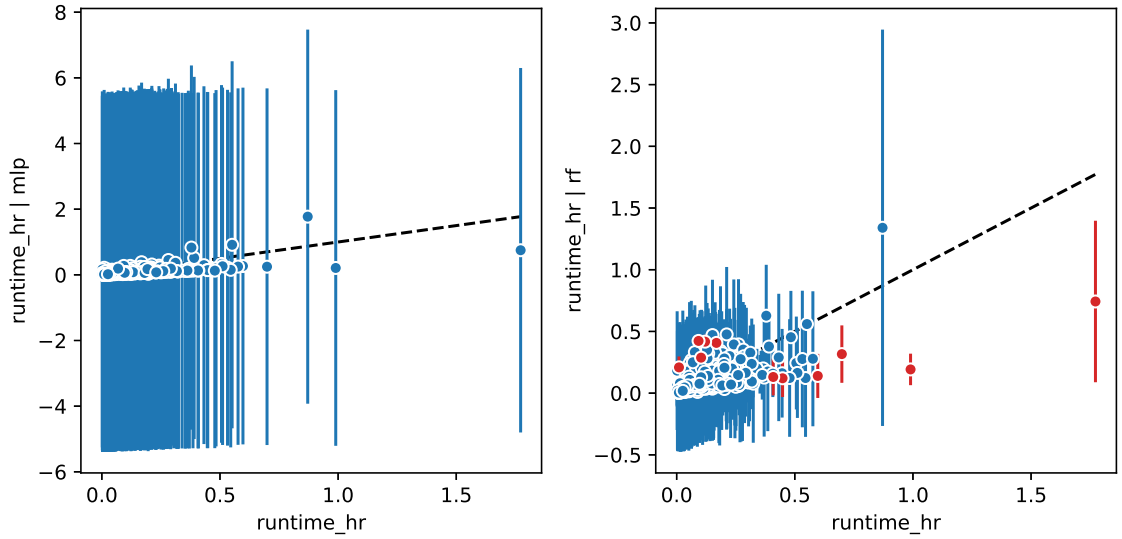
Table 2: Input features that were defined for each selected process

list of 100 gene sets drawn from the thyroid GEM itself. We then ran TSPG on each train/perturb split and the gene set list, for a total of 9 workflow runs, 7 of which were successful.

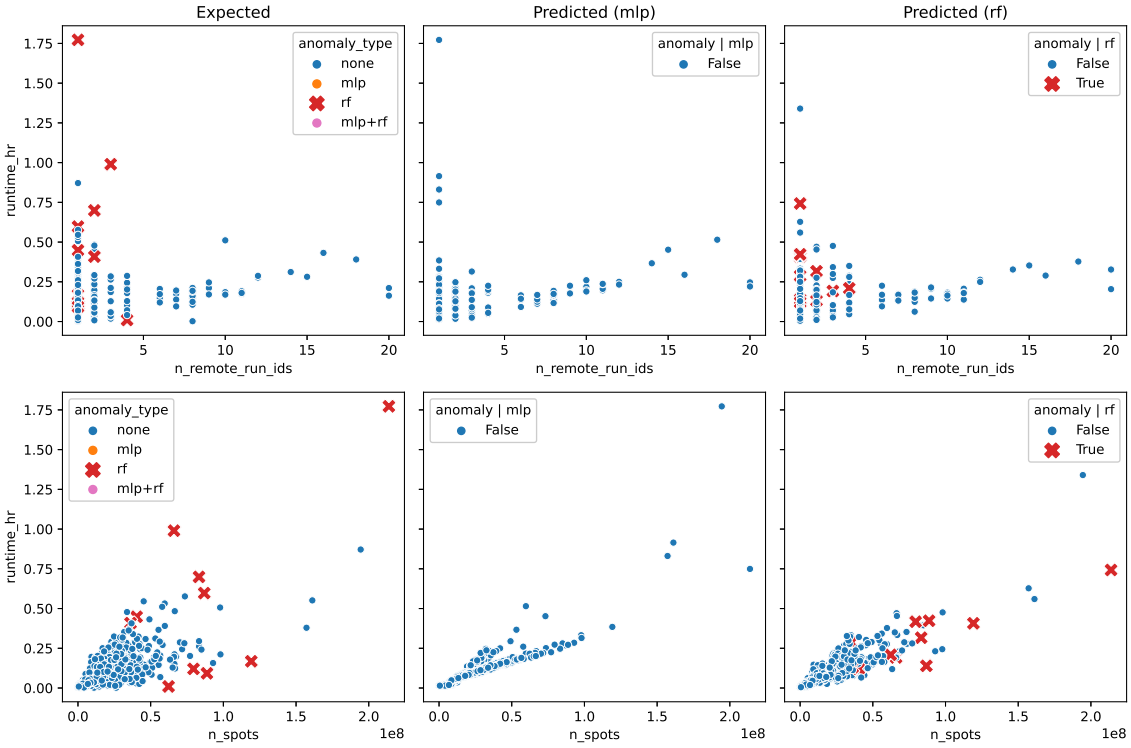
Appendix B: Resource Prediction Results

This appendix contains the full results for the resource prediction experiments. For each prediction target that was selected, we provide the following: (a) scatter plots of expected vs predicted values for the neural network (MLP) and random forest (RF) models; (b) expected and predicted target values plotted in terms of each input feature. Each scatter plot in (a) has a dashed black line denoting equality, and each point and vertical bar is a point prediction with corresponding 95% confidence interval. Points below the black line are predictions that were less than the target value, while points above the black line are predictions that were greater than the target value. Predictions that were identified as anomalies (anomaly score of 0.997 or greater) are colored red. Each row of plots in (b) shows the predicted and expected target value in terms of a single input feature. The rows in combination provide a full comparison between the original data distribution and the distribution of each model. Additionally, target values are marked in yellow, red, or pink if they were identified as anomalous by the neural network, random forest, or both, respectively. Each anomaly is marked across all three plots, in order to show the discrepancy between the predicted and actual resource usage.

This appendix also contains the results for the training set size experiments. For each prediction target, a neural network was trained and evaluated for each train/test split from 10/90 to 90/10. Each figure shows the prediction error (MAPE) and coverage error for the given prediction target.

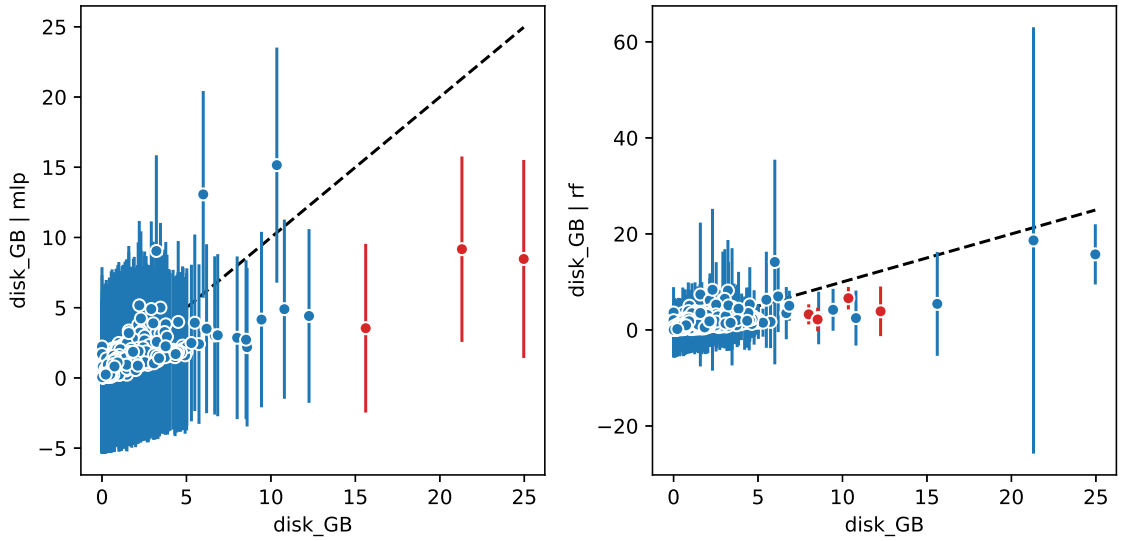


(a) Expected vs predicted scatter plots

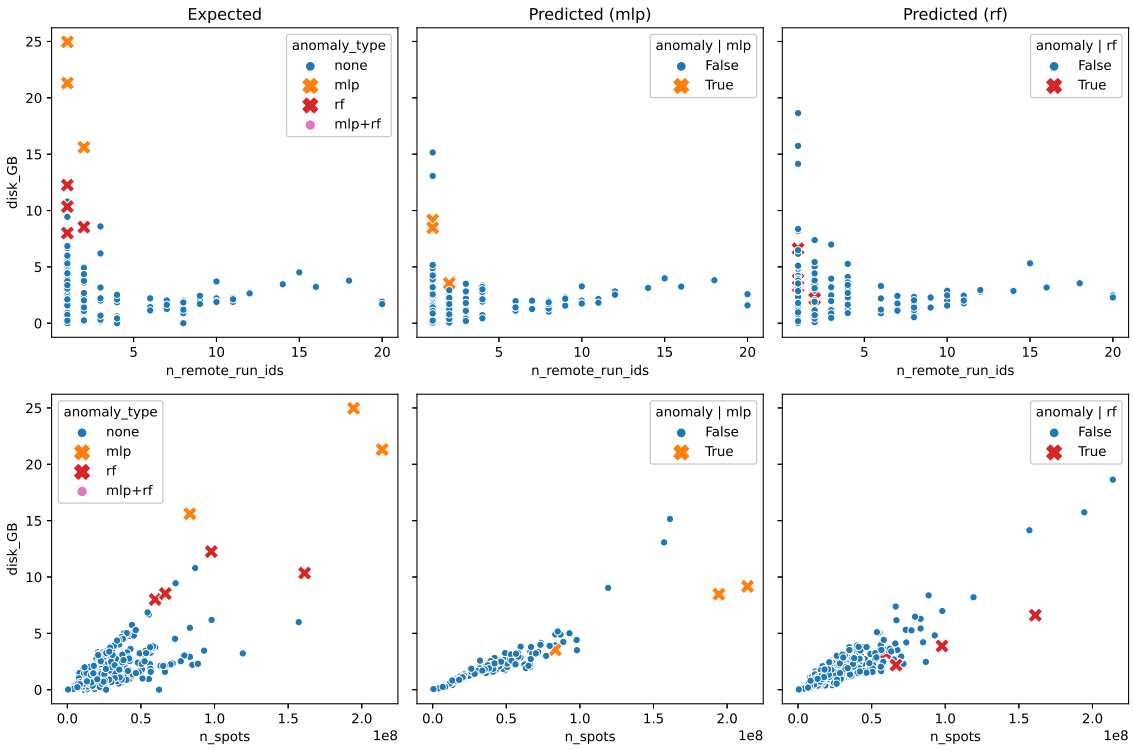


(b) Expected and predicted target values in terms of each input feature

Figure 1: Resource prediction results for GEMmaker / download_runs runtime.

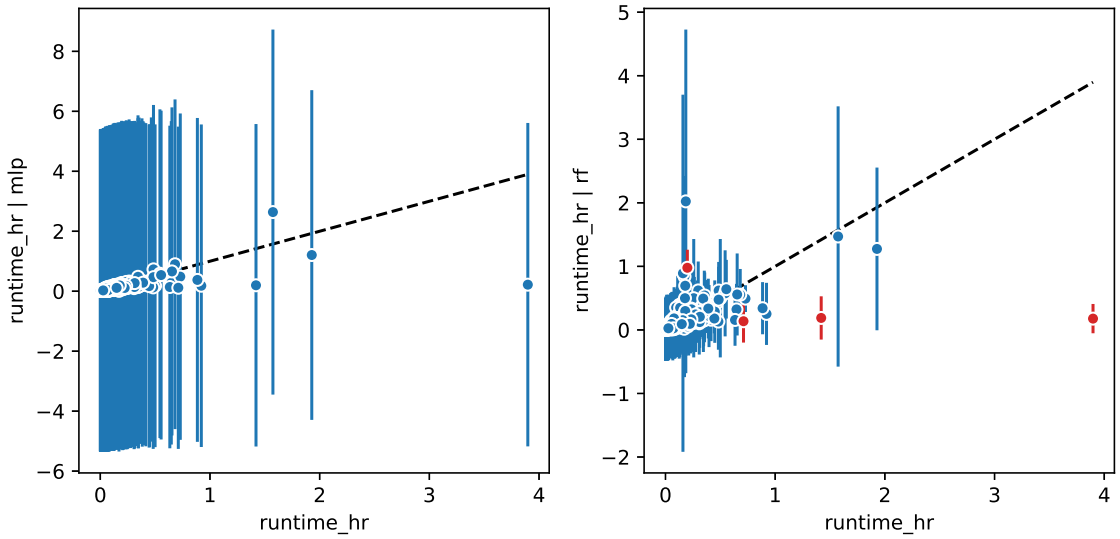


(a) Expected vs predicted scatter plots

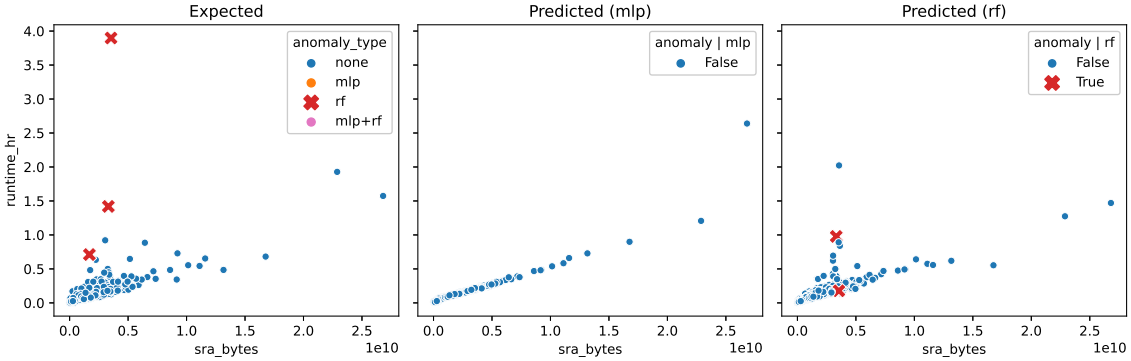


(b) Expected and predicted target values in terms of each input feature

Figure 2: Resource prediction results for GEMmaker / download_runs disk usage.

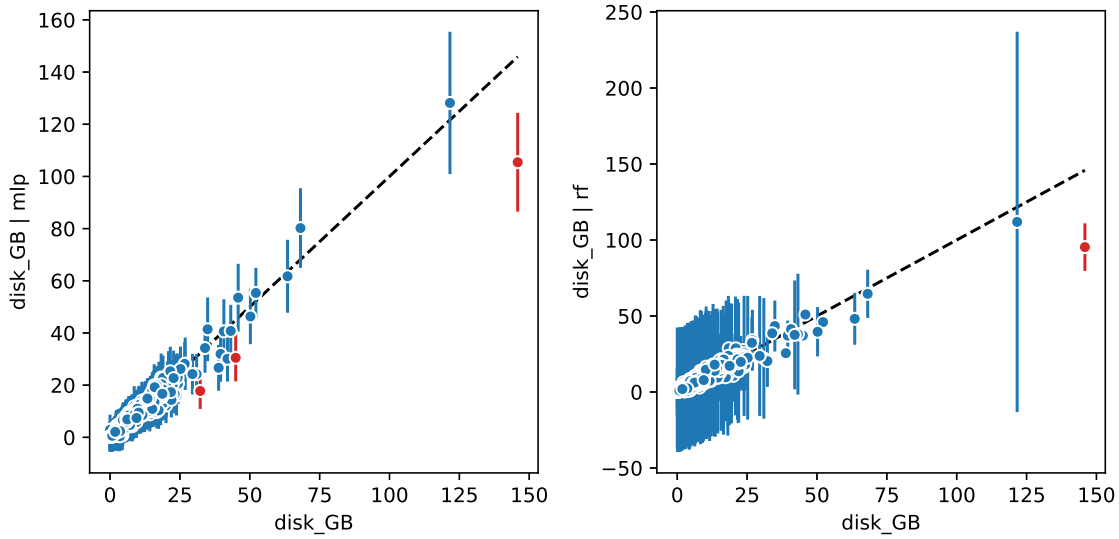


(a) Expected vs predicted scatter plots

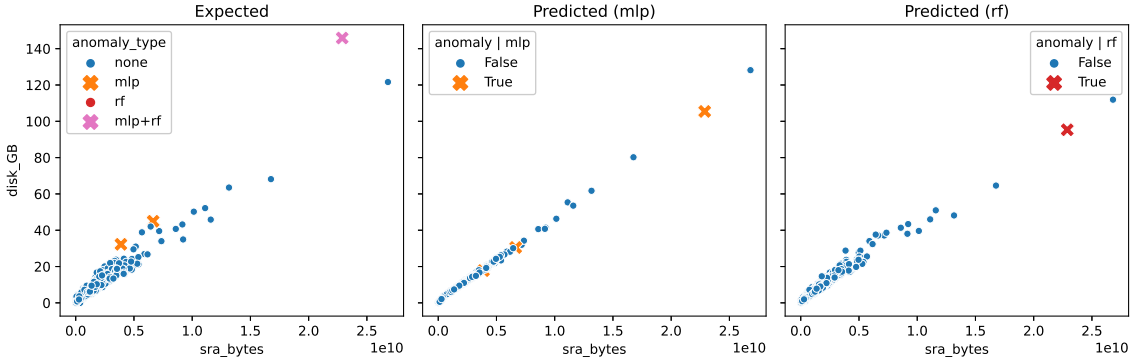


(b) Expected and predicted target values in terms of each input feature

Figure 3: Resource prediction results for GEMmaker / fastq_dump runtime.

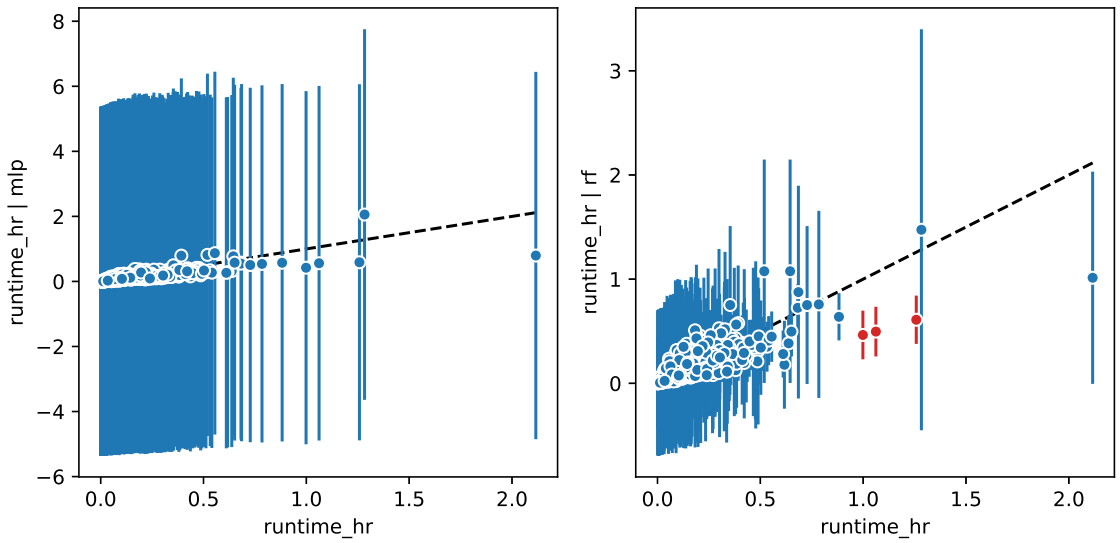


(a) Expected vs predicted scatter plots

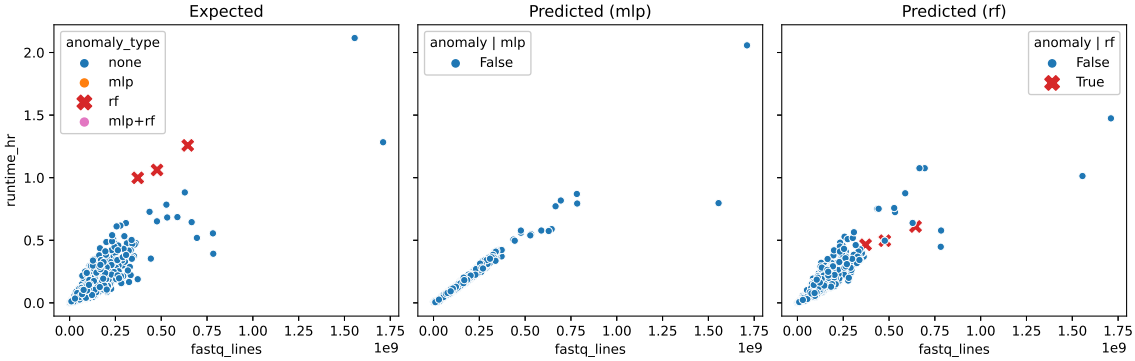


(b) Expected and predicted target values in terms of each input feature

Figure 4: Resource prediction results for GEMmaker / fastq_dump disk usage.

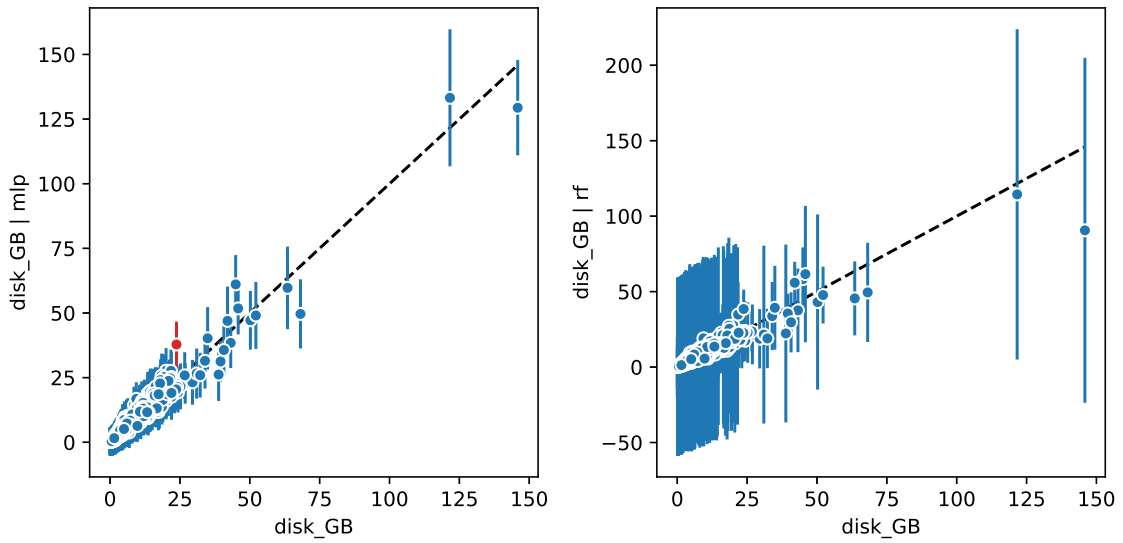


(a) Expected vs predicted scatter plots

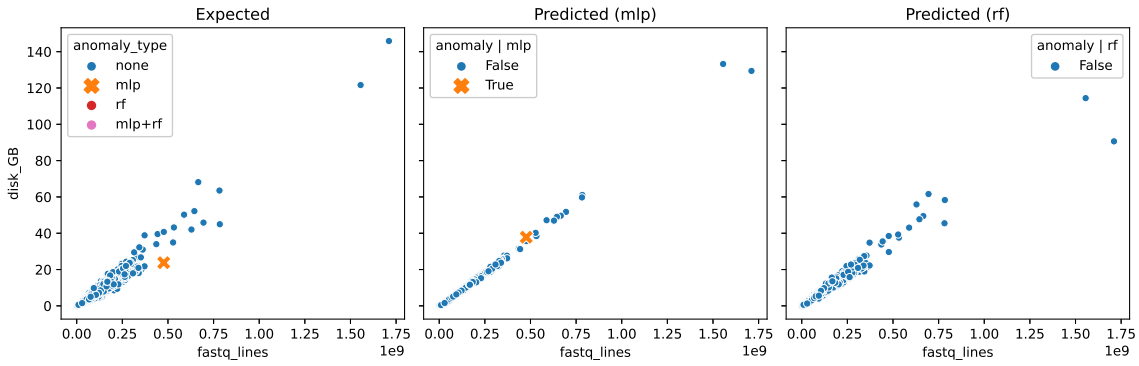


(b) Expected and predicted target values in terms of each input feature

Figure 5: Resource prediction results for GEMmaker / fastq_merge runtime.

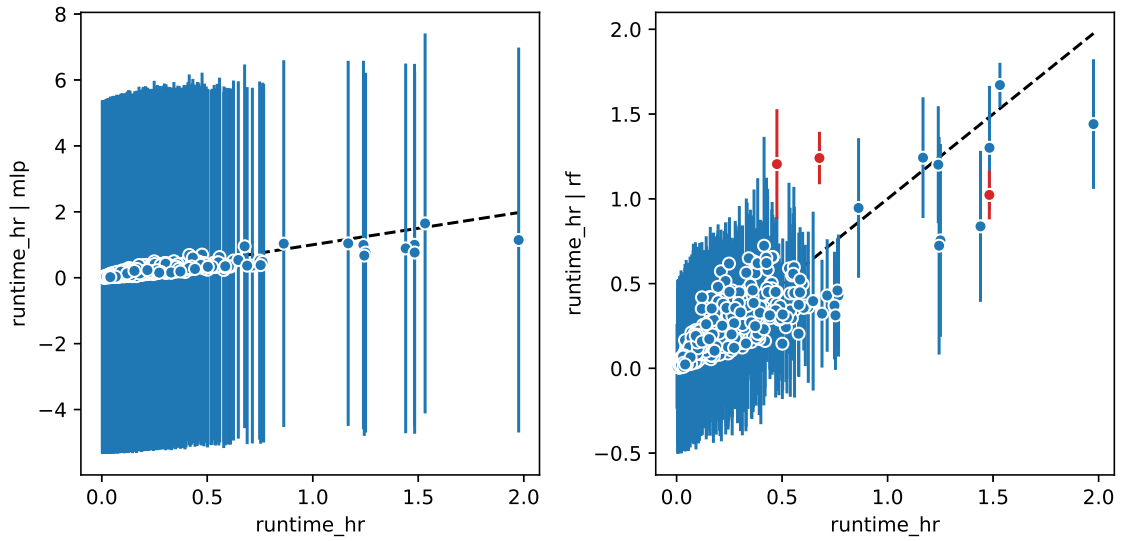


(a) Expected vs predicted scatter plots

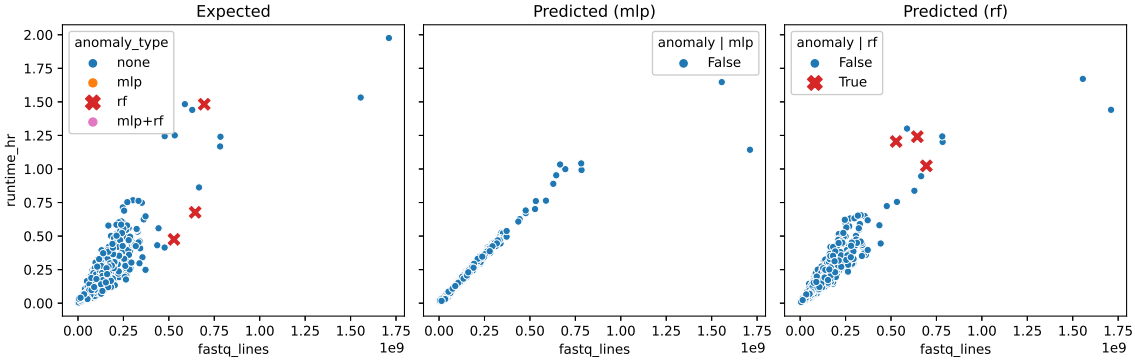


(b) Expected and predicted target values in terms of each input feature

Figure 6: Resource prediction results for GEMmaker / fastq_merge disk usage.

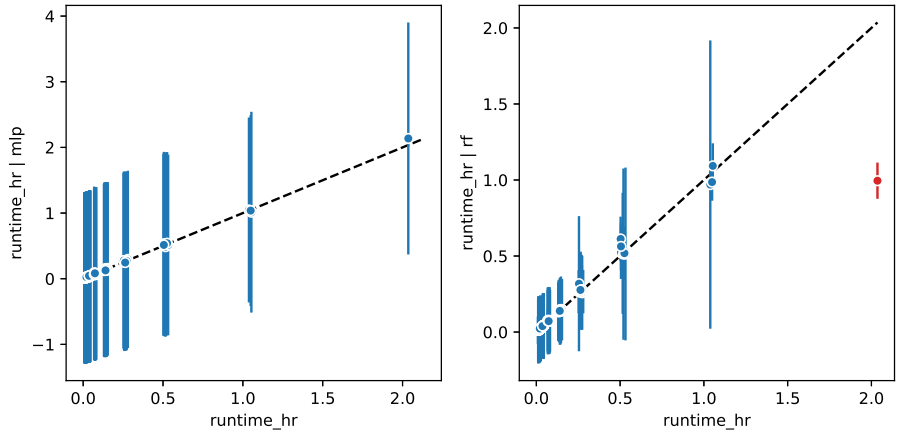


(a) Expected vs predicted scatter plots

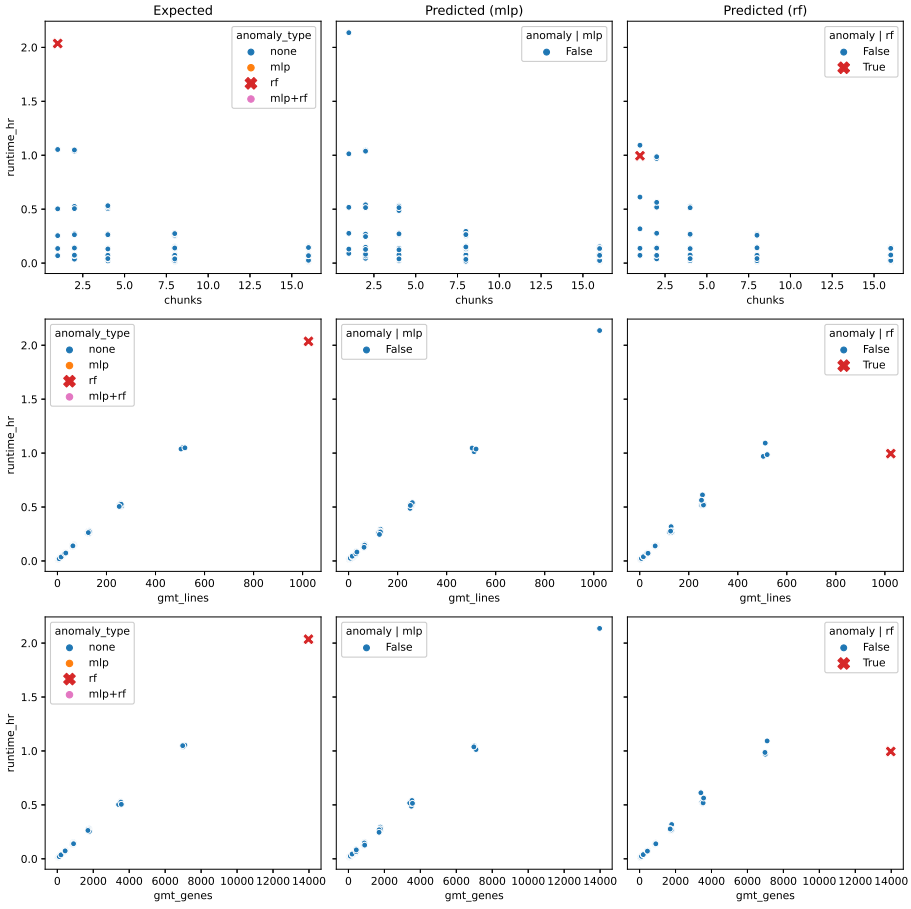


(b) Expected and predicted target values in terms of each input feature

Figure 7: Resource prediction results for GEMmaker / fastqc_1 runtime.

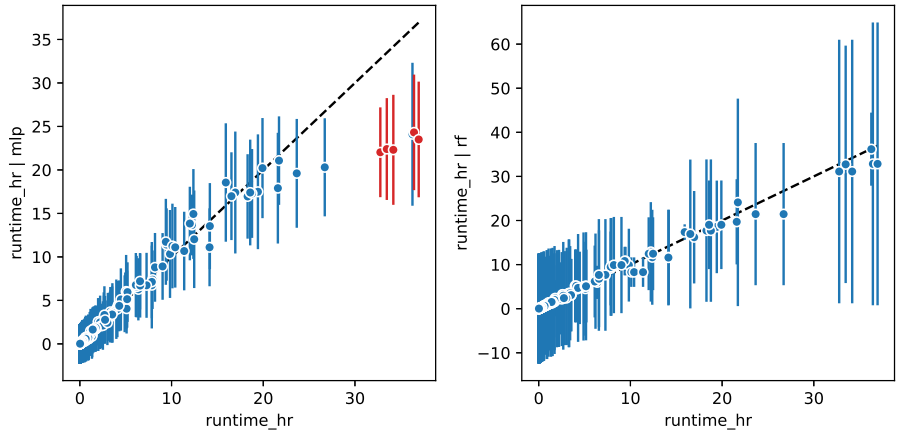


(a) Expected vs predicted scatter plots

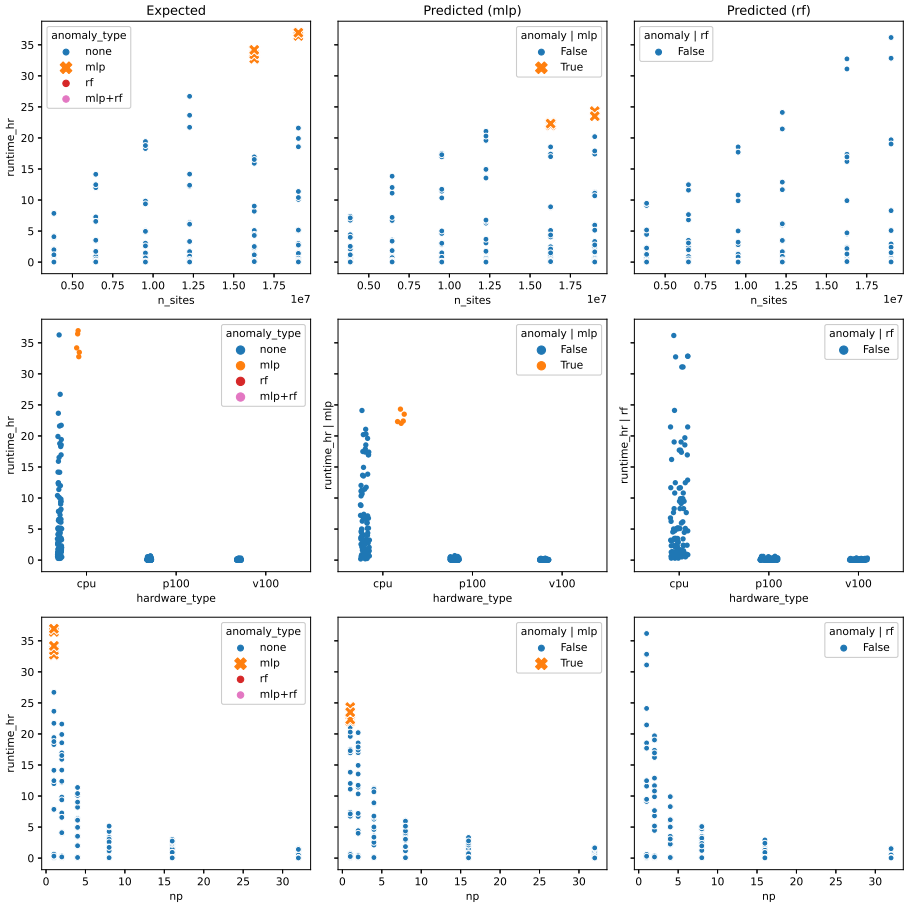


(b) Expected and predicted target values in terms of each input feature

Figure 8: Resource prediction results for Gene Oracle / phase1_fg runtime.

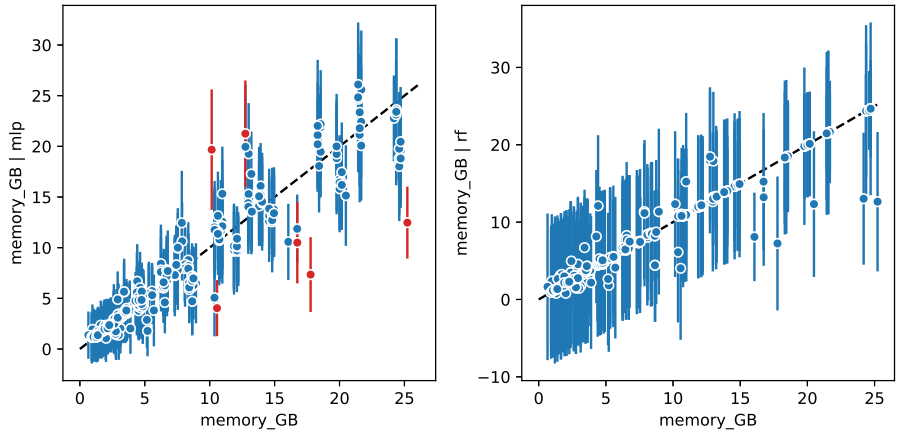


(a) Expected vs predicted scatter plots

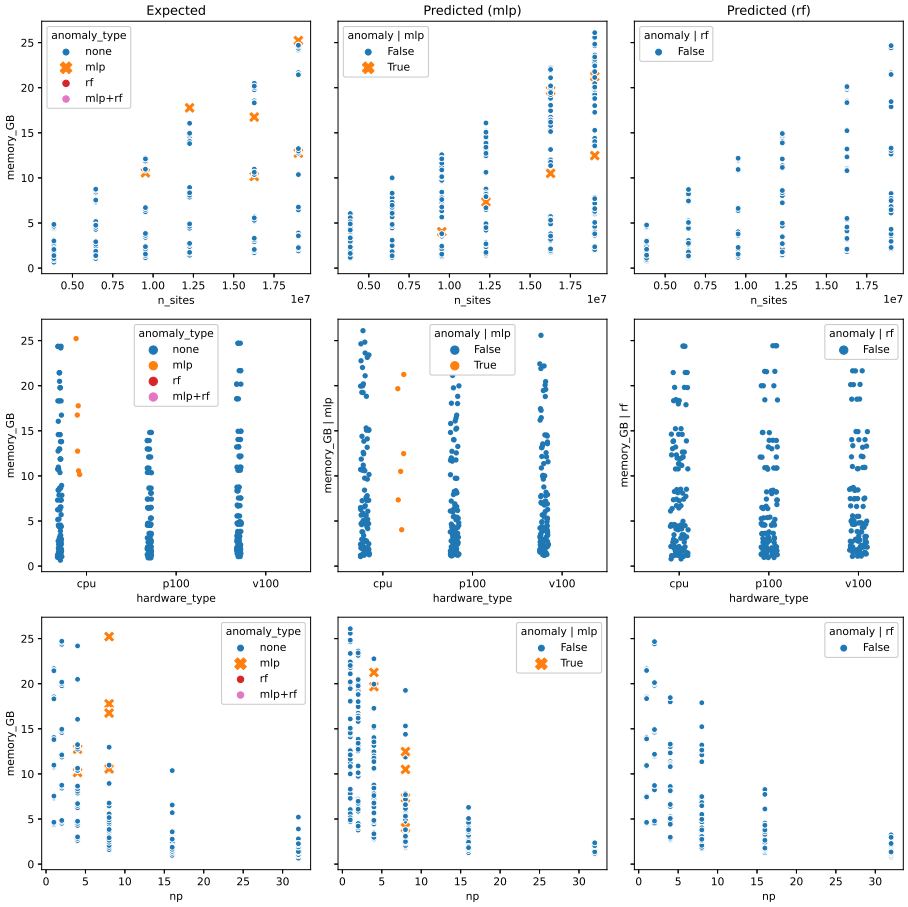


(b) Expected and predicted target values in terms of each input feature

Figure 9: Resource prediction results for HemeLB runtime.

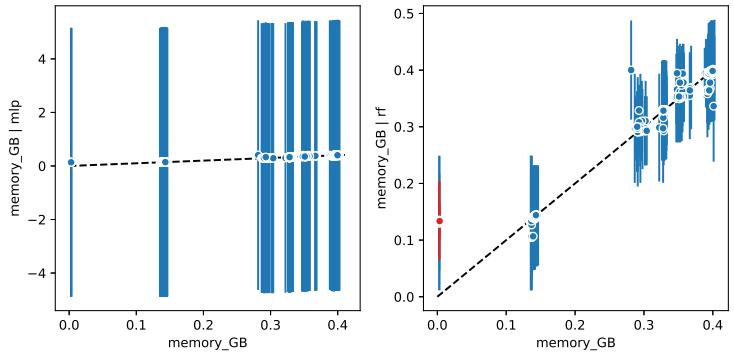


(a) Expected vs predicted scatter plots

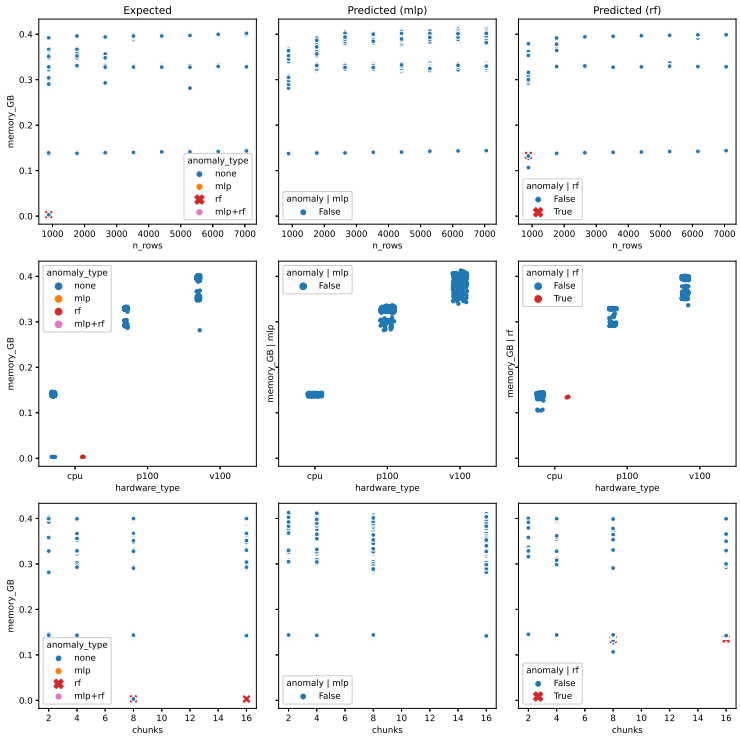


(b) Expected and predicted target values in terms of each input feature

Figure 10: Resource prediction results for HemeLB memory usage.

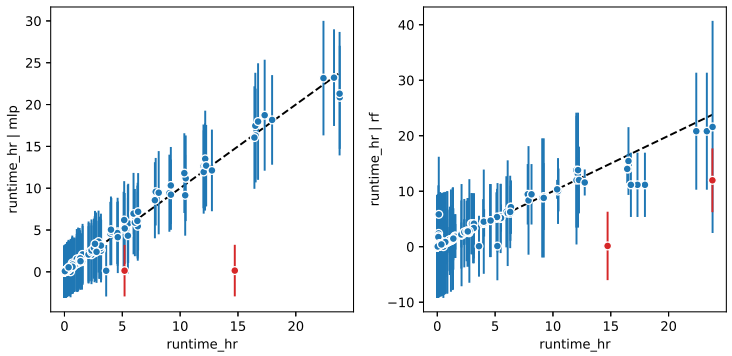


(a) Expected vs predicted scatter plots

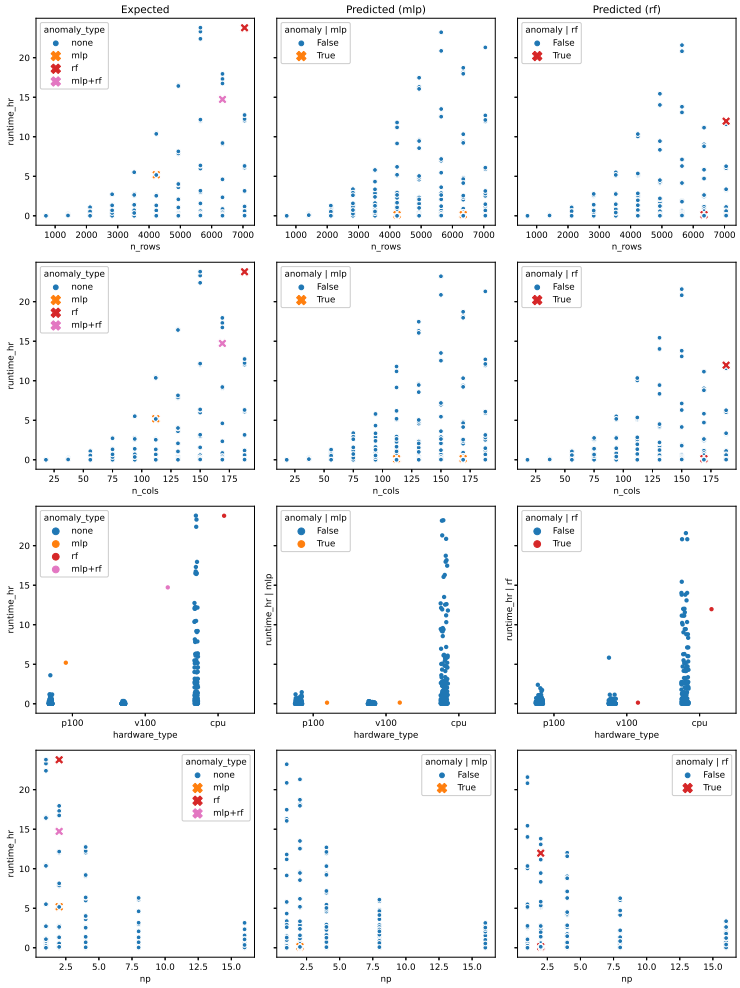


(b) Expected and predicted target values in terms of each input feature

Figure 11: Resource prediction results for KINC / similarity_chunk memory usage.

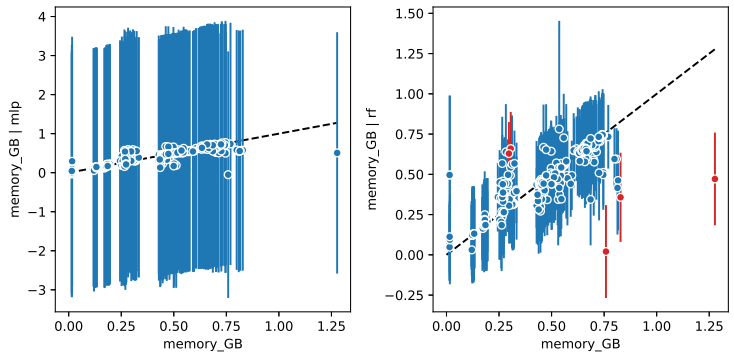


(a) Expected vs predicted scatter plots

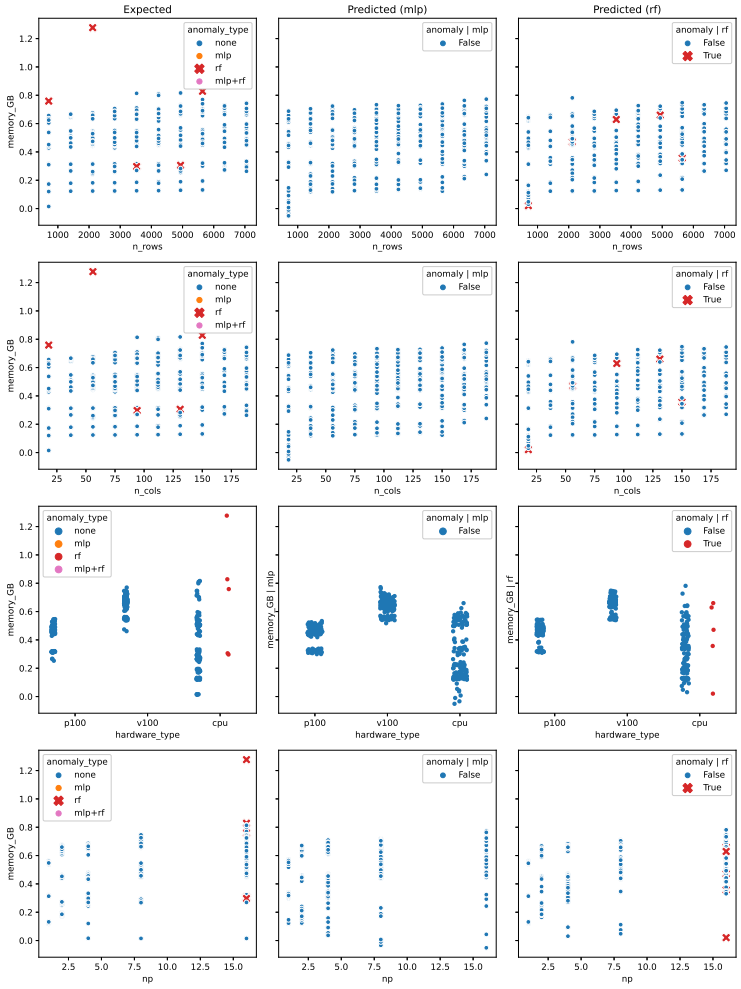


(b) Expected and predicted target values in terms of each input feature

Figure 12: Resource prediction results for KINC / similarity_mpi runtime.

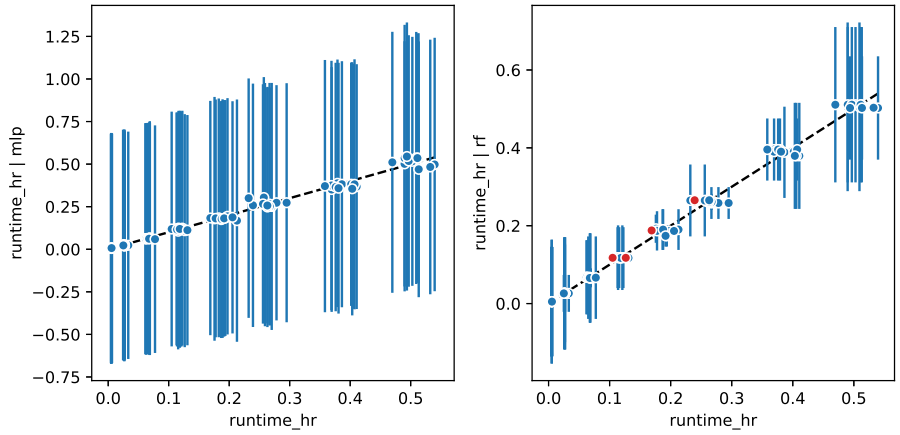


(a) Expected vs predicted scatter plots

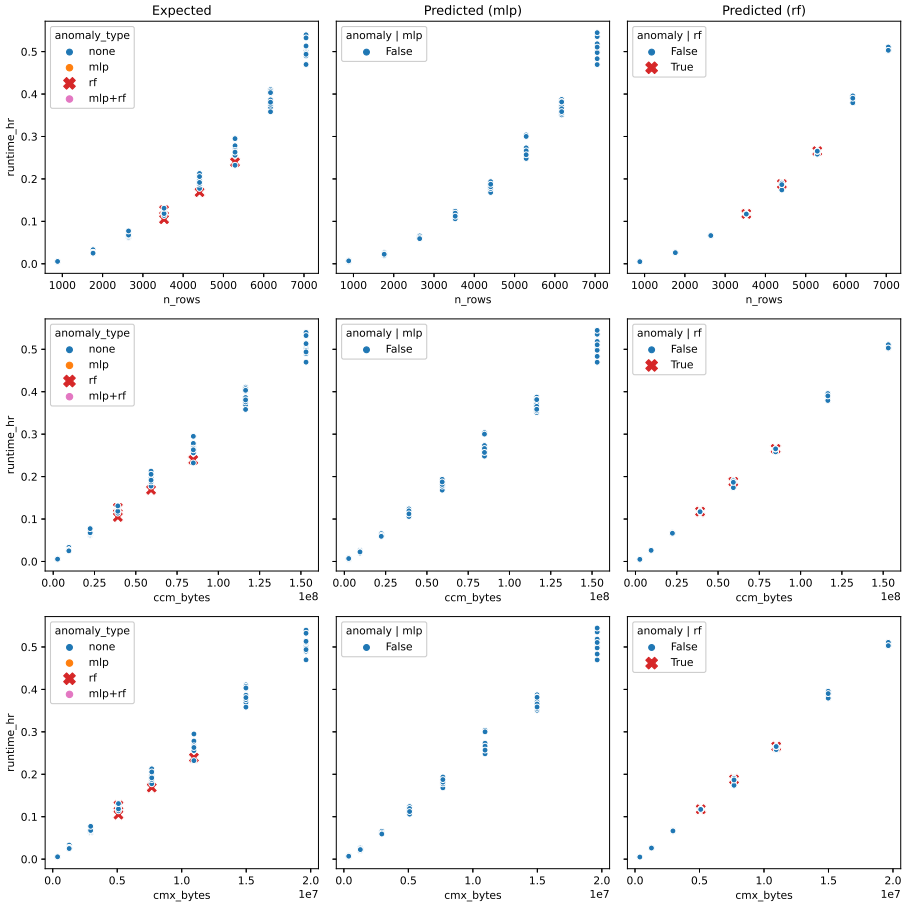


(b) Expected and predicted target values in terms of each input feature

Figure 13: Resource prediction results for KINC / similarity_mpi memory usage.

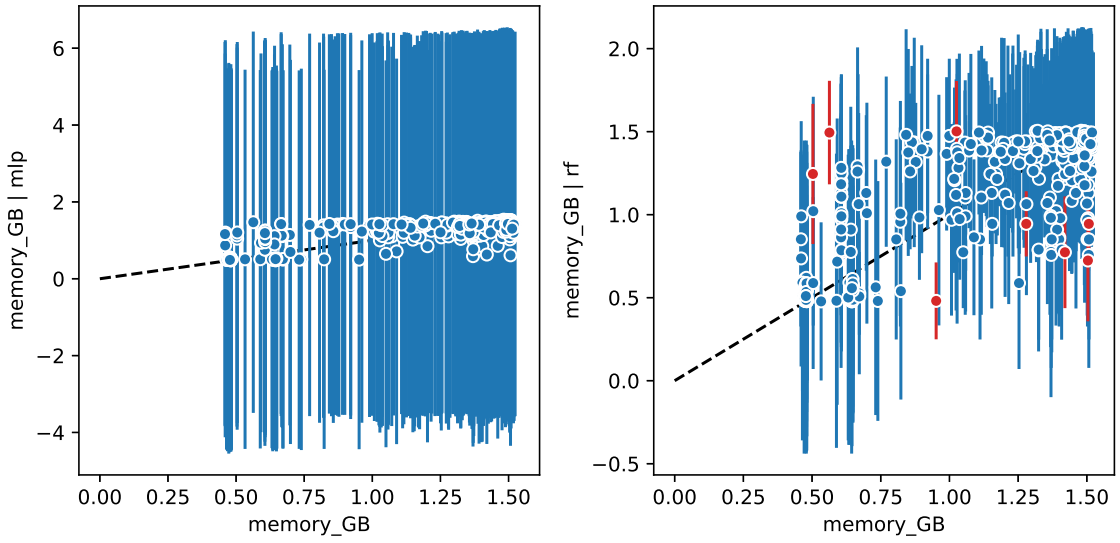


(a) Expected vs predicted scatter plots

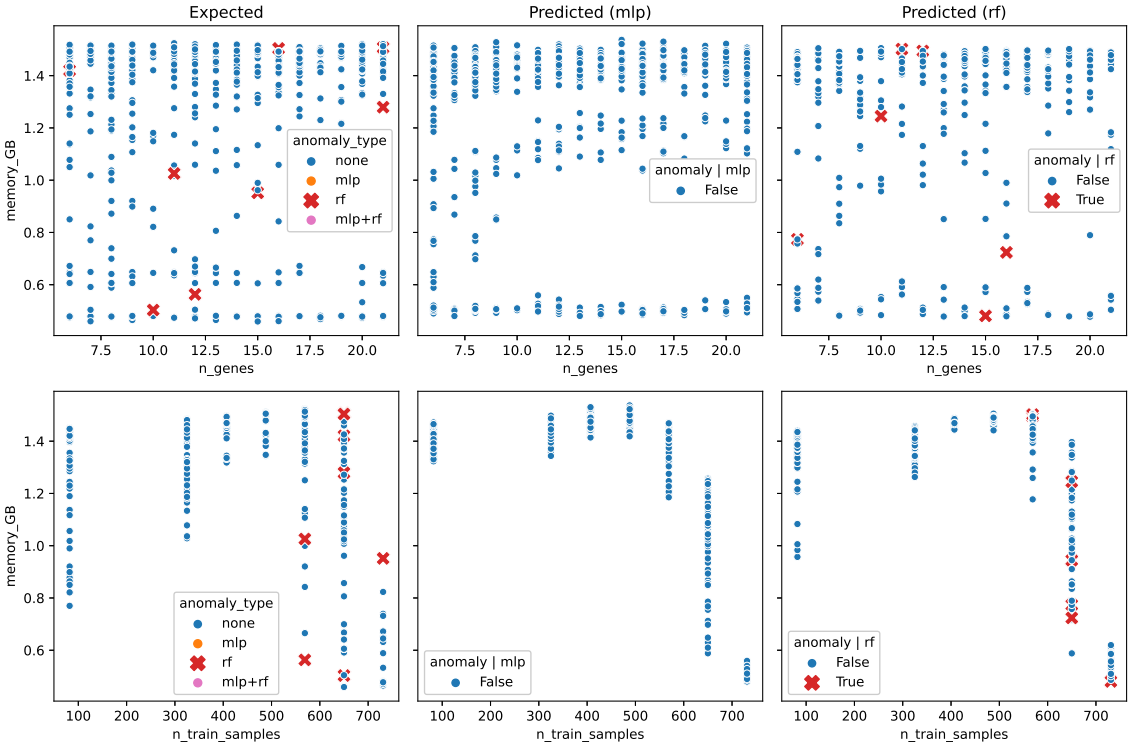


(b) Expected and predicted target values in terms of each input feature

Figure 14: Resource prediction results for KINC / extract runtime.

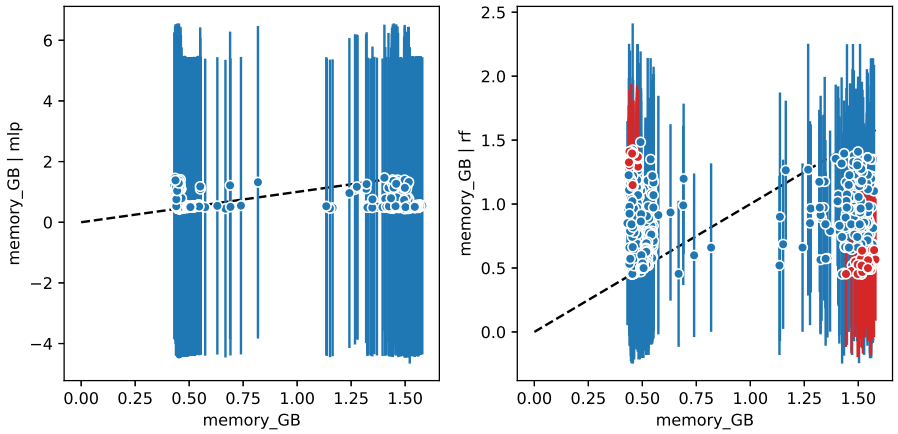


(a) Expected vs predicted scatter plots

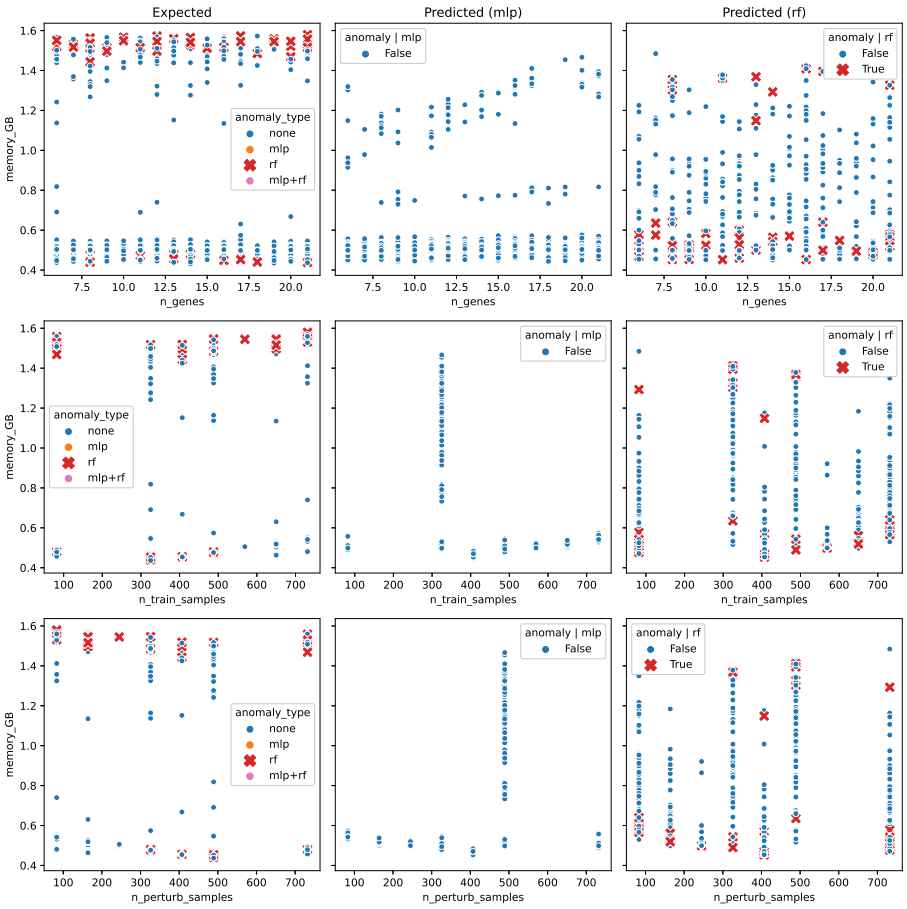


(b) Expected and predicted target values in terms of each input feature

Figure 15: Resource prediction results for TSPG / train_target memory usage.



(a) Expected vs predicted scatter plots



(b) Expected and predicted target values in terms of each input feature

Figure 16: Resource prediction results for TSPG / perturb memory usage.

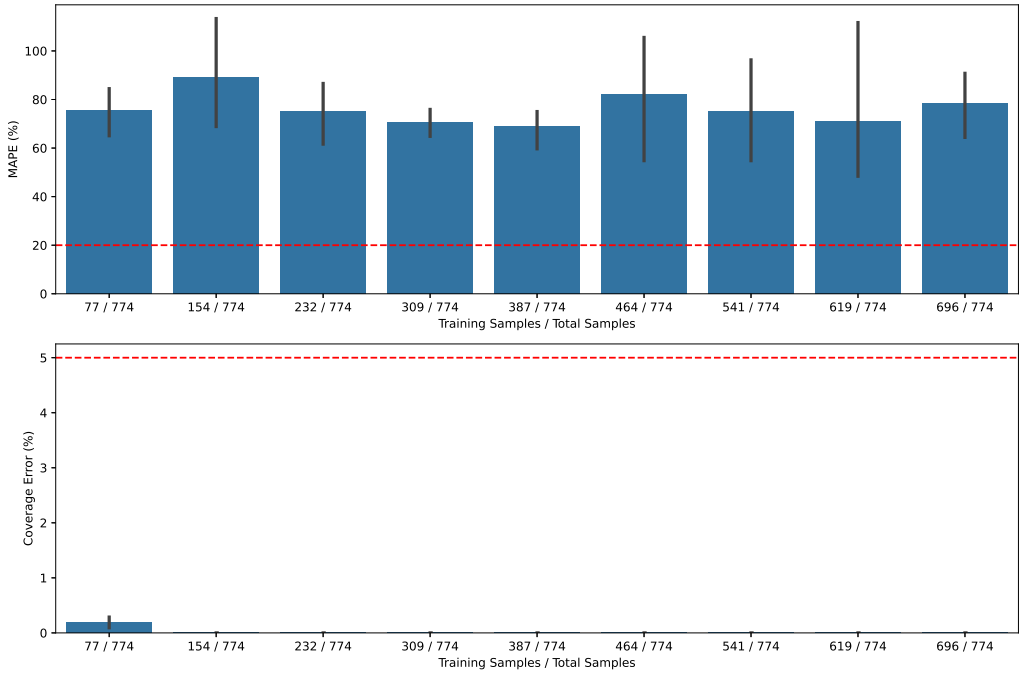


Figure 17: Training set size results for GEMmaker / download_runs runtime.

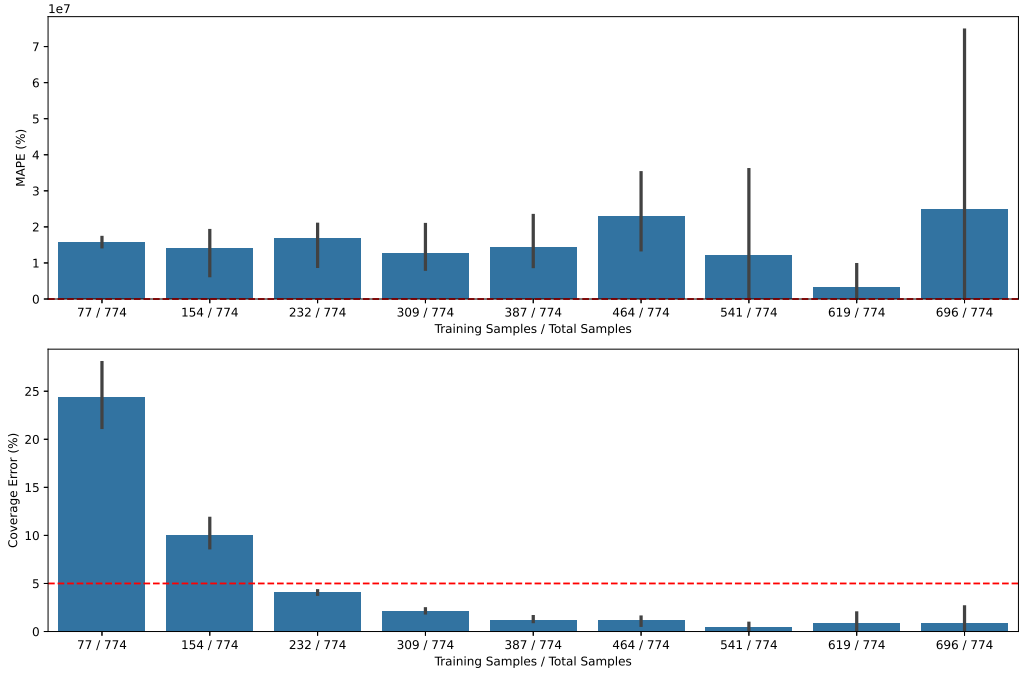


Figure 18: Training set size results for GEMmaker / download_runs disk usage.

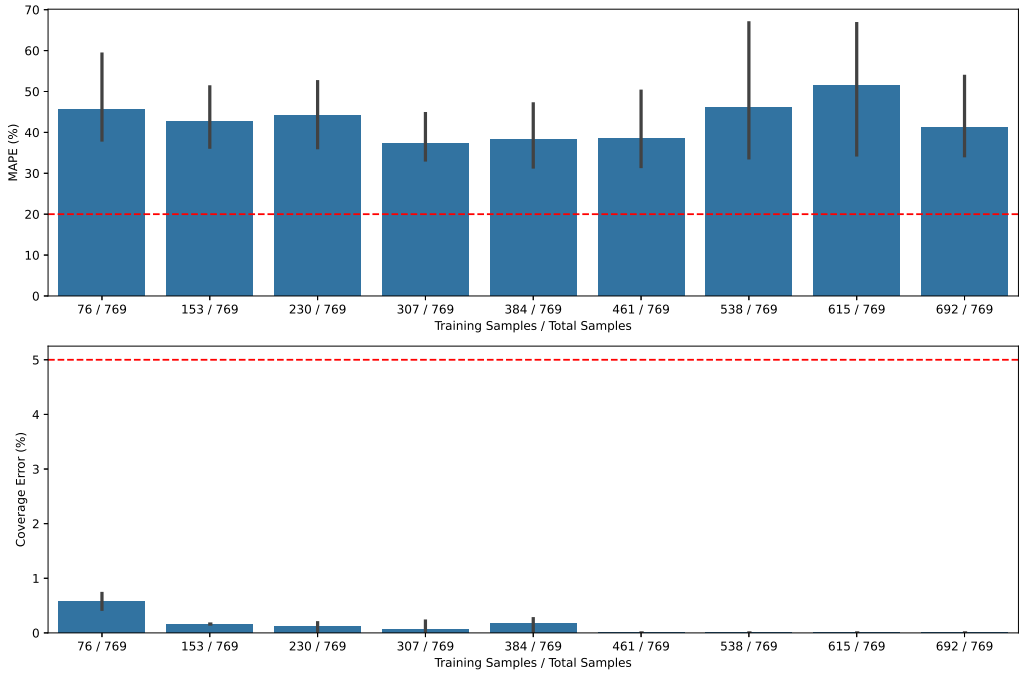


Figure 19: Training set size results for GEMmaker / fastq_dump runtime.

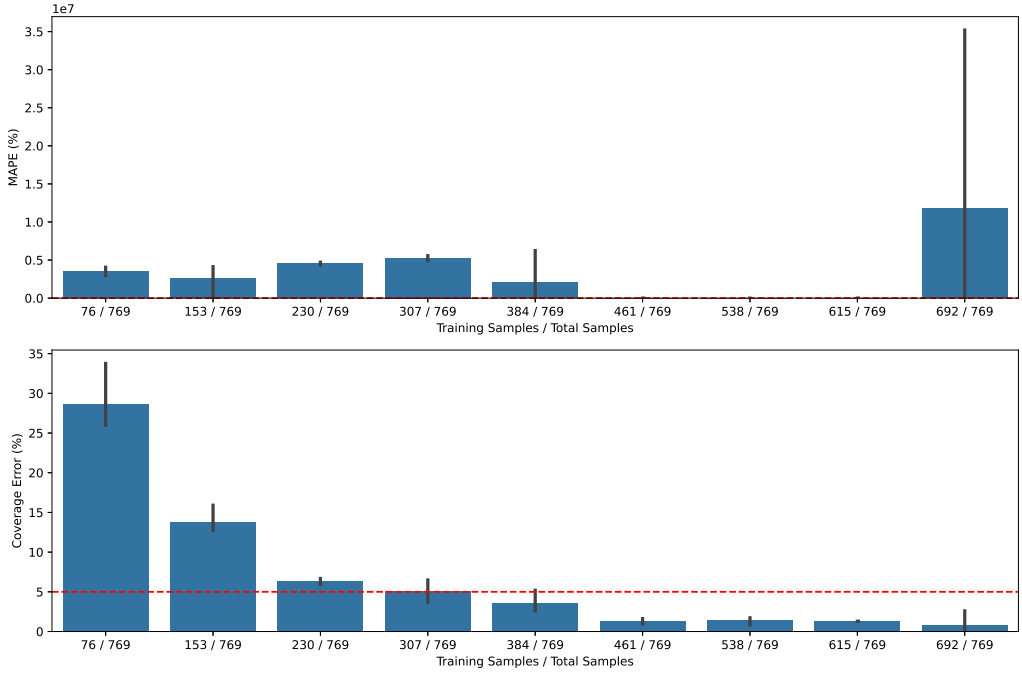


Figure 20: Training set size results for GEMmaker / fastq_dump disk usage.

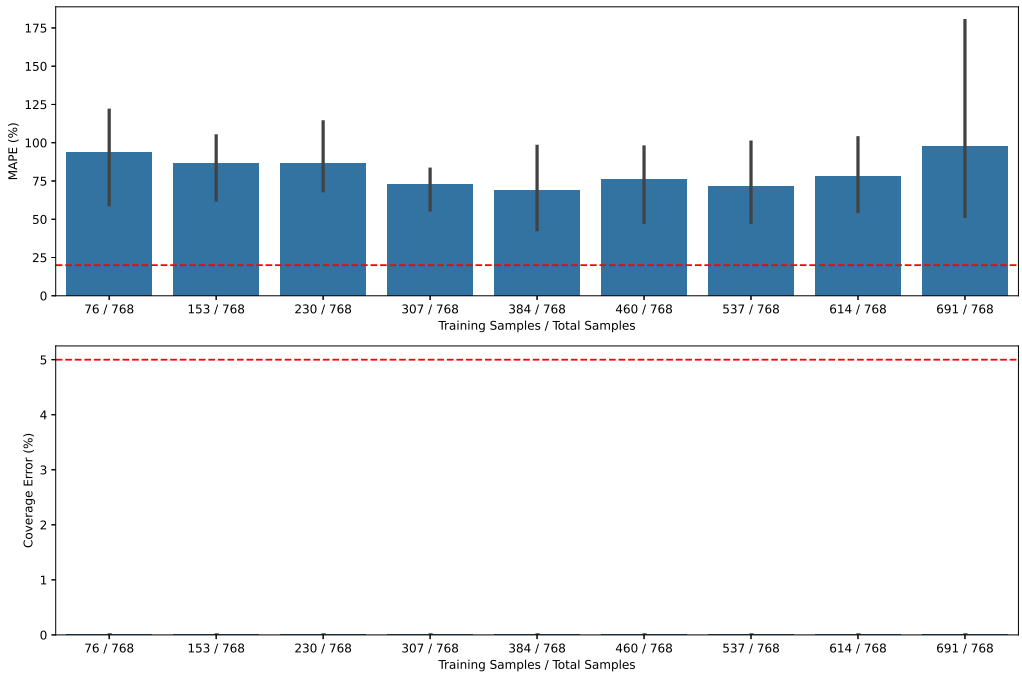


Figure 21: Training set size results for GEMmaker / fastq_merge runtime.

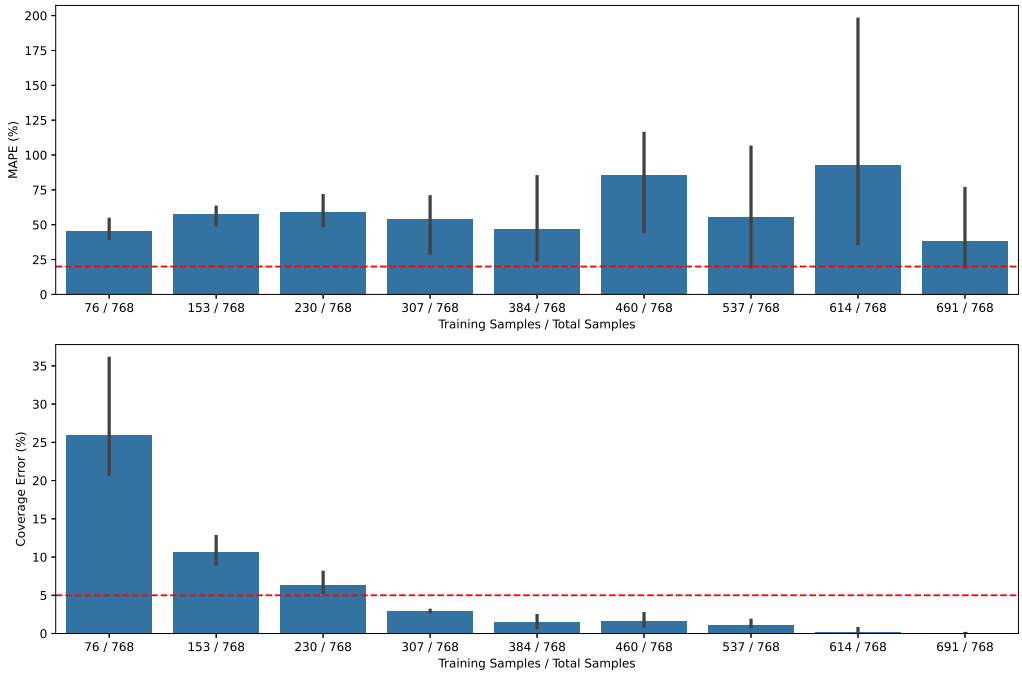


Figure 22: Training set size results for GEMmaker / fastq_merge disk usage.

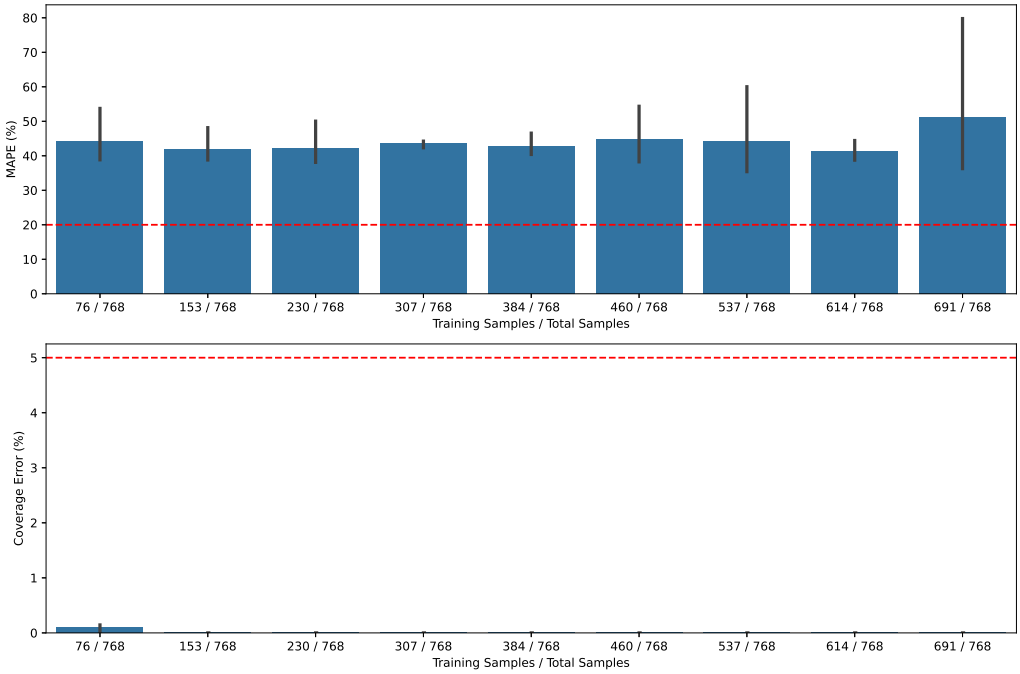


Figure 23: Training set size results for GEMmaker / fastqc_1 runtime.

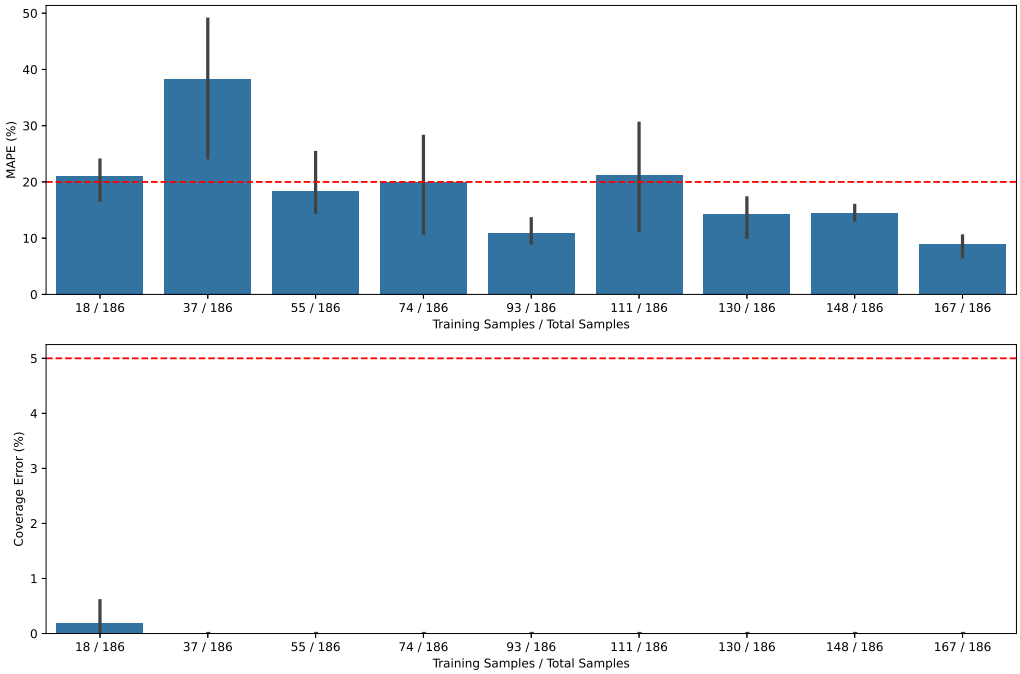


Figure 24: Training set size results for Gene Oracle / phase1_fg runtime.

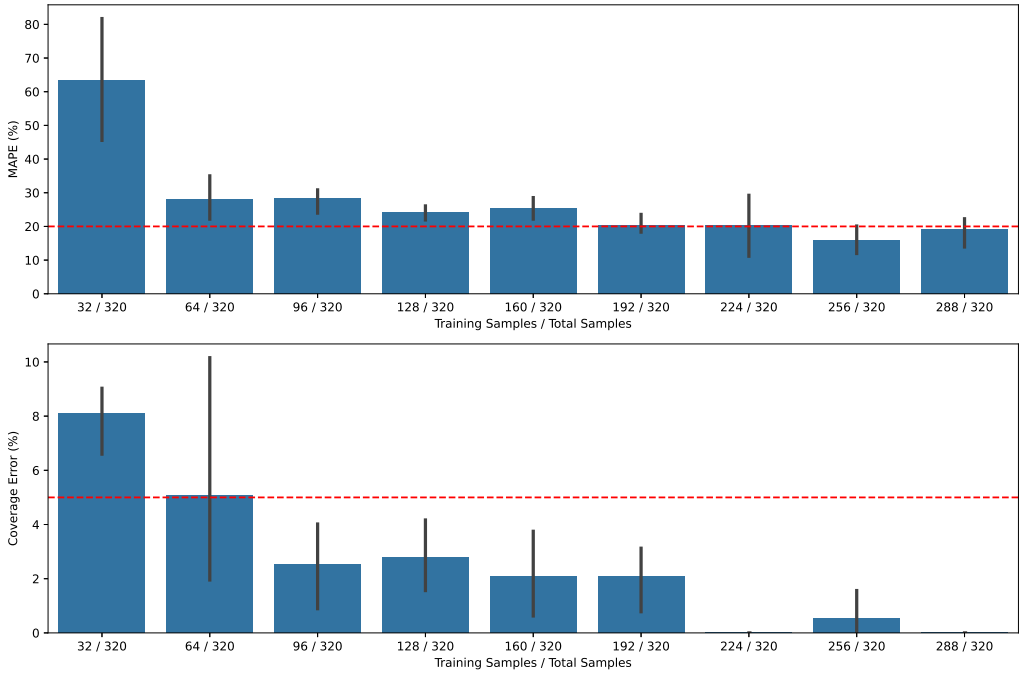


Figure 25: Training set size results for HemeLB runtime.

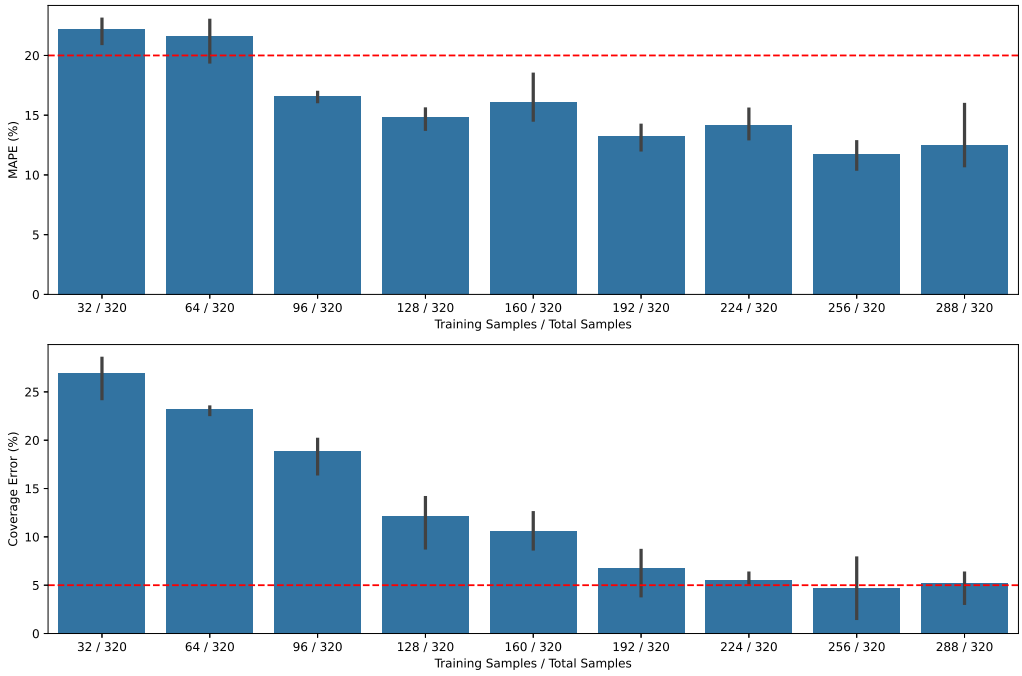


Figure 26: Training set size results for HemeLB memory usage.

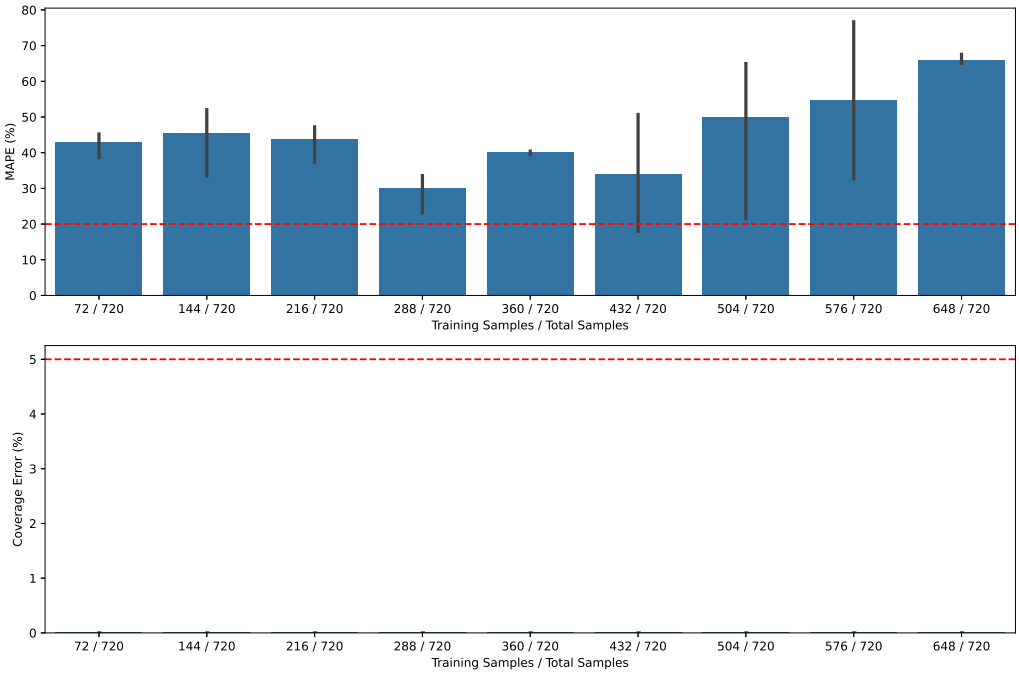


Figure 27: Training set size results for KINC / similarity_chunk memory usage.

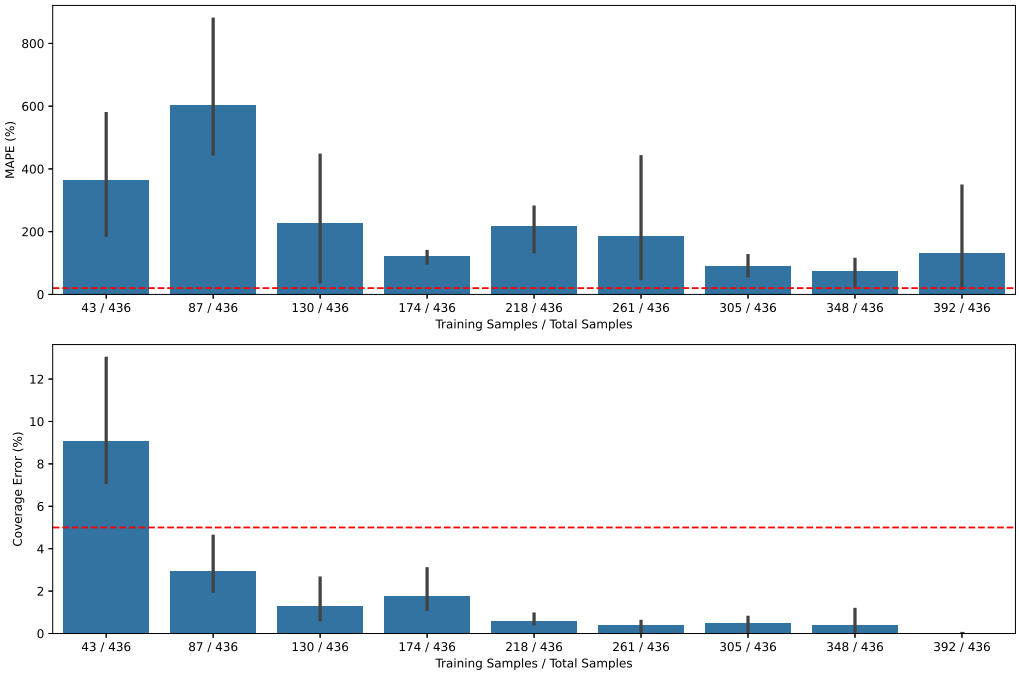


Figure 28: Training set size results for KINC / similarity_mpi runtime.

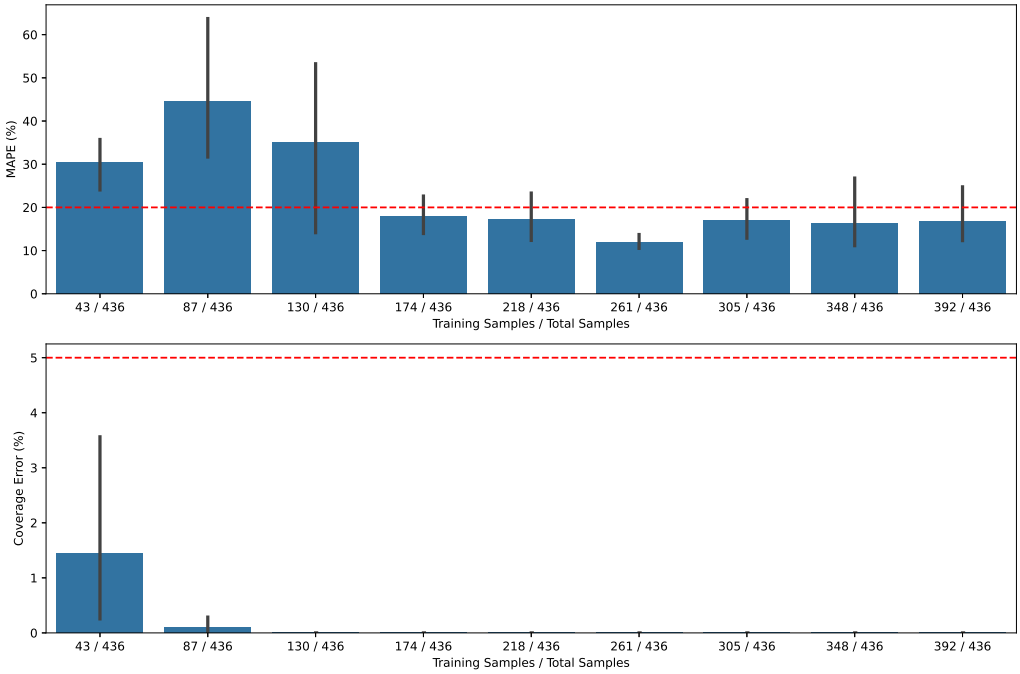


Figure 29: Training set size results for KINC / similarity_mpi memory usage.

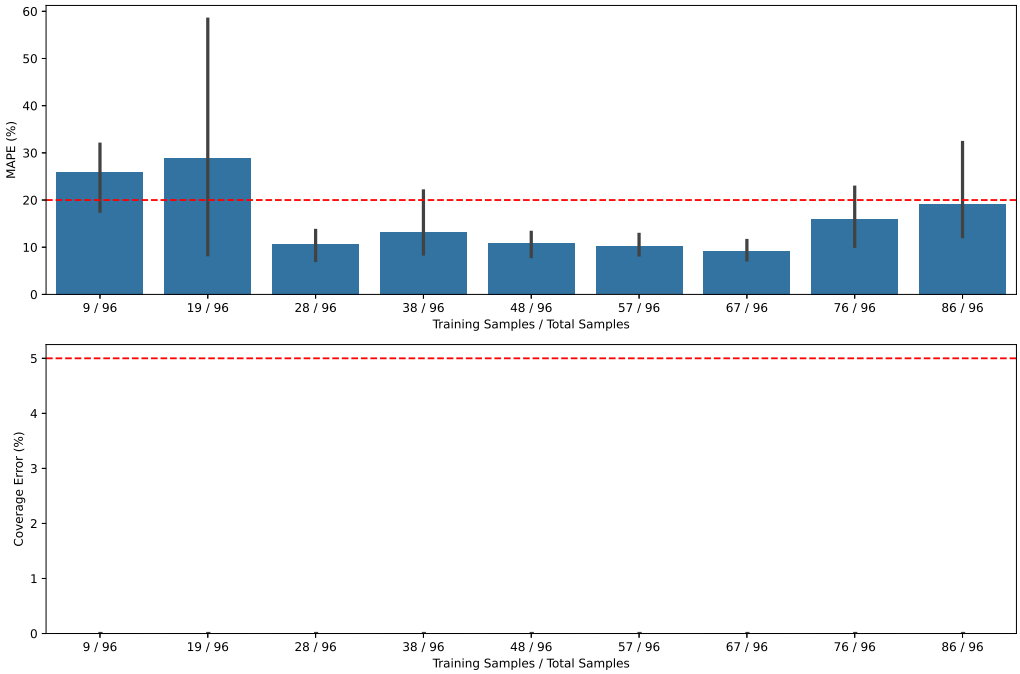


Figure 30: Training set size results for KINC / extract runtime.

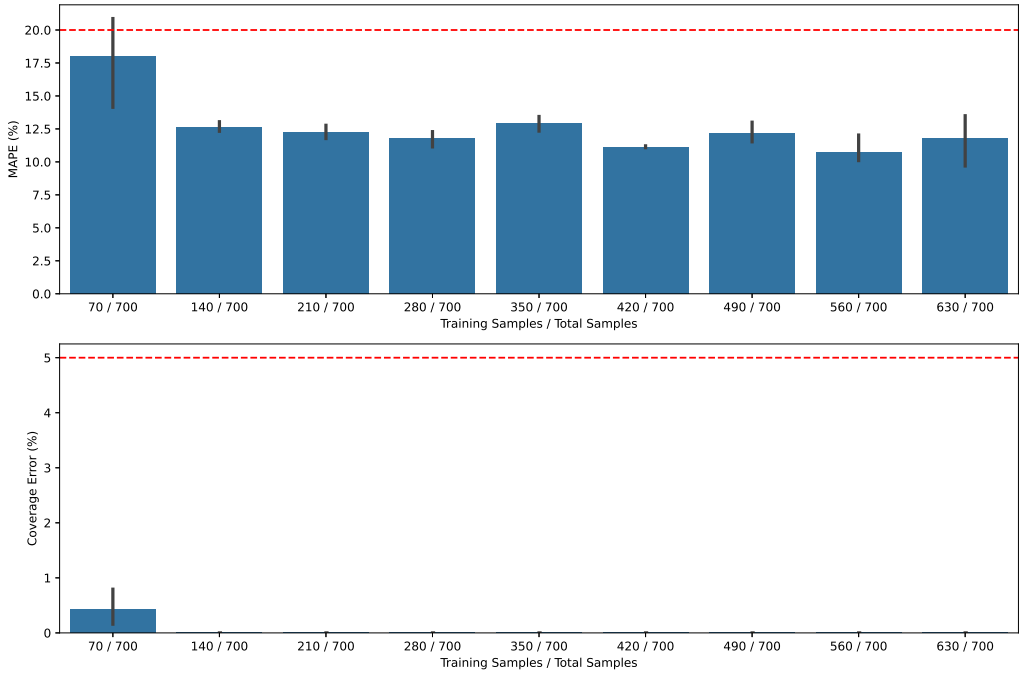


Figure 31: Training set size results for TSPG / train_target memory usage.

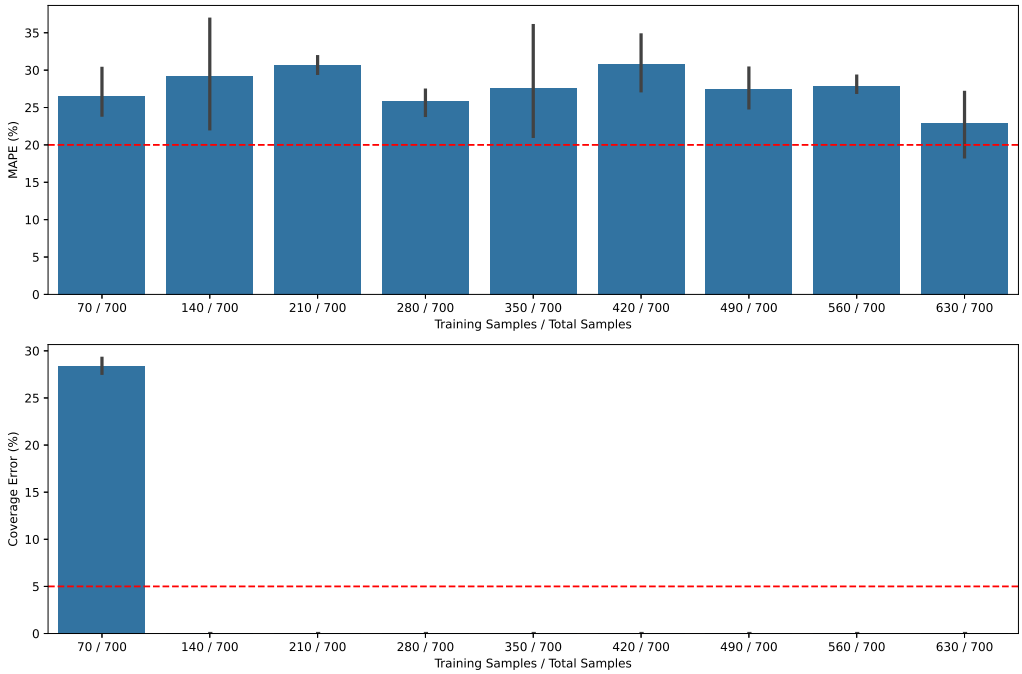


Figure 32: Training set size results for TSPG / perturb memory usage.

Appendix C: Minibench

This section contains the benchmark results for Minibench on each of the computing platforms that were used in this study. Refer to Table 3.1 for a description of each benchmark metric. Each measurement is the mean of three independent trials, with vertical bars denoting standard error.

Palmetto. Each node type corresponds to a phase. Each phase in the Palmetto cluster is a homogeneous collection of nodes.

Nautilus. The Nautilus cluster is extremely heterogeneous, with a wide variety of CPU and GPU models. As a result, each node type corresponds to a GPU model, and contains all of the nodes with that GPU model. Additionally, there is a single "CPU" node type for all nodes that do not have any GPUs.

Google Cloud Platform. While there are many VM types on GCP, only a subset of node types were measured in order to minimize cloud costs. The Life Sciences API uses the N1 machine type to provision VMs with custom CPU and memory requirements, and P100 and V100 GPUs are provisioned through a custom N1 instance. Therefore, we defined three node types, corresponding to a custom N1 instance with no GPU, a P100 GPU, or a V100 GPU.

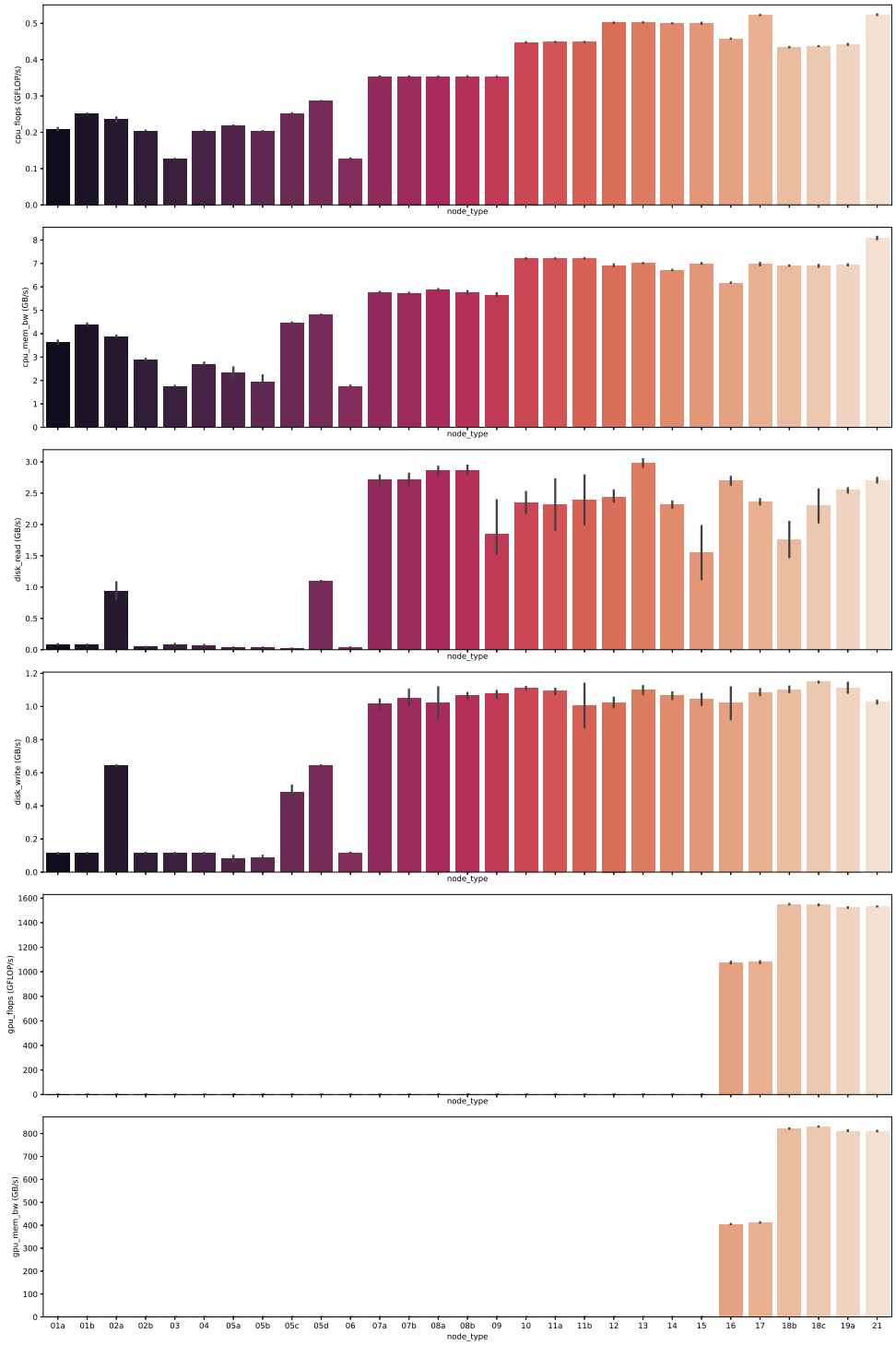


Figure 33: Minibench results for the Palmetto cluster.

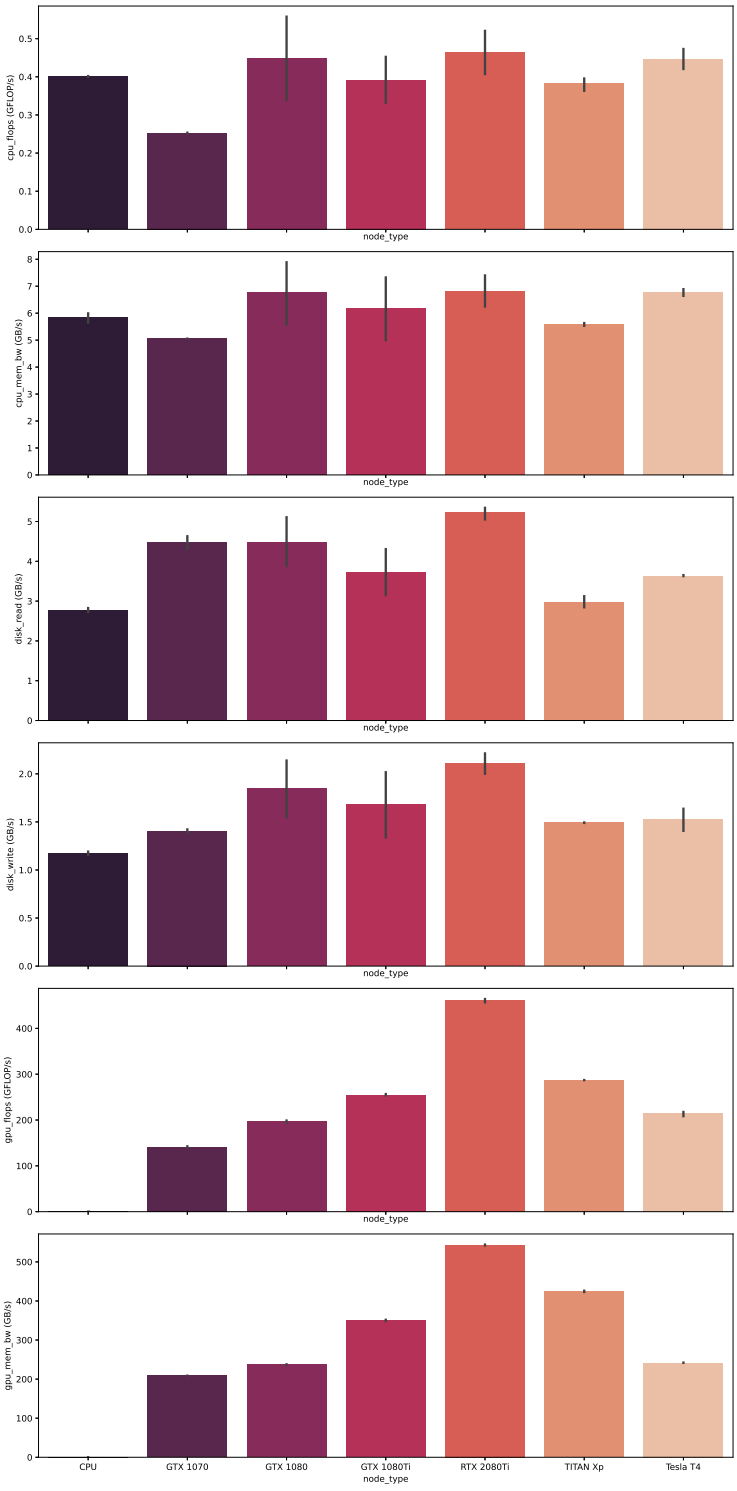


Figure 34: Minibench results for the Nautilus cluster.

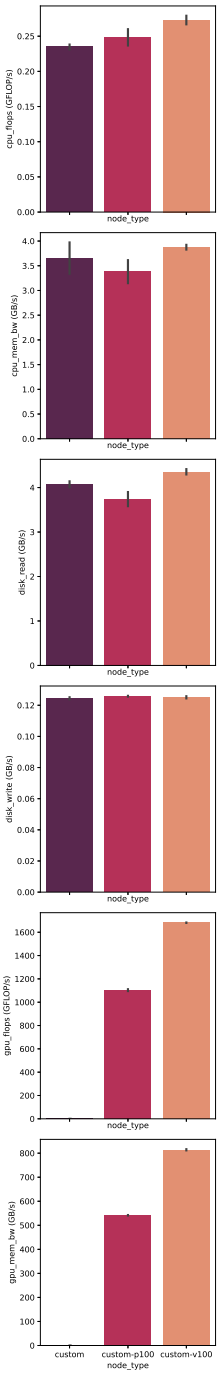


Figure 35: Minibench results for Google Cloud Platform.

Appendix D: GCP Cost Model

This appendix describes the cost model that was developed to compute the hourly cost of a VM on Google Cloud Platform (GCP). The hourly cost is combined with the predicted runtime to determine the predicted cost of a task. This model is formalized in the following equation,

$$C(t) = (r_{cpu}(t) + r_{mem}(t) + r_{disk}(t) + r_{gpu}(t)) \times T(t),$$

where $\{r_{cpu},\ r_{mem},\ r_{disk},\ r_{gpu}\}$ are the hourly rates of each component resource. These rates can be obtained from the GCP pricing documentation [11]. Table 3 lists the unit prices for the particular instance type that was used in the cost prediction experiment: a preemptible, custom N1 instance with some amount of standard disk storage and a P100 or V100 GPU. For example, a preemptible VM with 2 CPUs, 8 GB memory, 20 GB boot disk, 500 GB scratch disk, and 1 V100 GPU would cost $0.79 / hour based on current rates.

The above model only quantifies the computational cost of a task, and does not consider the cost of object storage and egress. While we do not investigate these costs in this dissertation, here we briefly describe how to model such costs. Object storage is used by Nextflow to store workflow inputs and outputs. The cost of object storage is denoted as $r_{storage}$ in Table 3. This cost depends on whether the workflow data is kept long-term or immediately transferred or removed from the cloud. In the former case, there is a monthly recurring cost; in the latter case, there is a relatively small cost for storing the workflow data during workflow execution. For example, 1 TB of object storage costs $20 / month, or $0.027 / hour. Network egress is the use of networking resources to transfer data from the cloud to an external location. Nextflow incurs egress costs only if it is configured to download workflow outputs to an external location, such as the user's local machine. The cost of egress is denoted as r_{egress} in Table 3. Network egress becomes cheaper at higher amounts. For

Name	Price	Unit
r_{cpu}	0.00698	$ / vCPU / hour
r_{mem}	0.00094	$ / GB / hour
r_{disk}	0.04	$ / GB / month
$r_{gpu,p100}$	0.43	$ / GPU / hour
$r_{gpu,v100}$	0.74	$ / GPU / hour
$r_{storage}$	0.02	$ / GB / month
$r_{egress,0}$	0.12	$ / GB
$r_{egress,1}$	0.11	$ / GB
$r_{egress,10}$	0.08	$ / GB

Table 3: Unit prices of selected GCP resources, Iowa (us-central1), November 2021

example, downloading 1 TB of data costs $120, another 1 TB in the same month costs $110, and so on. In other words, downloading 1 TB of data costs the same as storing 1 TB of data for 6 months. To summarize, the cost of storage and egress depends on how much output data is kept in the cloud, how long it is kept, and how much data, if any, is downloaded.

We provide concrete examples of compute, storage, and egress costs in order to highlight which components are likely to dominate. In a typical Nextflow run, input data is uploaded, each task is executed as a VM, the output data is downloaded, and all intermediate data is deleted. If the workflow produces a large amount of data, which is the case for most of the workflows used in this dissertation, the total cost will be dominated by network egress. These examples alone suggest that workflow outputs should be retained in the cloud, and downloaded only after they have been sufficiently reduced by downstream analyses (visualizations, statistical summaries, etc). In other words, the cost of retaining the output data, performing downstream workflows, and downloading the reduced output data, should be compared to the cost of immediately downloading the full output data, in order to determine the optimal "withdrawal" point. This strategy can be extended for an arbitrary sequence of workflows, and the cost of each component can be computed using the cost models described in this appendix.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Landing AI. Redefining quality control with ai-powered visual inspection for manufacturing, 2021.

[3] Maryam Amiri and Leyli Mohammad-Khanli. Survey on prediction models of applications for resources provisioning in cloud. *Journal of Network and Computer Applications*, 82:93–113, 2017.

[4] Aneurisk-Team. AneuriskWeb project website, http://ecm2.mathcs.emory.edu/aneuriskweb. Web Site, 2012.

[5] J.J.R. Burns, B.T. Shealy, M. Greer, J. Hadish, M.T. McGowen, T. Biggs, M.C. Smith, F.A. Feltus, and S.P. Ficklin. Addressing noise in co-expression network construction. *Briefings in Bioinformatics*, 2020.

[6] Laura Carrington, Allan Snavely, and Nicole Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3):336–346, 2006.

[7] Laura C Carrington, Michael Laurenzano, Allan Snavely, Roy L Campbell, and Larry P Davis. How well can simple metrics represent the performance of hpc applications? In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 48. IEEE Computer Society, 2005.

[8] Siddhartha Chatterjee, Erin Parker, Philip J Hanlon, and Alvin R Lebeck. Exact analysis of the cache behavior of nested loops. *ACM SIGPLAN Notices*, 36(5):286–297, 2001.

[9] Su-Hui Chiang, Andrea Arpaci-Dusseau, and Mary K Vernon. The impact of more accurate requested runtimes on production job scheduling performance. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 103–127. Springer, 2002.

[10] François Chollet et al. Keras. `https://keras.io`, 2015.

[11] Google Cloud. Compute engine pricing. `https://cloud.google.com/compute/all-pricing`, 2021.

[12] Rafael Ferreira Da Silva, Gideon Juve, Mats Rynge, Ewa Deelman, and Miron Livny. Online task resource consumption prediction for scientific workflows. *Parallel Processing Letters*, 25(03):1541003, 2015.

[13] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.

[14] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319, 2017.

[15] Michael D Ekstrand, John T Riedl, and Joseph A Konstan. *Collaborative filtering recommender systems*. Now Publishers Inc, 2011.

[16] Yuping Fan, Paul Rich, William E Allcock, Michael E Papka, and Zhiling Lan. Trade-off between prediction accuracy and underestimation rate in job runtime estimates. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 530–540. IEEE, 2017.

[17] Ian T Foster, Brian Toonen, and Patrick H Worley. Performance of massively parallel computers for spectral atmospheric models. *Journal of Atmospheric and Oceanic Technology*, 13(5):1031–1045, 1996.

[18] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016.

[19] Cristian Galleguillos, Alina Sîrbu, Zeynep Kiziltan, Ozalp Babaoglu, Andrea Borghesi, and Thomas Bridi. Data-driven job dispatching in hpc systems. In *International Workshop on Machine Learning, Optimization, and Big Data*, pages 449–461. Springer, 2017.

[20] Eric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. Improving backfilling by using machine learning to predict running times. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2015.

[21] Timnit Gebru and Andrej Karpathy. Cs231n. `https://cs231n.github.io/`, 2020.

[22] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):703–746, 1999.

[23] Jeremy Goecks, Anton Nekrutenko, James Taylor, Galaxy Team, et al. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):R86, 2010.

[24] Derek Groen, David Abou Chacra, Rupert W. Nash, Jirí Jaros, Miguel O. Bernabeu, and Peter V. Coveney. Weighted decomposition in high-performance lattice-boltzmann simulations: are some lattice sites more equal than others?, 2014.

[25] Derek Groen, James Hetherington, Hywel B Carver, Rupert W Nash, Miguel O Bernabeu, and Peter V Coveney. Analysing and modelling the performance of the hemelb lattice-boltzmann simulation environment. *Journal of Computational Science*, 4(5):412–422, 2013.

[26] Philip Guo. Cde: A tool for creating portable experimental software packages. *Computing in Science & Engineering*, 14(4):32–35, 2012.

[27] John Hadish, Tyler Biggs, Ben Shealy, Connor Wytko, Sai Prudhvi Oruganti, F. Alex Feltus, and Stephen Ficklin. Systemsgenetics/gemmaker: Release v1.1, January 2020.

[28] Muhammad Hafizhuddin Hilman, Maria Alejandra Rodriguez, and Rajkumar Buyya. Task runtime prediction in scientific workflows using an online incremental learning approach. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pages 93–102. IEEE, 2018.

[29] J. Howard and S. Gugger. *Deep Learning for Coders with Fastai and Pytorch: AI Applications Without a PhD*. O'Reilly Media, Incorporated, 2020.

[30] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[31] Engin Ipek, Bronis R De Supinski, Martin Schulz, and Sally A McKee. An approach to performance prediction for parallel applications. In *European Conference on Parallel Processing*, pages 196–205. Springer, 2005.

[32] Venkata Jagannath. Random forest, 2020.

[33] Tejas S Karkhanis and James E Smith. A first-order superscalar processor model. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 338–349. IEEE, 2004.

[34] Darren J Kerbyson, Henry J Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J Wasserman, and Mike Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 37–37, 2001.

[35] Jing Lei, Max G'Sell, Alessandro Rinaldo, Ryan J Tibshirani, and Larry Wasserman. Distribution-free predictive inference for regression. *Journal of the American Statistical Association*, 113(523):1094–1111, 2018.

[36] Jiangtian Li, Xiaosong Ma, Karan Singh, Martin Schulz, Bronis R de Supinski, and Sally A McKee. Machine learning based online performance prediction for runtime parallelization and task scheduling. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–100. IEEE, 2009.

[37] Anirban Mandal, Paul Ruth, Ilya Baldin, Dariusz Król, Gideon Juve, Rajiv Mayani, Rafael Ferreira Da Silva, Ewa Deelman, Jeremy Meredith, Jeffrey Vetter, et al. Toward an end-to-end framework for modeling, monitoring and anomaly detection for scientific workflows. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1370–1379. IEEE, 2016.

[38] Aniruddha Marathe, Rushil Anirudh, Nikhil Jain, Abhinav Bhatele, Jayaraman Thiagarajan, Bhavya Kailkhura, Jae-Seung Yeom, Barry Rountree, and Todd Gamblin. Performance modeling under resource constraints using deep transfer learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[39] Andréa Matsunaga and José AB Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE Computer Society, 2010.

[40] Tom M Mitchell et al. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877, 1997.

[41] Tudor Miu and Paolo Missier. Predicting the execution time of workflow activities based on their input features. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 64–72. IEEE, 2012.

[42] David A Monge, Matěj Holec, Filip Železnỳ, and Carlos García Garino. Ensemble learning of runtime prediction models for gene-expression analysis workflows. *Cluster Computing*, 18(4):1317–1329, 2015.

[43] Farrukh Nadeem, Daniyal Alghazzawi, Abdulfattah Mashat, Khalid Faqeeh, and Abdullah Almalaise. Using machine learning ensemble methods to predict execution time of e-science workflows in heterogeneous distributed systems. *IEEE Access*, 7:25138–25149, 2019.

[44] Farrukh Nadeem and Thomas Fahringer. Predicting the execution time of grid workflow applications through local learning. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12. IEEE, 2009.

[45] NCBI. Sra overview. `https://trace.ncbi.nlm.nih.gov/Traces/sra/`, 2019.

[46] Ryosuke Okuda, Yuki Kajiwara, and Kazuaki Terashima. A survey of technical trend of adas and autonomous driving. In *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*, pages 1–4. IEEE, 2014.

[47] Vivek K Pallipuram, Melissa C Smith, Nimisha Raut, and Xiaoyu Ren. A regression-based performance prediction framework for synchronous iterative algorithms on general purpose graphical processing unit clusters. *Concurrency and Computation: Practice and Experience*, 26(2):532–560, 2014.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[49] Thanh Phuong Pham, Juan J Durillo, and Thomas Fahringer. Predicting workflow task execution time in the cloud using a two-stage machine learning approach. *IEEE Transactions on Cloud Computing*, 2017.

[50] Ilia Pietri, Gideon Juve, Ewa Deelman, and Rizos Sakellariou. A performance model to estimate execution time of scientific workflows on the cloud. In *2014 9th Workshop on Workflows in Support of Large-Scale Science*, pages 11–19. IEEE, 2014.

[51] Kivan Polimis, Ariel Rokem, and Bryna Hazelton. Confidence intervals for random forests in python. *Journal of Open Source Software*, 2(1), 2017.

[52] Eduardo R Rodrigues, Renato LF Cunha, Marco AS Netto, and Michael Spriggs. Helping hpc users specify job memory requirements via machine learning. In *2016 Third International Workshop on HPC User Support Tools (HUST)*, pages 6–13. IEEE, 2016.

[53] Michel JF Rosa, Aletéia PF Araüjo, and Felipe LS Mendes. Cost and time prediction for efficient execution of bioinformatics workflows in federated cloud. In *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 1703–1710. IEEE, 2018.

[54] Ben Shealy, Coleman McKnight, and Fan Jiang. Nextflow-api. `https://github.com/SciDAS/nextflow-api`, 2020.

[55] Benjamin T Shealy, Josh JR Burns, Melissa C Smith, F Alex Feltus, and Stephen P Ficklin. Gpu implementation of pairwise gaussian mixture models for multi-modal gene co-expression networks. *IEEE Access*, 7:160845–160857, 2019.

[56] Benjamin T Shealy, Mehrdad Yousefi, Ashwin T Srinath, Melissa C Smith, and Ulf D Schiller. Gpu acceleration of the hemelb code for lattice boltzmann simulations in sparse complex geometries. *IEEE Access*, 9:61224–61236, 2021.

[57] Larry Smarr, Camille Crittenden, Thomas DeFanti, John Graham, Dmitry Mishin, Richard Moore, Philip Papadopoulos, and Frank Würthwein. The pacific research platform: Making high-speed networking a reality for the scientist. In *Proceedings of the Practice and Experience on Advanced Research Computing*, pages 1–8, 2018.

[58] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 21–21. IEEE, 2002.

[59] Allan Snavely, Nicole Wolter, and Laura Carrington. Modeling application performance by convolving machine signatures with application profiles. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*, pages 149–156. IEEE, 2001.

[60] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[61] Mohammed Tanash, Brandon Dunn, Daniel Andresen, William Hsu, Huichen Yang, and Adedolapo Okanlawon. Improving hpc system performance by predicting job resources via supervised machine learning. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, page 69. ACM, 2019.

[62] Colin Targonski, M Reed Bender, Benjamin T Shealy, Benafsh Husain, Bill Paseman, Melissa C Smith, and F Alex Feltus. Cellular state transformations using deep learning for precision medicine applications. *Patterns*, page 100087, 2020.

[63] Colin A Targonski, Courtney A Shearer, Benjamin T Shealy, Melissa C Smith, and F Alex Feltus. Uncovering biomarker genes with enriched classification potential from hallmark gene sets. *Scientific reports*, 9(1):1–10, 2019.

[64] James H Thrall, Xiang Li, Quanzheng Li, Cinthia Cruz, Synho Do, Keith Dreyer, and James Brink. Artificial intelligence and machine learning in radiology: opportunities, challenges, pitfalls, and criteria for success. *Journal of the American College of Radiology*, 15(3):504–508, 2018.

[65] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J Leung, Manuel Egele, and Ayse K Coskun. Diagnosing performance variations in hpc applications using machine learning. In *International Supercomputing Conference*, pages 355–373. Springer, 2017.

[66] Stefan Wager, Trevor Hastie, and Bradley Efron. Confidence intervals for random forests: The jackknife and the infinitesimal jackknife. *The Journal of Machine Learning Research*, 15(1):1625–1651, 2014.

[67] Qingguo Wang, Joshua Armenia, Chao Zhang, Alexander V Penson, Ed Reznik, Liguo Zhang, Thais Minet, Angelica Ochoa, Benjamin E Gross, Christine A Iacobuzio-Donahue, et al. Unifying cancer and normal rna sequencing data from different sources. *Scientific data*, 5(1):1–8, 2018.

[68] Michael Waskom and the seaborn development team. mwaskom/seaborn, September 2020.

[69] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. Probabilistic guarantees of execution duration for amazon spot instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 18. ACM, 2017.

[70] Michael R Wyatt, Stephen Herbein, Todd Gamblin, Adam Moody, Dong H Ahn, and Michela Taufer. Prionn: Predicting runtime and io using neural networks. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–12, 2018.

[71] Leo T Yang, Xiaosong Ma, and Frank Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 40–40. IEEE, 2005.