BlinkML: Approximate Machine Learning with Probabilistic Guarantees

Yongjoo Park Jingyi Qing Xiaoyang Shen Anthony Zheng
Jiamin Huang Barzan Mozafari
University of Michigan, Ann Arbor
{pyongjoo,jyqing,xyshen,zhengant,jiamin,mozafari}@umich.edu

ABSTRACT

As the volume of data grows, training a machine learning model is taking increasingly longer. The long training time becomes a serious bottleneck when humans are involved, e.g., interactive analytics, feature engineering, etc. Although we could much shorten this long training time by simply training an *approximate model* (using on a sample), typical ML systems do not guarantee the quality of the approximate model—its similarity to the *full model* (trained on the entire data). In fact, guaranteeing the quality of an approximate model is non-trivial since even if a same-sized sample is used, the approximate model's quality varies vastly depending on the types of ML tasks and the distribution of data.

In this paper, we present BlinkML, a system that trains a quality-guaranteed approximate model. Users can control the quality of BlinkML's model with two parameters, ε and δ. For classification, the approximate model's classification discrepancy (in comparison to the full model) is bounded by ε with probability at least $1-\delta$. For regression, the approximate mode's predictions differ less than ε compared to the full model with probability at least $1-\delta$. BlinkML supports a wide class of ML methods based on *maximum likelihood estimation*, which includes Generalized Linear Models and Probabilistic PCA. BlinkML's algorithms are implemented on top of Spark's distributed computations and scipy's optimization libraries. In our experiments, BlinkML saved 35.03%–99.69% training time of large-scale ML tasks at the cost of only 1% prediction discrepancy against full models.

PVLDB Reference Format:
Yongjoo Park, Jingyi Qing, Xiaoyang Shen, Anthony Zheng, Jiamin Huang, Barzan Mozafari. BlinkML: Approximate Machine Learning with Probabilistic Guarantees. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/TBD

1. INTRODUCTION

While data management systems have been much successful in supporting traditional OLAP-style analytics, they have *not* been quite as successful in attracting modern machine learning (ML) workloads. To circumvent this, most analytical database vendors have added integration layers for popular ML libraries in Python (e.g., Oracle's cx_Oracle [2], SQL Server's pymssql [5], and DB2's ibm_db [6]) or in R (e.g., Oracle's RODM [7], SQL Server's RevoScaleR [8], and DB2's ibmdbR [3]). These interfaces simply allow machine learning algorithms to run on the data *in-situ*.

However, recent efforts have shown that data management systems have much more to offer. For example, materialization and reuse opportunities [11, 12, 19, 51, 61], cost-based optimization of linear algebraic operators [15,18,28], array-based representations [33, 54], avoiding denormalization [37, 38, 50], lazy evaluation [64], declarative interfaces [45,53,57], and query planning [36,46,52] are all readily available (or at least familiar) database functionalities that can deliver significant speedups for various ML workloads.

One additional but key opportunity that has been largely overlooked is the *sampling abstraction* offered by nearly every database system. Sampling operators have been mostly used for approximate query processing (AQP) [16,20,22,26,40,41,47,48]. However, applying the lessons learned in the data management community regarding AQP, we could use a similar sampling abstraction to also speed up an important class of ML workloads.

In particular, ML is often a *human-in-the-loop* and *iterative* process. The analysts performs some initial data cleaning, feature engineering/selection, hyper-parameter tuning, and model selection. They then inspect the results and may repeat this process and invest more effort in some of these steps until they are satisfied with the model quality, say in terms of explanatory or predictive power. The entire process is therefore slow and computationally expensive. Much of the computational resources spent on early iterations are eventually wasted, as the eventual model can differ quite a bit from the initial ones (in terms of features, parameters or even model class). In fact, this is the reason why many practitioners use a small sample of their entire data during the initial steps of their analysis in order to reduce the computational burden and speed up the entire process. For instance, they may first train a model on a small sample (i.e., *approximate model*) to quickly test a new hypothesis, determine if the newly added feature improves accuracy, or tune their their hyper-parameters. Only if the initial results are promising, then they invest in training a *full model*, i.e., the model trained on the full datasets (since it may take significantly longer). The problem, however, is that this sampling process is ad-hoc and without any guarantees regarding the amount of error induced by sampling. The analysts do not know how much their sampling ratio or strategy affects the validity of their model tuning decisions. For example, had they trained a model with this new feature on the entire dataset rather than a small sample, maybe they would have seen a much higher accuracy, leading to include it in their final features.

Our Goal Given that (sub)sampling is already quite common in early stages of ML workloads—such as feature selection and

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
Copyright 2018 VLDB Endowment 2150-8097/18/10... $ 10.00.
DOI: https://doi.org/TBD

hyper-parameter tuning—we propose a high-level system abstraction for training ML models, using which analysts can explicitly request error-computation tradeoffs for several important classes of ML models. This involves systematic support for both (i) bounding the deviation of the approximate model from the full model given a sample size, and (ii) predicting the *minimum sample size* using which the trained model would meet a given error tolerance. The error guarantees can be placed on both model's parameters and its final predictions. Our abstraction provides strong, PAC-style guarantees for versatile scenarios. For example, using an error tolerance ε and a confidence level δ ($0 \leq \delta \leq 1$), the analysts can request a model whose parameters deviation from those of the full model is no more than ε, with probability at least $1 - \delta$. Likewise, the users can demand a model whose misclassification ratio is within ε from the full model with probability at least $1 - \delta$. (Analogous guarantees can be requested for the proximity of regression outputs.) Conversely, the analysts may train an approximate model, and then inquire about the probability of its predictions being more than ε different than the full model.

Benefits The benefits are multifold. First, predicting the sampling error enables analysts to make more reliable and informed decisions throughout their model tuning process. Second, guaranteeing the maximum error might make analysts more comfortable with using sampling, specially in their earlier explorations, which can lead to significant reduction of computational resources and significant improvement of analysts' productivity. Similarly, this will also eliminate the need for oversampling. Finally, rather than implementing their own sampling procedures, analysts will be more likely to rely on in-database sampling operators, allowing for many other optimization, reuse, and parallelism opportunities.

Challenges While estimating sampling error is a well-studied problem for SQL queries [10, 43], the problem is much more involved for ML models. The are two approaches here: (i) those that estimate the error before training the model (i.e., *predictive*), and (ii) those that estimate the error after a model is trained (i.e., *evaluative*). A well-known predictive technique is the so-called VC-dimension [42], which upper bounds the generalization error of a model. However, given that VC-dimension bounds are data-independent, they tend to be quite loose in practice [58]. The overly conservative error bounds means that the analysts still have to use the entire dataset, even if similar results could be obtained from a much smaller sample. [1]

Common techniques for the evaluative approach include cross-validation [14] and Radamacher complexity [42]. Since these techniques are data-dependent, they provide tighter error estimates. However, they only bound the generalization error. While useful for evaluating the model's quality on future (i.e., unseen) data, the generalization error provides little help in predicting how much the model quality would differ if the entire dataset were used instead of the current sample. Furthermore, when choosing the minimum sample size, the evaluative approaches can be quite expensive: one would need to train multiple models each on a different sample size until a desirable error tolerance is met. Given that most ML models do not have an incremental training procedure (besides a warm start [17]), training multiple models to find an appropriate sample size might take longer overall than simply training a model on the full dataset (see Section 6.4).

Our Approach Given a test example $\boldsymbol{x}$, the model's prediction is simply a function $m(\boldsymbol{x}; \theta)$, where θ is the model parameter learned during the training phase.[2] Let θ_N be the model parameter obtained if one trains on the entire dataset (say, of size N), and $\hat{\theta}_n$ be the model parameter obtained if one trains on a sample of size n.[3] Obtaining θ_n is fast when $n \ll N$; however, θ_N is unknown unless we use the entire dataset. Our key idea is to exploit the asymptotic distribution of $\theta_N - \hat{\theta}_n$ to analytically (thus, efficiently) derive the conditional distribution of $\hat{\theta}_N \mid \theta_n$, where $\hat{\theta}_N$ represents our (limited) probabilistic knowledge of θ_N (Theorem 1 and Corollary 1). The asymptotic distribution of $\theta_N - \hat{\theta}_n$ is available for the ML methods relying on maximum likelihood estimation for their parameter optimizations, which includes Generalized Linear Models [14] and Probabilistic PCA [56]. This indicates that, while we cannot determine the exact value of θ_N without training the full model, we can use the conditional distribution of $\hat{\theta}_N \mid \theta_n$ to probabilistically bound the deviation of θ_N from θ_n, and consequently, the deviation of $m(x; \theta_N)$ from $m(x; \theta_n)$ (Section 4.2). This also means we can even estimate the deviation of $m(x; \theta_N)$ from $m(x; \theta_{n''})$ for any other sample size, say n'', only using the model trained on the original sample of size n (Section 5.2). In other words, without having to perform additional training, we can efficiently search for the minimum sample size n', using which the approximate model, $m(x; \theta_{n'})$ would be guaranteed, with probability $1 - \delta$, not to deviate from $m(x; \theta_N)$ by more than ε.

Contributions We make the following contributions:

1. We introduce BlinkML, a system that offers error-computation tradeoffs for training ML models. (Sections 2 and 3)
2. We present an efficient algorithm that computes the probabilistic difference between an approximate model and the full model, without having to train the full model on the entire dataset. Our algorithm supports any ML model that relies on maximum likelihood estimation for its parameter optimization, which includes Generalized Linear Models (e.g., linear regression, logistic regression, max entropy classifier, Poisson regression) and Probabilistic Principal Component Analysis. (Section 4)
3. We propose a technique that analytically estimates the probabilistic difference between the full model and a model trained on a sample of an *arbitrary size*, without having to train either of them. BlinkML relies on this contribution for automatically and efficiently inferring the minimum sample size that would meet a given error tolerance requested by the user. (Section 5)
4. We empirically validate the statistical correctness and computational benefits of BlinkML through extensive experiments. (Section 6)

The remainder of this paper is organized as follows. Section 2 describe the user interaction with BlinkML. Section 3 explains the workflow and the architecture of BlinkML internals. Section 4 establishes statistical properties of BlinkML-supported ML models and describe how to exploit the statistical properties for estimating the accuracy of an approximate model. Section 5 describes how to efficiently estimate the minimum sample size that satisfies the user-requested accuracy. We present our experiments in Section 6 and discuss related work in Section 7.

[1]This is why VC-dimensions are sometimes used indirectly, as a comparative measure of quality [38].

[2]In this work, we focus on parametric models.

[3]Note that $\hat{\theta}_n$ is a random variable; a specific model parameter θ_n trained on a specific sample (of size n) is an instance of $\hat{\theta}_n$.

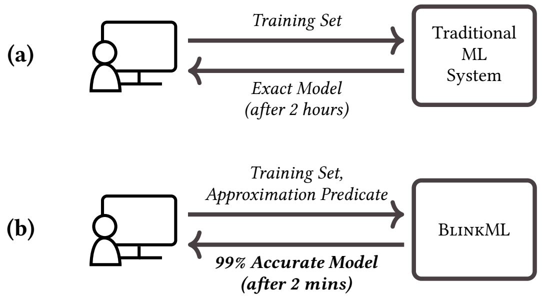


Figure 1: Interaction difference between (a) traditional ML systems and (b) BlinkML. BlinkML can quickly train an approximate ML model as specified by the user-supplied *approximation predicate*. Latency examples are based on our experiment results (Section 6).

2. USER INTERACTION

In this section, we describe how the user interacts with BlinkML. For BlinkML, the user is required to provide only one extra parameter in comparison to traditional ML systems.

2.1 Interface

This section describes BlinkML's interface. We first describe the typical interface of traditional ML systems. Then, we present the difference of BlinkML.

Traditional ML System For a traditional ML system, the user provides a training set D and specifies what class of model to train (e.g., linear regression, logistic regression, etc.). Then, the traditional ML system outputs a model $m_N(\cdot)$ trained on the given training set. We call $m_N(\cdot)$ a *full model*.

A training set D is a (multi-)set of N training examples sampled from an unknown distribution $\mathcal{D}$. For supervised learning (e.g., linear regression, logistic regression, max entropy classifier), each training example is a pair of a feature vector $\boldsymbol{x}$ and an observed output t. That is, $D = \{(\boldsymbol{x}_1, t_1), \ldots, (\boldsymbol{x}_N, t_N)\}$. The supervised learning includes both regression tasks and classification tasks. For regression tasks (e.g., linear regression), the observed output (or simply an *observation*) is a real value. For classification tasks (e.g., logistic regression, max entropy classifier), the observation is a class label (which is typically encoded using an integer). For unsupervised learning (e.g., PPCA), each training example is simply a feature vector. That is, $D = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$.

The trained model $m_N(\cdot)$ is used differently depending on the types of learning. For supervised learning, $m(\boldsymbol{x})$ makes a prediction for an unseen feature vector $\boldsymbol{x}$. For example, if $\boldsymbol{x}$ encodes a review of a restaurant, a trained logistic regression classifier $m_N(\boldsymbol{x})$ predicts if the review is positive or negative. For unsupervised learning, $m_N()$ captures certain characteristics of the training set. For example, if D is a set of hand-written digits, a trained PPCA model $m_N()$ outputs q number of underlying factors (or patterns) that best explains hand-written digits; here, q is the user parameter of PPCA.

BlinkML In addition to the inputs required for traditional ML systems, BlinkML needs one extra input: *approximation predicate*. The approximation predicate specifies the properties of the approximate model that will be trained by BlinkML. Figure 1 depicts this user interaction in comparison to traditional ML systems. The approximation predicate must be either of the following two types: (Mode A) a pair of an error bound ε and a confidence δ, or (Mode B) a pair of a sample size n and a confidence δ. Depending on the type of the given approximation predicate, BlinkML's operations differ as follows.

```
lr = blinkml.LogisticRegression()
ap = blinkml.ApproxPredicate(err=0.01, delta=0.05)
model = lr.train(training_set, ap)
predictions = model.predict(test_set)
```

Figure 2: A code snippet for training and making predictions with BlinkML.

Mode A: Given an error bound ε and a confidence δ, BlinkML returns an *approximate model* $m_{n'}(\cdot)$ such that the *model difference* between $m_{n'}(\cdot)$ and $m_N(\cdot)$ (i.e., full model) is within ε with probability at least $1 - \delta$. For classification tasks, the model difference is the chance that two models produce different class label predictions.[4] For regression tasks, the model difference is the expected difference between two models' predictions. The approximate model $m_{n'}(\cdot)$ is trained on a sample of size n', where the value of n' is automatically inferred by BlinkML. δ ensures that even though the approximate model is trained on a random sample, the estimated accuracy ε is guaranteed with high probability.

Mode B: Given a sample size n and a confidence δ, BlinkML returns (1) an *approximate model* $m_n(\cdot)$ trained on a sample of size n, and (2) the estimated accuracy ε_n such that the model difference between $m_n(\cdot)$ and $m_N(\cdot)$ is within ε_n with probability at least $1 - \delta$. The meanings of ε_n and δ, for classification and regression tasks, are identical to Mode A.

2.2 Supported ML Models

BlinkML can approximately train the ML models that rely on maximum likelihood estimation (MLE). Specifically, the MLE-based models are trained by minimizing an objective function $f_n(\theta)$ in the following form:

$$f_n(\theta) = \frac{1}{n} \sum_{i=1}^{n} \log \Pr(\boldsymbol{x}_i, t_i;\, \theta) + R(\theta) \tag{1}$$

These ML models include Generalized Linear Models (e.g., linear regression, logistic regression, maximum entropy classifier, Poisson regression) and Probabilistic Principal Component Analysis (PPCA). We describe in Section 4 how BlinkML exploits the characteristics of the above expression for computing the accuracies (i.e., ε) of approximate models.

However, the users of BlinkML are not required to understand any details (e.g., statistical properties) of these models. Training and making predictions with an approximate model can be accomplished by writing a short piece of code as in Figure 2.

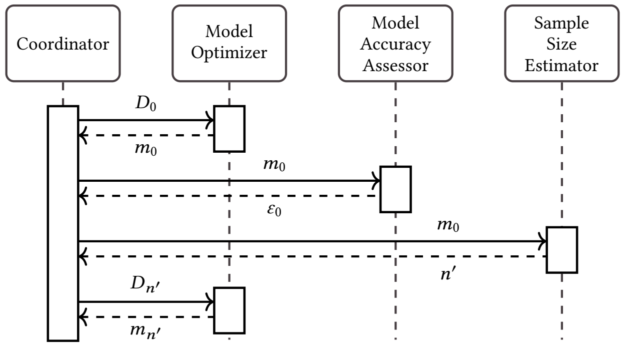
3. WORKFLOW AND ARCHITECTURE

This section describes the internal components of BlinkML and how they operate together to produce an accuracy-guaranteed approximate model. BlinkML has the following four components: (1) Model Optimizer, (2) Model Accuracy Assessor, and (3) Sample Size Estimator, and (4) Coordinator. We first present how those components interact in Section 3.1; then, we describe the details of each component in Section 3.3.

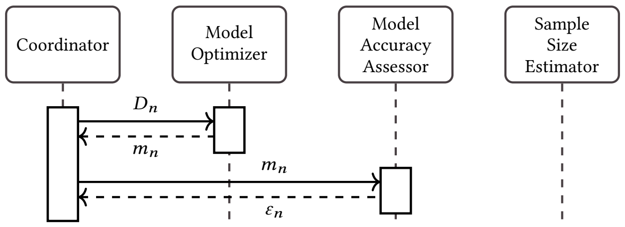
3.1 Workflow

BlinkML's operations differ based on the type of the given approximation predicate. Recall that an approximation predicate is

[4]The model difference is estimated based on the assumption that the unseen example $\boldsymbol{x}$ follows the same distribution as the training set.



(a) Mode A: Accuracy-guaranteed Approximate Model



(b) Mode B: Sample-size-specified Approximate Model

The letter D indicates a training set (e.g., D_0, D_n), m indicates a model (e.g., $m_0, m_n, m_{n'}$), n indicates a sample size (e.g., n'), and ε indicates an error (e.g., ε_0, ε_n).

Figure 3: Workflows of BlinkML's two modes of operations.

either (Mode A) an error bound ε and a confidence δ, or (Mode B) a sample size n and a confidence δ. We present their workflows individually below. Figure 3 depicts those workflows.

Mode A First, Coordinator obtains a size-n_0 sample D_{n_0} of the training set D. Here, D_{n_0} is called an *initial training set.* n_0 is set to 10K by default. Then, Coordinator lets Model Optimizer train an *initial model* m_0 on D_{n_0}, and lets Model Accuracy Assessor estimate the guaranteed accuracy ε_0 (with probability $1-\delta$) of m_0. If ε_0 is not larger than the requested error bound ε, Coordinator simply returns the initial model as an answer. Otherwise, Coordinator prepares to train another model, which is called a *final model.* To estimate the sample size n' for the final model $m_{n'}$, Coordinator lets Sample Size Estimator estimate the smallest n' such that the model difference between $m_{n'}$ and m_N (i.e., unknown full model) is not larger than ε with probability $1-\delta$. Note that this operation of Sample Size Estimator does not rely on Model Optimizer; that is, no additional (approximate) models are trained for producing n'. Finally, Coordinator lets Model Optimizer train $m_{n'}$ on a sample of size n' and returns $m_{n'}$ as an answer. A maximum of two approximate models are trained in Mode A.

Mode B First, Coordinator obtains a size-n sample D_n of the training set D, and lets Model Optimizer train an approximate model m_n on D_n. Next, Coordinator lets Model Accuracy Assessor compute ε_n such that the model difference between m_n and m_N (i.e., unknown full model) is not larger than ε_n with probability $1-\delta$. Finally, Coordinator returns m_n and ε_n as an output. A single approximate model is trained in Mode B.

3.2 Model Abstraction

To be agnostic to individual ML models, BlinkML relies on a abstract form presented below. How actual components of BlinkML utilize the abstract from is described in the following section.

The solution θ_n to the minimization problem in Section 2.2 is the values at which the gradient $g_n(\theta) = \nabla f_n(\theta)$ of the objective function $f_n(\theta)$ becomes a zero vector. That is,

$$g_n(\theta_n) = \left[\frac{1}{n} \sum_{i=1}^{n} q(\theta_n; \boldsymbol{x}_i, t_i) \right] + r(\theta_n) = \mathbf{0} \tag{2}$$

where we used $q(\theta; \boldsymbol{x}_i, t_i)$ to denote $\nabla_\theta \log \Pr(\boldsymbol{x}_i, t_i; \theta)$ and used $r(\theta)$ to denote $\nabla_\theta R(\theta)$. n is the number of examples used for training an approximate model m_n. For the full model m_N, n is set to N. We provide concrete examples of this abstraction in Appendix A.

3.3 Architecture

BlinkML's components rely on the abstract form presented in Section 3.2. To exploit the mathematical abstraction in actual ML training, BlinkML defines a class called *model class specification*. In this section, we first define the model class specification and then, describe the details of each of the three internal components: (1) Model Optimizer, (2) Model Accuracy Assessor, and (3) Sample Size Estimator. The last component, Coordinator, simply controls the flows among those internal components as described in Section 3.1; thus, no further details are provided.

Model Class Specification A model class specification is the Python class that includes model-specific information for BlinkML to approximately train a model. BlinkML defines a different model class specification for each model class (e.g., logistic regression, max entropy classifier, etc.). These model class specifications are used internally by BlinkML without being exposed to regular users.

A model class specification must implement (1) the `diff` function, (2) the `grads` function, and optionally implements (3) the `solve` function. The `diff` function, which we denote by $v(m_1, m_2)$, computes the difference between two models $m_1(\cdot)$ and $m_2(\cdot)$. For classification tasks (e.g., logistic regression, max entropy classifier), $v(m_1, m_2) = \mathrm{E}[m_1(\boldsymbol{x}) \neq m_2(\boldsymbol{x})]$ for $\boldsymbol{x} \sim \mathcal{D}$. For regression tasks (e.g., linear regression), $v(m_1, m_2) = \mathrm{E}[(m_1(\boldsymbol{x}) - m_2(\boldsymbol{x}))^2]$ for $\boldsymbol{x} \sim \mathcal{D}$. For PPCA, $v(m_1, m_2)$ computes one minus the average cosine similarity between two models' respective extracted factors. The `grads` function computes and returns a list of $q(\theta; \boldsymbol{x}_i, t_i) + r(\theta)$ for $i = 1, \ldots, n$ (defined in Equation (2)). The optional `solve` function computes the optimal model parameters using closed-form expressions. Currently, BlinkML implements the optional `solve` functions for linear regression and PPCA since closed-form expressions are available for their optimizations.

Model Optimizer Model Optimizer trains a model by finding the model's optimal parameter values θ. The inputs to Model Optimizer are two: (1) a subset D_n of training set D, where $n \leq N$, and (2) a model class specification. In practice, only the pointers to D_n and D are passed among components to avoid copying data.

For training the model m_n, Model Optimizer takes a different approach depending on the given model class specification. If the model class specification includes the optional `solve` function, Model Optimizer simply relies on the `solve` function for obtaining optimal model parameters. If the `solve` function is not present, BlinkML uses the `grads` function and the optimization libraries in the SciPy's `optimize` module. The optimizers in the `optimize` module requires the gradient $\nabla f_n(\theta)$ of an objective function $f_n(\theta)$

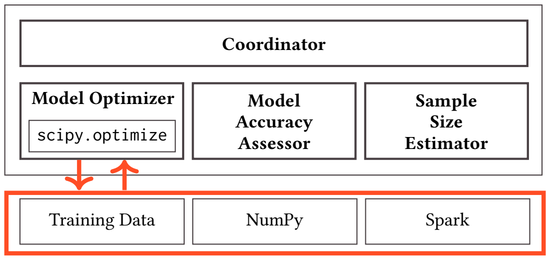


Figure 4: Architecture of BlinkML. Computation-intensive operations (in orange) are either processed locally or distributed to multiple workers using Spark, depending on the sizes of workloads.

for finding optimal parameter values[5]; the gradient is obtained by averaging the outputs of the `grads` function.

Note that when using gradient-based optimizers, computing gradients is typically the most time-consuming part. For small- and medium-scale problems (n < 1M, by default), all training examples are fed into the `grads` function, and all computations are performed locally using NumPy's `ndarray` and SciPy's sparse array for parallel computations. For large-scale problems ($n \geq$ 1M, by default), BlinkML distributes the gradient computations into multiple workers using Spark [59]. When distributing training examples to multiple workers, BlinkML sets a group of 10,000 training examples as a single element of Spark's RDD and lets Spark invoke `grads` on each element of RDD. We observed that this enables Spark to exploit NumPy's highly-efficient matrix operations. In contrast, when we set a single training example as an element of Spark's RDD, the performance of Spark's native parallelization was worse than NumPy (and also worse even than fully local processing) because the native parallelism is not aware that the operations performed on the elements of RDD are linear algebra operations; thus, Spark does not exploit fast matrix computations provided by BLAS and ATLAS (which are supported by NumPy and SciPy). Note that this distribution style is only available since the gradient expressions of the BlinkML-supported ML models have the special structure that easily enables data-parallelism; that is, $q(\theta; \mathbf{x}_i, t_i)$ is independent of $q(\theta; \mathbf{x}_j, t_j)$ given θ if $i \neq j$.

The output of Model Optimizer is the model $m_n(\cdot) = m(\,\cdot\,; \theta_n)$, which includes the parameter θ_n trained on D_n.

Model Accuracy Assessor Given an approximate model m_n, a model class specification, and a confidence δ, Model Accuracy Assessor computes ε_n such that $v(m_n, m_N) \leq \varepsilon_n$ with probability at least $1 - \delta$.

For this, Model Accuracy Assessor relies on that both m_n and m_N are essentially the same function (i.e., $m(\cdot)$) but with different model parameters, i.e., θ_n and θ_N, respectively. Although we cannot pinpoint the value of θ_N without training the full model m_N itself, we can still estimate its probability distribution (as will be described in Section 4.1). Model Accuracy Assessor uses that estimated probability distribution for estimating m_N's probability distribution (or more accurately, the probability distribution of the output of $m_N(\cdot)$). The probability distribution of m_N is then used for estimating the distribution of $v(m_n, m_N)$, which is the quantity we aim to upper-bound. The upper-bound ε_n is determined by finding the value that is larger than $v(m_n, m_N)$ with high probability. We describe more details of Model Accuracy Assessor in Section 4.

[5]The gradient may be numerically approximated using an objective function itself; however, BlinkML does not consider this option since numerical approximations are slower.

Sample Size Estimator Given an error bound ε, a confidence δ, and the initial model m_0, Sample Size Estimator estimates the minimum sample size n' such that $v(m_{n'}, m_N) \leq \varepsilon$ with probability at least $1 - \delta$, where $m_{n'}$ is the approximate model trained on a size-n' sample $D_{n'}$ of D. The core function of Sample Size Estimator is to compute $\Pr(v(m_{n''}, m_N) \leq \varepsilon)$ for an arbitrary n''. The probability is placed on $v(m_{n''}, m_N)$ since it is essentially a random variable; $D_{n''}$, on which $m_{n''}$ is trained, is a random sample.

Like Model Accuracy Assessor, Sample Size Estimator also exploits the probability distributions of $m_{n''}$ and m_N for computing $\Pr(v(m_{n''}, m_N) \leq \varepsilon)$. In this case, however, the model parameters of both $m_{n''}$ and m_N are unknown. Thus, Sample Size Estimator treats both of the model parameters as random variables and uses the joint probability distribution of $(\hat{\theta}_{n''}, \hat{\theta}_N)$, where $\hat{\theta}_{n''}$ is the random variable for the unknown model parameter $\theta_{n''}$ of $m_{n''}$. We describe more details of Sample Size Estimator in Section 5.

4. MODEL ACCURACY ASSESSOR

Model Accuracy Assessor is the component that estimates the accuracy of an approximate model. This section describes its estimation process. To understand Model Accuracy Assessor, it is crucial to understand the statistical properties of the abstract form presented in Equation (2), Section 3.2. In Section 4.1, we establish those statistical properties. Section 4.2 describes how to exploit the statistical properties for estimating the accuracy of an approximate model. Lastly, Section 4.3 presents how to efficiently compute some necessary statistics required for expressing the statistical properties.

4.1 Model Parameter Distribution

In this section, we present how to probabilistically express the parameter values of the (unknown) full model m_N only given an approximate model m_n. Let θ_n be the parameters of m_n, and θ_N be the parameters of m_N. Also, let $\hat{\theta}_n$ be a random variable that represents the distribution of the approximate model's parameters; θ_n is basically one instance of $\hat{\theta}_n$. We also use $\hat{\theta}_N$ to represent our (limited) knowledge of θ_N. In the remainder of this section, we first present the distribution of $\hat{\theta}_n - \hat{\theta}_N$. This distribution is then used to derive the conditional distribution of $\hat{\theta}_N \mid \theta_n$. This conditional distribution is used in the following section for estimating the accuracy of m_n.

The following theorem presents the distribution of $\hat{\theta}_n - \hat{\theta}_N$.

Theorem 1. *Let J be the Jacobian of $g_n(\theta) - r(\theta)$ evaluated at θ_n, and let H be the Jacobian of $g_n(\theta)$ evaluated at θ_n. Then,*

$$\hat{\theta}_n - \hat{\theta}_N \to \mathcal{N}(\mathbf{0}, \alpha H^{-1} J H^{-1}), \qquad \alpha = \frac{1}{n} - \frac{1}{N}$$

as $n \to \infty$. $\mathcal{N}$ denotes a normal distribution.

The above theorem is a generalization of the sampling distribution of the maximum likelihood estimator [24, 44], which we employ in the proof of the theorem. To minimize the overhead of Model Accuracy Assessor, it is important to compute those statistics, H^{-1} and J, efficiently. We describe our approach in Section 4.3.

Proof of Theorem 1. We first derive the distribution of $\hat{\theta}_n - \theta_\infty$, which will then be used to derive the distribution of $\hat{\theta}_n - \hat{\theta}_N$. Our derivation is the generalization of the result in [44]. The generalization is required since the original result does not include $r(\theta)$.

Let θ_∞ be the parameter values at which $g_\infty(\theta)$ becomes zero. Since the size of the training set is only N, θ_∞ exists only conceptually. Since θ_n is the optimal parameter values, it satisfies

$g_n(\theta_n) = \mathbf{0}$. According to the mean-value theorem, there exists $\bar{\theta}$ between θ_n and θ_∞ that satisfies:

$$H(\bar{\theta})(\theta_n - \theta_\infty) = g_n(\theta_n) - g_n(\theta_\infty) = -g_n(\theta_\infty)$$

where $H(\bar{\theta})$ is the Jacobian of $g_n(\theta)$ evaluated at $\bar{\theta}$. Note that $g_n(\theta_n)$ is zero since θ_n is obtained by finding the parameter at which $g_n(\theta)$ becomes zero.

Applying the multidimensional central limit theorem to the above equation produces the following:

$$\begin{aligned}\sqrt{n}\left(\hat{\theta}_n - \theta_\infty\right) &= -H(\bar{\theta})^{-1}\,\sqrt{n}\,g_n(\theta_\infty)\\ &= -H(\bar{\theta})^{-1}\,\frac{1}{\sqrt{n}}\sum_{i=1}^{n}\left(q(\theta_\infty;\boldsymbol{x}_i,t_i) + r(\theta_\infty)\right) \quad (3)\\ &\xrightarrow{n\to\infty} \mathcal{N}(\mathbf{0},\, H^{-1}JH^{-1}) \quad (4)\end{aligned}$$

where H is a shorthand notation of $H(\bar{\theta})$. To make a transition from Equation (3) to Equation (4), an important relationship called the *information matrix equality* is used. According to the information matrix equality, the covariance of $q(\theta;\boldsymbol{x}_i,t_i)$ is equal to the Hessian of the negative log-likelihood expression, which is equal to J.

Now, we derive the distribution of $\hat{\theta}_n - \hat{\theta}_N$. We use the fact that $\hat{\theta}_N$ is the optimal parameter for D_N which is a union of D_n and $D_N - D_n$, where $\hat{\theta}_n$ is the optimal parameter for D_n. To separately capture the randomness stemming from D_n and $D_N - D_n$, we introduce two random variables X_1, X_2 that independently follow $\mathcal{N}(\mathbf{0},\, H^{-1}JH^{-1})$. From Equation (4), $\hat{\theta}_n - \theta_\infty = (1/\sqrt{n})\,X_1$. Also, let $q_i = q(\theta_\infty;\boldsymbol{x}_i,t_i)$ for simplicity; then,

$$\begin{aligned}\sqrt{N}\left(\hat{\theta}_N - \theta_\infty\right) &= -H^{-1}\frac{1}{\sqrt{N}}\Big(\sum_{i=1}^{n} q_i + \sum_{i=1}^{N-n} q_i\Big)\\ &= -H^{-1}\left[\frac{\sqrt{n}}{\sqrt{N}}\frac{1}{\sqrt{n}}\sum_{i=1}^{n} q_i + \frac{\sqrt{N-n}}{\sqrt{N}}\frac{1}{\sqrt{N-n}}\sum_{i=1}^{N-n} q_i\right]\\ &\xrightarrow{n\to\infty} \frac{\sqrt{n}}{\sqrt{N}}X_1 + \frac{\sqrt{N-n}}{\sqrt{N}}X_2\end{aligned}$$

Since $\hat{\theta}_n - \hat{\theta}_N = (\hat{\theta}_n - \theta_\infty) - (\hat{\theta}_N - \theta_\infty)$,

$$\begin{aligned}\hat{\theta}_n - \hat{\theta}_N &= \frac{1}{\sqrt{n}}X_1 - \frac{\sqrt{n}}{N}X_1 - \frac{\sqrt{N-n}}{N}X_2\\ &= \left(\frac{1}{\sqrt{n}} - \frac{\sqrt{n}}{N}\right)X_1 - \frac{\sqrt{N-n}}{N}X_2\end{aligned}$$

Note that $\hat{\theta}_n - \hat{\theta}_N$ follows a normal distribution since it is a linear combination of two random variables that independently follow normal distributions. Thus,

$$\begin{aligned}&\hat{\theta}_n - \hat{\theta}_N \xrightarrow{n\to\infty}\\ &\mathcal{N}\left(\mathbf{0},\,\left(\frac{1}{\sqrt{n}} - \frac{\sqrt{n}}{N}\right)^2 H^{-1}JH^{-1} + \left(\frac{\sqrt{N-n}}{N}\right)^2 H^{-1}JH^{-1}\right)\\ &= \mathcal{N}\left(\mathbf{0},\,\left(\frac{1}{n} - \frac{1}{N}\right)H^{-1}JH^{-1}\right)\end{aligned}$$

□

Now we present the conditional distribution of $\hat{\theta}_N \mid \theta_n$ in the following corollary.

Corollary 1. Without any *a priori* preference on θ_N,

$$\hat{\theta}_N \mid \theta_n \to \mathcal{N}(\theta_n,\, \alpha\, H^{-1}JH^{-1}), \qquad \alpha = \frac{1}{n} - \frac{1}{N}$$

as $n \to \infty$.

Proof of Corollary 1. Observe that $\hat{\theta}_n - \hat{\theta}_N$ and $\hat{\theta}_N - \theta_\infty$ are independent because they are jointly normally distributed and the covariance between them are zero as shown below:

$$\begin{aligned}&\mathrm{Cov}(\hat{\theta}_n - \hat{\theta}_N,\, \hat{\theta}_N - \theta_\infty)\\ &= \frac{1}{2}\left(\mathrm{Var}(\hat{\theta}_n - \hat{\theta}_N + \hat{\theta}_N - \theta_\infty) - \mathrm{Var}(\hat{\theta}_n - \hat{\theta}_N) - \mathrm{Var}(\hat{\theta}_N - \theta_\infty)\right)\\ &= \frac{1}{2}\left(\frac{1}{n} - \left(\frac{1}{n} - \frac{1}{N}\right) - \frac{1}{N}\right) H^{-1}JH^{-1} = \mathbf{0}\end{aligned}$$

Thus, $\mathrm{Var}(\hat{\theta}_n - \theta_N) = \mathrm{Var}(\hat{\theta}_n - \hat{\theta}_N \mid \theta_N) = \alpha H^{-1}JH^{-1}$, which implies

$$\hat{\theta}_n \sim \left(\theta_N,\, \alpha\, H^{-1}JH^{-1}\right) \quad (5)$$

Using Bayes' theorem,

$$\Pr(\theta_N \mid \theta_n) = (1/Z)\ \Pr(\theta_n \mid \theta_N)\ \Pr(\theta_N)$$

for some normalization constant Z. Since there is no preference on $\Pr(\theta_N)$, we set a constant to $\Pr(\theta_N)$. Then, from Equation (5), $\hat{\theta}_N \mid \theta_n \sim \mathcal{N}(\theta_n, \alpha\, H^{-1}JH^{-1})$. □

The following section describes how to use this conditional probability distribution $\hat{\theta}_N \mid \theta_n$ for computing the accuracy of an approximate model.

4.2 Probabilistic Error Bound

In this section, we describe how Model Accuracy Assessor estimates the accuracy of an approximate model m_n. Specifically, we describe how to compute ε_n such that $v(m_n, m_N) \le \varepsilon_n$ with probability at least $1-\delta$, by exploiting the conditional distribution $\hat{\theta}_N \mid \theta_n$ derived in the previous section.

Let $h(\theta_N)$ denote the probability density function of the normal distribution with mean θ_n and covariance matrix $\alpha H^{-1}JH^{-1}$ (obtained in Corollary 1). Then, we aim to find ε_n that satisfies:

$$\left(\int \mathbb{1}\left[v(m(\,\cdot\,;\theta_n), m(\,\cdot\,;\theta_N)) \le \varepsilon_n\right]\, h(\theta_N)\, d\theta_N\right) \ge 1 - \delta \quad (6)$$

where $\mathbb{1}\,[\cdot]$ is the indicator function that returns 1 if its argument is true and returns 0 otherwise. The above expression involves blackbox functions such as $m(\cdot)$ and $v(\cdot,\cdot)$; thus, it cannot be analytically computed in general. We also considered analytic computations of Equation (6) by relying on model-specific information; however, those approaches are non-trivial. We discuss the hardness of the alternative approach at the end of this section.

Model Accuracy Assessor approximately computes the integration p_v in Equation (6) using the empirical distribution of $h(\theta_N)$ as follows. Let $\theta_{N,1}, \ldots, \theta_{N,k}$ be i.i.d. samples drawn from $h(\theta_N)$. Then,

$$\begin{aligned}p_v &= \int \mathbb{1}\left[v(m(\,\cdot\,;\theta_n),\, m(\,\cdot\,;\theta_N)) \le \varepsilon_n\right]\, h(\theta_N)\, d\theta_N \quad (7)\\ &\approx \frac{1}{k}\sum_{i=1}^{k}\mathbb{1}\left[v(m(\,\cdot\,;\theta_n),\, m(\,\cdot\,;\theta_{N,i})) \le \varepsilon_n\right] = \tilde{p}_v \quad (8)\end{aligned}$$

The value of $\tilde{p}_v$ converges to p_v as $k \to \infty$; thus, the estimate is consistent. We analyze its accuracy in more detail shortly. To obtain a large number k of sampled values, an efficient sampling algorithm is necessary. Since $\hat{\theta}_N$ follows a normal distribution, there are already standard libraries, such as `numpy.random`. However, BlinkML uses its own custom sampler to avoid computing the covariance matrix $H^{-1}JH^{-1}$ directly (see Section 5.3). To find ε_n that makes $\tilde{p}_v \ge 1-\delta$, it suffices to find the $k\delta$-th smallest value in the list of $v(m(\,\cdot\,;\theta_n),\, m(\,\cdot\,;\theta_{N,i}))$ for $i = 1, \ldots, k$.

Accuracy Now we analyze the accuracy of using Equation (8). For our analysis, we use the expected relative root mean square (rms) error of $\tilde{p}_v$ (in comparison to p_v). Observe that since $\hat{\theta}_{N,1}$, $\ldots, \hat{\theta}_{N,k}$ are i.i.d. samples, $\tilde{p}_v$ follows a binomial distribution with mean p_v and standard deviation $\sqrt{p_v(1-p_v)/k}$. Thus, the expected relative rms error is $\sqrt{(1/p_v - 1)/k}$. For instance, when $p_v = 0.90$ and $k = 1{,}000$, the relative rms error is about 0.01. BlinkML uses 1,000 for k by default and targets the cases where p_v is not smaller than 0.90 (i.e., $\delta \leq 0.10$). Thus, the expected relative rms error of $\tilde{p}_v$ is maintained small.

Alternative Approaches We briefly describe why using model-specific information does not necessarily lead to more efficient computation of Equation (7). For our description, we use logistic regression as an example. Since logistic regression is a classification algorithm, the model difference between $m_n = m(\boldsymbol{x};\theta_n)$ and $m_N = m(\boldsymbol{x};\theta_N)$ is computed by $\mathrm{E}[m(\boldsymbol{x};\theta_n) \neq m(\boldsymbol{x};\theta_N)]$, where the model prediction of logistic regression is $m(\boldsymbol{x};\theta) = \mathbb{1}\left[\theta^\top \boldsymbol{x} \geq 0\right]$. Thus,

$$p_v = \int \mathrm{E}[m(\boldsymbol{x};\theta_n) = m(\boldsymbol{x};\theta_N)]\, h(\theta_N)\, d\theta_N$$
$$= \frac{1}{\ell}\sum_{j=1}^{\ell} \int_{A_+} \mathbb{1}\left[\theta_n^\top \boldsymbol{x}_j \geq 0\right] dh(\theta_N) + \int_{A_-} \mathbb{1}\left[\theta_n^\top \boldsymbol{x}_j < 0\right] dh(\theta_N)$$

where ℓ is the number of training examples used for computing the expectation, A_+ is the area of θ_N such that $\theta_N^\top \boldsymbol{x}_j \geq 0$, and A_- is the area of θ_N such that $\theta_N^\top \boldsymbol{x}_j < 0$. The above integration computes the probability that the full model's prediction for $\boldsymbol{x}_j$ lies on the same side of the decision boundary (i.e., $\theta^\top \boldsymbol{x}_j \geq 0$ or $\theta^\top \boldsymbol{x}_j < 0$) as the approximate model's prediction for $\boldsymbol{x}_j$. Computing the above integration is non-trivial because θ_N is multidimensional and the areas, A_+ and A_-, cannot be analytically expressed in general. Moreover, integrating multivariate normal distribution, $h(\theta_N)$, cannot be computed analytically. This difficulty becomes even more severe for multi-class classification algorithms, such as max entropy classifier. In contrast, Model Accuracy Assessor's current approach easily generalizes to all supported ML models, while its runtime overhead is still low (see Section 6.6 for empirical study of runtime overhead).

4.3 Computing Necessary Statistics

In this section, we present several methods for computing H (which appears in Theorem 1) and discuss pros and cons of those methods. Computing J is straightforward given H since $J = H - J_r$, where J_r is the Jacobian of $r(\theta)$. We present three methods: (1) ClosedForm, (2) InverseGradients, and (3) ObservedFisher.

Method 1: ClosedForm ClosedForm uses the analytic form of the Jacobian $H(\theta)$ of $g_n(\theta)$, and set $\theta = \theta_n$ by the definition of H. For instance, $H(\theta)$ of logistic regression is expressed as follows:

$$H(\theta) = \frac{1}{n} X^\top Q X + \beta I$$

where X is a n-by-d matrix whose i-th row is $\boldsymbol{x}_i$, and Q is a d-by-d diagonal matrix whose i-th diagonal entry is $\sigma(\theta^\top \boldsymbol{x}_i)(1 - \sigma(\theta^\top \boldsymbol{x}_i))$. When $H(\theta)$ is available as in the case of logistic regression, ClosedForm is fast and exact.

However, inverting H is computationally expensive when d is large. Also, using ClosedForm is less straightforward when obtaining analytic expression of $H(\theta)$ is non-trivial. Thus, BlinkML supports ClosedForm only for linear regression and logistic regression. For other methods, such as max entropy classifier and PPCA, either of the following two methods is used.

Method 2: InverseGradients InverseGradients numerically computes H by relying on the Taylor expansion of $g_n(\theta)$: $g_n(\theta_n + d\theta) \approx g_n(\theta_n) + H d\theta$. Since $g_n(\theta_n) = \mathbf{0}$, the Taylor expansion simplifies to:

$$g_n(\theta_n + d\theta) \approx H d\theta$$

The value of $g_n(\theta_n + d\theta)$ and $g_n(\theta_n)$ are computed using the `grads` function provided by the model class specification. The remaining question is what values of $d\theta$ to use for computing H. Since H is a d-by-d matrix, BlinkML uses d number of linearly independent $d\theta$ to fully construct H. That is, let P be ϵI, where ϵ is a small real-value (10^{-6} by default). Also, let R be the d-by-d matrix whose i-th column is $g_n(\theta_n + P_{\cdot,i})$ where $P_{\cdot,i}$ is the i-th column of P. Then,

$$R \approx HP \quad \Rightarrow \quad H \approx RP^{-1}$$

Since InverseGradients only relies on the `grads` function, it is applicable to all supported models. Although InverseGradients is accurate, InverseGradients is still computationally inefficient for high-dimensional data since the `grads` function must be called d times. We study its runtime overhead in Section 6.7.

Method 3: ObservedFisher ObservedFisher numerically computes H by relying on the information matrix equality [24]. According to the information matrix equality, the covariance matrix C of $q(\theta_n; \boldsymbol{x}_i, t_i)$ for $i = 1, \ldots, n$ is equal to J; then, H can simply be computed as $H = J + J_r$. Instead of computing C directly, however, ObservedFisher takes a slightly different approach to ensure the positive definiteness of J and H. This process also ensures the positive definiteness of $H^{-1}JH^{-1}$, which is the requirement for the covariance matrix of a normal distribution. Let Q be the n-by-d matrix whose i-th row is $q(\theta_n; \boldsymbol{x}_i, t_i)$. Then, ObservedFisher performs the singular value decomposition of $Q^\top$ to obtain U, Σ, and V such that $Q^\top = U\Sigma V^\top$. Then, ObservedFisher computes J as follows:

$$J = C = Q^\top Q = U\,\Sigma^2\,U^\top \approx U\,\Sigma_+^2\,U^\top \tag{9}$$

where Σ_+ is a diagonal matrix with only positive singular values. Note that the diagonal entries of Σ_+^2 are the eigenvalues of J; because they are also all positive, J is positive definite.

ObservedFisher requires only a single call of the `grads` function; thus, ObservedFisher is faster than InverseGradients particularly for high-dimensional data (see Section 6.7 for experiments). Also, ObservedFisher exposes U and Σ_+, which are used for BlinkML's custom sampler (Section 5). Due to these reasons, ObservedFisher is BlinkML's default approach to computing H and J.

5. SAMPLE SIZE ESTIMATOR

Sample Size Estimator is the component that estimates the minimum sample size n' such that the difference between an approximate model $m_{n'}$ and the full model m_N—$v(m_{n'}, m_N)$—is not larger than the requested error bound ε with probability at least $1 - \delta$.

The core feature of Sample Size Estimator is that it does not train any additional approximate models; it only relies on the initial model m_0 given to this component. For this, Sample Size Estimator uses probabilistic modeling of $m_{n'}$ and m_N, which is described in Section 5.1. Section 5.2 presents how to estimate n' relying on the probabilistic modeling. Lastly, Section 5.3 describes several optimizations that enable fast estimation of n'.

5.1 Assessing Model Quality sans Training

This section presents that given the initial model m_0, how Sample Size Estimator computes the probability of $v(m_{n'}, m_N) \leq \varepsilon$. Since both models—$m_{n'} = m(\,\cdot\,; \hat{\theta}_{n'})$ and $m_N = m(\,\cdot\,; \hat{\theta}_N)$—are uncertain, Sample Size Estimator uses the probability distributions of $\hat{\theta}_{n'}$ and $\hat{\theta}_N$ for computing $p_v = \Pr(v(m_{n'}, m_N) \leq \varepsilon)$. The computed probability p_v is then used in the following section for estimating n'.

Like Model Accuracy Assessor, Sample Size Estimator approximately computes p_v using the i.i.d. samples from the joint distribution $h(\theta_{n'}, \theta_N)$ of $(\theta_{n'}, \theta_N)$, as follows:

$$p_v(n') = \iint \mathbb{1}\,[v(m(\,\cdot\,; \theta_{n'}), m(\,\cdot\,; \theta_N)) \leq \varepsilon]\; h(\theta_{n'}, \theta_N)\, d\theta_{n'}\, d\theta_N$$

$$\approx \frac{1}{k} \sum_{i=1}^{k} \mathbb{1}\left[v(m(\,\cdot\,; \theta_{n',i}), m(\,\cdot\,; \theta_{N,i})) \leq \varepsilon\right] = \tilde{p}_v(n'0) \qquad (10)$$

To obtain i.i.d. samples, $(\theta_{n',i}, \theta_{N,i})$ for $i = 1, \ldots, k$, from $h(\theta_{n'}, \theta_N)$, Sample Size Estimator uses the following relationship:

$$\Pr(\theta_{n'}, \theta_N \mid \theta_0) = \Pr(\theta_N \mid \theta_{n'})\; \Pr(\theta_{n'} \mid \theta_0)$$

where the conditional distributions, $\theta_N \mid \theta_{n'}$ and $\theta_{n'} \mid \theta_0$, are obtained using Corollary 1. That is, Model Accuracy Assessor uses the following two-stage sampling procedure. It first samples $\theta_{n',i}$ from $\mathcal{N}(\theta_0,\, \alpha_1 H^{-1} J H^{-1})$ where $\alpha_1 = (1/n_0 - 1/n')$; then, samples $\theta_{N,i}$ from $\mathcal{N}(\theta_{n',i},\, \alpha_2 H^{-1} J H^{-1})$ where $\alpha_2 = (1/n' - 1/N)$. This process is repeated for every $i = 1, \ldots, k$ to obtain k pairs of $(\theta_{n',i}, \theta_{N,i})$. Finally, $\tilde{p}_v(n')$ is obtained by counting the fraction of the cases that $\mathbb{1}\left[v(m(\,\cdot\,; \theta_{n',i}), m(\,\cdot\,; \theta_{N,i})) \leq \varepsilon\right]$ returns 1.

5.2 Sample Size Searching

To find the minimum n' such that $\tilde{p}_v(n') \geq 1 - \delta$, Sample Size Estimator uses binary search, relying on that $\tilde{p}_v(n')$ tends to be an increasing function of n'. We first provide an intuitive explanation; next, we present a formal argument.

Observe that $\tilde{p}_v(n')$ relies on the two model parameters $\theta_{n'}$ and θ_N. If $\theta_n = \theta_N$, $\tilde{p}_v(n)$ is trivially equal to 1. According to Theorem 1, the difference between those two parameters, i.e., $\hat{\theta}_{n'} - \hat{\theta}_N$, follows a normal distribution whose covariance matrix shrinks by a factor of $1/n' - 1/N$. Therefore, those parameter values become closer as $n' \to N$, which implies that the value of $\tilde{p}_v(n')$ must increase toward 1 as $n \to N$. The following theorem formally shows that $\tilde{p}_v(n')$ is guaranteed to be an increasing function for a large class of cases.

Theorem 2. *Let $h(\theta; \gamma)$ be the probability density function of a normal distribution with mean θ_N and covariance matrix $\gamma\, C$, where γ is a real number, and C is an arbitrary positive semidefinite matrix. Also, let B be the box area of θ such that $v(m(\,\cdot\,; \theta), m(\,\cdot\,; \theta_N)) \leq \varepsilon$. Then, the following function $p_v(\gamma)$*

$$p_v(\gamma) = \int_B h(\theta; \gamma)\, d\theta$$

is a decreasing function of γ.

Proof of Theorem 2. Without loss of generality, we prove Theorem 2 for the case where the dimension of training examples, d, is 2. It is straightforward to generalize our proof to the case with an arbitrary d. Also, without loss of generality, we assume B is the box area bounded by $(-1, -1)$ and $(1, 1)$. Then, $p_v(\gamma)$ is expressed as

$$\int_{(-1,-1)}^{(1,1)} \frac{1}{\sqrt{(2\pi)^2\, |\gamma C|}} \exp\left(-\theta^\top (\gamma C)^{-1} \theta\right)\, d\theta$$

To prove the theorem, it suffices to show that $\gamma_1 < \gamma_2 \Rightarrow p_v(\gamma_1) > p_v(\gamma_2)$ for arbitrary γ_1 and γ_2. By definition,

$$p_v(\gamma_1) = \int_{-(1,1)}^{(1,1)} \frac{1}{\sqrt{(2\pi)^2\, |\gamma_1 C|}} \exp\left(-\theta^\top (\gamma_1 C)^{-1}\, \theta\right)\, d\theta$$

By substituting $\sqrt{\gamma_1/\gamma_2}\, \theta$ for θ,

$$p_v(\gamma_1) = \int_{-\sqrt{\gamma_2/\gamma_1}\,(1,1)}^{\sqrt{\gamma_2/\gamma_1}\,(1,1)} \frac{1}{\sqrt{(2\pi)^2\, |\gamma_2 C|}} \exp\left(-\theta^\top (\gamma_2 C)^{-1}\, \theta\right)\, d\theta$$

$$> \int_{-(1,1)}^{(1,1)} \frac{1}{\sqrt{(2\pi)^2\, |\gamma_2 C|}} \exp\left(-\theta^\top (\gamma_2 C)^{-1}\, \theta\right)\, d\theta = p_v(\gamma_2)$$

because the integration range for $p_v(\gamma_1)$ is larger. □

Note that since binary search is used, Sample Size Estimator needs to compute $\tilde{p}_v(n')$ for different values of n' in total $O(\log_2(N - n_0))$ times; thus, fast mechanism for producing i.i.d. samples is desirable. The following section describes BlinkML's optimizations.

5.3 Optimizations for Fast Sampling

This section presents Sample Size Estimator's sampling mechanism. For computing $p_v(n')$, Sample Size Estimator relies on the i.i.d. samples from the normal distribution with covariance matrix $(1/n' - 1/N)\, H^{-1} J H^{-1}$. A basic approach would be to use off-the-shelf functions, such as the one shipped in the `numpy.random` module, for every different n'. Albeit simple, this basic approach involves many redundant operations that could be avoided. We describe two orthogonal approaches to reduce the redundancy, which enables much faster sampling operations.

Sampling by Scaling We can avoid invoking a sampling function multiple times for different n' by exploiting the structural similarity of the covariance matrices associated with different n'. Let $\hat{\theta}_n \sim \mathcal{N}(\mathbf{0},\, (1/n - 1/N) H^{-1} J H^{-1})$, and let $\hat{\theta}_0 \sim \mathcal{N}(\mathbf{0},\, H^{-1} J H^{-1})$. Then, there exists the following relationship:

$$\hat{\theta}_n = \sqrt{1/n - 1/N}\; \hat{\theta}_0$$

This indicates that we can first draw i.i.d. samples from the unscaled distribution $\mathcal{N}(\mathbf{0}, H^{-1} J H^{-1})$; then scale those sampled values by $\sqrt{1/n - 1/N}$ whenever the i.i.d. samples from $\mathcal{N}(\mathbf{0}, (1/n - 1/N)\, H^{-1} J H^{-1})$ are needed.

Avoiding Direct Covariance Computation When $r(\theta) = \beta\theta$ in Equation (2) (i.e., no regularization or L2 regularization), Sample Size Estimator avoids the direct computations of $H^{-1} J H^{-1}$. Instead, it simply draws samples from the standard normal distribution and applies an appropriate linear transformation L to the sampled values. Note that sampling from the standard normal distribution is much faster because no dependencies need to be enforced among sampled values.

For this, the following relationship is used:

$$z \sim \mathcal{N}(\mathbf{0},\, I) \quad \Rightarrow \quad L\, z \sim \mathcal{N}(\mathbf{0},\, L\, L^\top)$$

That is, if Sample Size Estimator finds L such that $L\, L^\top = H^{-1} J H^{-1}$, it can obtain the samples of $\hat{\theta}_0$ by multiplying L to the samples drawn from the standard normal distribution.

Specifically, Sample Size Estimator performs the following for obtaining L. Observe from Equation (9) that $J = U\,\Sigma_+^2\,U^\top$. Since $H = J + \beta$, $H = U\,(\Sigma_+^2 + \beta I)\,U^\top$. Thus,

$$H^{-1}JH^{-1} = U\,(\Sigma_+^2 + \beta I)^{-1}\,U^\top\,U\,\Sigma_+^2\,U^\top\,U\,(\Sigma_+^2 + \beta I)^{-1}\,U^\top$$
$$\Rightarrow H^{-1}JH^{-1} = (U\Lambda)\,(U\Lambda)^\top$$

where Λ is a diagonal matrix whose i-th diagonal entry is $s_i/(s_i^2 + \beta)$, where s_i is the i-th singular value of J contained in Σ_+. Note that both U and Σ_+ are anyway computed when ObservedFisher is used for computing necessary statistics in Section 4.3. Thus, computing L only involves a simple matrix multiplication.

6. EXPERIMENTS

In this section, we empirically evaluate BlinkML. Our evaluations aim to answer the following questions:

1. Mode A: Is approximate ML training much faster?
2. Mode A: Does BlinkML satisfy the requested accuracies?
3. Mode A: Are the estimated minimum sample sizes optimal?
4. Mode B: Are the accuracies of approximate models correctly estimated?
5. Both Modes: How large is the runtime overhead of BlinkML?

In sum, our experiments provide the following answers:

1. BlinkML reduced the training time by 35.03%–99.69% for training 99% accurate models. In Section 6.2, we present training time savings for various combinations of ML models and datasets, and for different requested accuracy levels.
2. The actual accuracies of BlinkML's approximate models were higher than the requested accuracies in most of the cases. In Section 6.3, we compare requested accuracies and actual accuracies for various combinations of ML models and datasets, and for different requested accuracy levels.
3. BlinkML's estimated minimum sample sizes were close to optimal. In Section 6.4, we compare BlinkML's estimated minimum sample size to the ones obtained by a (slow) naïve approach. We also show that BlinkML's training time was almost optimal; its training times were only slightly higher compared to the cases where *a prior* knowledge of optimal sample sizes was available.
4. The accuracies of approximate models were correct. In Section 6.5, we compare the BlinkML's guaranteed model accuracies and the actual model accuracies for various combinations of ML models and datasets, and for different sample sizes.
5. The runtime overheads of BlinkML were 2.3%–7.4% of the entire training in Mode A and 3%–22% of the entire training in Model B. In Section 6.6, we analyze the runtime consumed by different components of BlinkML.

Furthermore, we also empirically study the pros and cons of InverseGradients and ObservedFisher, general statistics computation methods described in Section 6.7. We present above experiment results after describing our experiment setup.

6.1 Experiment Setup

We present the ML models, the datasets, and the computational environments used in our experiments.

Models The following three ML models were used for evaluations. We describe them below.

1. Logistic Regression (LR): LR is a binary classifier. LR is trained by finding parameter values of a sigmoid prediction function such that the function minimizes the misclassification on a training set. No closed-form solutions exist for LR; thus, BlinkML trains LR models using an iterative optimization method (which we discuss shortly). We used L2-regularized LR with $\beta = 0.001$.

Dataset	No. of Examples (N)	Dimension	Size on Disk
`HIGGS`	77,000,000	28	30 GB
`Yelp`	10,000,000	1,000	3.1 GB
`MNIST`	4,500,000	784	6.9 GB

Table 1: Key statistics of experiment datasets.

2. Max Entropy Classifier (ME): ME is a multi-class classifier. ME is trained by finding parameter values of a softmax prediction function such that the function minimizes the misclassification on a training set. No closed-form solutions exist for ME; thus, BlinkML trains ME using an iterative optimization method. We used L2-regularized ME with $\beta = 0.001$.
3. Probabilistic Principal Component Analysis (PPCA): PPCA is a factor-analysis method. PPCA is trained by finding a small number (q) of vectors that can most accurately reconstruct the training set when linearly transformed [56]. The optimal parameters (i.e., factors) can be found by performing eigendecomposition of a sample covariance matrix constructed from a training set. We disabled the use of Spark specifically for PPCA because local covariance computation was faster.

For LR and ME, an optimization algorithm must be chosen. Based on our preliminary tests, BlinkML chooses the optimization algorithm as follows. If a training set is low-dimensional ($d < 100$), BlinkML uses the BFGS optimization algorithm. If a training set is high-dimensional ($d \geq 100$), BlinkML uses a memory-efficient alternative called L-BFGS. Note that BlinkML's operations are agnostic to the choice of an optimization algorithm.

Datasets In our experiments, we used three real-world datasets. We briefly describe them below.

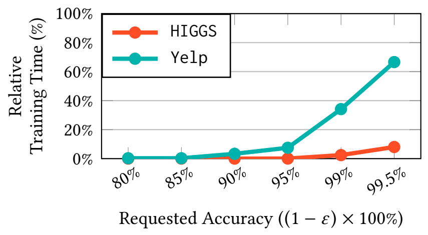
1. `HIGGS`: This is a 7× scaled version of a physical simulation dataset [13]. This dataset includes binary class labels; thus, we use `HIGGS` for LR and PPCA ($q = 2$).
2. `Yelp`: This is a 10% subset of publicly available Yelp reviews [9]. Each training example is a pair of an English review and a rating (between 0 and 5). The original dataset contains 737,530 distinct words; we performed feature selection and used 1,000 most frequently used words. For LR (a binary classifier), we converted a rating of 0–2 to 0 (negative) and a rating of 3–5 to 1 (positive).
3. `MNIST`: This dataset is a larger version [4] of a hand-written digit dataset called MNIST. Each training example is a pair of a bitmap image and the digit represented by the image. When PPCA was used for `MNIST`, we set q to 10.

Several key statistics of these datasets are summarized in Table 1.

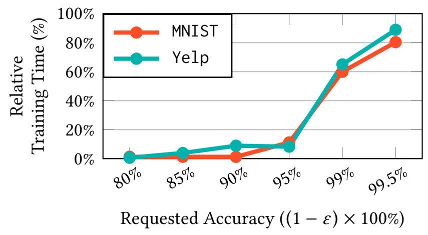
Environments All our experiments were conducted on an EC2 instance of the `m5.4xlarge` type (16 CPU cores, 64 GB memory, Ubuntu 16.04). For Spark, we used five additional instances of the `m5.2xlarge` type (8 CPU cores, 32 GB memory). We used Python 3.6 shipped with Conda [1] and Spark 2.2 shipped with Cloudera Manager 5.11. Our time measurements do not include the time for data loading or the time for distributing data to multiple Spark workers.

6.2 Training Time Saving (Mode A)

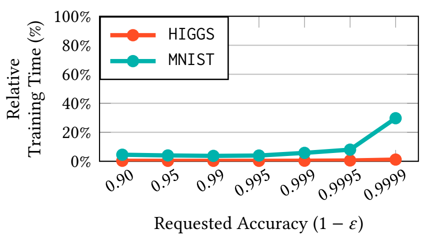
This section compares BlinkML's approximate model training time to the full model training time, when BlinkML operates in Mode A. We used six combinations of a model and a dataset: (LR, `HIGGS`), (LR, `Yelp`), (ME, `MNIST`), (ME, `Yelp`), (PPCA, `HIGGS`), and (PPCA, `MNIST`). For LR and ME, we varied the requested accuracy



(a) Logistic Regression (LR)

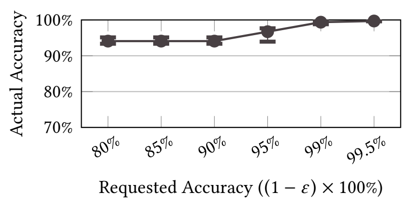


(b) Max Entropy Classifier (ME)

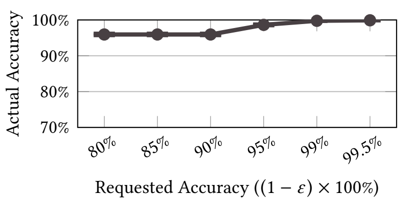


(c) Probabilistic PCA (PPCA)

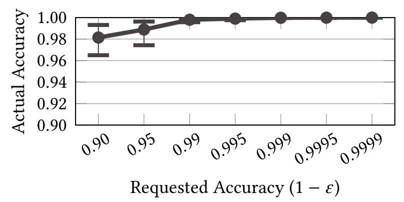
Figure 5: BlinkML's relative training time in comparison to full model training. The actual accuracies of the trained models are separately studied in Figure 6. The raw data of this figure are in Table 3.



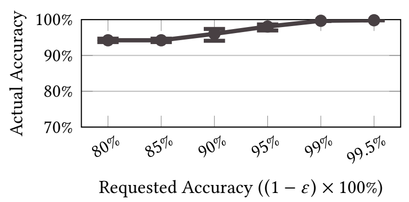
(a) LR, HIGGS



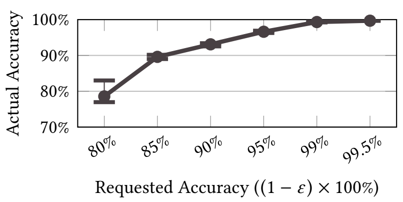
(b) ME, MNIST



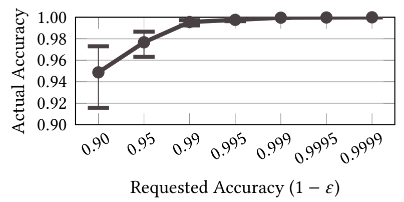
(c) PPCA, HIGGS



(d) LR, Yelp



(e) ME, Yelp



(f) PPCA, MNIST

Figure 6: The correctness of BlinkML's Mode A operations. The requested model accuracies are compared to the actual model accuracies. In most cases, the actual model accuracies were higher than the requested accuracies. The raw data of this figure are in Table 4.

($(1-\varepsilon) \times 100\%$) from 80% to 99.5%. For PPCA, we varied the requested accuracy ($1-\varepsilon$) from 0.90 to 0.9999. We fixed δ at 0.05. For each case, we repeated BlinkML's training 20 times.

Figure 5 shows the average training time of those 20 runs (for each case). The figure reports BlinkML's relative training time in comparison to the full model training time; for instance, 1% relative training time indicates that the training time was reduced by 99%, or equivalently, BlinkML was 100× faster. In the experiment, the full model training times were 1,040 seconds for (LR, HIGGS), 404 seconds for (LR, Yelp), 4,304 seconds for (ME, MNIST), 1,176 seconds for (ME, Yelp), 54 seconds for (PPCA, HIGGS), and 61 seconds for (PPCA, MNIST). Full models were all trained using Spark (except for PPCA as stated in Section 6.1). Using Spark made full model training 1.5×–6.5× faster. BlinkML selectively used Spark only when the samples (D_n) were not smaller than one million (i.e., $n \geq 1\text{M}$), according to its default configuration (Section 3).

Three patterns were observed. First, BlinkML took relatively longer for training a more accurate approximate model. This is because BlinkML's Sample Size Estimator correctly estimated that a larger sample was needed to satisfy the requested accuracy. Second, the relative training times were longer for complex models (ME was longer than LR). This is because multi-class classification (by ME) involved more possible class labels; thus, relatively small errors in parameter values could lead to misclassification. Thus, a larger sample was necessary to sufficiently upper-bound the chance of misclassification. Nevertheless, training 95% accurate ME models on MNIST and Yelp still took only 11.16% and 8.25% relative times, respectively. Third, training approximate PPCA models was much faster compared to LR and ME. This is the opposite case to ME. Unlike LR and ME, PPCA does not involve any hard decision boundaries; thus, the minimum sample sizes estimated by BlinkML's Sample Size Estimator were even smaller (see Section 6.4 for details). As a result, the relative training times were lower than LR and ME. In all cases, BlinkML's ability to automatically infer appropriate sample sizes and train approximate models on those samples led to significant training time savings. In the subsequent section, we analyze the actual accuracies of those approximate models.

6.3 Accuracy Guarantee (Mode A)

In this section, we validate the guaranteed accuracies associated with approximate models. For this, we compare the requested accuracies and the actual accuracies of the approximate models trained in Section 6.2. The actual accuracies were computed using the actual full models (which were, of course, not trained or used for training approximate models). According to BlinkML's guarantee, the actual accuracies must be not lower than their associated requested accuracies.

Figure 6 reports the 5th percentile, the mean, the 95th percentile of the actual accuracies for each combination of (LR, HIGGS), (LR, Yelp), (ME, MNIST), (ME, Yelp), (PPCA, HIGGS), and (PPCA, MNIST). When the requested accuracies were high, e.g., 99% or 99.5%, the associated actual accuracies are hard to read from the plots. To complement this, we report the raw numbers in Table 4. In general, the actual accuracies were higher than the associated requested accuracies, which indicates the correct operations of BlinkML.

One phenomenon that might seem unusual, is that in some cases, such as (LR, HIGGS), (LR, Yelp), and (ME, MNIST), the actual accu-

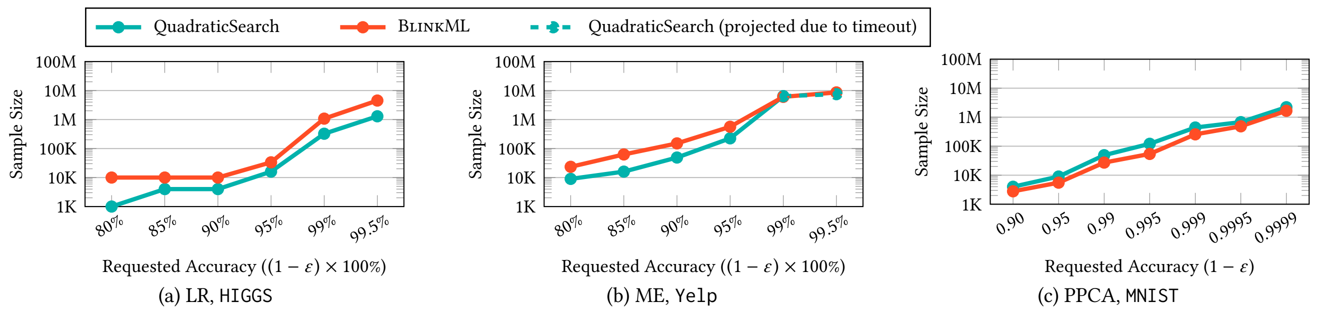


Figure 7: Goodness of BlinkML's estimated minimum sample sizes (used in Mode 1). BlinkML's estimated minimum sample sizes are compared to the minimum sample sizes determined by QuadraticSearch. The minimum sample sizes estimated by BlinkML were close to the ones determined by QuadraticSearch, a slow incremental searching technique.

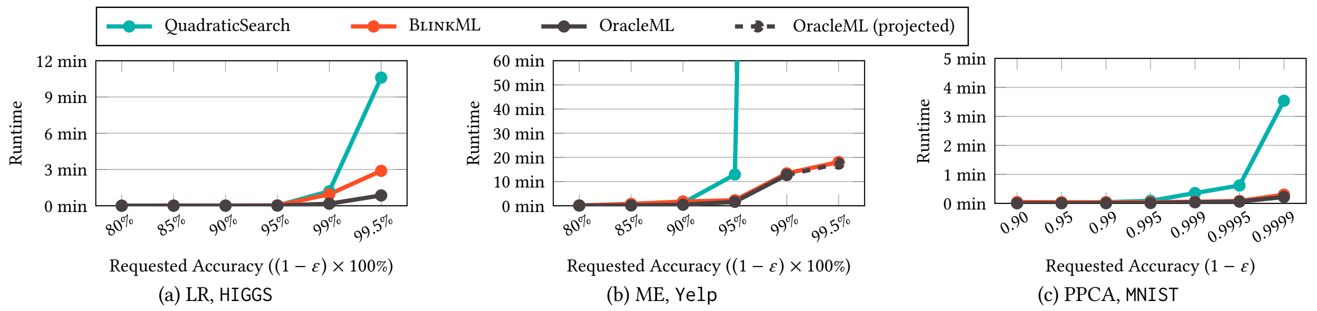


Figure 8: Optimality of BlinkML's end-to-end performance (of Mode 1). BlinkML's training time was compared to QuadraticSearch and OracleML. OracleML is assumed to be able to predict the minimum sample size without any runtime overhead.

racies were almost identical even though the requested accuracies were different. This is actually a natural phenomenon resulting from how BlinkML was designed (in Section 3). Recall that BlinkML first trains an initial model m_0; and subsequently trains a final model only when the estimated model difference ε_0 of m_0 is higher than the requested error ε. For those few cases, the initial models were already accurate enough; thus, no additional (final) models were trained. As a result, the actual accuracies of some approximate models did not vary for those cases. Also, this implies that the actual accuracies of those initial models had to be higher than the requested accuracies, which are reflected in the figure.

6.4 Sample Size Estimation (Mode A)

Sample Size Estimator (SSE)'s operation—estimating the minimum sample size—is crucial for BlinkML. Too a large sample harms training time savings; however, too a small sample breaks the accuracy guarantees. In this section, we closely examine SSE's operations. Our examination consists of two parts. In the first part, we analyze the optimality of the estimated minimum sample sizes. In the second part, we study the implication of the estimated sample sizes, i.e., the optimality of BlinkML's training time.

To analyze the optimality of the minimum sample sizes, we additionally implemented QuadraticSearch, a baseline algorithm that estimates the minimum sample size. To estimate the minimum sample size, QuadraticSearch gradually increases a candidate sample size until the approximate model trained on that candidate sample satisfies a requested accuracy. The candidate sample size was set to $k^2 \times 1,000$ where k is the iteration count starting from 1. The accuracies of the trained approximate models were assessed using Model Accuracy Assessor, a component of BlinkML. Unlike SSE, the quality of QuadraticSearch's minimum sample size is guaranteed: if the algorithm returns the minimum sample size at the k-th iteration, the relative error of the minimum sample size is less than $2/k$ compared to optimal (for $k \geq 2$).

Figure 7 compares the minimum sample sizes estimated by SSE (BlinkML's technique) and QuadraticSearch for three different combinations of a model and a dataset, i.e., (LR, HIGGS), (ME, Yelp), and (PPCA, MNIST), and for different levels of accuracies. In one case (ME, 99.5% accuracy), QuadraticSearch did not terminate within 24 hours; thus, we stopped its process. For LR and ME, QuadraticSearch produced slightly smaller minimum sample size estimates. For PPCA, QuadraticSearch and SSE were comparable. However, in general, their estimates were close, which implies that the qualities of the minimum sample sizes estimated by SSE were also close to optimal.

To study the impact of those differences in the sample size estimates (by SSE and QuadraticSearch), we analyzed the training time difference. Since QuadraticSearch is slow due to its iterative process, we also additionally implemented OracleML, an impossibly-good algorithm for training an accuracy-guaranteed approximate model. OracleML simply knows the minimum sample size returned by QuadraticSearch in advance, and directly trains an approximate model on that sample size. Although OracleML cannot be used in practice, comparing its training time to that of BlinkML helps us understand how far BlinkML's performance is from optimal.

Figure 8 shows the results. The figure shows that QuadraticSearch was considerably slow when accurate approximate models were needed; thus, QuadraticSearch would not be useful in general. Also, the figure shows that in most cases, the overall training time of BlinkML was not much longer than OracleML. Of course, there will always be small runtime overhead for BlinkML because it has to estimate the minimum sample size. We analyze the runtime overhead more closely in Section 6.6.

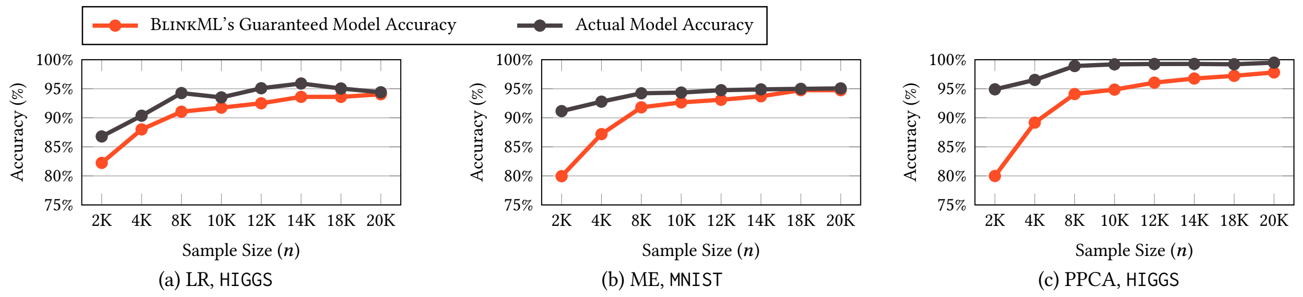


Figure 9: The correctness of BlinkML's model accuracy assessment. The assessed (i.e., guaranteed) model accuracies are compared to the actual model accuracies. In all cases, the actual model accuracies were higher than the guaranteed accuracies.

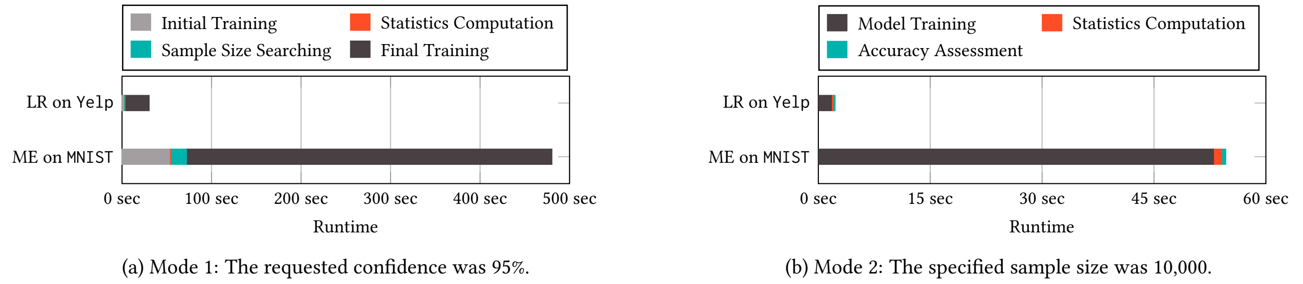


Figure 10: BlinkML's training time decomposition for both Mode 1 and Mode 2. BlinkML's runtime overhead for computing necessary statistics, searching minimum sample sizes, and estimating confidence were small fractions.

6.5 Accuracy Assessment (Modes A and B)

Model Accuracy Assessor (MAA) is the component used at the initial step of BlinkML's Mode A operations and is also the primary component for BlinkML's Mode B operations. This section studies the correctness of MAA's outputs—the estimated accuracy of an approximate model.

To study MAA's correctness, we compared MAA's guaranteed accuracy of an approximate model and the actual accuracy of the approximate model. Our comparisons were made for three different combinations of a model and a dataset, i.e., (LR, HIGGS), (ME, MNIST), and (PPCA, HIGGS); and for different sample sizes of training sets (on which approximate models were trained).

Figure 9 shows our experiment results. Observe that the gap between the actual accuracies and the guaranteed accuracies were higher when the sample sizes were relatively small. This is because when sample sizes were small, the variances of approximate models (or accurately, the variances of the outputs of approximate models) were estimated high; thus, MAA was more conservative to ensure the correctness of its outputs. The gap decreased as the sample sizes increased. In all cases, however, the actual accuracies of approximate models were higher than MAA's guaranteed accuracies, which indicates the correctness of MAA.

6.6 Runtime Analysis (Modes A and B)

This section analyzes BlinkML's runtime overhead, i.e., the time spent for the operations other than model training. Specifically, for Mode A, we measured the latencies of statistics computation (Section 4.3) and sample size searching (Section 5.2), and for Mode B, we measured the latencies of statistics computation (Section 4.3) and accuracy assessment (Section 4.2). Recall that Mode A always involves the accuracy assessment of an initial model; in our measurements, this latency was included in the latency of sample size searching.

For our analysis of Mode A, we let BlinkML train 95% accurate approximate models for two different combinations of a model and a dataset: (LR, Yelp) and (ME, MNIST). Figure 10a reports BlinkML's overhead. The gray and black bars are the times spent for model training, while the red and green bars are BlinkML's overhead. The majority of BlinkML's overhead was for sample size searching. However, the overall overhead was still a small proportion of the entire training for both cases. For LR and ME, BlinkML's overheads were 5.2% and 4.0% of the overall training, respectively.

For analyzing Mode B, we let BlinkML train approximate models using samples of size 10,000. The combinations of model and dataset were identical to Mode A. BlinkML's overhead for Mode B is shown in Figure 10b. When the entire training time was short (i.e., 2.25 secs for LR, Yelp), BlinkML's overhead took a significant portion (i.e., 22.4%); however, its absolute time was still short (0.50 secs). When the entire training time was longer (i.e., 54.6 secs for ME, MNIST); the proportion of BlinkML's overhead was only 3.0% (i.e., 1.64 secs). In all our analysis, BlinkML's runtime overhead was still low enough to bring sufficient training time savings in comparison to full model training.

6.7 Statistics Computation (Modes A and B)

In Section 4.3, we presented two general methods, InverseGradients and ObservedFisher, for computing important statistics, i.e., H and J. In this section, we compare their efficiencies and accuracies. This analysis serves as a reason that ObservedFisher is BlinkML's default choice.

Our analysis used two combinations of a model and a dataset, (LR, HIGGS) and (ME, MNIST). Recall from Table 1 that HIGGS is low-dimensional data ($d = 28$) while MNIST is high-dimensional data ($d = 784$). The difference in dimensions reveals interesting properties of InverseGradients and ObservedFisher. We used those two methods to compute the covariance matrix $H^{-1}JH^{-1}$ (which determines the parameter distribution in Theorem 1, Section 4). We measured the runtime of those methods and calculated the

Model, Dataset	Metric	Method	
		InverseGradients	ObservedFisher
LR, `HIGGS`	Runtime (sec)	1.88	1.18
	Accuracy ($\|\cdot\|_F$)	0.00332	0.0039
ME, `MNIST`	Runtime (sec)	357.0	3.23
	Accuracy ($\|\cdot\|_F$)	0.01298	0.00847

Table 2: Comparison of statistics computation techniques: InverseGradients and ObservedFisher. The accuracies of computed statistics were measured using the mean Frobenius norm of the difference between the true statistics and the estimated statistics. ObservedFisher was faster for `MNIST`, a high-dimensional dataset, while maintaining comparable accuracy to InverseGradients.

accuracies of the computed covariance matrices. The accuracies were measured by calculating the average Frobenius norm, i.e., $(1/d^2)\ \|C_t - C_e\|_F$ where C_t is the true covariance matrix and C_e is the estimated covariance matrix (by either of the two methods). This accuracy metric captures the average difference between the elements of C_t and C_e.

Table 1 summarizes the results of our analysis. For the low-dimensional data (`HIGGS`), the runtime and the accuracies of both methods were comparable. However, their performance differed much for the high-dimensional data (`MNIST`). Since InverseGradients must invoke the `grads` function repeatedly (i.e., d times), its runtime significantly increased. In contrast, ObservedFisher always calls the `grads` function only once; thus, the impact on its runtime was less significant. The accuracies of them were comparable also for the high-dimensional data.

7. RELATED WORK

In this section, we overview the related work to BlinkML. First, we introduce the ML systems inspired by familiar database techniques (DBMS-Inspired Optimization). In this first part, we only focus on the systems without approximation. Second, we present the ML systems employing different forms of approximation and discuss the major differences between BlinkML and those existing systems (Approximate ML Systems). Finally, we briefly overview recently developed optimization algorithms to highlight those developments are orthogonal to ML systems (Faster Optimization Algorithms). However, ML systems can still benefit from those developments by employing them.

DBMS-Inspired Optimization The success of database systems are largely attributed to the development of a declarative language (SQL) and accompanying opportunities for automatic optimizations. The database community has applied these ideas to speeding up ML workloads. SystemML [15, 28] optimizes linear algebra expressions by converting them into alternative expressions that enable more efficient evaluations. ScalOps [57], Pig latin [45], KeystoneML [53] propose high-level ML languages for automatic parallel iterations, automatic materialization, easier programming, etc. SystemML also proposes a new data compression algorithm for array data [25]. MADLib [18, 31] proposes a declarative language for ML and SQL-based parallel processing on top of existing database systems. Since MADLib exploits the existing database engines, the computations are close to the data, which leads to higher performance compared to other external ML libraries. RIOT [64] is an R interface that exploits in-database computing.

The database community has also developed a database system specialized for array data [33, 54]. In contrast to typical ML libraries, these systems can train a model even when the data does not fit in memory. MADLib shares the same benefit as it relies on database engines for large-scale parallel computation. The database community also observed that large-scale data are typically stored after normalization; thus, ML algorithms that exploit this normalization can bring increased performance [37, 50].

There are also ML systems that automatically choose an optimal optimization method depending on a given problem [35, 36, 46, 52]. For example, Hemingway [46], MLBase [35], and TuPAQ [52] automatically select an appropriate optimization algorithm for a given problem. Kumar et al. [36] proposes a model selection management system, which unifies feature engineering [11], algorithm selection, and parameter tuning.

Approximate ML Systems BlinkML is the first sampling-based ML training system applicable to a wide-class of ML methods. However, different types of approximation techniques have also been explored in the database community for speeding up ML training and prediction performance.

Hamlet [38] develops a rule for determining whether it is necessary to join dimension tables to a fact table. It avoids expensive denormalization if the estimated quality gain from join is low. Columbus [61] trains approximate models using a coreset, a representative subset of training data. Although Columbus also uses a sample of data for speeding up ML training, its approach offers quality guarantees only for the least squares problems including linear operators. In contrast, BlinkML supports much wider class of problems; in fact, the classification methods and the factor analysis technique used in our experiments all include non-linear objective functions, whose qualities cannot be guaranteed by Columbus. Zombie [12] employs clustering-based active learning technique for training approximate models; however, it does not offer quality guarantees on the trained models.

There are several methods that aim to speed up prediction at the cost of increased preprocessing. NoScope [32] first trains multiple deep neural networks; then, composes a different cascade depending on the target accuracy. tKDC [27] first builds a k-d tree; then, conducts early pruning at the prediction stage of kernel-density classifier. BlinkML differs from these methods since it focuses on speeding up model training process, which is a crucial bottleneck in the early stage of model developments.

Faster Optimization Algorithms The developments of optimization algorithms are largely orthogonal to ML systems; however, understanding the benefits of different optimization methods is necessary for developing fast ML systems. There are two classes of approaches in speeding up optimization algorithms: software and hardware.

We first overview software-based optimization improvements. The optimization methods that belong to gradient descent (e.g., SGD) perform optimization by gradually progressing toward an optimal point. Here, the key question is how fast we have to proceed toward the optimal point. Recent developments propose different adaptive rules for accelerating the convergence. Those works include Adagrad [23], Adadelta [60], RMSprop [55], and Adam [34]. Also, there is work that rediscovers the benefits of quasi-Newton optimization methods. Le et al. [39] shows that minibatch variants of quasi-Newton methods (e.g., L-BFGS or CG) can be superior to SGD due to its high parallelism. Hogwild! [49] and related techniques [29, 30, 63] disable locks to speed up asynchronously updates by SGD.

Next, we overview hardware-based optimization improvements. Hardware-based optimization techniques aim to speed up ML training by relaxing strict requirements imposed for traditional workloads. DimmWitted [62] and BuckWild! [21] achieve faster training by relaxing computational correctness.

8. CONCLUSION

In this work, we proposed BlinkML, a novel approximate machine learning system with probabilistic guarantees. BlinkML quickly trains an approximate ML model according to the user-supplied requirements, such as an accuracy guarantee or the size of a sample to use for training. BlinkML's techniques are applicable to a wide class ML models that rely on maximum likelihood estimation for their training; they include Generalized Linear Models (e.g., linear regression, logistic regression (LR), max entropy classifier (ME), Poisson regression, etc.) and Probabilistic Principal Component Analysis (PPCA). In our experiments with large-scale real-world datasets, BlinkML produced 99% accurate models of LR, ME, and PPCA after spending only 0.31%–64.97% of the full model training time. Our experiments also showed that albeit at preliminary stage, BlinkML's performance is close to optimal (if simple random sampling is used for approximate machine learning). Our future goal is to extend BlinkML's generality to currently untapped models, such as decision trees, k-nearest neighbor classifiers, Gaussian Process, Naïve Bayes classifiers, and neural networks. For this, the developments of new techniques will be needed.

9. REFERENCES

[1] Conda. https://conda.io/docs/index.html. Retrieved: April 25, 2018.

[2] cx_oracle version 6.2. https://oracle.github.io/python-cx_Oracle/. Retrieved: Mar 26, 2018.

[3] ibmdbr: Ibm in-database analytics for r. https://cran.r-project.org/web/packages/ibmdbR/index.html. Retrieved: Mar 26, 2018.

[4] The infinite mnist dataset. http://leon.bottou.org/projects/infimnist. Retrieved: April 25, 2018.

[5] Python sql driver - pymssql. https://docs.microsoft.com/en-us/sql/connect/python/pymssql/python-sql-driver-pymssql. Retrieved: Mar 26, 2018.

[6] Python support for ibm db2 and ibm informix. https://github.com/ibmdb/python-ibmdb. Retrieved: Mar 26, 2018.

[7] R interface to oracle data mining. http://www.oracle.com/technetwork/database/options/odm/odm-r-integration-089013.html. Retrieved: Mar 26, 2018.

[8] Revoscaler. https://docs.microsoft.com/en-us/sql/advanced-analytics/r/revoscaler-overview. Retrieved: Mar 26, 2018.

[9] Yelp dataset. https://www.kaggle.com/yelp-dataset/yelp-dataset. Retrieved: April 25, 2018.

[10] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *SIGMOD*, 2014.

[11] M. R. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. J. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A data system for feature engineering. In *CIDR*, 2013.

[12] M. R. Anderson and M. Cafarella. Input selection for fast feature engineering. In *ICDE*, pages 577–588, 2016.

[13] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 2014.

[14] C. M. Bishop. *Pattern recognition and machine learning*. 2006.

[15] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, et al. Systemml: Declarative machine learning on spark. *PVLDB*, pages 1425–1436, 2016.

[16] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *VLDB*, September 2000.

[17] B.-Y. Chu, C.-H. Ho, C.-H. Tsai, C.-Y. Lin, and C.-J. Lin. Warm start for parameter selection of linear classifiers. In *KDD*, 2015.

[18] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. Mad skills: new analysis practices for big data. *PVLDB*, pages 1481–1492, 2009.

[19] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, 2017.

[20] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: Interactive analytics through pen and touch. *PVLDB*, pages 2024–2027, 2015.

[21] C. De Sa, M. Feldman, C. Ré, and K. Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 561–574. ACM, 2017.

[22] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo-charging estimate convergence in dbo. *PVLDB*, 2009.

[23] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[24] B. Efron and D. V. Hinkley. Assessing the accuracy of the maximum likelihood estimator: Observed versus expected fisher information. *Biometrika*, pages 457–483, 1978.

[25] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed linear algebra for large-scale machine learning. *PVLDB*, pages 960–971, 2016.

[26] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting reuse for approximate query processing. *PVLDB*, pages 1142–1153, 2017.

[27] E. Gan and P. Bailis. Scalable kernel density classification via threshold-based pruning. In *SIGMOD*, pages 945–959, 2017.

[28] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[29] J. E. Gonzalez, P. Bailis, M. I. Jordan, M. J. Franklin, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Asynchronous complex analytics in a distributed dataflow architecture. *arXiv preprint arXiv:1510.07092*, 2015.

[30] S. Hadjis, C. Zhang, I. Mitliagkas, D. Iter, and C. Ré. Omnivore: An optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*, 2016.

[31] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *PVLDB*, pages 1700–1711, 2012.

[32] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *PVLDB*, pages 1586–1597, 2017.

[33] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes. Sciql, a query language for science applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 1–12, 2011.

[34] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[35] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[36] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Record*, pages 17–22, 2016.

[37] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984, 2015.

[38] A. Kumar, J. Naughton, J. M. Patel, and X. Zhu. To join or not to join?: Thinking twice about joins before feature selection. In *Proceedings of the 2016 International Conference on Management of Data*, pages 19–34. ACM, 2016.

[39] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng. On optimization methods for deep learning. In *ICML*, pages 265–272, 2011.

[40] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016.

[41] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54, 2011.

[42] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[43] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Eng. Bull.*, 2015.

[44] W. K. Newey and D. McFadden. Large sample estimation and hypothesis testing. *Handbook of econometrics*, pages 2111–2245, 1994.

[45] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[46] X. Pan, S. Venkataraman, Z. Tai, and J. Gonzalez. Hemingway: modeling distributed optimization algorithms. *arXiv*, 2017.

[47] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 4, 2011.

[48] Y. Park, A. S. Tajik, M. Cafarella, and B. Mozafari. Database Learning: Towards a database that becomes smarter every time. In *SIGMOD*, 2017.

[49] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[50] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, pages 3–18, 2016.

[51] J. Shin, S. Wu, F. Wang, C. De Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using deepdive. *PVLDB*, pages 1310–1321, 2015.

[52] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *SoCC*, pages

368–380, 2015.

[53] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. KeystoneML: Optimizing pipelines for large-scale advanced analytics. In *ICDE*, pages 535–546, 2017.

[54] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. Scidb: A database management system for applications with complex analytics. *Computing in Science & Engineering*, pages 54–62, 2013.

[55] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. *University of Toronto, Tech. Rep*, 2012.

[56] M. E. Tipping and C. M. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 1999.

[57] M. Weimer, T. Condie, R. Ramakrishnan, et al. Machine learning in scalops, a higher order cloud computing language. In *BigLearn*, pages 389–396, 2011.

[58] E. Xing. Vc dimension and model complexity. https://www.cs.cmu.edu/~epxing/Class/10701/slides/lecture16-VC.pdf. Retrieved: April 25, 2018.

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

[60] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[61] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD*, pages 265–276, 2014.

[62] C. Zhang and C. Ré. Dimmwitted: A study of main-memory statistical analytics. *Proceedings of the VLDB Endowment*, 7(12):1283–1294, 2014.

[63] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu. Asynchronous stochastic gradient descent for dnn training. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6660–6663. IEEE, 2013.

[64] Y. Zhang, W. Zhang, and J. Yang. I/o-efficient statistical computing with riot. In *ICDE*, pages 1157–1160, 2010.

APPENDIX

A. MODEL ABSTRACTION EXAMPLES

This section shows that BlinkML-supported ML models can be cast into the abstract form in Section 3.2, Section 3. For illustration, we use logistic regression and PPCA.

Logistic Regression The objective function of logistic regression captures the difference between true class labels and the predicative class labels. An optional regularization term may be placed to prevent a model to be overfitted to a training set. For instance, the objective function of L2-regularized logistic regression is expressed as follows:

$$f_n(\theta) = -\left[\frac{1}{n}\sum_{i=1}^{n} t_i \log \sigma(\theta^\top \boldsymbol{x}_i) + (1-t_i)\log \sigma(1-\theta^\top \boldsymbol{x}_i)\right] + \frac{\beta}{2}\|\theta\|^2$$

where $\sigma(y) = 1/(1+\exp(y))$ is a sigmoid function, and β is the coefficient that controls the strength of the regularization penalty. The observed class labels, i.e., t_i for $i = 1, \ldots, n$, are either 0 or 1. The above expression is minimized when θ is set to the value at which the gradient $\nabla f_n(\theta)$ of $f_n(\theta)$ becomes a zero vector; that is,

$$\nabla f_n(\theta) = \left[\frac{1}{n}\sum_{i=1}^{n}(\sigma(\theta^\top \boldsymbol{x}_i) - t_i)\,\boldsymbol{x}_i\right] + \beta\,\theta = \mathbf{0}$$

It is straightforward to cast the above expression into Equation (2). That is, $q(\theta; \boldsymbol{x}_i, t_i) = (\sigma(\theta^\top \boldsymbol{x}_i) - t_i)\,\boldsymbol{x}_i$ and $r(\theta) = \beta\,\theta$.

PPCA The objective function of PPCA captures the difference between the covariance matrix S of the training set and the covariance matrix $C = \Theta\,\Theta^\top + \sigma^2 I$ reconstructed from the q number of extracted factors Θ, as follows:

$$f_n(\Theta) = \frac{1}{2}\left(d \log 2\pi + \log|C| + \mathrm{tr}(C^{-1}S)\right)$$

where d is the dimension of feature vectors. Θ is d-by-q matrix in which each column represents a factor, and σ is a real-valued scalar that represents the noise in data not explained by those factors. The optimal value for σ can be obtained once the values for Θ are determined. The value of q, or the number of the factors to extract, is a user parameter. The above expression $f_n(\Theta)$ is minimized when its gradient $\nabla_\Theta f_n(\Theta)$ becomes a zero vector; that is,

$$\nabla_\Theta f(\Theta) = C^{-1}(\Theta - S\,C^{-1}\Theta) = \mathbf{0}$$

The above expression can be cast into the form in Equation (2) by observing $S = (1/n)\sum_{i=1}^{n} \boldsymbol{x}_i \boldsymbol{x}_i^\top$.[6] That is, $q(\Theta; \boldsymbol{x}_i) = C^{-1}\Theta - C^{-1}\boldsymbol{x}_i\boldsymbol{x}_i^\top C^{-1}\Theta$ and $r(\Theta) = 0$. t_i is omitted on purpose since PPCA does not need observations (e.g., class labels). Although we used a matrix form Θ in the above expression for simplicity, BlinkML internally uses a vector θ when passing parameters among components. The vector is simply flattened and unflattened as needed.

[6] This sample covariance expression assumes that the training set is zero-centered.

LR, `HIGGS`			**LR, `Yelp`**		
Requested Accuracy	**Training Time**	**Ratio**	**Requested Accuracy**	**Training Time**	**Ratio**
80.0%	0.30 secs	0.01%	80.0%	1.1 secs	0.27%
85.0%	0.30 secs	0.01%	85.0%	1.08 secs	0.27%
90.0%	0.30 secs	0.01%	90.0%	13.41 secs	3.32%
95.0%	2.44 secs	0.05%	95.0%	30.25 secs	7.49%
99.0%	122.00 secs	2.43%	99.0%	138.00 secs	34.16%
99.5%	403.00 secs	8.02%	99.5%	269.00 secs	66.58%

ME, `MNIST`			**ME, `Yelp`**		
Requested Accuracy	**Training Time**	**Ratio**	**Requested Accuracy**	**Training Time**	**Ratio**
80.0%	54.38 secs	1.26%	80.0%	7.01 secs	0.60%
85.0%	54.71 secs	1.27%	85.0%	45.72 secs	3.89%
90.0%	53.17 secs	1.24%	90.0%	104.5 secs	8.89%
95.0%	480.2 secs	11.16%	95.0%	97.00 secs	8.25%
99.0%	2,578 secs	59.90%	99.0%	764 secs	64.97%
99.5%	3,457 secs	80.32%	99.5%	1,045 secs	88.86%

PPCA, `HIGGS`			**PPCA, `MNIST`**		
Requested Accuracy	**Training Time**	**Ratio**	**Requested Accuracy**	**Training Time**	**Ratio**
0.9	0.189 secs	0.35%	0.9	2.764	4.53%
0.95	0.165 secs	0.31%	0.95	2.457	4.03%
0.99	0.168 secs	0.31%	0.99	2.266	3.72%
0.995	0.172 secs	0.32%	0.995	2.419	3.97%
0.999	0.208 secs	0.38%	0.999	3.540	5.80%
0.9995	0.309 secs	0.57%	0.9995	4.894	8.02%
0.9999	0.632 secs	1.17%	0.9999	18.087	29.65%

Table 3: Training time savings. This is the raw data for Figure 5.

	LR, `HIGGS`				**LR, `Yelp`**		
Requested Accuracy	**Actual Accuracy**			**Requested Accuracy**	**Actual Accuracy**		
	Mean	5th Percentile	95th Percentile		Mean	5th Percentile	95th Percentile
80.0%	94.09%	93.35%	95.13%	80.0%	94.23%	93.73%	94.68%
85.0%	94.09%	93.35%	95.13%	85.0%	94.23%	93.73%	94.68%
90.0%	94.09%	93.35%	95.13%	90.0%	96.03%	94.09%	97.39%
95.0%	96.72%	93.94%	97.67%	95.0%	98.06%	96.97%	98.65%
99.0%	99.37%	98.80%	99.63%	99.0%	99.67%	99.61%	99.78%
99.5%	99.70%	99.61%	99.82%	99.5%	99.82%	99.79%	99.87%

	ME, `MNIST`				**ME, `Yelp`**		
Requested Accuracy	**Actual Accuracy**			**Requested Accuracy**	**Actual Accuracy**		
	Mean	5th Percentile	95th Percentile		Mean	5th Percentile	95th Percentile
80.0%	95.94%	95.73%	96.18%	80.0%	78.56%	76.96%	83.01%
85.0%	95.94%	95.73%	96.18%	85.0%	89.60%	89.00%	90.17%
90.0%	95.94%	95.73%	96.18%	90.0%	93.10%	92.53%	93.41%
95.0%	98.62%	98.49%	98.76%	95.0%	96.61%	96.27%	96.90%
99.0%	99.76%	99.70%	99.81%	99.0%	99.32%	99.22%	99.41%
99.5%	99.90%	99.87%	99.93%	99.5%	99.69%	99.65%	99.75%

	PPCA, `HIGGS`				**PPCA, `MNIST`**		
Requested Accuracy	**Actual Accuracy**			**Requested Accuracy**	**Actual Accuracy**		
	Mean	5th Percentile	95th Percentile		Mean	5th Percentile	95th Percentile
0.9	0.98146	0.96499	0.99312	0.9	0.94874	0.91582	0.97298
0.95	0.98887	0.97428	0.99630	0.95	0.97668	0.96314	0.98660
0.99	0.99799	0.99569	0.99943	0.99	0.99556	0.99243	0.99729
0.995	0.99904	0.99758	0.99969	0.995	0.99756	0.99635	0.99866
0.999	0.99982	0.99967	0.99994	0.999	0.99958	0.99935	0.99979
0.9995	0.99991	0.99982	0.99996	0.9995	0.99977	0.99970	0.99985
0.9999	0.99998	0.99996	0.99999	0.9999	0.99995	0.99992	0.99998

Table 4: The comparison of requested model accuracies (in Mode A) to the the actual model accuracies. This is the raw data for Figure 6.