1. INTRODUCTION

With the rising public use of the Internet in modern times, companies such as Google or Facebook are gathering increasingly massive amounts of data. These companies often want to leverage their huge databases to learn and predict hidden information. For example, Netflix utilizes user ratings to improve their recommendation system, which finds new and interesting movies for their users to watch. In order to properly process these mountains of information, researchers have invested considerable into the development of distributed algorithms for various machine learning tasks.

For example, the Noisy Matrix Factorization problem is at the core of many online recommendation systems, and is also useful in concisely representing data. Recently, a strongly parallel algorithm for this problem called Divide-Factor-Combine was developed at UC Berkeley[9].

Though it is possible to determine the best parameters empirically, this is both computationally expensive and prone to error. In addition, different datasets may respond differently to parameter settings, and parameter settings that were good for a problem instance may become suboptimal as time progresses and the input data changes.

In this work we describe a general system that, given an input problem, a distributed algorithm, and a computational budget, automates parameter choices for distributed matrix computations. We implement this optimizer, focusing specifically on automating the choice of degree of parallelism. As a trial application, we use the optimizer on the Divide-Factor-Combine algorithm (DFC), a recently-developed matrix factorization algorithm. We find that running the algorithm with the optimizer-chosen parameters yields errors that are as low as possible given the time and monetary budgets imposed.

1.1 The Problem

These machine learning algorithms often take in many parameters, and to achieve the best results, the input parameters must be optimized for each specific application. In addition, the use of parallelism and the nature of convergent algorithms creates a tradeoff between time, accuracy, and the number of processors or machines tasked with the computation. On the one hand, too much parallelism can cause the algorithm to lose error guarantees and perform inadequately. On the other hand, too little parallelism can make the computation take an unacceptably long amount of time. On top of that, monetary budgets may restrict the number of machines or processors one is willing to dedicate to the task, limiting the capacity for parallelism. The objectives of minimizing both the error and time of the computation within a monetary budget are obviously in conflict.

Further, different users may have different requirements over these domains. In the case of an online recommendation system, as more customers buy or rate products, it is necessary to rerun the machine learning task in order to ensure relevant and current recommendations. Here the user may have some time budget describing how long he can wait for the machine learning algorithm to run, and would want to minimize the prediction error of the algorithm subject to this budget. On the other hand, a genome scientist who has collected partial genetic data across many organisms may care more about maximizing the accuracy of the phenotype prediction, but have a much more relaxed time budget.

Either extensive empirical tests or an operator who is extremely familiar with both the dataset and the algorithm is required in order to achieve optimal results for a particular problem instance. In addition, optimized parallelization choices might meet time budgets on one particular architecture but fail on another, as communication times and machine speeds can vary drastically. For these reasons, it is very desirable to have an automated process that can pick settings that meet budgets reliably, and solve the problem optimally within budget constraints; such an automated process could be architecture-independent, and adapt as the distribution of input data changes over time.

1.2 Our Contribution

In this work we propose an optimizer for choosing parameter settings for distributed machine learning algorithms, in particular for distributed matrix computations. Given input data and user-specified budgets, our optimizer automatically
chooses algorithm parameters to minimize either time, error, or monetary expenditure while not going over-budget in any of these realms. The optimizer maintains a database of the parameter choices and empirical performance of previous jobs, and uses this stored information to choose optimal parameters for new incoming jobs.

Our optimizer has done quite well in tests, consistently performing as well as the best possible manual choice of parameters. Our preliminary implementation currently optimizes the Divide-Factor-Combine algorithm for Noisy Matrix Factorization. Our tests were performed on both synthetic and real-world data. Our synthetic data was in the form of Gaussian random matrices, and our real-world data was the Movielens10M dataset. We have designed out optimizer to be general and easily extensible, and adding additional machine learning algorithms to our optimizer framework is the subject of ongoing work.

2. OPTIMIZER DESIGN

When designing the optimizer, our goals were to provide a simple interface to the user and ensure architecture-independence. To meet the objective of a simple user interface, the optimizer must be able to make parameter choices based only on user-specified time, accuracy, and monetary budget constraints. To meet the objective of architecture-independence, the optimizer needs to gather data about jobs run in its specific architecture context. Below, we describe the design of the optimizer in pursuit of these goals.

2.1 Optimizer System Context

In this subsection, we discuss interactions between the optimizer and the user and algorithmic infrastructure that together form the system.

2.1.1 Interface to User

The user’s interface with the optimizer is simple: the user provides a problem instance for the job and constraints on the amount of time and money that the job can spend and on the amount of error that is acceptable for the output. The user also specifies $X$, the parameter to be optimized (currently, $X$ can be total runtime, total error, or amount of money spent).

The user may also specify whether to run the optimizer in explore mode, a mode suitable for high-variance data applications or if the optimizer is in the initial stages of learning a new distribution (see Section 2.3.1). The user can also specify whether to run the optimizer in estimation mode, a mode in which the optimizer augments its data with estimates of runtime and error based on previous jobs from a different parameter subspace (see Section 2.3.3). This is for example useful if the optimizer has learned many examples of smaller or larger problem instances, but has relatively sparse data for instances of a given job’s size.

The user also specifies which distribution the data comes from (by distribution, we simply mean the source of the data, since data from different sources may behave differently). Currently, this is accomplished by having the user point the optimizer to the source of the learned data. When using the optimizer for the first couple of times, the user can input statistics from training on similarly distributed data. There is also a more cursory mode available, in which the optimizer runs a subproblem and obtains estimates for a wider variety of parameter settings (see Section 2.3.2). In this mode, the user need not specify the distribution of the data.

2.1.2 Interface with Algorithm and Infrastructure

Our optimizer relies on access to an already implemented distributed algorithm. The optimizer outputs the parameters that the algorithm should run with; in the current implementation this is limited to the number of processors and the number of iterations, although this can be easily extended to algorithm-specific parameters (i.e. learning rate). The algorithm itself is expected to communicate with the distributed computation framework, although this too could be altered if necessary.

After the algorithm returns, the optimizer requires information about the outcome of the job so that it can learn. We want to learn the error and runtime as a function of the number of iterations and the number of processors used. Because the input parameters are determined by the optimizer (and thus already known to it), all the optimizer requires from the algorithm’s output is the error and the runtime. In principle, if there is an additional parameter over which one wishes to optimize (say, for example, testing error in addition to training error), additional information about the algorithm output must be stored.

2.2 Learning

Our optimizer’s parameter selection strategy is based on statistical data from prior runs. To ensure that the optimizer’s choices are architecture-independent, and to allow the optimizer to adapt to data from different distributions and to continually improve its predictions, our optimizer learns from every job that it encounters.

This amounts to recording each job’s distribution (the actual probability distribution from which the data comes does not have to be specified—this is simply a tag, such as “movie-lens data”, since we expect different kinds of input to have different rates of convergence, etc.), runtime, error, and the algorithm parameters with which it was run. If there is another value which one wishes to optimize over, such as training vs. testing error, this is information which should be stored as well.

Additionally, it is possible to seed the optimizer with some manually constructed training data. If the user already has information about previous job runs, or if the user prefers to populate the database with some systematic parameter setting tests, the optimizer can learn from these as well.

2.3 Choosing Parameters

In this subsection, we discuss the way in which our optimizer utilizes its data to select parameters, and the way in which it obtains information in the absence of data.

2.3.1 Choosing Parameters using Learned Data

---

When given a problem instance with a particular budget, our optimizer consults data from prior runs in order to select the algorithm parameters.

What our optimizer does is it scans over all job instances from the same distribution. If estimation mode is enabled, the optimizer considers all previous similarly distributed jobs regardless of size, making estimates of error and runtime expected for an instance of the current size. If estimation mode is not enabled, then the optimizer discards information about all previous jobs which are not within a factor of 2 of its size—here, size is determined not by the dimensions of the matrix but rather by the number of entries. The factor of 2 was chosen using non-rigorous empirical trials—it may be that this is not an appropriate choice for all datasets, but in principle it is easy to learn this parameter as well, and this is one direction for future work.

After collecting the relevant prior jobs, the optimizer takes an average over values of time, error and money for identical parameter settings, and then chooses the settings that minimize the average value of X while still meeting the budget constraints. Though we use a straightforward average here, it may make sense to include the option of taking an exponentially weighted moving average. In the interest of keeping the interface simple, we did not implement this, though in principle this too could be added with very little additional effort.

2.3.2 Choosing Parameters in the Absence of Data

Thus far, our described methods of parameter selection rely on a database of information about previously completed jobs. However, when instances from a new data distribution are initially run, there is no past data upon which we can draw.

For this setting, we do the following: first, we populate our database with several smaller jobs—this is currently achieved by running the algorithm with a couple of parameter settings for very few iterations. This strategy can easily be modified as appropriate—for example, if minimizing error with a large time budget, and a small monetary budget, it may make sense even in this population step to have larger numbers of processors, or for some algorithms it may make sense to run the optimizer on a well-chosen submatrix. Exploring the best strategy for rapidly populating the database is an interesting direction for future work.

This gives us some very rough, cursory data at relatively little cost. We then run the optimizer on the job using this initial data in estimation mode and explore mode.

2.3.3 Estimation Mode

Under some circumstances, it is natural that only a limited subspace of parameter settings has been explored. It is natural to try to extend what we know about that subspace to gain information about other parameter choices. For example, if we see only relatively small matrices from a given distribution, we may want to use our data to select parameters for a larger one; or, if we have been operating with a tiny monetary budget and suddenly are able to afford more processors, we may want to use our data for 1 and 2 processor jobs to choose parameters when more parallelism is available.

Estimation mode tells the optimizer to create estimates of error, time to completion, and expenditure as a function of number of iterations, number of processors used, and the size of the instance. The optimizer can then use these estimates as synthetic data to populate unknown areas of the parameter space, and thus make better parameter choices when data is unavailable.

The way we’ve chosen to do estimation is to try to fit data from prior jobs to models for time and error as functions of iterations, number of processors, and instance size.

Error estimation for DFC: From our empirical observations, the error in DFC is an exponential function of the number of iterations, with an additive term which depends on the number of processors:

\[ \text{err}(s,t) = A \cdot s + e^{-B \cdot t}. \]

where \( s \) is the number of processors and \( t \) is the number of iterations, and \( A \) and \( B \) are unknown constants. As far as we have been able to discern, the error for DFC is not a function of instance size; however, this may be because of the particular algorithm we chose for our tests. In any case, this should vary from algorithm to algorithm.

Runtime estimation for DFC: For DFC, we have observed time to completion to be a function of the number of entries and the number of processors:

\[ \text{time}(s,t,n) = C \cdot \frac{t \cdot n}{s}. \]

where again \( s \) is the number of processors, \( t \) is the number of iterations, \( n \) is the number of revealed entries in the matrix, and \( C \) is some unknown constant. This is essentially an application of Amdahl’s Law \( \frac{1}{1 + \frac{1}{n}} \), but for the algorithm we chose to test on (DFC) the serial time is negligible compared to the parallel time. The model for the time dependence should also vary with the specific algorithm in question.

The way that the optimizer now works is it fits the extant data to these models, uses the models to estimate the range of parameters which meet the budgets, and then populates the database with synthetic data over the relevant range. Then, the optimizer performs its usual parameter selection process, choosing the number of iterations and number of processors that optimize \( X \) according to these estimates. In practice, the accuracy of our estimates is quite low, and also very dependent on the quality and quantity of data we already have (see Section 4.5). The direction of greatest potential improvement here is more carefully choosing and fitting the modeling functions—this is an algorithm-specific task.

However, what matters is that the estimates of \( X \) for different parameter settings are relatively accurate; that is, if the true value of \( X \) will be better with parameter settings \( \theta_1 \) than with parameter settings \( \theta_2 \), then the estimated one should be better as well. Then, if the optimizer is simultaneously run in explore mode (Section 2.3.4), the quality of predictions increases as the optimizer encounters more instances (again, see Section 4.5).
2.3.4 Explore Mode
If the optimizer has access to few previous job outcomes, or if the data comes from a very high-variance distribution, there is a risk that suboptimal parameter settings will be chosen again and again, simply because there was one very successful run with those settings, or because the optimal settings are not known and the optimizer’s estimated values of $X$ are too high for other settings.

In order to prevent the optimizer from returning again and again to these local optima, the optimizer is able to run in explore mode. In explore mode, randomization is used to avoid local optima in parameter selection. Say that we have parameter settings $p_1, \ldots, p_m$ with estimated optimized values $x_1, \ldots, x_m$. Then the optimizer chooses the output parameters $P$ to be $p_i$ with probability

$$Pr[P = p_i] = \frac{1}{\sum_{j=1}^{m} x_j}.$$ 

This scheme has the property that

$$\frac{x_i}{x_j} = \alpha \implies \frac{Pr[P = p_i]}{Pr[P = p_j]} = \frac{1}{\alpha},$$

That is, a particular parameter setting is chosen with probability proportional to its relative minimization of $X$.

In this way, the randomness guarantees that the optimizer explores the parameter space. Thus as the number of jobs encountered grows, the optimizer’s data about the entire parameter space improves, and the influence of outliers on parameter choice is subdued.

3. DIVIDE-FACTOR-COMBINE
In order to test and evaluate our optimizer we implemented several variants of the DFC (Divide-Factor-Combine) framework for distributed matrix completion, originally introduced in [9].

3.1 DFC Framework Overview
In this section we describe the matrix factorization problem that the DFC algorithm solves, and also outline the three main steps in the algorithm.

3.1.1 Problem Definition
The DFC algorithm is used to solve the following well-studied noisy matrix factorization problem: Let $A$ be an unknown $m \times n$ matrix of rank $r$, and assume $r << m, n$. Next $A'$ is obtained by adding a small amount of noise to every entry of $A$. Finally, let $M$ be the matrix obtained by sampling a small fraction (e.g. 10%) of the entries of $A'$, and setting all other entries to zero. We refer to $M$ as the matrix of revealed entries. The goal is then to find a low-rank matrix $B$ that is a good approximation of the original matrix $A$, using only the matrix $M$. The standard method of finding such a low rank matrix $B$ is to find a low-rank factorization $B = UV^T$ that agrees with $M$ on all of its non-zero entries.

A version of this problem is actually encountered in practice by online movie-recommendation systems. Here the matrix has rows corresponding to users and columns corresponding to movies. The assumption that the matrix is low rank corresponds to the assumption that users movie preferences are only based on a relatively small number of factors (e.g. movie genre, movie length, user age etc). The matrix $M$ of revealed entries corresponds to the set of actual movie ratings that users have submitted.

This problem can be solved by the Stochastic Gradient Descent; however, these are serial algorithms which are not feasible for large datasets. The DFC algorithm harnesses these algorithms as subprocesses in a distributed setting.

3.1.2 The DFC Algorithm
The DFC framework for noisy matrix completion consists of three steps:

- **Divide** the input matrix $M$ column-wise into $k$ slices $\{M_1, M_2, \ldots, M_k\}$. More specifically, $M_i$ consists of $n/k$ columns from $M$ and concatenating all the $M_i$ gives the original matrix $M$. We call $k$ the division parameter of DFC.
- **Factor** each matrix $M_i$ as $UV_i^T$ on a separate processor using some base matrix factorization algorithm.
- **Combine** the factorizations $U_iV_i^T$ into one large factorization $UV^T$ using some matrix column projection algorithm.

From this description it is clear that there are several significant choices to be made in the implementation of this framework. First, both the base factorization algorithm and the projection algorithm for the combine step must be chosen. Second, the parameter $k$ for determining the number of slices to cut the matrix into has an impact both on runtime and accuracy of the result.

3.2 Base Factorization Algorithm Choices
We implemented two base matrix factorization algorithms: Stochastic Gradient Descent (SGD) and Accelerated Proximal Gradient Descent (APG). Both algorithms seek to minimize some objective function which is a combination of the error of the approximation and the rank of the factorization.

3.2.1 Stochastic Gradient Descent
Let $\Omega$ be the set of indices for the revealed entries (i.e. non-zero entries) of the input matrix $M$. The objective function for the Stochastic Gradient Descent algorithm is given by:

$$F(U, V, M) = \sum_{(i,j) \in \Omega} \left( M_{ij} - (UV^T)_{ij} \right)^2 + \mu(||U||_F^2 + ||V||_F^2)$$

where $||U||_F$ denotes the Frobenius norm of $U$ and $\mu$ is a parameter.

Intuitively, the first term in the objective function penalizes errors in matching the revealed entries of $M$ and the second term is a regularization term to prevent over-fitting. The SGD algorithm uses standard gradient descent techniques to compute argmin $F(U, V, M)$.

There are two important issues to note about the SGD algorithm. Firstly, the above minimization requires fixed choices for the dimensions of both $U$ and $V$. In particular, this
means that we have to first decide on a rank $r$ for our factorization. Then we can run SGD to find a rank $r$ factorization that approximates $M$ well. The second issue is that the objective function $F(U, V, M)$ is not convex, so there can be local minima where the SGD algorithm gets stuck.

### 3.2.2 Accelerated Proximal Gradient Descent

We use the Accelerated Proximal Gradient Descent algorithm described in [11]. The objective function for the APG algorithm is given by:

$$F(U, V, M) = \sum_{(i,j) \in \Omega} \left( M_{ij} - (UV^T)_{ij} \right)^2 + \mu \|UV^T\|_*$$

where $\|UV^T\|_*$ is the nuclear norm of $UV^T$. The key point here is that the nuclear norm is a convex relaxation of the rank. This means that the above objective in some sense tries to minimize both the error in matching the revealed entries of $M$, and some reasonable approximation of the rank of $B = UV^T$.

The algorithm has neither of the drawbacks of SGD. The user does not have to provide a guess for the rank, and the objective function is convex. Thus, the APG algorithm converges to a unique minimum, and there are theoretical guarantees on the convergence time.

### 3.2.3 Base Algorithm Choice: APG

We implemented both APG and SGD as described above, but it quickly became apparent that APG was superior for our purposes. The requirement that the user specify a rank for the factorization in SGD was a particularly large problem. First of all, having to specify a rank does not make a lot of sense when working with real-world data. For example, with a matrix of movie ratings it is not obvious apriori how many factors influence a user’s movie preference. This is something that one would like the algorithm to determine automatically, as APG does.

The second issue is that choosing the target rank of the factorization adds another parameter to the algorithm. Our optimizer uses data about the parameter settings of previous tasks to automatically choose parameters for a new task. As a result, the number of previous examples we need to cover the whole parameter space is exponential in the total number of parameters. So any additional parameter (such as the rank parameter to APG) greatly increases the number of examples required for the optimizer to perform well. So, while it is true that our optimizer can be configured to optimize over the choice of the rank as well, this was inconvenient for testing purposes.

The fact that the SGD objective function is not convex also causes multiple issues in our setting. We observed empirically that there was much more variance in the runtime and accuracy of SGD based on the starting point. Since the objective is not convex, we had to simply run SGD until the improvement at each iteration got small enough. Since there are several local minima, this can result in quite different run times even on matrices from the same distribution.

In comparison, APG has theoretical guarantees on its convergence, and we empirically observed that running for a fixed number of iterations on matrices from the same distribution resulted in relatively small variation in both error and runtime. This small variation is critical to the success of our optimizer in predicting the error and runtime of new matrix factorization tasks based on previous runs.

### 3.3 Matrix Projection Algorithm Choices

We implemented two possible algorithms for the matrix projection that occurs in the combine step of DFC. The first was Column Projection and the second was Randomized Projection.

#### 3.3.1 Column Projection

Given factorizations $(U_i, V_i)$ of each column slice $M_i$, the Column Projection algorithm simply projects each factorization $U_iV_i^T$ for $i \geq 2$ onto the column space of $U_1$. For each $i \geq 2$ this gives a new set of factorizations $U_i\hat{V}_i^T$. Now we let $V^T$ be the matrix obtained by the concatenation $[\hat{V}_1^T, \hat{V}_2^T, ..., \hat{V}_k^T]$. Then our final combined factorization is given by $U_1V^T$.

Column Projection is very fast as it simply requires $k$ matrix projections. It does however rely on the assumption that the columns of $U_1$ span the column space of the full low-rank factorization of $M$. In situations where the size $n/k$ of a column slice is large relative to the rank of the factorization, this assumption is somewhat reasonable. However, there is still the risk that this projection method results in a small loss in accuracy due to the assumption.

#### 3.3.2 Randomized Projection

The Randomized Projection algorithm we use was introduced by Halko et. al. in [7]. The algorithm is inspired by the Johnson-Lindenstrauss randomized method for metric embeddings. The algorithm involves sampling a random Gaussian matrix and computing several QR factorizations. As a result, the algorithm is slower than the Column Projection algorithm. However, it has better theoretical guarantees on accuracy.

#### 3.3.3 Projection Algorithm Choice: Randomized Projection

Despite the aforementioned tradeoffs in speed versus accuracy, we chose to use only the Randomized Projection algorithm. The reason for this was that it gave consistently better error than Column Projection, but the difference in runtime was very minimal. Further, the time taken by the projection step in DFC is completely dominated by the time taken for the actual matrix factorizations. In particular, the projection time is less than 1% of the time taken by the Factor step. As a result, accuracy at this step is more important than slight differences in run time.

### 4. EVALUATION

In this section we evaluate the ability of the optimizer to optimize a single parameter in the DFC algorithm with respect to very simple utility functions and budgets. We test the optimizer on both synthetic and real-world datasets and conclude that most of the time it performs as well as the best manual parameter settings.
4.1 Methodology

We evaluate the optimizer on both Gaussian Random matrices and the Movielens10M dataset (available at [http://grouplens.org/datasets/movielens/](http://grouplens.org/datasets/movielens/)). For each of the two datasets, we divided the available data into several partitions. For the random matrices we used five partitions, and this simply meant one matrix per partition. For the Movielens data (which is one enormous matrix), we partitioned the rows of the matrix into seven pieces (meaning, we subdivided the data into groups of users). This gave the two datasets a comparable number of nonzero entries per partition.

After partitioning the data, we group the partitions into training data and testing data. We used all but one partition for both datasets as training data on which to run DFC computations. The optimizer was then fed the input and output profiles of these computations to use in its database. We used the final partition to test the predictions of the optimizer.

Our goal was for the optimizer to automatically choose the degree of parallelism in the DFC algorithm, i.e. how much to slice the input matrix in the Divide step. We asked for the computations to be completed within a specified amount of time while minimizing the error, and DFC computations with manually-chosen parameters designed to meet these budgets were performed on this testing data, as well as a DFC computation with optimizer-chosen division. Error was computed using the Root Mean Square Error (RMSE) of the returned factorization compared to the revealed entries, and the time of a computation is defined as the maximum time of a slice computation plus the time it takes to finish the projection step of DFC.

The evaluation metric we chose was, for each time budget, the error of the optimized parameters compared to the error of the best run with manually-chosen parameters. We define the regret of the optimizer on these instances as the average over the budgets of the percent difference in error of the optimized and manual parameters. We will see soon that the optimizer performs as well as the best parameter for every budget, even when that parameter changes depending on the budget. Finally, we performed cross-validation, i.e. training and testing was done with each of the partitions used as testing data, and we obtain very consistent results.

We implemented the DFC algorithm in SparkR [12], and the optimizer in Python. All of our tests were performed on the Amazon EC2 Cluster using medium instances.

4.2 Gaussian Random Matrices

The first dataset we tested the optimizer on was a synthetic dataset containing many low rank randomly generated matrices. The matrices were generated according to the following procedure:

1. Fix parameters \( r, p, \) and \( \sigma^2 \) which are the rank, percent of revealed entries, and the variance of the noise respectively.
2. Generate two \( n \times r \) matrices \( A \) and \( B \) with entries drawn from \( \mathcal{N}(0, 1/\sqrt{\sigma}) \).

4.3 Movielens Data

The second dataset we used to test the optimizer was the Movielens10M dataset. The Movielens dataset is a set of entries (UserID, MovieID, rating), with the rating parameter being a half-integer between 1 and 5. One can think of these tuples as specifying the entries of a matrix with rows and columns indexed by users and movies.

The reason we might want to find a low-rank completion of this matrix is the assumption that there are a small number of properties that influence the way people rate movies.

<table>
<thead>
<tr>
<th>Testing Set</th>
<th>% Over Optimal Error</th>
<th>% Over Time Budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.63%</td>
<td>-11.7%</td>
</tr>
<tr>
<td>2</td>
<td>0.50%</td>
<td>-13.1%</td>
</tr>
<tr>
<td>3</td>
<td>0.42%</td>
<td>-13.8%</td>
</tr>
<tr>
<td>4</td>
<td>0.94%</td>
<td>-16.1%</td>
</tr>
<tr>
<td>5</td>
<td>0.53%</td>
<td>-12.4%</td>
</tr>
</tbody>
</table>

Table 1: Average regret and time for k-fold cross-validation of optimizer run on five Gaussian random matrices of dimension 4000 by 4000. Percentages are in relationship to best possible error and time budget respectively.
Then the two factors tell you how a person values each property, and how each movie is correlated with these properties. This lets you make predictions even on the hidden entries of the matrix, i.e. the movies a user has yet to see or rate.

There are some qualitative differences between this data and the synthetic data in the previous section. The entries come from the range $[1, 5]$ rather than being concentrated around zero, and thus the RMSE's for this section are much larger. The Gaussian matrices also come from a well-understood distribution; this means we can make theoretical guarantees about the behavior of DFC on these instances, and also that it behaves pretty consistently across different random matrices. In the Movielens10M set we have no such guarantee, but under the assumption that the order of people is randomized, we can hope that Movielens users behave similarly across partitions.

The dataset was partitioned by the UserID into seven different pieces and the pieces were grouped to form the training/testing splits mentioned in the first section. Figure 2 displays the optimizer's performance on one of these splits. As before, each curve represents one of the choices of the division parameter in DFC. Once again we observe the intersection points where the division parameter should be changed for optimality to be satisfied. The bold line represents the optimized parameters for each budget. The graphs here are qualitatively the same as the graphs from the previous section. For every time budget, the optimizer picks the best division parameter.

Table 2 summarizes the results of each of the training/testing splits for the Movielens dataset. Over all these splits, the optimized parameter choices perform within 2% over the minimal error, and come within 3% over the budget.

### 4.4 Training with Limited Data

In an effort to discover how much training data we need for accurate optimizer predictions, we also ran some tests on training/testing splits with smaller training sets. Instead of training on six data partitions, we tried training the optimizer on only two partitions and testing it on the remaining five. The optimizer performs basically as well as if we had trained on almost all the partitions. Table 3 summarizes these results. We believe this is because the data is qualitatively the same across all the partitions, so the additional partitions do not add a lot of information to the optimizer. Distributions with higher variance in behavior may require significantly more learning, though we have not yet tested this.

**Table 2: Average regret and time for k-fold cross-validation of optimizer run on the seven Movielens matrices. Percentages are in relationship to best possible error and time budget respectively.**

<table>
<thead>
<tr>
<th>Testing Set</th>
<th>% Over Optimal Error</th>
<th>% Over Time Budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0%</td>
<td>-11.9%</td>
</tr>
<tr>
<td>2</td>
<td>0.1%</td>
<td>-10.9%</td>
</tr>
<tr>
<td>3</td>
<td>-1.8%</td>
<td>-10.2%</td>
</tr>
<tr>
<td>4</td>
<td>-1.0%</td>
<td>-9.3%</td>
</tr>
<tr>
<td>5</td>
<td>1.1%</td>
<td>-10.5%</td>
</tr>
<tr>
<td>6</td>
<td>0.9%</td>
<td>-14.9%</td>
</tr>
<tr>
<td>7</td>
<td>-2.9%</td>
<td>2.3%</td>
</tr>
</tbody>
</table>
4.5 Testing Explore and Estimation Modes

The previous sections show that our optimizer performs quite well when the training set accurately reflects the behavior of the algorithm on the parameter space of future jobs. However, sometimes an application can receive an instance that is unlike the previously seen jobs. For example, a matrix from the same distribution with more revealed entries could take more time to process (this is a realistic example for recommender systems, in which the users add more and more ratings to the system). In this case, we want to estimate the computational costs in the unknown regions of the parameter space, and to use some randomness while picking our parameters so that we can adjust our estimates about the parameter space to reflect reality. Ideally, the optimizer will complete the estimate and explore phase without sacrificing too much in the error or time of the computation, so we pick parameters proportionally to the error we expect to get with those settings.

We tested this functionality in the setting where the size of the problem instance changes. To do this, we gave the optimizer training data from $2000 \times 2000$ Gaussian random matrices, and applied our estimation mode to guess runtime profiles for $4000 \times 4000$ Gaussian random matrices. Then, we gave the optimizer a sequence of ten $4000 \times 4000$ Gaussian random matrices and asked the computation to finish within 200 seconds. While exploring, the optimizer was able to find a parameter setting with an error that was within 15% of the best parameter choice that satisfies that budget that we could find in hindsight. It did not find better settings because we over-estimated the amount of time per iteration, and so the optimizer did not try running with more iterations because it thought that would violate the budget. This is a problem that could be much improved with better models for time per iteration, as discussed in the conclusion.

5. RELATED WORK

There has been a large amount of work on automatic parameter choice for machine learning and optimization algorithms. For example, automatic choice of regularization parameters for ridge regression algorithms was considered in [5]. In [6], the authors present an automatic method for choosing a pair of parameters for image processing tasks—namely de-blurring and zooming. Their technique involves iteratively alternating between optimizing one of the two parameters. These aforementioned results depend on assuming some underlying structure in the data, and exploiting that in order to provide analytic techniques that compute the optimal or near-optimal choice of parameter. They are also quite application specific.

There have also been attempts to automatically choose parameters for various optimization tasks using genetic algorithms. In [4], the authors attempt to use genetic algorithms...
to choose parameters in agent-based simulations—an algorithm class where very large numbers of tunable parameters are the norm. A genetic algorithms approach to choosing parameters for convex programming was introduced in [2]. One disadvantage of genetic algorithms for parameter optimization is that they require many iterations, each of which requires many test runs of the algorithm to be optimized with different parameters in order to compute relative fitness. As a result, if the underlying datasets of interest are large and change over time— as they do in our application—it is impractical to frequently re-run the genetic algorithm.

A generic approach for optimization of machine learning parameters based on statistical methods is presented in [10]. Here the authors model a machine learning algorithm via a Gaussian process, and use this model to predict good parameter choices for the algorithm. A general method for tuning Mixed Integer-Linear Programming algorithms based on previous runs appears in [2]. In this work, the authors run experiments with many different parameter settings, and then fix the parameters to be the empirically best results from these tests.

None of the above approaches are suitable for a context in which the input data changes over time and the user can specify budgets for resources such as computation time. They instead focus on optimizing parameters as a preprocessing step, and then fixing a choice of parameters to use thereafter. This differs from our approach in that we allow the user to dictate their computational constraints, and we use complete information about previous jobs in order to choose the best possible parameters. This is especially useful in tasks which are done routinely over a span of time. Also, none of the above works are relevant for choices regarding parallelism in distributed machine learning.

The most similar framework to our own is the distributed machine learning library MLBase [8]. This library, given a machine learning problem, attempts to find the best possible algorithm for the task, running them concurrently and returning periodic results to the user. MLBase does not currently support the DFC framework, and part of the motivation for our work is to add the DFC algorithm, as well as the optimizer/budget framework to MLBase.

6. CONCLUSIONS AND FUTURE WORK

We have designed and implemented an optimizer for distributed matrix computation. Our optimizer stores data about previous jobs in order to choose the best parameters with which to run an algorithm, allowing the user to only specify a budget and freeing the user of making the choices manually. Our optimizer performs well in our tests on both synthetic and real-world data.

While our optimizer is a layer that is essentially independent of the algorithm upon which we tested it, our experiences with DFC yielded some interesting conclusions. Notably, we found that Stochastic Gradient Descent, while a popular and widely used algorithm, is not robust to noise and slight changes in initial conditions, and behaves erratically on high-variance data. The more complex Proximal Gradient Descent algorithm was much more stable, and also has the advantage of requiring fewer parameter settings (which allows systems such as our optimizer to provide good choices after fewer jobs).

While the current system is a useful tool for automated parameter selection, it can be extended in many directions, and some of the system features can be further improved. One such feature is the estimation mode— though it currently does somewhat successfully converge to good parameter choices without exceeding user budgets, the estimations of runtime and error as a function of iterations, matrix size, and number of processors are highly inaccurate. This is likely a problem with the models we have selected; more carefully fitting the data to a model that is not chosen based on relatively few empirical samples would probably go a long way towards improving these estimates, and this in turn would greatly decrease the number of samples that the optimizer needs to encounter in estimation mode before it can make optimal parameter choices.

Another way in which our optimizer is limited is that currently, unless estimation mode is enabled, it does not choose points in the parameter space which have not yet been tried. This means that sometimes we have a relatively large amount of time or money left in our budget, which we could use on more iterations, but the optimizer does not do this because it doesn’t have information about whether it will go over budget. While it is infeasible to correct this without first

<table>
<thead>
<tr>
<th>Test Matrix</th>
<th>RMSE</th>
<th>Running Time (sec)</th>
<th>Slices</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.26</td>
<td>41.7</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>0.27</td>
<td>37.0</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>0.08</td>
<td>67.7</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>0.08</td>
<td>74.8</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>0.26</td>
<td>70.0</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>0.09</td>
<td>239</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>0.26</td>
<td>121</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>0.26</td>
<td>50.8</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>0.27</td>
<td>33.7</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>0.09</td>
<td>140</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 4: Parameter choices and results for running the optimizer in explore and estimation modes on ten Gaussian random 4000 by 4000 matrices, using data from 2000 by 2000 Gaussian matrices.
improving the accuracy of our estimates, it makes sense to have the optimizer automatically make estimates for and consider points in the parameter space which are “close” to instances it has already seen—this will enable a more thorough charting of the parameter space, which may yield a higher utilization of the computational resources within the budget.

In line with this, it may make sense to require each algorithm to provide the optimizer with a notion of “distance” between problem instances. In the case of DFC, our notion of distance was comparing the number of revealed entries in the problem instance matrix. However, this may not be a good notion of distance in every algorithm. If, for example, the algorithm is sensitive to high variation in the input data, it would be good to base parameter choice based on previous jobs that are similar in the variation level as well as size. Though the parameter choices for DFC are quite good, it would be interesting to run see if there is a more refined notion of distance that can be used, and then to relax the accuracy of the choice of “input distribution” and see whether the optimizer can still make accurate predictions.

Another target area for further investigation is whether more recent data should be more heavily weighted by the optimizer, both in the averaging function used in parameter selection and in the probabilities assigned in explore mode—while it is unclear that this will be helpful in all cases, there are applications in which this could certainly be relevant, such as in a real-world dataset whose makeup can change over time. For example, if current events make people more politically polarized, then old jobs might yield runtime estimates that are not accurate for the newer, differently distributed data.

Finally, it would be interesting to see whether the optimizer can also successfully optimize over the choice of algorithm. In some cases, different algorithms are more successful on different kinds of datasets, and it is difficult to discern ahead of time which algorithm will be more successful. Our optimizer may be particularly suited to make this kind of choice.

7. ACKNOWLEDGMENTS
We would like to personally thank our project mentors, Ameet Talwalkar and Shivaram Venkataraman, for suggesting our project and supporting us throughout the project. We would also like to express our gratitude towards our course professors, John Kubiatowicz and Anthony Joseph, for feedback and helpful advice. We are also indebted to our officemates, Aviad Rubinstein, Christos-Alexandros Psomas, Ning Tan, and our friend Jarett Schwartz, for putting up with the incessant loud office-space conversations that were a byproduct of this work.

References