Distributed Sparse Support Vector Machine for Feature Selection on High Dimensional Datasets

Xiaowei KUANG

Department of Computer Science and Engineering

The Chinese University of Hong Kong

Abstract

Feature selection is very useful in reducing model complexity and gaining insights into data. As practical data often come in high volume and high dimension which exceed the computing capabilities and storage limit of a single machine, distributed implementations of feature selection algorithms are highly desirable. Incorporating an up-to-date feature selection algorithm called Feature Generating Machine (FGM) and an efficient distributed system named Husky, we propose a novel distributed implementation framework for feature selection in this report. The input data are partitioned among different workers and the workers can work jointly to solve model solutions with limited network communication. By cleverly caching important data, our method can handle large-scale high dimensional problems.

1 Introduction

It has been estimated that as of 2012, about 2.5 exabytes (10^{18}) of data are created each day, and that number is doubling every 40 months or so [1]. This exponential growth of data in terms of volume and dimension since the 21th century has spurred the development of feature selection methods and distributed computing systems. In the big data era, linear SVM has been one of the most popular tools for large-scale classification. Extensive efforts have also been put in applying SVM for feature selection. While many of the SVM algorithms proposed have shown state-of-the-art performance in certain applications, most of them are restricted to single machine training. This motivates us to develop a distributed solution that is more scalable and cost-efficient.

In a Machine Learning setting, feature selection refers to the process of selecting a subset of relevant features from the input data. Feature selection is useful in a number of ways [2]: First, it helps us visualize and understand the data better. Second, it alleviates the curse of dimensionality problem present in many real-world problems, which leads to a more accurate predictor. Third, as a dimension reduction technique, it significantly reduces the size of the input data by eliminating irrelevant features, which gives rise to faster and more cost-effective predictors in terms of time complexity and space requirement. To name one concrete example, consider the text classification problem: an input instance (document) lies in a feature space of dimension the size of the vocabulary containing word frequency count, which can count to millions! Recently, many features selection methods using SVM have been proposed. Many of them achieve the goal of feature selection through sparsity regularization. Instead of using the l_2-norm regularizer($\|\boldsymbol{w}\|_2^2$), which results in non-sparse solutions, regularizers such as l_0-norm regularizer($\|\boldsymbol{w}\|_0$) [3] and l_1-norm regularizer($\|\boldsymbol{w}\|_1$) [4] are employed. While the above algorithms are good for enforcing sparsity in the solutions, they are computationally more expensive than l_2 regularization even after convex relaxation. Besides sparse regularization, an effective wrapper feature selection method Recursive Feature Elimination (SVM-RFE) was proposed [5], which recursively trains a classifier, ranks all features according to some criteria and eliminates features that have lowest ranking. While this method has achieved state of the art performance on gene selection, it is computationally expensive and may even be infeasible for high dimensional problem. To address such a problem, a Feature Generating Machine(FGM) method was proposed [6], which iteratively generates a subset of most violated features and solve the resulting Multiple Kernel Learning (MKL) problem. We will base our distributed support vector machine algorithms on the Feature Generating Machine Algorithm in this report.

To deal with the challenges presented by the huge volume of data, many general purpose distributed computing systems such as Spark [7], Dryad [8] and Hadoop [9] were developed. These systems allow programmers to express the application logic in simple coarse-grained primitives like *map* and *reduce*, which simplifies programming for distributed algorithms. But these coarse-grained programming paradigms often do not result in efficient program. For example, a MapReduce program often has to dump the data to the distributed file system, load them into the main memory, parse the data, do the computation, dump the data back to the distributed file system and repeat the process again. As this process involves a lot of unnecessary disk IO, network traffic and serialization/deserialization, the program becomes very inefficient. To address these inefficiencies, programmers often need to resort to more sophisticated domain-specific languages

(DSLs), which raises development cost. This problem motivates the development of Husky, an efficient and expressive distributed computing system designed to strike a better balance between the efficiency and development cost problems mentioned above. There are several features that make Husky attractive as a distributed computing framework. First, Husky is good for interactive data analysis. User can easily perform exploratory data analysis on Husky by using scripting language. Second, Husky is designed to cooperate with the Hadoop ecosystem. This makes Husky attractive because HDFS and Hive are now the dominating big data systems in industry. Third, Husky provides an expressive and user-friendly Application Programming Interface (API). User only needs to understand a few core concepts such as *Object List*, *Channel*, *Aggregator* and *list_execute* to write sophisticated and powerful distributed program. Fourth, Husky is fault-tolerant and easily scalable. Machine failure and growth of computing cluster are handled seamlessly by the runtime "Master" in Husky. Last but not least, Husky can easily outperform computing system that offers coarse-grained primitives and is able to achieve similar or even better performance compared with domain-specific systems[10].

In this report, we propose a distributed version of the FGM algorithm described in [11]. It iteratively generates a pool of violated sparse feature subsets and solves the resulting MKL problem. Adapting the algorithm in [12] to our problem, we are able to divide the original MKL problem into several MKL problems of smaller instances and solve them distributively using a simple and efficient coordinate descent method [13]. The rest of this report is organized as follows. Section 2 gives an introduction to SVM, Sparse SVM and Husky. Section 3 describe our algorithm Distributed Sparse SVM. The last section gives a summary and remark of our algorithm.

2 Background

2.1 Support Vector Machine

Support Vector Machine (SVM) [14] is one of the most often used supervised learning algorithms for pattern recognition. A SVM for binary classification aims to learn a separating hyperplane that maximizes the margin between two classes of data. In a typical machine learning setting, given a labeled training set $\{\boldsymbol{x}_i, y_i\}_{i=1}^n$, where $\boldsymbol{x}_i \in \boldsymbol{R}^m$ is the m-dimensional input and $y_i \in \{+1, -1\}_{i=1}^n$ is the output label, we learn a m-dimensional decision hyperplane $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$ that maximizes the margin $\frac{1}{\|\boldsymbol{w}\|_2^2}$, or equivalently, minimizes the l_2 norm $\|\boldsymbol{w}\|_2^2$ of the normal of the hyperplane

such that all positive data lie on one side of the plane and all the negative data lie on the other side of the hyperplane (as shown in Figure 1). Then the parameter $\boldsymbol{w}$ can be found by solving the following optimization problem.

$$\min_{\boldsymbol{w}} \quad \frac{1}{2}\|\boldsymbol{w}\|_2^2 \qquad\qquad s.t. \quad y_i \boldsymbol{w}^T \boldsymbol{x}_i \geqslant 1, \quad i = 1, \ldots, n \tag{1}$$

The inequality specified above constrains the hyperplane to pass through the origin. In order to solve this problem, we can add a bias term b in the hyperplane by changing the input instance $\boldsymbol{x}$ and parameter $\boldsymbol{w}$ as the following [13]:

$$\boldsymbol{x}_i^T \leftarrow [\boldsymbol{x}_i^T, 1] \qquad\qquad \boldsymbol{w}_i^T \leftarrow [\boldsymbol{w}_i^T, b] \tag{2}$$

However, if the input data are not linearly separable, the constraints in equation (1) can not all be satisfied. To tackle this hard margin problem, slack variables are introduced to loosen the inequality constraints and we can reformulate the problem as a soft margin maximization problem:

$$\min_{\boldsymbol{w}} \quad \frac{1}{2}\|\boldsymbol{w}\|_2^2 + \frac{C}{2}\sum_{i=1}^{n} \xi_i^2 \qquad\qquad s.t. \quad y_i \boldsymbol{w}^T \boldsymbol{x}_i \geqslant 1 - \xi_i, \quad i = 1, \ldots, n \tag{3}$$

With ξ_i, we allow an input instance $\boldsymbol{x}_i$ to violate the original hard margin constraint but penalize such violations by adding a positive term $\frac{C}{2}\xi_i^2$ to the objective function. The parameter C is called the slack trade off. A smaller C allows more points to be misclassified and a greater C forces the margin to become "harder". The loss function we use here is called square hinge loss, taking the form $loss(\boldsymbol{w}, \boldsymbol{x}_i, y_i) = max(1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i, 0)^2$. Another often used loss function in training a SVM is hinge loss, which is in the form $loss(\boldsymbol{w}, \boldsymbol{x}_i, y_i) = max(1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i, 0)$. These two functions are used because they are both convex functions, which is a desired property in an optimization problem. In addition to being a linear classifier, SVM can also be used to model nonlinear relationship by using the kernel trick. The idea stems from the fact that when formulated in its dual form (see the next subsection), the objective function can be written as a linear combination of the dot product $\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$, which can be viewed as the similarity between instance $\boldsymbol{x}_i$ and instance $\boldsymbol{x}_j$ [15]. Therefore, we can replace $\boldsymbol{x}_i^T \boldsymbol{x}_j$ with a Kernel function $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$, where $\phi : \mathcal{X} \rightarrow \mathcal{Z}$ is a mapping from feature space X onto another feature space Z. Then a linear SVM corresponds to the Kernel $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^T \boldsymbol{x}_j$. One commonly used Kernel is the Gaussian Kernel $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = exp(-\gamma \|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2)$. Since the Taylor expansion of a function allows us to indefinitely

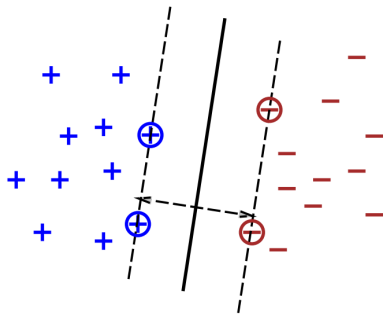


Figure 1: Figure 1. The solid line in the middle is the hyperplane that separates the positive(+) data points from the negative(-) data points. The circled points that lie on the dashed line are called support vectors.

approximate a continuous function by adding high order polynomial term, with each polynomial representing a new dimension, the Gaussian Kernel allows us to map our our data from the original n-dimensional feature space onto a feature space of infinite dimension without explicitly calculating the dot product in the infinite-dimensional space. One important consequence of this mapping is that if the input data are not linearly separable in its original feature space, we can always map it onto a feature space of higher dimension to make it separable by a hyperplane in the higher dimensional feature space. But consideration must be given to input data with noisy features, in which case such a method may not generalize well.

2.2 Sparse SVM

2.2.1 Primal and Dual formulation of Sparse SVM

Following the notation in [11], we denote $A \odot B$ to be the elementwise product between two vectors/matrices A and B. To obtain a sparse parameter $\boldsymbol{w}$ of SVM, a 0-1 control vector $\boldsymbol{d} = [d_1, \cdots, d_m]^T \in \mathcal{D}$ is introduced to control which features get selected. The decision hyperplane thus becomes: $f(x) = \boldsymbol{w}^T x = (\tilde{\boldsymbol{w}} \odot \boldsymbol{d})^T x = \tilde{\boldsymbol{w}}^T (\boldsymbol{x} \odot \boldsymbol{d})$, where $\mathcal{D} = \{\boldsymbol{d} | \sum_{j=1}^{m} \leqslant B, d_j \in \{0,1\}, j = 1, \cdots, m\}$ is the domain of $\boldsymbol{d}$, and B controls the sparsity of $\boldsymbol{d}$. Using square hinge loss, the objective function of Sparse SVM can be formulated as:

$$\min_{\boldsymbol{d} \in \mathcal{D}} \min_{\tilde{\boldsymbol{w}}, \boldsymbol{\xi}} \quad \frac{1}{2} \|\tilde{\boldsymbol{w}}\|_2^2 + \frac{C}{2} \sum_{i=1}^{n} \xi_i^2 \qquad s.t. \quad y_i \tilde{\boldsymbol{w}}^T (\boldsymbol{x}_i \odot \boldsymbol{d}) \geqslant 1 - \xi_i, i = 1, \cdots, n \tag{4}$$

To derive its dual form, we introduce one Lagrange multiplier for each constraint. Let $\mathcal{L}(\tilde{\boldsymbol{w}}, \boldsymbol{\xi}, \boldsymbol{\alpha}) = \frac{1}{2} \|\tilde{\boldsymbol{w}}\|_2^2 + \frac{C}{2} \sum_{i=1}^{n} \xi_i^2 - \sum_{i=1}^{n} \alpha_i (y_i \tilde{\boldsymbol{w}}^T (\boldsymbol{x}_i \odot \boldsymbol{d}) - 1 + \xi_i)$, where $\boldsymbol{\alpha} = [\alpha_1, \cdots, \alpha_n]^T$ is a vector of dual

variables for the inequality constraints and $\boldsymbol{\alpha} \in \mathcal{A}, \mathcal{A} = \{\boldsymbol{\alpha} | \alpha_i \geqslant 0, i = 1, \cdots, n\}$, it can be seen that,

$$\max_{\boldsymbol{\alpha} \in \mathcal{A}} \mathcal{L}(\tilde{\boldsymbol{w}}, \boldsymbol{\xi}, \boldsymbol{\alpha}) = \begin{cases} \infty & \text{if some constraints are violated} \\ \frac{1}{2}\|\tilde{\boldsymbol{w}}\|_2^2 + \frac{C}{2}\sum_{i=1}^n \xi_i^2 & \text{otherwise} \end{cases}$$

If a parameter candidate $\tilde{w}$ violates some constraints, then $\mathcal{L}(\tilde{\boldsymbol{w}}, \boldsymbol{\xi}, \boldsymbol{\alpha}) = \infty$ and such a solution will not be chosen when we want to minimize over $\tilde{w}$. Hence (4) is equivalent to

$$\min_{\boldsymbol{d} \in \mathcal{D}} \min_{\tilde{\boldsymbol{w}}, \xi} \max_{\boldsymbol{\alpha} \in \mathcal{A}} \quad \frac{1}{2}\|\tilde{\boldsymbol{w}}\|_2^2 + \frac{C}{2}\sum_{i=1}^n \xi_i^2 - \sum_{i=1}^n \alpha_i (y_i \tilde{\boldsymbol{w}}^T (\boldsymbol{x}_i \odot \boldsymbol{d}) - 1 + \xi_i)$$

Since the maximum of the minimum of a function is always less than or equal to the minimum of the maximum of the same function, we have

$$\max_{\boldsymbol{\alpha} \in \mathcal{A}} \min_{\tilde{\boldsymbol{w}}, \xi} \frac{1}{2}\|\tilde{\boldsymbol{w}}\|_2^2 + \frac{C}{2}\sum_{i=1}^n \xi_i^2 - \sum_{i=1}^n \alpha_i (y_i \tilde{\boldsymbol{w}}^T (\boldsymbol{x}_i \odot \boldsymbol{d}) - 1 + \xi_i) \leqslant \min_{\tilde{\boldsymbol{w}}, \xi} \max_{\boldsymbol{\alpha} \in \mathcal{A}} \frac{1}{2}\|\tilde{\boldsymbol{w}}\|_2^2 + \frac{C}{2}\sum_{i=1}^n \xi_i^2 - \sum_{i=1}^n \alpha_i (y_i \tilde{\boldsymbol{w}}^T (\boldsymbol{x}_i \odot \boldsymbol{d}) - 1 + \xi_i)$$

According to Lagrange Duality theory, the above inequality satisfies with equality and we can solve (4) by solving the following:

$$\min_{\boldsymbol{d} \in \mathcal{D}} \max_{\boldsymbol{\alpha} \in \mathcal{A}} \min_{\tilde{\boldsymbol{w}}, \xi} \quad \frac{1}{2}\|\tilde{\boldsymbol{w}}\|_2^2 + \frac{C}{2}\sum_{i=1}^n \xi_i^2 - \sum_{i=1}^n \alpha_i (y_i \tilde{\boldsymbol{w}}^T (\boldsymbol{x}_i \odot \boldsymbol{d}) - 1 + \xi_i) \tag{5}$$

Taking the above objective function with respect to $\tilde{\boldsymbol{w}}$ and ξ, setting them to 0, we have

$$\tilde{\boldsymbol{w}} = \sum_{i=1}^n \alpha_i y_i (\boldsymbol{x}_i \odot \boldsymbol{d}) \qquad \alpha_i = C\xi_i, i = 1, \cdots, n$$

Substitute the above back to (5), we get the dual form of the SSVM problem as follows. Notice how the objective function is expressed as a linear combination of the dot product $\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$ as mentioned in section 2.1.

$$\min_{\boldsymbol{d} \in \mathcal{D}} \max_{\boldsymbol{\alpha} \in \mathcal{A}} \quad -\frac{1}{2}\|\sum_{i=1}^n \alpha_i y_i (\boldsymbol{x}_i \odot \boldsymbol{d})\|^2 - \frac{1}{2C}\boldsymbol{\alpha}^T \boldsymbol{\alpha} + \boldsymbol{e}^T \boldsymbol{\alpha} \tag{6}$$

According to the convex relaxation in [11], (6) can be written as a MKL problem

$$\max_{\boldsymbol{\mu} \in \mathcal{M}} \min_{\boldsymbol{\alpha} \in \mathcal{A}} \quad f^{\mathcal{D}}(\boldsymbol{\alpha}, \boldsymbol{\mu}) = \frac{1}{2}\boldsymbol{\alpha}^T Y (\sum_{d^t \in \mathcal{D}} \mu_t X_t X_t^T + \frac{1}{C} I) Y \alpha - e^T \boldsymbol{\alpha} \tag{7}$$

Or equivalently,

$$\max_{\boldsymbol{\mu}\in\mathcal{M}} \min_{\boldsymbol{\alpha}\in\mathcal{A}} \quad f^{\mathcal{D}}(\boldsymbol{\alpha},\boldsymbol{\mu}) = \frac{1}{2}\boldsymbol{\alpha}^T\bar{Q}\boldsymbol{\alpha} - e^T\boldsymbol{\alpha}$$

where $\mathcal{M} = \{\boldsymbol{\mu}|\boldsymbol{e}^T\boldsymbol{\mu} = 1, \boldsymbol{\mu} \geqslant 0\}$, $\bar{Q} = Y(\sum_{d^t\in\mathcal{D}} \mu_t X_t X_t^T + \frac{1}{C}I)Y$, $X_t = [\boldsymbol{x_1} \odot \boldsymbol{d^t}, \cdots, \boldsymbol{x_n} \odot \boldsymbol{d^t}]^T$ and Y is a diagonal matrix such that $Y_{ii} = y_i$.

2.2.2 Cutting Plane Algorithm

The overall algorithm for solving (7) is described in Algorithm 1 [11]. We denote the subset of constraints by $\mathcal{C} \subset \mathcal{D}$. First the vector of dual variables $\boldsymbol{\alpha}$ is initialized to $\frac{1}{n}\mathbf{1}$. We then find the most violated $\hat{\boldsymbol{d}} \in \mathcal{D}$ and initialize the working set $\mathcal{C} = \{\hat{\boldsymbol{d}}\}$. To solve the MKL problem, we use an efficient method called SimpleMKL[16]. We fix $\boldsymbol{\mu}$ and run SVM solver to update $\boldsymbol{\alpha}$ one step. Then we fix $\boldsymbol{\alpha}$ and update $\boldsymbol{\mu}$ one step. This iterative updating procedure of $\boldsymbol{\alpha}$ and $\boldsymbol{\mu}$ is repeated until the MKL problem converges. Then the next most violated constraint $\boldsymbol{d}$ is calculated and added to $\mathcal{C}$. This process is repeated until the termination criterion is met. To find the most violated constraint, the constraint that has the highest feature score given the current values of $\boldsymbol{\alpha}$, we denote $c_j = \sum_{i=1}^n \alpha_i y_i x_{ij}$ and sort the c_js in descending order and set the first B numbers corresponding to $\boldsymbol{d_j}$ to 1 and the rests to 0.

2.2.3 Prediction

When the algorithm converges, the decision function can be obtained by $f(x) = \sum_{\boldsymbol{d^t}\in\mathcal{D}} \sum_{i=1}^n \alpha_i y_i (\boldsymbol{x}_i \odot \boldsymbol{d^t})^T \boldsymbol{x} = \sum_{i=1}^n \alpha_i y_i (\boldsymbol{x}_i \odot \tilde{\boldsymbol{d}})^T x$, where $\tilde{\boldsymbol{d}} = \sum_{\boldsymbol{d^t}\in\mathcal{D}} \mu_t \boldsymbol{d^t}$

Algorithm 1 The cutting plane algorithm for FGM

1: Initialize $\boldsymbol{\alpha} = \frac{1}{n}\mathbf{1}$. Find the most violated $\hat{\boldsymbol{d}}$, and set $\mathcal{C} = \{\hat{\boldsymbol{d}}\}$
2: Run MKL solver to solve for $\boldsymbol{\alpha}$ and $\boldsymbol{\mu}$
3: Find the most violated $\hat{\boldsymbol{d}}$ and set $\mathcal{C} = \mathcal{C} \cup \hat{\boldsymbol{d}}$
4: Repeat step 2-4 until convergence

2.3 Simple MKL

To solve Multiple Kernel Learning probelm like problem7, an efficient algorithm called SimpleMKL [16] has been proposed. Formally, the Multiple Kernel Learning problem corresponds to the

following primal optimization problem:

$$\begin{aligned}
\underset{f_m,b,\xi,\boldsymbol{d}}{\text{minimize}} \quad & \frac{1}{2}\sum_m \frac{1}{d_m}||f_m||^2_{\mathcal{H}_m} + C\sum_{i=1}^{n}\xi_i \\
\text{subject to} \quad & y_i(\sum_m f_m(x_i) + b) \geq 1 - \xi_i, \quad \forall i \\
& \xi_i \geq 0 \quad \forall i \\
& \sum_m d_m = 1, \quad d_m \geq 0, \quad \forall m
\end{aligned}$$

where $\mathcal{H}_m$ defines a space that is associated with a particular kernel K_m and d_m can be understood to be the weight of a particular kernel in the decision function. To solve the above, a constrained optimization problem is proposed:

$$\underset{\boldsymbol{d}}{\text{minimize}} \quad J(\boldsymbol{d}) \quad \text{subject to} \quad \sum_{m=1}^{M} d_m = 1, d_m \geq 0 \quad m = 1, \cdots, M \tag{8}$$

where $J(\boldsymbol{d})$ is the optimal value of the following SVM problem

$$\begin{aligned}
\underset{f_m,b,\xi}{\text{minimize}} \quad & \frac{1}{2}\sum_m \frac{1}{d_m}||f_m||^2_{\mathcal{H}_\Updownarrow} + C\sum_{i=1}^{n}\xi_i \\
\text{subject to} \quad & y_i(\sum_m f_m(\boldsymbol{x}_i) + b) \geq 1 - \xi_i, \quad \forall i \\
& \xi_i \geq 0 \quad \forall i
\end{aligned} \tag{9}$$

Thus the above can be solved using the following iterative procedure. First a SVM solver is invoked to solve the SVM problem (9), then reduced gradient method is used to find a feasible descent direction for $\boldsymbol{d}$, followed by a line search for the optimal step size. The above iterative process is repeated until the duality gap of the Multiple Kernel Learning problem is within a tolerable threshold. Note in order to ensure the equality constraint $\sum_{m=1}^{M} d_m = 1$ and the positivity constraint $d_m \geq 0$, reduced gradient method is used instead of a normal gradient method. In particular, the reduced gradient of $J(\boldsymbol{d})$ denoted by $\nabla_{red}J$ has the following components:

$$[\nabla_{red}J]_m = (\frac{\partial J}{\partial d_m} - \frac{\partial J}{d_u}) \quad \forall m \neq u$$

$$[\nabla_{red}J]_u = \sum_{m \neq u}(-\frac{\partial J}{\partial d_m} + \frac{\partial J}{\partial d_u})$$

where u can be any one of $1, \cdots, M$. Since $\gamma\sum_{m\neq u}\frac{\partial J}{\partial d_m} - \frac{\partial J}{\partial d_u} + \gamma\sum_{m\neq u} -\frac{\partial J}{\partial d_m} + \frac{\partial J}{\partial d_u} = 0$ for all possible step size γ, the equality constraint is satisfied. To ensure that the positivity constraint is

preserved, a feasible descent direction D can be found by projecting the reduced gradient to the feasible domain:

$$D_m = \begin{cases} 0, & \text{if } d_m = 0 \text{ and} \frac{\partial J}{\partial d_m} - \frac{\partial J}{\partial d_u} \geq 0 \\ -\frac{\partial J}{\partial d_m} + \frac{\partial J}{\partial d_u}, & \text{if } d_m > 0 \text{and } m \neq u \\ \sum_{v \neq u, d_v \geq 0} (\frac{\partial J}{\partial d_v} - \frac{\partial J}{\partial d_u}) & \text{if } m = u \end{cases} \tag{10}$$

The above simply says that if an entry d_m is equal to 0 and its reduced gradient is non-negative (negative of reduced gradient is positive), then we need to project its descent direction to be 0 indicating that it is already at its optimum. Otherwise we set its descent direction to be its reduced gradient. With the descent direction available, one can perform a simple step size search to find the optimal step size and update $\boldsymbol{d}$. A detailed algorithm adapted to the setting of this report will be shown in section 3.

2.4 Core Concepts in Husky

Husky follows a master-slave architecture. One Husky cluster is made up of a master, which is responsible for load balancing and coordination between workers, and multiple workers which can reside in one or multiple machines. Below is a description of the four most important concepts in Husky.

Object List is the primary data abstraction in Husky, where user-defined objects can be stored. In a distributed setting, *Object List* is a distributed abstraction of objects. When writing the program, users can treat the *Object List* as a single array of object when in fact it is distributed among different workers, with each worker storing a portion of the list. This is a very useful abstraction as users can write a distributed program as if he/she is writing a single machine program.

Channel is another important concept in Husky. It defines how *Object List*s communicate with each other. Messages, which can be any user-defined data types, are sent through channels to a distant location using the *push* function and are received through channels by a distant target using the *get* function.

Aggregator is a powerful *allreduce* like object designed for easily aggregating values from executors and broadcasting the aggregated values back to the executors.

list_execute is the most important function provided by Husky. Synchronization of the above three concepts across different workers are done through *list_execute*. Consider the following C++ example

```
for (int i = 0; i < objList.size(); i++)
std::cout << objList[i].id() << std::endl;
```

The piece of code above can be easily expressed in a distributed setting in Husky as follows.

```
// objects in objList reside in different workers
list_execute(objList, [](Obj& obj) {
log_msg(obj.id());
});
```

As can be seen, even though the objects are distributed among different workers, we can use *list_execute* to easily access the objects as if we are writing a simple *for-loop*. And most importantly, we have fine grained control at the level of objects instead of a coarse grained control at the level of tables like that provided by Spark[17] and Hive[18].

3 Distributed Feature Generating Machine

In this section, we present a distributed implementation framework for the Sparse SVM algorithm. Building on the communication pattern of Husky, we carefully investigate the mathematical relationship between different parameters and design a scheme that can coordinate different workers to jointly solve the Sparse SVM problem with low communication cost. We start by distributing the input data $\{\boldsymbol{x}_i, y_i\}_{i=1}^n$ among K workers. To make it clear, we introduce the following notation: Instances in machine k are denoted as $\{(\boldsymbol{x}_i, y_i)\}_{i\in J_k}$ where J_k are disjoint sets such that $\cup_{k=1}^K J_k = \{1, \cdots, n\}$. $\mathcal{P}(\boldsymbol{\alpha}) \equiv [-\alpha_1, \infty] \times \cdots \times [-\alpha_n, \infty]$ is the feasible domain of $\boldsymbol{\alpha}$. $\|\boldsymbol{u}\|_A^2 \equiv \boldsymbol{u}^T A \boldsymbol{u}$ for $\boldsymbol{u} \in R^t$ and $A \in R^{t\times t}$. $\boldsymbol{v}_{J_k}$ denotes the sub-vector of $\boldsymbol{v}$ that contains the coordinates in J_k and similarly for A_{J_k,J_m}.

3.1 Finding the Most Violated Constraints

Finding the most violated constraints involves finding the constraint $\hat{\boldsymbol{d}}$ that yields the largest objective value in (6), which can be expressed as a function of $\hat{\boldsymbol{d}}$: $\frac{1}{2}\sum_{j=1}^m(\sum_{i=1}^n \alpha_i y_i x_{ij})^2 d_j =$

$\frac{1}{2}\sum_{j=1}^{m} c_j^2 d_j$, where $c_j = \sum_{i=1}^{n} \alpha_i y_i x_{ij}$. Hence the most violated constraint corresponds to the set of d_js that has the B-largest c_js. We can calculate c_j as

$$c_j = \bigoplus_{k=1}^{K} X_{J_k,j}^T Y_{J_k} \boldsymbol{\alpha}_{J_k} \tag{11}$$

where $X_{J_k,j}$ stands for the vector obtained by taking the (i, j) entry of the input matrix X with $i \in J_k$. The symbol $\bigoplus$ represent the operation of receiving the information from all workers and broadcasting the results back to all workers. In Husky, this can be done using *Aggregator*, which implements the *allreduce* functionality of broadcasting the values to all the workers and collecting the aggregated values from all the workers. Hence, each worker will update c_j using the input data available in its local storage and receive a global version of c_j. With the availability of c_j, each worker will sort the c_j in descending order and set the first B numbers corresponding to d_j to 1 and the rests to 0. Then the new generated $\hat{\boldsymbol{d}}$ is added to the constraint set $\mathcal{C}$. Notice because each machine has the same copy of c_j, the resulting $\hat{\boldsymbol{d}}$ is also the same for all workers, which obviates the need for a centralized master to send the new $\hat{\boldsymbol{d}}$ to every other workers.

3.2 Solving the MKL Problem

As mentioned in section 2, we use SimpleMKL to solve the MKL problem (7). A detailed description of the algorithm is given in Algorithm 2. where $K_t(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i \odot \boldsymbol{d}_t)^T (\boldsymbol{x}_j \odot \boldsymbol{d}_t)$ is the kernel

Algorithm 2 The SimpleMKL algorithm

Set $\mu_t = \frac{1}{T}$ for $T = 1, \cdots, T$
while stopping criterion not met do
    Compute $J(\boldsymbol{\mu})$ using a SVM solver with kernel $K = \sum_t \mu_t K_t$
    Compute $\frac{\partial J}{\partial \mu_t} for \mu = 1, \cdots, T$ and descent direction D
    Set $u = \underset{t}{\text{argmax}}\ \mu_t$, $J' = 0$, $\boldsymbol{\mu}' = \boldsymbol{\mu}$, $\boldsymbol{D}' = \boldsymbol{D}$ while $J' < J(\boldsymbol{\mu})$ do
        $\boldsymbol{\mu} = \boldsymbol{\mu}'$, $\boldsymbol{D} = \boldsymbol{D}'$
        $v = \underset{\{t|D_t<0\}}{\text{argmin}} -\frac{\mu_t}{D_t}$, $\gamma_{max} = -\frac{\mu_v}{D_v}$
        $\boldsymbol{\mu}' = \boldsymbol{\mu} + \gamma_{max}\boldsymbol{D}$, $D'_u = D_u + D_v$, $D'_v = 0$ // normalization
        Compute J' using a SVM with kernel $K = \sum_t \mu_t K_t$
    end while
    line search along $\boldsymbol{D}$ for $\gamma \in [0, \gamma_{max}]$
    $\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} + \gamma \boldsymbol{D}$
end while

corresponding to the t-th control variable. As described in the SimpleMKL algorithm, we need a SVM solver to solve for $\boldsymbol{\alpha}$. In the following sections, we describe a distributed SVM solver called Box constrained Quadratic Optimization algorithm[12], a single machine SVM solver called Dual

Coordinate Descent algorithm[13] and adapt these two algorithms to our setting.

3.2.1 Updating $\boldsymbol{\alpha}$

The main computations in solving (7) are the computation of the Kernel Matrix $\sum_{t=1}^{T} \mu_t X_t X_t^T$, which is easily computed in a single machine setting since all input instances are easily available. However, in a distributed setting, computing the exact Kernel matrix is not easy. One naive approach is that each worker can have in their local storage all the input instances $\{\boldsymbol{x}_i, y_i\}_{i=1}^n$. Another approach is that all machines receive the necessary data from other machines by means of communication. It is easily seen that the first method will hit a bottle neck when the volume of the input data gets larger and larger and the second method will incur too much communication cost. Therefore, we consider an approximation of the Kernel matrix $\sum_{t=1}^{T} \mu_t X_t X_t^T$ in each local machine using only the input data available locally.

Computing the Update direction: The algorithm starts with an initial point $\boldsymbol{\alpha}^0$ for (7) and iteratively generates a sequence of solutions $\{\boldsymbol{\alpha^l}\}_{l=0}^{\infty}$ until convergence. At the l-th iteration, we update the current $\boldsymbol{\alpha^l}$ by

$$\boldsymbol{\alpha^{l+1}} = \boldsymbol{\alpha^l} + \eta^l \Delta \boldsymbol{\alpha^l} \tag{12}$$

where $\eta^l \in R$ is the step size and $\Delta\boldsymbol{\alpha}^l \in R^n$ is the update direction. To compute $\Delta\boldsymbol{\alpha^l}$, let $\boldsymbol{d} \in \mathcal{P}(\boldsymbol{\alpha^l})$ be any direction in the n-dimensional space ($\boldsymbol{d}$ not to be confused with the control variable), according to Taylor series expansion, we have with $\boldsymbol{\mu}$ fixed,

$$f^{\mathcal{D}}(\boldsymbol{\alpha^l} + \boldsymbol{d}, \boldsymbol{\mu}) = f^{\mathcal{D}}(\boldsymbol{\alpha^l}, \boldsymbol{\mu}) + \nabla f^{\mathcal{D}}(\boldsymbol{\alpha^l}, \boldsymbol{\mu})^T \boldsymbol{d} + \frac{1}{2} \boldsymbol{d}^T H \boldsymbol{d} \tag{13}$$

where $H = \tilde{Q} + \frac{1}{C} I$ and

$$\tilde{Q}_{i,j} = \begin{cases} Q_{i,j} & \text{if } i \in J_k \text{ and } j \in J_k \text{ for some } k \\ 0 & \text{otherwise} \end{cases}$$

where $Q = Y(\sum_{d^t \in \mathcal{D}} \mu_t X_t X_t^T) Y$ and H is a symmetric, block-diagonal matrix with K blocks, where the k-th block is

$$H_{J_k} = Y_{J_k} (\sum_{t=1}^{T} \mu_t X_{t,J_k} X_{t,J_k}^T + \frac{1}{C} I) Y_{J_k}$$

Then in order to minimize $f^{\mathcal{D}}(\boldsymbol{\alpha^l}, \boldsymbol{\mu})$, we find an update direction $\boldsymbol{d}$ such that (13) is minimized,

which results in the following problem

$$\Delta\boldsymbol{\alpha}^{\boldsymbol{l}} = \underset{\boldsymbol{d}\in\mathcal{P}(\boldsymbol{\alpha}^{\boldsymbol{l}})}{\operatorname{argmin}} \nabla f^{\mathcal{D}}(\boldsymbol{\alpha}^{\boldsymbol{l}}, \boldsymbol{\mu})^T \boldsymbol{d} + \frac{1}{2}\boldsymbol{d}^T H \boldsymbol{d} \tag{14}$$

Since each machine have some input instances available in their local storage, they each can solve a part of $\Delta\boldsymbol{\alpha}^{\boldsymbol{l}}$ locally

$$\Delta\boldsymbol{\alpha}^{\boldsymbol{l}}_{J_k} = \underset{\boldsymbol{d}_{J_k}\in\mathcal{P}(\boldsymbol{\alpha}^{\boldsymbol{l}})_{J_k}}{\operatorname{argmin}} \nabla f^{\mathcal{D}}(\boldsymbol{\alpha}^{\boldsymbol{l}}, \boldsymbol{\mu})^T_{J_k} \boldsymbol{d}_{J_k} + \frac{1}{2}\|\boldsymbol{d}_{J_k}\|^2_{H_{J_k}} \tag{15}$$

We can calculate $\nabla f^{\mathcal{D}}(\boldsymbol{\alpha}^{\boldsymbol{l}}, \boldsymbol{\mu})_{J_k}$ as follows

$$\nabla f^{\mathcal{D}}(\boldsymbol{\alpha}^{\boldsymbol{l}}, \boldsymbol{\mu})_{J_k} = Y_{J_k} X_{J_k} \boldsymbol{w}^l + \frac{1}{C}\boldsymbol{\alpha}^l_{J_k} - \boldsymbol{e}$$

where $\boldsymbol{w}^l$ can be obtained by gathering information from all workers.

$$\boldsymbol{w}^l = \bigoplus_{k=1}^{K} \sum_{\boldsymbol{d}^t\in\mathcal{D}} \mu_t X^T_{t,J_k} Y_{J_k} \boldsymbol{\alpha}^l_{J_k} \tag{16}$$

or alternatively,

$$\nabla f^{\mathcal{D}}(\boldsymbol{\alpha}^{\boldsymbol{l}}, \boldsymbol{\mu})_{J_k} = \sum_{t=1}^{T} \mu_t Y_{J_k} X_{t,J_k} \boldsymbol{w}_t + \frac{1}{C}\boldsymbol{\alpha}^l_{J_k} - \boldsymbol{e}$$

$$\boldsymbol{w}^l = \sum_{t=1}^{T} \mu_t \boldsymbol{w}^l_t \tag{17}$$

where $\boldsymbol{w}_t = \sum_{k=1}^{K} X^T_{t,J_k} Y_{J_k} \boldsymbol{\alpha}^l_{J_k}$ is the weight vector associated with the t-th kernel. In this report, we will take the second approach of computing $\boldsymbol{w}^l$. In fact this is a key step of reducing the communication cost and we will discuss the rationale for choosing this method later in the end of section 3.

As with (11), $\boldsymbol{w}^l$ can also be maintained in local storage of each worker using *Aggregator*. Notice with $\nabla f^{\mathcal{D}}(\boldsymbol{\alpha}^{\boldsymbol{l}}, \boldsymbol{\mu})_{J_k}$ and H_{J_k} available, (11) is in the same form as a standard SVM dual problem, with $\boldsymbol{\alpha}$, $\bar{Q}$, $-\boldsymbol{e}$ and $\mathcal{P}(0)$ replaced by $\boldsymbol{d}_{\boldsymbol{J_k}}$, H_{J_k}, $\nabla f^{\mathcal{D}}(\boldsymbol{\alpha}^{\boldsymbol{l}}, \boldsymbol{\mu})_{J_k}$ and $\mathcal{P}(\boldsymbol{\alpha}^l)_{J_k}$ respectively. We describe the dual coordinate descent method for solving (15) in the below.

Algorithm for solving (15) locally: For SVM problem in which the bias term b is not appended at the end of the decision hyperplane parameter $\boldsymbol{w}$ as in equation (2), a linear constraint $\sum_{i=1}^{n} \alpha_i y_i = 0$ will arise when deriving the dual form. This linear constraint forces any optimization method for the dual problem to update at least two variables at a time. For such kind of

problem, the Sequential Minimal Optimization (SMO) method [19] is an often-used solution. But in our case since we we do not have this linear constraint, we are able to optimize α one variable at a time, which leads us to a simple dual coordinate descent method [13]. For ease of notation, we rewrite (15) as the following

$$\min_{\boldsymbol{\gamma}\in\mathcal{F}} h(\boldsymbol{\gamma}) = \boldsymbol{\beta}^T\boldsymbol{\gamma} + \frac{1}{2}\boldsymbol{\gamma}^T A\boldsymbol{\gamma} \tag{18}$$

where $\boldsymbol{d}_{J_k}$, $\mathcal{P}(\boldsymbol{\alpha}^l)_{J_k}$, $\nabla f^{\mathcal{D}}(\boldsymbol{\alpha}^{\boldsymbol{l}}, \boldsymbol{\mu})_{J_k}$ and H_{J_k} are replaced with $\boldsymbol{\gamma}$, $\mathcal{F}$, $\boldsymbol{\beta}$ and A respectively. Since we are dealing with SVM with square hinge loss, $\mathcal{P}(\boldsymbol{\alpha}^l)_{J_k}$ only has lower bound. Let $\mathcal{F}_i = -\alpha_i, i = 1, \cdots, |J_k|$ be the lower bound of the i-th α in J_k. Because we do not have the equality constraint, we can update $\boldsymbol{\alpha}$ one variable at a time, suppose we are updating the i-th α, (18) becomes

$$\min_d h(\boldsymbol{\gamma} + de_i) \equiv \nabla_i h(\boldsymbol{\gamma})d + \frac{1}{2}A_{ii}d^2 + const \tag{19}$$

which is a quadratic function of d, where d is a scalar, $\nabla_i h$ is the i-th component of the gradient ∇h. Since the variable $\boldsymbol{\gamma}$ is constrained by a feasible set $\mathcal{F}$ with lower bound, we need to project the gradient to a feasible direction when optimizing $\boldsymbol{\gamma}$. We only need to consider two cases: When $\gamma_i = \mathcal{F}_i$ and $\nabla_i h(\gamma) > 0$, we can not further decrease γ_i and have to clip $\nabla_i h(\gamma)$ to 0. When $\gamma_i = \mathcal{F}_i$ and $\nabla_i h(\gamma) \leqslant 0$, we can move in the negative direction of the gradient to decrease (19). Denote the i-th component of the projected gradient as $\nabla_i^P h(\gamma)$, we have

$$\nabla_i^P h(\gamma) = \begin{cases} \nabla_i h(\gamma) & \text{if } \mathcal{F}_i < \gamma_i \\ \min(0, \nabla_i h(\gamma)) & \text{if } \mathcal{F}_i = \gamma_i \end{cases}$$

where $\nabla_i h(\boldsymbol{\gamma})$ can be computed as

$$\nabla_i h(\boldsymbol{\gamma}) = (A\boldsymbol{\gamma})_i + \beta_i$$

or equivalently,

$$\nabla_i h(\gamma) = y_i \boldsymbol{\kappa}^T \boldsymbol{x}_i + \beta_i + \frac{1}{C}\gamma_i = y_i \sum_{t=1}^{T} \boldsymbol{\kappa}_t^T \boldsymbol{x}_{t,i} + \beta_i + \frac{1}{C}\gamma_i \tag{20}$$

where

$$\boldsymbol{\kappa} = \sum_{t=1}^{T} \mu_t X_{t,J_k}^T Y_{J_k} \boldsymbol{\gamma}, \qquad \boldsymbol{\kappa}_t = \mu_t X_{t,J_k}^T Y_{J_k} \boldsymbol{\gamma}, \qquad \boldsymbol{x}_{t,i} = \boldsymbol{x}_i \odot \boldsymbol{d}_t$$

The above calculation of $\nabla_i h(\boldsymbol{\gamma})$ using (20) is only valid in linear SVM where the kernel $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is simply the dot product $\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$. By computing $\nabla_i h(\boldsymbol{\gamma})$ in this way and caching $\boldsymbol{\kappa}_t$, we have

avoided storing the dense kernel matrix A in the memory, which is more practical. If the projected gradient is 0, then (19) is already at its optimum. If not, we need to update γ_i and clip it to the feasible set. The algorithm is described in Algorithm 2. The general idea is that while $\boldsymbol{\gamma}$ is not optimal, we loop over all its coordinate entries and try to optimize them with respect to the objective function.

Algorithm 3 Dual Coordinate Descent Method

1: Given the initial value and feasible set of γ, compute $\boldsymbol{\kappa}_t = \mu_t X_{t,J_k}^T Y_{J_k} \boldsymbol{\gamma}$, $t = 1, \cdots, T$
2: For $i = 1, \cdots, |\boldsymbol{\gamma}|$
3: $\quad \nabla_i h(\boldsymbol{\gamma}) = y_i \sum_{t=1}^T \boldsymbol{\kappa}_t^T \boldsymbol{x}_{t,i} + \beta_i + \frac{1}{C}\gamma_i$
4: $\quad PG \leftarrow \nabla_i^P h(\boldsymbol{\gamma})$
5: $\quad$ If $PG \neq 0$
6: $\quad\quad \bar{\gamma_i} \leftarrow \gamma_i$
7: $\quad\quad \gamma_i \leftarrow \max(\gamma_i - \frac{\nabla_i h(\boldsymbol{\gamma})}{A_{ii}}, \mathcal{F}_i)$
8: $\quad\quad \boldsymbol{\kappa}_t \leftarrow \boldsymbol{\kappa}_t + \sum_{t=1}^T \mu_t(\gamma_i - \bar{\gamma_i}) y_i \boldsymbol{x}_{t,i}$, $t = 1, \cdots, T$
9: While $\boldsymbol{\gamma}$ is not optimal, go to step 2

Computing the Step Size: According to [12], the step size η^l can be computed as the following

$$\eta^l = \min((\eta^l)^\star, \min_{1 \leqslant i \leqslant n} \lambda_i) \tag{21}$$

where

$$(\eta^l)^\star = \frac{-\nabla f^{\mathcal{D}}(\boldsymbol{\alpha}^l, \boldsymbol{\mu}) \Delta \boldsymbol{\alpha}^l}{(\Delta \boldsymbol{\alpha}^l) \bar{Q} \Delta \boldsymbol{\alpha}^l}$$

$$\lambda_i = \begin{cases} \frac{-\alpha_i^l}{\Delta \alpha_i^l} & \text{if } \Delta\alpha_i^l < 0, \\ \infty & \text{if } \Delta\alpha_i^l \geqslant 0. \end{cases}$$

The parameters needed for computing η^l can be computed from

$$\Delta \boldsymbol{w}_t^l = \bigoplus_{k=1}^K X_{t,J_k}^T Y_{J_k} \Delta \boldsymbol{\alpha}_{J_k}^l, t = 1, \cdots, T$$

$$-\nabla f^{\mathcal{D}}(\boldsymbol{\alpha}^l, \boldsymbol{\mu}) \Delta \boldsymbol{\alpha}^l = \sum_{t=1}^T (\boldsymbol{w}_t^l)^T \Delta \boldsymbol{w}_t^l + \frac{1}{C} \bigoplus_{k=1}^K (\boldsymbol{\alpha}_{J_k}^l)^T \Delta \boldsymbol{\alpha}_{J_k}^l - \bigoplus_{k=1}^K \boldsymbol{e}^T \Delta \boldsymbol{\alpha}_{J_k}^l$$

$$(\Delta \boldsymbol{\alpha}^l)^T \bar{Q} \Delta \boldsymbol{\alpha}^l = \sum_{t=1}^T \|\Delta \boldsymbol{w}_t^l\|^2 + \frac{1}{C} \bigoplus_{k=1}^K \|\Delta \boldsymbol{\alpha}_{J_k}^l\|^2$$

To compute η^l on Husky, we use *Aggregator* and we use *Parameter Server* to compute $\Delta \boldsymbol{w}_t, t = 1, \cdots, T$. All the workers can then individually compute the step size and do the updates using (12) because every worker sees the same value of the parameters in the *Aggregator* and the *Parameter*

Server.

Convergence of $\boldsymbol{\alpha}$: With the availability of the new $\boldsymbol{\alpha}$, all the workers then check if $\boldsymbol{\alpha}$ has already been optimized by comparing the primal objective value with the dual objective value. Computing the primal objective value involves iterating all the input instances once, which is expensive in a single machine setting but this problem is greatly alleviated in a distributed setting. If the duality gap is smaller than a pre-determined threshold, then the problem of updating $\boldsymbol{\alpha}$ is solved. If not, all the workers will continue to optimize $\boldsymbol{\alpha}$. The algorithm is described in Algorithm 4.

Algorithm 4 Box-constrained Quadratic Optimization Algorithm

1: $l \leftarrow 0$, given $\boldsymbol{\alpha}^0$, computes $\boldsymbol{w}_t^0, t = 1, \cdots, T$
2: while $\boldsymbol{\alpha}^l$ is not optimal:
3: Obtain $\Delta\boldsymbol{\alpha}^l$ by distributively solving (15) on K workers in parallel
4: compute $\Delta\boldsymbol{w}_t^l = \bigoplus_{k=1}^K X_{t,J_k}^T Y_{J_k} \Delta\boldsymbol{\alpha}_{J_k}^l, t = 1, \cdots, T$
5: compute $\nabla f^{\mathcal{D}}(\boldsymbol{\alpha}^l, \boldsymbol{\mu})\Delta\boldsymbol{\alpha}^l$ and $(\Delta\boldsymbol{\alpha}^l)\bar{Q}\Delta\boldsymbol{\alpha}^l$
6: obtain η^l by (13)
7: $\boldsymbol{\alpha}^{\boldsymbol{l+1}} \leftarrow \boldsymbol{\alpha}^l + \eta^l \Delta\boldsymbol{\alpha}^l$,$\boldsymbol{w}_t^{l+1} \leftarrow \boldsymbol{w}_t^l + \eta^l \Delta\boldsymbol{w}_t^l$
8: $l \leftarrow l + 1$.

3.2.2 Updating $\boldsymbol{\mu}$

Once $\boldsymbol{\alpha}$ is solved, as described in 2, we first obtain the gradient $\frac{\partial J}{\partial \mu_t} = -\frac{1}{2}\boldsymbol{\alpha}^T(YX_tX_t^TY)\boldsymbol{\alpha} = -\frac{1}{2}\boldsymbol{w}_t^T\boldsymbol{w}_t$ and project this gradient to obtain a feasible descent direction $\boldsymbol{D}$. Then we use a simple binary search procedure to sample the objective values using half the maximum step size $\frac{\gamma_{max}}{2}$ and repeat this procedure until the step size to be applied is smaller than a pre-determined threshold.

3.2.3 Convergence of SimpleMKL

Once $\boldsymbol{\mu}$ has been updated, we test whether the duality gap of the MKL problem is small enough. In particular, the duality gap is measured by:

$$\max_t \boldsymbol{w}_t^T\boldsymbol{w}_t - \sum_{t=1}^T \mu_t \boldsymbol{w}_t^T\boldsymbol{w}_t \leq \epsilon$$

3.3 Convergence of Distributed Feature Generating Machine

Once SimpleMKL converges, we find a new violated constraint $\hat{\boldsymbol{d}}$ using (11). With this new $\hat{\boldsymbol{d}}$, we first check if it is in the constraint set $\mathcal{C}$. If yes, then the algorithm has converged and we can stop. If not, we add $\hat{\boldsymbol{d}}$ to the constraint set $\mathcal{C}$ and repeat the above procedure for solving the MKL problem. During implementation, other stopping criteria can also be used, for example, one can measure the relative difference between the current MKL objective value and the last MKL objective value.

3.4 Complexity

Let n be the number of input instances, m be the number of features in an instance, l be the average number of non-zero elements in each instance, K be the number of workers in a cluster and B be the number of non-zero entries in each kernel, which typically is set to $\{2, 5, 10, 20, 50, 100\}$, T be the cardinality of the violated constraints set $\mathcal{D}$ (According to [11], a maximum of 10 iterations are enough for Algorithm 1 to converge, which means T is a relatively small number). Since the primary objective of this algorithm is to deal with high dimensional problems, which typically has hundreds of thousands or even millions of features, we assume in this section $l >> TB$. We discuss in this section about the complexity of this algorithm in lieu of the implementation choices that we mentioned in section 3.2.1 about how to compute $\boldsymbol{w}^l$. In particular, we will focus on how different implementations affect the complexity of the algorithms in the Dual Coordinate Descent part and the communication part.

3.4.1 Implementation scheme 1: Do not cache $\boldsymbol{x}_i \odot \boldsymbol{d}$

If $\boldsymbol{x}_i \odot \boldsymbol{d}$ is not cached, we face a hurdle when computing $\sum_{t=1}^{T} \mu_t \boldsymbol{\alpha}^T Y X_t X_t^T Y \boldsymbol{\alpha}$. But the feature extraction operator "$\odot$" has an interesting property: $(\boldsymbol{x}_i \odot \boldsymbol{d})^T (\boldsymbol{x}_j \odot \boldsymbol{d}) = \boldsymbol{x}_i^T (\boldsymbol{x}_j \odot \boldsymbol{d})$, which is a result of the fact that elements in $\boldsymbol{d}$ can either be 0 or 1. This property allows us to factor

$\sum_{t=1}^{T} \mu_t \boldsymbol{\alpha}^T Y X_t X_t^T Y \boldsymbol{\alpha}$ as the following:

$$\begin{aligned}
\sum_{t=1}^{T} \mu_t \boldsymbol{\alpha}^T Y X_t X_t^T Y \boldsymbol{\alpha} &= \sum_{t=1}^{T} \mu_t \sum_i \sum_j \alpha_i \alpha_j y_i y_j (\boldsymbol{x}_i \odot \boldsymbol{d}_t)^T (\boldsymbol{x}_j \odot \boldsymbol{d}_t) \\
&= \sum_{t=1}^{T} \mu_t \sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^T (\boldsymbol{x}_j \odot \boldsymbol{d}_t) \\
&= \sum_{t=1}^{T} \mu_t (\sum_i \alpha_i y_i \boldsymbol{x})^T ((\sum_j \alpha_j y_j \boldsymbol{x}_j) \odot \boldsymbol{d}_t) \\
&= (\sum_i \alpha_i y_i \boldsymbol{x}_i)^T ((\sum_j \alpha_j y_j \boldsymbol{x}_j) \odot (\sum_{t=1}^{T} \mu_t \boldsymbol{d}_t)) \\
&= \boldsymbol{w}^T (\boldsymbol{w} \odot \tilde{\boldsymbol{d}})
\end{aligned}$$

where $\boldsymbol{w} = \sum_i \alpha_i y_i \boldsymbol{x}_i$ and $\tilde{\boldsymbol{d}} = \sum_{t=1}^{T} \mu_t \boldsymbol{d}_t$ can be cached to speed up the algorithm.

3.4.2 Implementation scheme 2: Cache $\boldsymbol{x}_i \odot \boldsymbol{d}$

If $\boldsymbol{x}_i \odot \boldsymbol{d}$ is cached, then inside the SVM solver, we can treat it as if we were solving T SVM problems and each one of the SVM problem is associated with a particular kernel $K_t(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i \odot \boldsymbol{d}_t)^T (\boldsymbol{x}_j \odot \boldsymbol{d}_t)$. And for each one of these problems we will train a weight vector $\boldsymbol{w}_t$. Then the complete weight vector $\boldsymbol{w}$ is just the sum of all these T weight vectors, weighted by μ_t.

3.4.3 Time Complexity

Implementation Scheme 1: Using the implementation scheme 1, $\tilde{\boldsymbol{d}}$ will be a dense vector of size m where each $\boldsymbol{d}_t$ is only a vector of size B and $B << m$. This also means that $\boldsymbol{w}$ will also be a dense vector of size m. In each Dual Coordinate Descent iteration, we update $\boldsymbol{w}$ with $\boldsymbol{w} \leftarrow \boldsymbol{w} + (\alpha_i - \bar{\alpha_i}) y_i (\boldsymbol{x}_i \odot \tilde{\boldsymbol{d}})$ Therefore, each inner iteration of Dual Coordinate Descent algorithm will need approximately $O(\frac{nl}{K})$ operations. After local $\boldsymbol{\alpha}_{J_k}$ is solved, we need an *allreduce* step to synchronize all the workers. In this scheme, we need to synchronize $\boldsymbol{w}$ and several scalar quantities, resulting in a communication cost of $O(m)$. Notice the use of sparse representation will not help here because $\boldsymbol{w}$ will be dense vector of size m.

Implementation Scheme 2: Using the implementation scheme 2, we need an extra step of caching the kernel each time we add a new control variable $\boldsymbol{d}_t$, which results in a cost of $O(\frac{n(B+l)}{K})$. For the representation of $\boldsymbol{w}_t$, we can store $\boldsymbol{w}_t$ using a sparse representation because

we have the information about $\boldsymbol{d}_t$, which tells us exactly which features will likely be non-zero. Since each instance $\boldsymbol{x}_i$ will have on-average l non-zero elements. $\boldsymbol{x}_i \odot \boldsymbol{d}_t$ will have on-average $\frac{l}{m}B$ non-zero elements. Therefore, each iteration of Dual Coordinate Descent method will need approximately $O(\frac{l}{m}\frac{n}{K}TB)$ operations. And synchronizing $\boldsymbol{w}_t, t = 1 \cdots, T$ has a communication cost of $O(TB)$. We thus obtain a speed up of $O(\frac{m}{TB})$. Thus we managed to cut down the running time cost and the communication cost dramatically. Notice in a lot of sense this algorithm is independent of the number of features of the input data because all we are interested in are TB number of useful features. Since we adopt a scheme that caches $\boldsymbol{x} \odot \boldsymbol{d}$, we reduce the original high dimensional problem to a low dimensional problem with dimension equal to TB during runtime.

Analysis for both Schemes: The above analysis mainly focuses on Dual Coordinate Descent method and the communication in the Box constrained Quadratic Optimization method. Now we analyze other aspects of the algorithm. In a distributed setting, when finding the most violated constraint, we need to synchronize the weight vectors $\boldsymbol{w}$ for all the features, which lead to a communication cost of $O(m)$. When we are dealing with high dimensional datasets, this is a very expensive step. And this step becomes more expensive as we increase the number of workers in the cluster. But luckily we do not need to do this step very often as T ranges from 1 to 20. It is proven in [12] that our distributed SVM solver Box constrained Quadratic Optimization method has linear convergence rate. As for SimpleMKL, according to our experiment, typically each iteration composes of 2 search calls for descent direction and 4 search calls for step size. Although this may seem very expensive because for each call of descent direction search or step size search we need to invoke the SVM solver, we can do more caching and initialize the solver with previous values of $\boldsymbol{\alpha}$ to speed up the algorithm. In summary, our algorithm takes advantage of the 0-1 control variable and caching to reduce the a high dimensional problem into a low dimensional one, thus enabling our algorithm to handle data of ultra-high dimension. In addition, we take advantage of a distributed computing framework to scale up our algorithm in the presence of input data with huge volume of input instances.

3.4.4 Space Complexity

Implementation Scheme 1: The space requirement of Scheme 1 is only $O(\frac{nl}{K})$

Implementation Scheme 2: The space requirement of Scheme 2 is $O(\frac{nl+n\frac{l}{m}TB}{K})$, which is not too worse than Scheme 1 considering that TB is a relatively small number. Besides, one of

the main advantages of a distributed algorithms is that we have more memory to spare and we can always add more machines to scale up the total memory available in the cluster. Therefore, we are able to use the nature of a distributed algorithm to cancel out the potential drawbacks caused by caching. In summary, this algorithm is computationally efficient as the Box constrained Quadratic Optimization method has linear convergence rate[12] and each iteration of the algorithms involves only tens of SVM solver call.

4 Experiments

In this section, we evaluate the performance of the single machine FGM algorithm and our distributed FGM algorithm on two synthetic dataset and a collection of real world datasets.

4.1 Datasets

4.1.1 Synthetic Datasets

The synthetic datasets consist of one small dataset (two variants) and one large dataset. Both datasets are binary classification problems with the ground truth known.

SYN-SMALL For the small dataset, we generate the first variant with 10,000 samples and 200 features with each value initialized with a uniform distribution $U(-1, 1)$ and a sparsity ratio of 15%. A ground truth weight vector $\boldsymbol{w}$ is then generated with 200 features with each value initialized with a uniform distribution $U(-1, 1)$ and sparsity ration of 15%. The dot product $\langle \boldsymbol{w}, \boldsymbol{x} \rangle$ is then used to generate the label $y_i \in \{-1, +1\}$. We call this dataset SYN-SMALL.

SYN-SMALL-HIGH-DIM Another variant of the small dataset is generated by appending 19800 irrelevant features to every instances in the first variant with each value initialized with a uniform distribution $U(-1, 1)$ and a sparsity ratio of 15%. We call this dataset SYN-SMALL-HIGH-DIM. Ideally, a feature selection algorithm should be able to recover the relevant features from the first 200 features.

SYN-LARGE The second synthetic dataset is generated with the same setting as the first synthetic dataset except the second one is generated with 1,000,000 samples and 200 features. We call this dataset SYN-LARGE.

4.1.2 Real-World Datasets

The real-world datasets consist of two small datasets and two large datasets. All these four datasets are from the LIBSVM[20] website without further pre-processing. The two small datasets, **BREAST-CANCER** and **LEUKEMIA** are insufficient in the amount of instances. For the **BREAST-CANCER** dataset, we first merge the train set and the test set together and then randomly split them into two datasets of equal size and perform our experiment several times. For the **RCV1.BINARY** dataset, we also randomly split the whole dataset into two dataset of equal size. The **URL** dataset[21] consists of 120 days of URL data used for detecting malicious URL. Since the features are engineered from host-based features, lexical features and other real-valued features using the bag of words model, the number of features goes up to millions! We randomly shuffle the 120 days of data and split them into three datasets of **small**, **large** and **extra large** sizes respectively with details described in table1.

Table 1: Datasets used in the experiments

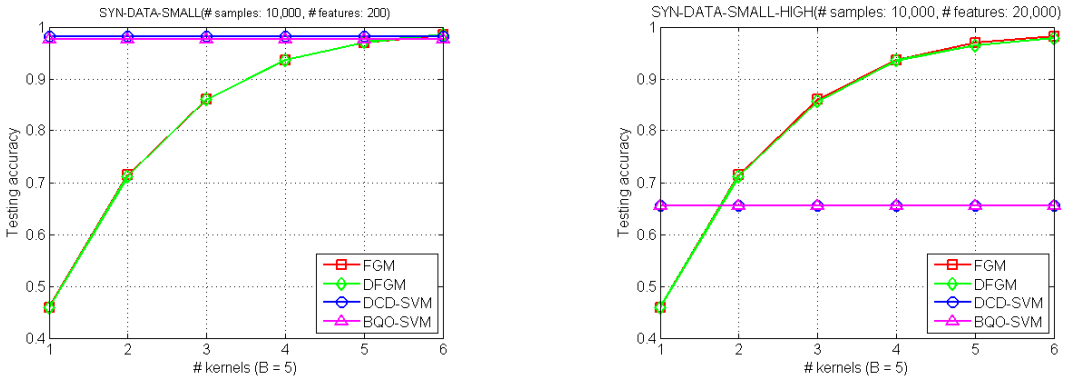
Dataset	# Training Points	# Testing Points	# Features
SYN-SMALL	10,000	10,000	200
SYN-SMALL-HIGH-DIM	10,000	10,000	20,000
BREAST-CANCER	21	21	7129
LEUKEMIA	38	34	7129
SYN-LARGE	1,000,000	10,000	10,000
RCV1.BINARY	338,700	338,699	47,236
URL-SMALL	159,745	159,740	3,231,961
URL-LARGE	958,460	479,220	3,231,961
URL-EXTRA	1,437,680	479,220	3,231,961

4.2 Experimental Set Up

In our experiments, comparisons are conducted among single machine FGM algorithm and distributed FGM algorithms. We denote distributed FGM run with K workers as DFGM-K. Dual Coordinate Descent SVM algorithm, denoted as DCD-SVM and Box constrained Quadratic Optimization SVM algorithm, denoted as BQO-SVM, from the LIBlinear library [22] serve as baselines, which select all features for classification. In all our experiments, the parameter C for SVM is set to 1 and the loss function for SVM is set to l2-loss.

4.3 Experiments on small datasets

To verify the correctness of our algorithms, two synthetic experiments are performed first. These two synthetic datasets share the same ground truth weight vector $\boldsymbol{w}$, with only 29 non-zero entries among the first 200 entries. As seen in Figure 2(a), as the number of kernels (number of selected features) increases, the testing accuracies of both FGM and DFGM increase and converge to the same testing accuracies as DCD-SVM and BQO-SVM when the number of kernels reaches 6 (number of selected features ≥ 29 because $B = 5$). From Figure 2(b), we can see that both FGM and DFGM successfully selects the relevant features among a pool of features mixed with a large number of irrelevant features when DCD-SVM and BQO-SVM suffers from giving weights to these irrelevant features and generalize poorly on the testing set. These two pictures show that our algorithm is effective as a feature selection algorithm in selecting useful features among noisy irrelevant features. We summarize the testing accuracies of the methods that we are interested in on both the synthetic small datasets and real-world small datasets in table 2. This table mainly serves to show the effectiveness of our algorithm in selecting useful features. As the primary purpose of this report is to propose a distributed algorithm that can scale in the presence of large scale high dimensional datasets, we focus on the next section, where we perform experiments on large datasets to show the scalability of our algorithm.



(a) Testing accuracy

(b) Testing accuracy

Figure 2: Testing accuracies on small synthetic datasets

4.4 Experiments on large datasets

In this subsection, we show the scalability of our algorithm first by comparing the performances of DFGM running on different number of worker machines with the performance of FGM on **SYN-DATA-LARGE**, **RCV1.BINARY** and **URL-SMALL** datasets. We then compare the

Table 2: Testing accuracies of various algorithms on small datasets

DATA SETS	DCD-SVM	BQO-SVM	B	FGM	DFGM
SYN-DATA-SMALL	0.9809	0.9761	2	0.9618	0.9767
	0.9809	0.9761	5	0.9686	0.9831
	0.9809	0.9761	10	0.9850	0.9807
SYN-DATA-SMALL-HIGH	0.6550	0.6559	2	0.9618	0.9772
	0.6550	0.6559	5	0.9816	0.9784
	0.6550	0.6559	10	0.9704	0.9661
BREAST-CANCER	0.8095	0.8095	2	0.7619	0.8095
	0.8095	0.8095	5	0.8095	0.8095
	0.8095	0.8095	10	0.7619	0.7619
	0.8095	0.8095	20	0.8571	0.9048
LEUKEMIA	0.7941	0.7941	2	0.9412	0.9412
	0.7941	0.7941	5	0.9412	1.0000
	0.7941	0.7941	10	0.9706	0.9706
	0.7941	0.7941	20	0.9412	0.9706

performances among DFGM running on different number of worker machines on **URL-LARGE** and **URL-EXTRA** datasets. We separate the experiments into these two parts because FGM can only run on single machine, which makes it infeasible for very large datasets with millions of samples.

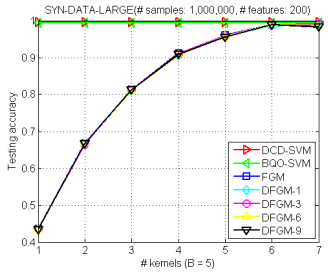


Figure 3: SYN-DATA-LARGE

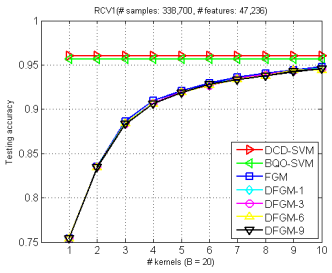


Figure 4: RCV1.BINARY

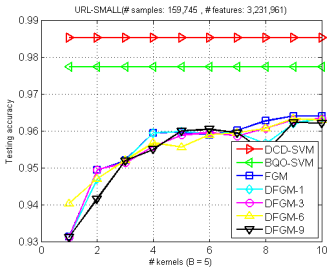


Figure 5: URL-SMALL

Figure 6: Testing accuracies on large datasets

4.4.1 FGM v.s. DFGM-K

Figure 3 summarizes the testing accuracies of various methods on the three large datasets respectively. Figure 7-9 summarizes the total running time and the speedup of DFGM compared to FGM. Finally, table 3 gives detailed information about the training time and communication time on the **URL** datasets. There are two simple observations that can be made from Figure 7-9. First, as expected, the training time reduces as we increase the number of worker machines in our cluster, which is what we expect of a distributed algorithm. Second, the speedup resulted by

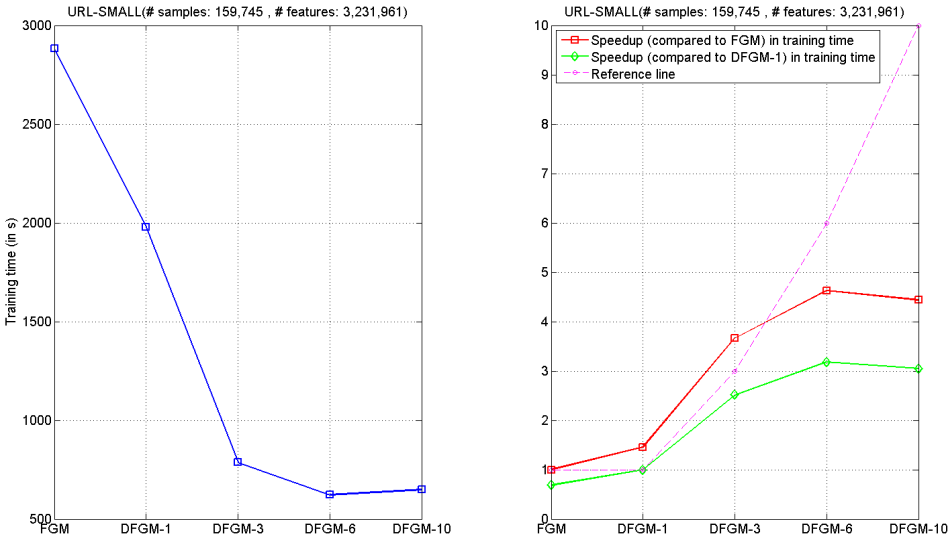


Figure 7: URL-SMALL

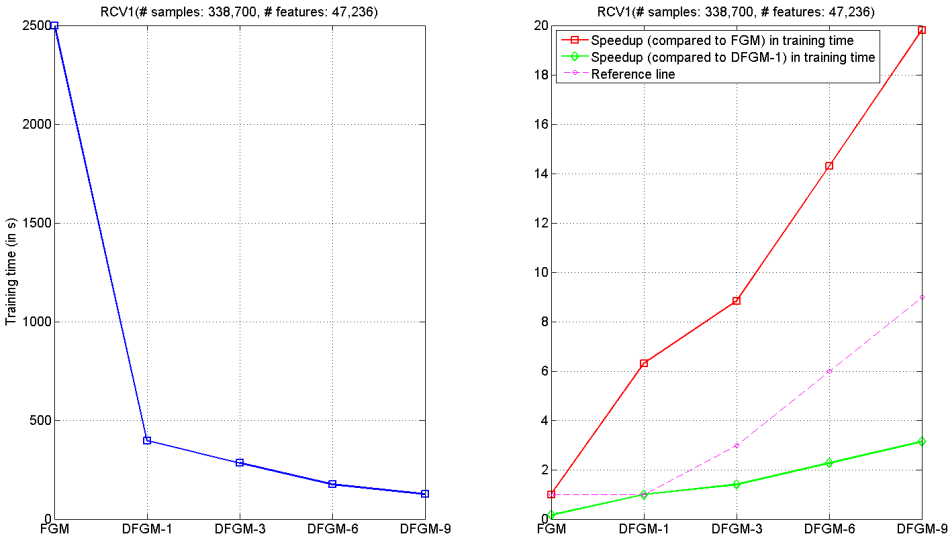


Figure 8: RCV1.BINARY

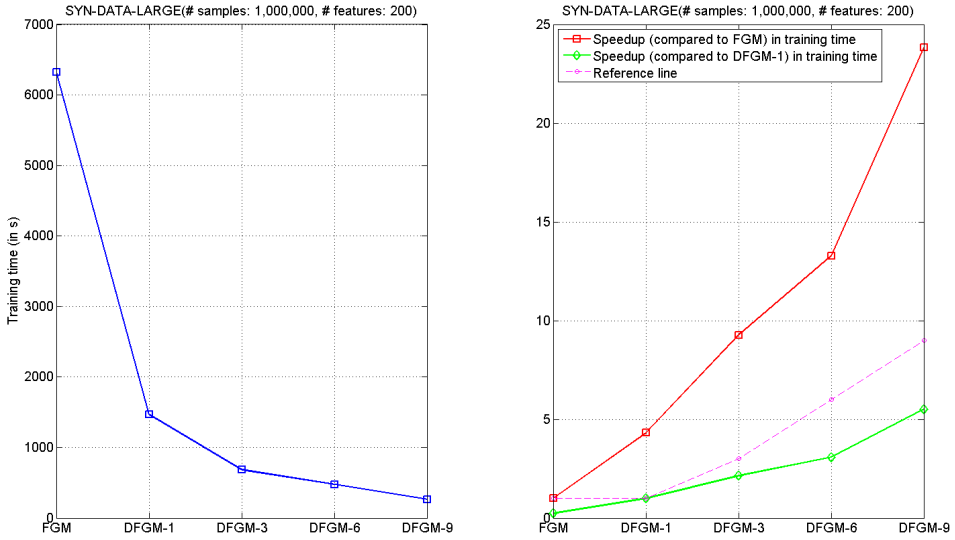


Figure 9: SYN-DATA-LARGE

increasing the number of worker machine is not linear but sub-linear.

Table 3: Training time and communication time on **URL** datasets

DATA SETS	DFGM-K	training time	communication time	$\frac{\text{communication time}}{\text{training time}}$
URL-SMALL	DFGM-1	1978.68	0	0%
	DFGM-3	786.40	96.35	12.25%
	DFGM-6	622.62	167.00	26.66%
	DFGM-10	649.45	256.58	39.51%
URL-LARGE	DFGM-5	1046.45	33.91	3.24%
	DFGM-10	707.76	55.14	7.79%
	DFGM-15	564.70	79.06	14.00%
URL-EXTRA	DFGM-5	2044.01	40.42	1.98%
	DFGM-10	1302.82	67.15	5.15%
	DFGM-15	969.79	92.68	9.57%

4.4.2 Comparisons among DFGM-K

Figure 10 shows our experiments of running 5, 10, 15 and 20 worker machines on **URL-LARGE** and **URL-EXTRA** respectively. This experiment is conducted mainly to investigate the scalability of our distributed algorithms in the presence of large amount of data. As shown in the figure, having a larger dataset (**URL-EXTRA** approximately 1.5 times larger than **URL-LARGE**) that is still within the capacity of the cluster does make the speed up curve steeper, indicating that the algorithm can scale in the presence of larger amount of data and more worker machines.

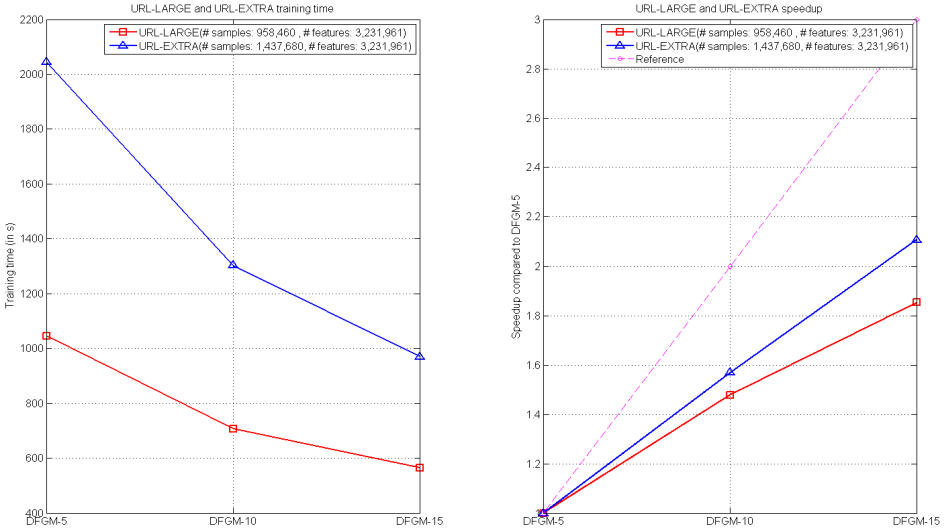


Figure 10: URL-LARGE and URL-EXTRA

5 Discussions

The simple observations and figures above show that our algorithm can scale in the presence of more data and more worker machines. However, the scaling factor, which is at the best in the **SYN-DATA-LARGE** dataset, achieving 0.6 (5.5 speedup with 9 worker machines), is still relatively small. In this section we discuss some of the issues that might have caused this problem and possible improvements. In the mean time, we further discuss in details about the results obtained in the experiments section.

5.1 Load Balancing and Data Heterogeneity

In a distributed computing setting, it is usually assumed that computing nodes are homogeneous in their capability. But this assumption is easily violated in a real-world setting where the users only communicate with the resource manager to negotiate computing resources for their jobs. It is possible that the some machines are running several different tasks submitted by different users while others are mostly idle, thus violating the nodes homogeneity assumptions. This heterogeneity not only affects the available computing performances at run-time but also the amount of data handled by each worker machine. Since Husky adopts a Master-Slave cluster architecture, during run time the Slave machine sends a request to the Master requesting locations of input data

and the Master responds with the block location in HDFS[9]. Thus if a worker machine is slow, then it may acquire far less data compared to other worker machines. For examples, in our experiments, it is observed that some slow workers can have as high as 25% less data than the worker machines holding the most data. Therefore, the difference of amount of data processed by each worker machine and the loading status of each worker machine causes the "stragglers" [23] in our experiments, which in turn hurts the scalability performance of our algorithm. Another issues that affect the scaling factor of our algorithm is the inherent heterogeneity of the input data. Intuitively, for a SVM problem, some samples, which at the optimum correspond to the "Support Vectors" are harder to optimize and some samples, which at the optimum have zero-weight in the α, are easier to optimize. This is evident from the fact that the scaling factor is at the best on the **SYN-DATA-LARGE** dataset. Intuitive this is so because the **SYN-DATA-LARGE** dataset is generated by a uniform distribution and each sample is relatively identical to each other and **URL** and **RCV1** are real world datasets, which contain more realistic and extreme cases. While the second issue is related to the input data and may not be easily solved without further preprocessing of the data, the first issue can be alleviated by doing a load balancing between different workers before running the DFGM algorithm. In particular, we can make use of the *Migrate Channel* in Husky to allow highly loaded worker machines to offload some of the samples to less-loaded worker machines. By doing load balancing, we can better utilize the computing resources available and bridging the gap between normal workers and the "stragglers", thereby reducing the average waiting time during the communication procedure.

5.2 Number of worker machines

From the right picture of Figure 7-9 we can see that approximately it takes two times or even three times amount of worker machines to reduce the training time to half on **RCV1** and **URL-SMALL** datasets when compared to DFGM-1. In particular, we can observe that DFGM-10 is worse than DFGM-6 on the **URL-SMALL** dataset. To understand why this is so, we refer the readers to table 3 where it can be seen from the first row of the table that DFGM-10 spends 40% of the time in communication, which is a clear indication that with only 159,745 samples in the **URL-SMALL** dataset, having 10 worker machines in the cluster is an overkill and hurts the performance. Furthermore, we can see from **RCV1**, **SYN-DATA-LARGE**, **URL-LARGE** and **URL-EXTRA**, which has more samples than **URL-SMALL**, that the speedup curve gets steeper as we increase the number of samples in the data given the same number of worker

machines. Thus the lesson learned is that when running a distributed machine learning algorithm where communication through network is frequent and unavoidable, we need to tune the number of worker machines according to the size of the input data rather than blindly increasing the number of worker machines in the cluster.

5.3 SVM-Solver

From the comparisons between FGM and DFGM-K, we can see that our distributed algorithm running on only one machine, performs extremely well when compared to FGM. We investigate into this problem and find out that the difference of performance is due to the differences of the SVM solver that these two methods use. In particular, FGM uses DCD-SVM, which circularly goes through the training dataset and use projected gradient information to update the Lagrange multiplier α_i until all the α_is are at optimum, at which point the projected gradient vanishes to a zero vector. In comparison, DFGM uses BQO-SVM, which also circularly goes through the training dataset but considers second order approximation. For convergence criterion, BQO-SVM stops when the first order term of the Taylor series expansion of the objective function with respect to $\boldsymbol{\alpha}$ becomes greater than 0. Although each iteration of BQO-SVM is slower due to the use of second order approximation, we find out in our experiments that BQO-SVM usually takes far less iterations to converge when compared to DCD-SVM. For example, DCD-SVM takes 72 iterations, totaling 50 seconds to converge and achieve a classification accuracy of 97.74% on the testing set. In comparison, BQO-SVM takes only 2 iterations, totaling 8 seconds to converge on a solution that yields classification accuracy of 97.41% on the testing set. From this example, we can see that DCD-SVM actually spends a majority of the training time achieving a solution that is not significantly better than the one found by BQO-SVM. Since the SimpleMKL algorithm performs several steps of searching of the parameter using the SVM solver in each iteration, sometimes the solution returned from the solver may simply be discarded due to some other constraints. Thus the problem of the DCD-SVM solver gets magnified during the many iterations of the SimpleMKL algorithm. Therefore, it may not be a good idea to spend too much computational resources in finding a solution that brings very little or maybe no returns and this explains why DFGM-1 performs much better than FGM in our experiment. Although it is not done in this report, this problem can be avoided if we modify the solver to use relative changes in the objective values as a stopping criterion.

5.4 Trade-off between Communication and Local Computation

As mentioned in section 3.2.1, each iteration of BQO-SVM involves solving a local SVM problem using the DCD-SVM solver and a communication procedure to synchronize with other worker machines. Thus there is a trade-off between communication cost and local computational cost. If we allow the DCD-SVM more time to find a better solution, then the communication cost will be smaller. On the other hand, if we stop DCD-SVM prematurely, there may be more noises/perturbations in our descent direction and the BQO-SVM algorithm will need more iteration to converge, thus increasing the communication cost. Since for the datasets used in our experiments, BQO-SVM typically takes a few iterations to converge, we set the maximum iteration for DCD-SVM to 1 like the implementation given by [12], which yields faster convergence for BQO-SVM but incurs more communication cost. A possible improvement should be to adopt a more general stopping criterion for DCD-SVM by taking into consideration the maximum number of iterations, the duality gap and the relative changes in the objective values. For tunning of the parameters for the general stopping criterion, we can sample a portion of the input data and list out a similar table as table 3 to determine whether the training time is dominated by communication or by local computation. If the training time is dominated by communication, we can impose a more stringent stopping criterion for DCD-SVM and vice versa.

6 Conclusion

In this report, we propose a distributed algorithm to learn a Sparse SVM for feature selection on high dimensional data. By introducing a 0-1 control variable, our method achieves the goal of sparsity while being computationally efficient. The use of 0-1 control variable allows our algorithm to be unaware of the number of features in the input data. By taking advantages of the linear Kernel and cleverly caching important data, we design an efficient and scalable distributed solver for the SVM problem with multiple linear kernels. Finally, by leveraging an efficient distributed computing framework - Husky, and making an implementation choice that reduces the amount of communication between workers, we only need to pay a communication cost that is linear in the number of desired features to scale up the performance in the presence of large scale high dimensional data.

References

[1] Andrew McAfee, Erik Brynjolfsson, Thomas H Davenport, DJ Patil, and Dominic Barton. Big data. *The management revolution. Harvard Bus Rev*, 90(10):61–67, 2012.

[2] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[3] Jason Weston, André Elisseeff, Bernhard Schölkopf, and Mike Tipping. Use of the zero-norm with linear models and kernel methods. *Journal of machine learning research*, 3(Mar):1439–1461, 2003.

[4] Paul S Bradley and Olvi L Mangasarian. Feature selection via concave minimization and support vector machines. In *ICML*, volume 98, pages 82–90, 1998.

[5] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1-3):389–422, 2002.

[6] Mingkui Tan, Ivor W Tsang, and Li Wang. Towards ultrahigh dimensional feature selection for big data. *Journal of Machine Learning Research*, 15(1):1371–1429, 2014.

[7] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[8] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[9] Apache Hadoop. `http://hadoop.apache.org/`.

[10] Fan Yang, Jinfeng Li, and James Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *Proceedings of the VLDB Endowment*, 9(5):420–431, 2016.

[11] Mingkui Tan, Li Wang, and Ivor W Tsang. Learning sparse svm for feature selection on very high dimensional datasets. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 1047–1054, 2010.

[12] Ching-Pei Lee and Dan Roth. Distributed box-constrained quadratic optimization for dual linear svm. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 987–996, 2015.

[13] Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S Sathiya Keerthi, and Sellamanickam Sundararajan. A dual coordinate descent method for large-scale linear svm. In *Proceedings of the 25th international conference on Machine learning*, pages 408–415. ACM, 2008.

[14] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[15] Bernhard Schiilkopf. The kernel trick for distances. *Advances in neural information processing systems*, 13:301–307, 2001.

[16] Alain Rakotomamonjy, Francis R Bach, Stéphane Canu, and Yves Grandvalet. Simplemkl. *Journal of Machine Learning Research*, 9(Nov):2491–2521, 2008.

[17] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[18] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[19] John Platt et al. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[20] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, May 2011.

[21] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Identifying suspicious urls: An application of large-scale online learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 681–688, New York, NY, USA, 2009. ACM.

[22] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *Journal of machine learning research*, 9(Aug):1871–1874, 2008.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.