BlinkML: Efficient Maximum Likelihood Estimation with Probabilistic Guarantees

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

ABSTRACT

The rising volume of datasets has made training machine learning (ML) models a major computational cost in the enterprise. Given the *iterative* nature of model and parameter tuning, many analysts use a small sample of their entire data during their *initial* stage of analysis to make quick decisions (e.g., what features or hyperparameters to use) and use the entire dataset only in later stages (i.e., when they have converged to a specific model). This sampling, however, is performed in an ad-hoc fashion. Most practitioners cannot precisely capture the effect of sampling on the quality of their model, and eventually on their decision-making process during the tuning phase. Moreover, without systematic support for sampling operators, many optimizations and reuse opportunities are lost.

In this paper, we introduce BlinkML, a system for *fast, quality-guaranteed ML training*. BlinkML allows users to make error-computation tradeoffs: instead of training a model on their full data (i.e., *full model*), BlinkML can quickly train an *approximate model* with quality guarantees using a sample. The quality guarantees ensure that, with high probability, the approximate model makes the same predictions as the full model. BlinkML currently supports any ML model that relies on *maximum likelihood estimation* (MLE), which includes Generalized Linear Models (e.g., linear regression, logistic regression, max entropy classifier, Poisson regression) as well as PPCA (Probabilistic Principal Component Analysis). Our experiments show that BlinkML can speed up the training of large-scale ML tasks by 6.26×–629× while guaranteeing the same predictions, with 95% probability, as the full model.

ACM Reference Format:
Yongjoo Park, Jingyi Qing, Xiaoyang Shen, Barzan Mozafari. 2019. BlinkML: Efficient Maximum Likelihood Estimation with Probabilistic Guarantees. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, Article 4, 18 pages. https://doi.org/10.1145/3299869.3300077

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300077

1 INTRODUCTION

While data management systems have been widely successful in supporting traditional OLAP-style analytics, they have *not* been equally 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 [4], SQL Server's pymssql [9], and DB2's ibm_db [10]) or R (e.g., Oracle's RODM [11], SQL Server's RevoScaleR [12], and DB2's ibmdbR [6]). 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 [15, 16, 31, 93, 106], cost-based optimization of linear algebraic operators [24, 29, 48], array-based representations [59, 96], avoiding denormalization [64, 65, 89], lazy evaluation [109], declarative interfaces [80, 95, 101], and query planning [63, 81, 94] 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) [27, 32, 35, 45, 54, 68, 73, 82, 84, 85]. 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.

Our Goal Given that (sub)sampling is already quite common in early stages of ML workloads—such as feature selection and hyper-parameter tuning—we propose a high-level system abstraction for training ML models, with which analysts can explicitly request error-computation trade-offs for several important classes of ML models. This involves systematic support for (i) bounding the deviation of the approximate model's predictions from those of the full model

given a sample size, and (ii) predicting the *minimum sample size* with which the trained model would meet a given prediction error.

Challenges While estimating sampling error is a well-studied problem for SQL queries [14, 75], it is more involved for ML models. There are two types of 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., *descriptive*). A well-known predictive technique is the so-called VC-dimension [74], 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 [103]. Overestimated error bounds would lead the analysts to use the entire dataset even if similar results could be obtained from a small sample. [1]

Common techniques for the descriptive approach include cross-validation [23] and Radamacher complexity [74]. 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 needed to achieve a user-specified accuracy, the descriptive 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 [28]), training multiple models to find an appropriate sample size might take longer overall than simply training a model on the full dataset (see Section 5.4).

Our Approach BlinkML's underlying statistical technique offers tight error bounds for approximate models (Section 3), and does so without training multiple approximate models (Section 4).

BlinkML's statistical techniques are based on the following observation: given a test example x, the model's prediction is simply a function $m(x;\theta)$, where θ is the model parameter learned during the training phase. Therefore, if we could understand how θ would differ when trained on a sample (instead of the entire dataset), we could also infer its impact on the model's prediction, i.e., $m(x;\theta)$.

Specifically, let θ_N be the model parameter obtained if one trains on the entire dataset (say, of size N), and θ_n be the model parameter obtained if one trains on a sample of size n. 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 $\hat{\theta}_n$ is the random variable for θ_n, and $\hat{\theta}_N$ represents our (limited) probabilistic knowledge of θ_N (Theorem 1 and Corollary 1). A specific model parameter θ_n trained on a specific sample (of size n) is an instance of $\hat{\theta}_n$ The asymptotic distribution of $\theta_N - \hat{\theta}_n$ is available for the ML methods relying on maximum likelihood estimation.

This indicates that, while we cannot determine the exact value of θ_N without training the full model, we can use $\hat{\theta}_N \mid \theta_n$ 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 3.3). Moreover, we can estimate the deviation of $m(x;\theta_N)$ from $m(x;\theta_n)$ for any other sample size, say n, using only the model trained on the initial sample of size n_0 (Section 4). In other words, without having to perform additional training, we can efficiently search for the minimum sample size n, with 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.

Difference from Previous Work Existing sampling-based techniques are typically designed for a very specific type of model, such as non-uniform sampling for linear regression [17, 22, 34, 36–38, 41, 47], logistic regression [62, 100], clustering [42, 56], kernel matrices [49, 76], Gaussian mixture models [70], and point processes [67]. In contrast, BlinkML exploits uniform random sampling for training a much wider class of models, i.e., any MLE-based model; thus, no sampling probabilities need to be determined in advance. BlinkML's contributions also include an efficient accuracy estimation for the approximate model and an accurate minimum sample size estimation for satisfying a user-requested accuracy. (See Sections 6 and 7 for discussions.)

Contributions We make the following contributions:

1. We introduce a system (called BlinkML) that offers error-computation trade-offs for training any MLE-based ML model, including Generalized Linear Models (e.g., linear regression, logistic regression, max entropy classifier, Poisson regression) and Probabilistic Principal Component Analysis. (Section 2)
2. We formally study the sampling distribution of an approximate model's parameters, which we use to design an efficient algorithm that computes the probabilistic difference between an approximate model and a full one, without having to train the latter. (Section 3)
3. We develop a technique that can analytically infer the quality of a new approximate model, only using a previous model and without having to train the new one. This ability enables BlinkML to automatically and efficiently

[1] This is why VC-dimensions are sometimes used indirectly, as a comparative measure of quality [65].

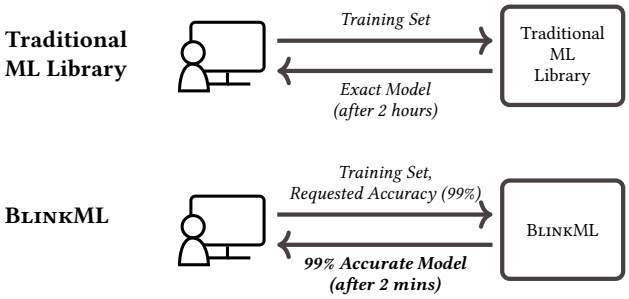


Figure 1: Interaction difference between traditional ML libraries and BlinkML. BlinkML can quickly train an approximate ML model in accordance to a user-specified accuracy request.

infer the appropriate sample size for satisfying an error tolerance requested by the user. (Section 4)

4. We empirically validate the statistical correctness and computational benefits of BlinkML through extensive experiments. (Section 5)

2 SYSTEM OVERVIEW

In this section, we provide an overview of BlinkML. We describe BlinkML's user interface in Section 2.1. In Section 2.2, we formally describe the models supported by BlinkML. We describe BlinkML's internal workflow in Section 2.3.

2.1 User Interface

In this section, we first describe the interface of a traditional ML library (e.g., scikit-learn [86], Weka [53], MLlib [1]), and then present the difference in BlinkML's interface. To simplify our presentation, here we focus on classification models; however, our description can be easily generalized to both regression models (e.g., linear regression) and unsupervised learning (e.g., PPCA), as described in Appendix C.

Traditional ML Libraries As depicted in Figure 1 (top), with a typical ML library, the user provides a *training set* $D \sim \mathcal{D}$ and specifies a *model class* (e.g., linear regression, logistic regression, PPCA) along with model-specific configurations (e.g., regularization coefficients for linear or logistic regression, the number of factors for PPCA). A training set D is a (multi-)set of N training examples, which we denote as $\{(x_1; y_1); \cdots; (x_N; y_N)\}$. The d-dimensional vector x_i is called a *feature vector*, and a real-valued y_i is called a *label*. Then, the traditional ML library outputs a model m_N trained on the given training set. We call m_N a *full model*.

In classification tasks, $m(x)$ predicts a class label for an unseen feature vector x. For example, if x encodes a review of a restaurant, a trained logistic regression classifier $m_N(x)$ may predict whether the review is positive or negative.

Table 1: Notations

Sym.	Meaning
N	the size of dataset
n	the size of a sample
D	the training set (drawn from a distribution $\mathcal{D}$)
D_n	a size-n random sample of D
m_N	the full model, which is trained on D
m_n	an approximate model, which is trained on D_n
θ_N	the parameter of the full model
θ_n	the parameter of an approximate model
$v(m_n)$	the probability that m_n makes a different prediction than m_N (for the test set)
ε	the error bound on $v(m_n)$
δ	the probability of error bound violation
n_0	the size of initial training set (10K by default)
D_0	a size-n_0 random sample of D
m_0	an initial model trained on D_0

BlinkML In addition to the inputs required by traditional ML libraries, BlinkML needs one extra input: an *approximation contract* that consists of an error bound ε and a confidence level δ. Then, BlinkML returns an *approximate model* m_n such that the prediction difference between m_n and m_N is within ε with probability at least $1 - \delta$. That is,

$$\Pr[v(m_n) \leq \varepsilon] \geq 1 - \delta$$
$$\text{where} \quad v(m_n) = E_{x \sim \mathcal{D}}(\mathbb{1}[m_n(x) \neq m_N(x)])$$

where the expectation is over a test set. To estimate the above probability, BlinkML uses a holdout set that is not used for training the approximate model. The approximate model m_n is trained on a sample of size n, where the value of n is automatically inferred by BlinkML.

The following lemma shows that BlinkML's accuracy guarantee also implies a probabilistic bound on the full model's generalization error.

Lemma 1. *Let ε_∂ be the generalization error of BlinkML's approximate model; that is, $\varepsilon_\partial = E_{(x;y)\sim\mathcal{D}}(\mathbb{1}[m_n(x) \neq y])$. Then, the full model's generalization error is bounded as:*

$$E_{(x;y)\sim\mathcal{D}}(\mathbb{1}[m_N(x) \neq y]) \leq \varepsilon_\partial + \varepsilon - \varepsilon_\partial \cdot \varepsilon$$

with probability at least $1 - \delta$.

We defer the proof to Appendix B. We also empirically confirm this result in Section 5.5.

2.2 Supported Models & Abstraction

Here, we formally describe BlinkML's supported models; then, we describe how BlinkML expresses the supported models in an abstract way.

Formal Description of Supported Models BlinkML supports any model that can be trained by solving the following

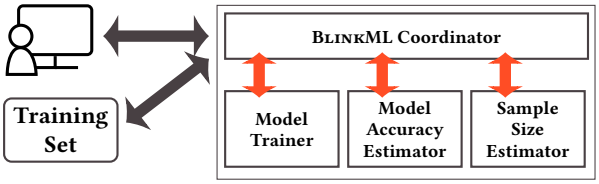


Figure 2: Architecture of BlinkML.

convex optimization problem:

$$\arg\min_\theta \quad f_n(\theta) \tag{1}$$

$$\text{where} \quad f_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} -\log \Pr(x_i; y_i; \theta) + R(\theta) \tag{2}$$

Here, $\Pr(x_i; y_i; \theta)$ indicates the likelihood of observing the pair (x_i, y_i) given θ, $R(\theta)$ is the optional regularization term typically used to prevent overfitting, and n is the number of training examples used. When n=N, this results in training the full model m_N, and otherwise we have an approximate model m_n. Different ML models use different expressions for $\Pr(x_i; y_i; \theta)$. We provide specific examples in Appendix A.

The solution θ_n to the minimization problem in Equation (1) is a value of θ at which the gradient $\partial_n(\theta) = \nabla f_n(\theta)$ of the objective function $f_n(\theta)$ becomes zero.[2] That is,

$$\partial_n(\theta_n) = \frac{1}{n}\sum_{i=1}^{n} q(\theta_n; x_i; y_i) + r(\theta_n) = 0 \tag{3}$$

where $q(\theta; x_i; y_i)$ denotes $-\nabla_\theta \log \Pr(x_i; y_i; \theta)$ and $r(\theta)$ denotes $\nabla_\theta R(\theta)$.

Examples of Supported Models BlinkML currently supports the following four model classes: linear regression, logistic regression, max entropy classifier, and PPCA. However, BlinkML's core technical contributions (Theorems 1 and 2 in Sections 3 and 4) can be generalized to any ML algorithms that rely on maximum likelihood estimation.

Model Abstraction A model class specification (MCS) is the abstraction that allows BlinkML's components to remain generic and not tied to the specific internal logic of the supported ML models. Each MCS must implement the following two methods:

1. **`diff`**(m_1, m_2): This function computes the prediction difference between two models m_1 and m_2, using part of the training set (i.e., holdout set) that was not used during model training.
2. **`grads`**: This function returns a list of $q(\theta; x_i; y_i) + r(\theta)$ for $i = 1; \cdots; n$, as defined in Equation (3). Although iterative optimization algorithms typically rely *only* on the *average* of this list of values (i.e., the gradient $\nabla_\theta f_n(\theta)$), the `grads` function must return individual values (without averaging them), as they are internally used by BlinkML (see Section 3.4).

BlinkML already includes the necessary MCS definitions for the currently supported model classes.

[2] In this work, we assume θ_n has fully converged to the optimal point, satisfying Equation (3). The non-fully converged cases can be handled by simply adding small error terms to the diagonal elements of J in Theorem 1.

2.3 System Workflow

We describe the workflow between BlinkML's components depicted in Figure 2. First, Coordinator obtains a size-n_0 sample D_0 of the training set D. We call D_0 the *initial training set* (10K by default). Coordinator then invokes Model Trainer to train an *initial model* m_0 on D_0, and subsequently invokes Model Accuracy Estimator to estimate the accuracy ε_0 of m_0 (with confidence $1-\delta$). If ε_0 is smaller than or equal to the user-requested error bound ε, Coordinator simply returns the initial model to the user. Otherwise, Coordinator prepares to train a second model, called the *final model* m_n. To determine the sample size n required for the final model to satisfy the error bound, Coordinator consults Sample Size Estimator to estimate the smallest n with which the model difference between m_n and m_N (i.e., the *unknown* full model) would not exceed ε with probability at least $1-\delta$. Note that this operation of Sample Size Estimator does not rely on Model Trainer; that is, no additional (approximate) models are trained for estimating n. Finally, Coordinator invokes Model Trainer (for a second time) to train on a sample of size n and return m_n to the user. Therefore, in the worst case, at most two approximate models are trained.

3 MODEL ACCURACY ESTIMATOR

Model Accuracy Estimator estimates the accuracy of an approximate model. That is, given an approximate model m_n and a confidence level δ, Model Accuracy Estimator computes ε, such that $v(m_n) \le \varepsilon$ with probability at least $1-\delta$. In Section 3.1, we first overview the process of Model Accuracy Estimator. In Section 3.2, we establish the statistical properties of the models supported by BlinkML. Then, in Section 3.3, we explain how BlinkML exploits these statistical properties to estimate the accuracy of an approximate model. Finally, Section 3.4 describes how to efficiently compute those statistics.

3.1 Accuracy Estimation Overview

To compute the probabilistic upper bound ε, Model Accuracy Estimator exploits the fact that both m_n and m_N are essentially the same function (i.e., $m(x)$) but with different model parameters (i.e., θ_n and θ_N). Although we cannot compute the exact value of θ_N without training the full model m_N, we can still estimate its probability distribution (we explain this in Section 3.2). The probability distribution of θ_N is then used to estimate the distribution of $v(m_n)$, which is the quantity

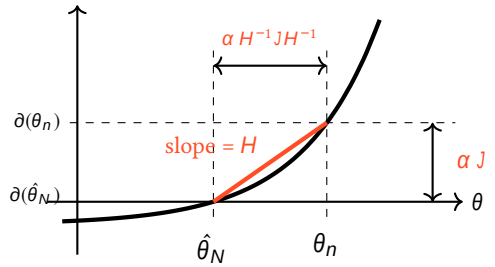


Figure 3: The variances $\alpha H^{-1}JH^{-1}$ of sampling-based model parameters are obtained using two model/data-aware quantities (i.e., αJ and H).

we aim to upper bound. The upper bound ε is determined by simply finding the value that is larger than $v(m_n)$ for $100 \cdot (1-\delta)\%$ of the holdout examples. (Section 3.3).

3.2 Model Parameter Distribution

In this section, we present how to probabilistically express the parameter θ_N of the (unknown) full model m_N given only the parameter θ_n of an approximate model m_n. Let $\hat{\theta}_n$ be a random variable representing the distribution of the approximate model's parameters; θ_n is simply one instance of $\hat{\theta}_n$. We also use $\hat{\theta}_N$ to represent our (limited) knowledge of θ_N. Then, our goal is to obtain the distribution of $\theta_n - \hat{\theta}_N$, and then use this distribution to estimate the prediction difference between $m(\theta_n)$ and $m(\hat{\theta}_N)$.

Intuition Since θ_n is the value that satisfies Equation (3) for n training examples (instead of N), we can obtain the difference αJ between $\partial_n(\theta_n)$ and $\partial_N(\hat{\theta}_N)$. In addition, we can obtain the relationship H between $\partial_n(\theta_n) - \partial_N(\hat{\theta}_N)$ and $\theta_n - \hat{\theta}_N$ using the Taylor expansion of $\partial_n(\theta)$. Then, we can finally derive the difference between θ_n and $\hat{\theta}_N$.

Figure 3 depicts this idea intuitively. In the figure, the slope H captures the surface of the gradient, and αJ captures the variance of the gradient; αJ decreases as n increases. Thus, if a model is more flexible (e.g., smaller regularization coefficients), the slope becomes more moderate (i.e., smaller elements in H), which leads to a larger distance between $\hat{\theta}_N$ and θ_n given the same αJ. In other words, the approximate model will be less accurate given a fixed sample size. Below, we formally present this idea. To account for these differences between models, BlinkML automatically adjusts its sample size n when it trains an approximate model to satisfy the requested error bound (Section 4).

Parameter Distribution The following theorem provides the distribution of $\hat{\theta}_n - \hat{\theta}_N$ (its proof is in Appendix B).

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

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

as $n \to \infty$ and $N \to \infty$. $\mathcal{N}$ denotes a normal distribution, which means that $\hat{\theta}_n - \hat{\theta}_N$ asymptotically follows a multivariate normal distribution with covariance matrix $\alpha H^{-1}JH^{-1}$.

Directly computing H and J requires $\Omega(d^2)$ space where d is the number of features. This can be prohibitively expensive when d is large. To address this, these quantities are indirectly computed (as described in Section 3.4 and Section 4.3), reducing the computational cost to only $O(d)$. Our empirical study shows that BlinkML can scale up to datasets with a million features (Section 5).

In Theorem 1, J is essentially the covariance matrix of gradients (computed on individual examples). Since BlinkML uses uniform random sampling, estimating J is simpler; however, even when non-uniform random sampling is used, J can still be estimated if we know the sampling probabilities. By assigning those sampling probabilities in a task-specific way, one could obtain higher accuracy, which we leave as future work.

The following corollary provides the conditional distribution of $\hat{\theta}_N \mid \theta_n$ (proof in Appendix B).

Corollary 1. *Without any* a priori *knowledge of 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 $n \to \infty$ and $N \to \infty$.

The following section uses the conditional distribution $\hat{\theta}_N \mid \theta_n$ (in Corollary 1) to obtain an error bound on the approximate model.

3.3 Error Bound on Approximate Model

In this section, we describe how Model Accuracy Estimator estimates the accuracy of an approximate model m_n. Specifically, we show how to estimate $v(m_n)$ without training m_N.

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 the error bound ε of an approximate model that holds with probability at least $1-\delta$. That is,

$$\Pr_{\theta_N}[v(m_n) \le \varepsilon] \ge \frac{1}{1} - \delta \qquad \text{where}$$

$$\Pr_{\theta_N}[v(m_n) \le \varepsilon] = \int \mathbb{1}[v(m_n;\theta_N) \le \varepsilon]\, h(\theta_N)\, d\theta_N; \quad (4)$$

$$m_n(x) = m(x;\, \theta_n); \quad m_N(x) = m(x;\, \theta_N \mid \theta_n)$$

where the integration is over the domain of θ_N ($\in \mathcal{R}^d$); $v(m_n;\theta_N)$ is the error of m_n when the full model's parameter

is θ_N; and $\mathbb{1}[\cdot]$ is the indicator function that returns 1 if its argument is true and returns 0 otherwise. Since the above expression involves the model's (blackbox) prediction function $m(x)$, it cannot be analytically computed in general.

To compute Equation (4), BlinkML's Model Accuracy Estimator uses the empirical distribution of $h(\theta_N)$ as follows. Let $\theta_{N;1}; \cdots; \theta_{N;k}$ be i.i.d. samples drawn from $h(\theta_N)$. Then,

$$\int \mathbb{1}[v(m_n; \theta_N) \le \varepsilon]\, h(\theta_N)\, d\theta_N \approx \frac{1}{k} \sum_{i=1}^{k} \mathbb{1}\ v(m_n; \theta_{N;i}) \le \varepsilon \tag{5}$$

To take into account the approximation error in Equation (5), BlinkML uses conservative estimates on ε as formally stated in the following lemma (see Appendix B for proof).

Lemma 2. *If* ε *satisfies*

$$\frac{1}{k} \sum_{i=1}^{k} \mathbb{1}\ v(m_n; \theta_{N;i}) \le \varepsilon \ = \frac{1-\delta}{0{:}95} + \sqrt{\frac{\log 0{:}95}{-2k}}$$

then $\Pr[v(m_n) \le \varepsilon] \ge 1 - \delta$:

The above lemma implies that by using a larger k (i.e., number of sampled values), we can obtain a tighter ε. To obtain a large k, an efficient sampling algorithm is necessary. Since $\hat{\theta}_N$ follows a normal distribution, one can simply use an existing library, such as **`numpy.random`**. However, BlinkML uses its own fast, custom sampler to avoid directly computing the covariance matrix $H^{-1} J H^{-1}$ (see Section 4.3).

3.4 Computing Necessary Statistics

We present three methods—(1) ClosedForm, (2) InverseGradients, and (3) ObservedFisher—for computing H. Given H, computing J is straightforward since $J = H - J_r$, where J_r is the Jacobian of $r(\theta)$. BlinkML uses ObservedFisher by default since it achieves high memory-efficiency by avoiding the direct computations of H.

Method 1: ClosedForm ClosedForm uses the analytic form of the Jacobian $H(\theta)$ of $\partial_n(\theta)$, and sets $\theta = \theta_n$ by the definition of H. For instance, $H(\theta)$ of L2-regularized logistic regression is expressed as follows:

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

where X is an n-by-d matrix whose i-th row is x_i, and Q is a d-by-d diagonal matrix whose i-th diagonal entry is $\sigma(\theta^\top x_i)(1 - \sigma(\theta^\top x_i))$, and β is the coefficient of L2 regularization. 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.

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

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

The values of $\partial_n(\theta_n + d\theta)$ and $\partial_n(\theta_n)$ are computed using the `grads` function provided by the MCS. 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 number (10^{-6} by default). Also, let R be the d-by-d matrix whose i-th column is $\partial_n(\theta_n + P_{:,i})$ where $P_{:,i}$ is the i-th column of P. Then, $H \approx R P^{-1}$.

Since InverseGradients only relies on the **`grads`** function, it is applicable to all supported models. Although InverseGradients is accurate, it is still computationally inefficient for high-dimensional data, since the **`grads`** function must be called d times. We study its runtime overhead in Section 5.6.

Method 3: ObservedFisher ObservedFisher numerically computes H by relying on the information matrix equality [77].[3] According to the information matrix equality, the covariance matrix C of $q(\theta_n; x_i; y_i)$, for $i = 1; \cdots; n$, is asymptotically identical to J (i.e., as $n \to \infty$). In addition, given J, we can simply obtain H as $H = C + J_r$. Our empirical study in Section 5.6 shows that ObservedFisher is highly accurate for $n \ge 5K$.

Instead of computing the d-by-d matrix C directly, ObservedFisher takes a slightly different approach. That is, ObservedFisher computes *factors* U and Σ such that $C = U \Sigma^2 U^\top$, where U is a d-by-n matrix, and Σ is an n-by-n diagonal matrix. As described below, this factor-based approach is significantly more efficient when d is large, hence allowing BlinkML to scale up to high-dimensional data.

Specifically, let Q be the n-by-d matrix whose i-th row is $q(\theta_n; x_i; y_i)$. 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, the following relationship holds:

$$C = Q^\top Q = U \Sigma^2 U^\top \tag{6}$$

As stated above, ObservedFisher never computes C; it only stores U and Σ, which are used directly for obtaining samples from $\mathcal{N}(0; H^{-1} J H^{-1})$ (see Section 4.3). The cost of singular value decomposition is $O(\min(n^2 d; n d^2))$. When $d \ge n$, this time complexity becomes $O(n^2 d) = O(d)$ for a fixed sample size (n_0 = 10K by default). Moreover, ObservedFisher requires only a single call of the **`grads`** function. Section 5.6 empirically studies the relationship between n and ObservedFisher's runtime.

[3] ObservedFisher is also inspired by Hessian-free optimizations [69, 72].

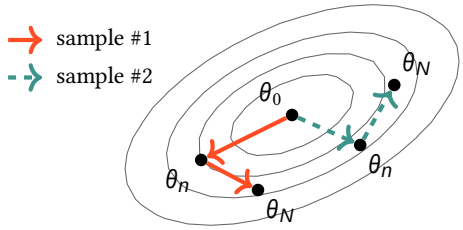


Figure 4: BlinkML repeats this parameter sampling process multiple times to estimate the accuracy of an approximate model m_n (with param θ_n) without having to train it.

4 SAMPLE SIZE ESTIMATOR

Sample Size Estimator estimates the minimum sample size n such that $E(m_n(x)\ ,\ m_N(x))$ is not larger than the requested error bound ε with probability at least $1-\delta$. In this process, Sample Size Estimator does not train any additional approximate models; it only relies on the initial model m_0 given to this component.

4.1 Quality Estimation sans Training

This section explains how Sample Size Estimator computes the probability of $E_x(m_n(x)\ ,\ m_N(x)) \le \varepsilon$ given the initial model m_0. Since both models—$m_n(x) = m(x;\hat{\theta}_n)$ and $m_N(x) = m(x;\hat{\theta}_N)$—are uncertain, Sample Size Estimator uses the joint probability distributions of $\hat{\theta}_n$ and $\hat{\theta}_N$ to compute the probability. To make it clear that both models involve uncertain parameters, we use the following notation:

$$v(m_n;m_N;\theta_n;\theta_N) = E_{x\sim\mathcal{D}}(1\,[m(x;\theta_n)\ ,\ m(x;\theta_N)])$$

The computed probability, i.e., $\Pr(v(m_n;m_N;\theta_n;\theta_N) \le \varepsilon)$, is then used in the following section for finding n that makes the probability at least $1-\delta$.

Like Model Accuracy Estimator, Sample Size Estimator computes the probability using the i.i.d. samples from the joint distribution $h(\theta_n;\theta_N)$ of $(\theta_n;\theta_N)$ as follows:

$$\Pr(v(m_n;m_N;\theta_n;\theta_N) \le \varepsilon) \tag{7}$$

$$= \int 1\,[v(m_n;m_N;\theta_n;\theta_N) \le \varepsilon]\ h(\theta_n;\theta_N)\,d\theta_n\,d\theta_N$$

$$\approx \frac{1}{k}\sum_{i=1}^{k} 1\ \ v(m_n;m_N;\theta_{n;i};\theta_{N;i}) \le \varepsilon \tag{8}$$

where the integration is over the domain of θ_n and the domain of θ_N, both of which are $\mathcal{R}^d$. To offset the approximation error in Equation (8), BlinkML makes conservative estimates using Lemma 2.

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

$$\Pr(\theta_n;\theta_N \mid \theta_0) = \Pr(\theta_N \mid \theta_n)\ \Pr(\theta_n \mid \theta_0)$$

where the conditional distributions, $\theta_N \mid \theta_n$ and $\theta_n \mid \theta_0$, are obtained using Corollary 1. That is, Model Accuracy Estimator uses the following two-stage sampling procedure. It first samples $\theta_{n;i}$ from $\mathcal{N}(\theta_0;\ \alpha_1 H^{-1}JH^{-1})$ where $\alpha_1 = (1/n_0 - 1/n)$; then, samples $\theta_{N;i}$ from $\mathcal{N}(\theta_{n;i};\ \alpha_2 H^{-1}JH^{-1})$ where $\alpha_2 = (1/n - 1/N)$. This process is repeated for every $i = 1;\cdots;k$ to obtain k pairs of $(\theta_{n;i};\theta_{N;i})$. Figure 4 depicts this process.

4.2 Sample Size Searching

To find the minimum n such that $\Pr(v(m_n;m_N;\theta_n;\theta_N) \le \varepsilon) \ge 1-\delta$, Sample Size Estimator uses binary search, exploiting that the probability tends to be an increasing function of n. We first provide an intuitive explanation; then, we present a formal argument.

Observe that $\Pr(v(m_n;m_N;\theta_n;\theta_N) \le \varepsilon)$ relies on the two model parameters θ_n and θ_N. If $\theta_n = \theta_N$, the probability 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 probability must increase toward 1 as $n \to N$. The following theorem formally shows that $\Pr(v(m_n;m_N;\theta_n;\theta_N) \le \varepsilon)$ is guaranteed to be an increasing function for a large class of cases (its proof is in Appendix B).

Theorem 2. *Let $h(\theta;\gamma C)$ be the probability density function of a normal distribution with mean θ_N and covariance matrix γC, 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 $v(m_n;m_N;\theta_n;\theta_N) \le \varepsilon$. Then, the following function $p_v(\gamma)$*

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

is a decreasing function of γ.

Since binary search is used, Sample Size Estimator needs to compute $\frac{1}{k}\sum_{i=1}^{k} 1\ \ v(m_n;m_N;\theta_{n;i};\theta_{N;i}) \le \varepsilon$ (in Equation (8)) for different values of n; in total, $O(\log_2(N-n_0))$ times. Thus, a fast mechanism for producing i.i.d. samples is desirable. The following section describes BlinkML's optimizations.

4.3 Optimizations for Fast Sampling

This section describes how to quickly generate i.i.d. samples from the normal distribution with covariance matrix $(1/n - 1/N)\,H^{-1}JH^{-1}$. 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.

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 $\hat{\theta}_n \sim \mathcal{N}(0;\ (1/n - 1/N)H^{-1}JH^{-1})$, and let $\hat{\theta}_0 \sim \mathcal{N}(0;\ H^{-1}JH^{-1})$. Then, there exists the following relationship:

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

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

Avoiding Direct Covariance Computation When $r(\theta) = \beta\theta$ in Equation (3) (i.e., no regularization or L2 regularization), Sample Size Estimator avoids the direct computations of $H^{-1}JH^{-1}$. Instead, it simply draws samples from the standard normal distribution and applies an appropriate linear transformation L to the sampled values (L is obtained shortly). This approach is used in conjunction with ObservedFisher, which is BlinkML's default strategy for computing its necessary statistics (Section 3.4).

Avoiding the direct computation of the covariances has two benefits. First, we can completely avoid the $\Omega(d^2)$ cost of computing/storing $H^{-1}JH^{-1}$. Second, sampling from the standard normal distribution is much faster because no dependencies need to be enforced among sampled values.

We use the following relationship:

$$z \sim \mathcal{N}(0; I) \quad \Rightarrow \quad Lz \sim \mathcal{N}(0; LL^\top):$$

That is, if there exists L such that $LL^\top = H^{-1}JH^{-1}$, we 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 (6) that $J = U\,\Sigma^2 U^\top$. Since $H = J + \beta$, $H = U\,(\Sigma^2 + \beta I)U^\top$. Thus,

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

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 already available as part of computing the necessary statistics in Section 3.4. Thus, computing L only involves a simple matrix multiplication.

5 EXPERIMENTS

Our experimental results show the following:

1. BlinkML reduces training time by 84.04%–99.84% (i.e., 6.26×–629×) when training 95% accurate models. and by 7.20%–96.47% (i.e., 1.07×–28.31× faster) when training 99% accurate models. (Section 5.2)
2. The actual accuracy of BlinkML's approximate models is, in most cases, even higher than the requested accuracy. (Section 5.3)
3. BlinkML's estimated minimum sample sizes are close to optimal. (Section 5.4)
4. BlinkML is highly effective and accurate even for high-dimensional data, and its runtime overhead is much smaller than the time needed for training a full model. (Section 5.5)
5. BlinkML's default statistics computation method, ObservedFisher, is both accurate and efficient. (Section 5.6)
6. BlinkML offers significant benefits in hyperparameter optimization compared to full model training. (Section 5.7)
7. BlinkML's sample size estimation is adaptive to the properties of models. (Section 5.8)

Table 2: Datasets used in our experiments

Dataset	# of Rows (N)	Dimension (d)	Size
Gas	4,178,504	57	1.9 GB
Power	2,075,259	114	1.8 GB
Criteo	45,840,616	998,922	2.86 GB
HIGGS	11,000,000	28	7.5 GB
MNIST	8,000,000	784	47.5 GB
Yelp	5,261,667	100,000	487 MB

5.1 Experiment Setup

Here, we present our computational environment as well as the different models and datasets used in our experiments.

Models We tested BlinkML with four different ML models:

1. **Linear Regression (Lin).** Lin is the standard linear regression model with L2 regularization coefficient β set as 0.001. Different values of β are tested in Section 5.8.
2. **Logistic Regression (LR).** LR is the standard logistic regression (binary) classifier with L2-regularization coefficient β set as 0.001.
3. **Max Entropy Classifier (ME).** ME is the standard max entropy (multiclass) classifier with L2-regularization coefficient β set as 0.001.
4. **PPCA.** PPCA is the standard probabilistic principal component analysis model [99], with the number of factors q set as 10.

BlinkML is configured to use the BFGS optimization algorithm for low-dimensional datasets ($d < 100$) and to use a memory-efficient alternative, called L-BFGS, for high-dimensional datasets ($d \geq 100$).

Datasets We used six real-world datasets. The key characteristics of these datasets are summarized in Table 2.

Linear Regression:

1. **Gas**: This dataset contains chemical sensor readings exposed to gas mixtures at varying concentration levels [44].

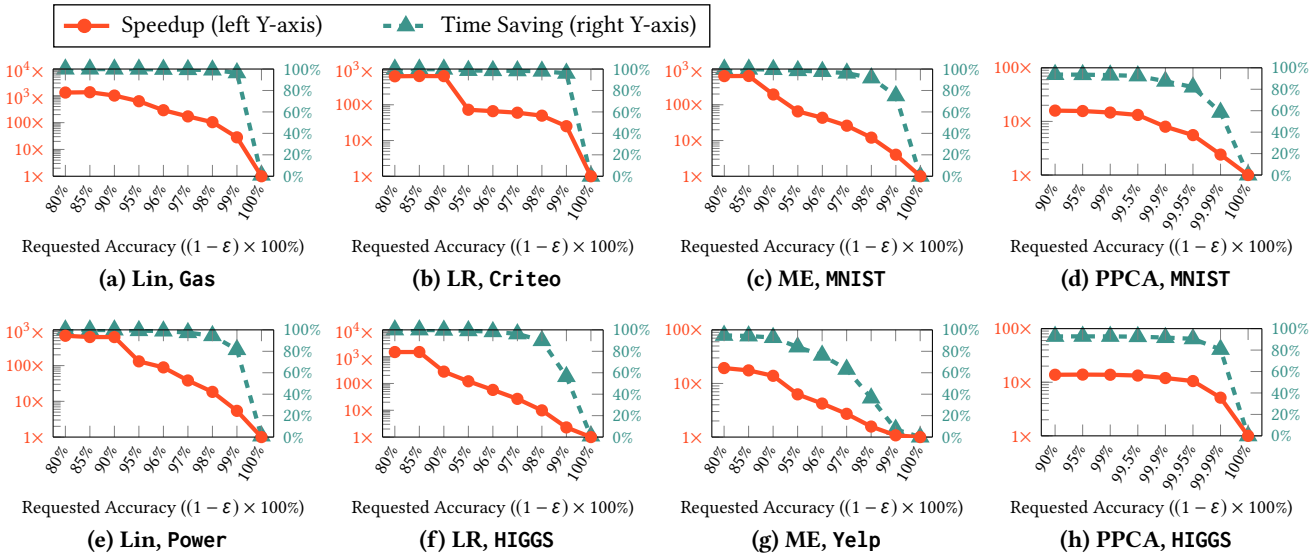


Figure 5: BlinkML's speedups compared to full model training (the accuracies of the approximate models are studied in Figure 6).

We use the sensor reading as a target variable and gas concentration levels as independent variables.

2. **Power**: This dataset contains power consumption measurements [7]. We use the household power consumption as a target variable and global power consumptions as independent variables.

Logistic Regression:

3. **Criteo**: This is a click-through rates dataset made publicly available by Criteo Labs [5]. The label of each example indicates if an ad was clicked or not.
4. **HIGGS**: This is a Higgs bosons simulation dataset [18]. Each example is a pair of physical properties of an environment and a binary indicator of Higgs bosons production.

Max Entropy Classifier:

5. **MNIST**: This is a hand-written digits image dataset (a.k.a. infinite MNIST [8]). Each example is a pair of an image (intensity value per pixel) and the actual digit in the image.
6. **Yelp**: This is a collection of publicly available Yelp reviews [13]. Each example is a pair of an English review and a rating (between 0 and 5).

Probabilistic Principal Component Analysis:

7. **MNIST**: We use the features of MNIST for PPCA.
8. **HIGGS**: We use the features of HIGGS for PPCA.

For each dataset, we used 80% for training and 20% for testing.

Environment All of our experiments were conducted on an EC2 cluster with one m5.4xlarge node as a master and five m5.2xlarge nodes as workers.[4] We used Python 3.6 shipped with Conda [3] and Apache Spark 2.2 shipped with Cloudera Manager 5.11.

5.2 Training Time Savings

This section measures the time savings by BlinkML, compared to training the full model. We used eight model and dataset combinations: (Lin, Gas), (Lin, Power), (LR, Criteo), (LR, HIGGS), (ME, MNIST), (ME, Yelp), (PPCA, MNIST), and (PPCA, HIGGS). For Lin, LR and ME, we varied the requested accuracy $(1-\varepsilon)\times 100\%$ from 80% to 99%. For PPCA, we varied the requested accuracy $((1-\varepsilon)\times 100\%)$ from 90% to 99.99%. We fixed δ at 0.05. For each case, we repeated BlinkML's training 20 times.

Figure 5 shows BlinkML's speedups and training time savings in comparison to full model training. The full model training times were 345 seconds for (Lin, Gas), 876 seconds for (Lin, Power), 5,727 seconds for (LR, Criteo), 530 seconds for (LR, HIGGS), 35,361 seconds for (ME, MNIST), 3,048 seconds for (ME, Yelp), 35 seconds for (PPCA, MNIST), and 2 seconds for (PPCA, HIGGS).

Two patterns were observed. First, as expected, 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 took longer than LR). This is because multi-class classification (by ME) involves more possible class labels; thus, even a small error in parameter values could lead to misclassification, so a larger sample was needed to sufficiently upper bound the chances

[4] m5.4xlarge instances had 16 CPU cores and 64 GB memory. m5.2xlarge instances had 8 CPU cores and 32 GB memory.

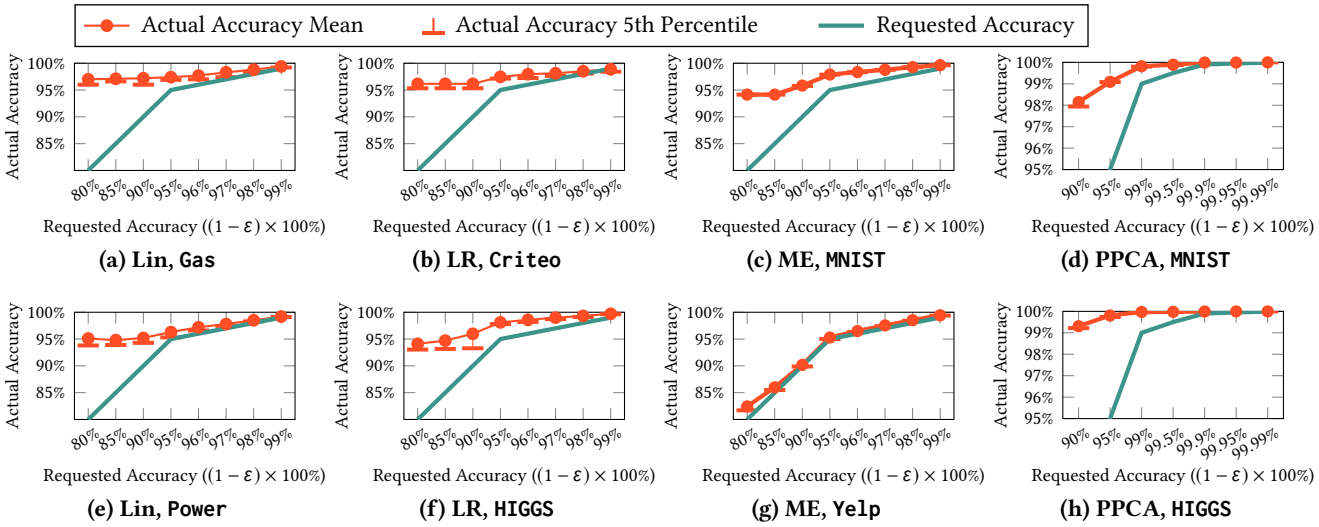


Figure 6: The correctness of BlinkML. The requested model accuracies were compared to the actual model accuracies. In most cases, 95% of the actual model accuracies were (or equivalently, the 5th percentile of the actual accuracies was) higher than the requested accuracies.

of a misclassification. Nevertheless, training 95% accurate ME models on MNIST still took only 1.53% of the time needed for training a full model. In other words, even in this case, we observed a 65.27× speedup, or equivalently, 98.47% savings in training time. 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.

5.3 Accuracy Guarantees

As stated in Equation (8) in Section 4, the actual accuracy of BlinkML's approximate model is guaranteed to never be less than the requested accuracy, with probability at least $1-\delta$. In this section, we also empirically validate BlinkML's accuracy guarantees. Specifically, we ran an experiment to compare the requested accuracies and the actual accuracies of the approximate models returned by BlinkML.

We varied the requested accuracy from 80% to 99% and requested a confidence level of 95%, i.e., $\delta = 0{:}05$. The results are shown in Figure 6 for the same combinations of models and datasets used in the previous section.

In each case, 5th percentile of the actual accuracies was higher than the requested accuracy. In other words, in 95% of the cases, the delivered accuracy was higher than the requested one, confirming that BlinkML's probabilistic accuracy guarantees were satisfied.

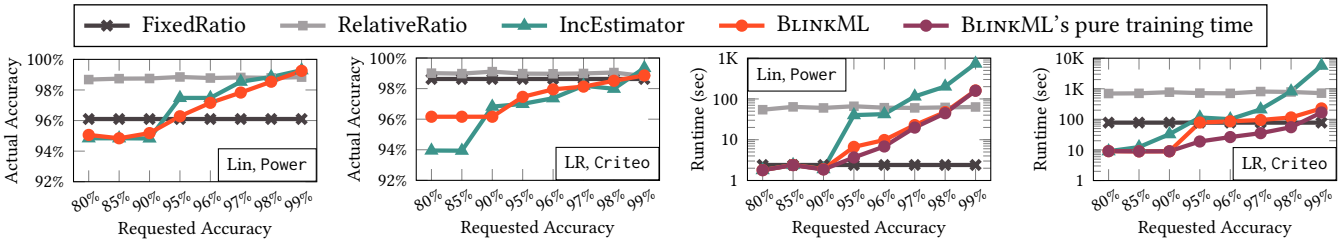
Notice that, in some cases, e.g., (LR, Criteo), (ME, MNIST), the actual accuracies remained identical even though the requested accuracies were different. This was due to BlinkML's design. Recall that BlinkML first trains an initial model m_0 and then trains a subsequent model only when the estimated model difference ε_0 of m_0 is higher than the requested error ε. In the aforementioned cases, the initial models were already accurate enough; therefore, no additional models needed to be trained. Consequently, the actual accuracies did not vary in those cases. In other words, the actual accuracies of those initial models were higher than the requested accuracies.

5.4 Sample Size Estimation

Sample Size Estimator (SSE) is responsible for estimating the minimum sample size, which is a crucial operation in BlinkML. Too large a sample eliminates the training time savings; likewise, too small a sample can violate the accuracy guarantees. In this section, we examine SSE's operations in BlinkML. We analyze both the accuracy and the efficiency of SSS.

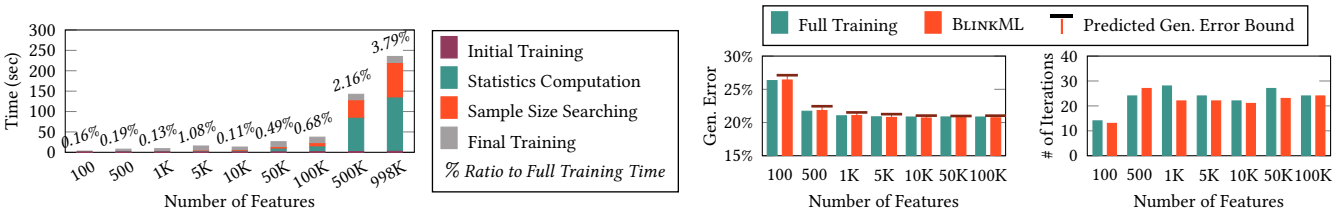
Accuracy To analyze the accuracy of SSS, we implemented three other baselines: FixedRatio, RelativeRatio, and IncEstimator. FixedRatio always used 1% samples for training approximate models. RelativeRatio used $(1-\varepsilon) * 10\%$ samples for training approximate models (e.g., 9.5% sample for 95% requested accuracy). IncEstimator gradually increased the sample size until the approximate model trained on that sample satisfied the requested accuracy; the sample size at k-th iteration was $1000 \cdot k^2$. We tested these three baselines and BlinkML on both (Lin, Power) and (LR, Criteo).

Figure 7a shows the results. Since FixedRatio and RelativeRatio set the sample sizes regardless of the model, they either failed to satisfy the requested accuracies or were overly costly. In contrast, IncEstimator and BlinkML adjusted their



(a) Effectiveness of Sample Size Estimator (b) Efficiency of Sample Size Estimator

Figure 7: The effectiveness and efficiency of BlinkML's sample size estimator. Two baselines (FixedRatio and RelativeRatio) either failed to satisfy the requested accuracies or were costly. IncEstimator met the requested accuracies, but was often quite slow.



(a) Runtime Overhead (b) Gen. Error (c) # of Iterations

Figure 8: (Left) BlinkML's runtime overhead, (Middle) generalization error (i.e., the errors on test sets), and (Right) the number of iterations taken by an optimization algorithm (L-BFGS) for different numbers of features. In (a), the ratio above each bar is BlinkML's entire training time compared to full training.

sample sizes according to the models and the requested accuracies; hence, they were able to satisfy the requested accuracies. However, IncEstimator was much more expensive than BlinkML, as described next.

Efficiency To measure the efficiency, we measured the training times of BlinkML and all three baselines. To show the overhead of SSS, we also measured BlinkML's training time *excluding* the time spent by SSS (referred to as "BlinkML's pure training time"). Figure 7b shows the results. In this figure, the runtimes of IncEstimator were significantly larger than those of BlinkML. For instance, IncEstimator took 5,704 seconds for a 99% accurate model of (LR, `Criteo`), while BlinkML took only 228 seconds (i.e., 25× faster than IncEstimator). Moreover, the runtime overhead of SSS was small enough to keep BlinkML's entire approximate training fast enough. We also study this overhead more systematically, as reported in the following section.

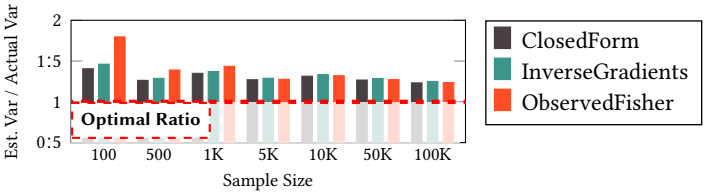
5.5 Impact of Data Dimension

In this section, we analyze the impact of data dimension (i.e., the number of features) on BlinkML's runtime overhead, generalization error, and number of optimization iterations.

Runtime Overhead To analyze BlinkML's runtime overhead, we separately measured (1) the time taken for training an initial model (using a sample of size n_0), (2) the time taken by ObservedFisher for computing statistics, (3) the time taken for sample size estimation, and (4) the time taken for training a final model. Figure 8a shows the results for (LR, `Criteo`). As expected, the times for statistics computations and sample size estimation increased as the number of features increased. However, the overall training time was still significantly smaller than training the full model (e.g., 0.8% for 100K features).

Generalization Error Figure 8b compares the generalization error (or equivalently, test error) of full models and BlinkML's approximate models. Since BlinkML probabilistically guarantees that its approximate models produce the same predictions as the full models (for the same number of features), the generalization errors of BlinkML's approximate models were highly similar to those of the full models. Moreover, the full model's generalization errors were within the predicted bounds for more than 95% of the test cases (as stated in Lemma 1).

Number of Iterations Lastly, to better understand the reason behind BlinkML's training time savings, we measured the number of iterations taken by an optimization algorithm (i.e., L-BFGS). Figure 8c shows the results. In general, the number of iterations for BlinkML were comparable to those for full model training. The number of iterations were similar between the full training and BlinkML's training, indicating that BlinkML's time savings are indeed due to faster gradient computations (because of sampling).



(a) Estimation Tightness vs. Sample Size

Model, Data	Metric	IG	OF
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

(b) InverseGradients (IG) vs. ObservedFisher (OF)

Figure 9: A study of statistics computation methods: (left) a comparison of estimated parameter variances to the actual parameter variances, and (right) a comparison of two statistics computation methods.

5.6 Statistics Computation

In this section, we compare ClosedForm, InverseGradients, and ObservedFisher in terms of both accuracy and efficiency. This analysis serves as an empirical justification for why ObservedFisher is BlinkML's default choice.

Accuracy To compare the accuracy, we first estimated the variances of the parameters using each of ClosedForm, InverseGradients, and ObservedFisher. Then, we computed the ratio between those estimated variances and the actual variances. Thus, a ratio close to 1 would indicate high accuracy. Figure 9 shows the results for (Lin, Power). In general, for small samples ($n \leq 1000$), ObservedFisher was relatively inaccurate in comparison to other methods. However, as the sample size increased, the accuracy of all methods improved (i.e., the ratio approached 1.0); further, the accuracy of ObservedFisher was comparable to other methods. Recall that ClosedForm is applicable to certain types of models (e.g., Lin and LR) while InverseGradients and ObservedFisher are applicable to all MLE-based models. Overall, the estimated variances of the parameters were larger than the actual variances, showing that BlinkML's probabilistic guarantees were met. Below, we compare the runtime overhead of InverseGradients and ObservedFisher.

Efficiency For an in-depth comparison of InverseGradients and ObservedFisher, we used two more combinations (LR, HIGGS) and (ME, MNIST). Recall from Table 2 that HIGGS is a low-dimensional dataset ($d = 28$) while MNIST is high-dimensional ($d = 784$). We used each method to compute the covariance matrix $H^{-1}JH^{-1}$, which determines the parameter distribution in Theorem 1 (Section 3). We measured the runtime of each method and calculated the accuracy of its estimated covariance matrix. To measure the accuracy, we calculated the average Frobenius norm, i.e., $(1/d^2)\,\|C_t - C_e\|_F$, where C_t was the true covariance matrix and C_e was the estimated covariance matrix.

Figure 9b summarizes the results of this experiment. For the low-dimensional data (HIGGS), the runtimes and accuracies of the two methods were comparable. While their accuracies remained comparable for the high-dimensional data (MNIST), their performance differed. Since InverseGradients had to invoke the grads function repeatedly (i.e., d times), its runtime increased drastically. In contrast, ObservedFisher had to call the grads function only once.

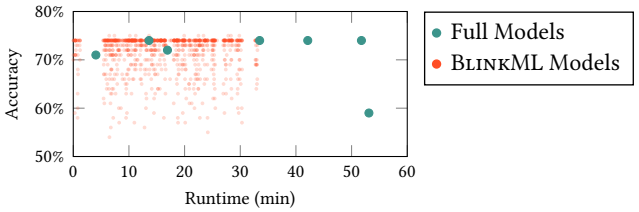
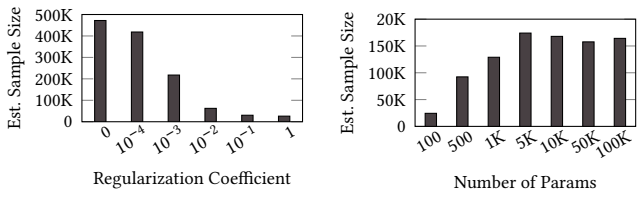


Figure 10: Hyperparameter optimization

5.7 Hyperparameter Optimization

This section studies BlinkML's overall benefit for performing hyperparameter optimization. Specifically, we compared BlinkML to a traditional approach (i.e., full model training) in searching for an optimal combination of a feature set and a hyperparameter. As performed by the Random Search hyperparameter optimization method [21], we first generated a sequence of (pairs of) a randomly chosen feature set and a regularization coefficient. Then, we let BlinkML train a series of 95% accurate models using the feature set and the regularization coefficient in the sequence. Similarly, we let the traditional approach train a series of exact models using the same combination of the feature set and the regularization coefficient.

Figure 10 shows the result, where each dot represents a model. Within half an hour, BlinkML trained 961 models while the traditional approach was able to train only 3. Both BlinkML and the traditional approach found the second-best model (with test accuracy 74%) at the second iteration (since they used the same sequence); however, BlinkML took only 1.03 seconds while the traditional approach took 817 seconds. After 387 seconds (about 6 mins), BlinkML found the best model (with test accuracy 75%; found at iteration #91), while the traditional approach could not find the model in an hour. The sizes of the samples used by BlinkML varied between 10,000 and 9,211,426, depending on the feature set



(a) Regularization **(b) Number of Params**

Figure 11: Model complexity vs. estimated sample sizes

and the regularization coefficient. Note that this is expected, since BlinkML automatically chooses the smallest sample size that is still large enough to train an accurate model. We study BlinkML's sample size estimation in more detail in the following section.

5.8 Impact of Model Complexity on Estimated Sample Sizes

We also study the impact of model complexity (i.e., regularization coefficients and the number of parameters) on BlinkML's estimated sample size. Intuitively, if a model is more complex (i.e., smaller regularization coefficients or more parameters), a larger sample would be needed to satisfy a user-requested error bound.

Figure 11 shows the results. The left subfigure (Figure 11a) shows that as the regularization coefficient increased, the estimated sample size decreased. The right subfigure (Figure 11b) shows that as the model had a larger number of parameters, the estimated sample size increased. Both observations are consistent with our intuition (Section 3).

6 RELATED WORK

In this section, we first discuss three types of work closely related to BlinkML: Sampling for ML, Hyperparameter Optimization, and Feature Selection. We summarize their key contributions in Table 3.

Next, we overview the approaches inspired by familiar database techniques (DBMS-Inspired Optimization) and the recent advances in statistical optimization methods (Faster Optimization Algorithms).

Sampling for ML Sampling for ML has been extensively studied in the literature. Typically, the sampling is optimized for a specific type of model, such as linear regression [17, 22, 30, 34, 36–38, 41, 47], logistic regression [62, 100], clustering [42, 56], Kernel matrices [49, 76], point processes [67], and Gaussian mixture model [70]. Woodruf [102] overviews sketching techniques for least squares and low rank approximation. Zombie [16] employs a clustering-based active learning technique for training approximate models, but does not offer any error guarantees. In contrast, BlinkML offers probabilistic error guarantees for any MLE-based model. Cohen [30] studies uniform sampling for ML, but for only linear regression. Bottou [25] and Shalev-Shwartz [91] study the optimization errors.

Hyperparameter Optimization Automatic hyperparameter optimization (i.e., AutoML) has been pursued by AutoWeka [97], auto-sklearn [43], and Google's Cloud AutoML [2]. These techniques have typically taken a Bayesian approach to hyperparameter optimization [90]. The ML community has also studied how to optimize deep neural networks [20, 21, 71]. What BlinkML offers is orthogonal to these hyperparameter optimization methods, since BlinkML speeds up the individual model training via approximation, while hyperparameter optimization methods focus on finding an optimal sequence of hyperparameters to test.

Feature Selection Choosing a relevant set of features is an important task in ML [26, 51]. Feature selection can either be performed before training a final model [57] or during model training relying on sparsity-inducing models [78, 104]. One approach to accelerate feature selection is preprocessing the data in advance [106]. BlinkML can be used in conjunction with these methods.

DBMS-Inspired Optimization A salient feature of database systems is their declarative interface (SQL) and the resulting optimization opportunities. These ideas have been applied to speed up ML workloads. SystemML [24, 40, 48] ScalOps [101], Pig latin [80], and KeystoneML [95] propose high-level ML languages for automatic parallelization and materialization, as well as easier programming. Hamlet [65] and others [64, 89] avoid expensive denormalizations. Hemingway [81], MLBase [61], and TuPAQ [94] automatically choose an optimal plan for a given ML workload. SciDB [59, 96], MADLib [29, 55], and RIOT [109] exploit in-database computing. Kumar et al. [63] uses a model selection management system to unify feature engineering [15], algorithm selection, and parameter tuning. NoScope [58], tKDC [46], and NSH [83] speed up the prediction at the cost of more preprocessing.

Faster Optimization Algorithms Advances in optimization algorithms are largely orthogonal to ML systems; however, understanding their benefits is necessary for developing faster ML systems. Recent advances can be categorized into software- and hardware-based approaches.

Software-based methods are mostly focused on improving gradient descent variants (e.g., SGD), where a key question is how to adjust the step size in order to accelerate the convergence rate towards the optimal solution [92]. Recent advances include adaptive rules for accelerating this rate, e.g., Adagrad [39], Adadelta [105], RMSprop [98], and Adam

Table 3: Previous work on Sampling for ML / Hyperparameter Optimization / Feature Selection

	Approach	Key Contributions
Sampling for ML	Coreset (LR) [17, 41, 47]	Proposes a coreset (non-uniform sample) for linear regression
	Cohen [30]	Approximates linear regression with uniform random sampling
	Derezinski [34]	Proposes *volume sampling* for linear regression
	Zombie [16]	Applies active learning to sampling
	Drineas [36–38], Bhojanapalli [22]	Develops non-uniform sampling (based on leverage scores) for linear regression
	Wang [100], Krishnapuram [62]	Develops non-uniform sampling for logistic regression
	Coreset (Clustering) [42, 56]	Proposes a coreset (non-uniform sample) for clustering
	Gittens [49], Musco [76]	Approximates kernel matrices (i.e., matrices containing inner products)
	Coreset (GMM) [70]	Proposes a coreset (non-uniform sample) for Gaussian mixture models
	Li [67]	Proposes sampling for point processes
	SafeScreening [79]	Removes non-support vectors prior to training a SVM model
	BlinkML (ours)	Develops efficient algorithms for training MLE models with error guarantees
Hyperparameter Optimization	AutoWeka [97], auto-sklearn [43], Cloud AutoML [2]	Applies a wide range of feature selection & hyperparameter optimization techniques to existing ML frameworks/libraries
	Bergstra [21], Bergstra [20]	Optimizes hyperparameter optimization for deep neural network
	Maclaurin [71]	Proposes gradient-based hypeparameter optimization
	Shahriari [90]	Reviews Bayesian approach to hyperparameter optimization
	Bardenet [19]	Proposes a colloborative hyperparameter tuning among similar tasks
	TuPAQ [94]	Provides a systematic support for hyperparameter optimization
Feature Selection	Guyon [51], Boyce [26], John [57]	Introduces concepts related to feature selection
	Ng [78], Yang [104]	Performs feature selection via sparse models
	AutoWeka [97]	See the description above
	Columnbus [106]	Optimizes feature selection by materialization

[60]. Hogwild! [87] and related techniques [50, 52, 108] disable locks to speed up SGD via asynchronous updates. There is also recent work on rediscovering the benefits of quasi-Newton optimization methods, e.g., showing that minibatch variants of quasi-Newton methods (such as L-BFGS or CG) can be superior to SGD due to their higher parallelism [66].

Hardware-based techniques speed up training by relaxing strict precision requirements, e.g., DimmWitted [107] and BuckWild! [33].

7 CONCLUSION

In this work, we have developed BlinkML, an approximate machine learning system with probabilistic guarantees for MLE-based models. BlinkML uses sampling to dramatically reduce time and computational costs, which is particularly beneficial in the early stages of model tuning. Through an extensive set of experiments on several large-scale, real-world datasets, we showed that BlinkML produced 95% accurate models of linear regression, logistic regression, max entropy classifier, and Probabilistic Principal Component Analysis, while using only 0.16%–15.96% of the time needed for training the full model. Our future plan is to extend beyond the maximum likelihood estimation models, such as decision trees, Gaussian Process regression, Naïve Bayes classifiers, and Deep Boltzmann Machines [88]. We also plan to open-source BlinkML, with wrappers for various popular ML libraries, including scikit-learn (Python), glm (R), and MLlib. Finally, given that BlinkML's underlying techniques are extensible to non-uniform sampling, we plan to further explore task- and model-specific non-uniform sampling strategies.

8 ACKNOWLEDGEMENT

This research is in part supported by National Science Foundation through grants 1553169 and 1629397.

REFERENCES

[1] Apache spark's scalable machine learning library. https://spark.apache.org/mllib/. Retrieved: July 18, 2018.
[2] Cloud automl. https://cloud.google.com/automl/. Retrieved: Oct 30, 2018.
[3] Conda. https://conda.io/docs/index.html. Retrieved: July 18, 2018.
[4] cx_oracle version 6.2. https://oracle.github.io/python-cx_Oracle/. Retrieved: July 18, 2018.
[5] Download kaggle display advertising challenge dataset. http://labs.criteo.com/2014/02/download-kaggle-display-advertising-challenge-dataset/. Retrieved: Oct 30, 2018.
[6] ibmdbr: Ibm in-database analytics for r. https://cran.r-project.org/web/packages/ibmdbR/index.html. Retrieved: July 18, 2018.
[7] Individual household electric power consumption data set. https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption. Retrieved: Oct 30, 2018.
[8] The infinite mnist dataset. http://leon.bottou.org/projects/infimnist. Retrieved: July 18, 2018.
[9] Python sql driver - pymssql. https://docs.microsoft.com/en-us/sql/connect/python/pymssql/python-sql-driver-pymssql. Retrieved: July 18, 2018.
[10] Python support for ibm db2 and ibm informix. https://github.com/ibmdb/python-ibmdb. Retrieved: July 18, 2018.
[11] R interface to oracle data mining. http://www.oracle.com/technetwork/database/options/odm/odm-r-integration-089013.html. Retrieved: July 18, 2018.
[12] Revoscaler. https://docs.microsoft.com/en-us/sql/advanced-analytics/r/revoscaler-overview. Retrieved: July 18, 2018.
[13] Yelp dataset. https://www.kaggle.com/yelp-dataset/yelp-dataset. Retrieved: July 18, 2018.
[14] 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.
[15] 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.
[16] M. R. Anderson and M. Cafarella. Input selection for fast feature engineering. In *ICDE*, pages 577–588, 2016.
[17] O. Bachem, M. Lucic, and A. Krause. Practical coreset constructions for machine learning. *arXiv preprint arXiv:1703.06476*, 2017.
[18] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 2014.
[19] R. Bardenet, M. Brendel, B. Kégl, and M. Sebag. Collaborative hyperparameter tuning. In *ICML*, 2013.
[20] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.
[21] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *NIPS*, 2011.
[22] S. Bhojanapalli, P. Jain, and S. Sanghavi. Tighter low-rank approximation via sampling the leveraged element. In *SODA*, 2015.
[23] C. M. Bishop. *Pattern recognition and machine learning*. 2006.
[24] 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.
[25] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *NIPS*, 2008.
[26] D. Boyce, A. Farhi, and R. Weischedel. *Optimal subset selection: Multiple regression, interdependence and optimal network algorithms*. 2013.
[27] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *VLDB*, September 2000.
[28] 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.
[29] 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.
[30] M. B. Cohen, Y. T. Lee, C. Musco, C. Musco, R. Peng, and A. Sidford. Uniform sampling for matrix approximation. In *ITCS*, 2015.
[31] 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.
[32] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: Interactive analytics through pen and touch. *PVLDB*, pages 2024–2027, 2015.
[33] 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.
[34] M. Derezinski and M. K. Warmuth. Unbiased estimates for linear regression via volume sampling. In *NIPS*, 2017.
[35] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo-charging estimate convergence in dbo. *PVLDB*, 2009.
[36] P. Drineas, M. Magdon-Ismail, M. W. Mahoney, and D. P. Woodruff. Fast approximation of matrix coherence and statistical leverage. *JMLR*, 2012.
[37] P. Drineas, M. W. Mahoney, and S. Muthukrishnan. Sampling algorithms for l 2 regression and applications. In *SODA*, 2006.
[38] P. Drineas, M. W. Mahoney, S. Muthukrishnan, and T. Sarlós. Faster least squares approximation. *Numerische mathematik*, 2011.
[39] 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.
[40] 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.
[41] D. Feldman and M. Langberg. A unified framework for approximating and clustering data. In *STOC*, 2011.
[42] D. Feldman, M. Schmidt, and C. Sohler. Turning big data into tiny data: Constant-size coresets for k-means, pca and projective clustering. In *SODA*, 2013.
[43] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *NIPS*, 2015.
[44] J. Fonollosa, S. Sheik, R. Huerta, and S. Marco. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical*, 2015.
[45] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting reuse for approximate query processing. *PVLDB*, pages 1142–1153, 2017.
[46] E. Gan and P. Bailis. Scalable kernel density classification via threshold-based pruning. In *SIGMOD*, pages 945–959, 2017.
[47] M. Ghashami and J. M. Phillips. Relative errors for deterministic low-rank matrix approximations. In *SODA*, 2014.
[48] 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.
[49] A. Gittens and M. W. Mahoney. Revisiting the nyström method for improved large-scale machine learning. *JMLR*, 2016.
[50] 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.
[51] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.
[52] 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.
[53] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 2009.
[54] W. He, Y. Park, I. Hanafi, J. Yatvitskiy, and B. Mozafari. Demonstration of verdictdb, the platform-independent aqp system. In *SIGMOD*, 2018.
[55] 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.
[56] R. Jaiswal, A. Kumar, and S. Sen. A simple d 2-sampling based ptas for k-means and other clustering problems. 2014.
[57] G. H. John, R. Kohavi, and K. Pfleger. Irrelevant features and the subset selection problem. In *Machine Learning Proceedings*. 1994.
[58] 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.
[59] 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.
[60] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[61] 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.
[62] B. Krishnapuram, L. Carin, M. A. Figueiredo, and A. J. Hartemink. Sparse multinomial logistic regression: Fast algorithms and generalization bounds. *TPAMI*, 2005.
[63] 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.
[64] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984, 2015.
[65] 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.
[66] 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.
[67] C. Li, S. Jegelka, and S. Sra. Efficient sampling for k-determinantal point processes. *arXiv preprint arXiv:1509.01618*, 2015.
[68] 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.
[69] C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust region newton method for logistic regression. *JMLR*, 2008.
[70] M. Lucic, M. Faulkner, A. Krause, and D. Feldman. Training gaussian mixture models at scale via coresets. *JMLR*, 2017.
[71] D. Maclaurin, D. Duvenaud, and R. Adams. Gradient-based hyperparameter optimization through reversible learning. In *ICML*, 2015.
[72] J. Martens. Deep learning via hessian-free optimization. In *ICML*, 2010.
[73] 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.
[74] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT press, 2012.
[75] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Eng. Bull.*, 2015.
[76] C. Musco and C. Musco. Recursive sampling for the nystrom method. In *NIPS*, 2017.
[77] W. K. Newey and D. McFadden. Large sample estimation and hypothesis testing. *Handbook of econometrics*, pages 2111–2245, 1994.
[78] A. Y. Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *ICML*, 2004.
[79] K. Ogawa, Y. Suzuki, and I. Takeuchi. Safe screening of non-support vectors in pathwise svm computation. In *ICML*, 2013.
[80] 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.
[81] X. Pan, S. Venkataraman, Z. Tai, and J. Gonzalez. Hemingway: modeling distributed optimization algorithms. *arXiv*, 2017.
[82] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 4, 2011.
[83] Y. Park, M. Cafarella, and B. Mozafari. Neighbor-sensitive hashing. *PVLDB*, 2015.
[84] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: universalizing approximate query processing. In *SIGMOD*, 2018.
[85] Y. Park, A. S. Tajik, M. Cafarella, and B. Mozafari. Database Learning: Towards a database that becomes smarter every time. In *SIGMOD*, 2017.
[86] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *JMLR*, 2011.
[87] 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.
[88] R. Salakhutdinov and H. Larochelle. Efficient learning of deep boltzmann machines. In *AISTATS*, 2010.
[89] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, pages 3–18, 2016.
[90] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 2016.
[91] S. Shalev-Shwartz and N. Srebro. Svm optimization: inverse dependence on training set size. In *ICML*, 2008.
[92] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl. Measuring the effects of data parallelism on neural network training. 2018.
[93] 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.
[94] 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.
[95] 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.
[96] 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.
[97] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *KDD*, 2013.
[98] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. *University of Toronto, Tech. Rep*, 2012.
[99] M. E. Tipping and C. M. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 1999.
[100] H. Wang, R. Zhu, and P. Ma. Optimal subsampling for large sample logistic regression. *JASA*, 2018.
[101] M. Weimer, T. Condie, R. Ramakrishnan, et al. Machine learning in scalops, a higher order cloud computing language. In *BigLearn*, pages 389–396, 2011.
[102] D. P. Woodruff et al. Sketching as a tool for numerical linear algebra. *FnT-TCS*, 2014.
[103] E. Xing. Vc dimension and model complexity. https://www.cs.cmu.edu/~epxing/Class/10701/slides/lecture16-VC.pdf. Retrieved: July 18, 2018.
[104] A. Y. Yang, J. Wright, Y. Ma, and S. S. Sastry. Feature selection in face recognition: A sparse representation perspective. *TPAML*, 2007.
[105] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.
[106] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD*, pages 265–276, 2014.
[107] C. Zhang and C. Ré. Dimmwitted: A study of main-memory statistical analytics. *Proceedings of the VLDB Endowment*, 7(12):1283–1294, 2014.
[108] 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.
[109] Y. Zhang, W. Zhang, and J. Yang. I/o-efficient statistical computing with riot. In *ICDE*, pages 1157–1160, 2010.

A MODEL ABSTRACTION EXAMPLES

This section shows that BlinkML-supported ML models can be cast into the abstract form in Equation (2). 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 predicted class labels. An optional regularization term may be placed to prevent a model from being overfitted to a training set. For instance, the objective function of L2-regularized logistic regression is expressed as follows:

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

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 $i = 1, \cdots, n$, 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,

$$\nabla f_n(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\sigma(\theta^\top x_i) - t_i\right) x_i + \beta\theta = 0$$

It is straightforward to cast the above expression into Equation (3). That is, $q(\theta; x_i; t_i) = (\sigma(\theta^\top x_i) - t_i)\, x_i$ 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 a 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,

$$\nabla_\Theta f(\Theta) = C^{-1}(\Theta - SC^{-1}\Theta) = 0$$

The above expression can be cast into the form in Equation (3) by observing $S = (1/n)\sum_{i=1}^{n} x_i x_i^\top$.[5] That is, $q(\Theta; x_i) = C^{-1}\Theta - C^{-1}x_i x_i^\top C^{-1}\Theta$ 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.

B DEFERRED PROOFS

B.1 Generalization Error

Proof of Lemma 1. BlinkML's error guarantee is probabilistic because of the random sampling process for obtaining D_n. This proof exploits the fact that this random sampling process is independent of the distribution of $(x; y) \sim \mathcal{D}$.

The generalization error of the full model can be bounded as follows:

$$\begin{aligned}
&E_{(x;y)\sim\mathcal{D}}(1[m_N(x) \neq y]) = \int 1[m_N(x) \neq y]\, p(x)dx \\
&= \int_{(m_n(x) = y \wedge m_n(x) = m_N(x))} \\
&\quad \vee (m_n(x) = y \wedge m_n(x) \neq m_N(x))p(x)dx \\
&\leq \int_{(m_n(x) \neq y) \vee (m_n(x) = y \wedge m_n(x) \neq m_N(x))} p(x)dx \\
&\leq \int_{(m_n(x) \neq y)} p(x)dx \\
&\quad + \int_{(m_n(x) = y \wedge m_n(x) \neq m_N(x))} p(x)dx
\end{aligned}$$

By definition, $\int_{(m_n(x) \neq y)} p(x)dx = \varepsilon_\partial$. Based on the independence we stated above and $\Pr(E_x(m_n(x) \neq m_N(x)) \leq \varepsilon) \geq 1 - \delta$,

$$\int_{(m_n(x) = y \wedge m_n(x) \neq m_N(x))} p(x)dx \leq (1 - \varepsilon_\partial) \cdot \varepsilon$$

with probability at least $1 - \delta$. Thus,

$$E_{(x;y)\sim\mathcal{D}}(1[m_N(x) \neq y]) \leq \varepsilon_\partial + \varepsilon - \varepsilon_\partial \cdot \varepsilon$$

with probability at least $1 - \delta$.

B.2 Model Parameter Distribution

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 [77]. The generalization is required since the original result does not include $r(\theta)$.

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

Let θ_∞ be the parameter values at which $\partial_\infty(\theta)$ 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 $\partial_n(\theta_n) = 0$. According to the mean-value theorem, there exists $\bar{\theta}$ between θ_n and θ_∞ that satisfies:

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

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

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

$$\begin{aligned}
\sqrt{n}\,(\hat{\theta}_n - \theta_\infty) &= -H(\bar{\theta})^{-1}\sqrt{n}\,\partial_n(\theta_\infty) \\
&= -H(\bar{\theta})^{-1}\frac{1}{\sqrt{n}}\sum_{i=1}^{n}(q(\theta_\infty; x_i; y_i) + r(\theta_\infty)) \qquad (9) \\
&\xrightarrow{n\to\infty} \mathcal{N}(0; H^{-1}JH^{-1}) \qquad (10)
\end{aligned}$$

where H is a shorthand notation of $H(\bar{\theta})$. To make a transition from Equation (9) to Equation (10), an important relationship called the *information matrix equality* is used. According to the information matrix equality, the covariance of $q(\theta; x_i; y_i)$ 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 $\mathcal{N}(0; H^{-1}JH^{-1})$. From Equation (10), $\hat{\theta}_n - \theta_\infty \to (1/\sqrt{n})X_1$. Also, let $q_i = q(\theta_\infty; x_i; y_i)$ for simplicity; then,

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

Since $\hat{\theta}_n - \hat{\theta}_N = (\hat{\theta}_n - \theta_\infty) - (\hat{\theta}_N - \theta_\infty)$, and the limit of each of $(\hat{\theta}_n - \theta_\infty)$ and $(\hat{\theta}_N - \theta_\infty)$ exists,

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

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

that independently follow normal distributions. Thus,

$$\hat{\theta}_n - \hat{\theta}_N \xrightarrow{n\to\infty \text{ and } N\to\infty}$$
$$\mathcal{N}\left(0;\ \left(\frac{1}{\sqrt{n}} - \frac{\sqrt{n}}{N}\right)^2 H^{-1}JH^{-1} + \left(\frac{\sqrt{N-n}}{N}\right)^2 H^{-1}JH^{-1}\right)$$
$$= \mathcal{N}\left(0;\ \frac{1}{n} - \frac{1}{N}\ H^{-1}JH^{-1}\right)$$

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 is zero, as shown below:

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

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

$$\hat{\theta}_n \sim \theta_N;\ \alpha H^{-1}JH^{-1} \tag{11}$$

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 (11), $\hat{\theta}_N \mid \theta_n \sim \mathcal{N}(\theta_n; \alpha H^{-1}JH^{-1})$.

Proof of Lemma 2. We first define three events A, B, and C as follows:

$$A: \quad \int_1 v(m_n) \le \varepsilon$$
$$B: \quad \int 1\left[v(m_n) \le \varepsilon_1\right] h(\theta_N)\, d\theta_N \ge \frac{1-\delta}{0.95}$$
$$C: \quad \frac{1}{k}\sum_{i=1}^{k} 1\ v(m_n;\theta_{N;i}) \le \varepsilon_1 \ = \frac{1-\delta}{0.95} + \sqrt{\frac{\log 0.95}{-2k}}$$

It suffices to show that $P(A \mid C) \ge 1-\delta$, for which we use the relationship $P(A \mid C) = P(A \mid B)\,P(B \mid C)$.

First, by the basic relationship between the probability and the indicator function, $P(A \mid B) = (1-\delta)/0.95$.

Second, by the Hoeffding's inequality,

$$\Pr\left(\int_1 a \ge b - \sqrt{\frac{\log 0.95}{-2k}}\right) \ge 0.95$$

$$\text{where} \quad a = \int 1\left[v(m_n;\theta_N) \le \varepsilon\right] h(\theta_N)\, d\theta_N$$

$$b = \frac{1}{k}\sum_{i=1}^{k} 1\ v(m_n;\theta_{N;i}) \le \varepsilon_1$$

since b is the degree-k U-statistics of a. Thus, $P(B \mid C) \ge 0.95$.

Therefore, $P(A \mid C) = P(A \mid B)\,P(B \mid C) \ge (1-\delta)/0.95 \times 0.95 = 1-\delta$.

B.3 Sample Size Estimation

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 for 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.

C MODEL SIMILARITIES

The model difference $v(m_n)$ defined for classification in Section 2.1 can be extended to regression and unsupervised learning in a straightforward way, as follows. The experiment results in Section 5 used these definitions for corresponding models.

Regression For regression, $v(m_n)$ captures the expected prediction difference between m_n and m_N.

$$v(m_n) = \left(\mathrm{E}_{x\sim\mathcal{D}}\left[(m_n(x) - m_N(x))^2\right]\right)^{1/2}$$

Unsupervised Learning For unsupervised learning, the model difference captures the difference between the model parameters. For instance, BlinkML uses the following expression for PPCA:

$$v(m_n) = 1 - \mathrm{cosine}(\theta_n;\ \theta_N)$$

where θ_n and θ_N are the parameters of m_n and m_N, respectively, and $\mathrm{cosine}(\cdot;\cdot)$ indicates the cosine similarity.