Predictive Analytics Using Statistical, Learning, and Ensemble Methods to Support Real-Time Exploration of Discrete Event Simulations

Walid Budgaga[a], Matthew Malensek[a], Sangmi Pallickara[a], Neil Harvey[b], F. Jay Breidt[c], Shrideep Pallickara[a]

[a] *Department of Computer Science*
Colorado State University
Fort Collins, Colorado, USA

[b] *School of Computer Science*
University of Guelph
Guelph, Ontario, Canada

[c] *Department of Statistics*
Colorado State University
Fort Collins, Colorado, USA

Abstract

Discrete event simulations (DES) provide a powerful means for modeling complex systems and analyzing their behavior. DES capture all possible interactions between the entities they manage, which makes them highly *expressive* but also compute-intensive. These computational requirements often impose limitations on the breadth and/or depth of research that can be conducted with a discrete event simulation.

This work describes our approach for leveraging the vast quantity of computing and storage resources available in both private organizations and public clouds to enable real-time exploration of discrete event simulations. Rather than directly targeting simulation execution speeds, we autonomously generate and execute novel *scenario variants* to explore a representative subset of the simulation parameter space. The corresponding outputs from this process are analyzed and used by our framework to produce models that accurately forecast simulation outcomes in real time, providing interactive feedback and facilitating exploratory research.

Our framework distributes the workloads associated with generating and executing scenario variants across a range of commodity hardware, including public and private cloud resources. Once the models have been created, we evaluate

Email addresses: budgaga@cs.colostate.edu (Walid Budgaga), malensek@cs.colostate.edu (Matthew Malensek), sangmi@cs.colostate.edu (Sangmi Pallickara), neilharvey@gmail.com (Neil Harvey), jbreidt@stat.colostate.edu (F. Jay Breidt), shrideep@cs.colostate.edu (Shrideep Pallickara)

their performance and improve prediction accuracy by employing dimensionality reduction techniques and ensemble methods. To make these models highly accessible, we provide a user-friendly interface that allows modelers and epidemiologists to modify simulation parameters and see projected outcomes in real time.

Keywords: Discrete Event Simulation, Latin Hypercube Sampling, Distributed Execution, Cloud Infrastructure

---

1. Introduction

The behavior of complex, real-world systems is often difficult to predict or fully understand. These systems may be influenced by any number of internal or external stimuli, and direct experimentation is often prohibitively expensive, time-consuming, or simply not feasible. In these situations, computer simulation is a compelling solution. Specifically, discrete event simulations (DES) model all possible interactions between entities in a system, making them highly *expressive*. To model uncertainty in these interactions, *stochastic* discrete event simulations associate probabilities with each of their events. However, this expressiveness comes at the cost of increased computational complexity and prolonged execution times.

Our subject discrete event simulation, the North American Animal Disease Spread Model (NAADSM) [1] is an epidemiological model of disease outbreaks in livestock populations. Livestock are simulated as individual *herds* and interact with their environment through events; for instance, an *exposure* event may occur when a particular herd has come into contact with a disease of interest. The simulation has been applied in studies of several different diseases, including foot-and-mouth disease [2], avian influenza [3], and pseudorabies [4]. NAADSM is a stochastic DES: simulations are run many times, with each *iteration* contributing to an overall representation of the output variables' *probability distributions*. Iterations often require several hours of CPU time to execute depending on how events unfold.

The computational complexity of these stochastic iterations makes it difficult for planners and epidemiologists to perform exploratory "what if" analysis that plays an important role in planning and preparedness. For instance, a planner may make subtle adjustments to quarantine procedures or the number of vaccines available in order to analyze economic consequences or how disease spread might change. Each modification of the input parameters requires a new set of iterations to be run. Dividing the target simulation into several units and executing them in parallel is one way to improve overall execution times [5, 6], but generally does not enable real-time exploration. In this work, we target real-time computational guarantees that involve providing sub-second, interactive responses to the user as simulation parameters are changed.

This paper describes our approach for retaining the expressiveness of stochastic DES while addressing the weaknesses in the timeliness of their outcomes. We

achieve this by utilizing voluminous epidemic simulation data to glean insights and derive relationships between scenarios and outcomes. We then use this information to create models that can forecast the results for an entire class of input parameters, enabling our system to provide real-time answers to exploratory investigations.

1.1. Research Challenges

We consider the problem of generating fast, accurate DES forecasts for a given subset of the input parameter space. These forecasts are generated in lieu of compute-intensive simulation runs. Challenges involved in accomplishing this include:

1. *Data Dimensionality*: Each input parameter represents a dimension, the number of which can be quite high (approximately 1800-2500 in this particular study). Furthermore, input parameters come in a variety of types: integers, floats, or even probability distributions.
2. *Interactive Exploration*: The "what if" scenarios in question must provide immediate feedback during exploration; every parameter change will result in slightly different outputs that must be forecast in real time.
3. *Accuracy*: Outputs produced during exploration must be reasonably accurate to ensure their usefulness. Once a planner has determined parameters of interest, he or she may decide to perform a set of actual simulation runs.

1.2. Research Questions

Specific research questions we explore include:

1. How can we minimize the number of iterations required to build our models while still ensuring statistical coverage of the parameter space?
2. What are the implications of our execution model, and how can we obtain necessary processing resources?
3. A large amount of training data is necessary for making predictions. How can this data be managed in a scalable and fault-tolerant manner?
4. How can we deal with increases in dimensionality as the number of input parameters grows?
5. What prediction models can provide both **accurate** and **real-time** results?
6. How can we improve model performance? What impact does the relative error, feature correlations, and input dataset size have on predictive performance?
7. Once the models are built, how can we make their insights available to users in an accessible and efficient manner?

1.3. Summary of Approach

Our approach treats the DES in question as a black box and focuses on deriving relationships between the inputs and outputs. Given a disease spread scenario, our framework views input tuples as points in the multidimensional parameter space. We first derive bounds for each of the dimensions from both historical data and subject-matter experts, and then sample within this parameter space to create novel *scenario variants*. Our objective is two-fold: we wish to ensure adequate coverage of the parameter space, while also controlling the size of computational workloads.

For each scenario, we inspect the variances of key output variables to derive the number of iterations that must be executed. Both the variant generation and their subsequent simulation iterations are implemented as MapReduce [7] jobs that are orchestrated by our *Forager* framework. Forager deals with highly elastic resource pools and can scavenge for CPU cycles on both physical and virtual machines, including spot instances in the cloud. These simulation runs generate a large amount of data, often producing terabytes of outputs in a few hours. To cope with these storage demands, we use a distributed storage system to manage the data in a scalable and fault-tolerant manner.

Once the simulation iterations have been executed, we model the relationships between inputs and outputs. To facilitate predictions, we create a model for each output variable. We consider both linear (multivariate linear regression) and non-linear (artificial neural networks) methods to construct these models, and use k-fold cross-validation to assess their generalizability. To further improve predictive performance, we investigate the use of ensemble methods to reduce model bias (gradient boosting) and variance (random forests). We also consider the effects of dimensionality and collinearity in the input dataset to reduce model noise and creation times.

The technologies discussed in this study enable our system to provide accurate answers to "what if" scenarios in real time. We make this information accessible to planners and epidemiologists through a web-based user interface that targets a broad range of devices and platforms. This allows interactive modification of scenario parameters with direct feedback.

1.4. Paper Contributions

This paper describes our approach for supporting interactive exploration of discrete event simulations. The research involves several key features, including the use of analytics to ensure accurate and timely forecasts that account for statistical coverage of the parameter space, orchestration of workloads, generation and management of training data, correlations between inputs and outputs, dimensionality reduction, and the use of learning structures. Our specific contributions include:

- *Applicability*: The framework is broadly applicable to other compute-intensive simulations. We treat a given simulation as a black box and focus on deriving the relationship between inputs and outputs.

- *Dimensionality*: Our approach copes with high dimensionality and diversity of dimension types.
- *Data Management*: Scenario variants and their outputs comprise a voluminous dataset. We use a distributed key-value store and MapReduce computations to deal with these storage and processing demands.
- *Resource Management*: Processing and storage resources can be sourced from physical or virtual machines. Our framework accounts for running in a highly elastic environment and can scale both up and down to meet changing requirements.
- *Forecasting*: The proposed approach learns from the data to derive relationships between inputs and outputs. We have incorporated support for both linear and nonlinear models and exploration of the parameter space can be performed in real time.

1.5. Paper Extensions

Since the publication of *Using Distributed Analytics to Enable Real-Time Exploration of Discrete Event Simulations* [8], we have extended our manuscript to incorporate several new techniques for increasing prediction accuracy during the knowledge extraction process, as well as details on the design and functionality of our "What-If" interactive exploratory analytics tool. The prediction improvements discussed in this work come from multiple techniques: bias-variance analysis, ensemble methods including random forests and gradient boosting, and collinearity analysis. Sections 6 and 7 discuss the development of our predictive models and the design of our What-If tool, respectively. The addition of these two sections accounts for a 50% increase in overall manuscript content.

1.6. Paper Organization

The rest of the paper is organized as follows. Section 2 describes our scenario variant generation process. Section 3 focuses on our distributed execution platform, Forager. Section 4 outlines how we manage outputs and distributed state, followed by Section 5, which describes how we build our models and make predictions. Section 6 discusses improvements to increase predictive accuracy of the models, and Section 7 describes our "What-If" tool, its user interface, and system integration. Section 8 surveys related work, and Section 9 provides concluding material and our future research directions.

2. Generating Novel Scenario Variants

In our subject simulation, input parameters are used to describe disease properties and outbreak characteristics. These variables include factors such as the probability of infection transfer, maximum airborne distance of disease spread, and the overall area at risk for infection. The first piece of information

required to generate a new scenario is the *data type* for each input variable, which could include booleans, integers, floating point values, or even line charts and probability density functions (PDFs). Our framework can identify most data types automatically, but for more exotic parameters we provide an XML-based variable description language (shown in Figure 1).

While some input variables have predefined ranges of valid values (such as a percentage ranging from 0 to 100%), others are completely unconstrained. In both cases, a value may be **valid** but not **plausible**, i.e., extremely unlikely to occur due to environmental conditions or other factors. To produce plausible ranges for the input variables, our framework consults historical data available in previous scenarios as a preliminary step; for instance, Figure 2 contains a variety of probability density functions that were used previously for the "cattle latent period" input parameter (amount of time between an infection and the onset of infectiousness, in days). Next, the ranges determined by this process are inspected and refined by subject-matter experts if necessary. This helps reduce or potentially avoid user intervention while maintaining accuracy.

2.1. Complex Data Types: Charts and Probability Densities

While most input variables represent a numerical value or discrete state, simulations also frequently employ two-dimensional line charts or probability density functions to describe complex behavior. These data types play a vital role in simulation outcomes and must be varied to enable the exploration of a scenario's parameter space. However, a simple range of values does not capture the multidimensional relationships that these data types describe.

To generate a 2D line chart variant, we consider four variables that describe chart behavior: the span of x- and y-values, maximum x- and y-values, x-value at the maximum y-value, and the percent distance across the x-axis where the maximum y-value occurs. Next, we perturb the data points to create a new chart that exhibits behavior similar to the source chart, while still representing the

```
<param name="max-spread" type="range">
  <bounds>2.83106, 6.0</bounds>
</param>

<param name="latent" type="distribution">
  <bounds>0, 9.3631</bounds>
  <mean>1.98418, 4.1</mean>
  <variance>1.21, 4.20754</variance>
  <skewness>-0.235034, 1.16052</skewness>
</param>
```

Figure 1: A sample XML variable description file showing range and probability distribution parameters.

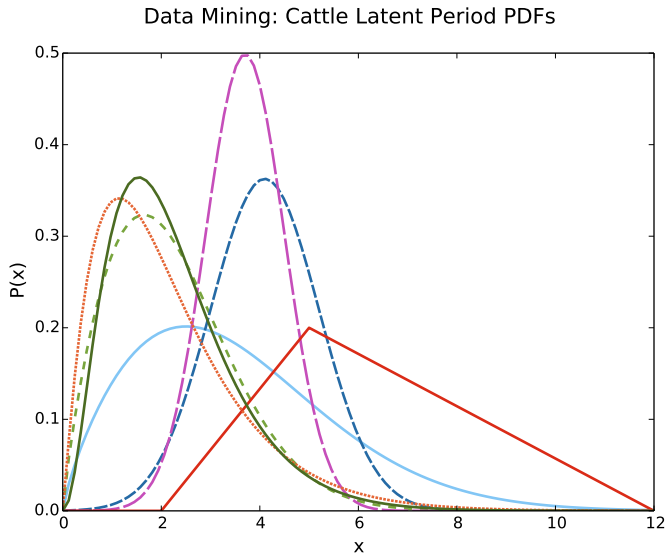

Figure 2: A variety of probability density functions (PDFs) that were used to describe the "cattle latent period" parameter in previous scenarios. Historical data provided by modelers and epidemiologists is used to determine upper and lower bounds of valid parameter ranges.

shift in values derived from historical trends and feedback from subject-matter experts.

Probability density functions can often be decomposed into a few distinguishing variables as well: mean, variance, and skewness can all be manipulated to create new PDF variations. Unfortunately, each type of distribution has its own formulas for modifying these attributes, and the fact that our simulation supports 22 different types of distributions only further complicates matters. Instead of dealing with this issue as 22 separate problems (or possibly more for other simulations), we generalize the PDFs by transforming them to *piecewise linear approximations*. Once this step has been completed, we inspect the resulting upper and lower bounds, mean, variance, and skewness. These attributes are modified to create a new linear approximation of a curve. Next, a *beta distribution* is mathematically fit to the curve. Beta distributions are described by two *shape parameters*, α and β, which can be adjusted to model a wide range of distributions: normal, continuous, skew normal, exponential, etc. An overview of our PDF generation algorithm is provided in Figure 3. We have verified that this approach works with the 22 PDFs supported by NAADSM, including Bernoulli, hypergeometric, binomial, and logistic distributions.

2.2. Latin Hypercube Sampling

After establishing plausible input ranges, one might elect to generate new scenario variants with random samples across the parameter space. However, simple random sampling assigns an equal probability to each possible input

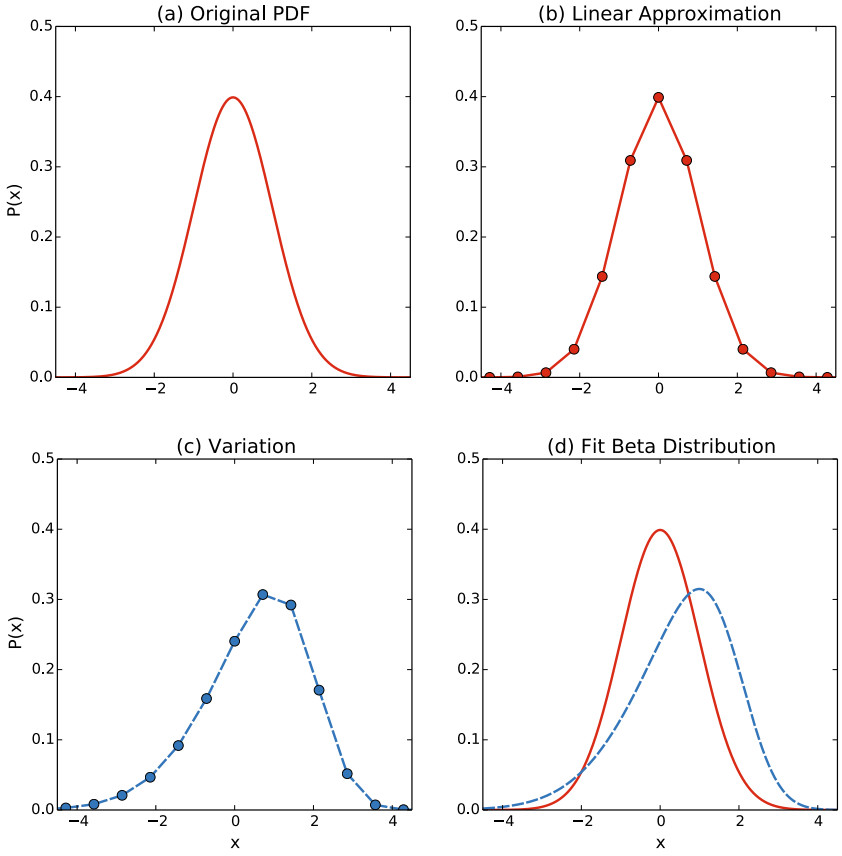

Figure 3: A visual overview of our probability density function (PDF) variation algorithm. The original PDF **(a)** is converted to a piecewise linear approximation **(b)** by inspecting y-coordinates at each LHS bin. Next, key attributes (upper and lower bounds, mean, variance, and skewness) are modified to create a linear approximation of a new PDF **(c)**. Finally, a beta distribution is fit to the modified linear approximation to create a new PDF variant **(d)**.

X	Y	Z
Z	X	Y
Y	Z	X

Figure 4: A 3×3 Latin square.

without considering the plausibility of the values. To circumvent this limitation, weighted sampling draws values from a probability distribution. This preserves the plausibility curve of potential inputs, but also increases the likelihood of choosing highly probable values; if a small number of samples are drawn from the probability distribution, a correspondingly small portion of the input space will be represented. This means that we would have to generate (and execute) a large number of scenario variants to adequately explore the parameter space with weighted sampling.

Unlike simple random sampling or weighted sampling methods, Latin Hypercube Sampling (LHS) [9] *stratifies* the input probability distributions to better represent their underlying variability. This reduces the number of samples required for our algorithms to adequately explore the scenario parameter space, which in turn decreases the overall computational footprint of the framework. LHS owes its name to *Latin squares*, which are $N \times N$ arrays that contain N different elements. Each element in a Latin square occurs exactly once in each row and column (see Figure 4). When this concept is applied in a multidimensional setting the elements occur once in each hyperplane, forming a *Latin hypercube*. This allows us to produce samples across all the variables in the parameter space in a single sampling step. Based on the number of samples required, each stratum represented by the elements in the hypercube is divided into equal intervals. These produce samples in the range $[0, 1]$, which are converted back to the original units using the information from our data mining process. A visual comparison of unweighted, weighted, and Latin Hypercube sampling is provided in Figure 5. Latin Hypercube Sampling provides the best overall representation of the underlying distribution (a standard normal distribution) when the number of samples is held constant across methods.

2.3. Measuring and Verifying Output Variance

Once a scenario variant has been created, it must be executed several times to obtain an understanding of its output distribution and behavior. To begin this process, 32 pilot runs are executed for each variant. After the pilot runs are complete, our framework determines whether the overall variation of the output variables is of *practical significance* or not. Doing so requires two pieces of information: (1) the outputs that are most meaningful from an analytical standpoint, and (2) the *minimal significant difference* in output variances that must be achieved. As with input range discovery, both of these items are obtained from mining historical data and subject-matter experts.

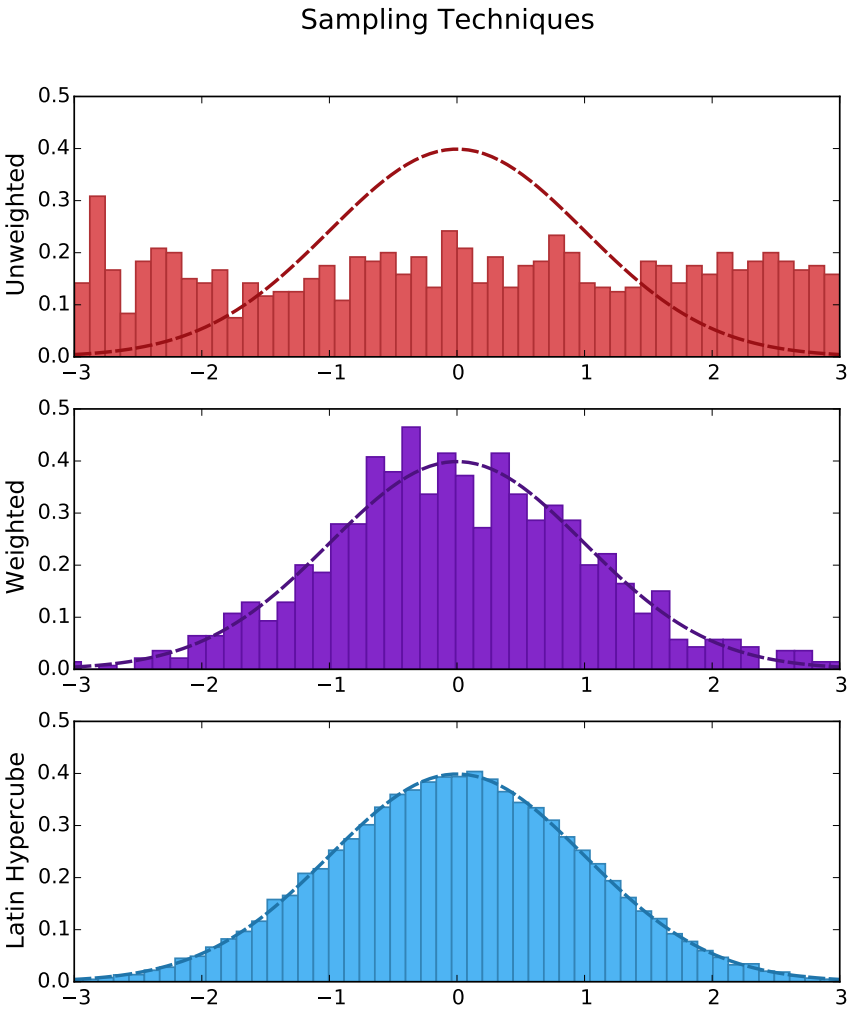

Figure 5: Sampling performed over a normal distribution using unweighted, weighted, and Latin Hypercube sampling. In each case 1,000 samples were taken and are represented by 50 bins.

To establish a set of variables that have a strong analytical significance, our framework inspects reports generated from past scenario executions to determine which variables were requested most frequently by end users. This process can be supervised by subject-matter experts or performed autonomously. Once the prediction engine has been initialized, our framework can also make recommendations on which output variables might warrant further analysis.

Determining the minimal significant difference in output variables requires the knowledge of a subject-matter expert. These values can be expressed as percentages, numeric ranges, or confidence intervals, and tell the framework whether or not there is enough variation in the output variables. They also determine how well the input parameter space has been explored based on the overall variance of the outputs. If the minimum variance is met, execution of the scenario variant is complete. Otherwise, the observed variance is used to calculate how many more executions of the scenario must be carried out to achieve the required minimum variance.

3. Distributed Simulation Orchestration: Forager

Our framework requires a large number of processing resources, with each scenario produced by the simulation variant generator representing several discrete units of computation. In our initial tests, 10,000 variants were generated and each was run for at least 32 iterations. After running additional simulations to achieve target output variances (as discussed in the previous section), the total number of iterations reached approximately 400,000. This makes our framework an ideal candidate for execution in an elastic cloud or clustered environment due to its computational footprint and the uncertainty in total iterations required. Compared to the problems addressed by distributed execution frameworks such as Hadoop [10], Dryad [11], or the myriad of other MapReduce [7] implementations, the tasks we execute and manage have several distinct features:

- Run times are uncertain, and vary due to the stochastic nature of the underlying simulation.
- The overall number of tasks is significantly larger than the number of available processing resources.
- Tasks can be completed out-of-order, and there are no "waves" of execution.
- The pool of processing resources is highly elastic, fluctuating constantly as availability changes over time.
- The framework must deal with other users and processes contending for resources.

Because of these processing requirements, we chose to design a new distributed execution engine, called *Forager*. Similar to animal behavior observed in nature, Forager must adapt to constantly changing external conditions to acquire resources. Forager is based on our *Granules* cloud runtime [12, 13], and provides new scheduling and orchestration functionality for our specific use case. Granules is backed by the NaradaBrokering project [14], a reliable content distribution infrastructure and messaging substrate. Each of these related projects is an open source effort.

3.1. Resource Acquisition

For a given scenario, our framework may produce hundreds of thousands of executable tasks. This creates a large demand for processing elements, which Forager acquires from a variety of sources: clusters, idle workstations, and both public and private clouds. Unlike volunteer computing [15] deployments, typical Forager installations are managed by a single entity in a trusted environment. This constraint helps us ensure that confidential or proprietary information being used by the simulation is not made publicly available.

A lightweight Forager *daemon* is run on each participating resource. Rather than being managed by a central server or a coordinating node, the daemons securely connect to a distributed file system that maintains a set of pending tasks and *processing directives*. These processing directives allow the administrator(s) of the Forager cluster to assign specific rules to the resources. Directives include items such as the particular time of day that the resource may be used, the maximum number of cores that should be assigned to tasks, process priorities, and requirements for specific hardware. When the processing directives permit, the daemon will remove one or more of the tasks from the *pending task queue* and begin execution. The pending task queue contains a *task entry* for each task submitted to the system. Task entries describe:

- Current status (pending or executing)
- The process to be executed and its parameters
- Time of submission
- Resources actively executing the task and their associated start times

An additional list is maintained to record completed and failed tasks. Modifications to the lists are submitted as *transactions*, ensuring the list state will remain consistent even in the event of a failure.

3.2. Task Composition

Forager tasks are composed of several processing steps. Scenario variants are created in an initial partitioning phase and stored in the distributed file system. Next, the variants are loaded and executed during the *map* phase. In the *reduce* phase that follows, raw simulation outputs are compressed and filtered to produce a final dataset that is used for knowledge extraction.

Since our MapReduce implementation must deal with frequently changing execution conditions, it provides three options for stopping a running task: immediate termination, memory-resident suspension, and hibernation. *Immediate termination* results in the loss of all progress made on the task, but releases all resources immediately. This feature is useful in situations where the host machine must shift resources at a moment's notice. *Memory-resident suspension*, on the other hand, ceases execution but keeps the simulation in memory, which is helpful when an idle resource becomes busy for a short period of time. Finally, we added an optional *hibernation* function that Forager tasks can implement to gracefully serialize their state and store it in the distributed file system. This allows Forager to cope with situations where a task needs to be migrated to a different machine or when processing directives will prohibit execution for a substantial amount of time. Hibernating a task requires some time to complete; for our particular simulation, tasks took about 4.2 seconds to hibernate on average (over 1000 samples).

3.3. Situational Scheduling

Diverging from the standard MapReduce execution model, Forager daemons *pull* tasks from the distributed task queue when they can contribute CPU cycles or other resources. This allows dedicated hardware to continually execute new tasks, while daemons on shared systems can wait for free resources or until processing directives are met. To facilitate this approach, the Forager daemon monitors performance statistics on a per-machine basis. These statistics include the CPU idle time, steal time (in virtualized environments), memory and disk usage, load averages, the context switch rate, and the current number of active users on the machine. The lack of keyboard or mouse activity can also be used to inform the system of an idle resource, but is used on a case-by-case basis; modern workstations often have several CPU cores available, and our goal is to be able to partially leverage resources even while others are using them.

While a scenario variant that has been running much longer than usual may simply represent a particularly CPU-intensive chain of events unfolding in the simulation, it can also indicate that the daemon managing the task has insufficient resources. We use the aforementioned performance statistics to track resource utilization, and generate *speculative tasks* in the event of a slowdown. Speculative tasks in our framework have the option of using previously hibernated state information to help reduce duplicate processing work, unlike the speculative tasks seen in frameworks such as Hadoop [10].

Idling workstations and servers were one of the primary motivating factors cited by Tanenbaum [16] for distributed computing. These idle resources are especially prevalent in large businesses and academic settings, and are prime targets for *cycle scavenging* (exploiting unused processing resources). Figure 6 demonstrates this phenomenon on a busy web server at Colorado State University: during working hours (from about 7 AM to 3 PM) the load average (number processes waiting for or using the CPU) remains fairly high. However, during nighttime the server is mostly idle, presenting an opportunity for our framework to run scenario variants. Forager conducts *situational scheduling* to

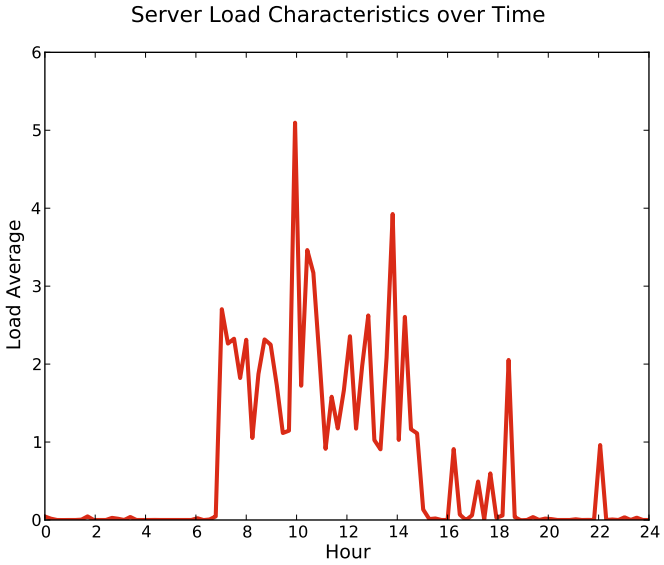

Figure 6: One-minute load average on a busy 4-core web server. Note the increase in peak load during working hours (7 AM to about 3 PM). At night, the server is mostly idle.

exploit these occurrences, where task scheduling is based on current machine load characteristics and the time of day (derived from temporal usage patterns or preset intervals).

3.4. Leveraging Elastic Clouds

For situations where the processing requirements of a simulation outstrip the capabilities of the resources acquired through cycle scavenging and private clusters, we incorporated support for cloud deployments as well. Forager allows participating hardware to join and leave the system at any time, making elastic cloud services an ideal means to supplement its resource pool. Furthermore, Amazon provides EC2 *spot instances*, which enable users to trade reliability for lower prices. In essence, spot instances apply the laws of supply and demand to virtualization: users issue a *spot request* with their desired virtual machine (VM) characteristics, a maximum price, and the dates and times the request is valid for. If the market price of the VM exceeds the maximum specified, the spot instance is terminated. On average, spot instances cost 50-60% less than their traditional counterparts.

In our use case, reliability is a secondary concern behind overall processing throughput. Tasks are lightweight, self-contained, and are executed over a relatively short period of time. However, the spot VMs we used were never terminated during our testing period (likely due to running during low-demand summer months). We tested a range of VM configurations on the Amazon public cloud as well as our own private cloud. Table 1 illustrates the performance

differences between the configurations. These include VMs from the Amazon m1, c1, m3, c3, and t2 instance families, along with virtualized and bare metal results for our hosts running OpenStack Compute with the KVM hypervisor on Fedora 20 (Xeon E5620 with 12 GB RAM, Xeon E3-1220v2 with 8 GB RAM).

Table 1: Scenario execution times in a virtualized environment averaged over 1000 iterations.

Hardware	**Mean Execution Time (s)**	**SD σ (s)**
t2.micro	442.94	215.79
m3.medium	118.84	21.04
c3.large	57.07	10.06
m1.medium	51.80	8.87
c1.xlarge	50.79	8.82
c1.medium	48.66	7.03
E5620 (KVM)	64.90	10.47
E5620 (Bare)	60.02	8.60
E3-1220v2 (KVM)	49.86	7.76
E3-1220v2 (Bare)	47.33	5.70

While these results closely mirror the theoretical performance differences between instances, it is worth noting that the t2.micro can also achieve competitive results when enough CPU credits are available for "burst" performance, which temporarily allows the instance to consume more than its baseline allotted CPU time. When burstable, the average scenario execution time on a t2.micro was 99.27 seconds. Since Forager can migrate longer-running tasks and monitor CPU steal time at each resource, burstable instances are a viable option for cycle scavenging as long as the primary function of the VM does not tax the CPU. Interestingly, the previous-generation m1 and c1 instances exhibited better performance than their newer counterparts for our particular workload. We configured our AWS spot requests based on the price:performance ratio derived from these results and also set a hard upper bound for price based on our budget. In general, our spot requests were designed to choose the lowest priced VMs available, and avoided m3.medium instances unless there was a substantial cost benefit. Our systems experienced 8.1% and 5.3% virtualization overheads on the E5620 and E3-1220v2 processors, respectively.

4. Output Management and Storage

Rather than relying on a central server or coordinator process, our framework stores its persistent state in a distributed file system. This information includes pending and executing tasks, resource performance statistics, and the overall

```
Minnesota_Variant542.json
{
    "OutbreakDuration" : 16,
    "Infections" : {
        "Direct" : 32,
        "Indirect" : 6,
        "Airborne" : 0,
    },
    "AnimalsDestroyed" : {
        "SwineSmall" : 0,
        "Sheep" : 24,
        "Beef" : 542,
        "Dairy" : 312,
    },

    ...
}
```

Figure 7: A simplified example of the JSON output produced by a simulation run. Each output file contains a separate entry for every simulation day and its corresponding outputs.

system status. Even more importantly, the distributed file system is tasked with managing and storing simulation outputs as well. These outputs must be stored in a scalable and fault-tolerant manner: the 400,000 iterations produced from our single pilot scenario (one in a multitude of exploration possibilities) consumed about **1 TB** of storage space. We use our Galileo [17, 18, 19, 20] DHT-based key-value store to fulfill these requirements. Galileo is a distributed, fault-tolerant, and document-oriented storage system, making it an ideal candidate for managing the JSON output files produced by our subject simulation. Figure 7 provides an example of a JSON output file along with some of the variables that might be produced by a simulation run. These variables are used by our framework during the dimensionality reduction and modeling process.

4.1. Output Compression

To help manage output file sizes, we evaluated each of the compression algorithms available in Galileo: LZO, DEFLATE, Burrows-Wheeler, and LZMA. Outputs were stored in append-only *blocks* of approximately 1,000 MB each (320 simulation iterations) before compression. Table 2 contains the resulting file sizes and their corresponding compression ratios for each of the algorithms surveyed.

For the JSON outputs produced by our simulation, the LZMA algorithm provided the best average compression ratios. However, we also considered compression speed; Table 3 provides compression and decompression times for each of the algorithms.

Table 2: Output compression averaged over 32,000 iterations.

Algorithm	**File Size (MB)**	**Compression Ratio**
No Compression	978.2	1.0
LZO	174.1	5.6
DEFLATE	97.2	10.0
Burrows-Wheeler	37.6	26.0
LZMA	34.1	28.6

Table 3: Compression and decompression times averaged over 32,000 iterations.

Algorithm	**Compression (s)**	**Decompression (s)**
LZO	1.9	6.4
DEFLATE	16.5	6.6
Burrows-Wheeler	179.0	15.7
LZMA	211.43	7.61

In our specific use case, raw outputs will be compressed once and decompressed several times later during analysis and forecasting. This led us to choose the LZMA algorithm due to the compression ratios it achieved on our dataset as well as its decompression times, which were competitive with the fastest algorithms (LZO and DEFLATE). Ultimately, output compression saves a substantial amount of disk space and greatly increases exploration capabilities on a given set of hardware.

5. Knowledge Extraction: Modeling and Prediction

After orchestrating our scenario variants across the resource pool and storing their outputs, we begin the final processing step of our framework: building predictive models. These models generalize the scenarios to allow interactive exploration of their parameter space. Building the models is a one-time process that initializes our real-time forecasting engine. To make the predictions, we used both multivariate linear regression and artificial neural networks (ANNs).

5.1. Dimensionality Reduction

Our particular simulation involves a large number of inputs and outputs. When making predictions, these values contribute to a very high overall dimensionality. To help reduce the effects of the *curse of dimensionality*, we evaluated two methods commonly used for dimensionality reduction: *principal component*

analysis (PCA) and *correlation analysis*. PCA is a method that can be used to project a dataset onto a lower dimensional space. This is achieved by selecting components that contribute the most to the underlying variability in the data. However, PCA does not consider the relationship between input and output variables in our case, which can lead to the removal of some inputs that may influence outcomes. To avoid this issue, we used correlation analysis with the Pearson correlation coefficient (PCC) to measure the degree of correlation between input variables and the outputs. Using this approach enables us to select input variables that tend to have a strong influence on simulation outcomes, which are then used to build our models.

5.2. Prediction Methods

To create our validation and test datasets, we used k-fold cross-validation with $k = 10$. We generated our models with artificial neural networks (ANNs) and multivariate linear regression, and then evaluated the root-mean-square error (RMSE), creation times, and prediction times of both methods. In our case, RMSE (the average prediction deviation) and prediction times are critical in evaluating both the accuracy of our forecasts and the overall forecasting speed. For both methods, we generated an individual model for each output variable.

Artificial neural networks are non-linear computational models inspired by the characteristics of biological neural networks. ANNs can be used for a variety of machine learning applications, and are composed of interconnected *neurons* that are responsible for processing information. In our framework we used a feedforward neural network from the PyBrain machine learning library, which was trained with backpropagation. The network was configured with one hidden layer, and the number of hidden units was selected empirically through an iterative process.

Multivariate linear regression is an approach used for modeling the relationships between multiple dependent variables and multiple independent variables. It produces a set of linear predictor functions that we then use to forecast scenario outcomes. We also evaluated several methods for optimizing regression results and settled on the Least Absolute Shrinkage and Selection Operator (Lasso), which penalizes the absolute size of regression coefficients to help reduce the influence of variables that have little impact on the model. Out of the optimization methods we tested, Lasso provided the best predictive results.

5.3. Experimental Results

To evaluate our models, we used the inputs and outputs from the 10,000 scenario variants produced by our framework. We considered a total of 1,812 raw input variables, and selected 10 key output parameters based on the guidance we received from epidemiologists. These outputs include items such the disease duration and number of infected animals in the scenario, which are helpful in various forms of analysis (such as determining the economic impact of a particular type of outbreak).

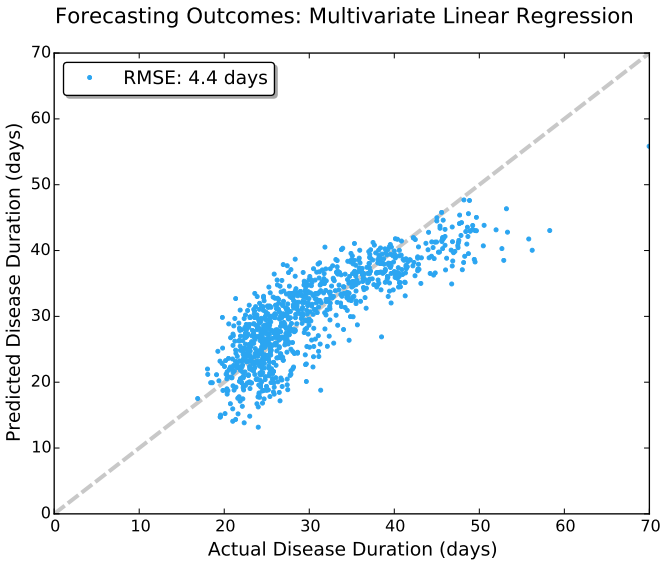

Figure 8: Prediction accuracy for disease duration using multivariate linear regression. Samples close to the 45-degree reference are highly accurate.

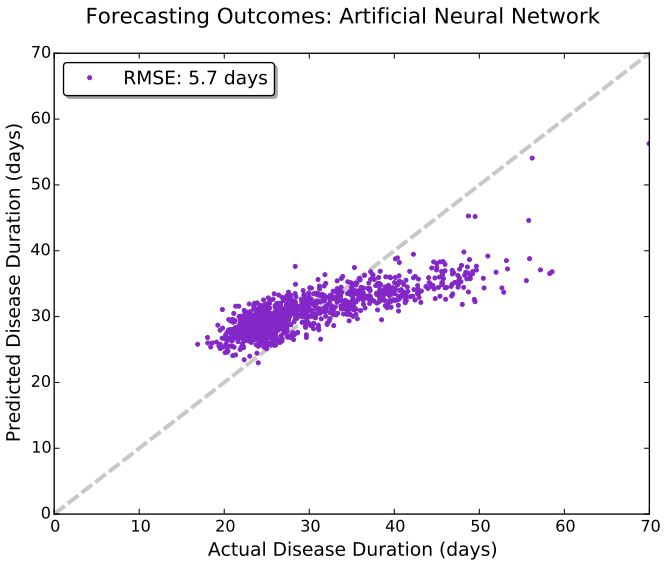

Figure 9: Prediction accuracy for disease duration using an artificial neural network. Samples close to the 45-degree reference are highly accurate.

Table 4 contains performance statistics for both of our prediction models built with a 133 input parameter set that exhibited high correlation with the output parameters. Predictions were performed 1,000 times to illustrate how the framework would perform in a situation that required results from a large number of models. Multivariate linear regression offered the best performance in both of our timing criteria (time to build the model, and time to make a prediction), but it is worth noting that both methods provided sub-second prediction performance.

Table 4: Model generation times and prediction performance.

Model	**Build (s)**	**Predict 1,000 Times (ms)**
Regression	1.5	0.2
Neural Network	7998.0	75.0

To visualize the prediction accuracy of our framework, Figures 8 and 9 contain the actual values of the disease duration output variable plotted against predictions for both multivariate linear regression and an artificial neural network. Values close to the 45-degree reference line are highly accurate. The root-mean-squared error (RMSE) for the multivariate linear regression test was **4.4 days**, while the RMSE exhibited by our neural network was **5.7 days**. Overall, these values indicate that both models were able to fit the data and provide forecasts in a timely manner, but multivariate linear regression provided better performance in our specific use case. Figure 10 demonstrates prediction accuracy using an alternative representation; 70 randomly-selected points have been plotted from the predicted outcomes along with their corresponding actual values to illustrate how closely the predictions have fit the data.

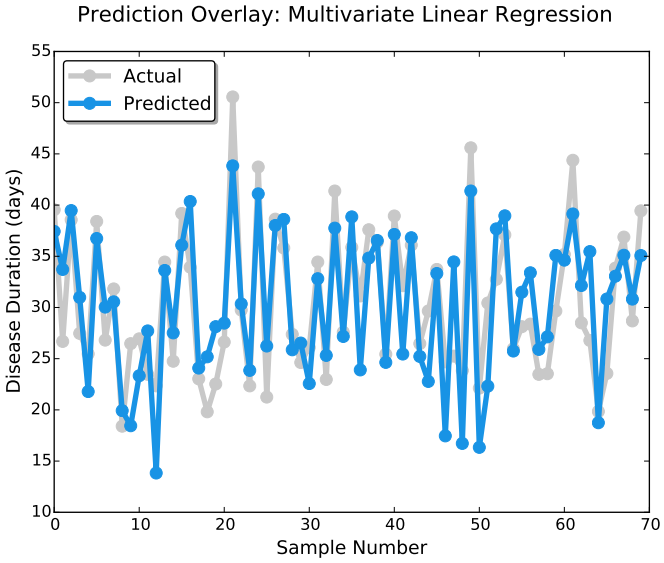

Figure 10: Predictions using multivariate linear regression for 70 randomly-selected samples. Here predicted values are overlaid on the actual values to illustrate how closely our predictions fit the data.

6. Increasing Prediction Accuracy

Our initial experiments with multivariate linear regression and artificial neural networks illustrate the efficacy of our framework in making accurate, real-time projections of discrete event simulation outputs. While these models provide a baseline predictive performance measure, there are several other techniques that we considered to improve the accuracy of our predictions: (1) analysis of model errors to determine which methods would yield accuracy improvements, (2) use of *ensemble methods* to combine the insights of several disparate models, and (3) improvements to our dimensionality reduction process to avoid collinearity and decrease training times. To determine whether our models would benefit from a larger training set, we generated a new 100,000-variant dataset. This dataset was also used to retrain the model with the best performance from the previous section (multivariate linear regression) to act as a point of reference for performance comparisons.

6.1. Prediction Error Analysis

The first step we took toward improving model performance was to investigate prediction errors and determine their source. We chose to analyze the mean squared error (MSE) of the predictions, which measures the average squared difference between the expected and predicted outputs produced by a model. For a given predictor $\hat{\theta}$, the MSE can be decomposed into *bias* and *variance*

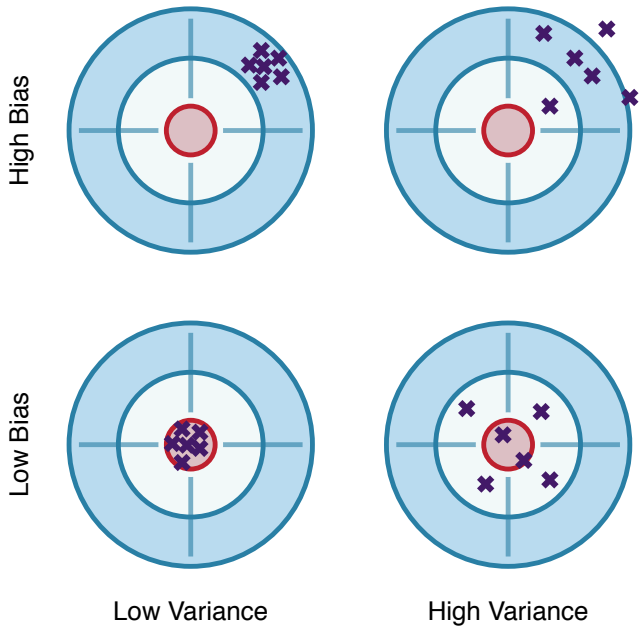

Figure 11: An illustration of the Bias-Variance Trade-off. An ideal model, shown in the lower left-hand corner, has low bias and variance.

components [21, 22, 23]:

$$MSE(\hat{\theta}) = Bias(\hat{\theta})^2 + Var(\hat{\theta})$$

Bias is influenced by assumptions made in the model that cause some of the relationships between inputs and outputs to be overlooked. High bias relative to the MSE often results in *underfitting*, which means that the model does not fully capture underlying trends in the data. On the other hand, *variance* describes how sensitive the model is to variations (noise) in the training dataset. Models with high variance relative to the MSE are greatly influenced by noise in the data, often resulting in *overfitting*. These components and their respective influences are referred to as the *Bias-Variance Trade-off* [21]. Figure 11 illustrates this concept; an ideal model should have low variance and be unbiased.

In practice the entire training set is used to build a model, resulting in a single output space. This complicates the process of determining the bias and variance of a model. To overcome this issue, we use *bootstrapping* [21] to generate additional training sets. Bootstrapping involves sampling with replacement from the base training set to create several new representative training sets. In this particular study we created 100 new models from the base training set and produced 100 individual forecasts for each output variable. Table 5 illustrates the bias-variance decomposition of the errors associated with predicting the NAADSM *outbreak duration* output variable. These results were produced

Table 5: Bias-Variance decomposition of our multivariate linear regression model for the *outbreak duration* output variable on our 100,000-variant dataset. In this case, the model exhibits high bias but low variance.

Dimensions	$RMSE$	MSE	$Bias^2$	$Variance$
333	6.31	39.83	39.75	0.08
309	6.31	39.84	39.76	0.08
261	6.32	39.89	39.84	0.05
200	6.42	41.20	41.17	0.03
170	6.75	45.50	45.48	0.02
121	6.81	46.38	46.36	0.02
59	7.48	55.94	55.93	0.01

using our 100,000-variant dataset and the multivariate linear regression model described in the previous section, and include a spectrum of dimensionality reduction levels to capture the impact of dimensionality on the prediction error.

Testing the influence of dimensionality on the predictions enables us to reduce dimensionality while retaining accuracy; in general, having a large number of input dimensions results in longer model generation and training times. Additionally, analyzing the influence of bias and variance on model accuracy allows us to select the appropriate methods for reducing overall prediction error. However, there is a trade-off between bias and variance: a model could significantly reduce bias while also increasing variance, for example. To help manage this trade-off, *ensemble methods* employ multiple models to improve aggregate forecast performance.

6.2. Applying Ensemble Methods

To improve model accuracy, we use *ensemble methods* to build a collection of models for each output parameter. Ensemble methods rely on the combined insights of multiple learning algorithms to form several models (an ensemble). A *homogeneous* ensemble applies a single learning algorithm multiple times, whereas a *heterogeneous* ensemble makes use of multiple algorithms. The ensemble learning process for regression can be divided into three phases [24]:

1. ***Model creation***, where several models are generated for each output parameter;
2. ***Pruning***, which removes under-performing models from the ensemble to increase overall performance;
3. ***Integration***, where the aggregate insights from the models are merged into a single forecast

The main advantages derived from using ensemble methods over a single model are increased prediction accuracy and better robustness (ability to function well

on new, unseen data). Several methods exist for creating ensembles, but the two most widely used approaches are *bootstrap aggregating* (commonly referred to as *bagging*) and *boosting*.

Bagging [25] aims to build ensembles with stable outputs by combining several models that make highly accurate predictions at a particular part of the input space, but may be less accurate for other inputs. Similar to the method described in the previous subsection for performing prediction error analysis, model inputs are created by generating several bootstrap samples. These samples incorporate duplicate information, causing portions of each model input space to *overlap* with the others. This ensures coverage of the input space while allowing each model to specialize for its unique inputs, boosting accuracy. Compared to the base predictor, ensembles created with bagging have a tendency to reduce the overall prediction variance of the model [25, 26]. Additionally, each model in the ensemble is completely independent, enabling parallel creation and execution.

Boosting [27, 28] refers to a number of techniques that iteratively convert *weak* predictors to *strong* ones. A weak predictor is only somewhat effective at forecasting outputs, whereas a strong predictor will produce results with low relative error. Unlike bagging, boosting builds the prediction models in a sequential stagewise fashion. Boosting methods assign each observation a *weight* and then update the weights at each iteration of the training process based on observed prediction errors. Weights associated with weak predictions will be increased, while those associated with strong predictions will be decreased. This process guides the subsequent training stages by shifting the focus of the models toward observations that were inaccurately predicted. In general, boosting methods are most applicable in situations where the base model is highly biased.

This study evaluates both bagging and boosting ensemble methods; we selected *random forests* as a bagging method to reduce prediction variance, and *gradient boosting* to target reductions in model bias. Comprehensive discussion of these methods along with their experimental results are provided in the following subsections. Similar to the experiments reported in the previous section, we used k-fold cross-validation with $k = 10$ to create and validate our test datasets. The reported results are based on the averages for predictive accuracy over the k rounds.

6.3. Random Forests

Random forests are an ensemble method that independently build decorrelated decision trees on different bootstrap sampled versions of the training data. Decision trees are used because they are able to capture complex patterns in the data and have relatively low bias. By averaging the decision trees, random forests tend to reduce prediction variance. We have applied random forests on our 100,000 scenario variant dataset to build prediction models for each output variable. In this experiment, we observed different degrees of improvement in prediction accuracy for some output variables; Figure 12-b shows the prediction improvement for the *disease duration* NAADSM output variable, and Table 6

compares RMSE values produced by linear regression and random forests across a variety of output variables. The ensemble was composed of 300 decision trees whose depth extended until there was only one observation left to split.

Table 6: Prediction RMSE for linear regression and random forests across a variety of NAADSM output variables.

Output Parameter	**Linear Regression**	**Random Forest**
Vaccine immune units	390.00	395.00
Destroyed units	8.09	4.88
Disease Duration (days)	4.38	2.84

6.4. Gradient Boosting

Gradient boosting is a method that attempts to optimize a differentiable loss function, which iteratively increases the expressive power of the base prediction model. In turn, this decreases prediction bias. We built prediction models based on decision trees for each output variable using gradient boosting with our 100,000 scenario variant dataset. The prediction results we obtained show improvement across all output variables. Figure 12-c shows the predicted disease duration in days using gradient boosting against the actual disease duration. Gradient boosting outperformed linear regression (which was configured to use 216 input variables for this particular experiment) and random forests for all output parameters. Table 7 includes the lowest RMSE of the three approaches across a variety of output variables. This experiment involved 300 gradient boosting stages and limited the maximum depth of the decision trees to three levels.

Table 7: Prediction RMSE for linear regression, random forests, and gradient boosting across a variety of NAADSM output variables.

Output Parameter	**Linear Regression**	**Random Forest**	**Gradient Boosting**
Outbreak duration (days)	6.23	6.53	4.68
Total units infected	7.79	7.71	5.28
Disease duration (days)	4.38	2.84	2.18

6.5. Performance Evaluation

Both of the ensemble methods we evaluated outperformed the multivariate linear regression model trained on our 100,000-variant dataset. Figure 12 compares all three approaches; note that there was a slight improvement in RMSE

Table 8: Model generation times, in seconds (s), across a variety of dimensionality reduction levels.

Dimensions	**Linear Regression**	**Random Forest**	**Gradient Boosting**
28	2.5	318	293
55	234	642	616
107	175	1248	984
153	659	1741	1730

Table 9: Time to make 10,000 predictions for each model, in milliseconds (ms). A variety of dimensionality reduction levels is provided for comparison.

Dimensions	**Linear Regression**	**Random Forest**	**Gradient Boosting**
28	2	900	90
55	2	1410	110
107	2	1280	110
153	4	2420	130

over the previous experiments conducted with multivariate linear regression, but both random forests and gradient boosting have clearly improved over our previous models.

To further compare the trade-offs between techniques, we evaluated model training times on 90,000 scenario variants from our training set, as well as the time taken to produce 10,000 predictions for each model. These benchmarks were run on hardware from our test cluster equipped with a Xeon E5620 processor and 12 GB of RAM. Table 8 reports generation times for each of our models across a variety of dimensionality reduction levels. Based on our observed model performance there is a trade-off between prediction accuracy and generation times, but since model generation occurs infrequently these results do not pose a significant obstacle for our framework. Table 9 contains benchmark results from each model for after producing **10,000** predictions. All three models are capable of making a single prediction in far less than a second, ensuring the responsiveness of our framework.

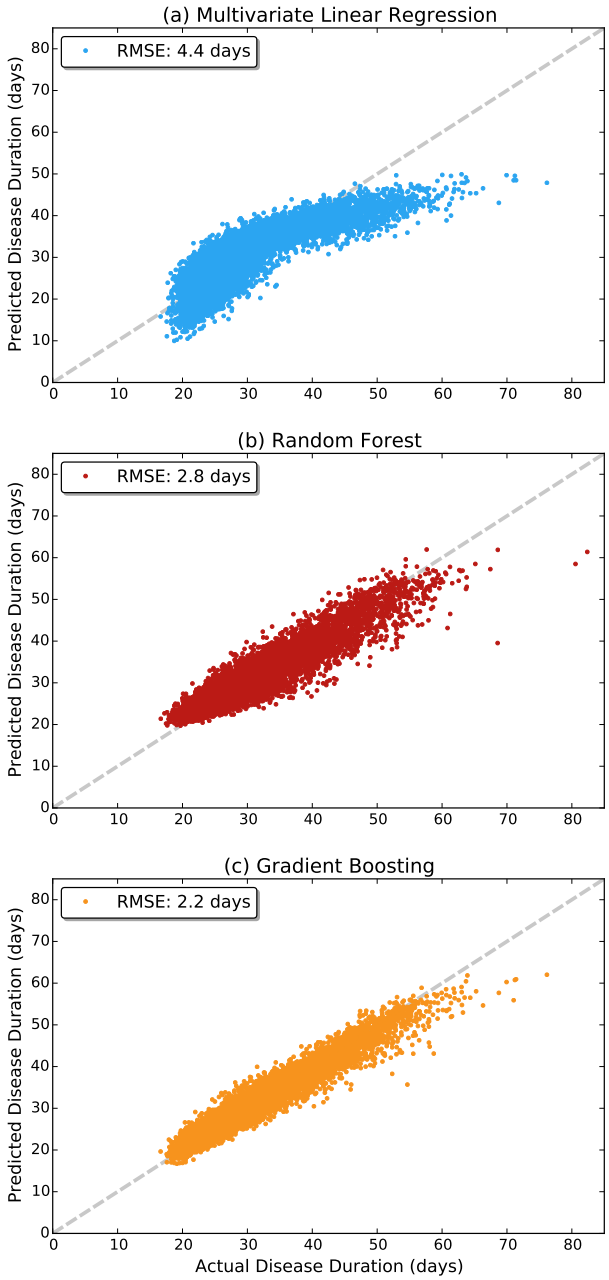

Figure 12: A comparison of predictions for the *disease duration* NAADSM output variable produced by multivariate linear regression, random forests, and gradient boosting on our 100,000-variant dataset.

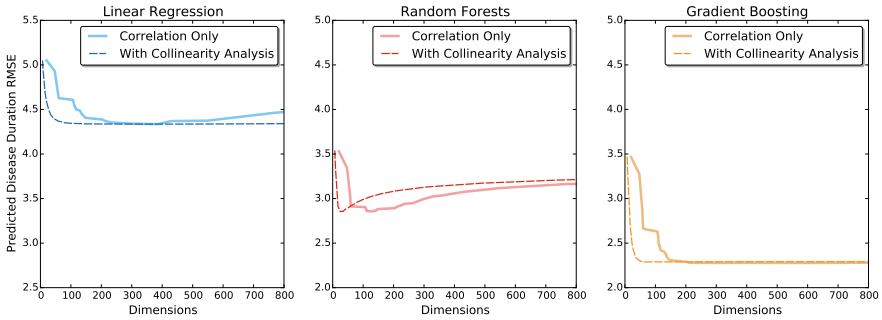

Figure 13: A comparison of prediction performance across a variety of dimensions with and without accounting for collinearity. In all three cases, removing inputs with high collinearity resulted in improved performance and required fewer dimensions.

6.6. Accounting for Collinearity

To further improve predictive performance, we extended our dimensionality reduction process to remove *collinearity* from the input data. The presence of collinearity is marked by very strong correlations between input variables, which could be used to linearly predict one another with high accuracy. In general, effective models contain input variables that exhibit strong correlations with the outputs, but not with each other. These inter-input correlations result in increased model noise because small variations in the variables with high collinearity can cause large changes in the model outputs.

We reduce dimensionality in two stages: first, input and output variables are inspected and any inputs that are not correlated with an output are removed. This decision is made using a configurable *correlation threshold*. Next, we inspect input variables and create subsets with high correlation coefficients based on a *collinearity threshold*. For each of these subsets we remove inputs that exhibit the lowest correlation with the output variables, ensuring that the variables that play the largest role in model outcomes are preserved. This process significantly reduces the number of input variables that must be considered when building our models, which in turn reduces processing times.

In general, reducing dimensionality will decrease processing times while also improving accuracy. However, finding the appropriate level of dimensionality depends on the models and input datasets. To select our correlation and collinearity thresholds, we ran several experiments that gradually decreased both thresholds iteratively and then inspected the RMSEs. Figure 13 plots the RMSE of the *disease duration* NAADSM output variable against dimensionality for each model, with and without accounting for collinearity. In all three cases, removing input variables that exhibited high collinearity improved the performance of the model and reduced the number of dimensions required to achieve best performance. Also note that higher dimensionality does not always result in improved prediction performance; in this experiment, both multivariate linear regression and random forest performance decreases as dimensions are added.

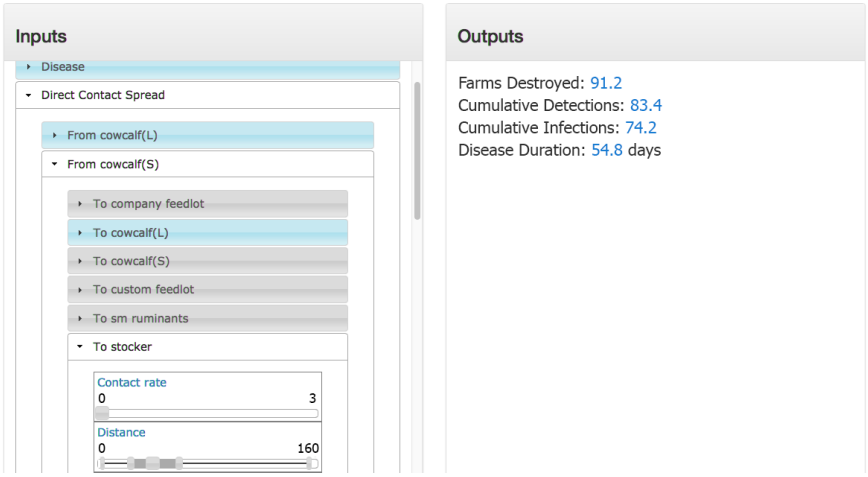

Figure 14: A demonstration of the What-If tool user interface. Changes made to the input parameters on the left are reflected in the output variables on the right.

7. The What-If Tool

To facilitate exploratory analysis, we integrated the prediction models discussed in the previous sections into a user-friendly "What-If Tool." This tool allows modelers to interactively adjust input parameters and observe their effect on a set of output variables in real time. The primary goal of the What-If tool is to remove the time lag imposed by needing to re-run a simulation after each change to the parameters. To target the largest possible audience (including both users in a lab setting as well as mobile devices in the field), the tool is implemented as a web interface. We used the Twitter Bootstrap framework for basic page layout tasks and jQuery UI for the widgets. The layout works in a two-pane setup: input parameters on the left, output variables on the right. Changes to an input parameter result in immediate recalculation of any outputs dependent on the input. Figure 14 illustrates this process.

The large number of input parameters is managed by using "accordion" widgets, which display one "open" section of parameters at a time, automatically closing all other sections of parameters. Some sections of parameters have subsections: for example, the "Direct Contact Spread" section opens to reveal sections for spread from each farm type, which in turn open to reveal sections for spread from the selected farm type to the other farm types.

The tool has three kinds of input parameters: numeric, probability density function (PDF), and line chart (a piecewise linear function). Numeric parameters are represented by a simple slider. PDF parameters are represented by a 5-handle slider, styled to resemble a box-and-whisker plot. The 5 handles correspond to 0, 25, 50, 75, and 100% of the cumulative area. (If the original model scenario used PDFs with infinite left or right tails, the leftmost and rightmost slider handles are positioned to cover 99% of the PDF's area.) For line chart

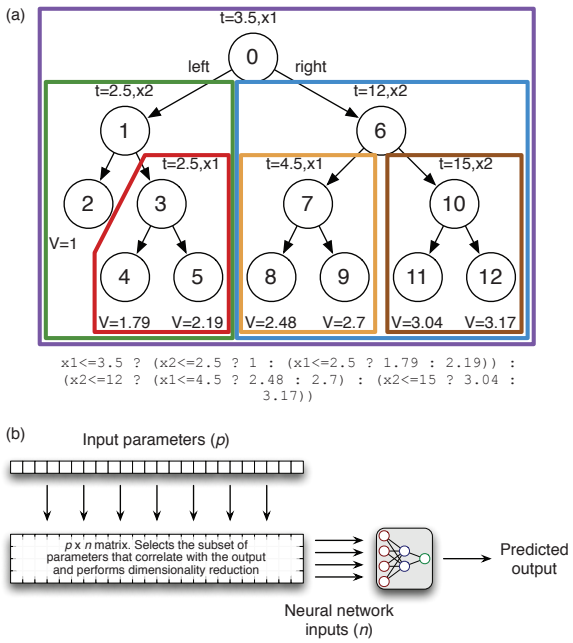

Figure 15: (a) A decision tree transformed to Javascript code by depth-first traversal. (b) Each Predictor (in this example, an artificial neural network) operates on a subset of the input parameters.

parameters, the chart from the original model scenario is drawn using embedded SVG graphics. A pair of range sliders, one vertical and one horizontal, allow the modeler to shift, stretch, or compress the chart along either or both axes. This is the same set of transformations to an line chart that are allowed by the scenario variant generator (described in Section 2).

The web interface is generated using PageFactory objects, which read the original model scenario file containing the names of the farm types, zones, etc., and set up the sections of input parameters. A PredictorFactory generates Predictor objects from the files created by the training process, with one output variable per Predictor. Each prediction model (artificial neural network, linear regression model, etc.) is implemented as a subclass of the Predictor class and is responsible for generating the appropriate Javascript code for its prediction task. Figure 15-a illustrates how a decision tree is translated to Javascript by system, and Figure 15-b shows how predictors operate on subsets of the input space. Similarly, a LinearRegressionPredictor will generate a JavaScript expression of the form:

$$output = a_1 x_1 + a_2 x_2 + ... + a_n x_n + intercept$$

Random forest decision trees are translated into ternary expressions:

$$output = a_i > threshold_j \ ? \ right : left$$

Where $right$ and $left$ may be either nested ternary expressions (interior nodes in the tree) or numbers (leaf nodes). Several expressions of this form are averaged to compute the final prediction from the Random Forest.

An artificial neural network is translated into JavaScript that sets up three 2D arrays, representing the input layer (l_i), the weights connecting the input layer to the hidden layer (w_{ih}), and the weights connecting the hidden layer to the output layer (w_{ho}). The code to compute the output node's value given the weights and activation function then looks like:

$$output = matrixMult([matrixMult(l_i, w_{ih})[0].map(tanh)], w_{ho})[0][0]$$

Translating each prediction model into a JavaScript expression avoids the need to load specialty libraries (e.g., a neural network library) along with the web interface. However, the expressions generated for Random Forest and artificial neural network predictors may be bulky. We are examining options for managing the size, for example, reducing the precision of numbers, or unpacking the data from a compressed binary form. We may also shift computations to a web service for models that require large amounts of memory or processing time.

8. Related Work

Parallel and distributed discrete event simulation has been well-studied in the literature [5, 29, 30]. While these simulations often have numerous parallelization opportunities, they also generally require fine-grained synchronization. Most importantly, a parallel version of a DES must ensure correctness, i.e., the parallelization does not change the output of an identical simulation run on a single thread. Our framework avoids these issues by treating the DES as a black box, and contrasts with parallel DES by providing real-time forecasts. However, parameters discovered with our framework that produce an outcome of scientific interest could also be run in a parallel DES to confirm the results in an efficient manner.

Distributed task queues like Celery [31] leverage the technologies behind message-oriented middleware (such as RabbitMQ [32]) to manage distributed execution. Like Forager, Celery supports automatic scaling, resource monitoring, and rate/time limits. However, these limits are generally designed to enable load balancing rather than sharing resources with other users. This class of system is often designed for short-lived task execution and requires a more involved setup and administration process. Additionally, many message broker implementations are centralized or represent single points of failure. Further, implementing an execution queue on a system with transactional semantics instead (such as a distributed database) has several advantages [33].

Hadoop [10, 34] and its accompanying file system, HDFS [35] share some common objectives with our framework. Hadoop is an implementation of MapReduce [7], which often involves multiple *waves* of execution that must be completed before the next wave can start. Additionally, the system is often installed on a dedicated cluster and does not need to suspend or migrate tasks when the underlying resources become busy. Like the distributed file system we use in this work, HDFS replicates information across multiple physical machines to ensure failures do not result in data loss.

MongoDB [36] is a distributed document store that can employ MapReduce computations for analysis. Similar to Galileo, MongoDB supports range queries, data replication, and clustering. Its storage format, BSON, is a binary serialization of JSON documents. This means that other formats (such as XML, INI, YAML) must be converted to their equivalent JSON representation before being stored in the system. While MongoDB is scalable and efficient at resolving queries, it also imposes some limitations on the size and quantity of documents being stored.

The Berkeley Open Infrastructure for Network Computing (BOINC) [15] is a volunteer computing platform that enables home users or organizations to contribute their idle processing resources towards a variety of scientific projects. The platform can also be used privately by organizations to create a lightweight grid environment. Unlike our framework, BOINC is generally run on untrusted hardware and requires duplication of tasks to ensure the validity of their outputs. Additionally, BOINC clients are usually deployed on single-user computing devices rather than public resources, and are administered individually.

Grid computing technologies, such as the Globus Toolkit [37] or computing management frameworks such as HTCondor [38] share a common goal of creating distributed processing and storage environments. These deployments often combine the computing hardware of multiple organizations into a single coherent (and heterogeneous) resource pool. They also support *cycle scavenging*, where idle resources are used for background processing. Unlike volunteer computing, these execution frameworks are administered by a central organization and do not have to deal with untrusted resources. Our framework is designed for single-organization installations, and requires less administrative setup and maintenance.

GEODISE [39, 40, 41] is a user-friendly wrapper for HTCondor that enables multidisciplinary processing and data management functionality. GEODISE supports scripting environments such as MATLAB and Jython and can model the dependencies and flow of information with an integrated scientific workflow editor. Like Forager, GEODISE monitors resources to determine which are currently available for use. However, the framework is much more involved compared to the ad-hoc usage pattern Forager is intended for.

9. Conclusions and Future Work

Producing accurate forecasts for discrete event simulations involves: (1) ensuring coverage of the parameter space, (2) efficient orchestration of workloads, (3) amortizing the I/O costs associated with data accesses, (4) coping with dimensionality, (5) building and training lightweight prediction models, and (6) carefully planning which aspects of the framework are in the critical path.

Ensuring statistical coverage of the parameter space provides us with better training data for the learning structures and prediction models. Forager's pull-based approach during orchestration of workloads allows nodes to take on tasks when they are able: lightly used machines execute more tasks than others and a have greater share of the computational workload. Our orchestration scheme works well with both physical machines in shared, public clusters and virtual machines in cloud settings, including spot instances. The use of a DHT-based key-value store, Galileo, allows us to distribute the I/O loads of the outputs generated by the simulation for training of the prediction models.

Dimensionality reduction allows us to identify inputs that contribute to the outputs. Pruning of the input parameter space allows us to better train the prediction models. The pruning process reduces training times and also improves the accuracy of the predictions by eliminating inputs that are sources of statistical noise. Our approach balances the costs associated with training and making predictions. Though the training process is compute-intensive, it is not in the critical path during predictions. Once the training process is complete, making the forecasts based on the constructed models is lightweight and simply involves dot product calculations that can be done in real time. Building a prediction model per output parameter allows identification of inputs (and their respective contributions) to the output. Though this increases the overall training time, the improved accuracy in the predictions offsets this cost.

Our future work will focus on both the orchestration framework and our predictive models. While Forager can exploit spot instances, learning structures or rule-based directives could be used to optimize for the cost of VMs as well. On the prediction front, we will investigate additional learning structures and dimensionality reduction techniques. The ensemble methods that we have explored (random forests and gradient boosting), attempt to reduce only one of the prediction error components, but other ensemble approaches have been proposed to reduce the bias and variance simultaneously. Examples of such approaches include stochastic gradient boosting and iterated bagging. In stochastic gradient boosting, the ensemble is built with the objective of reducing prediction variance of a gradient boosting ensemble that has already reduced the model bias. Iterated bagging, on the other hand, attempts to reduce the prediction bias of bagging ensemble methods. As part of our future efforts we will be exploring the suitability of these methods in the construction of prediction models.

Acknowledgments

This research has been supported by funding (HSHQDC-13-C-B0018) from the US Department of Homeland Security's Long Range program and the US National Science Foundation's Computer Systems Research Program (CNS-1253908).

References

[1] N. Harvey, A. Reeves, M. Schoenbaum, F. Zagmutt-Vergara, C. Dubé, A. Hill, B. Corso, W. McNab, C. Cartwright, M. Salman, The North American Animal Disease Spread Model: A simulation model to assist decision making in evaluating animal disease incursions, Preventive Veterinary Medicine 82 (2007) 176–197.

[2] D. Pendell, J. Leatherman, T. Schroeder, G. Alward, The economic impacts of a foot-and-mouth disease outbreak: a regional analysis, Journal of Agricultural and Applied Economics 39 (2007) 19–33.

[3] C. Green, T. Whiting, G. Duizer, D. Douma, H. Kloeze, W. Lees, A. Reeves, Simulation modeling of alternative control strategies for an HPAI outbreak using NAADSM, in: Canadian Association of Veterinary Epidemiology Preventive Medicine (CAVEPM) Meeting, May 29 - 30 2010, Guelph, Ontario, Canada.

[4] K. Portacci, A. Reeves, B. Corso, M. Salman, Evaluation of vaccination strategies for an outbreak of pseudorabies virus in US commercial swine using the NAADSM, in: ISVEE 12: Proceedings of the 12th Symposium of the International Society for Veterinary Epidemiology and Economics, Durban, South Africa, p. 78.

[5] Z. Sui, N. Harvey, S. Pallickara, On the distributed orchestration of stochastic discrete event simulations, Concurrency and Computation: Practice and Experience (2013).

[6] M. Malensek, Z. Sui, N. Harvey, S. Pallickara, Autonomous, failure-resilient orchestration of distributed discrete event simulations, in: Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference, CAC '13, ACM, New York, NY, USA, 2013, pp. 3:1–3:10.

[7] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Communications of the ACM 51 (2008) 107–113.

[8] M. Malensek, W. Budgaga, S. Pallickara, N. Harvey, F. Breidt, S. Pallickara, Using distributed analytics to enable real-time exploration of discrete event simulations, in: Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on, pp. 49–58.

[9] M. D. McKay, R. J. Beckman, W. J. Conover, A comparison of three methods for selecting values of input variables in the analysis of output from a computer code, Technometrics 21 (1979) 239–245.

[10] A. Bialecki, M. Cafarella, D. Cutting, O. O'Malley, Hadoop: a framework for running applications on large clusters built of commodity hardware, Wiki at http://lucene. apache. org/hadoop (2005).

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, ACM SIGOPS Operating Systems Review 41 (2007) 59–72.

[12] S. Pallickara, J. Ekanayake, G. Fox, Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce, in: Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on, IEEE, pp. 1–10.

[13] S. Pallickara, J. Ekanayake, G. Fox, An overview of the granules runtime for cloud computing, in: eScience, 2008. eScience'08. IEEE Fourth International Conference on, IEEE, pp. 412–413.

[14] G. Fox, S. Pallickara, M. Pierce, H. Gadgil, Building messaging substrates for web and grid applications, Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 363 (2005) 1757–1773.

[15] D. Anderson, BOINC: a system for public-resource computing and storage, in: Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on, pp. 4–10.

[16] A. S. Tanenbaum, Distributed operating systems, Prentice Hall, 1995.

[17] M. Malensek, S. Pallickara, S. Pallickara, Galileo: A framework for distributed storage of high-throughput data streams, in: Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on, pp. 17 –24.

[18] M. Malensek, S. Pallickara, S. Pallickara, Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals, Future Generation Computing Systems 29 (2013) 1049–1061.

[19] M. Malensek, S. Pallickara, S. Pallickara, Expressive query support for multidimensional data in distributed hash tables, in: Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing, UCC '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 31–38.

[20] M. Malensek, S. Pallickara, S. Pallickara, Autonomously improving query evaluations over multidimensional data in distributed hash tables, in: Proceedings of the 2013 ACM International Conference on Cloud and Autonomic Computing.

[21] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, R. Tibshirani, The elements of statistical learning, volume 2, Springer, 2009.

[22] C. M. Bishop, et al., Pattern recognition and machine learning, volume 4, springer New York, 2006.

[23] D. Wackerly, W. Mendenhall, R. Scheaffer, Mathematical statistics with applications, Cengage Learning, 2007.

[24] J. a. Mendes-Moreira, C. Soares, A. M. Jorge, J. F. D. Sousa, Ensemble approaches for regression: A survey, ACM Comput. Surv. 45 (2012) 10:1–10:40.

[25] L. Breiman, Bagging predictors, Machine learning 24 (1996) 123–140.

[26] P. Domingos, Why does bagging work? a bayesian account and its implications., in: SIGKDD, ACM, pp. 155–158.

[27] M. Kearns, Thoughts on hypothesis boosting, Unpublished results (1988).

[28] R. E. Schapire, The strength of weak learnability, Machine learning 5 (1990) 197–227.

[29] R. M. Fujimoto, Parallel discrete event simulation, Commun. ACM 33 (1990) 30–53.

[30] J. Misra, Distributed discrete-event simulation, ACM Comput. Surv. 18 (1986) 39–65.

[31] Celery: Distributed task queue, `http://www.celeryproject.org` (2014).

[32] Rabbitmq, `https://www.rabbitmq.com` (2014).

[33] J. Gray, Queues are databases, arXiv preprint cs/0701158 (2007).

[34] T. White, Hadoop: The definitive guide, Yahoo Press, 2010.

[35] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, Ieee, pp. 1–10.

[36] mongoDB Developers, Mongodb, http://www.mongodb.org/ (2013).

[37] I. Foster, C. Kesselman, Globus: A metacomputing infrastructure toolkit, International Journal of Supercomputer Applications 11 (1996) 115–128.

[38] M. Litzkow, M. Livny, M. Mutka, Condor-a hunter of idle workstations, in: Distributed Computing Systems, 1988., 8th International Conference on, pp. 104–111.

[39] F. Xu, M. H. Eres, D. J. Baker, S. J. Cox, Tools and support for deploying applications on the grid, in: Services Computing, 2004.(SCC 2004). Proceedings. 2004 IEEE International Conference on, IEEE, pp. 281–287.

[40] J. Wason, M. Molinari, Z. Jiao, S. J. Cox, Delivering data management for engineers on the grid, in: Euro-Par 2003 Parallel Processing, Springer, 2003, pp. 412–416.

[41] G. Xue, M. J. Fairman, S. J. Cox, Exploiting web technologies for grid architecture, in: Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on, IEEE, pp. 272–272.