MLbase: Distributed Machine Learning Made Easy

Xinghao Pan, Evan R. Sparks, Andre Wibisono

ABSTRACT

MLbase is a system designed to address many of the problems that arise when applying Machine Learning techniques at scale. MLbase provides end users with a simple, declarative query language to build and apply models, while at the same time providing developers of machine learning algorithms the tools they need to implement and deploy their algorithms over a cluster of machines with minimal complexity. Our initial prototype includes implementations of several such algorithms, some of which have not been presented in a distributed setting. MLbase also provides a novel query optimizer designed to to take best-practices machine learning workflows, execute them efficiently across a cluster of machines, and automate model selection.

1. INTRODUCTION

The application of state of the art machine learning techniques is often difficult for end users. Many algorithms have parameters that need to be carefully tuned in order to achieve good performance, yet there is often no systematic rule for choosing the correct parameter values. As a result, tasks such as parameter tuning and model selection are often left to the experts in the field. At the same time, experts in the field are often most comfortable developing cutting edge algorithms in platforms like MATLAB or R, which are well known not to scale easily to problems that need to be solved across multiple machines [26, 1]. With MLbase, a prototype of the system recently proposed in CIDR [22], we present a solution to these challenges using three important ideas: clean interfaces, high level machine learning primitives, and a query language/optimizer designed to simplify life for the end user. By forcing a clean interface between ML developers and the system in the form of machine learning primitives, as well as between the query language and the system in the form of contracts, we enable query optimization and automated model selection. By extending traditional cluster computing frameworks with additional primitives tailored to machine learning developers, we simplify the lives of those developers and make it easy to develop and deploy new algorithms at scale. In doing so, we offer implementations of several distributed machine learning algorithms, several of which have not been presented in a distributed setting. By offering a consistently defined interface to these underlying algorithms, we enable support for a general query language to build new models, and an optimizer capable of performing intelligent model selection. Our system consists of a set of primitives, implementations of several machine learning algorithms, a contract subsystem, and a query optimizer designed to select the best model for a user's data, all built in Scala with Spark [31] as the system's runtime.

Listing 1: An example MLbase program.

```
data = load('/tmp/feature_1.txt')
target = project(data, 1)
dset = project(data, [2,3,4,5,6])
model_object = learn(target, dset)
print(model_object)

test_dset = load('/tmp/feature_2.txt')
results = predict(model_object, test_dset)
print(results)
```

2. LANGUAGE

The highest level of interaction with the system is the query language presented by MLbase. In its current form, it supports just a few basic operations – the loading of datasets, the projection of features from that dataset, the training of a model on the dataset, and the ability to make new predictions using that model on new input data. This simplicity means that even a casual end user should be capable of building and running models with our system. An example MLbase program is shown in Listing 1.

This program loads data from file, selects a target feature for classification or regression, as well as training features, trains a model on this data, displays a model summary, and then loads and and generates predictions on a new dataset. Our system compiles and expands this program into a a logical learning plan that is represented as a directed acyclic graph.

In particular, we expand the `learn` step into a machine learning workflow that searches across model families and their parameters for models which optimally fit the user's data.

Our work includes a parser and compiler for this language, which includes primitives for assignment, data loading, projection, sampling, prediction, and printing. These commands can be chained into arbitrarily large programs, and multiple models can be learned from multiple datasets in one script. Further, predictions from multiple models can be presented to the user for deeper analysis. The specifics of how these queries are processed and how models are found optimally is presented in our discussion of the optimizer in Section 6.

3. PRIMITIVES

In this section we describe two types of primitives that we expose to the machine learning developer to write his distributed algorithms in. The first, MapReduce, is a well-known distributed programming paradigm which has been used to great success in parallelizing many machine learning algorithms. However, MapReduce suffers from some shortcomings that make it impossible or unnatural to use as a primitive for certain classes of machine learning algorithms. This motivates our development of the second primitive, Batch Map, to overcome these issues.

3.1 MapReduce

MapReduce is a sequence of two operations, `map` and `reduce`, that apply a transformation to data and aggregate the results. Given a dataset of key-value pairs $S = \{(k_1, v_1), \ldots, (k_n, v_n)\}$, the map operation applies a function f to each data point (k_i, v_i) to produce a sequence of key-value pairs $f(k_i, v_i) = \{(k'_1, v'_1), \ldots, (k'_m, v'_m)\}$. The number $m \equiv m_i$ of key-value pairs in the output can differ for each input (k_i, v_i), and different input pairs can produce output values with the same key. The `reduce` operation takes in the output of the `map` operation, which is the union of all the output sets $\bigcup_{i=1}^{n} f(k_i, v_i)$, and collects the values v' indexed by the same key k'. The `reduce` operation then applies an aggregator function g to each set $\{v' : k' = k\}$ of output values corresponding to the same output key k. This allows MapReduce to return a collection of values, one for each output key.

In [8], it was shown that several machine learning algorithms can be cast in this framework. Both the `map` and `reduce` primitives are readily available in Spark, and exposed for use by developers in MLbase.

3.2 Batch Map

In using the MapReduce programming framework, the developer is constrained to computing with functions f that are *decomposable* over the input data points. This effectively prevents the implementation of algorithms that exploit the joint structure of multiple data points.

Furthermore, as the `map` primitive is defined to operate over a single dataset, it is impossible to have interactions between multiple distributed data structures (except through a highly expensive join operation). As we will argue in Section 5, the ability to have such interactions is advantageous for both computational efficiency and fault tolerance in implementing distributed machine learning algorithms.

To address these issues, we propose a `BatchMap` primitive that allows operations on the (locally available) partitions of all distributed data structures. Recall that since our system is implemented in Spark, we represent our data as *resilient distributed datasets* (RDDs). Suppose we have a collection of RDDs $X_1, X_2, \ldots, X_s$, not necessarily of the same type or size, partitioned across K machines. Let $X_i^{(k)}$ be the partition of the i-th RDD stored on the k-th machine.

Let f be a function that takes in a collection of subsets of data and performs some computation on the data to produce a collection of results. Then the operation

$$\texttt{BatchMap}(f, X_1, \ldots, X_s)$$

returns an RDD X_{s+1} whose partitions are

$$X_{s+1}^{(k)} = f(X_1^{(k)}, \ldots, X_s^{(k)}).$$

We also preserve the partitioning of RDDs, by storing the partition $X_{s+1}^{(k)}$ formed by $f(X_1^{(k)}, \ldots, X_s^{(k)})$ on the k-th machine.

In effect, the `BatchMap` primitive provides the developer to manipulate data object on any slave machine, as if he were performing computations on an individual machine, but with his operations homogeneously replicated across all slave machines. Since developers typically develop and prototype single-machine, sequential algorithms before attempting to deploy in a distributed setting, we believe the `BatchMap` primitive provides a natural and powerful abstraction for developers to think about parallelization. By imposing fewer constraints, we also lower the effort required of the developer to convert his sequential algorithm into a distributed one.

While `BatchMap` is a generic primitive, in practice when implementing our algorithms, we often treat the RDDs as blocks of dense or sparse vectors. This usage of `BatchMap` lends itself naturally to matrix-oriented algorithms that are common in machine learning.

In order to efficiently support our algorithms under `BatchMap`, we had to modify the `DAGScheduler` component of Spark and add a caching layer to handle exponential growth in the depth first search traversal of the lineage structure of the RDDs we constructed.

4. CONTRACTS

While our Machine Learning primitives make up one half of the equation connecting the machine learning algorithms to the system, the Contracts make up the other half. Motivating the Contracts is the notion that new Machine Learning algorithms should be callable in a standard and well-defined way. Thus, the burden is on the ML developer to specify this interface. Contracts are represented as XML files in a "Library" that the system has access to. Included in each contract is the name and model family of each algorithm, class names of their associated implementation, and a list of parameters associated with the algorithm. These parameters are typed, and may contain hints about how they are to be tuned for a particular algorithm. For example, in searching for an optimal solution to Logistic Regression, it may make sense for callers to vary the lambda parameter multiplicatively. Each contract conforms to a predefined *contract schema*, which allows for automatic translation of the contracts to Scala objects, which we use to enumerate possible model configurations and search over their parameter space. Listing 2 contains an example contract.

While the introduction of Contracts to the system increases overhead for the ML developer, the potential that is presented by them is enormous. By providing a consistent and well defined interface to every algorithm in the system, we enable programmatic access to every algorithm registered with the system. This directly enables our ability to perform optimization across model families as in Section 6. Future enhancements to contracts will enable the ML developers to provide hints about runtime complexity and memory requirements to the system, as well as hints about what feature transformations may yield better models.

5. ALGORITHMS

We use the primitives described in Section 3 to implement several distributed machine learning algorithms on top of

Listing 2: Example contract for Logistic Regression.

```
<?xml version="1.0"?>
<contract>
  <modelFamily>
    <className>
        org.dmx.ml.LogisticRegression
    </className>
    <parameters>
      <enumerationParameter>
        <name>optimizationType</name>
        <values>
            org.dmx.ml.optimization.GradientDescent
        </values>
      </enumerationParameter>
      <rangeParameter>
        <name>optimizationType.learningRate</name>
        <range>
          <min>0.01</min>
          <max>100</max>
        </range>
        <stepSize>10</stepSize>
        <stepType>multiplicative</stepType>
      </rangeParameter>
    </parameters>
  </modelFamily>
</contract>
```

MLbase. In this section we briefly describe the algorithms that we have implemented and how we implement the distributed version of the algorithms using our set of primitives. More details on the algorithms and how we convert them into distributed algorithms can be found in Appendix A.

5.1 LogitBoost

In the supervised learning setting, boosting is a general framework for combining a set of "weak" learners into a "strong" learning algorithm with better guarantees [27]. When the weak learners are easy to train, boosting is an efficient method to construct a good learning algorithm. For supervised classification problems, where the goal is to predict the sign $y \in \{-1, +1\}$ of an input data $x = (x(1), \dots, x(d)) \in \mathbb{R}^d$, the weak classifiers are usually taken to be decision stumps, i.e. $f(x) = \text{sign}(x(j) - t)$ for some dimension $1 \leq j \leq d$ and threshold $t \in \mathbb{R}$. The boosting algorithm forms a weighted sum of the weak classifiers, $F(x) = \sum_{m=1}^{M} c_m f_m(x)$ for some weights $c_m \in \mathbb{R}$, and the resulting classification function is $\text{sign}(F(x))$.

LogitBoost [15] is a boosting algorithm that attempts to find the best decision stumps $f_1, \dots, f_M$ and the corresponding weights $c_1, \dots, c_M$ by minimizing the logistic loss function over a given set of training points. The sequential version of LogitBoost is a greedy algorithm that finds the next best decision stump f and weight c to be added to the current function F. Each iteration can be performed in $O(nd)$ time, where n is the number of training points and d is the dimension of the feature vectors, if the values along each feature dimension are pre-sorted; see Appendix A.1 for more details.

However, in a distributed environment, each slave machine only has access to a subset of the data, so finding the optimal weak classifier efficiently is non-trivial. This problem is aggravated if we are restricted to using the `map` primitive, as each `map` operation only has access to a single data point. Fortunately, we do not require that the weak classifier learned at each iteration be the optimal solution, but only that it decreases the logistic loss. We now propose two methods that use our `BatchMap` primitive for finding new weak classifiers in a distributed setting.

Locally best weak classifier.

The first method is to have each slave machine propose a weak classifier that minimizes the logistic loss evaluated over its local data points. The local logistic loss is also used as a proxy for the global logistic loss. Thus, the master simply picks the "best" locally trained classifier (which results in the largest decrease in the local logistic loss) to be added to the LogitBoost model.

This scheme is fast because each machine only needs to perform local computation with no synchronization overhead. However, there are a number of potential problems with this method. For instance, there is no guarantee that the globally optimal feature dimension and threshold will be selected by any of the local optimizations. In particular, the chosen classifier weights are locally, but not necessarily globally, optimal, and as such may not be guaranteed to lower the global logistic loss. Moreover, by approximating the global objective by local objective functions, the master may not be taking the globally optimal decision.

Reweighted (random) classifier.

To overcome the issues with the previous scheme, we propose a second method where a slave machine may at random propose a classifier with random feature dimension and threshold. We assume that the local logistic loss is still a sufficiently good approximation for global logistic loss, in the sense that among all proposed classifiers, the classifier that minimizes the global logistic loss is in the top κ classifiers with lowest local logistic loss, for some parameter $1 \leq \kappa \leq K$ that we can tune. We distribute these top κ classifiers to all slaves to compute their global logistic loss as well as the globally optimal classifier weight. The master then selects the globally best classifier and adds it to the LogitBoost model with the recomputed weight.

Unlike the first scheme proposed above, this method guarantees that the global objective function is decreasing at each iteration. This comes at a higher communication cost because recalculating the weights requires computation over all the points across all the machines. However, in practice we do not see big performance difference between the locally-best scheme and this method.

5.2 DP Means

The Dirichlet process (DP) [29] is a popular non-parametric Bayesian mixture model that allows for flexible clustering of data without having to determine a priori the number of clusters. The inference process for DPs typically involves some form of Gibbs sampling (via the Chinese restaurant process, or CRP) or a variational approximation.

Recent work in [23, 21] explored the behavior of the CRP Gibbs sampler for exponential families DP mixture models, in the limit when $p(\cdot \mid \mu_k)$ goes to 0. Under small variance assumptions, the CRP Gibbs sampler reduces to the k-means algorithm, but with a twist: any datapoint which is not "close" to any cluster mean is placed in its own cluster. This algorithm, named DP means, achieves fast yet flexible hard clustering.

However, both the CRP Gibbs sampler and DP means suffer from the need to sequentially process data points. Each datapoint may potentially create a new cluster, which must then be taken into account when processing the subsequent datapoint. In this project, we extend the DP means algorithm to give a distributed implementation using our `BatchMap` primitive.

5.2.1 *Distributed DP means*

To the best of our knowledge, the algorithm we propose in this paper is the first to parallelize/distribute DP means over a cluster.

Instead of sequentially sampling datapoints, our distributed algorithm introduces a distributed, conservative cluster creation process. Given the current global set of means μ_c, each slave machine simulates the cluster assignment step by iterating through its data points, and locally creating a new cluster centered at x_i whenever it encounters an uncovered datapoint x_i.

The set of new local cluster means are then sent to the master, which simulates the cluster assignment on the proposed cluster means, promoting a local cluster mean to a global mean whenever an uncovered local cluster mean is found.

As new global clusters are created, some of the local cluster means x_i's may now be covered by the new global clusters. There may, however, be some data points on the slave machines that had been assigned to the simulated local cluster x_i, but remain up to 2λ after from any global cluster mean. Thus, we send the new set of global centers to all slave machines, and repeat the cluster creation process after discarding all simulated local clusters, until no datapoint is left uncovered.

By the end of the cluster creation process, every datapoint is guaranteed to be covered by some cluster. Next, we run the cluster assignment step of DP means once. Since all data points are now covered, no new clusters will be generated, and the assignment can take place in parallel.

Finally, we update the means by collecting the sufficient statistics $(\sum_i \mathbb{I}(z_i = c),\ \sum_i \mathbb{I}(z_i = c)x_i)$ for each cluster c, and computing the new mean $\mu_c \leftarrow \frac{\sum_i \mathbb{I}(z_i=c)x_i}{\sum_i \mathbb{I}(z_i=c)}$.

We execute the distributed cluster creation operation using the `BatchMap` primitive. While we could have used the standard `map` primitive, by using `BatchMap` we can reduce the potential number of new clusters proposed, and minimize the load on the master in synchronizing the clusters' creation, thus achieving a more scalable solution.

5.3 SVM

Support vector machine (SVM) is a classical supervised classification algorithm that attempts to find the maximum margin hyperplane separating the two classes of the data [30]. For input data $x \in \mathbb{R}^d$, SVM finds a separating hyperplane parameterized by a weight vector $w \in \mathbb{R}^d$ and an offset $b \in \mathbb{R}$ such that the classification function $f(x) = \text{sign}(w^\top x + b)$ predicts the label $y \in \{-1, +1\}$ while maximizing the distance from the data points to the hyperplane. Sequential single-node SVM is typically solved via a stochastic iterative algorithm that requires multiple passes over the training data. When our dataset is large and distributed across machines, this method incurs a substantial communication overhead.

We implement distributed SVM as a consensus algorithm using our `BatchMap` primitive. The algorithm is based on a technique from convex optimization called Alternating Direction Method of Multiplier (ADMM) [6], and it proceeds as follows. Suppose our training data is stored as an RDD partitioned across K machines. In each iteration, we perform the following steps:

1. Each machine k computes an estimate v_k of the best classifier based on its local view of the data.

2. The master node combines the estimates $v_1, \ldots, v_K$ and compute the best global classifier w.

3. Each machine k computes a penalty z_k of how much its estimate v_k deviates from the global estimate w. We go back to step 1 in the next iteration.

Thus, the distributed SVM algorithm needs to keep track two RDDs, v and z, which are recomputed and reassigned at each iteration. The computation of v (local classifiers) at each iteration depends on the training data RDD and the RDDs v and z from the previous iteration. This step can be accomplished by our `BatchMap` primitive. The computation of w (global classifier) depends on the RDDs v and z, and this step can be performed using the standard `reduce` operation. Finally, the computation of z (synchronization penalty) at each iteration depends on the new RDD v and the previous RDD z; this step can also be performed using the `BatchMap` primitive.

6. OPTIMIZER

When a user submits their query to the system, all variable references are resolved and the query is compiled into a directed acyclic graph of logical operations. These operations are then expanded to form a logical machine learning workflow. Our goal here is to select the best model for a user's data, and so we perform a search over possible model families and their parameterizations and use a cross-validation procedure to select the best-fitting model.

Optimization is broken down into two stages. First, we optimize a physical learning plan based on the logical learning plan derived from a user's query. Second, we perform our search over model and parameter space to best fit a user's workflow.

6.1 Logical Learning Plan

In its current form, our optimizer provides support for learning an optimal model for classification, but can be extended to support other types of supervised, and eventually unsupervised learning tasks. When a user submits a query to learn a new model, the system expands that query into a logical learning plan. That is, the "learn" portion of a user's query is expanded into a logical learning plan that consists of the following:

1. Subsampling – input data is subsampled.

2. Contract Loading and Expansion – Contracts are loaded from the Library and selected based on whether they are appropriate for the learning problem at hand.

3. Search – parameter space is enumerated and searched. Each enumeration is called a "model configuration"

4. Cross-validation – potential model configurations are cross-validated and their performance is evaluated in this cross-validated scenario.

5. Model selection and Final Model Construction – the best candidate model configuration based on average cross-validated out-of-sample performance is selected, and the model is estimated on *all* of the user's input data.

These steps represent a machine-learning best practices workflow. Over time, this workflow will be expanded to handle tasks like feature engineering and data normalization where appropriate.

6.2 Physical Learning Plan

Once a logical learning plan is constructed, the system automatically optimizes a query to make efficient use of the resources presented to it by the runtime, in our case, the *Spark* system. The physical optimizations made by this component of the system fall into three broad categories: caching, equivalent operation reordering, and lineage awareness.

Because our runtime is the Spark system, we represent our datasets as *resilient distributed datasets* (RDDs). These datasets are loaded into distributed memory and operations on them are evaluated lazily. In our system, RDDs that are frequently used are cached to speed up future operations on the same data. For example, we cache the sampled datasets that are used during cross-validation.

Three important operations in our system are sampling, projection, and merge. Projection allows a user or the system to select a subset of columns in an RDD to operate on (for example, when selecting features to train their model on). Sampling allows the system to select a subset of rows in an RDD, with or without replacement. Merging in this sense (not to be confused with a database merge-join) is defined as column-wise concatenation of two datasets that have the same number of rows. This can be useful for combining features from two files or two different sets of features. The astute reader will note that projection, selection, and merging are order independent in many cases. For example, selecting the first ten rows of a dataset, then projecting the second and third columns is exactly equivalent to projecting the second and third columns then selecting the first 10 rows. However, in our system, the speed of projection is linearly dependent on the number of rows in the dataset, while selection by row index is quick. As such, our system automatically reorders these order independent operations where appropriate. However, in workflows where eventually the entire dataset will be accessed under the same projection, we select an optimal physical plan based on minimizing repeated access to the same RDD tuples.

Our query language allows users to load and select data from multiple sources (different files on Amazon S3, HDFS, or a local filesystem, for example). However, because users often only want to learn models on data from a few sources, we introduce a set of optimization rules designed to avoid reading data from the same file more than once, and to avoid applying multiple transformations to a dataset where one would suffice. Simply speaking, we track which expressions are derivative of the same rows in a master dataset and combine these expressions (via a merge operation) as early as possible in the execution graph.

For example, in the case of the query presented in listing 1, since both the target feature and the training features come from the same file, the system merges these as early as possible, and the result is a dataset that combines both training and target features. Indeed, the two projection operations

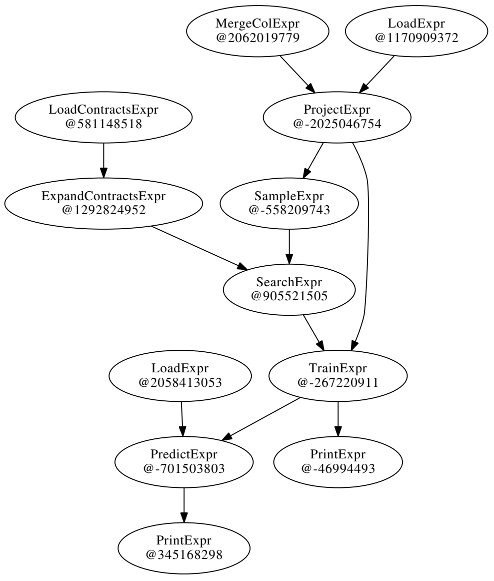

Figure 1: An optimized physical learning plan.

that are specified by the user can be turned be turned into a combination of column indexes followed by a single projection. A physically optimized version of the query presented in Section 1 is shown in Figure 1.

6.3 Model Search

Model family and parameter selection are difficult problems, particularly for novice users. Our optimizer has the objective to find the best possible model for the input data. We define best-possible model as the model that has the best possible out-of-sample performance. As has been often discussed [5], this means choosing a model family that adequately captures features of the data while avoiding overfitting. We can optimize over the possible space of model parameters thanks to the hints presented by the ML developer in the contracts. For example, in the contract shown in Section 4, we can see that logistic regression has a step function parameter, and that that parameter should be searched multiplicatively with a step size of 10. That is, the values 0.01, 0.1, 1.0, and so on, will be attempted when searching over this space. The contracts also support enumerated parameters and additive range parameters, and will soon support functional parameters. In its current implementation, the optimizer will search over all possible configuration options allowed for by the contracts and select the best performing model built on its subset of the data. It should be noted that out-of-sample classification accuracy is only the relevant metric to use for classification models, and as such that is the only class of model supported by the optimizer at this time. However, we do intend to support regression models as well as unsupervised learning models in the future using similar techniques.

The model search algorithm operates on subsamples of the dataset and selects the model with the best cross-validated performance as determined by Algorithm 1.

In Figure 6.3, we show estimated out-of-sample performance using this method in the LogitBoost and SVM families on an initial sample size of 100,000 points using the same synthetic data that we use for the classifiers in Sec-

Algorithm 1: Cross-validated optimization algorithm.

```
Data   : Datapoints $X$, contract, configuration,
         sampleSize 0.05, iterations 10
Result: Estimated out of sample performance
1 Initialized testData ← sample($X$, sampleSize);
2 Initialize oosPrediction ← [];
3 for $i \in$ iterations do
4     samp ← sample(X, sampleSize);
5     model ← fitModel(samp, contract, configuration);
6     modelAccuracy ← calcAccuracy(model, testData);
7     oosPrediction[i] ← modelAccuracy
8 end
9 return mean(oosPrediction)
```

tion 7. We can see that the best models in this case are those from the SVM learners with a lambda ranging from 1×10^{-6} to 1×10^{-2}. The optimizer chose lambda $= 0.001$, which corresponded to an out-of-sample accuracy of 86.61%. This was the second-best model in the out-of-sample experiment – the best model was for lambda $= 1 \times 10^{-5}$ which had an out-of-sample accuracy of 86.69%. While not perfect, the optimizer managed to pick a reasonable model based only on the cross-validation procedure we employed on a subset of the data.

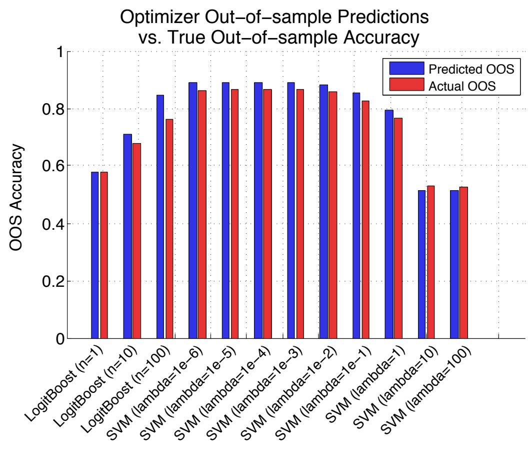

Figure 2: Sample model selection optimization.

Each model is submitted to the system for fitting using `akka` Actors – a paradigm for asynchronous concurrent programming that is popular in the Scala community. The asynchronous nature of the Actor paradigm enables us to submit multiple jobs to the Spark cluster asynchronously, and wait for results to come back. This allows us to support timeouts for model configurations that may never converge, as well as future syntax enhancements to the MLbase query language. For example, an end user might want to have the "best model after 5 minutes", or "best model after trying 1,000 possible models." By architecting the system with Actors handling model-building, we are able to support these types of queries with trivial language extensions.

7. EVALUATION

We evaluate our system across two important dimensions: complexity and scalability.

7.1 Experimental Setup

Our experimental setup consists of a Spark cluster on Amazon EC2 with instance type `m1.large`. Data is copied to HDFS before each experiment is executed. We scale the number of cores used in our experiments with the Spark `spark.cores.max` setting.

PSVM runs on this cluster using MPICH2. In this setup, data copied to nodes via RSYNC before our experiments were executed.

MATLAB experiments for LogitBoost and DPMeans were conducted on a 2012 model Macbook Pro Retina with an Intel core i7 processor and 8GB of RAM. The MATLAB experiments for SVM were conducted on a 2010 model Macbook Pro with an Intel core i5 processor and 4GB of RAM. The difference in machine environment is due to the need to use CVX in MATLAB for the SVM experiments.

Data for DP means clustering was generated by sampling data points x_i from 27 three-dimensional normal distributions $\mathcal{N}(\mu_k, I_{3\times3})$, with $\mu_k \in \{0, 3, 9\}^3$. The distributions are truncated to a finite support of radius 3 from the mean (any data point x_i with $\|x_i - \mu_k\|_2 > 3$ is re-sampled) to model the finite support of DP means clusters.

Data for the LogitBoost and SVM experiments was generated from a 100-dimensional normal distribution with zero mean and a diagonal covariance matrix. The diagonal entries of the covariance matrix were sampled independently from the χ^2_1 distribution. The label y_i for each feature vector x_i was generated by first sampling a random hyperplane $w \in \mathbb{R}^{100}$ and $b \in \mathbb{R}$ (common to all the data points) from the standard normal distribution, and then taking $y_i = \text{sign}(w^\top x_i + b)$. The labels were further corrupted by independent sign flips with probability 0.1. This results in an ideal classification accuracy of approximately 90%.

7.2 Complexity

While it is clear that the end-user query system is easy to use, we have not yet shown how easy it is to write a complete algorithm for the ML developers. In Table 1 we provide a summary of the number of lines of Scala code required to implement several popular Machine Learning algorithms and compare both to MATLAB and to existing parallel implementations of these algorithms, if they are publicly available.

We can clearly see that the Scala versions of these implementations, built using our ML primitives, are about as concise as the MATLAB representations, and if an open-source distributed implementation exists the Scala version is always significantly smaller than that version. The main exception to this is the case of SVM, where the MATLAB implementation is significantly smaller. This is due to the use of CVX, a MATLAB plugin for solving a general convex optimization problem [10, 19]. However, CVX currently only works well for small problems, both in terms of computation time and memory requirements. This is demonstrated by the much weaker scaling property of CVX compared to Spark and MLbase; see Figure 8. We also note that the MLbase implementation of SVM is still an order of magnitude more simple than PSVM, which contains nearly 5000 lines of custom C++ code that operates via MPI, and only supports SVM fitting.

7.3 Scalability

Algorithm	MATLAB	MLbase	Mahout	Other	Ratio
Logistic Regression	82	99	577	X	1.21
SVM	23	256	X	4697	11.13
DPMeans	86	288	X	X	3.35
LogitBoost	116	380	X	X	3.28

Table 1: Lines of code comparison of algorithm implementations. Ratio indicates ratio of lines of code of MLbase to MATLAB.

An important question is whether these algorithms actually scale. Can we run jobs across multiple machines with near-linear speedup? Can we handle more data because of our ability to spread work out over a cluster? In this section, we evaluate if our algorithms are able to achieve strong and weak scalability.

7.3.1 *Strong scaling*

An algorithm is said to be *strongly scalable* if, for a fixed problem size, its run time decreases as more machines are used. Ideally, the run time should be at least inversely proportional to the number of machines. Strong scalability is important as it implies that we can obtain solutions faster as long as we can bring more machines to bear on the problem.

LogitBoost.

We tested our two versions of distributed LogitBoost using a fixed dataset size of 200,000 data points on Spark clusters of 1, 2, 5, 10, 15 and 20 machines. For comparison, we also ran the sequential LogitBoost on MATLAB using the same 200,000 data points on a single machine. Figure 3 shows the speedup comparison of our two LogitBoost implementations on Spark, compared with the MATLAB implementation, obtained by dividing the running time for one machine by the running time for variable numbers of machines.

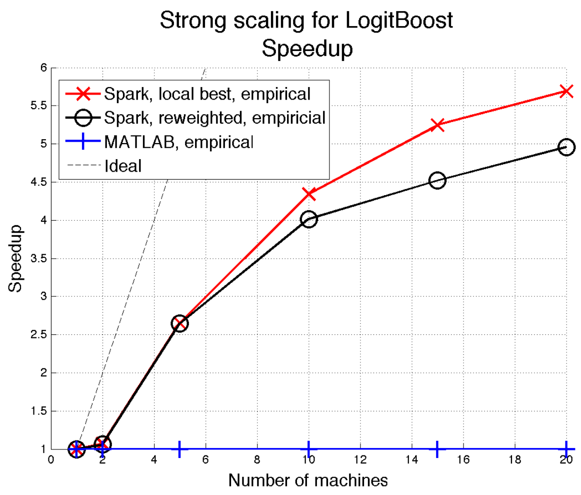

Figure 3: Speedup of distributed LogitBoost in the strong scaling setting.

With 20 machines, both Spark implementations are either faster or almost as fast as the MATLAB implementation.

DP Means.

DP means was tested with a dataset consisting of 270,000 data points. In addition to our distributed implementation on Spark, we also implemented both the original sequential DP means and our batched cluster creation algorithm in MATLAB.

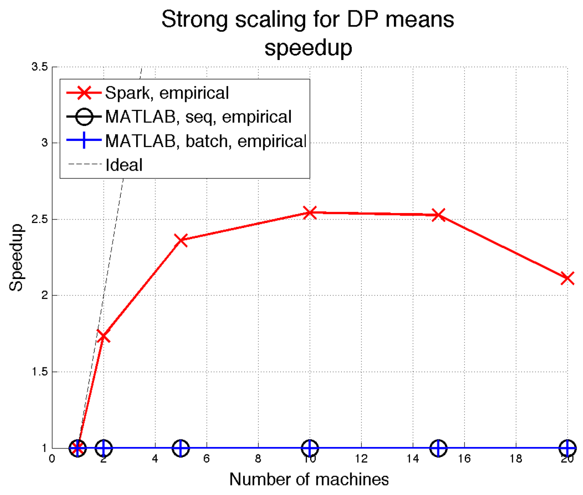

Figure 4: Speedup of distributed DP means in the strong scaling setting.

The run time for sequential DP means in MATLAB is approximately 1500 seconds, and for readability reasons, have been left out of the plot. On the other hand, the batched cluster creation program in MATLAB performs much better, completely in just over 100s. Neither MATLAB version, however, scales as we increase the number of machines.

Our Spark implementation of distributed DP means does decrease its run time as more machines are added. We do note that the reduction in run time plateaus and possibly increases; however, we do not yet have a good explanation to why this is so, and further investigation is required.

Figure 4 shows the speedup comparison of our DP means implementation on Spark, obtained by dividing the running time for one machine by the running time for variable numbers of machines.

SVM.

We tested our distributed implementation of SVM on a fixed dataset of $200,000$ points on Spark clusters of 1, 2, 5, 10, 15 and 20 machines. We also ran two other algorithms for comparison. The first one is the single-node MATLAB SVM solver using CVX. The second one is PSVM, a parallel implementation of SVM that achieves good performance by performing a low-rank approximation of the input data [7]. For these experiments we used the parameter `rank_ratio` of $1/\sqrt{n}$, where $n = 200,000$ is the number of data points. This is the value that is suggested in the PSVM paper, and it corresponds to an approximation of the input data by a rank-$\sqrt{n}$ matrix.

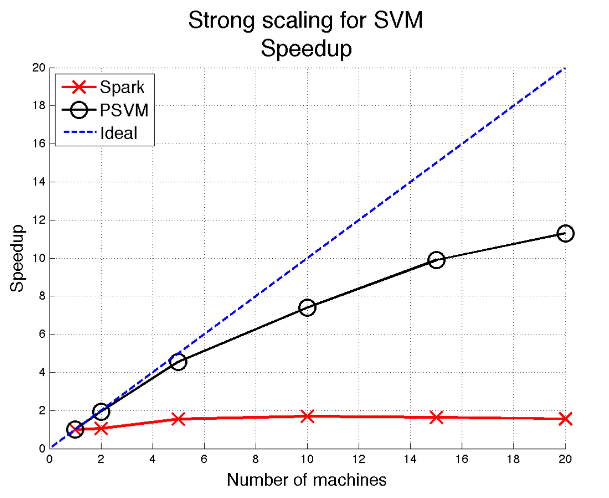

Figure 5: Speedup of distributed SVM in the strong scaling setting.

PSVM achieves excellent performance and scaling property, but it is unclear to what extent this can be attributed to the low-rank approximation that it performs. Our total runtime is about twice as high as total runtime for PSVM, and does not scale as well. The running time for our Spark implementation of SVM initially decreases as we add more machines, but as in the case of DP means, it eventually flattens out as the number of machines increases. This might be due to the communication overhead between the machines, but it requires further investigation. Figure 5 shows the speedup comparison of both our Spark implementation and PSVM, obtained by dividing the running time for one machine by the running time for variable numbers of machines.

7.3.2 Weak Scaling

An alternative notion of scalability is *weak scaling*, where the number of machines increases proportionally to the data set size, so that each machine has a fixed size of dataset. The implication of having a weakly scalable algorithm is that the computation time is bounded and largely independent of the data set size, so it is possible to work with arbitrarily large data sets by having sufficiently many machines.

LogitBoost.

We again evaluate both distributed versions of LogitBoost implemented in Spark, together with the sequential LogitBoost in MATLAB. Cluster sizes of 1, 2, 5, 10, 15, and 20 machines were used, with 50,000 data points on each machine. The run times for each implementation across different data set sizes is shown in Figure 6.

As the data set sizes are increased, the run times of the distributed LogitBoost implementations increase slowly. In comparison, the MATLAB sequential LogitBoost implementation's run time increases at least linearly with the data set size. For data sets with more than 500,000 data points, sequential LogitBoost is handily beaten in run time by the distributed implementations.

DP Means.

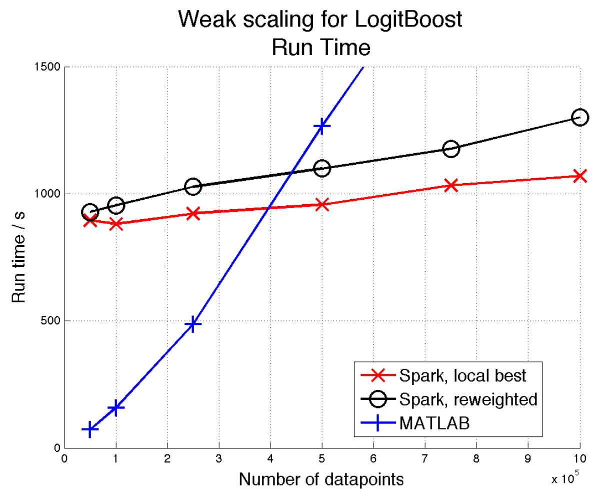

Figure 6: Weak scalability of LogitBoost. The running time of LogitBoost grows slowly as we add more machines to deal with increasing data size.

Spark cluster sizes of 1, 2, 5, 10, 15, and 20 machines were used, with 270,000 data points per machines, for evaluating the weak scalability of DP means. In Figure 7, it is clear that the original sequential DP means is highly inefficient when implemented in MATLAB.

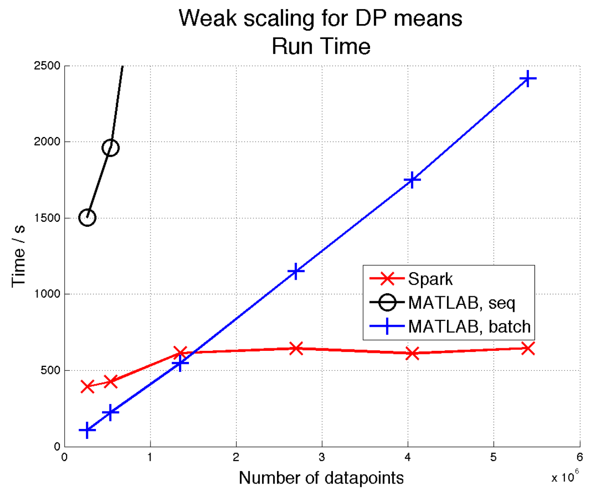

Figure 7: Weak scalability of distributed DP means. There is little degradation in DP means' run time if we keep the data size on each slave machine constant.

Even if we use the batched cluster creation algorithm in MATLAB, we still get a linear increase in run time. Impressively, the run time of our distributed Spark implementation is hardly affected by the number of data points, demonstrating weak scalability.

SVM.

We run the SVM experiments on Spark clusters with 1, 2, 5, 10, 15, and 20 machines with 200,000 data points per machine. We compare our distributed SVM implementation with the MATLAB CVX solver and the PSVM algorithm. As shown in Figure 8, our Spark implementation achieves excellent weak scaling property that beats PSVM. The computation time of CVX, on the other hand, scales superlinearly and it is not usable for larger problems.

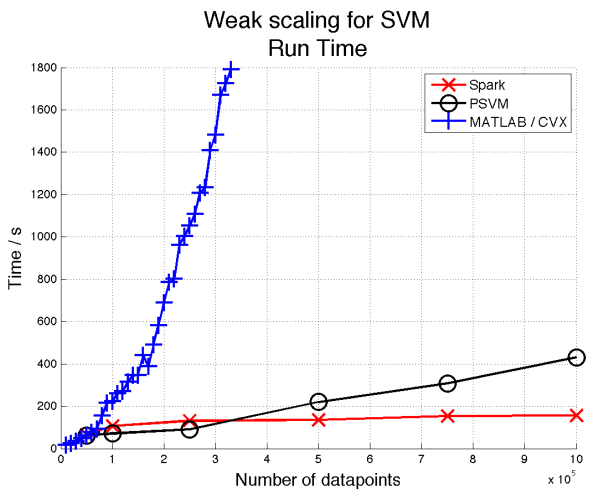

Figure 8: Weak scalability of distributed SVM. Our Spark implementation achieves better scaling property than PSVM. CVX is not scalable to larger problems.

8. RELATED WORK

In related work, the Weka [4], MADLib [20], and Mahout [3] projects attempt to make machine learning accessible. Weka is a collection of machine learning tools for data mining that simplifies their usage by providing a simple user interface, but requires expert knowledge of machine learning to use does not work in a cluster environment. Mahout aims is to build a set of scalable ML routines for Hadoop, and MADLib provides an ML library for relational database systems. Neither system addresses the difficult but necessary challenge of optimizing the learning algorithms. Google Predict [2] is Google's proprietary web-service for prediction problems, but restricts the maximum training data-size to 250MB. SystemML [18] from IBM Research provides a system of machine learning primitives based on linear algebra operations and attempts to optimize optimal plans with these primitives in a multi-stage MapReduce environment, but does nothing to ease the life for the end user and does not perform automated model selection.

[11] presents a parallel Monte Carlo sampler for Dirichlet processes (DPs). To the best of our knowledge, this is the first theorectically sound distributed sampling procedure for DPs. However, in the distributed form, the sampler requires movement of data across the network, which we consider to be an expensive operation. Supporting distributed sampling for DPs in the framework of [11] without excessive network traffic is a possible future research direction.

A distributed Gibbs sampling for DPs is implemented in Mahout [3]. However, a fixed maximum number of clusters is assumed. This is not truly a Dirichlet process, but more akin to a finite mixture model with a Dirichlet prior over mixture priors.

Distributed boosting algorithms have been proposed by various parties [13, 24]. In such algorithms, weak classifiers are first trained using the locally available data set on each slave machine, much like our `BatchMap` training of LogitBoost. Most recently, [25] proposed boosting using MapReduce, by intelligently combining weak classifiers trained independently on slave machines. We point out that the training was performed over data subsets (i.e. using a `BatchMap`-type primitive), rather than over individual data points (i.e. using a `map`-type primitive).

There have been plenty of efforts to parallelize SVM efficiently, both in primal and dual forms. Many of these algorithms, such as the P-packSVM [32], require special considerations on the distribution and partitioning of the data, while other approaches such as PSVM [7] only work with a subset or an approximation of the data. In this project we choose to implement a distributed version of SVM using ADMM [6] for the simplicity of the algorithm and the low complexity overhead, as well as the good performance that we observe in practice. In our experiments, we compare the performance of our distributed SVM implementation with PSVM, which performs a low-rank approximation of the input data to achieve good performance.

9. CONCLUSION

We have presented MLbase, a complete system to make development of machine learning models in a distributed computing environment easier for developers and end users. We have introduced new primitives in our distributed processing environment, which enables faster and more concise development of machine learning algorithms, and implemented three new algorithms – two of which do not seem to have publicly available distributed implementations. We have also described and implemented a model optimization strategy designed to handle model selection, and defined interfaces between the optimizer subsystem and the machine learning algorithms. Our results look promising from several perspectives – first, we have shown that our system allows for reasonably good scaling of several distributed algorithms, without sacrificing too much in terms of model accuracy. Second, we have shown that an optimization system can be implemented to select models across families and does a pretty good job picking an ideal model configuration based on our test data. We intend to continue to explore the potential presented by this system in our future work.

10. ACKNOWLEDGEMENTS

This project is part of a larger research effort underway in the AMPLab, and could not have been possible without the help of several people. Denny Britz, Ameet Talwalkar, Tim Kraska, John Duchi, and Joey Gonzalez all contributed ideas, design, algorithmic insight, or code.

11. REFERENCES

[1] BigMemory Project. `http://bigmemory.org/`.
[2] Google Prediction API. `https://developers.google.com/prediction/`.
[3] Mahout. `http://mahout.apache.org/`.
[4] Weka. `http://www.cs.waikato.ac.nz/ml/weka/`.
[5] H. Akaike. Factor analysis and aic. *Psychometrika*, 52(3):317–332, 1987.

[6] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1), 2011.
[7] E. Y. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui. Psvm: Parallelizing support vector machines on distributed computers. In *NIPS*, 2007. `http://code.google.com`.
[8] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.
[9] C. Cortes and V. N. Vapnik. Support-Vector Networks. *Machine Learning*, 20(3):273–297, 1995.
[10] I. CVX Research. CVX: Matlab software for disciplined convex programming, version 2.0 beta. `http://cvxr.com/cvx`, Sept. 2012.
[11] R. P. A. Dan Lovell and V. K. Mansinghka. Parallel markov chain monte carlo for dirichlet process mixtures. In *Big Learn workshop, NIPS'12*, 2012.
[12] J. C. Duchi and Y. Singer. Efficient online and batch learning using forward-backward splitting. *Journal of Machine Learning Research*, 10:2873–2898, 2009.
[13] W. Fan, S. Stolfo, and J. Zhang. The application of adaboost for distributed, scalable and on-line learning. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 362–366. ACM, 1999.
[14] Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119 – 139, 1997.
[15] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.
[16] D. Gabay. Applications of the method of multipliers to variational inequalities. In M. Fortin and R. Glowinski, editors, *Augmented Lagrangian Methods: Applications to the Solution of Boundary Value Problems*, chapter IX, pages 299–340. North-Holland, 1983.
[17] D. Gabay and B. Mercier. A dual algorithm for the solution of non linear variational problems via finite element approximation. *Computers and Mathematics with Applications*, 2:17–40, 1976.
[18] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 231–242, Washington, DC, USA, 2011. IEEE Computer Society.
[19] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008. `http://stanford.edu/~boyd/graph_dcp.html`.
[20] J. M. Hellerstein, C. Ré, F. Schoppmann, Z. D. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library or mad skills, the sql. Technical Report UCB/EECS-2012-38, EECS Department, University of California, Berkeley, Apr 2012.
[21] K. Jiang, B. Kulis, and M. Jordan. Small-variance asymptotics for exponential family dirichlet process mixture models. In *Advances in Neural Information Processing Systems 25*, pages 3167–3175, 2012.
[22] T. Kraska, A. Talwalkar, J.Duchi, R. Griffith, M. Franklin, and M. Jordan. Mlbase:a distributed machine learning system. In *In Conference on Innovative Data Systems Research*, 2013.
[23] B. Kulis and M. I. Jordan. Revisiting k-means: New algorithms via bayesian nonparametrics. *CoRR*, abs/1111.0352, 2011.
[24] A. Lazarevic and Z. Obradovic. The distributed boosting algorithm. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 311–316. ACM, 2001.
[25] I. Palit and C. Reddy. Scalable and parallel boosting with mapreduce. *Knowledge and Data Engineering, IEEE Transactions on*, 24(10):1904–1916, 2012.
[26] M. J. Quinn, A. G. Malishevsky, and N. Seelam. Otter: Bridging the gap between matlab and scalapack. In *HPDC*, pages 114–, 1998.
[27] R. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.
[28] B. Schölkopf, C. Burges, and A. Smola, editors. *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1998.
[29] Y. W. Teh. Dirichlet process. In *Encyclopedia of Machine Learning*, pages 280–287. 2010.
[30] V. N. Vapnik. *Statistical Learning Theory*. Wiley, 1998.
[31] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10. USENIX Association, 2010.
[32] Z. A. Zhu, W. Chen, G. Wang, C. Zhu, and Z. Chen. P-packsvm: Parallel primal gradient descent kernel svm. In W. Wang, H. Kargupta, S. Ranka, P. S. Yu, and X. Wu, editors, *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, pages 677–686. IEEE Computer Society, 2009.

APPENDIX

A. MORE DETAILS ON THE ALGORITHMS

In this section we provide more details on the machine learning algorithms that we described in Section 5.

A.1 LogitBoost

AdaBoost [14] is a popular Boosting algorithm that can be interpreted as minimizing the exponential loss function

$$\mathcal{L}_{\text{AdaBoost}}(F) = \frac{1}{n}\sum_{i=1}^{n} \exp(-y_i F(x_i))$$

over the training points $S = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. LogitBoost [15] is another variant of boosting that seeks to minimize the logistic loss

$$\mathcal{L}(F) = \frac{1}{n}\sum_{i=1}^{n} \log(1 + \exp(-y_i F(x_i)), \tag{1}$$

which is equivalent to maximizing the log-likelihood of the data under the binomial generative model

$$\mathbb{P}(y = 1 \mid x) = \frac{\exp(F(x))}{1 + \exp(F(x))}.$$

Since the logistic loss is a more conservative approximation to the $0-1$ loss compared to the exponential loss, in practice LogitBoost is usually observed to be more robust to noisy data and outliers compared to AdaBoost.

In the remainder of this section, we first describe the greedy algorithm for optimizing the loss function (1), and then provide more details on the two distributed LogitBoost schemes that we proposed in Section 5.1.

A.1.1 Sequential LogitBoost

Fix the number of weak classifiers M. Performing an exact minimization of the loss (1) over functions of the form $F(x) = \sum_{m=1}^{M} c_m f_m(x)$ is intractable. Therefore, we use a greedy algorithm where we sequentially find the next best weak classifier and weight. Specifically, for each $1 \le m \le M-1$, once we have a partial function $F_m(x) = c_1 f_1(x) + \cdots + c_m f_m(x)$, we seek to find the best f_{m+1} and c_{m+1} via

$$(f_{m+1}, c_{m+1}) = \operatorname*{argmin}_{f,c}\; \mathcal{L}(F_m + cf),$$

where the minimization is over all decision stumps f and weights $c \in \mathbb{R}$. In order to perform this computation efficiently, we further approximate the logistic loss function by a second-order Taylor expansion: for each $s \in \mathbb{R}$ and for small $t \in \mathbb{R}$, the value $\log(1 + e^{-s-t})$ is approximately

$$\log(1 + \exp(-s)) - \frac{\exp(-s)}{1 + \exp(-s)}t + \frac{1}{2}\frac{\exp(-s)}{(1 + \exp(-s))^2}t^2.$$

Then for small c, we can replace the function $\mathcal{L}(F_m + cf)$ by

$$\mathcal{L}(F_m) - \frac{c}{n}\sum_{i=1}^{n} y_i p_i f(x_i) + \frac{c^2}{2n}\sum_{i=1}^{n} p_i(1-p_i) f(x_i)^2,$$

where $p_i = 1/(1 + \exp(-y_i F_m(x_i)))$. Thus, recalling that $f(x)^2 = 1$ since f is a decision stump, we find (f_{m+1}, c_{m+1}) by minimizing the quantity

$$\tilde{\mathcal{L}}(f, c) = -c\sum_{i=1}^{n} y_i p_i f(x_i) + \frac{c^2}{2}\sum_{i=1}^{n} p_i(1-p_i).$$

Since $\tilde{\mathcal{L}}(f, c)$ is quadratic in c, we can solve it explicitly. For each fixed function f, the optimal weight c is given by

$$c_f^* = \frac{\sum_{i=1}^{n} y_i p_i f(x_i)}{\sum_{i=1}^{n} p_i(1-p_i)},$$

yielding the objective value of

$$\min_c \tilde{\mathcal{L}}(f, c) = -\frac{\left(\sum_{i=1}^{n} y_i p_i f(x_i)\right)^2}{2\sum_{i=1}^{n} p_i(1-p_i)}.$$

Each decision stump $f(x) = \operatorname{sign}(x(j) - t)$ is characterized by an index $1 \le j \le d$ and threshold $t \in \mathbb{R}$. Therefore, finding the optimal decision stump f_{m+1} now amounts to solving the following optimization problem:

$$\begin{aligned}\min_{f,c} \tilde{\mathcal{L}}(f,c) = &-\frac{1}{2\sum_{i=1}^{n} p_i(1-p_i)} \\ &\times \left\{ \max_{1 \le j \le d}\, \max_{t \in \mathbb{R}} \left(\sum_{i=1}^{n} y_i p_i \operatorname{sign}(x(j) - t) \right)^2 \right\}.\end{aligned} \tag{2}$$

Note that even though the threshold t can take any real values, for each fixed j there are only at most n distinct values of the objective function (2), one for each interval between any two consecutive projections of the x_i's on to the j-th dimension. Thus, we can solve the inner maximization problem above by trying all n possible values for the threshold t, and then picking the value of j that gives the best optimal value. This computation can be performed in $O(nd)$ time if the values along each feature are pre-sorted.

A.1.2 Distributed LogitBoost

We now describe how to perform the computation for the two proposed distributed LogitBoost algorithms.

Locally best weak classifier.

Recall that in this method, each slave machine proposes a weak classifier that minimizes the logistic loss evaluated over its local data points. Formally, suppose the training set S is partitioned into disjoint subsets $S_1, \dots, S_K$, where each partition S_k is stored in a separate machine. In a slight abuse of notation, we write $i \in S_k$ to mean the data point $(x_i, y_i) \in S_k$. We define the local objective on the k-th machine as

$$\tilde{\mathcal{L}}_k(f, c) = -c\sum_{i \in S_k} y_i p_i f(x_i) + \frac{c^2}{2}\sum_{i \in S_k} p_i(1-p_i).$$

Each machine finds the locally best classifier $(f_{m+1}^{(k)}, c_{m+1}^{(k)})$ that minimizes the local loss function $\tilde{\mathcal{L}}_k(f, c)$. The master then selects the classifier with the best average local objective: we set $(f_{m+1}, c_{m+1}) = (f_{m+1}^{(k^*)}, c_{m+1}^{(k^*)})$, where

$$k^* = \operatorname*{argmin}_{1 \le k \le K} \frac{1}{|S_k|}\tilde{\mathcal{L}}_k(f_{m+1}^{(k)}, c_{m+1}^{(k)}).$$

Reweighted (random) classifier.

In this scheme, we pick the top κ local classifiers with lowest local logistic loss, for some parameter $1 \le \kappa \le K$ that we can tune. We distribute these top κ classifiers to all slaves to compute their global logistic loss as well as the globally optimal classifier weight. The master then selects the globally best classifier and adds it to the LogitBoost model with the recomputed weight.

Formally, each slave machine can either propose its locally best classifier

$$(f_{m+1}^{(k)}, c_{m+1}^{(k)}) = \operatorname*{argmin}_{f,c} \tilde{\mathcal{L}}_k(f, c),$$

or choose a random decision stump f along with its locally optimal weight. Without loss of generality, we may assume $(f_{m+1}^{(1)}, c_{m+1}^{(1)}), \dots, (f_{m+1}^{(\kappa)}, c_{m+1}^{(\kappa)})$ are the top κ classifiers with lowest local logistic losses. For these κ classifiers, we keep their decision stumps but recompute the optimal weights as

$$\tilde{c}_{m+1}^{(k)} = \frac{\sum_{i=1}^{n} y_i p_i f_{m+1}^{(k)}(x_i)}{\sum_{i=1}^{n} p_i(1-p_i)}.$$

The master then chooses, among the κ reweighted classifiers, the one that minimizes the global objective: we set $(f_{m+1}, c_{m+1}) = (f_{m+1}^{(k^*)}, \tilde{c}_{m+1}^{(k^*)})$, where

$$k^* = \operatorname*{argmin}_{1 \le k \le \kappa} \tilde{\mathcal{L}}(f_{m+1}^{(k)}, c_{m+1}^{(k)}).$$

A.2 DP Means

The Dirichlet process (DP) [29] is a popular non-parametric Bayesian mixture model that allows for flexible clustering of data without having to determine a priori the number of clusters. The inference process for DPs typically involves some form of Gibbs sampling (via the Chinese restaurant process, or CRP) or a variational approximation.

Recent work in [23, 21] explored the behavior of the CRP Gibbs sampler for exponential families DP mixture models, in the limit when $p(\cdot \mid \mu_k)$ goes to 0. Under small variance assumptions, the CRP Gibbs sampler reduces to the k-means algorithm, but with a twist: any datapoint which is not "close" to any cluster mean is placed in its own cluster. This algorithm, named DP means, achieves fast yet flexible hard clustering.

However, both the CRP Gibbs sampler and DP means suffer from the need to sequentially process data points. Each datapoint may potentially create a new cluster, which must then be taken into account when processing the subsequent datapoint. In this project, we extend the DP means algorithm to give a distributed implementation using our `BatchMap` primitive.

We first describe Dirichlet processes, the CRP Gibbs sampler and DP means update operations, before describing our distributed DP means algorithm.

A.2.1 Dirichlet process

There are multiple formulations of the Dirichlet process. We present in this section the infinite mixture model view of DPs. In particular, we consider DPs to be an infinite mixture of exponential family distribution models parameterized by their means.

To define an infinite mixture model, we first draw an infinite number of means $\{\mu_k\}_{k=1}^{\infty}$ from a conjugate prior H,

$$\mu_k \mid H \sim H(\cdot).$$

Next, we define the model mixing weights $\{\pi_k\}_{k=1}^{\infty}$ drawn from a stick-breaking process as follows:

$$\beta_k \mid \alpha \sim \text{Beta}(1, \alpha), \qquad \pi_k \mid \{\beta_k\} = \beta_k \prod_{l=1}^{k-1} (1 - \beta_l)$$

where α is the concentration parameter of the DP.

Now to draw an observation x_i, we first pick a model from the infinite mixture model. Letting z_i be the indicator variable for the model of x_i, we draw

$$z_i \mid \{\pi_k\} \sim \text{Mult}(\pi), \qquad x_i \mid z_i, \{\mu_k\} \sim p(\cdot \mid \mu_{z_i}).$$

Equivalently, if we let $G_0 \sim DP(\alpha, H)$ be a draw from the Dirichlet process, then we can write $G_0 = \sum_{k=1}^{\infty} \pi_k \delta_{\mu_k}$, where δ_{μ_k} is a Dirac distribution at μ_k. Note that G_0 is an infinite discrete distribution over the mean parameters. The generative process for $\{x_i\}$ is equivalently,

$$\begin{aligned} &G_0 \sim DP(\alpha, H) \\ &\nu_i \sim G_0 \\ &x_i \sim p(\cdot \mid \nu_i). \end{aligned}$$

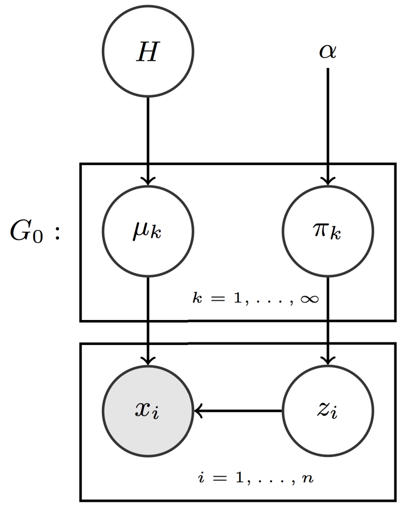

Figure 9: Graphical model for Dirichlet process.

The graphical model of DP as an infinite mixture model is shown in Figure 9.

A.2.2 CRP Gibbs sampler

The CRP sampler proceeds in two steps. First, the indicator variable z_i of each datapoint x_i is sampled, given the cluster assignment $z_{-i} = \{z_j\} \backslash z_i$ of all other variables and cluster means, according to the distribution:

$$\mathbb{P}(z_i = c \mid z_{-i}, x_i, \mu) \propto p(x_i \mid \mu_c) \sum_{j \neq i} \mathbb{I}(z_j = c),$$

$$\mathbb{P}(z_i = c_{new} \mid z_{-i}, x_i, \mu) \propto \int p(x_i \mid \mu) \, dG_0 \, \alpha.$$

Next, the cluster means are updated by sampling from the posterior distribution:

$$p(\mu_c \mid x, z) \propto G_0(\mu_c) \, p(x_i \mid \mu_c)$$

A.2.3 DP means

It can be shown [21] that when the variance of the underlying distribution goes to 0, the cluster assignment step in the CRP Gibbs sampler becomes a hard assignment:

$$z_i \leftarrow \operatorname*{argmin}_{c \in \{1, \ldots, k, c_{\text{new}}\}} \tilde{D}_\phi(x_i, c)$$

where

$$\tilde{D}_\phi(x_i, x) = \begin{cases} D_\phi(x_i, \mu_c) & \text{if } c = 1, \ldots, k, \\ \lambda & \text{if } c = c_{\text{new}}, \end{cases}$$

and $D_\phi(x_i, \mu_c)$ is the Bregman divergence from x_i to μ_c associated with the exponential family, and λ is a user-defined threshold. Hence, if the Bregman divergence of x_i to every cluster mean is greater than λ, we create a new cluster with mean exactly at x_i.

The sampling of the mean parameter reduces to computation of the posterior mean:

$$\mu_c \leftarrow \frac{\sum_i \mathbb{I}(z_i = c) x_i}{\sum_i \mathbb{I}(z_i = c)}$$

The resultant algorithm can be seen as a coordinate descent on the objective function

$$\sum_{c=1}^{k} \sum_{x} \mathbb{I}(z_i = k) D_\phi(x, \mu_c) + \lambda k$$

where k is the number of clusters. Hence, the algorithm tries to optimize the k-means objective, while taking a penalty of λ for each cluster created.

A.2.4 Distributed DP means

Both the CRP Gibbs sampler and the original DP means algorithm are sequential in nature. To the best of our knowledge, the algorithm we propose in this paper is the first to parallelize/distribute DP means over a cluster.

The main difficulty in parallelizing DP means lies in the potential of every datapoint creating a new cluster. We observe, however, if all the Bregman divergences $\min_c D_\phi(x, \mu_c)$ is less than λ, then no new clusters will be created. A datapoint x is said to be "covered" by a cluster c centered at μ_c if $D_\phi(x, \mu_c) < \lambda$. If x is not covered by any cluster, it is said to be "uncovered".

At the end of the cluster assignment step of DP means, the following invariants are achieved:

1. All data points are covered by at least one cluster.
2. Any new cluster created must have mean at some datapoint x_i, which was uncovered at the start of the cluster assignment.
3. Every datapoint is assigned to a cluster that covers it.

Our distributed algorithm maintains the first 2 invariants by introducing a distributed, conservative cluster creation process. Given the current global set of means μ_c, each slave machine simulates the cluster assignment step by iterating through its data points, and locally creating a new cluster centered at x_i whenever it encounters an uncovered datapoint x_i. Note that some data points may subsequently be assigned to the new local cluster; we will deal with these data points later.

The set of new local cluster means are then sent to the master, which simulates the cluster assignment on the proposed cluster means, promoting a local cluster mean to a global mean whenever an uncovered local cluster mean is found.

As new global clusters are created, some of the local cluster means x_i's may now be covered by the new global clusters. There may, however, be some data points on the slave machines that had been assigned to the simulated local cluster x_i, but remain up to 2λ after from any global cluster mean. Thus, we send the new set of global centers to all slave machines, and repeat the cluster creation process after discarding all simulated local clusters, until no datapoint is left uncovered.

It is easy to see that the iteration will eventually terminate, since there are a finite number of data points, and each iteration covers at least one datapoint. Alternatively, we can configure the master to promote, at each iteration, the entire set of local cluster means from any particular machine. This is possible because the local cluster means from a single slave do not cover one another. Furthermore, upon promotion of its entire set of local cluster means, all data points from the machine will now be covered, so no new clusters are proposed. Thus, the cluster creation process makes progress at at least one slave machine per iteration.

Upon completion of the cluster creation process, we have achieved the invariant of covering all data points, and with new global clusters centered at previously uncovered data points. To achieve the third invariant, we simply run the cluster assignment step of DP means once. Since all data points are now covered, no new clusters will be generated, and the assignment can take place in parallel.

Finally, we update the means by collecting the sufficient statistics ($\sum_i \mathbb{I}(z_i = c)$, $\sum_i \mathbb{I}(z_i = c)x_i$) for each cluster c, and computing the new mean $\mu_c \leftarrow \frac{\sum_i \mathbb{I}(z_i=c)x_i}{\sum_i \mathbb{I}(z_i=c)}$.

Algorithm 2: Distributed DP means algorithm.

```
   Data   : RDD $\mathcal{X} = \{x_i\}$, threshold $\lambda$
   Result: Set of cluster means $\{\mu_c\}$
1  Initialize centers ← {};
2  while not converged do
3      while ∃ x uncovered do
4          newLocalMean ← cover($\mathcal{X}$, centers, $\lambda$);
5          newGlobalMean ← cover(newLocalMean, {}, $\lambda$);
6          centers ← centers ∪ newGlobalMean;
7      end
8      assignment ← classify($\mathcal{X}$, centers);
9      suffStats ← computeSufficientStats($\mathcal{X}$, assignment);
10     Update centers using suffStats;
11 end
12 return centers
```

Algorithm 3: Cluster creation algorithm.

```
   Data   : Data points $\{x_i\}$, current centers, threshold $\lambda$
   Result: Proposed new centers
1  Initialize newCenters ← {};
2  for $x \in \mathcal{X}$ do
3      if $\forall \mu_c \in$ centers ∪ newCenters, $D_\phi(x, \mu_c) > \lambda$ then
4          newCenters ← newCenters ∪ {x}
5      end
6  end
7  return newCenters
```

The distributed cluster creation (line 3 of Algorithm 2) is executed using our BatchMap primitive. While we could have used the standard Map primitive, by using BatchMap we can reduce the potential number of new clusters proposed, and minimize the load on the master in executing line 5, thus achieving a more scalable solution.

A.2.5 Serializability of distributed DP means

Our distributed DP means algorithm can be shown to be serializable, in that there is some order of updates that would have resulted in the same outcome if the updates were performed serially on a single machine.

To see why this is so, we first observe that in the distributed cluster creation process, the slave machines simulate the classify procedure locally, which induces a partial ordering of the data store locally, based on the time at which they are determined to be covered. Similarly, the master machine induces a partial ordering on the data points that it accepts as new global centers. It is clear that these partial orderings are consistent with one another, which in turn implies there is some total ordering that is consistent with the partial orderings. By running the classify procedure on data points according to the total ordering, we will obtain the same set of new clusters.

The remainder of the algorithm (running classify after all data points are covered, and then updating the means) are easily serializable. As such, the entire algorithm is serializable, so the distributed DP means algorithm is guaranteed to return a solution that is attainable by the sequential DP means algorithm.

A.3 SVM

Given a training set of labeled data $S = \{(x_i, y_i) \colon i = 1, \dots, n\}$ with $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, +1\}$, the SVM algorithm can be cast as solving the following optimization problem:

$$\min_{w,b} \frac{1}{n} \sum_{i=1}^{n} \big(1 - y_i(w^\top x_i + b)\big)_+ + \lambda \|w\|_2^2, \tag{3}$$

where $(t)_+ = \max\{0, t\}$ and $\lambda > 0$ is a regularization parameter that controls the trade-off between the discriminative power and the generalization ability of the resulting classifier. Equation (3) is the primal form of SVM where we are working with the Euclidean geometry of the input data. In the dual formulation of SVM we can use a kernel function $K(x, x')$ in place of the inner product $x^\top x'$ to capture nonlinear correlation among the data [9, 28]. However, we presently only focus on linear SVM.

By considering the $(d+1)$-dimensional augmented vectors $\tilde{x}_i = (x_i^\top, 1)^\top$ and $\tilde{w} = (w^\top, b)^\top$, we can write the optimization problem (3) as

$$\min_{\tilde{w}} \frac{1}{n} \sum_{i=1}^{n} (1 - y_i \tilde{w}^\top x_i)_+ + \lambda \mathcal{P}(\tilde{w}), \tag{4}$$

where now the penalty function is $\mathcal{P}(\tilde{w}) = \sum_{i=1}^{d} \tilde{w}(i)^2 = \|w\|_2^2$. That is, we only regularize the weight vector w but not the offset b. In the rest of this section we are working with these augmented vectors, and for simplicity we write w and x_i in place of $\tilde{w}$ and $\tilde{x}_i$.

A.3.1 Parallelizing SVM via ADMM

Alternating Direction Method of Multipliers (ADMM) is a technique for solving a general convex optimization problem that combines the decomposability property of dual ascent methods with the good convergence properties of the methods of multipliers [17, 16, 6]. ADMM forms the augmented Lagrangian for a given convex problem and proceeds by alternatingly optimizing the primal and dual variables. ADMM is well suited to distributed convex optimization problems because some of the update steps could be performed in parallel. In particular, given a large-scale convex optimization problem, we can convert it into a consensus problem that can be solved in a distributed fashion via ADMM.

In the context of SVM, suppose our training data S is partitioned into K disjoint subsets $S_1, \dots, S_K$, where $|S_k| = n_k$ with $\sum_{k=1}^K n_k = n$. For $1 \le k \le K$, define the local loss of the classifier in each machine:

$$f_k(w) = \frac{1}{n} \sum_{i \in S_k} (1 - y_i w^\top x_i)_+.$$

Observe that we can write the optimization problem (4) as the following consensus problem (recall that we are working with $(d+1)$-dimensional augmented vectors):

$$\min_{\mathbf{v}, w} \sum_{k=1}^{K} f_k(v_k) + \lambda \mathcal{P}(w) \;\text{ s.t. }\; v_k = w \;\text{ for }\; 1 \le k \le K, \tag{5}$$

where $\mathbf{v} = (v_1, \dots, v_K)$ with $v_k \in \mathbb{R}^{d+1}$. Proceeding in the ADMM framework, we form the augmented Lagrangian of the problem above:

$$\mathcal{L}_\rho(\mathbf{v}, w; \mathbf{z}) = \sum_{k=1}^{K} f_k(v_k) + \lambda \mathcal{P}(w) + \sum_{k=1}^{K} z_k^\top (v_k - w) + \frac{\rho}{2} \sum_{k=1}^{K} \|v_k - w\|_2^2.$$

Here, $z_k \in \mathbb{R}^{d+1}$ is a dual variable for the constraint that $v_k = w$, and $\rho > 0$ is a smoothness parameter that makes the augmented Lagrangian strongly convex. ADMM solves the consensus problem (5) by iteratively performing the following updates:

$$\mathbf{v}^{(t+1)} = \operatorname*{argmin}_{\mathbf{v}} \mathcal{L}_\rho(\mathbf{v}, w^{(t)};\, \mathbf{z}^{(t)}) \tag{6}$$

$$w^{(t+1)} = \operatorname*{argmin}_{w} \mathcal{L}_\rho(\mathbf{v}^{(t+1)}, w;\, \mathbf{z}^{(t)}) \tag{7}$$

$$\mathbf{z}^{(t+1)} = \mathbf{z}^{(t)} + \rho(\mathbf{v}^{(t+1)} - w^{(t+1)}). \tag{8}$$

We make some remarks below.

v-update. Since the augmented Lagrangian $\mathcal{L}_\rho$ is separable in the v_k's, the update in (6) splits into K separate updates: for each $1 \le k \le K$,

$$v_k^{(t+1)} = \operatorname*{argmin}_{v_k} f_k(v_k) + z_k^{(t)\top} v_k + \frac{\rho}{2} \|v_k - w^{(t)}\|_2^2. \tag{9}$$

If the data partitions $S_1, \dots, S_K$ lie in different machines, then the updates (9) can be performed in parallel. We use stochastic forward-backward splitting to solve each instance of the update (9); see Section A.3.2 for more details.

w-update. Recalling that the penalty function is $\mathcal{P}(w) = \sum_{i=1}^d w(i)^2$, the update of $w \in \mathbb{R}^{d+1}$ takes the form

$$w^{(t+1)} = \operatorname*{argmin}_{w} \lambda \sum_{i=1}^{d} w(i)^2 - \sum_{k=1}^{K} z_k^{(t)\top} w + \frac{\rho}{2} \sum_{k=1}^{K} \|v_k^{(t+1)} - w\|_2^2.$$

The quadratic optimization problem above can be solved explicitly for each component of the vector w:

$$w^{(t+1)}(i) = \begin{cases} \dfrac{K}{\rho K + 2\lambda} \big(\overline{z}^{(t)}(i) + \rho \overline{v}^{(t+1)}(i)\big) & \text{if } 1 \le i \le d, \\ \dfrac{1}{\rho} \overline{z}^{(t)}(d+1) + \overline{v}^{(t+1)}(d+1) & \text{if } i = d+1, \end{cases}$$

where $\overline{z}^{(t)} = \frac{1}{K} \sum_{k=1}^K z_k^{(t)}$ and $\overline{v}^{(t+1)} = \frac{1}{K} \sum_{k=1}^K v_k^{(t+1)}$.

z-update. The update in (8) also splits into K separate updates:

$$z_k^{(t+1)} = z_k^{(t)} + \rho(v_k^{(t+1)} - w^{(t+1)}) \quad \text{for } 1 \le k \le K.$$

As in the case of the **v**-update, this step can be performed in parallel. Thus, at each iteration of ADMM we can perform the **v** and **z** updates in parallel across K machines, and in the w-update we synchronize the results from all the machines.

Stopping condition. Following [6], a practical stopping condition for the ADMM algorithm is when the residuals fall below an adaptive threshold. That is, we run the ADMM algorithm until

$$\|r^{(t)}\|_2 \le \epsilon_{\text{pri}}^{(t)} \quad \text{and} \quad \|s^{(t)}\|_2 \le \epsilon_{\text{dual}}^{(t)},$$

where

$$\|r^{(t)}\|_2 = \Big(\sum_{k=1}^{K} \|v_k^{(t)} - w^{(t)}\|_2^2\Big)^{1/2}$$

is the primal residual,

$$\|s^{(t)}\|_2 = \rho\sqrt{K}\|w^{(t)} - w^{(t-1)}\|_2$$

is the dual residual, and

$$\begin{aligned}\epsilon_{\text{pri}}^{(t)} &= \epsilon_{\text{abs}}\sqrt{(d+1)K} \\ &\quad + \epsilon_{\text{rel}} \max\big\{\big(\sum_{k=1}^{K}\|v_k^{(t)}\|_2^2\big)^{1/2},\ \sqrt{K}\|w^{(t)}\|_2\big\}, \\ \epsilon_{\text{dual}}^{(t)} &= \epsilon_{\text{abs}}\sqrt{(d+1)K} + \epsilon_{\text{rel}}\|z^{(t)}\|_2\end{aligned}$$

are adaptive thresholds for the primal and dual residuals. Here ϵ_{abs} and ϵ_{rel} are absolute and relative tolerance thresholds, respectively. In our simulations we set $\epsilon_{\text{abs}} = \epsilon_{\text{rel}} = 10^{-3}$, and we observe that the ADMM algorithm typically converges within 50 iterations.

A.3.2 Forward-Backward Splitting for the $\mathbf{v}$-updates

In this section we describe how to perform the $\mathbf{v}$-update described in (9). Fix an index $1 \le k \le K$ and iteration t, and for simplicity of exposition we remove all notational dependence on k and t, so we want to solve the following optimization problem:

$$v^* = \operatorname*{argmin}_v f(v) + z^\top v + \frac{\rho}{2}\|v - w\|_2^2.$$

A consideration of the objective value above at $v = w$ and a simple algebraic calculation reveals that the unique minimizer v^* lies in the ℓ_2-ball $B(w_c, R)$ centered around $w_c = w - \frac{1}{\rho}z$ with radius $R = \sqrt{\frac{2}{\rho}(f(w) + \frac{1}{2\rho}\|z\|_2^2)}$. Thus, we can rewrite the optimization problem above as

$$v^* = \operatorname*{argmin}_{v \in B(w_c,R)} f(v) + \varphi(v) \qquad (10)$$

where $\varphi(v) = z^\top v + \frac{\rho}{2}\|v - w\|_2^2$.

We solve the problem (10) using forward-backward splitting [12]. The idea of this technique is to replace the function $f(v)$ by its first-order approximation around our current estimate of the minimizer, and then use this approximation to update our estimate. Specifically, at each iteration j we have our estimate $v^{(j)}$ and we update it via

$$\begin{aligned}v^{(j+1)} = \operatorname*{argmin}_{v \in B(w_c,R)} \Big\{ & f(v^{(j)}) + \langle \nabla f(v^{(j)}),\ v - v^{(j)}\rangle + \varphi(v) \\ & + \frac{1}{2\alpha_j}\|v - v^{(j)}\|_2^2 \Big\},\end{aligned}$$

where α_j is a step size and $\nabla f(v^{(j)})$ is a subgradient of f at $v^{(j)}$. This is equivalent to the following two-step update:

$$\begin{aligned}v^{(j+\frac{1}{2})} &= v^{(j)} - \alpha_j \nabla f(v^{(j)}), \\ v^{(j+1)} &= \operatorname*{argmin}_{v \in B(w_c,R)} \Big\{\frac{1}{2}\|v - v^{(j+\frac{1}{2})}\|_2^2 + \alpha_j\varphi(v)\Big\}.\end{aligned}$$

Therefore, forward-backward splitting can be interpreted as first taking an unconstrained gradient step and then adjusting it to achieve low complexity, as measured by the statistical regularizer $\varphi(v)$.

In our case we have $f(v) = \frac{1}{n}\sum_{i \in S_k}(1 - y_i v^\top x_i)_+$, so computing the subgradient $\nabla f(v^{(j)})$ involves summing over $n_k = |S_k|$ data points. As in the stochastic gradient descent, we approximate $\nabla f(v^{(j)})$ by

$$\frac{n_k}{n}\nabla(1 - y_i v^{(j)\top} x_i)_+ = \begin{cases} -\frac{n_k}{n} y_i x_i & \text{if } y_i v^{(j)\top} x_i < 1, \\ 0 & \text{otherwise,}\end{cases}$$

where (x_i, y_i) is sampled uniformly at random from the set S_k. The presence of the ratio n_k/n is to make the approximation an unbiased estimator of the true subgradient. Once we have $v^{(j+\frac{1}{2})}$, computing $v^{(j+1)}$ amounts to minimizing a quadratic function followed by a projection step to $B(w_c, R)$, which can be performed efficiently.

B. ALGORITHM PERFORMANCE

It is important the models learnt by the distributed algorithms are of high quality. For classifiers such as LogitBoost and SVM, the metric of quality is the accuracy of the classifier on a held-out data set. For clustering, recall that DP means has an objective function that it attempts to iteratively minimize. We evaluate these measures of quality in the weak scalability seting, i.e. number of machines grow with data set size.

LogitBoost.

Accuracy of LogitBoost models on held-out test data is shown in Figure 10. As expected, the sequential LogitBoost on MATLAB outperforms our distributed LogitBoost algorithms, since the former is able to always find the globally optimal solution. Interestingly, the LogitBoost model that always picks the locally best weak classifier at each iteration has a nearly constant accuracy, regardless of the data set size. This may be indicative that the LogitBoost model cannot be further optimized with more data, if it is constrained to always choose the weak classifier with the lowest local loss. If, however, the LogitBoost model can propose random weak classifiers, and recompute the optimal weights, then its test accuracy can increase with more data.

Note that the differences in test accuracies are less than 1%, so more runs are required before we can make conclusive statements about the difference in performance of the 3 models.

DP Means.

The DP means objective is $\sum_{c=1}^{k}\sum_x \mathbb{I}(z_i = k) D_\phi(x, \mu_c) + \lambda k$. As the amount of data increases, we expect that the objective of the true model increases linearly. This is in fact what we observe in Figure 11. The objectives of DP means models learnt by the three algorithms are virtually indistinguishable – this should be of little surprise, since our distributed DP means algorithm is serializable and thus equivalent to some execution of the sequential DP means algorithm.

SVM.

We compare the accuracy of our implementation of distributed SVM against PSVM. We set the λ parameters to be

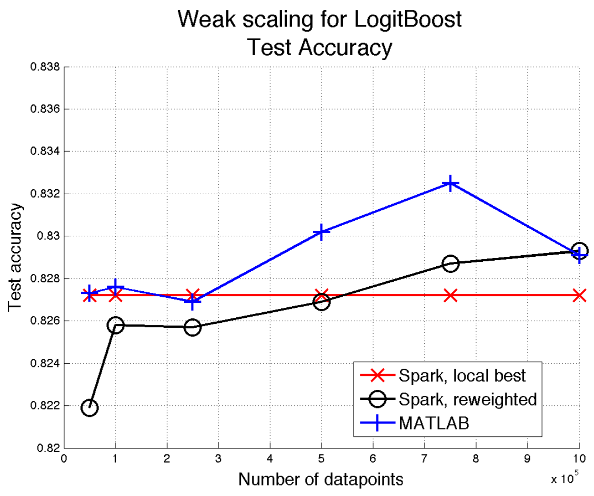

Figure 10: Test accuracy of LogitBoost as more data is used for training.

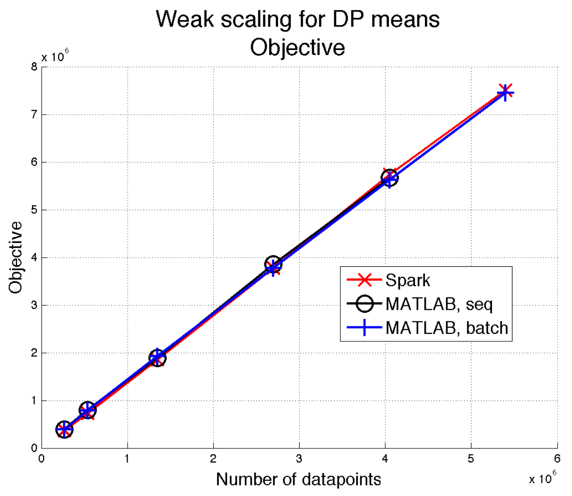

Figure 11: DP means objective, as more data is clustered.

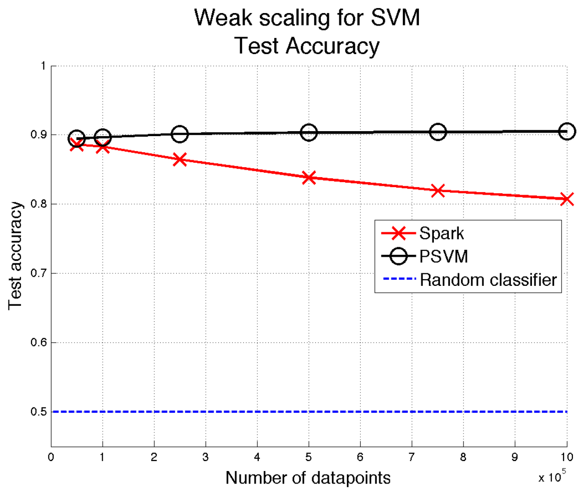

Figure 12: Test accuracy of SVM as more data is used for training.

the same in both sets of experiments. We do not include the accuracy results for the MATLAB SVM solver because the choice of λ that we used was not scaled properly, and consequently, the accuracy results are not comparable to those from Spark and PSVM.

From Figure 12 we see that PSVM achieved essentially constant accuracy across different number of data points and machines, while the accuracy of our Spark implementation deteriorates slightly over time. We currently do not have a good explanation for this behavior; we suspect that this might be because we use the same absolute and relative threshold parameters for checking the convergence of the ADMM algorithm (although we then use these thresholds to construct adaptive primal and dual thresholds at each iteration), but this requires further investigation.