BlinkML: Approximate Machine Learning with Probabilistic Guarantees

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

ABSTRACT

With the increase of dataset volumes, training machine learning (ML) models has become a major computational cost in most organizations. 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 on which direction to take (e.g., whether and what types of features to add, what type of model to pursue), and use the entire dataset only in later stages (i.e., when they have converged to specific model). This sampling, however, is performed in an ad-hoc fashion. Most practitioners are not able to 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 training of ML models with quality guarantees. BlinkML allows users to make error-computation tradeoffs: instead of training a model on their full data (i.e., *full model*), they can train on a smaller sample and thus obtain an *approximate model*. With BlinkML, however, they can explicitly control the quality of their approximate model by either (i) specifying an upper bound on how much the approximate model is allowed to deviate from the full model, or (ii) inquiring about the *minimum sample size* using which the trained model would meet a given accuracy requirement. Moreover, the error guarantees can be expressed in terms of the model's parameters or its final (classification or regression) predictions. BlinkML currently supports any ML model that relies on *maximum likelihood estimation*—an important class of ML algorithms, 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 1.54×–322× while incurring only 1% accuracy loss compared to the full model trained on the entire data.

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

Proceedings of the VLDB Endowment, Vol. 12, No. xxx

DOI: https://doi.org/TBD

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 [3], SQL Server's pymssql [6], and DB2's ibm_db [7]) or R (e.g., Oracle's RODM [8], SQL Server's RevoScaleR [9], and DB2's ibmdbR [4]). 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 [13, 14, 21, 57, 67], cost-based optimization of linear algebraic operators [17,20,30], array-based representations [37, 60], avoiding denormalization [41, 42, 56], lazy evaluation [70], declarative interfaces [49,59,63], and query planning [40,51,58] 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) [18,22,24,28,44,45,52,53]. However, applying the lessons learned in the data management community regarding AQP, we could use a similar sampling abstraction to also speed up an important class of ML workloads.

In particular, ML is often a *human-in-the-loop* and *iterative* process. The analysts perform some initial data cleaning, feature engineering/selection, hyper-parameter tuning, and model selection. They, then, inspect the results and may repeat this process, investing more effort in some of these steps, until they are satisfied with the model quality in terms of explanatory or predictive power. The entire process is therefore slow and computationally expensive. Many of the computational resources spent on early iterations are ultimately wasted, as the eventual model can differ quite a bit from the initial ones (in terms of features, parameters, or even model class). In fact, this is the reason why many practitioners use a small sample of their entire data during the initial steps of their analysis in order to reduce the computational burden and speed up the entire process. For instance, they may first train a model on a small sample (i.e., *approximate model*) to quickly test a new hypothesis, determine if the newly added feature improves accuracy, or tune a hyper-parameter. Only when the initial results are promising do they invest in training a *full model*,[1] i.e., the model trained on the full dataset (which can take significantly longer). The problem, however, is that this sampling process is ad-hoc and comes with no guarantees regarding the extent of error induced by sampling. The analysts do not know how much

[1]We do not call this model an exact model since all ML models are inherently approximate. However, the goal of BlinkML is to quantify the amount of error induced by sampling, i.e., how much could the accuracy improve had we used the entire data.

their sampling ratio or strategy affects the validity of their model tuning decisions. For example, had they trained a model with this new feature on the entire dataset rather than a small sample, they might have witnessed a much higher accuracy, which would have led them to include that feature in their final model.

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 both (i) bounding the deviation of the approximate model from the full model given a sample size, and (ii) predicting the *minimum sample size* with which the trained model would meet a given error tolerance. The error guarantees can be placed on both the model's parameters and its final predictions. Our abstraction is versatile and provides strong, PAC-style guarantees for different scenarios. For example, using an error tolerance ε and a confidence level δ ($0 \leq \delta \leq 1$), the analysts can request a model whose parameters' deviation from those of the full model is no more than ε, with probability at least $1 - \delta$. Likewise, the users can demand a model whose classification (or regression) error is within ε of the full model's error, with probability at least $1 - \delta$. Conversely, the analysts may train an approximate model, and then inquire about the probability of its predictions being more than ε different from the full model.

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

Challenges While estimating sampling error is a well-studied problem for SQL queries [11,47], it is much 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., *evaluative*). A well-known predictive technique is the so-called VC-dimension [46], 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 [64]. Overestimated error bounds would lead the analysts to use the entire dataset, even if similar results could be obtained from a much smaller sample. [2]

Common techniques for the evaluative approach include cross-validation [16] and Radamacher complexity [46]. Since these techniques are data-dependent, they provide tighter error estimates. However, they only bound the generalization error. While useful for evaluating the model's quality on future (i.e., unseen) data, the generalization error provides little help in predicting how much the model quality would differ if the entire dataset were used instead of the current sample. Furthermore, when choosing the minimum sample size, the evaluative approaches can be quite expensive: one would need to train multiple models each on a different sample size until a desirable error tolerance is met. Given that most ML models do not have an incremental training procedure (besides a warm start [19]), training multiple models to find an appropriate sample size might take longer overall than simply training a model on the full dataset (see Section 6.4).

Our Approach Given a test example $\boldsymbol{x}$, the model's prediction is simply a function $m(\boldsymbol{x}; \theta)$, where θ is the model parameter learned during the training phase. Let θ_N be the model parameter obtained if one trains on the entire dataset (say, of size N), and $\hat{\theta}_n$ be the model parameter obtained if one trains on a sample of size n.[3] Obtaining θ_n is fast when $n \ll N$; however, θ_N is unknown unless we use the entire dataset. Our key idea is to exploit the asymptotic distribution of $\theta_N - \hat{\theta}_n$ to analytically (thus, efficiently) derive the conditional distribution of $\hat{\theta}_N \mid \theta_n$, where $\hat{\theta}_N$ represents our (limited) probabilistic knowledge of θ_N (Theorem 1 and Corollary 1). The asymptotic distribution of $\theta_N - \hat{\theta}_n$ is available for the ML methods relying on maximum likelihood estimation for their parameter optimizations, which includes Generalized Linear Models [16] and Probabilistic PCA [62]. This indicates that, while we cannot determine the exact value of θ_N without training the full model, we can use the conditional distribution of $\hat{\theta}_N \mid \theta_n$ to probabilistically bound the deviation of θ_N from θ_n, and consequently, the deviation of $m(x; \theta_N)$ from $m(x; \theta_n)$ (Section 4.2). This also means we can 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 original sample of size n (Section 5.2). In other words, without having to perform additional training, we can efficiently search for the minimum sample size n', with which the approximate model, $m(x; \theta_{n'})$ would be guaranteed, with probability $1 - \delta$, to not deviate from $m(x; \theta_N)$ by more than ε.

Contributions We make the following contributions:

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

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

[2] This is why VC-dimensions are sometimes used indirectly, as a comparative measure of quality [42].

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

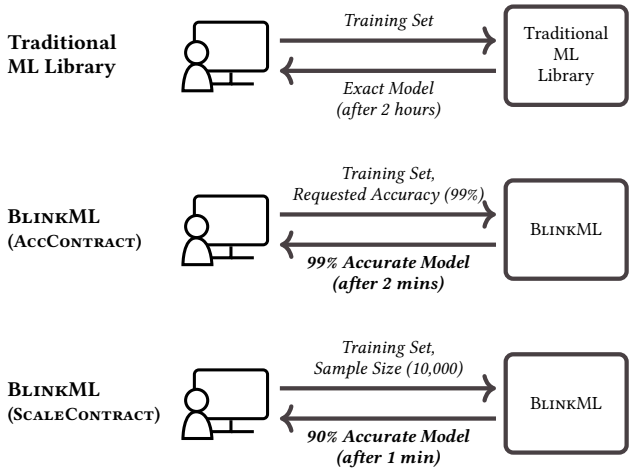


Figure 1: Interaction difference between traditional ML libraries and BlinkML. BlinkML can quickly train an approximate ML model in accordance to a user-specified *approximation contract*: AccContract or ScaleContract. Latency and accuracy examples are based on our experiment results (Section 6).

```
1  from sklearn import linear_model as sml
2  lr = sml.LogisticRegression()
3  model = lr.fit(features, labels)
```

(a) scikit-learn

```
1  import blinkml as bml
2  lr = bml.LogisticRegression()
3  ap = bml.ApproxContract(err=0.01, delta=0.05)
4  model = lr.fit(features, labels, ap)
```

(b) BlinkML (AccContract)

```
1  import blinkml as bml
2  lr = bml.LogisticRegression()
3  ap = bml.ApproxContract(n=10000, delta=0.05)
4  model = lr.fit(features, labels, ap)
```

(c) BlinkML (ScaleContract)

Figure 2: Examples of training a model in a typical ML library (scikit-learn in this case) versus BlinkML.

2 User Interaction

In this section, we describe how the user interacts with BlinkML.

2.1 Interface

In this section, we first describe the interface of a traditional ML library (e.g., scikit-learn [54], Weka [33], MLlib [1]), and then present the difference in BlinkML's interface.

Traditional ML Libraries As depicted in Figure 1 (top), with a typical ML library, the user provides a training set 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 $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$. The d-dimensional vector $\boldsymbol{x}_i$ 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*.

Figure 2a is an example of a code snippet that trains a logistic regression classifier using scikit-learn (i.e., the `sklearn` module) [54].

The trained model m_N is used differently depending on the type of learning. In supervised learning, $m(\boldsymbol{x})$ predicts a label for an unseen feature vector $\boldsymbol{x}$. For example, if $\boldsymbol{x}$ encodes a review of a restaurant, a trained logistic regression classifier $m_N(\boldsymbol{x})$ may predict whether the review is positive or negative. In unsupervised learning, $m_N()$ returns parameters that capture certain characteristics of the training set. For example, if D consists of bitmap images of handwritten digits, $m_N()$ might be a factor analysis model (e.g., PPCA) which returns a small number (q) of images that most accurately reconstruct the handwritten digit images when linearly combined [62]; here, the value of q is the model-specific configuration of PPCA. When the meaning is clear, we may omit the parameter (i.e., $\boldsymbol{x}$) of m_N and simply use $m_N(\cdot)$ or m_N.

BlinkML In addition to the inputs required by traditional ML libraries, BlinkML needs one extra input: an *approximation contract*. The approximation contract specifies the quality of the approximate model trained by BlinkML. The approximation contract is one of the following types: AccContract, which consists of an error bound ε and a confidence level δ, or ScaleContract, which consists of a sample size n and a confidence level δ. Figure 1 (middle and bottom) shows examples of the two user interaction scenarios with BlinkML. These contracts are defined as follows.

1. AccContract: Given an error bound ε and a confidence level δ, BlinkML returns an *approximate model* $m_{n'}$ such that the difference between $m_{n'}$ and the full model m_N is within ε with probability at least $1-\delta$. That is,

$$\Pr[v(m_{n'}, m_N) \leq \varepsilon] \geq 1 - \delta$$

where $v(\cdot, \cdot)$ is the *model difference*, a function that defines the difference between the two models.

The model difference depends on the ML task. For example, in supervised learning (i.e., classification or regression), the model difference captures the expected difference between the predictions of two models. That is,

$$v(m_1, m_2) = \mathrm{E}[m_1(\boldsymbol{x}) \neq m_2(\boldsymbol{x})] \qquad \textbf{(classification)}$$
$$v(m_1, m_2) = \mathrm{E}[(m_1(\boldsymbol{x}) - m_2(\boldsymbol{x}))^2] \qquad \textbf{(regression)}$$

For unsupervised learning (e.g., PPCA), the model difference captures the difference between the model parameters. For instance,

$$v(m_1, m_2) = 1 - \mathrm{cosine}(\theta_1, \theta_2) \qquad \textbf{(PPCA)}$$

where $\mathrm{cosine}(\cdot, \cdot)$ indicates the cosine similarity.

The approximate model $m_{n'}$ is trained on a sample of size n', and the value of n' is automatically inferred by BlinkML. Figure 2b shows an example of training an approximate model in BlinkML with AccContract. The values of ε and δ are specified when instantiating an object of type `ApproxContract`, which is then passed to the `fit` function to initiate the training.

2. ScaleContract: Given a sample size n and a confidence level δ, BlinkML returns (1) an *approximate model* m_n trained on a sample of size n, and (2) the estimated accuracy ε_n, such that the model difference between $m_n(\cdot)$ and $m_N(\cdot)$ is guaranteed to be within ε_n, with probability at least $1-\delta$. That is,

$$\Pr[v(m_n(\cdot), m_N(\cdot)) \leq \varepsilon_n] \geq 1 - \delta$$

Here, the meanings of ε_n and δ for classification and regression tasks are identical to AccContract. Figure 2c shows an example of training a model in BlinkML with ScaleContract.

Model Class	BlinkML Class Name
Linear regression	`LinearRegression`
Logistic regression	`LogisticRegression`
Max entropy classifier	`MaxEntropy`
PPCA	`PPCA`

Table 1: ML model classes currently available in BlinkML.

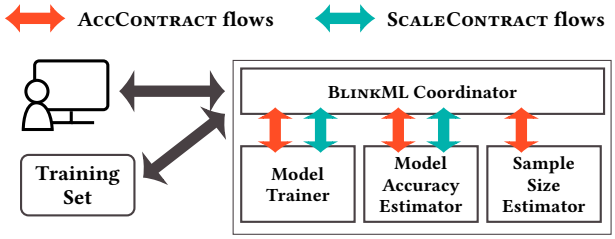


Figure 3: Architecture of BlinkML. All communications are through Coordinator. For AccContract, orange flows are used. For ScaleContract, green flows are used.

2.2 Supported ML 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 4 and 5) can be generalized to any ML algorithms that rely on maximum likelihood estimation. We are currently focused on adding more configuration options for these currently available models, but we also plan to extend our supported model classes in the future (e.g., naïve Bayes classifier and decision trees). Although neural networks are also quite popular, BlinkML's near-term vision is to focus on better understood models that can still be quite costly when faced with large datasets.

To use a specific model class, the user simply invokes the corresponding class name in BlinkML (see Figures 2b and 2c for an example of training a `LogisticRegression`). Table 1 summarizes the currently implemented model classes and their corresponding class names. **Note** that the users of BlinkML are *not* required to understand or derive any mathematical expressions in order to use (and benefit from) the approximation contracts offered by BlinkML for any of these supported models.

3 Workflow and Architecture

This section describes the internal components of BlinkML and how they operate together to produce an accuracy-guaranteed approximate model. As depicted in Figure 3, BlinkML has four main components: (1) Model Trainer, (2) Model Accuracy Estimator, (3) Sample Size Estimator, and (4) Coordinator. Those components are designed to be agnostic to the individual model, using an abstraction introduced in Section 3.2. We then describe the operations of each component in Section 3.3 and our sampling process in Section 3.4.

3.1 Internal Workflow

BlinkML's operations differ based on the approximation contract (AccContract or ScaleContract). Coordinator controls these different workflows, as follows.

AccContract First, Coordinator obtains a size-n_0 sample D_{n_0} of the training set D. We call D_{n_0} the *initial training set* (n_0 is 10K by default). Coordinator then invokes Model Trainer to train an

```python
class LogisticRegressionModel(ModelClass):

  def __init__(self, config, holdout_features):
    self._c = config  # ex. regularization coeff
    self._hold = holdout_features

  def diff(self, model1, model2):
    """
    Computes misclassification ratio.
    """
    y1 = model1.predict(self._hold)
    y2 = model2.predict(self._hold)
    return numpy.mean(y1 != y2)

  def grads(self, theta, features, labels):
    """
    Computes a list of per-example gradients
    """
    q = sigmoid(features.dot(theta)-labels)
    q = features * q
    r = self._c.beta * theta
    return q + r

  def solve(self, features, labels):
    """
    If available, this method directly computes
    optimal parameter values.
    """
    raise NotImplementedError
```

Figure 4: Example of a model class specification (MCS) in BlinkML.

initial model m_0 on D_{n_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 with AccContract.

ScaleContract First, Coordinator obtains a size-n sample D_n of the training set D, and invokes Model Trainer to train an approximate model m_n on D_n. Next, Coordinator invokes Model Accuracy Estimator to compute ε_n such that the model difference between m_n and m_N (i.e., the unknown full model) would not exceed ε_n with probability $1-\delta$. Finally, Coordinator returns m_n and ε_n to the user. Thus, with ScaleContract, only one approximate model is trained.

3.2 Model Abstraction

For generality and to be agnostic to the different ML models, BlinkML relies on an abstraction, called *model class specification (MCS)*. We first present the common mathematical form of the ML models supported by BlinkML, and then introduce our MCS abstraction.

Mathematical Form BlinkML currently supports any ML model that relies on maximum likelihood estimation. Such models are

trained by finding a model parameter value θ that maximizes the likelihood of the training set D. This optimization is equivalent to solving the following minimization problem:

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

where

$$f_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \log \Pr(\boldsymbol{x}_i, y_i;\, \theta) + R(\theta) \tag{2}$$

Here, $\Pr(\boldsymbol{x}_i, y_i;\, \theta)$ indicates the likelihood of observing the pair $(\boldsymbol{x}_i, y_i)$ 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 $\Pr(\boldsymbol{x}_i, y_i;\, \theta)$. For example, PPCA uses the Gaussian distribution with covariance matrix $\boldsymbol{x}_i\boldsymbol{x}_i^\top$ whereas logistic regression uses the Bernoulli distribution with its mean being the sigmoid function of $\boldsymbol{x}_i^\top\theta$.

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

$$g_n(\theta_n) = \left[\frac{1}{n}\sum_{i=1}^{n} q(\theta_n; \boldsymbol{x}_i, y_i)\right] + r(\theta_n) = \mathbf{0} \tag{3}$$

where $q(\theta; \boldsymbol{x}_i, y_i)$ denotes $\nabla_\theta \log \Pr(\boldsymbol{x}_i, y_i; \theta)$ and $r(\theta)$ denotes $\nabla_\theta R(\theta)$.

Model Class Specification (MCS) A model class specification is the abstraction that allows BlinkML's components to remain generic and not tied to the specific internal logic of the supported ML models. Note that BlinkML already includes the necessary MCS definitions for the currently supported model classes (see Section 2.2). Regular users therefore do not need to provide their own MCS. However, more advanced developers can easily integrate new ML models (as long as they are based on maximum likelihood estimation) by defining new MCS. To define a new MCS in BlinkML, the developers must inherit from the main `ModelClass` class and implement three virtual functions: `diff()`, `grads()`, and optionally, `solve()`.

1. `diff`(m_1, m_2): This function computes the model difference $v(m_1, m_2)$ between two models m_1 and m_2. For example, for classification (e.g., logistic regression, max entropy classifier), $v(m_1, m_2) = \mathrm{E}[m_1(\boldsymbol{x}) \neq m_2(\boldsymbol{x})]$ for $\boldsymbol{x} \sim \mathcal{D}$. For regression (e.g., linear regression), $v(m_1, m_2) = \mathrm{E}[(m_1(\boldsymbol{x}) - m_2(\boldsymbol{x}))^2]$ for $\boldsymbol{x} \sim \mathcal{D}$. For PPCA, $v(m_1, m_2)$ computes one minus the average cosine similarity between two models' respective extracted factors.
2. `grads`: This function computes and returns a list of $q(\theta; \boldsymbol{x}_i, y_i) + r(\theta)$ for $i = 1, \ldots, n$, 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 4.3).
3. `solve`: This optional function computes the optimal model parameters for models that have a closed-form expression. (Currently, BlinkML implements the optional `solve` functions for linear regression and PPCA, as they both have closed-form solutions.)

Figure 4 shows an example of a model class specification.

3.3 Architecture

As despited in Figure 3, BlinkML is comprised of four major components. We discussed the Coordinator's role in Section 3.1. Here, we discuss the remaining components: Model Trainer, Model Accuracy Estimator, and Sample Size Estimator.

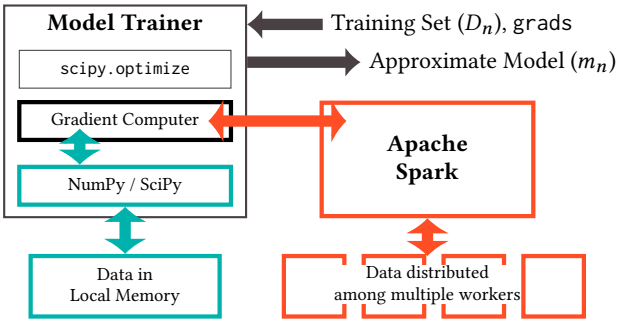


Figure 5: Iterative training in Model Trainer. Depending on the size of D_n, gradients are either computed locally (green flow) or using multiple workers in Apache Spark (orange flow).

3.3.1 *Model Trainer*

Model Trainer trains a model by finding its optimal parameter value, θ. Model Trainer takes two inputs: (1) a subset D_n of the training set D, where $n \leq N$, and (2) a model class specification (MCS).

To train and output a model m_n, Model Trainer takes a different approach depending on the size of D_n and the given model class specification, as follows.

1. Closed-Form If the MCS includes the optional `solve` function, Model Trainer simply relies on its `solve` function to find its optimal model parameters.

2. Iterative If the `solve` function is not present, BlinkML uses the `grads` function and the optimization libraries in the SciPy's `optimize` module [54]. To find the optimal parameter values, most optimization libraries require the gradient $\nabla f_n(\theta)$ of the objective function $f_n(\theta)$.[4] The gradient is computed by averaging the outputs of the `grads` function. When using SciPy's `optimize` module, computing the gradients is typically the dominant cost. Model Trainer therefore computes the `grads` function either locally or in a distributed fashion, as described next. (This is depicted in Figure 5.)

2.1. Iterative (Local) For small- and medium-scale problems ($n <$ 1M, by default), the `grads` function is computed locally using NumPy (for dense features) or SciPy (for sparse features).

2.2. Iterative (Distributed) For large-scale problems ($n \geq$ 1M, by default), BlinkML computes the gradients using multiple workers (via Apache Spark [65]). To distribute D_n between the remote workers, BlinkML packs each 10K training examples into a single element of Spark's RDD. Then, the `grads` function is invoked on each element of the RDD. According to our empirical studies, this enables Spark to exploit NumPy's highly-efficient matrix operations for each RDD element. Note that this distributed computation is possible only because the gradient expression in Equation (3) has a special structure that enables data-parallelism. That is, given θ, $q(\theta; \boldsymbol{x}_i, t_i)$ is independent of $q(\theta; \boldsymbol{x}_j, t_j)$ for $i \neq j$.

The output of Model Trainer is the model $m_n(\cdot) = m(\,\cdot\,; \theta_n)$, which includes the parameter θ_n trained on D_n. Figure 5 depicts Model Trainer's input and output.

3.3.2 *Model Accuracy Estimator*

Given an approximate model m_n, a confidence level δ, and an MCS, Model Accuracy Estimator computes ε_n, such that $v(m_n, m_N) \leq$

[4] The gradient of a function can be numerically approximated using the function itself; however, these approximations are much slower.

ε_n with probability at least $1-\delta$.

To achieve this goal, Model Accuracy Estimator exploits the fact that both m_n and m_N are essentially the same function (i.e., $m(\cdot)$) but with different model parameters (i.e., θ_n and θ_N). 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 4.1). Model Accuracy Estimator uses this estimated probability distribution to estimate m_N's probability distribution (or more accurately, the probability distribution of the output of $m_N(\cdot)$). The probability distribution of m_N is then used to estimate the distribution of $v(m_n, m_N)$, which is the quantity we aim to upper bound. The upper bound ε_n is determined by simply finding the value that is larger than $v(m_n, m_N)$ with probability at least $1-\delta$. We discuss this process more formally in Section 4.

3.3.3 *Sample Size Estimator*

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

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

3.4 Sampling Process

When the training set D fits in memory, BlinkML produces D_n by simply invoking `numpy.random.choice` to perform uniform sampling *without* replacement. When D does not fit in memory, BlinkML tries to use offline samples whenever possible, in the form of a pre-shuffled dataset on disk (similar to [34]) or a sequence of telescopic samples (similar to [12]). Otherwise, BlinkML performs Bernoulli sampling (as a more efficient approximation of uniform sampling without replacement) on-the-fly by reading from disk or querying a SQL database. For the latter, it uses the `TABLESAMPLE` operator when supported by the database, and otherwise issues the following SQL query: `select * from D where rand() < ratio`. Note that, although our theoretical contributions in Sections 4 and 5 are primarily designed for uniform random sampling, our results can be extended to biased (i.e., stratified) sampling by applying the asymptotic theory on biased sampling (instead of central limit theorem) [50].

4 Model Accuracy Estimator

Model Accuracy Estimator is the component that estimates the accuracy of an approximate model. This section describes its estimation process. To understand Model Accuracy Estimator, it is crucial to understand the statistical properties of Equation (3) from Section 3.2. In Section 4.1, we first establish these statistical properties. Then, in Section 4.2, we exokaun how BlinkML exploits these statistical properties to estimate the accuracy of an approximate model. Lastly, in Section 4.3, we show how to efficiently compute certain statistics required for expressing these properties.

4.1 Model Parameter Distribution

In this section, we present how to probabilistically express the parameter values of the (unknown) full model m_N only given an approximate model m_n. Let θ_n and θ_N be the parameters of m_n and m_N, respectively. Also, 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. In the remainder of this section, we first derive the distribution of $\hat{\theta}_n - \hat{\theta}_N$, and then use this distribution to derive the conditional distribution of $\hat{\theta}_N \mid \theta_n$. We will use this conditional distribution in the following section to estimate the accuracy of m_n.

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

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

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

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

The above theorem is a generalization of the sampling distribution of the maximum likelihood estimator [26, 48]. We defer the proof to Appendix B.

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,

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

as $n \to \infty$.

The following section uses the conditional probability distribution $\hat{\theta}_N \mid \theta_n$ to estimate the accuracy of an approximate model.

4.2 Probabilistic Error Bounds

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

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

$$\left(\int \mathbb{1}\left[v(m(\cdot\,; \theta_n), m(\cdot\,; \theta_N)) \leq \varepsilon_n \right] h(\theta_N)\, d\theta_N \right) \geq 1 - \delta \quad (4)$$

where $\mathbb{1}[\cdot]$ is the indicator function that returns 1 if its argument is true and returns 0 otherwise. The above expression involves blackbox functions such as $m(\cdot)$ and $v(\cdot\,, \cdot)$; thus, it cannot be analytically computed in general.[5]

Model Accuracy Estimator approximately computes the integration p_v in Equation (4) using the empirical distribution of $h(\theta_N)$

[5] We have also considered analytic computations of Equation (4) by relying on model-specific information; See the end of Section 4.2 for a discussion of these alternatives.

as follows. Let $\theta_{N,1}, \ldots, \theta_{N,k}$ be i.i.d. samples drawn from $h(\theta_N)$. Then,

$$p_v = \int \mathbb{1}\left[v(m(\cdot;\theta_n),\, m(\cdot;\theta_N)) \le \varepsilon_n\right] h(\theta_N)\, d\theta_N \tag{5}$$

$$\approx \frac{1}{k}\sum_{i=1}^{k} \mathbb{1}\left[v(m(\cdot;\theta_n),\, m(\cdot;\theta_{N,i})) \le \varepsilon_n\right] = \tilde{p}_v \tag{6}$$

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

Accuracy Now we analyze the accuracy of using Equation (6). For our analysis, we use the expected relative root mean square (rms) error of $\tilde{p}_v$ (in comparison to p_v). Observe that since $\hat{\theta}_{N,1}$, $\ldots$, $\hat{\theta}_{N,k}$ are i.i.d. samples, $\tilde{p}_v$ follows a binomial distribution with mean p_v and standard deviation $\sqrt{p_v(1-p_v)/k}$. Thus, the expected relative rms error is $\sqrt{(1/p_v - 1)/k}$. For instance, when $p_v = 0.90$ and $k = 1{,}000$, the relative rms error is about 0.01. BlinkML uses 1,000 for k by default and targets the cases where p_v is not smaller than 0.90 (i.e., $\delta \le 0.10$). Thus, the expected relative rms error of $\tilde{p}_v$ is small.

Alternative Approaches We briefly describe why using model-specific information does not necessarily lead to a more efficient computation of Equation (5). For our description, we use logistic regression as an example. Since logistic regression is a classification algorithm, the model difference between $m_n = m(\boldsymbol{x};\theta_n)$ and $m_N = m(\boldsymbol{x};\theta_N)$ is computed by $\mathrm{E}[m(\boldsymbol{x};\theta_n) \ne m(\boldsymbol{x};\theta_N)]$, where the model prediction of logistic regression is $m(\boldsymbol{x};\theta) = \mathbb{1}\left[\theta^\top \boldsymbol{x} \ge 0\right]$. Thus,

$$p_v = \int \mathrm{E}[m(\boldsymbol{x};\theta_n) = m(\boldsymbol{x};\theta_N)]\, h(\theta_N)\, d\theta_N$$

$$= \frac{1}{\ell}\sum_{j=1}^{\ell}\left(\int_{A_+} \mathbb{1}\left[\theta_n^\top \boldsymbol{x}_j \ge 0\right] dh(\theta_N) + \int_{A_-} \mathbb{1}\left[\theta_n^\top \boldsymbol{x}_j < 0\right] dh(\theta_N)\right)$$

where ℓ is the number of training examples used for computing the expectation, A_+ is the area of θ_N such that $\theta_N^\top \boldsymbol{x}_j \ge 0$, and A_- is the area of θ_N such that $\theta_N^\top \boldsymbol{x}_j < 0$. The above integration computes the probability that the full model's prediction for $\boldsymbol{x}_j$ lies on the same side of the decision boundary (i.e., $\theta^\top \boldsymbol{x}_j \ge 0$ or $\theta^\top \boldsymbol{x}_j < 0$) as the approximate model's prediction for $\boldsymbol{x}_j$. Computing the above integration is non-trivial because θ_N is multidimensional and the areas, A_+ and A_-, cannot be analytically expressed in general. Moreover, integrating multivariate normal distribution, $h(\theta_N)$, cannot be computed analytically. This difficulty becomes even more severe for multi-class classification algorithms, such as max entropy classifier. In contrast, Model Accuracy Estimator's current approach easily generalizes to all supported ML models, while its runtime overhead is still low (see Section 6.5 for empirical study of runtime overhead).

4.3 Computing Necessary Statistics

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

Method 1: ClosedForm ClosedForm uses the analytic form of the Jacobian $H(\theta)$ of $g_n(\theta)$, and 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 $\boldsymbol{x}_i$, and Q is a d-by-d diagonal matrix whose i-th diagonal entry is $\sigma(\theta^\top \boldsymbol{x}_i)(1 - \sigma(\theta^\top \boldsymbol{x}_i))$, 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. Thus, BlinkML supports ClosedForm only for linear regression and logistic regression. For other methods, such as max entropy classifier and PPCA, either of the following two methods is used.

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

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

The values of $g_n(\theta_n + d\theta)$ and $g_n(\theta_n)$ 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 $g_n(\theta_n + P_{\cdot,i})$ where $P_{\cdot,i}$ is the i-th column of P. Then,

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

Since InverseGradients only relies on the `grads` function, it is applicable to all supported models. Although InverseGradients is accurate, 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 6.6.

Method 3: ObservedFisher ObservedFisher numerically computes H by relying on the information matrix equality [26]. According to the information matrix equality, the covariance matrix C of $q(\theta_n;\boldsymbol{x}_i,y_i)$ for $i = 1,\ldots,n$ is equal to J; then, H can simply be computed as $H = J + J_r$. Instead of computing C directly, however, ObservedFisher takes a slightly different approach to ensure the positive definiteness of J and H. This process also ensures the positive definiteness of $H^{-1}JH^{-1}$, which is the requirement for the covariance matrix of a normal distribution. Let Q be the n-by-d matrix whose i-th row is $q(\theta_n;\boldsymbol{x}_i,y_i)$. 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, ObservedFisher computes J as follows:

$$J = C = Q^\top Q = U\,\Sigma^2\,U^\top \approx U\,\Sigma_+^2\,U^\top \tag{7}$$

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

ObservedFisher requires only a single call of the `grads` function; thus, ObservedFisher is faster than InverseGradients, particularly for high-dimensional data (see Section 6.6 for experiments). Also, ObservedFisher exposes U and Σ_+, which are used for BlinkML's

custom sampler (Section 5). Due to these reasons, ObservedFisher is BlinkML's default approach to computing H and J.

5 Sample Size Estimator

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

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

5.1 Estimating Model Quality sans Training

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

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

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

$$\approx \frac{1}{k}\sum_{i=1}^{k} \mathbb{1}\left[v(m(\cdot\,;\theta_{n',i}), m(\cdot\,;\theta_{N,i})) \leq \varepsilon\right] = \tilde{p}_v(n'0) \tag{8}$$

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

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

where the conditional distributions, $\theta_N \mid \theta_{n'}$ and $\theta_{n'} \mid \theta_0$, are obtained using Corollary 1. That is, Model Accuracy 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 $i = 1, \ldots, k$ to obtain k pairs of $(\theta_{n',i}, \theta_{N,i})$. Finally, $\tilde{p}_v(n')$ is obtained by counting the fraction of the cases in which $\mathbb{1}\left[v(m(\cdot\,;\theta_{n',i}), m(\cdot\,;\theta_{N,i})) \leq \varepsilon\right]$ returns 1.

5.2 Sample Size Searching

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

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

Theorem 2. *Let $h(\theta;\gamma)$ 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 $v(m(\cdot\,;\theta), m(\cdot\,;\theta_N)) \leq \varepsilon$. Then, the following function $p_v(\gamma)$*

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

is a decreasing function of γ.

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

5.3 Optimizations for Fast Sampling

This section describes Sample Size Estimator's sampling mechanism. For computing $p_v(n')$, Sample Size Estimator relies on the i.i.d. samples from the normal distribution with covariance matrix $(1/n' - 1/N)\, H^{-1}JH^{-1}$. 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, which enables much faster sampling operations.

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

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

This indicates that we can first draw i.i.d. samples from the unscaled distribution $\mathcal{N}(\mathbf{0}, H^{-1}JH^{-1})$; then, we can scale those sampled values by $\sqrt{1/n - 1/N}$ whenever the i.i.d. samples from $\mathcal{N}(\mathbf{0}, (1/n - 1/N)\, H^{-1}JH^{-1})$ 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). Note that sampling from the standard normal distribution is much faster because no dependencies need to be enforced among sampled values.

For this, the following relationship is used:

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

That is, if there exists L such that $LL^\top = H^{-1}JH^{-1}$, Sample Size Estimator 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 (7) that $J = U\,\Sigma_+^2\,U^\top$. Since $H = J + \beta$, $H = U\,(\Sigma_+^2 + \beta I)\,U^\top$. Thus,

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

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

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

Table 2: Key statistics of the datasets used in our experiments.

6 Experiments

In addition to our theoretical guarantees in Sections 4 and 5, we also evaluate BlinkML empirically to answer several questions:

1. How faster is approximate training in practice?
2. Does BlinkML meet the user-requested accuracies?
3. Is BlinkML's estimated minimum sample size optimal?
4. How large is the runtime overhead of BlinkML?

Overall, our experimental results show the following:

1. BlinkML reduced training time by 35.03%–99.69% (i.e., 1.54×–322× faster) when training 99% accurate models. (Section 6.2)
2. The actual accuracy of BlinkML's approximate models was in most cases even higher than the requested accuracy. (Section 6.3)
3. BlinkML's estimated minimum sample sizes were close to optimal. In fact, BlinkML's training time was only slightly higher than an oracle that had the *perfect knowledge* of the optimal sample size *a priori*. (Section 6.4)
4. The runtime overheads of BlinkML was only 0.39%–0.44% for AccContract and 0.03%–0.12% for ScaleContract compared to the time needed for training a full model. (Section 6.5)
5. We also studied the pros and cons of the two statistics computation methods: InverseGradients and ObservedFisher. (Section 6.6)

6.1 Experiment Setup

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

Models We used three different ML models in our evaluations:

1. **Logistic Regression (LR).** LR is a binary classifier. LR is trained by finding parameter values of a sigmoid prediction function such that the misclassification rate is minimized on the training set. To avoid overfitting, LR is often trained with regularization. We used L2-regularized LR with coefficient $\beta = 0.001$. Given that LR has no closed-form solution, it is typically trained using an iterative optimization method (e.g., the optimization libraries in `scipy.optimize`).
2. **Max Entropy Classifier (ME).** ME is a multi-class classifier. ME is trained by finding parameter values of a softmax prediction function such that the misclassification rate is minimized on the training set. Similar to LR, we used a L2-regularized ME with regularization coefficient $\beta = 0.001$, and since there is no closed-form solution for ME, we used the optimization libraries in `scipy.optimize`.
3. **Probabilistic Principal Component Analysis (PPCA).** PPCA is a factor-analysis method. PPCA is trained by finding a small number (q) of vectors that can most accurately reconstruct the training set using linear transformations [62]. The optimal parameters (i.e., factors) can be found by performing eigendecomposition of a sample covariance matrix constructed from the training set. For PPCA, we disabled the use of Apache Spark since computing the covariance locally was significantly faster.

As mentioned earlier, an optimization method must be chosen for LR and ME. Based on our empirical studies, BlinkML uses the BFGS optimization when the dataset is low-dimensional ($d < 100$). For high-dimensional datasets ($d \geq 100$), BlinkML uses a memory-efficient alternative called L-BFGS.

Datasets We have used three real-world datasets.

1. **HIGGS.** This is a 7× scaled version of a Higgs bosons simulation dataset [15]. Each training example is a pair of physical properties of an environment and a binary indicator of Higgs bosons production.
2. **Yelp.** This is a 10% subset of the publicly available Yelp reviews [10]. Each training example is a pair of an English review and a rating (between 0 and 5). The original dataset contains 737,530 distinct words; we performed feature selection and used the 1,000 most frequently used words. When applying LR to `Yelp`, we formulated a binary classification task by converting ratings of 0–2 to 0 (negative) and ratings of 3–5 to 1 (positive).
3. **MNIST.** This is an image dataset (a.k.a. infinite MNIST [5]), which contains the bitmap images of hand-written digits. Each training example is a pair of a bitmap image and the actual digit in the image. When applying PPCA on `MNIST`, we used q=10.

Several key statistics of these datasets are summarized in Table 2.

Environment All our experiments were conducted on an EC2 cluster with one `m5.4xlarge` node as a master and five `m5.2xlarge` nodes as workers.[6] We used Python 3.6 shipped with Conda [2] and Apache Spark 2.2 shipped with Cloudera Manager 5.11.

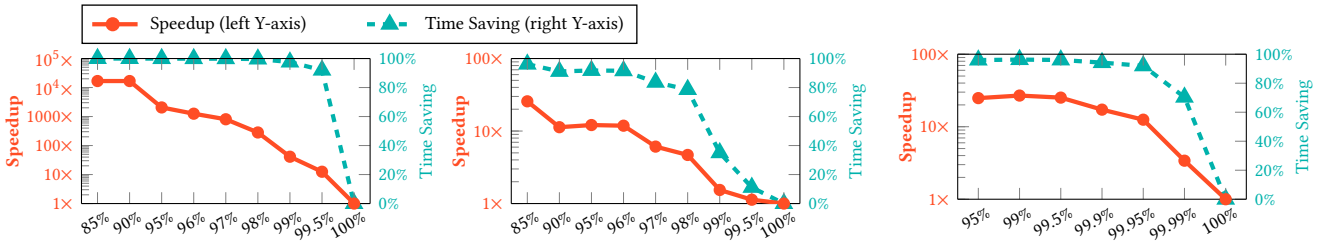
6.2 Training Time Savings

This section measures the time savings when one uses BlinkML's approximate model training (AccContract), compared to training the full model . We used eight model and dataset combinations: (LR, `HIGGS`), (LR, `Yelp`), (ME, `HIGGS`), (ME, `Yelp`), (ME, `MNIST`), (PPCA, `HIGGS`), (PPCA, `Yelp`), and (PPCA, `MNIST`). For LR and ME, we varied the requested accuracy $(1-\varepsilon)\times 100\%$ from 85% to 99.5%. For PPCA, we varied the requested accuracy ($(1-\varepsilon)\times 100\%$) from 95% to 99.99%. We fixed δ at 0.05. For each case, we repeated BlinkML's training 20 times.

Due to space constraints, Figure 6 shows the average training time of those 20 runs only for three combination, i.e., (LR, `HIGGS`), (ME, `Yelp`), and (PPCA, `MNIST`). The results for other combinations are reported in Appendix C. The figure reports BlinkML's both speedups and training time savings compared to training the full model. The full model training times were 1,040 seconds for (LR, `HIGGS`), 1,176 seconds for (ME, `Yelp`), and 61 seconds for (PPCA, `MNIST`). Full models were all trained using Spark (except for PPCA, as explained in Section 6.1). The use of Apache Spark made the training of the full model 1.5×–6.5× faster.

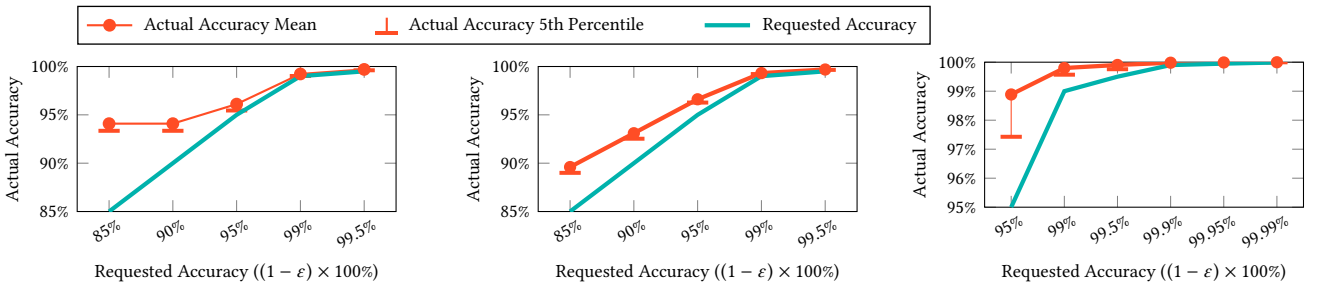
Three patterns were observed. First, as expected, BlinkML took relatively longer for training a more accurate approximate model. This is because BlinkML's Sample Size Estimator correctly estimated that a larger sample was needed to satisfy the requested accuracy. Second, the relative training times were longer for complex models (ME took longer than LR). This is because multi-class classification (by ME) involved more possible class labels; thus, even a small error in parameter values could lead to misclassification. Thus, a larger sample was needed to sufficiently upper bound the chances of a misclassification. Nevertheless, training 95% accurate ME models on `Yelp` still took only 8.25% of the time needed for training a full model. In other words, even in this case, we observed a 12.12× speedup, or equivalently, 91.75% savings in training time. Third, the speedups for training PPCA models were significantly larger than those for LR and ME. This was because of the fact that, unlike LR and ME, PPCA does not involve any hard

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



(a) Logistic Regression (LR), HIGGS (b) Max Entropy Classifier (ME), Yelp (c) Probabilistic PCA (PPCA), MNIST

Figure 6: BlinkML's speedups in comparison to full model training (the actual accuracies of the trained models are studied in Figure 7).



(a) LR, HIGGS (b) ME, Yelp (c) PPCA, HIGGS

Figure 7: The correctness of BlinkML's AccContract operations. 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. The raw data of this figures are in Table 5.

decision boundaries; thus, the minimum sample sizes estimated by BlinkML's Sample Size Estimator were much smaller (see Section 6.4 for details). As a result, the relative training times were lower than LR and ME. In all cases, BlinkML's ability to automatically infer appropriate sample sizes and train approximate models on those samples led to significant training time savings. In the subsequent section, we analyze the actual accuracies of those approximate models.

6.3 Accuracy Guarantees

As stated in Equation (8) in Section 5, 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 85% to 99.5% and requested a confidence level of 95%, i.e., $\delta = 0.05$. The results are shown in Figure 7 for the same combinations of models and datasets used in the previous section. (The raw numbers can be found in Table 5.)

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.

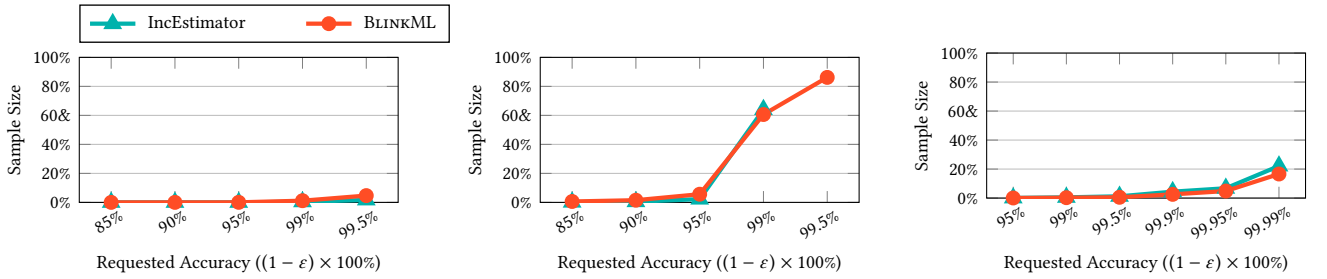
Notice that, in the case of (LR, HIGGS), the actual accuracies remained identical even though the requested accuracies were different. This was due to BlinkML's design. Recall that BlinkML first trains an initial model m_0 and then trains a subsequent model only when the estimated model difference ε_0 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.

6.4 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. Our examination consists of two parts. First, we analyze the optimality of the estimated minimum sample sizes. Then, we study the implication of the estimated sample sizes, namely the optimality of BlinkML's training time.

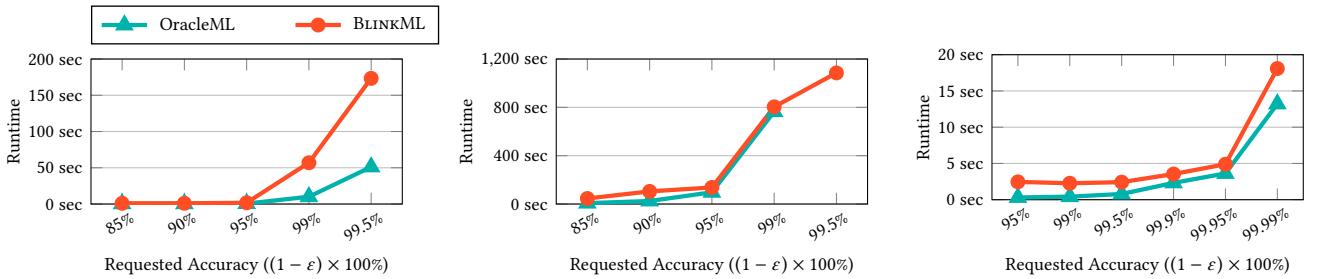
Sample Size Optimality To analyze the optimality of the sample sizes, we additionally implemented IncEstimator, another algorithm that estimates the minimum sample size, but much more accurately. Specifically, to calculate the minimum sample size, IncEstimator gradually increases the sample size until the approximate model trained on that sample satisfies the requested accuracy. The sample size is set to $k^2 \times 1,000$, where k is the iteration count starting from 1. Since IncEstimator repeatedly trains and measures the accuracy of multiple approximate models, its estimated minimum sample size is much more accurate than SSE; however, its runtime is also significantly slower.

Figure 8 compares the minimum sample sizes estimated by SSE (BlinkML's technique) and IncEstimator for three different model and dataset combinations, i.e., (LR, HIGGS), (ME, Yelp), and (PPCA, MNIST). Needless to mention, IncEstimator was too slow. In one case (ME, 99.5% accuracy), it did not finish within 24 hours, and we had to terminate it. For PPCA, IncEstimator and SSE found



(a) LR, HIGGS (b) ME, Yelp (c) PPCA, MNIST

Figure 8: Goodness of BlinkML's estimated minimum sample sizes. BlinkML's estimated minimum sample sizes were compared to the minimum sample sizes determined by IncEstimator. The minimum sample sizes estimated by BlinkML were close to the ones determined by IncEstimator, a slow incremental searching technique.



(a) LR, HIGGS (b) ME, Yelp (c) PPCA, MNIST

Figure 9: Optimality of BlinkML's end-to-end performance. BlinkML's training time was compared to OracleML. OracleML is assumed to be able to predict the minimum sample size without any runtime overhead.

nearly identical sample sizes. For LR and ME, however, IncEstimator found a sample that was 2.3×–2.7× smaller than that of SSE. We show shortly that these relatively small differences in the estimated minimum sample sizes lead to only marginal differences in terms of the end-to-end training time. Overall, this experiment confirmed that the minimum sample sizes returned by SSE were quite close to optimal.

Training Time Optimality To study the impact of the different sample sizes, we also measured the training times. Given a requested accuracy, we compared the training time on BlinkML's estimated minimum sample size to the training time on the optimal sample size (as found by IncEstimator). We refer to the latter as OracleML, since it cannot be used in practice. This is because computing the exact sample size using IncEstimator is too costly, due to its iterative process. Nonetheless, comparing BlinkML's training time to that of OracleML helps in understanding how far BlinkML's performance is from optimal.

Figure 9 shows the results. Here, for BlinkML, we included its overhead of estimating the minimum sample size, whereas we excluded IncEstimator's overhead from OracleML's runtime. In most cases, the overall training time of BlinkML was quite close to that of OracleML, showing that BlinkML's SSE is both efficient and accurate. (We separately analyze SSE's overhead in Section 6.5).

6.5 Overhead Analysis

This section analyzes BlinkML's runtime overhead, i.e., the time spent on its internal operations other than model training. Specifically, for AccContract, we measured the time taken by computing statistics (Section 4.3) and searching the sample size (Section 5.2). For ScaleContract, we measured the time taken by computing statistics (Section 4.3) and estimating accuracy (Section 4.2). Recall that AccContract always involves the accuracy estimation of the initial model; in our measurements, we included this latency as part of the search for determining the sample size.

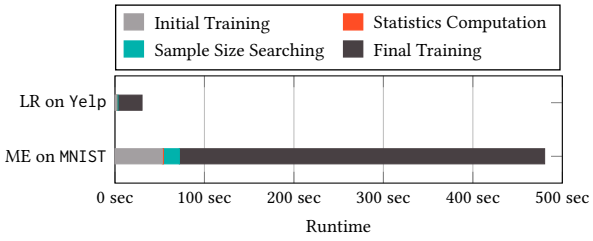
For AccContract, we requested 95% accurate models for different combinations of models and datasets. The results are reported in Figure 10a, where we only show (LR, Yelp) and (ME, MNIST) due to the lack of space. Here, the gray and black bars are the times spent for model training, while the red and green bars are BlinkML's overhead. The majority of BlinkML's overhead was for sample size searching. However, the overall overhead was still a small proportion of the entire training. For LR and ME, for instance, BlinkML's overhead was only 5.2% and 4.0% of the overall training, respectively.

To analyze the overhead of ScaleContract, we requested approximate models using samples of size 10,000. As shown in Figure 10b, when the entire training time was short (i.e., 2.25 secs for LR, Yelp), BlinkML's overhead was a more considerable portion (i.e., 22.4%); however, its absolute time was still very short (0.50 secs). When the entire training time was longer (i.e., 54.6 secs for ME, MNIST), BlinkML's overhead was only 3.0% of the entire training time (i.e., 1.64 secs).

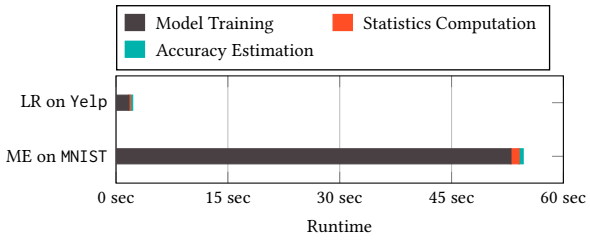
6.6 Statistics Computation

In Section 4.3, we presented two general methods—InverseGradients and ObservedFisher—for computing important statistics, i.e., H and J. In this section, we compare them in terms of both efficiency and accuracy. This analysis serves as an empirical justification of why ObservedFisher is BlinkML's default choice.

In this experiment, we used two combinations (LR, HIGGS) and (ME, MNIST). Recall from Table 2 that HIGGS is a low-dimensional dataset ($d = 28$) while MNIST is high-dimensional ($d = 784$). This difference in number of dimensions reveals interesting properties regarding InverseGradients versus ObservedFisher. We used each



(a) AccContract: The requested confidence was 95%.



(b) ScaleContract: The specified sample size was 10,000.

Figure 10: BlinkML's training time decomposition for both AccContract and ScaleContract. BlinkML's runtime overhead for computing necessary statistics, searching minimum sample sizes, and estimating confidence were small compared to the overall training time.

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

Table 3: Comparison of statistics computation techniques: InverseGradients and ObservedFisher. The accuracies of computed statistics were measured using the mean Frobenius norm of the difference between the true statistics and the estimated statistics. ObservedFisher was faster for MNIST, a high-dimensional dataset, while maintaining comparable accuracy to InverseGradients.

method to compute the covariance matrix $H^{-1}JH^{-1}$ (which determines the parameter distribution in Theorem 1, Section 4). 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.

Table 2 summarizes the results of this experiment. For the low-dimensional data (HIGGS), the runtimes and the accuracies of the two methods were comparable. However, their performance differed much for the high-dimensional data (MNIST). Since InverseGradients had to invoke the grads function repeatedly (i.e., d times), its runtime increased significantly. In contrast, ObservedFisher had to call the grads function only once. However, their accuracies remained comparable even for the high-dimensional data.

7 Related Work

In this section, we discuss the related work under three categories: ML systems inspired by familiar database techniques (DBMS-Inspired Optimization), ML systems employing different forms of approximation (Approximate ML Systems), and the recent advances in statistical optimization methods (Faster Optimization Algorithms). The developments in the last category are mostly orthogonal to BlinkML (or ML systems in general). However, ML systems can still benefit from employing these developments.

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 [17,30] optimizes linear algebra expressions by converting them into alternative expressions that enable a more efficient evaluation. They also propose new compression algorithms for array data [27]. ScalOps [63], Pig latin [49], and KeystoneML [59] propose high-level ML languages for automatic parallelization and materialization, as well as easier programming. MADLib [20, 35] proposes a declarative language for ML and exploits existing database engines, hence moves the computation closer to the data than external ML libraries. Similarly, RIOT [70] is an R interface that exploits in-database computing.

The database community has also developed array-based database systems, which can train a model even when the data does not fit in memory [37, 60]. Database researchers have also shown that certain ML models can be trained directly on the normalized data without the need for a join [41, 56].

Another line of work automatically chooses an optimal plan for a given ML workload, e.g., Hemingway [51], MLBase [39], and TuPAQ [58]. Likewise, Kumar et al. [40] use a model selection management system to unify feature engineering [13], algorithm selection, and parameter tuning.

Approximate ML Systems BlinkML is the first sampling-based ML system, applicable to any MLE-based (maximum likelihood estimation) algorithm. However, other forms of approximation have been previously explored in the database community for speeding up ML workloads. Hamlet [42] avoids expensive denormalizations if the quality gain of performing a join is estimated to be low. Perhaps the most related to BlinkML is Columbus [67], which trains approximate models on a coreset, a representative subset of the training data. Although Columbus also uses a sample of the data to speed up training, they can only provide an accuracy guarantee for ordinary least squares methods. In contrast, BlinkML supports a much wider class of problems; in fact, the classification methods and the factor analysis technique used in our experiments include any non-linear objective functions, none of which can be approximated with guarantees by Columbus. Likewise, Zombie [14] employs a clustering-based active learning technique for training approximate models, but does not offer any quality guarantees.

There are other methods that speed up the prediction time at the cost of increased preprocessing. NoScope [36] first trains multiple deep neural networks and then composes a different cascade depending on the target accuracy. tKDC [29] first builds a k-d tree and then conducts early pruning at the prediction stage of a kernel-density classifier. BlinkML differs from these methods by focusing on speeding up the training phase, which is a crucial bottleneck in early stages of model tuning.

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. Recent advances include adaptive rules for accelerating this rate, e.g., Adagrad [25], Adadelta [66], RMSprop [61], and Adam [38]. Hogwild! [55] and related techniques [31,32,69] 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 [43].

Hardware-based techniques speed up training by relaxing strict precision requirements, e.g., DimmWitted [68] and BuckWild! [23].

8 Conclusion

In this work, we have developed BlinkML, a novel approximate machine learning system with probabilistic guarantees. BlinkML uses sampling to quickly trains an approximate ML model in accordance to user's error tolerance, to dramatically reduce time and computational costs during the early stages of model tuning. BlinkML's techniques are applicable to a wide class of ML models, i.e., any algorithm that relies on maximum likelihood estimation for their training. This includes Generalized Linear Models (e.g., linear regression, logistic regression (LR), max entropy classifier (ME), Poisson regression, etc.) and Probabilistic Principal Component Analysis (PPCA). Through an extensive set of experiments on several large-scale, real-world datasets, we showed that BlinkML produced 99% accurate models of LR, ME, and PPCA while using only 0.31%–64.97% of the time needed for training the full model. Our future plan is to extend BlinkML's capabilities beyond maximum likelihood estimation models, such as decision trees, Gaussian Process regression, and Naïve Bayes classifiers. We also plan to open-source BlinkML, with wrappers for various popular ML libraries, including scikit-learn (Python), `glm` (R), and MLlib.

9 References

[1] Apache spark's scalable machine learning library. https://spark.apache.org/mllib/. Retrieved: April 25, 2018.

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

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

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

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

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

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

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

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

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

[11] 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.

[12] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[13] 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.

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

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

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

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 2009.

[34] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[35] 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.

[36] 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.
[37] 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.
[38] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[39] 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.
[40] 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.
[41] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984, 2015.
[42] 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.
[43] 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.
[44] 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.
[45] 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.
[46] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT press, 2012.
[47] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Eng. Bull.*, 2015.
[48] W. K. Newey and D. McFadden. Large sample estimation and hypothesis testing. *Handbook of econometrics*, pages 2111–2245, 1994.
[49] 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.
[50] A. B. Owen. *Monte Carlo theory, methods and examples.* 2013.
[51] X. Pan, S. Venkataraman, Z. Tai, and J. Gonzalez. Hemingway: modeling distributed optimization algorithms. *arXiv*, 2017.
[52] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 4, 2011.
[53] Y. Park, A. S. Tajik, M. Cafarella, and B. Mozafari. Database Learning: Towards a database that becomes smarter every time. In *SIGMOD*, 2017.
[54] 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.
[55] 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.
[56] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, pages 3–18, 2016.
[57] 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.
[58] 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.
[59] 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.
[60] 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.
[61] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. *University of Toronto, Tech. Rep*, 2012.
[62] M. E. Tipping and C. M. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 1999.
[63] M. Weimer, T. Condie, R. Ramakrishnan, et al. Machine learning in scalops, a higher order cloud computing language. In *BigLearn*, pages 389–396, 2011.
[64] E. Xing. Vc dimension and model complexity. https://www.cs.cmu.edu/~epxing/Class/10701/slides/lecture16-VC.pdf. Retrieved: April 25, 2018.
[65] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[66] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.
[67] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD*, pages 265–276, 2014.
[68] C. Zhang and C. Ré. Dimmwitted: A study of main-memory statistical analytics. *Proceedings of the VLDB Endowment*, 7(12):1283–1294, 2014.
[69] 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.
[70] Y. Zhang, W. Zhang, and J. Yang. I/o-efficient statistical computing with riot. In *ICDE*, pages 1157–1160, 2010.

APPENDIX

A Model Abstraction Examples

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

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

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

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

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

It is straightforward to cast the above expression into Equation (3). That is, $q(\theta; \boldsymbol{x}_i, t_i) = (\sigma(\theta^\top \boldsymbol{x}_i) - t_i)\,\boldsymbol{x}_i$ and $r(\theta) = \beta\,\theta$.

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

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

where d is the dimension of feature vectors. Θ is d-by-q matrix in which each column represents a factor, and σ is a real-valued scalar that represents the noise in data not explained by those factors. The optimal value for σ can be obtained once the values for Θ are determined. The value of q, or the number of the factors to

extract, is a user parameter. The above expression $f_n(\Theta)$ is minimized when its gradient $\nabla_\Theta f_n(\Theta)$ becomes a zero vector; that is,

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

The above expression can be cast into the form in Equation (3) by observing $S = (1/n)\sum_{i=1}^{n} \boldsymbol{x}_i \boldsymbol{x}_i^\top$.[7] That is, $q(\Theta; \boldsymbol{x}_i) = C^{-1}\Theta - C^{-1}\boldsymbol{x}_i\boldsymbol{x}_i^\top C^{-1}\Theta$ and $r(\Theta) = 0$. t_i is omitted on purpose since PPCA does not need observations (e.g., class labels). Although we used a matrix form Θ in the above expression for simplicity, BlinkML internally uses a vector θ when passing parameters among components. The vector is simply flattened and unflattened as needed.

B Deferred Proofs

B.1 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 [48]. The generalization is required since the original result does not include $r(\theta)$.

Let θ_∞ be the parameter values at which $g_\infty(\theta)$ becomes zero. Since the size of the training set is only N, θ_∞ exists only conceptually. Since θ_n is the optimal parameter values, it satisfies $g_n(\theta_n) = \mathbf{0}$. According to the mean-value theorem, there exists $\bar{\theta}$ between θ_n and θ_∞ that satisfies:

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

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

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

$$\begin{aligned}\sqrt{n}\left(\hat{\theta}_n - \theta_\infty\right) &= -H(\bar{\theta})^{-1}\,\sqrt{n}\,g_n(\theta_\infty) \\ &= -H(\bar{\theta})^{-1}\,\frac{1}{\sqrt{n}}\sum_{i=1}^{n}\left(q(\theta_\infty; \boldsymbol{x}_i, y_i) + r(\theta_\infty)\right) \quad (9) \\ &\xrightarrow{n\to\infty} \mathcal{N}(\mathbf{0},\, H^{-1}JH^{-1}) \quad (10)\end{aligned}$$

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 $q(\theta; \boldsymbol{x}_i, y_i)$ 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 $\mathcal{N}(\mathbf{0},\, H^{-1}JH^{-1})$. From Equation (10), $\hat{\theta}_n - \theta_\infty = (1/\sqrt{n})\,X_1$. Also, let $q_i = q(\theta_\infty; \boldsymbol{x}_i, y_i)$ for simplicity; then,

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

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

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

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

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

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

□

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:

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

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

$$\hat{\theta}_n \sim \left(\theta_N,\, \alpha\,H^{-1}JH^{-1}\right) \quad (11)$$

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

B.2 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,

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

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

C Raw Experiment Data

Fine the tables in the following pages.

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

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

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

Table 4: Training time savings. This is the raw data for Figure 6.

LR, HIGGS				LR, Yelp			
Requested Accuracy	Actual Accuracy			Requested Accuracy	Actual Accuracy		
	Mean	5th Percentile	95th Percentile		Mean	5th Percentile	95th Percentile
80.0%	94.09%	93.35%	95.13%	80.0%	94.23%	93.73%	94.68%
85.0%	94.09%	93.35%	95.13%	85.0%	94.23%	93.73%	94.68%
90.0%	94.09%	93.35%	95.13%	90.0%	96.03%	94.09%	97.39%
95.0%	96.11%	95.45%	96.72%	95.0%	98.06%	96.97%	98.65%
99.0%	99.37%	98.80%	99.63%	99.0%	99.67%	99.61%	99.78%
99.5%	99.70%	99.61%	99.82%	99.5%	99.82%	99.79%	99.87%

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

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

Table 5: The comparison of requested model accuracies (in AccContract) to the the actual model accuracies. This is the raw data for Figure 7.

LR, HIGGS			ME, Yelp			PPCA, MNIST		
Requested Accuracy	**QudraticSearch** Size	**SSE** Size	**Requested Accuracy**	**QudraticSearch** Size	**SSE** Size	**Requested Accuracy**	**QudraticSearch** Size	**SSE** Size
80.0%	1,000	10,000	80.0%	9,000	23,453	0.9	4,000	2,785
85.0%	4,000	10,000	85.0%	16,000	62,591	0.95	9,000	5,531
90.0%	4,000	10,000	90.0%	49,000	150,622	0.99	49,000	27,224
95.0%	16,000	33,529	95.0%	225,000	563,784	0.995	121,000	54,134
99.0%	324,000	1,073,803	99.0%	6,400,000	6,060,729	0.999	441,000	256,239
99.5%	1,296,000	4,515,812	99.5%	7,500,000	8,615,240	0.9995	676,000	484,703
						0.9999	2,209,000	1,663,136

Table 6: The comparison of the sample sizes estimated by IncEstimator and BlinkML's Sample Size Estimator (SSE). This is the raw data for Figure 8.

LR, HIGGS				ME, Yelp			
Requested Accuracy	**QudraticSearch** Time	**SSE** Time	**OracleML** Time	**Requested Accuracy**	**QudraticSearch** Time	**SSE** Time	**OracleML** Time
80.0%	0.112 sec	0.06 sec	0.27 sec	80.0%	6.87 sec	7.01 sec	4.11 sec
85.0%	0.287 sec	0.98 sec	0.33 sec	85.0%	14.1 sec	45.7 sec	7.33 sec
90.0%	0.285 sec	1.05 sec	0.33 sec	90.0%	67.2 sec	104 sec	25.1 sec
95.0%	0.837 sec	1.72 sec	0.60 sec	95.0%	775.0 sec	137 sec	97.0 sec
99.0%	72.2 sec	56.9 sec	10.1 sec	99.0%	125,509 sec	804 sec	764 sec
99.5%	636 sec	173 sec	51.2 sec	99.5%	N/A (timeout)	1,085 sec	1,045 sec

PPCA, MNIST			
Requested Accuracy	**QudraticSearch** Time	**SSE** Time	**OracleML** Time
0.9	0.85 sec	2.76 sec	0.22 sec
0.95	1.00 sec	2.46 sec	0.31 sec
0.99	2.51 sec	2.27 sec	0.42 sec
0.995	5.28 sec	2.42 sec	0.79 sec
0.999	21.4 sec	3.54 sec	2.32 sec
0.9995	37.0 sec	4.89 sec	3.62 sec
0.9999	212 sec	18.09 sec	13.21 sec

Table 7: The comparison of the training time by QuadraticSearch, BlinkML, and OracleML. This is the raw data for Figure 9.

LR, Yelp		ME, MNIST	
Operation	**Runtime**	**Operation**	**Runtime**
Initial Model Training	1.74 sec	Initial Model Training	53.0 sec
Statistics Computation	0.22 sec	Statistics Computation	1.09 sec
Sample Size Searching	1.35 sec	Sample Size Searching	17.9 sec
Final Model Training	26.93 sec	Final Model Training	408.2 sec
Total	30.25 sec	Total	480.2 sec

(a) AccContract.

LR, Yelp		ME, MNIST	
Operation	**Runtime**	**Operation**	**Runtime**
Initial Model Training	1.74 sec	Initial Model Training	53.0 sec
Statistics Computation	0.22 sec	Statistics Computation	1.09 sec
Accuracy Estimation	0.28 sec	Sample Size Searching	0.55 sec
Total	2.25 sec	Total	54.6 sec

(b) ScaleContract.

Table 8: Runtime analysis. This is the raw data for Figure 10.