Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics

Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica, *University of California, Berkeley*

https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

This paper is included in the Proceedings of the
13th USENIX Symposium on Networked Systems
Design and Implementation (NSDI '16).

March 16–18, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-29-4

Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics

Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, Ion Stoica
University of California, Berkeley

Abstract

Recent workload trends indicate rapid growth in the deployment of machine learning, genomics and scientific workloads on cloud computing infrastructure. However, efficiently running these applications on shared infrastructure is challenging and we find that choosing the right hardware configuration can significantly improve performance and cost. The key to address the above challenge is having the ability to predict performance of applications under various resource configurations so that we can automatically choose the optimal configuration.

Our insight is that a number of jobs have predictable structure in terms of computation and communication. Thus we can build performance models based on the behavior of the job on small samples of data and then predict its performance on larger datasets and cluster sizes. To minimize the time and resources spent in building a model, we use optimal experiment design, a statistical technique that allows us to collect as few training points as required. We have built Ernest, a performance prediction framework for large scale analytics and our evaluation on Amazon EC2 using several workloads shows that our prediction error is low while having a training overhead of less than 5% for long-running jobs.

1 Introduction

In the past decade we have seen a rapid growth of large-scale *advanced* analytics that implement complex algorithms in areas like distributed natural language processing [24, 74], deep learning for image recognition [34], genome analysis [72, 61], astronomy [17] and particle accelerator data processing [19]. These applications differ from traditional analytics workloads (e.g., SQL queries) in that they are not only data-intensive but also computation-intensive, and typically run for a long time (and hence are expensive). Along with new workloads, we have seen widespread adoption of cloud computing with large data sets being hosted [7, 1], and the emergence of sophisticated analytics services, such as machine learning, being offered by cloud providers [9, 6].

With cloud computing environments such as Amazon EC2, users typically have a large number of choices in terms of the instance types and number of instances they can run their jobs on. Not surprisingly, the amount of memory per core, storage media, and the number of instances are crucial choices that determine the running time and thus indirectly the cost of running a given job. Using common machine learning kernels we show in §2.2 that choosing the right configuration can improve performance by up to 1.9x at the same cost.

In this paper, we address the challenge of choosing the configuration to run large advanced analytics applications in heterogeneous multi-tenant environments. The choice of configuration depends on the user's goals which typically includes either minimizing the running time given a budget or meeting a deadline while minimizing the cost. The key to address this challenge is developing a performance prediction framework that can accurately predict the running time on a specified hardware configuration, given a job and its input.

One approach to address this challenge is to predict the performance of a job based on monitoring the job's previous runs [39, 44]. While simple, this approach assumes the job runs repeatedly on the same or "similar" data sets. However, this assumption does not always hold. First, even when a job runs periodically it typically runs on data sets that can be widely different in both size and content. For example, a prediction algorithm may run on data sets corresponding to different days or time granularities. Second, workloads such as interactive machine learning [9, 55] and parameter tuning generate unique jobs for which we have little or no relevant history. Another approach to predict job performance is to build a detailed parametric model for the job. Along these lines, several techniques have been recently proposed in the context of MapReduce-like frameworks [77, 52]. These techniques have been aided by the inherent simplicity of the two-stage MapReduce model. However, the recent increase in the popularity of more complex parallel computation engines such as Dryad [51] and Spark [83] make these parametric techniques much more difficult to apply.

In this paper, we propose a new approach that can accurately predict the performance of a given analytics job. The main idea is to run a set of instances of the entire job on samples of the input, and use the data from these training runs to create a performance model. This approach has low overhead, as in general it takes much less time and resources to run the training jobs than running the job itself. Despite the fact that this is a black-box approach (i.e., requires no knowledge about the internals of

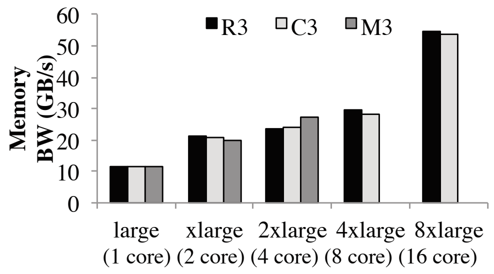

Figure 1: **Comparison of memory bandwidths across Amazon EC2 `m3`/`c3`/`r3` instance types. There are only three sizes for `m3`. Smaller instances (large, xlarge) have better memory bandwidth per core.**

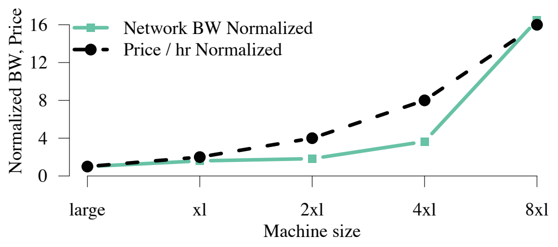

Figure 2: **Comparison of network bandwidths with prices across different EC2 `r3` instance sizes normalized to `r3.large`. `r3.8xlarge` has the highest bandwidth per core.**

the job), it works surprisingly well in practice.

The reason this approach works so well is because many advanced analytics workloads have a simple structure and the dependence between their running times and the input sizes or number of nodes is in general characterized by a relatively small number of smooth functions. This should come as no surprise as big data has naturally lead researchers and practitioners to develop algorithms that are linear [22] or quasi-linear in terms of the input size, and which scale well with the number of nodes. As a simple example, consider a mini-batch gradient descent algorithm used for linear regression. For a dataset with m data points and n features per partition, the time taken by each task to compute the gradient is uniform (mn) and similarly the size of output from every task is the same, a vector of size n.

The cost and utility of training data points collected is important for low-overhead prediction and we address this problem using *optimal experiment design* [63] , a statistical technique that allows us to select the most useful data points for training. We augment experiment design with a cost model and this helps us find the training data points to explore within a given budget. We have built support for the above techniques in Ernest and we find that a number of advanced analytics workloads can be accurately modeled using simple features that reflect commonly found computation and communication patterns. We include a cross-validation based verification scheme in Ernest to detect when a workload does not match the features being used and show how we can easily extend our model in such cases.

Using Amazon EC2 as our execution environment, we evaluate the accuracy of our system using a number of workloads including (a) several machine learning algorithms that are part of Spark MLlib [56], (b) queries from GenBase [73] and I/O intensive transformations using ADAM [61] on a full genome, and (c) a speech recognition pipeline that achieves state-of-the-art results [50]. Our evaluation shows that our average prediction error is under 20% and that this is sufficient for choosing the appropriate number or type of instances. Our training overhead for long-running jobs is less than 5% and we also find that using experiment design improves prediction error for some algorithms by $30-50\%$ over a cost-based scheme. Finally, using our predictions we show that for a long-running speech recognition pipeline, finding the appropriate number of instances can reduce cost by around $4x$ compared to a greedy allocation scheme. In summary, the main contributions of this paper are:

- We propose Ernest, a performance prediction framework that works with unmodified jobs and achieves low overhead using optimal experiment design.
- We show how Ernest can detect when a model isn't appropriate and how small extensions can be used to model complex workloads.
- Using experiments on EC2, we show that Ernest is accurate for a variety of algorithms, input sizes, and cluster sizes.

2 Background

In this section we first present an overview of different approaches to performance prediction. We then discuss recent hardware and workload trends for large scale data analysis. We also present an example of an end-to-end machine learning pipeline and discuss some of the computation and communication patterns that we see using this example.

2.1 Performance Prediction

Performance modeling and prediction have been used in many different contexts in various systems [59, 16, 39]. At a high level performance modeling and prediction proceeds as follows: select an output or response variable that needs to be predicted and the features to be used for prediction. Next, choose a relationship or a model that can provide a prediction for the output variable given the input features. This model could be rule based [27, 21] or use machine learning techniques [60, 80] that build an estimator using some training data. We focus on machine learning based techniques in this paper and we next discuss two major approaches in modeling that influences the training data and machine learning algorithms used.

Performance counters: Performance counter based approaches typically use a large number of low level counters to try and predict application performance characteristics. Such an approach has been used with CPU counter for profiling [14], performance diagnosis [81, 25] and virtual machine allocation [60]. A similar approach has also been used for analytics jobs where the MapReduce counters have been used for performance prediction [77] and straggler mitigation [80]. Performance-counter based approaches typically use advanced learning algorithms like random forests, SVMs. However as they use a large number of features, they require large amounts of training data and are well suited for scenarios where historical data is available.

System modeling: In the system modeling approach, a performance model is developed based on the properties of the system being studied. This method has been used in scientific computing [16] for compilers [11], programming models [21, 27]; and by databases [29, 57] for estimating the progress made by SQL queries. System design based models are usually simple and interpretable but may not capture all the execution scenarios. However one advantage of this approach is that only a small amount of training data is required to make predictions.

In this paper, we look at how to perform efficient performance prediction for large scale advanced analytics. We use a system modeling approach where we build a high-level end-to-end model for advanced analytics jobs. As collecting training data can be expensive, we further focus on how to minimize the amount of training data required in this setting. We next survey recent hardware and workload trends that motivate this problem.

2.2 Hardware Trends

The widespread adoption of cloud computing has led to a large number of data analysis jobs being run on cloud computing platforms like Amazon EC2, Microsoft Azure and Google Compute Engine. In fact, a recent survey by Typesafe of around 500 enterprises [4] shows that 53% of Apache Spark users deploy their code on Amazon EC2. However using cloud computing instances comes with its own set of challenges. As cloud computing providers use virtual machines for isolation between users, there are a number of fixed-size virtual machine options that users can choose from. Instance types vary not only in capacity (i.e. memory size, number of cores etc.) but also in performance. For example, we measured memory bandwidth and network bandwidth across a number of instance types on Amazon EC2. From Figure 1 we can see that the smaller instances i.e. `large` or `xlarge` have the highest memory bandwidth available per core while Figure 2 shows that `8xlarge` instances have the highest network bandwidth available per core. Based on our experiences with Amazon EC2, we believe these performance variations are not necessarily due to poor isolation between tenants but are instead related to how various instance types are mapped to shared physical hardware.

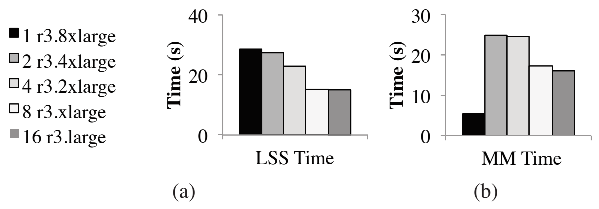

Figure 3: **Performance comparison of a Least Squares Solver (LSS) job and Matrix Multiply (MM) across similar capacity configurations.**

The non-linear relationship between price vs. performance is not only reflected in micro-benchmarks but can also have a significant effect on end-to-end performance. For example, we use two machine learning kernels: (a) A least squares solver used in convex optimization [37] and (b) a matrix multiply operation [75], and measure their performance for similar capacity configurations across a number of instance types. The results (Figure 3(a)) show that picking the right instance type can improve performance by up to 1.9x at the same cost for the least squares solver. Earlier studies [47, 79] have also reported such performance variations for other applications like SQL queries, key-value stores. These performance variations motivate the need for a performance prediction framework that can automate the choice of hardware for a given computation.

Finally, performance prediction is important not just in cloud computing but it is also useful in other shared computing scenarios like private clusters. Cluster schedulers [15] typically try to maximize utilization by packing many jobs on a single machine and predicting the amount of memory or number of CPU cores required for a computation can improve utilization [36]. Next, we look at workload trends in large scale data analysis and how we can exploit workload characteristics for performance prediction.

2.3 Workload trends

The last few years have seen the growth of advanced analytics workloads like machine learning, graph processing and scientific analyses on large datasets. Advanced analytics workloads are commonly implemented on top of data processing frameworks like Hadoop [35], Naiad [58] or Spark [83] and a number of high level libraries for machine learning [56, 2] have been developed on top of these frameworks. A survey [4] of Apache Spark users shows that around 59% of them use the machine learning library in Spark and recently launched services like Azure ML [9] provide high level APIs which implement commonly used machine learning algorithms.

Advanced analytics workloads differ from other workloads like SQL queries or stream processing in a number of ways. These workloads are typically numerically

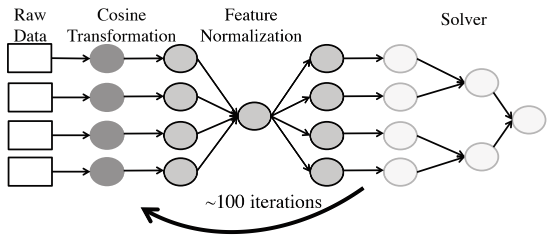

Figure 4: **Execution DAG of a machine learning pipeline used for speech recognition [50]. The pipeline consists of featurization and model building steps which are repeated for many iterations.**

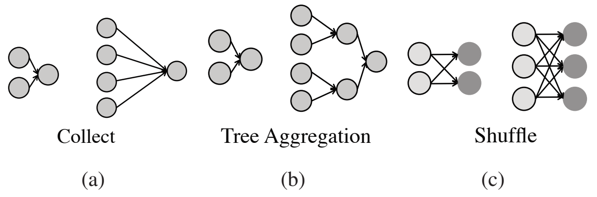

Figure 5: **Scaling behaviors of commonly found communication patterns as we increase the number of machines.**

intensive, i.e. performing floating point operations like matrix-vector multiplication or convolutions [32], and thus are sensitive to the number of cores and memory bandwidth available. Further, such workloads are also often iterative and repeatedly perform parallel operations on data cached in memory across a cluster. Advanced analytics jobs can also be long-running: for example, to obtain the state-of-the-art accuracy on tasks like image recognition [34] and speech recognition [50], jobs are run for many hours or days.

Example: Speech Recognition Pipeline. As an example of an advanced analytics job, we consider a speech recognition pipeline [50] that achieves state-of-the-art accuracy on the TIMIT [26] dataset. The pipeline trains a model using kernel SVMs and the execution DAG is shown in Figure 4. From the figure we can see that such pipelines consist of a number of stages, each of which may be repeated for some iterations. This TIMIT pipeline contains three main stages. The first stage of the DAG reads input data, and featurizes the data by applying a random cosine transformation [64] to each record. Assuming dense input data and equal-sized partitions, we can also see that each task in the first stage will take a similar amount of time to compute. Further, we observe that, unlike SQL queries with selectivity clauses, the transformation here results in the same amount of output data per input record across all tasks. The second stage in the pipeline normalizes data, which requires computing the mean and variance of the features by aggregating values across all the partitions. In the last stage, the normalized features are fed into a convex solver [23] to build a model. The model is then refined by generating more features and these steps are repeated for 100 iterations to achieve state-of-the-art accuracy.

Workload Properties: Since advanced analytics jobs run on large datasets are expensive, we observe that developers have focused on algorithms that are scalable across machines and are of low complexity (e.g., linear or quasi-linear) [22]. Otherwise, using these algorithms to process huge amounts of data might be infeasible. The natural outcome of these efforts is that these workloads admit relatively simple performance models. Specifically, we find that the computation required per data item remains the same as we scale the computation.

Further, we observe that only a few communication patterns repeatedly occur in such jobs. These patterns (Figure 5) include (a) the all-to-one or *collect* pattern, where data from all the partitions is sent to one machine, (b) *tree-aggregation* pattern where data is aggregated using a tree-like structure, and (c) a *shuffle* pattern where data goes from many source machines to many destinations. These patterns are not specific to advanced analytics jobs and have been studied before [30, 20]. Having a handful of such patterns means that we can try to automatically infer how the communication costs change as we increase the scale of computation. For example, assuming that data grows as we add more machines (i.e., the data per machine is constant), the time taken for the collect increases as $O(machines)$ as a single machine needs to receive all the data. Similarly the time taken for a binary aggregation tree grows as $O(log(machines))$.

Finally we observe that many algorithms are iterative in nature and that we can also *sample the computation* by running just a few iterations of the algorithm. Next we will look at the design of the performance model.

3 Modeling Advanced Analytics Jobs

In this section we outline a model for predicting execution time of advanced analytics jobs. This scheme only uses end-to-end running times collected from executing the job on smaller samples of the input and we discuss techniques for model building and data collection.

At a high level we consider a scenario where a user provides as input a parallel job (written using any existing data processing framework) and a pointer to the input data for the job. We do not assume the presence of any historical logs about the job and our goal here is to build a model that will predict the execution time for any input size, number of machines for this given job. The main steps in building a predictive model are (a) determining what training data points to collect (b) determining what features should be derived from the training data and (c) performing feature selection to pick the simplest model that best fits the data. We discuss all three aspects below.

3.1 Features for Prediction

One of the consequences of modeling end-to-end unmodified jobs is that there are only a few parameters that we can change to observe changes in performance. Assuming that the job, the dataset and the machine types are fixed, the two main features that we have are (a) the number of rows or fraction of data used (scale) and (b) the number of machines used for execution. Our goal in the modeling process is to derive as few features as required for the amount of training data required grows linearly with the number of features.

To build our model we add terms related to the computation and communication patterns discussed in §2.3. The terms we add to our linear model are (a) a fixed cost term which represents the amount of time spent in serial computation (b) the interaction between the scale and the inverse of the number of machines; this is to capture the *parallel* computation time for algorithms whose computation scales *linearly* with data, i.e., if we double the size of the data with the same number of machines, the computation time will grow linearly (c) a $log(machines)$ term to model communication patterns like aggregation trees (d) a linear term $O(machines)$ which captures the all-to-one communication pattern and fixed overheads like scheduling / serializing tasks (i.e. overheads that scale as we add more machines to the system). Note that as we use a linear combination of *non-linear features*, we can model non-linear behavior as well.

Thus the overall model we are fitting tries to learn values for $\theta_0, \theta_1, \theta_2$, and θ_3 in the formula

$$\begin{aligned} time = \theta_0 + \theta_1 \times (scale \times \frac{1}{machines}) + \\ \theta_2 \times \log(machines) + \\ \theta_3 \times machines \end{aligned} \tag{1}$$

Given these features, we then use a *non-negative least squares* (NNLS) solver to find the model that best fits the training data. NNLS fits our use case very well as it ensures that each term contributes some non-negative amount to the overall time taken. This avoids over-fitting and also avoids corner cases where say the running time could become negative as we increase the number of machines. NNLS is also useful for feature selection as it sets coefficients which are not relevant to a particular job to zero. For example, we trained a NNLS model using 7 data points on all of the machine learning algorithms that are a part of MLlib in Apache Spark 1.2. The final model parameters are shown in Table 1. From the table we can see two main characteristics: (a) that not all features are used by every algorithm and (b) that the contribution of each term differs for each algorithm. These results also show why we cannot reuse models across jobs.

Additional Features: While the features used above capture most of the patterns that we see in jobs, there could other patterns which are not covered. For example in linear algebra operators like QR decomposition the computation time will grow as $scale^2/machines$ if we scale the number of columns. We discuss techniques to detect when the model needs such additional terms in §3.4.

Benchmark	$intercept$	$scale/mc$	mc	$log(mc)$
spearman	0.00	4887.10	0.00	4.14
classification	0.80	211.18	0.01	0.90
pca	6.86	208.44	0.02	0.00
naive.bayes	0.00	307.48	0.00	1.00
summary stats	0.42	39.02	0.00	0.07
regression	0.64	630.93	0.09	1.50
als	28.62	3361.89	0.00	0.00
kmeans	0.00	149.58	0.05	0.54

Table 1: **Models built by Non-Negative Least Squares for MLlib algorithms using `r3.xlarge` instances. Not all features are used by every algorithm.**

3.2 Data collection

The next step is to collect training data points for building a predictive model. For this we use the input data provided by the user and run the *complete* job on small samples of the data and collect the time taken for the job to execute. For iterative jobs we allow Ernest to be configured to run a certain number of iterations (§4). As we are not concerned with the accuracy of the computation we just use the first few rows of the input data to get appropriately sized inputs.

How much training data do we need?: One of the main challenges in predictive modeling is minimizing the time spent on collecting training data while achieving good enough accuracy. As with most machine learning tasks, collecting more data points will help us build a better model but there is time and a cost associated with collecting training data. As an example, consider the model shown in Table 1 for *kmeans*. To train this model we used 7 data points and we look at the importance of collecting additional data by comparing two schemes: in the first scheme we collect data in an increasing order of machines and in the second scheme we use a mixed strategy as shown in Figure 6. From the figure we make two important observations: (a) in this case, the mixed strategy gets to a lower error quickly; after three data points we get to less than 15% error. (b) We see a trend of diminishing returns where adding more data points does not improve accuracy by much. We next look at techniques that will help us find how much training data is required and what those data points should be.

3.3 Optimal Experiment Design

To improve the time taken for training without sacrificing the prediction accuracy, we outline a scheme based on *optimal experiment design*, a statistical technique that can be used to minimize the number of experiment runs required. In statistics, experiment design [63] refers to the study of how to collect data required for any experiment given the modeling task at hand. *Optimal* experiment design specifically looks at how to choose experiments that are optimal with respect to some statistical criterion. At a high-level the goal of experiment design is to determine data points that can give us most information to build an accurate model.

More formally, consider a problem where we are trying to fit a linear model X given measurements $y_1, \ldots, y_m$ and features $a_1, \ldots, a_m$ for each measurement. Each feature vector could in turn consist of a number of dimensions (say n dimensions). In the case of a linear model we typically estimate X using linear regression. We denote this estimate as $\hat{X}$ and $\hat{X} - X$ is the estimation error or a measure of how far our model is from the true model.

To measure estimation error we can compute the Mean Squared Error (MSE) which takes into account both the bias and the variance of the estimator. In the case of the linear model above if we have m data points each having n features, then the variance of the estimator is represented by the $n \times n$ covariance matrix $(\sum_{i=1}^{m} a_i a_i^T)^{-1}$. The key point to note here is that the covariance matrix only depends on the feature vectors that were used for this experiment and not on the model that we are estimating.

In optimal experiment design we choose feature vectors (i.e. a_i) that minimize the estimation error. Thus we can frame this as an optimization problem where we minimize the estimation error subject to constraints on the number of experiments. More formally we can set λ_i as the fraction of times an experiment is chosen and minimize the trace of the inverse of the covariance matrix:

$$\begin{aligned} \text{Minimize} \quad & \mathbf{tr}((\sum_{i=1}^{m} \lambda_i a_i a_i^T)^{-1}) \\ \text{subject to} \quad & \lambda_i \geq 0, \lambda_i \leq 1 \end{aligned}$$

Using Experiment Design: The predictive model described in the previous section can be formulated as an experiment design problem. Given bounds for the scale and number of machines we want to explore, we can come up with all the features that can be used. For example if the scale bounds range from say 1% to 10% of the data and the number of machine we can use ranges from 1 to 5, we can enumerate 50 different feature vectors from all the scale and machine values possible. We can then feed these feature vectors into the experiment design setup described above and only choose to run those experiments whose λ values are non-zero.

Accounting for Cost: One additional factor we need to consider in using experiment design is that each experiment we run costs a different amount. This cost could be in terms of time (i.e. it is more expensive to train with larger fraction of the input) or in terms of machines (i.e. there is a fixed cost to say launching a machine). To account for the cost of an experiment we can augment the optimization problem we setup above with an additional

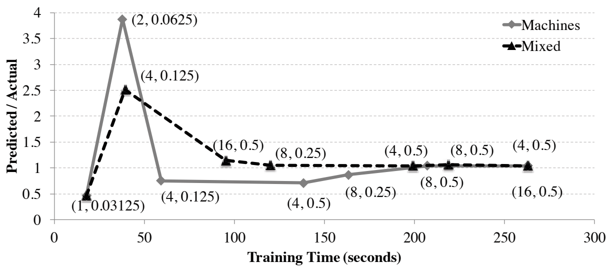

Figure 6: **Comparison of different strategies used to collect training data points for KMeans. The labels next to the data points show the (number of machines, scale factor) used.**

	Residual Sum of Squares	Percentage Err Median	Percentage Err Max
without $\sqrt{n}$	1409.11	12.2%	64.9%
with $\sqrt{n}$	463.32	5.7%	26.5%

Table 2: **Cross validation metrics comparing different models for Sparse GLM run on the splice-site dataset.**

constraint that the total cost should be lesser than some budget. That is if we have a cost function which gives us a cost c_i for an experiment with scale s_i and m_i machines, we add a constraint to our solver that $\sum_{i=1}^{m} c_i \lambda_i \leq B$ where B is the total budget. For the rest of this paper we use the time taken to collect training data as the cost and ignore any machine setup costs as we usually amortize that over all the data we need to collect. However we can plug-in in any user-defined cost function in our framework.

3.4 Model extensions

The model outlined in the previous section accounts for the most common patterns we see in advanced analytics applications. However there are some complex applications like randomized linear algebra [43] which might not fit this model. For such scenarios we discuss two steps: the first is adding support in Ernest to detect when the model is not adequate and the second is to easily allow users to extend the model being used.

Cross-Validation: The most common technique for testing if a model is valid is to use hypothesis testing and compute test statistics (e.g., using the t-test or the chi-squared test) and confirm the null hypothesis that data belongs to the distribution that the model describes. However as we use non-negative least squares (NNLS) the residual errors are not normally distributed and simple techniques for computing confidence limits, p-values are not applicable. Thus we use *cross-validation*, where subsets of the training data can be used to check if the model will generalize well. There are a number of methods to do cross-validation and as our training data size is small, we use a leave-one-out-cross-validation scheme in Ernest. Specifically if we have collected m training data points, we perform m cross-validation runs where each run uses $m-1$ points as training data and tests the model on the left out data point.

Model extension example: As an example, we consider the GLM classification implementation in Spark MLLib for sparse datasets. In this workload the computation is linear but the aggregation uses two stages (instead of an aggregation tree) where the first aggregation stage has $\sqrt{n}$ tasks for n partitions of data and the second aggregation stage combines the output of $\sqrt{n}$ tasks using one task. This communication pattern is not captured in our model from earlier and the results from cross validation using our original model are shown in Table 2. As we can see in the table both the residual sum of squares and the percentage error in prediction are high for the original model. Extending the model in Ernest with additional terms is simple and in this case we can see that adding the $\sqrt{n}$ term makes the model fit much better. In practice we use a configurable threshold on the percentage error to determine if the model fit is poor. We investigate the end-to-end effects of using a better model in §6.6.

4 Implementation

Ernest is implemented using Python as multiple modules. The modules include a job submission tool that submits training jobs, a training data selection process which implements experiment design using a CVX solver [42, 41] and finally a model builder that uses NNLS from SciPy [53]. Even for a large range of scale and machine values we find that building a model takes only a few seconds and does not add any overhead. In the rest of this section we discuss the job submission tool and how we handle sparse datasets, stragglers.

4.1 Job Submission Tool

Ernest extends existing job submission API [5] that is present in Apache Spark 1.2. This job submission API is similar to Hadoop's Job API [10] and similar job submission APIs exist for dedicated clusters [65, 78] as well. The job submission API already takes in the binary that needs to run (a JAR file in the case of Spark) and the input specification required for collecting training data.

We add a number of optional parameters which can be used to configure Ernest. Users can configure the minimum and maximum dataset size that will be used for training. Similarly the maximum number of machines to be used for training can also be configured. Our prototype implementation of Ernest uses Amazon EC2 and we amortize cluster launch overheads across multiple training runs i.e., if we want to train using 1, 2, 4 and 8 machines, we launch a 8 machine cluster and then run all of these training jobs in parallel.

The model built using Ernest can be used in a number of ways. In this paper we focus on a cloud computing use case where we can choose the number and type of EC2 instances to use for a given application. To do this we build one model per instance type and explore different sized instances (i.e. r3.large,...r3.8xlarge). After training the models we can answer higher level questions like selecting the cheapest configuration given a time bound or picking the fastest configuration given a budget. One of the challenges in translating the performance prediction into a higher-level decision is that the predictions could have some error associated with them. To help with this, we provide the cross validation results (§3.4) along with the prediction and these can be used to compute the range of errors observed on training data. Additionally we plan to provide support for visualizing the scaling behavior and Figure 20 in §6.6 shows an example.

4.2 Handling Sparse Datasets

One of the challenges in Ernest is to deal with algorithms that process sparse datasets. Because of the difference in sparsity across data items, each record could take different time to process. We observe that operations on sparse datasets depend on the number of non-zero entries and thus if we can sample the data such that we use a *representative* sparse subset during training, we should be able to apply modeling techniques described before. However in practice, we don't see this problem as even if there is a huge skew in sparsity across rows, the skew across partitions is typically smaller.

To illustrate, we chose three of the largest sparse datasets that are part of the LibSVM repository [70, 82] and we measured the maximum number of non-zero entries present in every partition after loading the data into HDFS. We normalize these values across partitions and a CDF of partition densities is shown in Figure 7. We observe the the difference in sparsity between the most loaded partition and the least loaded one is less than 35% for all datasets and thus picking a random sample of partitions [76] is sufficient to model computation costs.

4.3 Straggler mitigation by over-allocation

The problem of dealing with stragglers, or tasks which take much longer than other tasks is one of the main challenges in large scale data analytics [80, 13, 33]. Using cloud computing instances could further aggravate the problem due to differences in performance across instances. One technique that we use in Ernest to overcome variation among instances is to launch a small percentage of extra instances and then discard the worst performing among them before running the user's job. We use memory bandwidth and network bandwidth measurements (§2) to determine the slowest instances.

In our experiences with Amazon EC2 we find that even having a few extra instances can be more than sufficient in eliminating the slowest machines. To demonstrate this, we set the target cluster size as $N = 50$ `r3.2xlarge` instances and have Ernest automatically allocate a small percentage of extra nodes. We then run

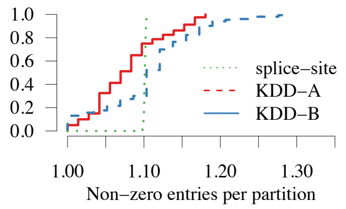

Figure 7: **CDF of maximum number of non-zero entries in a partition, normalized to the least loaded partition for sparse datasets.**

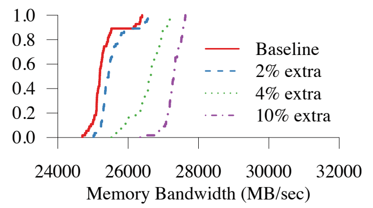

Figure 8: **CDFs of STREAM memory bandwidths under four allocation strategies. Using a small percentage of extra instances removes stragglers.**

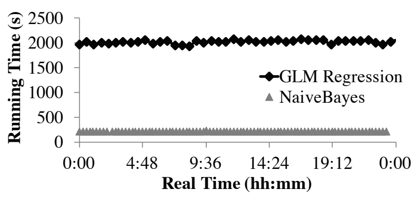

Figure 9: **Running times of GLM and Naive Bayes over a 24-hour time window on a 64-node EC2 cluster.**

STREAM [54] at 30 second intervals and collect memory bandwidth measurements on all instances. Based on the memory bandwidths observed, we eliminate the slowest nodes from the cluster. Figure 8 shows for each allocation strategy, the CDF of the memory bandwidth obtained when picking the best *N* instances from all the instances allocated. We see that Ernest only needs to allocate as few as 2 (or 4%) extra instances to eliminate the slowest stragglers and improve the target cluster's average memory bandwidth from 24.7 GB/s to 26 GB/s.

5 Discussion

In this section we look at *when* a model should be retrained and also discuss the trade-offs associated with including more fine-grained information in Ernest.

5.1 Model reuse

The model we build using Ernest predicts the performance for a given job for a specific dataset and a target cluster. One of the questions while using Ernest is to determine when we need to retrain the model. We consider three different circumstances here: changes in code, changes in cluster behavior and changes in data.

Code changes: If different jobs use the same dataset, the cluster and dataset remain the same, but the computation being run changes. As Ernest treats the job being run as a black-box, we will need to retrain the model for any changes to the code. This can be detected by computing hashes of the binary files.

Variation in Machine Performance: One of the concerns with using cloud computing based solutions like EC2 is that there could be performance variations over time even when a job is using the same instance types and number of instances. We investigated if this was an issue by running two machine learning jobs GLM regression and NaiveBayes repeatedly on a cluster of 64 `r3.xlarge` instances. The time taken per run of each algorithm over a 24 hour period is shown in Figure 9. We see that the variation over time is very small for both workloads and the standard deviation is less than 2% of the mean. Thus we believe that Ernest models should remain relevant across relatively long time periods.

Changes in datasets: As Ernest uses small samples of the data for training, the model is directly applicable as the dataset grows. When dealing with newly collected data, there are some aspects of the dataset like the number of data items per block and the number of features per data item that should remain the same for the performance properties to be similar. As some of these properties might be hard to measure, our goal is to make the model building overhead small so that Ernest can be rerun for newly collected datasets.

5.2 Using Per-Task Timings

In the model described in the previous sections, we only measure the end-to-end running time of the whole job. Existing data processing frameworks already measure fine grained metrics [8, 3] and we considered integrating task-level metrics in Ernest. One major challenge we faced here is that in the BSP model a stage only completes when its last task completes. Thus rather than predicting the average task duration, we need to estimate the maximum task duration and this requires more complex non-parametric methods like Bootstrap [38]. Further, to handle cases where the number of tasks in a stage are greater than the number of cores available, we need adapt our estimate based on the number of waves [12] of tasks. We found that there were limited gains from incorporating task-level information given the additional complexity. While we continue to study ways to incorporate new features, we found that simple features used in predicting end-to-end completion time are more robust.

6 Evaluation

We evaluate how well Ernest works by using two metrics: the prediction accuracy and the overhead of training for long-running machine learning jobs. In experiments where we measure accuracy, or how close a prediction is to the actual job completion time, we use the ratio of the predicted job completion time to the actual job completion time $^{\text{Predicted Time}}/_{\text{Actual Time}}$ as our metric.

The main results from our evaluation are:

- Ernest's predictive model achieves less than 20% error on most of the workloads with less than 5% overhead for long running jobs.(§6.2)
- Using the predictions from Ernest we can get up to 4*x* improvement in price by choosing the optimal number of instances for the speech pipeline. (§6.3)

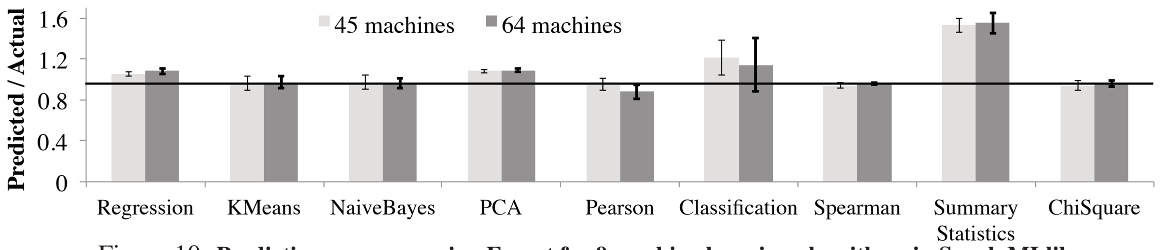

Figure 10: **Prediction accuracy using Ernest for 9 machine learning algorithms in Spark MLlib.**

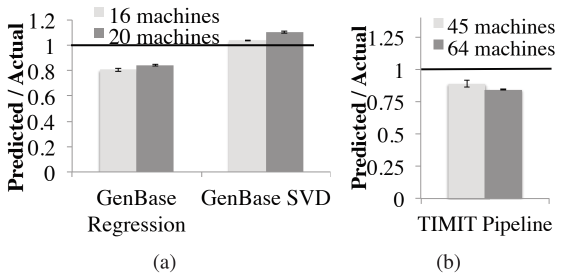

Figure 11: **Prediction accuracy for GenBase queries and TIMIT pipeline.**

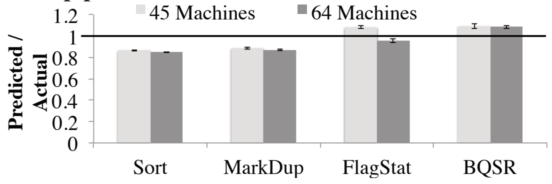

Figure 12: **Prediction accuracy for four transformations run using ADAM.**

- Given a training budget, experiment design improves accuracy by 30% – 50% for some workloads when compared to a cost-based approach. (§6.5)
- By extending the default model we are also able to accurately predict running times for sparse and randomized linear algebra operations. (§6.6)

6.1 Workloads and Experiment Setup

We use five workloads to evaluate Ernest. Our first workload consists of 9 machine learning algorithms that are part of MLlib [56]. For algorithms designed for dense inputs, the performance characteristics are independent of the data and we use synthetically generated data with 5 million examples. We use 10K features per data point for regression, classification, clustering and 1K features for the linear algebra and statistical benchmarks.

To evaluate Ernest on sparse data, we use `splice-site` and `kdda`, two of the largest sparse classification datasets that are part of LibSVM [28]. The `splice-site` dataset contains 10M data points with around 11M features and the `kdda` dataset contains around 6.7M data points with around 20M features. To see how well Ernest performs on end-to-end pipelines, we use GenBase, ADAM and a speech recognition pipeline (§2). We run regression and SVD queries from GenBase on the `Large` dataset [40] (30K genes

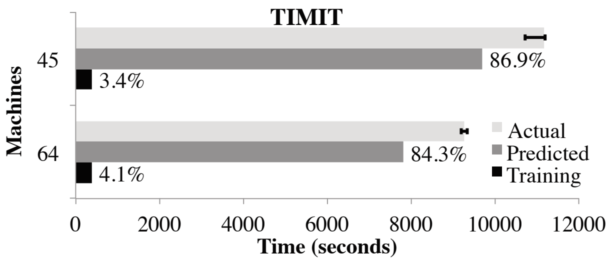

Figure 13: **Training times vs. accuracy for TIMIT pipeline running 50 iterations. Percentages with respect to actual running times are shown.**

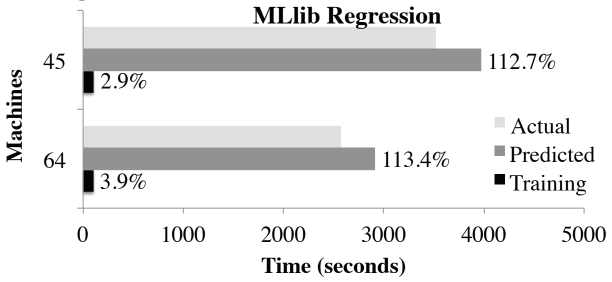

Figure 14: **Training times vs. accuracy for MLlib Regression running 500 iterations. Percentages with respect to actual running times are shown.**

for 40K patients). For ADAM we use the high coverage NA12878 full genome from the 1000 Genomes project [1] and run four transformations: sorting, marking duplicate reads, base quality score recalibration and quality validation. The speech recognition pipeline is run on the TIMIT [50] dataset using an implementation from KeystoneML [71]. All datasets other than the one for ADAM are cached in memory before the experiments begin and we do warmup runs to trigger the JVM's just-in-time compilation. We use `r3.xlarge` machines from Amazon EC2 (each with 4 vCPUs and 30.5GB memory) unless otherwise specified. Our experiments were run with Apache Spark 1.2. Finally all our predictions were compared against at least three actual runs and the values in our graphs show the average with error bars indicating the standard deviation.

6.2 Accuracy and Overheads

Prediction Accuracy: We first measure the prediction accuracy of Ernest using the nine algorithms from MLlib. In this experiment we configure Ernest to use between 1 and 16 machines for training and sample between 0.1% to 10% of the dataset. We then predict the performance for cases where the algorithms use the entire dataset on 45 and 64 machines. The prediction accuracies shown in Figure 10 indicate that Ernest's predictions are within 12% of the actual running time for most jobs. The two exceptions where the error is higher are the `summary statistics` and `glm-classification` job. In the case of `glm-classification`, we find that the training data and the actual runs have high variance (error bars in Figure 10 come from this) and that Ernest's prediction is within the variance of the collected data. In the case of summary statistics we have a short job where the absolute error is low: the actual running time is around 6 seconds while Ernest's prediction is around 8 seconds.

Next, we measure the prediction accuracy on GenBase and the TIMIT pipeline; the results are shown in Figure 11. Since the GenBase dataset is relatively small (less than 3GB in text files), we partition it into 40 splits, and restrict Ernest to use up to 6 nodes for training and predict the actual running times on 16 and 20 machines. As in the case of MLlib, we find the prediction errors to be below 20% for these workloads. Finally, the prediction accuracy for four transformations on ADAM show a similar trend and are shown in Figure 12. We note that the ADAM queries read input and write output to the *distributed filesystem* (HDFS) in these experiments and that these queries are also shuffle heavy. We find that Ernest is able to capture the I/O overheads and the reason for this is that the time to read / write a partition of data remains similar as we scale the computation.

Our goal in building Ernest is not to enforce strict SLOs but to enable low-overhead predictions that can be used to make coarse-grained decisions. We discuss how Ernest's prediction accuracy is sufficient for decisions like how many machines (§6.3) and what type of machines (§6.4) to use in the following sections.

Training Overheads: One of the main goals of Ernest is to provide performance prediction with low overhead. To measure the overhead in training we consider two long-running machine learning jobs: the TIMIT pipeline run for 50 iterations, and MLlib Regression with a mini-batch SGD solver run for 500 iterations. We configure Ernest to run 5% of the overall number of iterations during training and then linearly scale its prediction by the target number of iterations. Figures 13 and 14 show the times taken to train Ernest and the actual running times when run with 45 or 64 machines on the cluster. From the figures, we observe that for the regression problem the training time is below 4% of the actual running time and that Ernest's predictions are within 14%. For the TIMIT pipeline, the training overhead is less than 4.1% of the total running time. The low training overhead with these applications shows that Ernest efficiently handles long-running, iterative analytics jobs.

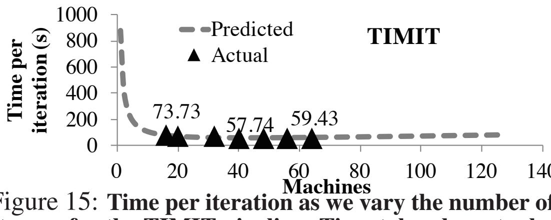

Figure 15: **Time per iteration as we vary the number of instances for the TIMIT pipeline. Time taken by actual runs are shown in the plot.**

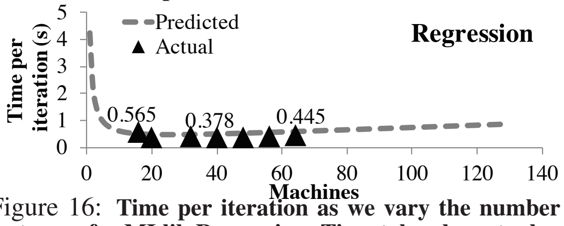

Figure 16: **Time per iteration as we vary the number of instances for MLlib Regression. Time taken by actual runs are shown in the plot.**

6.3 Choosing optimal number of instances

When users have a fixed-time or fixed-cost budget it is often tricky to figure out how many instances should be used for a job as the communication vs. computation trade-off is hard to determine for a given workload. In this section, we use Ernest's predictions to determine the optimum number of instances. We consider two workloads from the previous section: the TIMIT pipeline and GLM regression, but here we use subsets of the full data to focus on how the job completion time varies as we increase the number of machines to 64[1]. Using the same models trained in the previous section, we predict the time taken per iteration across a wide range of number of machines (Figures 15 and 16). We also show the actual running time to validate the predictions.

Consider a case where a user has a fixed-time budget of 1 hour (3600s) to say run 40 iterations of the TIMIT pipeline and an EC2 instance limit of 64 machines. Using Figure 15 and taking our error margin into account, Ernest is able to infer that launching 16 instances is sufficient to meet the deadline. Given that the cost of an `r3.xlarge` instance is $0.35/hour, a greedy strategy of using all the 64 machines would cost $22.4, while using the 16 machines as predicted by Ernest would only cost $5.6, a 4x difference. We also found that the 15% prediction error doesn't impact the decision as actual runs show that 15 machines is the optimum. Similarly, if the user has a budget of $15 then we can infer that using 40 machines would be faster than using 64 machines.

6.4 Choosing across instance types

We also apply Ernest to choose the optimal instance type for a particular workload; similar to the scenario above,

[1]We see similar scaling properties in the entire data, but we use a smaller dataset to highlight how Ernest can handle scenarios where the algorithm does not scale well.

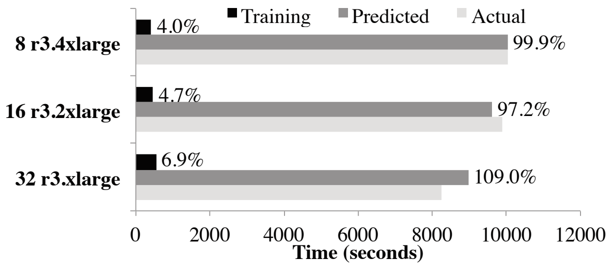

Figure 17: **Time taken for 50 iterations of the TIMIT workload across different instance types. Percentages with respect to actual running times are shown.**

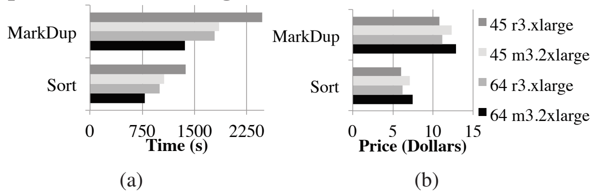

Figure 18: **Time taken for Sort and MarkDup workloads on ADAM across different instance types.**

we can optimize for cost given a deadline or optimize for performance given a budget. As an example of the benefits of choosing the right instance type, we re-run the TIMIT workload on three instance types (`r3.xlarge`, `r3.2xlarge` and `r3.4xlarge`) and we build a model for each instance type. With these three models, Ernest predicts the expected performance on *same-cost* configurations, and then picks the cheapest one. Our results (Figure 17) show that choosing the smaller `r3.xlarge` instances would actually be 1.2*x* faster than using the `r3.4xlarge` instances, while *incurring the same cost*. Similar to the previous section, the prediction error does not affect our decision here and Ernest's predictions choose the appropriate instance type.

We next look at how choosing the right instance type affects the performance of ADAM workloads that read and write data from disk. We compare `m3.2xlarge` instances that have two SSDs but cost $0.532 per hour and `r3.xlarge` instances that have one SSD and cost $0.35 an hour[2]. Results from using Ernest on 45 and 64 machines with these instance types is shown in Figure 18. From the we can see that using `m3.2xlarge` instances leads to better performance and that similar to the memory bandwidth analysis (§2.2) there are non-linear price-performance trade-offs. For example, we see that for the mark duplicates query, using 64 `m3.2xlarge` instances provides a 45% performance improvement over 45 `r3.xlarge` instances while only costing 20% more.

6.5 Experiment Design vs. Cost-based

We next evaluate the benefits of using optimal experiment design in Ernest. We compare experiment design to a greedy scheme where all the candidate training data

[2]Prices as of September 2015

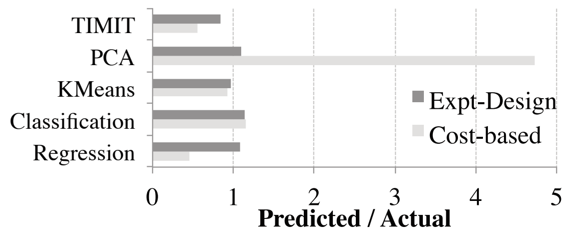

Figure 19: **Prediction accuracy when using Ernest vs. a cost-based approach for MLlib and TIMIT workloads.**

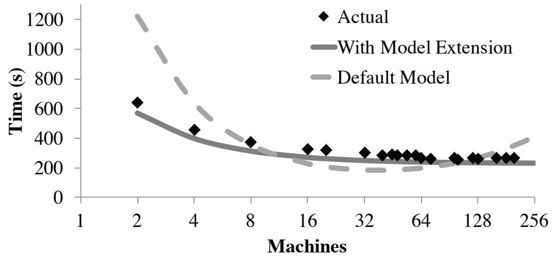

Figure 20: **Comparing KDDA models with and without extensions for different number of machines.**

points are sorted in increasing order of cost and we pick training points to match the cost of the points chosen in experiment design. We then train models using both configurations. A comparison of the prediction accuracy on MLlib and TIMIT workloads is shown in Figure 19.

From the figure, we note that for some workloads (e.g. KMeans) experiment design and the cost-based approach achieve similar prediction errors. However, for the Regression and TIMIT workloads, Ernest's experiment design models perform 30% – 50% better than the cost-based approach. The cost-based approach fails because when using just the cheapest training points, the training process is unable to observe how different stages of the job behave as scale and number of machines change. For example, in the case of TIMIT pipeline, the cost-based approach explores points along a weak scaling curve where *both* data size and number of machines increase, thus it is unable to model how the Solver stage scales when the amount of data is kept constant. Ernest's optimal experiment design mechanism successfully avoids this and chooses the most useful training points.

6.6 Model Extensions

We also measure the effectiveness of the model extensions proposed in §3.4 on two workloads: GLM classification run on sparse datasets (§4.2) and a randomized linear algebra workload that has non-linear computation time [43]. Figure 21 shows the prediction error for the default model and the error after the model is extended: with a $\sqrt{n}$ term for the Sparse GLM and a $\frac{n log^2 n}{mc}$ term which is the computation cost of the random projection. As we can see from the figure, using the appropriate model makes a significant difference in prediction error.

To get a better understanding of how different models can affect prediction error we use the KDDA dataset

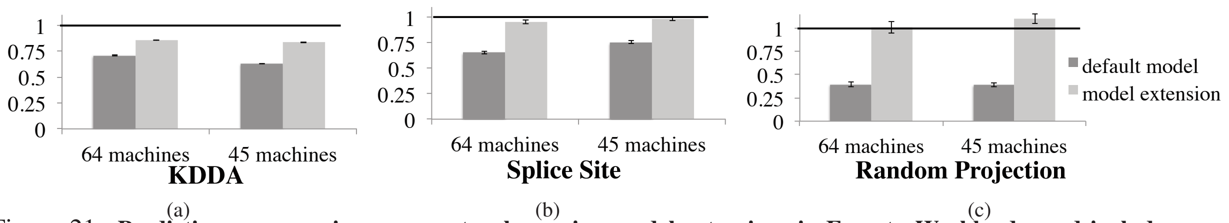

Figure 21: **Prediction accuracy improvements when using model extensions in Ernest. Workloads used include sparse GLM classification using KDDA, splice-site datasets and a random projection linear algebra job.**

and plot the predictions from both models as we scale from 2 to 200 machines (Figure 20). From the figure we can see that the extending the model with $\sqrt{n}$ ensures that the scaling behavior is captured accurately and that the default model can severely over-predict (at 2 machines and 200 machines) or under-predict (32 machines). Thus, while the default model in Ernest can capture a large number of workloads we can see that making simple model extensions can also help us accurately predict more complex workloads.

7 Related work

Performance Prediction: There have been a number of recent efforts at modeling job performance in datacenters to support SLOs or deadlines. Techniques proposed in Jockey [39] and ARIA [77] use historical traces and dynamically adjust resource allocations in order to meet deadlines. In Ernest we build a model with no historic information and try to minimize the amount of training data required. Bazaar [52] proposed techniques to model the network utilization of MapReduce jobs by using small subsets of data. In Ernest we capture computation and communication characteristics and use high level features that are framework independent. Projects like MRTuner [68] and Starfish [48] model MapReduce jobs at very fine granularity and set optimal values for options like memory buffer sizes etc. In Ernest we use few simple features and focus on collecting training data will help us maximize their utility. Finally scheduling frameworks like Quasar [36] try to estimate the scale out and scale up factor for jobs using the progress rate of the first few tasks. Ernest on the other hand runs the entire job on small datasets and is able to capture how different stages of a job interact in a long pipeline.

Query Optimization: Database query progress predictors [29, 57] solve a performance prediction problem similar to Ernest. Database systems typically use summary statistics [67] of the data like cardinality counts to guide this process. Further, these techniques are typically applied to a known set of relational operators. Similar ideas have also been applied to linear algebra operators [49]. In Ernest we use advanced analytics jobs where we know little about the data or the computation being run. Recent work has also looked at providing SLAs for OLTP [62] and OLAP workloads [46] in the cloud and some of our observations about variation across instance types in EC2 are also known to affect database queries.

Tuning, Benchmarking: Ideas related to experiment design, where we explore a space of possible inputs and choose the best inputs, have been used in other applications like server benchmarking [69]. Related techniques like Latin Hypercube Sampling have been used to efficiently explore file system design space [45]. Autotuning BLAS libraries [18] like ATLAS [31] also solve a similar problem of exploring a state space efficiently.

8 Future Work and Conclusion

In the future, we plan to study how statistical properties change in conjunction with the hardware. For example, in algorithms like HOGWILD! [66], the network latency between machines could affect the convergence rate. Further, based on our benchmarking experiments (§2) we see that there are a few key metrics which dictate the performance characteristics of a cluster. In the future we plan to study how we can integrate these metrics with the algorithm specific features used in Ernest.

In conclusion, the rapid adoption of advanced analytics workloads makes it important to consider how these applications can be deployed in a cost and resource-efficient fashion. In this paper, we studied the problem of performance prediction and show how simple models can capture computation and communication patterns. Using these models we have built Ernest, a performance prediction framework that intelligently chooses training points to provide accurate predictions with low overhead.

Acknowledgments

We would like to thank Ganesh Ananthanarayanan, Aurojit Panda, Kay Ousterhout, Evan Sparks, the NSDI reviewers and our shepherd Ranjita Bhagwan for their comments on earlier drafts of this paper. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, Guavus, HP, Huawei, Intel, Microsoft, Pivotal, Samsung, Schlumberger, Splunk, State Farm, Virdata and VMware.

References

[1] 1000 Genomes Project and AWS. http://aws.amazon.com/1000genomes/.

[2] Apache Mahout. http://mahout.apache.org/.

[3] Apache Spark: Monitoring and Instrumentation. http://spark.apache.org/docs/latest/monitoring.html.

[4] Apache spark, preparing for the next wave of reactive big data. http://goo.gl/FqEh94.

[5] Apache Spark: Submitting Applications. http://spark.apache.org/docs/latest/submitting-applications.html.

[6] Big data platform, hp haven. http://www8.hp.com/us/en/software-solutions/big-data-platform-haven/.

[7] Common crawl. http://commoncrawl.org.

[8] Hadoop History Server REST APIs. http://archive.cloudera.com/cdh4/cdh/4/hadoop/hadoop-yarn/hadoop-yarn-site/HistoryServerRest.html.

[9] Machine learning, microsoft azure. http://azure.microsoft.com/en-us/services/machine-learning/.

[10] MapReduce Tutorial. hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[11] V. S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D. A. Reed. An integrated compilation and performance analysis environment for data parallel programs. In *Supercomputing, 1995*.

[12] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *USENIX NSDI*, 2012.

[13] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *USENIX OSDI*, 2010.

[14] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390.

[15] Apache Hadoop NextGen MapReduce (YARN). Retrieved 9/24/2013, URL: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[16] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 213–223, Williamsburg, Virginia, USA.

[17] G. B. Berriman, G. Juve, E. Deelman, M. Regelson, and P. Plavchan. The application of cloud computing to astronomy: A study of cost and performance. In *Sixth IEEE International Conference on e-Science*, 2010.

[18] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Supercomputing 1997*, pages 340–347.

[19] I. Bird. Computing for the large hadron collider. *Annual Review of Nuclear and Particle Science*, 61:99–118, 2011.

[20] Blaise Barney. Message Passing Interface. https://computing.llnl.gov/tutorials/mpi/.

[21] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97.

[22] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, volume 20, pages 161–168. 2008.

[23] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

[24] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867, Prague, Czech Republic, June 2007.

[25] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing 2000*, Dallas, Texas, USA.

[26] J. P. Campbell Jr and D. A. Reynolds. Corpora for the evaluation of speaker recognition systems. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 829–832, 1999.

[27] B. L. Chamberlain, C. Lin, S.-E. Choi, L. Snyder, E. C. Lewis, and W. D. Weathersby. Zpl's wysiwyg performance model. In *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 50–61.

[28] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[29] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating progress of execution for SQL queries. In *SIGMOD 2004*, pages 803–814.

[30] M. Chowdhury and I. Stoica. Coflow: a networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36, 2012.

[31] R. Clint Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.

[32] A. Coates and A. Y. Ng. Learning feature representations with k-means. In *Neural Networks: Tricks of the Trade*, pages 561–580. Springer, 2012.

[33] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[34] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, pages 1232–1240, 2012.

[35] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1), 2008.

[36] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ASPLOS 2014*, pages 127–144.

[37] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[38] B. Efron. *The jackknife, the bootstrap and other resampling plans*, volume 38. SIAM, 1982.

[39] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Eurosys 2012*, pages 99–112.

[40] GenBase repository. `https://github.com/mitdbg/genbase`.

[41] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008. `http://stanford.edu/~boyd/graph_dcp.html`.

[42] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx`, Mar. 2014.

[43] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[44] B. He, M. Yang, Z. Guo, R. Chen, W. Lin, B. Su, H. Wang, and L. Zhou. Wave computing in the cloud. In *HotOS*, 2009.

[45] J. He, D. Nguyen, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Reducing file system tail latencies with chopper. In *FAST 2015*, pages 119–133, Santa Clara, CA.

[46] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *VLDB 2011*, 4(11):1111–1122.

[47] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SOCC 2011*.

[48] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[49] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD 2015*.

[50] P.-S. Huang, H. Avron, T. N. Sainath, V. Sindhwani, and B. Ramabhadran. Kernel methods match deep neural networks on timit. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, page 6, 2014.

[51] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Eurosys 2007*.

[52] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *SOCC 2012*.

[53] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001–.

[54] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[55] X. Meng, J. Bradley, E. Sparks, and S. Venkataraman. ML Pipelines: A New High-Level API for MLlib. https://goo.gl/pluhq0, 2015.

[56] Apache Spark MLLib. https://spark.apache.org/mllib/.

[57] K. Morton, M. Balazinska, and D. Grossman. Paratimer: A progress indicator for mapreduce dags. In *SIGMOD 2010*, pages 507–518, Indianapolis, Indiana, USA.

[58] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP 2013*, pages 439–455.

[59] D. Narayanan. *Operating System Support for Mobile Interactive Applications*. PhD thesis, CMU, 2002.

[60] O. Niehorster, A. Krieger, J. Simon, and A. Brinkmann. Autonomic resource management with support vector machines. In *International Conference on Grid Computing (GRID '11)*, pages 157–164, 2011.

[61] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. J. Franklin, A. D. Joseph, and D. A. Patterson. Rethinking data-intensive science using scalable analytics systems. In *SIGMOD 2015*, pages 631–646.

[62] L. Ortiz, V. de Almeida, and M. Balazinska. Changing the Face of Database Cloud Services with Personalized Service Level Agreements. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, 2015.

[63] F. Pukelsheim. *Optimal design of experiments*, volume 50. SIAM, 1993.

[64] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2007.

[65] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC 1998*.

[66] B. Recht, C. Re, S. Wright, and F. Niu. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[67] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, 1979.

[68] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang. MRTuner: A Toolkit to Enable Holistic Optimization for MapReduce Jobs. *VLDB 2014*, 7(13).

[69] P. Shivam, V. Marupadi, J. Chase, T. Subramaniam, and S. Babu. Cutting corners: workbench automation for server benchmarking. In *USENIX ATC 2008*, pages 241–254.

[70] S. Sonnenburg and V. Franc. Coffin: A computational framework for linear svms. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 999–1006, 2010.

[71] E. Sparks. Announcing KeystoneML. https://amplab.cs.berkeley.edu/announcing-keystoneml, 2015.

[72] L. D. Stein et al. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[73] R. Taft, M. Vartak, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker. Genbase: A complex analytics genomics benchmark. In *SIGMOD 2014*, pages 177–188.

[74] J. Uszkoreit, J. M. Ponte, A. C. Popat, and M. Dubiner. Large scale parallel document mining for machine translation. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 1101–1109. Association for Computational Linguistics, 2010.

[75] R. A. Van De Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4):255–274, 1997.

[76] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *OSDI 2014*, pages 301–316.

[77] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *ICAC 2011*, pages 235–244, Karlsruhe, Germany.

[78] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Eurosys 2015*.

[79] W. Wang, L. Xu, and I. Gupta. Scale up Vs. Scale out in Cloud Storage and Graph Processing System. In *Proceedings of the 2nd IEEE Workshop on Cloud Analytics*, 2015.

[80] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *SOCC 2014*.

[81] W. Yoo, K. Larson, L. Baugh, S. Kim, and R. H. Campbell. Adp: Automated diagnosis of performance pathologies using hardware events. In *International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2012*, pages 283–294, London, England, UK.

[82] H.-F. Yu, H.-Y. Lo, H.-P. Hsieh, J.-K. Lou, T. G. McKenzie, J.-W. Chou, P.-H. Chung, C.-H. Ho, C.-F. Chang, Y.-H. Wei, et al. Feature engineering and classifier ensemble for KDD Cup 2010. *KDD Cup 2010*.

[83] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI 2012*.