Journal of Machine Learning Research 17 (2016) 1-37 Submitted 4/15; Revised 11/15; Published 6/16

# Multi-Task Learning for Straggler Avoiding Predictive Job Scheduling

**Neeraja J. Yadwadkar** NEERAJAY@EECS.BERKELEY.EDU
*Division of Computer Science*
*University of California*
*Berkeley, CA 94720-1776, USA*

**Bharath Hariharan** BHARATH2@EECS.BERKELEY.EDU
*Division of Computer Science*
*University of California*
*Berkeley, CA 94720-1776, USA*

**Joseph E. Gonzalez** JEGONZAL@EECS.BERKELEY.EDU
*Division of Computer Science*
*University of California*
*Berkeley, CA 94720-1776, USA*

**Randy Katz** RANDY@CS.BERKELEY.EDU
*Division of Computer Science*
*University of California*
*Berkeley, CA 94720-1776, USA*

**Editor:** Urun Dogan, Marius Kloft, Francesco Orabona, and Tatiana Tommasi

## Abstract

Parallel processing frameworks (Dean and Ghemawat, 2004) accelerate jobs by breaking them into tasks that execute in parallel. However, slow running or *straggler* tasks can run up to 8 times slower than the median task on a production cluster (Ananthanarayanan et al., 2013), leading to delayed job completion and inefficient use of resources. Existing straggler mitigation techniques wait to detect stragglers and then relaunch them, delaying straggler detection and wasting resources. We built Wrangler (Yadwadkar et al., 2014), a system that predicts when stragglers are going to occur and makes scheduling decisions to avoid such situations. To capture node and workload variability, Wrangler built separate models for every node and workload, requiring the time-consuming collection of substantial training data. In this paper, we propose multi-task learning formulations that share information between the various models, allowing us to use less training data and bring training time down from 4 hours to 40 minutes. Unlike naive multi-task learning formulations, our formulations capture the shared structure in our data, improving generalization performance on limited data. Finally, we extend these formulations using group sparsity inducing norms to automatically discover the similarities between tasks and improve interpretability.

## 1. Introduction

Distributed processing frameworks, such as (Dean and Ghemawat, 2004; Isard et al., 2007; Li, 2009), split a data intensive computation job into multiple smaller tasks, which are then executed in parallel on commodity clusters to achieve faster job completion. A natural con-

sequence of such a parallel execution model is that the slow-running tasks, commonly called *stragglers*, potentially delay overall job completion. In a recent study, Ananthanarayanan et al. (2013) show that straggler tasks are on an average 6 to 8 times slower than the median task of the corresponding job, despite existing mitigation techniques. Suri and Vassilvitskii (2011) describe the intensity of straggler problem as the "curse of the last reducer", causing the last 1% of the computation to take much longer to finish execution. Mitigating stragglers thus remains an important problem. In this paper, our focus is on machine learning approaches for predicting and avoiding these stragglers.

Existing approaches to straggler mitigation, whether reactive or proactive, fall short in multiple ways. Commonly used reactive approaches such as speculative execution (Dean and Ghemawat, 2004), act after the tasks have already slowed down. Proactive approaches that launch redundant copies of a task in a hope that at least one of them will finish in a timely manner (Ananthanarayanan et al., 2013), waste resources. It is hard to, *a priori*, identify which factors, and configurations lead to stragglers, motivating a data driven machine learning approach to straggler prediction. Bortnikov et al. (2012) explored predictive models for predicting slowdown for tasks. But they learn a single model for a cluster of nodes, ignoring the variability caused due to the heterogeneity across nodes and across jobs from different workloads. Moreover, they built models but did not show if these models could indeed improve job completion times, which is what matters ultimately. Gupta et al. (2013) explored learning based job scheduling by matching node capabilities to job requirements. However, their approach is not designed to deal with stragglers effectively because it does not model the time-varying state of the nodes (for instance, how busy they are). Moreover, the effectiveness of these methods in reducing job-completion times in real world clusters hasn't been shown. To this end, we built Wrangler (Yadwadkar et al., 2014), a system that builds predictive models of straggler behavior and incorporates this model in the scheduler, improving the $99^{th}$ percentile job completion time by up to 61% as compared to speculative execution for real world production level workloads[1] from Facebook and Cloudera's customers.

Proactive model based approaches including Wrangler, have previously treated each workload and even compute node as a separate straggler estimation task with independent models. The decision to model each workload and node independently is motivated by the observation that the resource contention patterns that cause stragglers can vary from node to node and workload to workload. For a detailed analysis about causes of stragglers across nodes and workloads, see Section 4.2.2. of Wrangler, Yadwadkar et al. (2014). To address such heterogeneity in the nodes and changing workload patterns, training separate models for each workload being run on each node, is considered necessary. However, such independent models pose two critical challenges: (1) each new node and workload requires new training data which can take hours to collect, delaying the application of model based scheduling, and (2) clusters with many nodes may only have limited data for a given workload on each node leading to lower quality models.

These shortcomings can be addressed if each classifier is able to leverage information gleaned at other nodes and from other workloads. For instance, when there is not enough

1. Clusters are used for different purposes, and statistics such as the kinds of jobs submitted, their resource requirements and the frequency at which they are submitted vary depending upon the usage. We call one such distribution of jobs a workload. Section 3.2 explains the notion of workloads in further detail.

data at a node for a workload, we can gain from the data collected at that node while it was executing other workloads, or from other nodes running the same workload. Such information sharing falls in the ambit of multi-task learning (MTL), where the learner is embedded in an environment of related tasks, and the learner's aim is to leverage similarities between the tasks to improve performance of all tasks.

In this paper, we propose an MTL formulation for learning predictors that are more accurate and generalize better than existing straggler predicting models. Basic MTL formulations often assume only limited correlation structure between learning tasks. However, our application has an additional overlapping group structure: models built on the same node for different workloads can be grouped together, as can be the models corresponding to the same workload running on different nodes. We propose a new formulation that leverages this overlapping group structure, and show that doing so significantly reduces the number of parameters and improves generalization to tasks with very little data.

In applications such as ours, high accuracy is not the only objective: we also want to be able to gain some insight into what is causing these stragglers. Such insight can aid in debugging root causes of stragglers and system performance degradation. Automatically learned classifiers can be opaque and hard to interpret. To this end, we propose group sparsity inducing mixed-norm regularization on top of our basic MTL formulation to automatically reveal the group structure that exists in the data, and hopefully provide insights into how straggler behavior is related across nodes and workloads. We also explore sparsity-inducing formulations based on feature selection that attempt to reveal the characteristics of straggler behavior.

For our experimental evaluation, we use a collection of real-world workloads from Facebook and Cloudera's customers. We evaluate not only the prediction accuracy but also the improvement in job completion times, which is the metric that directly impacts the end user. We show that our formulation to predict stragglers allows us to reduce job completion times by up to 59% over the previous state-of-the-art learning base system, Wrangler (Yadwadkar et al. (2014)). This large reduction arises from a 7 point increase in prediction accuracy. Further, we can get equal or better accuracy than Wrangler (Yadwadkar et al. (2014)) using a sixth of the training data, thus bringing the training time down from 4 hours to about 40 minutes. We also provide empirical evidence that compared to naive MTL formulations, our formulation indeed generalizes to new tasks, i.e, it is significantly better at predicting straggler behavior for nodes for which we have not collected enough data.

Finally, while our initial motivation and experimental validation focus on the straggler avoidance problem, our learning formulations are general and can be applied to other systems that train node or workload dependent classifiers (Gupta et al., 2013; Delimitrou and Kozyrakis, 2014). For instance, ThroughputScheduler (Gupta et al., 2013) uses such classifiers to allot resources to tasks, and can benefit from such multitask reasoning. We leave these extensions to future work.

To summarize, our key contributions are:

1. An MTL formulation of the straggler estimation problem that allows us to use less data and reduce parameters, improving generalization (Section 4.1).

2. A mixed-norm group sparsity inducing formulation, that automatically detects group structure (Section 4.3) facilitating interpretability.

3. An efficient optimization technique based on a reduction to the standard SVM (Cortes and Vapnik (1995)) formulation (Section 4.2).

4. A comprehensive evaluation as part of a complete system for predicting and avoiding stragglers in real world production cluster traces (Section 5). We show that, compared to existing approaches, our formulations

   (a) avoid stragglers better with 7% improved prediction accuracy, improving job completion times significantly with up to 57.8% improvement in the $99^{th}$ percentile, and reducing net resource usage by up to 40%, and

   (b) can work even with a sixth of the training data and thus a much shorter training period, reducing the training data collection time significantly from 4 hours to 40 minutes.

In what follows, we first give some background on stragglers in Section 2. We then describe Wrangler and discuss its shortcomings in Section 3. In Section 4, we describe our multi-task learning formulations. In Section 5, we empirically evaluate our formulations on real world production level traces from Facebook and Cloudera's customers. We end with a discussion of related work.

## 2. Background and Motivation

The growing popularity of Internet-based applications, in addition to reduced costs of storage media, has resulted in generation of data at an unprecedented scale. Analytics on such huge datasets has become a driving factor for business. Parallel processing on commodity clusters has emerged as the de-facto way of dealing with the scale and complexity of such data-intensive applications. Dean and Ghemawat (2004) originally proposed the MapReduce framework at Google to process enormous amounts of data. MapReduce is highly scalable to large clusters of inexpensive commodity computers. Hadoop (White, 2009), an open source implementation of MapReduce, has been widely adopted by industry over the last decade.

A data intensive application is submitted to a cluster of commodity computers as a MapReduce *job*. To accelerate completion, MapReduce divides a job into multiple *tasks*. A cluster scheduler assigns these to machines (nodes), where they are executed in parallel. A job finishes when all its tasks have finished execution. A key benefit of such frameworks is that they automatically handle failures (which are more likely to occur on a cluster of commodity computers) without needing extra efforts from the programmer. Two basic modes of failures are the failure of a node and the failure of a task. If a node crashes, MapReduce re-runs all the tasks it was executing on a different node. If a task fails, MapReduce automatically re-launches it.

However, a tricky situation arises when a node is available, but is performing poorly. This causes the tasks scheduled on that node to execute slower than other tasks of the same job scheduled on other nodes in the cluster. Since a job finishes execution only when all its

tasks have finished execution, such slow-running tasks, called *stragglers*, extend the job's completion time. This, in turn, leads to increased user costs.

Stragglers abound in the real world. According to Ananthanarayanan et al. (2013), if stragglers did not exist in real-world production clusters, the average job completion times would have been improved by 47%, 29% and 36% in the Facebook, Bing and Yahoo traces respectively. We observed that 22-28% of the total tasks are stragglers in a replay[2] of Facebook and Cloudera's customers' Hadoop production cluster traces. Thus, stragglers are a major hurdle in achieving faster job completions.

It is challenging to deal with stragglers (Dean and Ghemawat, 2004; Zaharia et al., 2008; Ananthanarayanan et al., 2010). Dean and Ghemawat (2004) mentioned that stragglers could arise due to various reasons, such as competition for resources, problematic hardware, and mis-configurations. Ananthanarayanan et al. (2010) report that consistently slow nodes are rarely the reason behind stragglers; the majority are caused by transient node behavior. The reason is simple: cluster schedulers assign multiple tasks, belonging to the same or different jobs, to be executed simultaneously on a node. Depending on the node's characteristics and current load, as well as the characteristics of the workloads, these jobs may result in counterproductive resource contention patterns. Moreover, this gives rise to dynamically changing task execution environments, causing large variations in task execution times. We cannot simply look at the initial configuration of the node and decide if it will cause stragglers. We must track the continuously changing state of the node (its memory usage, cpu usage etc.) and use that to predict straggler behavior.

Above and beyond this temporal variability is the variability across nodes and workloads. If the scheduler assigns memory hungry tasks on a memory-constrained node, stragglers could occur due to contention for memory. While for another node that has slower disk, straggler behavior will primarily depend on how many I/O-intensive tasks are being run. To predict straggler behavior, we need to consider both the current state of each node and tailor the decision to the node type (its resource capabilities) and particular kind of tasks being executed (their resource requirements). In our experiments, we find that models that do not take this into account do about 6-7% worse than models that do (see Table 5 in Section 5.3).

Due to these difficulties in understanding the causes behind stragglers, and the challenges involved in predicting stragglers, initial attempts at mitigating stragglers have been *reactive* (Dean and Ghemawat, 2004; Zaharia et al., 2008; Ananthanarayanan et al., 2010). Dean and Ghemawat (2004) suggested *speculative execution* as a mitigation mechanism for stragglers. This is a reactive scheme that is dominantly used on production clusters including those at Facebook and Microsoft Bing (Ananthanarayanan et al., 2014). It operates in two steps: (1) wait-and-speculate if a task is executing slower than other tasks of the same job, and (2) replicate or spawn multiple redundant copies of such tasks hoping a copy will reach completion before the original. As soon as one of the copies or the original task finishes execution, the rest are killed.

The benefits of Speculative execution, in terms of improved job completion times at the cost of resources consumed, are unclear. Due to the wait-and-speculate step, this scheme is inefficient in time, leading to a delay before speculation begins. Also, due to the second

2. Please see Chen et al. (2012) for details on the faithful replay of production level traces.

| Trace | % of speculatively executed tasks that were killed |
|---|---|
| Facebook 2009 (FB2009) | 77.9% |
| Facebook 2010 (FB2010) | 88.6% |
| Cloudera's Customer b (CC_b) | 74.4% |
| Cloudera's Customer e (CC_e) | 48.8% |

Table 1: Majority of the speculatively executed tasks eventually get killed since the original task finishes execution before them.

step that replicates tasks, such mechanisms lead to increased resource consumption without necessarily gaining performance benefits. Moreover, we observed that the original task, that was marked as a straggler in the wait-and-speculate step, often finishes execution before any of the redundant copies launched in the second step. Thus, a significant number of such redundant copies end up getting killed, resulting in wastage of resources and perhaps additional unnecessary contention. Table 1 shows that about 48% to 88% of the speculatively executed copies were killed in our replay of Facebook and Cloudera's customers' production cluster traces. LATE (Zaharia et al., 2008) improves over speculative execution using a notion of progress scores, but still results in resource wastage due to replication.

Since reactive mechanisms fall short of efficiently mitigating stragglers, some proactive approaches have been proposed. Dolly (Ananthanarayanan et al., 2013) is a cloning mechanism that avoids the wait-and-speculate phase of speculative execution and immediately launches redundant copies of tasks belonging to small jobs. However, being replication-based, it also incurs resource overhead.

Machine learning has shown promise in dealing with the challenges of estimating task completion times and predicting stragglers in parallel processing environments (Bortnikov et al., 2012; Gupta et al., 2013). We built Wrangler (Yadwadkar et al., 2014) (see Section 3), a system that learns to predict nodes that might create stragglers and uses these predictions as hints to the scheduler so as to avoid creating stragglers by rejecting bad placement decisions. Thus, being proactive, Wrangler is time efficient. Also, by smarter scheduling, we avoid replication of straggler tasks. Thus, Wrangler is also efficient in terms of reducing the resources consumed. Wrangler was the first complete system that demonstrated the utility of learning methods in reducing job-completion times in real world clusters.

In the following sections, we review Wrangler, describe its architecture, discuss its limitations and avenues for improvements. Then in Section 4, we present our multi-task learning based approach that improves upon Wrangler by resolving its limitations.

## 3. Wrangler

Wrangler is a system that predicts stragglers based on cluster resource usage counters and uses these predictions to inform scheduling decisions. Wrangler achieves this by adding two main components to the scheduling system of data intensive computational frameworks: (i) model-builder, and (ii) predictive scheduler. Figure 1 summarizes Wrangler's architecture. The model-builder learns to predict stragglers using cluster resource usage counters. We explain this component in more detail in the following Sections 3.1, 3.2, and 3.3. Then, every time a task is scheduled to run on a particular node, Wrangler makes a prediction

using these models, for whether the task will become a straggler. It then uses this prediction to modify the scheduling decisions if straggler behavior is predicted. Section 3.4 provides a brief explanation of the predictive scheduler.

### 3.1 Features and labels

To predict whether scheduling a task at a particular node will lead to straggler behavior, Wrangler uses the resource usage counters at the node. Dean and Ghemawat (2004) mentioned that stragglers could arise due to various reasons such as competition for CPU, memory, local disk, network bandwidth. Zaharia et al. (2008) further suggest that stragglers could be caused due to faulty hardware and misconfiguration. Ananthanarayanan et al. (2010) report that the dynamically changing resource contention patterns on an underlying node could give rise to stragglers. Based on these findings, we collected the performance counters for CPU, memory, disk, network, and other operating system level counters describing the degree of concurrency before launching a task on a node. The counters we collected span multiple broad categories as follows:

1. *CPU utilization:* CPU idle time, system and user time and speed of the CPU, etc.
2. *Network utilization:* Number of bytes sent and received, statistics of remote read and write, statistics of RPCs, etc.
3. *Disk utilization:* The local read and write statistics from the datanodes, amount of free space, etc.
4. *Memory utilization:* Amount of virtual, physical memory available, amount of buffer space, cache space, shared memory space available, etc.
5. *System-level features:* Number of threads in different states (waiting, running, terminated, blocked, etc.), memory statistics at the system level.

In total, we collect 107 distinct features characterizing the state of the machine. See Section 5.1 for details of our dataset.

**Features:** Multiple tasks from jobs of each workload may be run on each node. Therefore to simplify notation, we index the execution of a particular task by $i$ and define $S_{n,l}$ as the set of tasks corresponding to workload $l$ executed on node $n$. *Before* executing task $i \in S_{n,l}$ corresponding to workload $l$ on node $n$ we collect the resource usage counters described above on node $n$ to form the feature vector $x_i \in \mathbb{R}^{107}$. For each feature described above we subtract the minimum across the entire dataset and rescale so that it lies between 0 and 1 for the entire dataset.

**Labels:** After running task $i$ we measure the normalized task duration $nd(i)$ which is the ratio of task execution time to the amount of work done (bytes read/written) by task $i$. From the normalized duration, we determine whether a task has *straggled* using the following definition:

**Definition 1** *A task $i$ of a job $J$ is called a straggler if*

$$nd(i) > \beta \times \underset{\forall j \in J}{\text{median}}\{nd(j)\} \tag{1}$$

*where $nd(i)$ is the normalized duration of task $i$ computed as the ratio of task execution time to the amount of work done (bytes read/written) by task $i$.*

In this paper, as in Wrangler, we set $\beta$ to 1.3. Given the definition of a straggler, for task $i$ we define $y_i \in \{0, 1\}$ as a binary label indicating whether the corresponding task $i$ ended up being a straggler relative to other tasks in the same job.

### 3.2 Prediction Task

While the resource usage counters do track the time-varying state of the node, they do not model the variability across nodes, or the properties of the particular task we are executing. To deal with the variability of nodes, Wrangler builds a separate predictor for each node.

Modeling the variability across tasks is harder since it would require understanding the code that the task is executing. Instead, Wrangler uses the notion of "workloads", which we define next. Companies such as Facebook, Google, use compute clusters for various computational purposes. The specific pattern of execution of jobs on these clusters is called a *workload*. These workloads are specified using various statistics, such as submission times of multiple jobs, number of their constituent tasks along with their input data sizes, shuffle sizes, and output sizes. Wrangler assumes that all tasks in a particular workload have similar properties in terms of resource requirements etc., and it captures the variability across workloads by building separate predictors for each workload. Thus Wrangler builds separate predictors for each node and for every workload.

Putting it all together, we can state the binary classification problem for predicting straggler tasks more formally as follows. A datapoint in our setting corresponds to a task $i$ of job $J$ from a particular workload $l$ that is executed on a node $n$ in our cluster. Before running task $i$ we collect the features $x_i$ which characterize the state of node $n$. After running task $i$ we then measure the normalized duration and determine whether the task straggled with respect to other tasks of job $J$ (see Definition 1). Our goal is to learn a function:

$$f_{n,l} : x \longrightarrow y,$$

for each node $n$ and workload $l$ that maps the current characteristics $x \in \mathbb{R}^d$ of that node (e.g., current CPU utilization) to the binary variable $y \in \{0, 1\}$ indicating whether that task will straggle (i.e., run longer than $\beta$ times the median runtime).

### 3.3 Training

For any workload $l$, Wrangler first collects data and ground truth labels for training and validation. In particular, for every task $i$ launched on node $n$, Wrangler records both the resource usage counters $x_i$ and the label $y_i$ which indicates if the task straggles or not. Since there is a separate predictor for each node and workload, Wrangler produces separate datasets for each node and workload. Let $S_{n,l}$ be the set of tasks of jobs corresponding to workload $l$, executed on node $n$. Thus, we record the dataset for node $n$, workload $l$ as:

$$D_{n,l} = \{(x_i, y_i) : i \in S_{n,l}\}.$$

Then Wrangler divides each dataset into a training set and test set temporally, i.e, the first few jobs constitute the train set and the rest form the validation. The predictors are

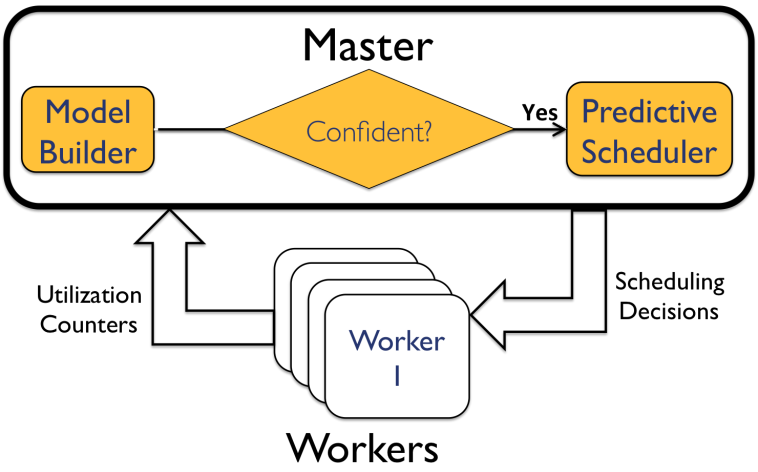


Figure 1: Architecture of *Wrangler*.

then trained on these datasets. Stragglers, by definition are fewer than the non-straggler tasks. Therefore for training, Wrangler statistically oversampled the class of stragglers to avoid the skew in the strengths of the two classes. This gets the two classes represented equally.

After this initial training phase, the classifiers are frozen, and the trained classifiers are incorporated into the job scheduler as described above. They are then tested on the next $T_{test} = 10$ hours by measuring the impact of these classifiers on job completion times. Further details about the train and test splits are in Section 5.

### 3.4 Model-aware scheduling

Job scheduling in Hadoop is handled by a master, which controls the workers. The master assigns tasks to the worker nodes. The assignments depend upon the number of available slots as well as data-locality. Wrangler modifies this scheduler to incorporate its predictions. Before launching a task $i$ of a job coming from a workload $l$, on a node $n$, the scheduler collects the node's resource usage counters $x_i$ and runs the classifier $f_{n,l}$. If the classifier predicts that the task will be a straggler with high enough "confidence" (Wrangler uses SVMs with scaling by Platt (1999) to produce a confidence), the scheduler does not assign the task to that node. It is later assigned to a node that is not predicted to create a straggler. See Yadwadkar et al. (2014) for details on the algorithm and implementation.

### 3.5 Shortcomings of Wrangler and avenues for improvements

Due to the heterogeneity of nodes in a cluster, the model builder trains a separate classifier for each node. Note that to build a training set per node, every node should have executed sufficient number of tasks. Wrangler takes a few hours (approximately 2-4 hours, depending on the workload) for this process. Additionally, because each workload might be different, these models are retrained for every new workload. Thus, for every new workload that is executed on the cluster, there is a 2-4 hour model building period. In typical large

production clusters with tens of thousands of nodes, it might be a long time before a node collects enough data to train a classifier. (In Table 5 we show that the prediction accuracy of Wrangler rapidly degrades as the amount of training data is reduced). In some cases, workloads may only be run a few times in total, limiting the ability of systems like Wrangler to make meaningful predictions.

Alternatively, large clusters with locality aware scheduling can lead to poor sample coverage. Recall that, in our case, each task of a workload executed on a node amounts to a training data point. The placement of input data on nodes in a cluster is managed by the underlying distributed file system (Ghemawat et al., 2003). To achieve locality for faster reading of input data, sophisticated locality-aware schedulers (Zaharia et al., 2008, 2010) try to assign tasks to nodes already having the appropriate data. Based on the popularity of the data, number of tasks assigned to a node could vary. Hence, we may not get uniform number of training data points, i.e., tasks executed, across all the nodes in a cluster. There could be other reasons behind skewed assignment of tasks to nodes (Kwon et al., 2012): even when every map task has the same amount of data, a task may take longer depending on the code path it takes based on the data it processes. Hence, the node slots will be busy due to such long running tasks. This could lead to some nodes executing fewer tasks than others.

These observations suggest that our modeling framework needs to be robust to limited and potentially skewed data. Thus, there is a need for straggler prediction models (1) that can be trained using minimum available data, and (2) that generalize to unseen nodes or workloads. In the following section, we describe our multi-task learning based approach with these goals for avoiding stragglers. We evaluate it using real world production level traces from Facebook and Cloudera's customers, and compare the gains with Wrangler.

## 4. Multi-task learning for straggler avoidance

As described in the previous section, Wrangler builds separate models for each workload and for every node. Thus, every {node, workload} tuple is a separate learning problem. However, learning problems corresponding to different workloads executed on the same node clearly have something in common, as do learning tasks corresponding to different nodes executing the same workload. We want to use this shared structure between the learning problems to reduce data collection time.

Concretely, a task executing on a node will be a straggler because of a combination of factors. Some of these factors involve the properties of the node where the task is executing (for instance, the node may be memory-constrained) and some others involve particular requirements that the tasks might have in terms of resources (for instance, the task may require a lot of memory). These are workload-related factors. When collecting data for a new workload executing on a given node, one must be able to use information about the workload collected while it executed on other nodes, and information collected about the node collected while it executed other workloads.

We turn to multi-task learning to leverage this shared structure. In the terminology of multi-task learning, each {node, workload} pair forms a separate learning-task[3] and these learning problems have a shared structure between them. However, unlike typical MTL formulations, our learning-tasks are not simply correlated with each other; they share a specific structure, clustering along node- or workload-dependent axes. In what follows, we first describe a general MTL formulation that can capture such learning-task grouping. We then detail how we apply this formulation to our application of straggler avoidance.

### 4.1 Partitioning tasks into groups

Suppose there are $T$ learning tasks, with the training set for the $t$-th learning-task denoted by $D_t = \{(\mathbf{x}_{it}, y_{it}) : i = 1, \ldots, k_t\}$, with $\mathbf{x}_{it} \in \mathbb{R}^d$. We begin with the formulation proposed by Evgeniou and Pontil (2004). Evgeniou, et al. proposed a basic hierarchical regression formulation with the linear model, $\mathbf{w}_t$, for learning-task $t$ as:

$$\mathbf{w}_t = \mathbf{w}_0 + \mathbf{v}_t \tag{2}$$

The intuition here is that $\mathbf{w}_0$ captures properties common to all learning-tasks, while $\mathbf{v}_t$ captures how the individual learning-tasks deviate from $\mathbf{w}_0$.

We then frame learning in the context of empirical loss minimization. Given the variability in node behavior and the need for robust predictions we adopt the hinge-loss and apply $l_2$ regularization to both the base $w_0$ and learning-task specific weights $v_t$:

$$\min_{\mathbf{w}_0, \mathbf{v}_t, b} \lambda_0 \|\mathbf{w}_0\|^2 + \frac{\lambda_1}{T} \sum_{t=1}^{T} \|\mathbf{v}_t\|^2 + \sum_{t=1}^{T} \sum_{i=1}^{k_t} L_{Hinge} \left( y_{it}, (\mathbf{w}_0 + \mathbf{v}_t)^T \mathbf{x}_{it} + b \right) \tag{3}$$

where $L_{Hinge}(y_{it}, s_{it}) = \max(0, 1 - y_{it} s_{it})$ is the hinge loss.

In the above formulation, all learning-tasks are treated equivalently. However, as discussed above, in our application some learning-tasks naturally cluster together. Suppose that the learning-tasks cluster into $G$ non-overlapping *groups*, with the $t$-th learning-task belonging to the $g(t)$-th group. Note that while we derive our formulations for non-overlapping groups, which is true in our application, the modification for overlapping groups is trivial. Using the same intuition as Equation 2, we can write the classifier $\mathbf{w}_t$ as:

$$\mathbf{w}_t = \mathbf{w}_0 + \mathbf{v}_t + \mathbf{w}_{g(t)} \tag{4}$$

In general, there may be more than one way of dividing our learning-tasks into groups. In our application, one may split learning-tasks into groups based on workload or based on nodes. We call one particular way of dividing learning-tasks into groups a *partition*. The $p$-th partition has $G_p$ groups, and the learning-task $t$ belongs to the $g_p(t)$ group under this partition. Now, we also have a separate set of weight vectors for each partition $p$, and the weight vector of the $g$-th group of the $p$-th partition is denoted by $\mathbf{w}_{p,g}$. Then, we can write the classifier $\mathbf{w}_t$ as:

$$\mathbf{w}_t = \mathbf{w}_0 + \mathbf{v}_t + \sum_{p=1}^{P} \mathbf{w}_{p,g_p(t)} \tag{5}$$

3. The machine learning notion of a "task" as a learning problem differs from the cloud computing notion of a "task" as a part of a job that is run in parallel. The intended meaning should be clear from the context.

Finally, note that $\mathbf{w}_0$ and $\mathbf{v}_t$ can also be seen as weight vectors corresponding to trivial partitions: $\mathbf{w}_0$ corresponds to the partition where all learning-tasks belong to a single group, and $\mathbf{v}_t$ corresponds to the partition where each learning-task is its own group. Thus, we can include $\mathbf{w}_0$ and $\mathbf{v}_t$ in our partitions and write Equation 5 as:

$$\mathbf{w}_t = \sum_{p=1}^{P} \mathbf{w}_{p,g_p(t)} \tag{6}$$

Intuitively, at test time, we get the classifier for the $t$-th learning-task by summing weight vectors corresponding to each group to which $t$ belongs.
As in Equation 3, the learning problem involves minimizing the sum of $l_2$ regularizers on each of the weight vectors and the hinge loss:

$$\min_{\mathbf{w}_{p,g},b} \sum_{p=1}^{P}\sum_{g=1}^{G_p} \lambda_{p,g}\|\mathbf{w}_{p,g}\|^2 + \sum_{t=1}^{T}\sum_{i=1}^{k_t} L_{Hinge}\left(y_{it}, \left(\sum_{p=1}^{P} \mathbf{w}_{p,g_p(t)}\right)^T \mathbf{x}_{it} + b\right) \tag{7}$$

Here, the regularizer coefficient $\lambda_{p,g} = \frac{\lambda_p \#(p,g)}{T}$, where $\#(p,g)$ denotes the number of learning-tasks assigned to the $g$-th group of the $p$-th partitioning. The scaling factor $\frac{\lambda_p \#(p,g)}{T}$ interpolates smoothly between $\lambda_0$, when all learning-tasks belong to a single group, and $\frac{\lambda_1}{T}$, when each learning-task is its own group. Lowering $\lambda_p$ for a particular partition will reduce the penalty on the weights of the $p$-th partition and thus cause the model to rely more on the $p$-th partitioning. For the base partition $p = 0$, setting $\lambda_p = 0$ would thus favor as much parameter sharing as feasible.

### 4.2 Reduction to a standard SVM

One advantage of the formulation we use is that it can be reduced to a standard SVM (Cortes and Vapnik, 1995), allowing the usage of off-the-shelf SVM solvers. Below, we show how this reduction can be achieved. Given $\lambda$, for every group $g$ of every partition $p$, define:

$$\tilde{\mathbf{w}}_{p,g} = \sqrt{\frac{\lambda_{p,g}}{\lambda}}\mathbf{w}_{p,g} \tag{8}$$

Now concatenate these vectors into one large weight vector $\tilde{\mathbf{w}}$ :

$$\tilde{\mathbf{w}} = [\tilde{\mathbf{w}}_{1,1}^T, \ldots, \tilde{\mathbf{w}}_{p,g}^T, \ldots, \tilde{\mathbf{w}}_{P,G_P}^T]^T \tag{9}$$

Then, it can be seen that $\lambda\|\tilde{\mathbf{w}}\|^2 = \sum_{p=1}^{P}\sum_{g=1}^{G_p} \lambda_{p,g}\|\mathbf{w}_{p,g}\|^2$. Thus, with this change of variables, the regularizer in our optimization problem resembles a standard SVM. Next, we transform the data points $\mathbf{x}_{it}$ into $\phi(\mathbf{x}_{it})$ such that we can replace the scoring function with $\tilde{\mathbf{w}}^T\phi(\mathbf{x}_{it})$. This transformation is as follows. Again, define:

$$\phi_{p,g}(\mathbf{x}_{it}) = \delta_{g_p(t),g}\sqrt{\frac{\lambda}{\lambda_{p,g}}}\mathbf{x}_{it} \tag{10}$$

Here, $\delta_{g_p(t),g}$ is a kronecker delta, which is 1 if $g_p(t) = g$ (i.e. , if the learning-task $t$ belongs to group $g$ in the $p$-th partitioning) and 0 otherwise. Our feature transformation is then the concatenation of all these vectors:

$$\phi(\mathbf{x}) = [\phi_{1,1}(\mathbf{x})^T, \ldots, \phi_{p,g}(\mathbf{x})^T, \ldots, \phi_{P,G_P}(\mathbf{x})^T]^T \tag{11}$$

It is easy to see that:

$$\tilde{\mathbf{w}}^T \phi(\mathbf{x}_{it}) = \left( \sum_{p=1}^{P} \mathbf{w}_{p,g_p(t)} \right)^T \mathbf{x}_{it} \tag{12}$$

Intuitively, $\tilde{\mathbf{w}}$ concatenates all our parameters with their appropriate scalings into one long weight vector, with one block for every group of every partitioning. $\phi(\mathbf{x}_{it})$ transforms a data point into an equally long feature vector, by placing scaled copies of $\mathbf{x}_{it}$ in the appropriate blocks and zeros everywhere else.

With these transformations, we can now write our learning problem as :

$$\min_{\tilde{\mathbf{w}},b} \lambda \|\tilde{\mathbf{w}}\|^2 + \sum_{t=1}^{T} \sum_{i=1}^{k_t} L_{Hinge}(y_{it}, \tilde{\mathbf{w}}^T \phi(\mathbf{x}_{it}) + b) \tag{13}$$

which corresponds to a standard SVM. In practice, we use this transformation and change of variables, both at train time and at test time.

### 4.3 Automatically selecting groups or partitions

In many real world applications including our own, good classification accuracy is not an end in itself. Model interpretability is of critical importance. A large powerful classifier working on tons of features may do a very good job of classification, but will not provide any insight to an engineer trying to identify the root causes of a problem. It will also be difficult to debug such a classifier if and when it fails. In the absence of sufficient training data, large models may also overfit. Thus, interpretability and simplicity of the model are important goals. In the context of the formulation above, this means that we want to keep only a minimal set of groups/partitions while maintaining high classification accuracy.

We can use mixed $l_1$ and $l_2$ norms to induce a sparse selection of groups or partitions (Bach et al., 2011). Briefly, suppose we are given a long weight vector $\mathbf{w}$ divided into $M$ blocks, with the $m$-th block denoted by $\mathbf{w}_m$. Then the squared mixed $l_1$ and $l_2$ norm is:

$$\Omega(\mathbf{w}) = \left( \sum_{m=1}^{M} \|\mathbf{w}_m\|_2 \right)^2 \tag{14}$$

This can be considered as an $l_1$ norm on a vector with elements $\|\mathbf{w}_m\|_2$. When $\Omega(\cdot)$ is used as a regularizer, the $l_1$ norm will force some elements of this vector to be set to 0, which in turn would force the corresponding *block* of weights $\mathbf{w}_m$ to be set to 0. It thus encourages entire blocks of the weight vector to be set to 0, a sparsity pattern that is often called "group sparsity".

In our context, our weight vector $\tilde{\mathbf{w}}$ is made up of $\tilde{\mathbf{w}}_{p,g}$. We consider two alternatives:

1. We can put all the weight vectors corresponding to a single partition in the same block. Then, the regularizer becomes:

$$\Omega_{ps}(\tilde{\mathbf{w}}) = \left( \sum_{p=1}^{P} \|\tilde{\mathbf{w}}_p\|_2 \right)^2 \tag{15}$$

where $\tilde{\mathbf{w}}_p = [\tilde{\mathbf{w}}_{p,1}^T, \ldots, \tilde{\mathbf{w}}_{p,G_p}^T]^T$ concatenates all weight vectors corresponding to the $p$-th partition. This regularizer will thus tend to kill entire partitions. In other words, it will uncover notions of grouping, or learning-task similarity, that are the most important.

2. We can put the weight vector of each group of each partition in separate blocks. Then the regularizer becomes:

$$\Omega_{gs}(\tilde{\mathbf{w}}) = \left( \sum_{p=1}^{P} \sum_{g=1}^{G} \|\tilde{\mathbf{w}}_{p,g}\|_2 \right)^2 \tag{16}$$

This regularizer will tend to select a small set of groups which are needed to get a good performance.

### 4.4 Automatically selecting features

Mixed norms can also be used to select a sparse set of feature blocks that are most useful for classification. This is again useful for interpretability. In our application, features are resource counters at a node. Some of these features are related to memory, others to cpu usage, etc. If our model is able to predict straggler behavior solely on the basis of memory usage counters, for instance, then it might indicate that the cluster or the workload is memory constrained.

In our multi-task setting, such feature selection can also interact with the different groups and partitions of learning-tasks. Suppose each feature vector $\mathbf{x}$ is made up of blocks corresponding to different kinds of features, denoted by $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots \mathbf{x}^{(B)}$. We can similarly divide each weight vector $\tilde{\mathbf{w}}_{p,g}$ into blocks denoted by $\tilde{\mathbf{w}}_{p,g}^{(1)}$, $\tilde{\mathbf{w}}_{p,g}^{(2)}$ and so on. Then we have two alternatives.

1. One can concatenate corresponding feature blocks from all the weight vectors to get $\tilde{\mathbf{w}}^{(1)}, \ldots, \tilde{\mathbf{w}}^{(B)}$. Then a mixed $l_1$ and $l_2$ regularizer using these weight blocks can be written as:

$$\Omega_{fs1}(\tilde{\mathbf{w}}) = \left( \sum_{b=1}^{B} \|\tilde{\mathbf{w}}^{(b)}\|_2 \right)^2 \tag{17}$$

Such a regularizer will encourage the model to select a sparse set of feature blocks on which to base its decision, setting *all* weights corresponding to all the other feature blocks to 0.

2. An alternative would be to let each group vector choose its own sparse set of feature blocks. This can be achieved with the following regularizer:

$$\Omega_{fs2}(\tilde{\mathbf{w}}) = \left( \sum_{p,g} \sum_{b=1}^{B} \|\tilde{\mathbf{w}}_{p,g}^{(b)}\|_2 \right)^2 \tag{18}$$

### 4.5 Kernelizing the formulation

We have till now described our formulation in primal space. However, kernelizing this formulation is simple. First, note that if we use a simple squared $l_2$ regularizer, then our formulation is equivalent to a standard SVM in a transformed feature space. This transformed feature space corresponds to a new kernel:

$$\begin{aligned} K_{new}(\mathbf{x}_{it}, \mathbf{x}_{ju}) &= \langle \phi(\mathbf{x}_{it}), \phi(\mathbf{x}_{ju}) \rangle \\ &= \sum_{p,g} \langle \phi_{p,g}(\mathbf{x}_{it}), \phi_{p,g}(\mathbf{x}_{ju}) \rangle \\ &= \sum_{p,g} \delta_{g_p(t),g} \delta_{g_p(u),g} \frac{\lambda}{\lambda_{p,g}} K(\mathbf{x}_{it}, \mathbf{x}_{jt}) \end{aligned} \tag{19}$$

Thus, the transformed kernel corresponds to the linear combination of a set of kernels, one for each group. Each group kernel is zero unless both data points belong to the group, in which case it equals a scaled version of the kernel in the original feature space.

When using the mixed-norm regularizers, the derivation is a bit more involved. We first note that (Aflalo et al., 2011):

$$\left( \sum_{m=1}^{M} \|\mathbf{w}_m\|_2 \right)^2 = \min_{0 \le \eta \le 1, \|\eta\|_1 \le 1} \sum_{m=1}^{M} \frac{\|\mathbf{w}_m\|_2^2}{\eta_m} \tag{20}$$

Using this transformation leads to the following primal objective:

$$\min_{\mathbf{w}, b, 0 \le \eta \le 1, \|\eta\|_1 \le 1} \sum_{m=1}^{M} \frac{\|\mathbf{w}_m\|_2^2}{\eta_m} + \sum_{i,t} L(x_{it}, y_{it}, \mathbf{w}) \tag{21}$$

where $L(x_{it}, y_{it}, \mathbf{w})$ is the hinge loss. This corresponds to a 1-norm multiple kernel learning formulation (Kloft et al., 2011) where each block of feature weights corresponds to a separate kernel. We refer the reader to Kloft et al. (2011) for a description of the dual of this formulation.

We note that these kernelized versions are very similar to the ones derived in (Widmer et al., 2010; Blanchard et al., 2011). However, the group and partition structure and the application domain are unique to ours.

### 4.6 Application to straggler avoidance

We apply this formulation to straggler avoidance as follows. Suppose there are $N$ nodes and $L$ workloads. Then there are $N \times L$ learning-tasks, and Wrangler trains as many models, one for each {node, workload} tuple. For our proposal, we consider four different partitions:

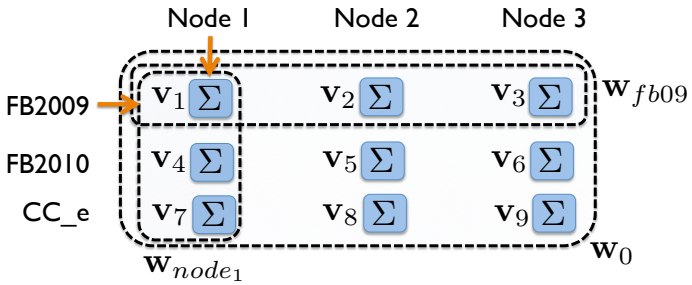


Figure 2: In our context of straggler avoidance, the learning-tasks naturally cluster into various groups in multiple partitions. When a particular learning- task, for example, node 1 and workload FB2009 ($\mathbf{v}_1$), has limited training data available, we learn its weight vector, $\mathbf{w}_1$, by adding the weight vectors of groups it belongs to from different partitions.

1. A single group consisting of all nodes and workloads. This gives us the single weight vector $\mathbf{w}_0$.

2. One group for each node, consisting of all $L$ learning-tasks belonging to that node. This gives us one weight vector for each node $\mathbf{w}_n, n = 1, \ldots, N$, that captures the heterogeneity of nodes.

3. One group for each workload, consisting of all $N$ learning-tasks belonging to that workload. This gives us one weight vector for each workload $\mathbf{w}_l, l = 1, \ldots, L$, that captures peculiarities of particular workloads.

4. Each {node, workload} tuple as its own group. Since there are $N \times L$ such pairs, we get $N \times L$ weight vectors, which we denote as $\mathbf{v}_t$, following the notation considered in Evgeniou and Pontil (2004).

Thus, if we use all four partitions, the weight vector $\mathbf{w}_t$ for a given workload $l_t$ and a given node $n_t$ is:

$$\mathbf{w}_t = \mathbf{w}_0 + \mathbf{w}_{n_t} + \mathbf{w}_{l_t} + \mathbf{v}_t \tag{22}$$

Figure 2 shows an example. The learning problem for the FB2009[4] workload running on node 1 belongs to one group from each of the four partitions mentioned above: (1) the global partition, denoted by the weight vector, $\mathbf{w}_0$, (2) the group corresponding to node 1 from the node-wise partition, denoted by the weight vector $\mathbf{w}_{node_1}$, (3) the group corresponding to the FB2009 workload from the workload-wise partition, denoted by the weight vector $\mathbf{w}_{fb09}$, and (4) the group containing just this individual learning problem, denoted by the weight vector $\mathbf{v}_1$. Thus, we can learn the weight vector $\mathbf{w}_1$ as:

$$\mathbf{w}_1 = \mathbf{w}_0 + \mathbf{w}_{node_1} + \mathbf{w}_{fb09} + \mathbf{v}_1 \tag{23}$$

The corresponding training problem is then:

4. See Section 5.1 for details about the workloads we use.

$$\min_{\mathbf{w},b} \lambda_0 \|\mathbf{w}_0\|^2 + \frac{\nu}{N}\sum_{n=1}^{N}\|\mathbf{w}_n\|^2 + \frac{\omega}{L}\sum_{l=1}^{L}\|\mathbf{w}_l\|^2 + \frac{\tau}{T}\sum_{t=1}^{T}\|\mathbf{v}_t\|^2 + \sum_{t=1}^{T}\sum_{i=1}^{k_t} L_{Hinge}\left(y_{it}, \left(\mathbf{w}_0 + \mathbf{w}_{n_t} + \mathbf{w}_{l_t} + \mathbf{v}_t\right)^T \mathbf{x}_{it} + b\right) \quad (24)$$

where $\lambda_0$, $\nu$, $\omega$, $\tau$ are hyperparameters. We ran an initial grid search on a validation set to fix these hyperparameters and found that prediction accuracy was not very sensitive to these settings: several settings gave very close to optimal results. We used $\lambda_0 = \nu = \omega = \tau = 1$, which was one of the highest performing settings, for all our experiments. Appendix A provides the details of this grid search experiment along with the sensitivity of these hyperparameters to a set of values. As described in Section 4.2, we work in a transformed space in which the above problem reduces to a standard SVM. In this space the weight vector is

$$\tilde{\mathbf{w}} = [\tilde{\mathbf{w}}_0^T, \tilde{\mathbf{w}}_{n_1}^T, \ldots, \tilde{\mathbf{w}}_{n_N}^T, \tilde{\mathbf{w}}_{l_1}^T, \ldots, \tilde{\mathbf{w}}_{l_L}^T, \tilde{\mathbf{v}}_{t_1}^T, \ldots, \tilde{\mathbf{v}}_{t_T}^T]^T \quad (25)$$

This decomposition will change depending on which partitions we use.

As described in the previous section, we can also use mixed $l_1$ and $l_2$ norms to automatically select groups or partitions. To select partitions, we combine all the node-related weight vectors $\tilde{\mathbf{w}}_{n_1}, \ldots, \tilde{\mathbf{w}}_{n_N}$ into one long vector $\tilde{\mathbf{w}}_N$, all workload vectors $\tilde{\mathbf{w}}_{l_1}, \ldots, \tilde{\mathbf{w}}_{l_L}$ into $\tilde{\mathbf{w}}_L$ and all $\tilde{\mathbf{v}}_1, \ldots, \tilde{\mathbf{v}}_T$ into $\tilde{\mathbf{v}}_T$. We then solve the following optimization:

$$\min_{\tilde{\mathbf{w}},b} \left(\|\tilde{\mathbf{w}}_0\|_2 + \|\tilde{\mathbf{w}}_N\|_2 + \|\tilde{\mathbf{w}}_L\|_2 + \|\tilde{\mathbf{v}}_T\|_2\right)^2 + C\sum_{i,t} L_{Hinge}\left(y_{it}, \tilde{\mathbf{w}}^T\phi\left(\mathbf{x}_{it}\right) + b\right) \quad (26)$$

This model will learn which notion of grouping is most important. For instance if $\tilde{\mathbf{w}}_L$ is set to 0, then we might conclude that straggler behaviour doesn't depend that much on the particular workload being executed.

To select individual groups, we solve the optimization problem:

$$\min_{\tilde{\mathbf{w}},b,\xi\geq 0} \left(\|\tilde{\mathbf{w}}_0\|_2 + \sum_{n=1}^{N}\|\tilde{\mathbf{w}}_n\|_2 + \sum_{l=1}^{L}\|\tilde{\mathbf{w}}_l\|_2 + \sum_{t=1}^{T}\|\tilde{\mathbf{v}}_t\|_2\right)^2 + C\sum_{i,t} L_{Hinge}\left(y_{it}, \tilde{\mathbf{w}}^T\phi\left(\mathbf{x}_{it}\right) + b\right) \quad (27)$$

This formulation can set some individual node or workload models to 0. This would mean that straggler behavior on some nodes or workloads can be predicted by generic models, but others require more node-specific or workload-specific reasoning.

We can also use mixed norms for feature selection. The features in our feature vector correspond to resource usage counters, and we divide them into 5 categories, as explained

in Section 3.1: counters based on cpu, those based on network, those based on disk, those based on memory, and other system-level counters. Then each $\tilde{\mathbf{w}}_n$, $\tilde{\mathbf{w}}_l$ and $\tilde{\mathbf{v}}_t$ gets similarly split up. As described before, we can either let each model choose its own set of features using this regularizer:

$$\begin{aligned}\Omega(\tilde{\mathbf{w}}) \quad &= (\|\tilde{\mathbf{w}}_0^{mem}\|_2 + \|\tilde{\mathbf{w}}_0^{disk}\|_2 + \|\tilde{\mathbf{w}}_0^{cpu}\|_2 + \|\tilde{\mathbf{w}}_0^{network}\|_2 + \|\tilde{\mathbf{w}}_0^{system}\|_2 + \\ &+ \textstyle\sum_{n=1}^{N}[\|\tilde{\mathbf{w}}_n^{mem}\|_2 + \|\tilde{\mathbf{w}}_n^{disk}\|_2 + \|\tilde{\mathbf{w}}_n^{cpu}\|_2 + \|\tilde{\mathbf{w}}_n^{network}\|_2 + \|\tilde{\mathbf{w}}_n^{system}\|_2] \\ &+ \textstyle\sum_{l=1}^{L}[\|\tilde{\mathbf{w}}_l^{mem}\|_2 + \|\tilde{\mathbf{w}}_l^{disk}\|_2 + \|\tilde{\mathbf{w}}_l^{cpu}\|_2 + \|\tilde{\mathbf{w}}_l^{network}\|_2 + \|\tilde{\mathbf{w}}_l^{system}\|_2] \\ &+ \textstyle\sum_{t=1}^{T}[\|\tilde{\mathbf{v}}_t^{mem}\|_2 + \|\tilde{\mathbf{v}}_t^{disk}\|_2 + \|\tilde{\mathbf{v}}_t^{cpu}\|_2 + \|\tilde{\mathbf{v}}_t^{network}\|_2 + \|\tilde{\mathbf{v}}_t^{system}\|_2])^2 \end{aligned} \tag{28}$$

or choose a single set of features globally using this regularizer:

$$\Omega(\tilde{\mathbf{w}}) = \left(\|\tilde{\mathbf{w}}^{mem}\|_2 + \|\tilde{\mathbf{w}}^{cpu}\|_2 + \|\tilde{\mathbf{w}}^{disk}\|_2 + \|\tilde{\mathbf{w}}^{network}\|_2 + \|\tilde{\mathbf{w}}^{system}\|_2\right)^2 \tag{29}$$

where $\tilde{\mathbf{w}}^{mem}$ concatenates all memory related weights from all models,

$$\tilde{\mathbf{w}}^{mem} = [\tilde{\mathbf{w}}_0^{memT}, \tilde{\mathbf{w}}_{n_1}^{memT}, \ldots, \tilde{\mathbf{w}}_{n_N}^{memT}, \tilde{\mathbf{w}}_{l_1}^{memT}, \ldots, \tilde{\mathbf{w}}_{l_L}^{memT}]^T \tag{30}$$

$\tilde{\mathbf{w}}^{cpu}$, $\tilde{\mathbf{w}}^{disk}$ etc. are defined similarly.

### 4.7 Exploring the relationships between the weight vectors

Before getting into the experiments, we can get some insights on what our formulation will learn by looking at the KKT conditions. Equation 24 can equivalently be written as:

$$\begin{aligned}\min_{\mathbf{w},b,\xi} \lambda_0\|\mathbf{w}_0\|^2 + \frac{\nu}{N}\sum_{n=1}^{N}\|\mathbf{w}_n\|^2 + \frac{\omega}{L}\sum_{l=1}^{L}\|\mathbf{w}_l\|^2 \\ + \frac{\tau}{T}\sum_{t=1}^{T}\|\mathbf{v}_t\|^2 + \sum_{t=1}^{T}\sum_{i=1}^{k_t}\xi_{it} \\ \text{s.t. } y_{it}\left(\left(\mathbf{w}_0 + \mathbf{w}_{n_t} + \mathbf{w}_{l_t} + \mathbf{v}_t\right)^T\mathbf{x}_{it} + b\right) \geq 1 - \xi_{it} \;\; \forall i,t \\ \xi_{it} \geq 0 \;\; \forall i,t\end{aligned} \tag{31}$$

The Lagrangian of the formulation in Equation 31 is:

$$\begin{aligned}\mathcal{L}(\mathbf{w},b,\boldsymbol{\alpha},\boldsymbol{\gamma}) = \lambda_0\|\mathbf{w}_0\|^2 + \frac{\nu}{N}\sum_{n=1}^{N}\|\mathbf{w}_n\|^2 + \frac{\omega}{L}\sum_{l=1}^{L}\|\mathbf{w}_l\|^2 \\ + \frac{\tau}{T}\sum_{t=1}^{T}\|\mathbf{v}_t\|^2 + \sum_{t=1}^{T}\sum_{i=1}^{k_t}\xi_{it} - \sum_{t=1}^{T}\sum_{i=1}^{k_t}\gamma_{it}\xi_{it} \\ + \sum_{t=1}^{T}\sum_{i=1}^{k_t}\alpha_{it}\left(1 - \xi_{it} - y_{it}\left(\mathbf{w}_t^T\mathbf{x}_{it} + b\right)\right)\end{aligned} \tag{32}$$

Taking derivatives w.r.t the primal variables and setting to 0 gives us relationships between $\mathbf{w}_0, \mathbf{v}_t, \mathbf{w}_n$ and $\mathbf{w}_l$:

$$\lambda_0 \mathbf{w}_0^* = \frac{\tau}{T} \sum_t \mathbf{v}_t^* \tag{33}$$

$$\nu \mathbf{w}_n^* = \frac{\tau}{T/N} \sum_{t:n_t=n} \mathbf{v}_t^* \tag{34}$$

$$\omega \mathbf{w}_l^* = \frac{\tau}{T/L} \sum_{t:l_t=l} \mathbf{v}_t^* \tag{35}$$

$$\lambda_0 \mathbf{w}_0^* = \frac{\nu}{N} \sum_n \mathbf{w}_n^* \tag{36}$$

$$\lambda_0 \mathbf{w}_0^* = \frac{\omega}{L} \sum_l \mathbf{w}_l^* \tag{37}$$

Evgeniou and Pontil (2004) also obtain Equation 33 in their formulation, but the other relationships are specific to ours. These relationships imply that these variables shouldn't be considered independent. $\mathbf{w}_n$, $\mathbf{w}_l$ and $\mathbf{w}_0$ are scaled means of the $\mathbf{v}_t$'s of the group they capture.

### 4.8 Generalizing to unseen nodes and workloads

Consider what happens when we remove the partition corresponding to individual {node, workload} tuples, i.e., $\mathbf{v}_t$, from our formulations. We now do not have any parameters specific to a node-workload combination, but can still capture both node- and workload-dependent properties of the learning problem. Such formulations are thus similar to factorized models where the node and workload dependent factors are grouped into separate blocks. We end up with only $(N + L)d$ parameters, whereas a formulation like that of (Evgeniou and Pontil, 2004; Evgeniou et al., 2005; Jacob et al., 2009) will still have $NLd$ parameters (here $d$ is the input dimensionality). Thus, we can *reduce* the number of parameters while still capturing the essential properties of the learning problems.

In addition, since we no longer have a separate weight vector for each {node, workload} tuple, we can generalize to node-workload pairs that are completely unseen at train time: the classifier for such an unseen combination $t$ will simply be $\mathbf{w}_0 + \mathbf{w}_{n_t} + \mathbf{w}_{l_t}$. We thus explicitly use knowledge gleaned from prior workloads run on this node (through $\mathbf{w}_{n_t}$) and other nodes running this workload (through $\mathbf{w}_{l_t}$). This is especially useful in our application where there may be a large number of nodes and workloads. In such cases, collecting data for each node-workload pair will be time consuming, and generalizing to unseen combinations will be a significant advantage.

In most of our experiments, therefore, we remove the partition corresponding to individual {node, workload} tuples. We explicitly evaluate how well we generalize by doing so in Section 5.4.

## 5. Empirical Evaluation

In this section, we describe our dataset, provide variants to our proposed formulation and then evaluate them using the following metrics: first, classification accuracy when there is

| Trace | #Machines | Length | Date | #Jobs |
|---|---|---|---|---|
| *FB2009* | 600 | 6 month | 2009 | 1129193 |
| *FB2010* | 3000 | 1.5 months | 2010 | 1169184 |
| *CC_b* | 300 | 9 days | 2011 | 22974 |
| *CC_e* | 100 | 9 days | 2011 | 10790 |
| Total | $\approx$ 4000 | $\approx$ 8.5 months | - | 2332141 |

Table 2: Dataset. *FB*: Facebook, *CC*: Cloudera Customer.

sufficient data and also when sufficient data is not available, and second, improvement in overall job completion times, and third, reduction in resources consumed.

### 5.1 Datasets

The set of real-world workloads considered in this paper are collected from the production compute clusters at Facebook and Cloudera's customers, which we denote as $FB2009$, $FB2010$, $CC\_b$ and $CC\_e$. Table 2 provides details about these workloads in terms of the number of machines in the actual clusters, the length and date of data capture, total number of jobs in those workloads. Chen, et al., explain the data in further details in (Chen et al., 2012). Together, the dataset consists of traces from over about 4000 machines captured over almost eight months. For faithfully replaying these real-world production traces on our 20 node EC2 cluster, we used a statistical workload replay tool, SWIM (Chen et al., 2011) that synthesizes a workload with representative job submission rates and patterns, shuffle/input data size and output/shuffle data ratios (see Chen et al. (2011) for details of replay methodology). SWIM scales the workload to the number of nodes in the experimental cluster.

For each workload, we need data with ground-truth labels for training and validating our models. We collect this data by running tasks from the workload as described above and recording the resource usage counters $x_i$ at a node at the time a task $i$ is launched, and the ground truth label $y_i$ by checking if it ends up becoming a straggler. Then we divide this dataset temporally into a training set and a validation set. In other words, the first few tasks that were executed form the train set and the rest of the tasks form the validation set. In the experiments below, we vary the percentage of data that is used for training, and compute the prediction accuracy on the validation set. We train our final model using two-thirds of this dataset and proceed to evaluate it on our ultimate metric, i.e., job completion times.

To measure job completion times, we then incorporate the trained models into the job scheduler (as in Wrangler). We then run the replay for the workload again, but with a fresh set of tasks. These fresh set of tasks form our test set, and this test set is only used to measure job completion times. Table 3 shows the sizes of the datasets.

Each data point is represented by a 107 dimensional feature vector comprising the node's resource usage counters at the time of launching a task on it. We optimize all our formulations using Liblinear (Fan et al., 2008) for the $l_2$ regularized variants and using the algorithm proposed by Aflalo et al. (2011) (modified to work in the primal) for the mixed norm variants.

| Workload | No of tasks (Training+Validation) | No of tasks Test |
|---|---|---|
| $FB2009$ | 4885 | 13632 |
| $FB2010$ | 3843 | 38158 |
| $CC\_b$ | 5991 | 30203 |
| $CC\_e$ | 39014 | 94550 |

Table 3: Number of tasks we use for each workload in the train+val and test sets.

Below, we describe (1) how we use different MTL formulations and prediction accuracy achieved by these formulations, (2) how we learn a classifier for previously unseen node and/or workload and the prediction accuracy it achieves, (3) the improvement in overall job completion times achieved by our formulation and Wrangler over speculative execution, and (4) reduction in resources consumed using our formulation compared to Wrangler.

### 5.2 Variants of proposed formulation

We consider several variants of the general formulation described in Section 4. Using a simple squared $l_2$ regularizer, we first consider $\mathbf{w}_0$, $\mathbf{w}_n$ and $\mathbf{w}_l$, individually:

- $f_0$: In this formulation, we consider only the global partition in which all learning problems belong to a single group. This corresponds to removing $\mathbf{v}_t$, $\mathbf{w}_n$ and $\mathbf{w}_l$. This formulation thus learns a single global weight vector, $\mathbf{w}_0$, for all the nodes and all the workloads.

- $f_n$: Here we consider only the partition based on nodes. This corresponds to only learning a $\mathbf{w}_n$, that is, one model for each node. This model learns to predict stragglers based on a node's resource usage counters across different workloads, but it cannot capture any properties that are specific to a particular workload.

- $f_l$: Here we consider only the partition based on workloads. This means we only learn $\mathbf{w}_l$, i.e., a workload dependent model across nodes executing a particular workload. This model learns to predict stragglers based on the resource usage pattern caused due to a workload across nodes, but ignores the characteristics of a specific node.

The above three formulations either discard the node information, the workload information, or both. We now consider multi-task variants that capture both, node and workload properties:

- $f_{0,t}$: This is the formulation proposed by Evgeniou and Pontil (2004), and corresponds to using the global partition where all learning-tasks belong to one group, and the partition where each learning-task is its own group. This learns $\mathbf{w}_0$ and $\mathbf{v}_t$. Note that this formulation still has to learn on the order of $NLd$ different parameters, and has to collect enough data to learn a separate weight vector for each {node, workload} combination.

- $f_{0,t,l}$: This formulation extends the formulation in $f_{0,t}$ by additionally adding the partition based on workloads. It learns $\mathbf{w}_0, \mathbf{w}_l$ and $\mathbf{v}_t$.

- $f_{0,n,l}$: We remove the partition corresponding to individual {node, workload} tuples, removing $\mathbf{v}_t$ entirely and only learning $\mathbf{w}_0$, $\mathbf{w}_l$ and $\mathbf{w}_n$. As described in Section 4.8, this formulation reduces the total number of parameters to $(N + L)d$ and can also generalize to unseen {node, workload} tuples.

For all these formulations, the hyperparameters $\lambda_0$, $\nu$, $\omega$ and $\tau$ were set to 1 wherever applicable. We found this setting to be close to optimal in our initial cross-validation experiments (see Appendix A).

In addition to these, we also consider sparsity-inducing formulations for automatically selecting partitions or groups or blocks of features, as explained in Sections 4.3, 4.4, and 4.6.

- $f_{ps}$ : We use the global, node-based and workload-based formulations thus removing $\mathbf{v}_t$ entirely as in $f_{0,n,l}$ and use mixed $l_1$ and $l_2$ norms to automatically select the useful partitions from among these. This model will learn the notion of grouping that is most important. To select partitions, we combine all the node-related weight vectors $\tilde{\mathbf{w}}_{n_1}, \ldots, \tilde{\mathbf{w}}_{n_N}$ into one long vector $\tilde{\mathbf{w}}_N$ and all workload vectors into $\tilde{\mathbf{w}}_L$ as shown below:

$$\tilde{\mathbf{w}} = [\tilde{\mathbf{w}}_0^T, \underbrace{\tilde{\mathbf{w}}_{n_1}^T, \ldots, \tilde{\mathbf{w}}_{n_N}^T}_{\tilde{\mathbf{w}}_N^T}, \underbrace{\tilde{\mathbf{w}}_{l_1}^T, \ldots, \tilde{\mathbf{w}}_{l_L}^T}_{\tilde{\mathbf{w}}_L^T}]^T \tag{38}$$

  Thus our weight vector $\tilde{\mathbf{w}}$ is split up into blocks corresponding to node, workload and global models.

- $f_{gs}$ : This is the formulation where we use mixed $l_1$ and $l_2$ norms to automatically select groups within a partition. Again, we only consider the global, node-based and workload-based formulations. This formulation can set individual node or workload models to zero, unlike $f_{ps}$, that can set a complete partition, i.e. in our case, a combined model of all the nodes or all the workloads to zero. This would mean that predicting straggler behavior on some nodes or workloads does not need reasoning that is specific to those nodes or workloads; instead a generic model would work.

Finally, we also try the following two mixed norms formulations for feature selection. As before, both these formulations remove the partition corresponding to individual {node, workload} tuples, i.e., remove $\mathbf{v}_t$.

- $f_{fs1}$ : As explained in Equation 28, we divide our features into five categories corresponding to cpu, memory, disk, network and other system-level counters. Then each weight vector gets similarly split up into these categories. This formulation learns which categories of features are more important for some nodes or workloads than others.

- $f_{fs2}$ : This formulation, as given in Equation 29, selects a category of features across all the weight vectors of nodes and workloads. This formulation can learn if stragglers in a cluster are caused due to contention for a specific resource.

### 5.3 Prediction accuracy

In this section, we evaluate the formulations described in the previous section for their straggler prediction accuracy. In the following subsection 5.3.1, we evaluate formulations that use $l_2$ regularizers, viz., $f_0$, $f_n$, $f_l$, $f_{0,n,l}$, $f_{0,t}$, and $f_{0,t,l}$. Then, in Section 5.3.2, we evaluate the mixed norm formulations (a) that automatically selects partitions, $f_{ps}$, (b) that automatically selects groups, $f_{gs}$, and then the formulations (c) $f_{fs1}$, and (d) $f_{fs2}$ that can automatically select features. We list our formulations with a brief description in Table 4.

#### 5.3.1 Formulations with $l_2$ regularizers

We aim at learning to predict stragglers using as small amount of data as feasible, as this means shorter data capture time. Note that stragglers are fewer than non-stragglers, so we oversample from the stragglers' class to represent the two classes equally in both, the training and validation sets[5]. Table 5 shows the percentage accuracy of predicting stragglers with varying amount of training data. We observe that:

- With very small amounts of data, all MTL variants outperform Wrangler. In fact, all of $f_0$ to $f_{0,t,l}$ need only one sixth of the training data to achieve the same or better accuracy.
- It is important to capture both node- and workload-dependent aspects of the problem: $f_{0,n,l}$, $f_{0,t}$ and $f_{0,t,l}$ consistently outperform $f_0$, $f_n$ and $f_l$.
- $f_{0,t}$ and $f_{0,t,l}$ perform up to 7 percentage points better than Wrangler with the same amount of training data, with $f_{0,n,l}$ not far behind.

For a better visualization, Figure 3 shows the comparison of prediction accuracy of these formulations, in terms of percentage true positives and percentage false positives when 50% of total data is available.

Next, we evaluate and discuss the sparsity-inducing formulations, $f_{ps}$ (Equation 26), $f_{gs}$ (Equation 27), $f_{fs1}$ (Equation 28), and $f_{fs2}$ (Equation 29).

#### 5.3.2 Formulations with mixed $l_1$ and $l_2$ norms

**Automatically selecting partitions or groups:** In this Section, we evaluate $f_{ps}$ and $f_{gs}$. Table 6 shows the prediction accuracy of these formulations compared to $f_0$, $f_n$, $f_l$ and $f_{0,n,l}$. $f_{ps}$ and $f_{gs}$ show comparable prediction accuracy to $f_{0,n,l}$. We also found interesting sparsity patterns in the learnt weight vectors.

- $f_{ps}$ : Recall that $f_{ps}$ attempts to set the weights of entire partitions to 0. In our case we have a global partition, a node-based partition and a workload-based partition. We observed that only $\tilde{\mathbf{w}}_0$ is zero in the resulting weight vector learned. This means that given node-specific and workload-specific factors, the global factors that are common across all the nodes and workloads do not contribute to the prediction. In other words, similar accuracy could be achieved without using $\tilde{\mathbf{w}}_0$. However, both node- and workload-dependent weights are necessary.

5. An alternative to statistical oversampling would be to use class-sensitive miss-classification penalties.

| Formulation | Description |
|---|---|
| $f_0$ | uses a single, global weight vector |
| $f_n$ | uses only node-specific weights |
| $f_l$ | uses only workload-specific weights |
| $f_{0,n,l}$ | uses global, node- and workload-specific weights |
| $f_{0,t}$ | uses global weights and weights specific to {node, workload} tuples |
| $f_{0,t,l}$ | uses global and workload-specific weights and weights specific to {node, workload} tuples |
| $f_{ps}$ | selects partitions automatically |
| $f_{gs}$ | selects groups automatically |
| $f_{fs1}$ | selects feature-blocks automatically for individual groups |
| $f_{fs2}$ | selects feature-blocks automatically across all the groups |

Table 4: Brief description of all our formulations.

| %Training Data | Wrangler Yadwadkar et al. (2014) | $f_0$ | $f_n$ | $f_l$ | $f_{0,n,l}$ | $f_{0,t}$ | $f_{0,t,l}$ |
|---|---|---|---|---|---|---|---|
| 1 | Insufficient data | 66.9 | 63.5 | 66.5 | 65.5 | 63.7 | 66.2 |
| 2 | Insufficient Data | 67.1 | 63.3 | 67.7 | 67.5 | 64.3 | 67.7 |
| 5 | Insufficient Data | 67.5 | 68.1 | 69.1 | 69.8 | 69.6 | 69.1 |
| 10 | 63.9 | 67.8 | 70.9 | 69.4 | 72.3 | 73.1 | 72.9 |
| 20 | 67.2 | 68.0 | 72.6 | 70.1 | 72.9 | 74.7 | 74.8 |
| 30 | 68.5 | 68.5 | 73.2 | 70.3 | 74.1 | 75.9 | 75.8 |
| 40 | 69.7 | 68.2 | 73.9 | 70.5 | 74.3 | 76.4 | 76.4 |
| 50 | 70.1 | 68.5 | 74.1 | 70.4 | 75.3 | 77.1 | 77.2 |

Table 5: Prediction accuracies (in %) of various MTL formulations for straggler prediction with varying amount of training data. See Section 5.3.1 for details.

- $f_{gs}$ : In $f_{gs}$, we encourage individual nodes-specific or individual workloads-specific weight vectors separately to be set to zero. We observed that some of the nodes' weight vectors and some of the workload-specific weight vectors were zero, indicating that in some cases we do not need node or workload specific reasoning. (One can use the learnt sparsity pattern and attempt to correlate it with some node and workload characteristics; however we have not explored this in this paper.)

We also note that our mixed-norm formulations automatically learn a grouping that achieves comparable accuracy with lesser total number of groups, and thus fewer parameters.

**Automatically selecting features:** In this section, we evaluate the remaining formulations $f_{fs1}$ and $f_{fs2}$. These two formulations group sets of features based on resources. We divide the features in five different categories viz., features measuring (1) CPU utilization, (2) memory utilization, (3) network usage, (5) disk utilization, (6) other system level performance counters. We evaluate their straggler prediction accuracy and then discuss their interpretability in terms of understanding the causes behind stragglers.

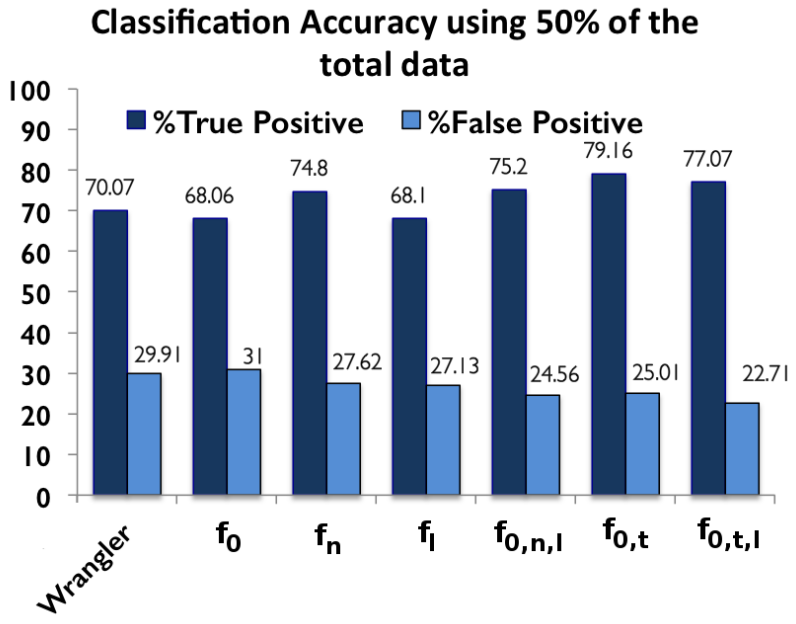


Figure 3: Classification accuracy of various MTL formulations as compared to Wrangler using 50% of the total data. This plot shows the percentage of true positives and the percentage of false positives in each of the cases. These quantities are computed as: % True Positive = (fraction of stragglers predicted correctly as stragglers) $\times$ 100, and % False Positive = (fraction of non-stragglers predicted incorrectly to be stragglers) $\times$ 100.

| Formulation | Straggler Prediction Accuracy |
|---|---|
| $f_0$ | 68.5 |
| $f_n$ | 74.1 |
| $f_l$ | 70.4 |
| $f_{0,n,l}$ | 75.3 |
| $f_{ps}$ | 74.4 |
| $f_{gs}$ | 73.8 |

Table 6: Straggler prediction accuracies (in %) using the four mixed-norm formulations $f_{ps}$ and $f_{gs}$ compared with formulations that use $l_2$ regularizers. Note that $f_{ps}$ and $f_{gs}$ perform with comparable accuracy with $f_{0,n,l}$, however, use lesser number of groups and parameters, resulting in simpler models.

Table 7 shows the percentage prediction accuracies of these formulations on our test set. Note that these formulations show comparable prediction accuracy. Next, we discuss the impact of $f_{fs1}$ and $f_{fs2}$ on understanding the straggler behavior.

- $f_{fs1}$: Because this formulation divides each group weight vector further into blocks based on the kind of features, it can potentially provide fine-grained insight into what kinds of features are most important for each group weight vector. Indeed, we found that some node models assign zero weight to features from the network category, while others assign a zero weight to the disk category. However, no global patterns emerge.

| Formulation | Straggler Prediction Accuracy (in %) |
|---|---|
| $f_{fs1}$ | 74.9 |
| $f_{fs2}$ | 74.9 |

Table 7: Straggler prediction accuracies (in %) using $f_{fs1}$ and $f_{fs2}$ that encourage sparsity across bocks of features.

| FB2009 | | FB2010 | | CC_b | | CC_e | |
|---|---|---|---|---|---|---|---|
| $f_{0,n,l}$ | $f_{0,t}$ | $f_{0,n,l}$ | $f_{0,t}$ | $f_{0,n,l}$ | $f_{0,t}$ | $f_{0,n,l}$ | $f_{0,t}$ |
| 73.1 | 45.3 | 46.7 | 48.3 | 50.2 | 49.4 | 52.8 | 68.2 |
| 56.2 | 57.5 | 57.3 | 58.7 | 61.0 | 53.5 | 64.4 | 48.9 |
| 63.9 | 55.5 | 50.0 | 48.8 | 59.4 | 53.4 | 48.9 | 65.1 |
| 63.2 | 47.7 | 60.6 | 57.4 | 55.7 | 49.5 | 47.3 | 73.9 |
| 50.7 | 42.4 | 51.4 | 56.2 | 50.8 | 44.6 | 71.2 | 59.9 |

Table 8: Straggler Prediction accuracies (in %) of $f_{0,n,l}$ and $f_{0,t}$ on test data from an unseen node-workload pair. See Section 5.4 for details.

This reinforces our belief that the causes of stragglers vary quite a bit from node to node or workload to workload.

- $f_{fs2}$: This formulation considers these feature categories across the various nodes and workloads and provides a way of knowing if there are certain dominating factors causing stragglers in a cluster. However, we observed that none of the feature categories had zero weights in the weight vector learned for our dataset. Again, this means that there is no single, easily discoverable reason for straggler behavior, and provides evidence to the claim made by Ananthanarayanan et al. (2013, 2014) that the causes behind stragglers are hard to figure out.

### 5.4 Prediction accuracy for a {node, workload} tuple with insufficient data

One of our goals in this work is to reduce the amount of training data required to get a straggler prediction model up and running. When a new workload begins to execute on a node, we want to learn a model as quickly as possible. Recall that (Section 5.3) with enough training data available we found that $f_{0,n,l}$, $f_{0,t}$ and $f_{0,t,l}$ seem to perform similarly, with $f_{0,n,l}$ performing slightly worse. However, as mentioned in Section 4.8, formulation $f_{0,n,l}$ has fewer parameters and, because it has no weight vector specific to a particular {node, workload} tuple, can generalize to new {node, workload} tuples unseen at train time. This is in contrast to $f_{0,t}$ which has to fall back on $\mathbf{w}_0$ in such a situation, and thus may not generalize as well. In this section, we see if this is indeed true.

We trained classifiers based on $f_{0,n,l}$ and $f_{0,t}$ leaving out 95% of the data of one node-workload pair every time. We then test the models on the left-out data. Table 8 shows the percentage classification accuracy from 20 such runs. We note the following:

- For 13 out of 20 classification experiments, $f_{0,n,l}$ performs better than $f_{0,t}$. For 10 out of these 13 cases, the difference in performance is more than 5 percentage points.
- For workloads $FB2009$ and $CC\_b$, we see $f_{0,n,l}$ performs better consistently.

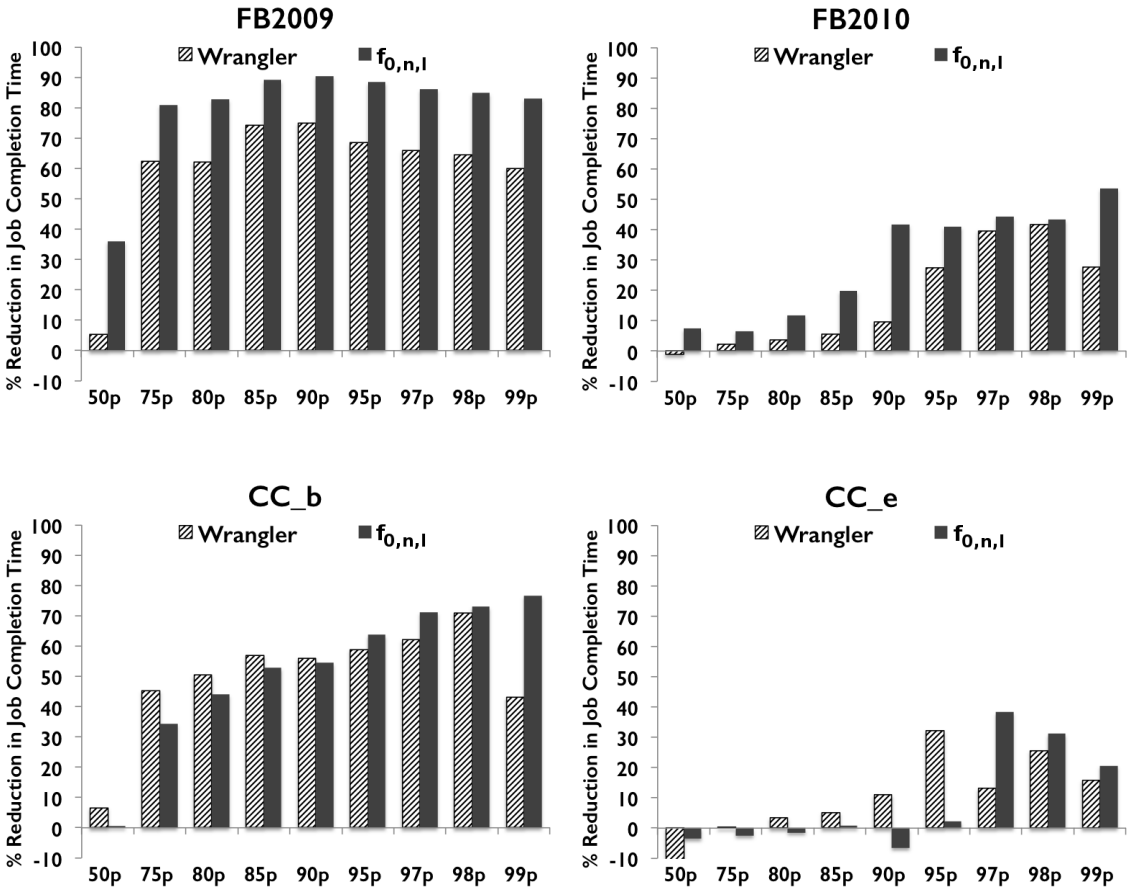


Figure 4: Improvement in the overall job completion times achieved by $f_{0,n,l}$ and Wrangler over speculative execution.

- $f_{0,n,l}$ sometimes performs worse, but in only 3 of these cases is it significantly worse (worse by more than 5 percentage points). All 3 of these instances are in case of the $CC\_e$ workload. In general, for this workload, we also notice a huge variance in the numbers obtained across multiple nodes. See Yadwadkar et al. (2014), for a discussion of some of the issues in this workload.

This shows that $f_{0,n,l}$ works better in real-world settings where one cannot expect enough data for all node-workload pairs. Therefore, we evaluate $f_{0,n,l}$ in our next experiment (Section 5.5) to see if it improves job completion times.

### 5.5 Improvement in overall job completion time

We now evaluate our formulation, $f_{0,n,l}$, using the second metric, improvement in the overall job completion times over speculative execution. We compare these improvements to that achieved by Wrangler (Figure 4). Improvement at the $99^{th}$ percentile is a strong indicator of the effectiveness of straggler mitigation techniques. We see that $f_{0,n,l}$ significantly

| Workload | % Reduction in total task-seconds | |
|---|---|---|
| | (MTL with $f_{0,n,l}$) | (Wrangler) |
| FB-2009 | 73.33 | 55.09 |
| FB-2010 | 8.9 | 24.77 |
| CC_b | 64.12 | 40.15 |
| CC_e | 13.04 | 8.24 |

Table 9: Resource utilization with $f_{0,n,l}$ and with *Wrangler* over speculative execution, in terms of total task execution times (in seconds) across all the jobs. $f_{0,n,l}$ reduces resources consumed over Wrangler for $FB2009$, $CC\_b$ and $CC\_e$.

improves over Wrangler, reflecting the improvements in prediction accuracy. At the $99^{th}$ percentile, we improve Wrangler's job completion times by $57.8\%, 35.8\%, 58.9\%$ and $5.7\%$ for $FB2009, FB2010, CC\_b$ and $CC\_e$ respectively. Note that Wrangler is already a strong baseline. Hence, the improvement in job completion times on top of the improvement achieved by Wrangler is significant.

### 5.6 Reduction in resources consumed

When a job is launched on a cluster, it will be broken into small tasks and these tasks will be run in a distributed fashion. Thus, to calculate the resources used, we can sum the resources used by all the tasks. As in Yadwadkar et al. (2014), we use the time taken by each task as a measure of the resources used by the task. Note that, because these tasks will likely be executing in parallel, the total time taken by the tasks will be much larger than the time taken for the whole job to finish, which is what job completion time measures (shown in Figure 4). Ideally, straggler prediction will prevent tasks from becoming stragglers. Fewer stragglers means fewer tasks that need to be replicated by straggler mitigation mechanisms (like speculative execution) and thus lower resource consumption. Thus, improved straggler prediction should also reduce the total task-seconds i.e., resources consumed.

Table 9 compares the percentage reduction in resources consumed in terms of total task-seconds achieved by $f_{0,n,l}$ and Wrangler over speculative execution. We see that the improved predictions of $f_{0,n,l}$ reduce resource consumption significantly more than Wrangler for 3 out of 4 workloads, thus supporting our intuitions. In particular, for $FB2009$ and $CC\_b$, $f_{0,n,l}$ reduces Wrangler's resource consumption by about 40%, while for $CC\_e$ the reduction is about 5%.

## 6. Related Work on Multi-task Learning

The idea that multiple learning problems might be related and can gain from each other dates back to Thrun (1996) and Caruana (1993). They pointed out that humans do not learn a new task from scratch but instead reuse knowledge gleaned from other learning tasks. This notion was formalized by, among others, Baxter (2000) and Ando and Zhang (2005), who quantified this gain. Much of this early work relied on neural networks as a means of learning these shared representations. However, contemporary work has also focused on SVMs and kernel machines.

Our work is an extension of the work of Evgeniou and Pontil (2004), who proposed an additive model for MTL that decomposes classifiers into a shared component and a task-

specific component. In later work, Evgeniou et al. (2005) propose an MTL framework that uses a general quadratic form as a regularizer. They show that if the tasks can be grouped into clusters, they can use a regularizer that encourages all the weight vectors of the group to be closer to each other. Jacob et al. (2009) extend this formulation when the group structure is not known a priori. Xue et al. (2007) infer the group structure using a Bayesian approach. The approach of Widmer et al. (2010) is similar to ours and groups tasks into "meta-tasks", and tries to automatically figure out the meta-tasks required to get good performance. The formulation we propose is also designed to handle group structure, but allows us to dispense with task-specific classifiers entirely, reducing the number of parameters drastically. This allows us to handle tasks that have very little training data by transferring parameters learnt on other tasks. Our formulation shares this property with that of Blanchard et al. (2011), and indeed our basic formulation can be written down in the kernel-based framework they describe for learning to generalize onto a completely unseen task. Other ways of controlling parameters include learning a distance metric (Parameswaran and Weinberger, 2010), and using low rank regularizers (Pong et al., 2010).

Our setting is an example of multilinear multitask learning where each learning problem is indexed by two indices: the node and the workload. Previous work on this subfield of multitask learning has typically used low-rank regularizers on the weight matrix represented as tensors (Romera-Paredes et al., 2013; Wimalawarne et al., 2014). It is also possible to define a similarity between tasks based on how many indices they share (Signoretto et al., 2014). Our formulation captures some of the same intuitions, but has the added advantage of simplicity and ease of implementation.

Using mixed norms for inducing sparsity has a rich history. Donoho (2006) showed that minimizing the $l_1$ norm recovers sparse solutions when solving linear systems. When used as a regularizer, the $l_1$ norm learns sparse models, where most weights are 0. The most well known of such sparse formulations is the lasso (Tibshirani, 1996), which uses the $l_1$ norm to select features in regression. Yuan and Lin (2006) extend the lasso to group lasso, where they use a mixed $l_1$ and $l_2$ norm to select a sparse set of *groups* of features. Bach (2008) study the theoretical properties of group lasso. Since these initial papers, mixed norms have found use in a variety of applications. For instance , Quattoni et al. (2008) use a mixed $l_\infty$ and $l_1$ norm for feature selection. Such mixed norms also show up in the literature on kernel learning, where they are used to select a sparse set of kernels (Varma and Ray, 2007; Kloft et al., 2011) or a sparse set of groups of kernels (Aflalo et al., 2011). Bach et al. (2011) provides an accessible review of mixed norms and their optimization, and we direct the interested reader to that article for more details.

## 7. Conclusion

Through this work, we have shown the utility of multitask learning in solving the real-world problem of avoiding stragglers in distributed data processing. We have presented a novel MTL formulation that captures the structure of our learning-tasks and reduces job completion times by up to 59% over prior work (Yadwadkar et al., 2014). This reduction comes from a 7 percentage point increase in prediction accuracy. Our formulation can achieve better accuracy with only a sixth of the training data and can generalize better than other MTL approaches for learning-tasks with little or no data. We have also presented extensions

to our formulation using group sparsity inducing mixed norms that automatically discover the structure of our learning tasks and make the final model more interpretable. Finally, we note that, although we use straggler avoidance as the motivation, our formulation is more generally applicable, especially for other prediction problems in distributed computing frameworks, such as resource allocation (Gupta et al., 2013; Delimitrou and Kozyrakis, 2014).

## Appendix A. Cross-validating hyperparameter settings

Our formulation reduces the number of hyperparameters to just one per partitioning, which makes it much easier to cross-validate to set their values. In particular, our formulation in the context of straggler avoidance, Equation (24), has four hyperparameters: $\lambda_0$, $\nu$, $\omega$, $\tau$. To tune these parameters we used a simple grid search with cross-validation (results are shown in Tables 10, 11, 12, and 13). In general we found that the model formulation is relatively robust to the choice of hyperparameters so long as they are within the correct order of magnitude.

## References

Jonathan Aflalo, Aharon Ben-Tal, Chiranjib Bhattacharyya, Jagarlapudi Saketha Nath, and Sankaran Raman. Variable sparsity kernel learning. *Journal of Machine Learning Research*, 12, 2011.

Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, 2010.

Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, 2013.

Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. Grass: Trimming stragglers in approximation analytics. In *NSDI*, 2014.

Rie Kubota Ando and Tong Zhang. A framework for learning predictive structures from multiple tasks and unlabeled data. *JMLR*, 6, 2005.

Francis Bach, Rodolphe Jenatton, Julien Mairal, Guillaume Obozinski, et al. Convex optimization with sparsity-inducing norms. *Optimization for Machine Learning*, pages 19–53, 2011.

Francis R Bach. Consistency of the group lasso and multiple kernel learning. *The Journal of Machine Learning Research*, 9:1179–1225, 2008.

Jonathan Baxter. A model of inductive bias learning. *JAIR*, 12, 2000.

Gilles Blanchard, Gyemin Lee, and Clayton Scott. Generalizing from several related classification tasks to a new unlabeled sample. In *NIPS*, 2011.

| $\lambda_0$ | $\nu$ | $\omega$ | $\tau$ | Accuracy (%) |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 75.38 |
| 1 | 1 | 1 | 10 | 75.57 |
| 1 | 1 | 1 | 100 | 75.24 |
| 1 | 1 | 1 | 1000 | 74.39 |
| 1 | 1 | 10 | 1 | 74.93 |
| 1 | 1 | 10 | 10 | 75.51 |
| 1 | 1 | 10 | 100 | 75.05 |
| 1 | 1 | 10 | 1000 | 74.38 |
| 1 | 1 | 100 | 1 | 74.84 |
| 1 | 1 | 100 | 10 | 74.84 |
| 1 | 1 | 100 | 100 | 73.95 |
| 1 | 1 | 100 | 1000 | 73.04 |
| 1 | 1 | 1000 | 1 | 73.03 |
| 1 | 1 | 1000 | 10 | 72.60 |
| 1 | 1 | 1000 | 100 | 72.11 |
| 1 | 1 | 1000 | 1000 | 71.17 |
| 1 | 10 | 1 | 1 | 74.95 |
| 1 | 10 | 1 | 10 | 75.17 |
| 1 | 10 | 1 | 100 | 74.34 |
| 1 | 10 | 1 | 1000 | 74.05 |
| 1 | 10 | 10 | 1 | 75.42 |
| 1 | 10 | 10 | 10 | 75.05 |
| 1 | 10 | 10 | 100 | 74.69 |
| 1 | 10 | 10 | 1000 | 75.00 |
| 1 | 10 | 100 | 1 | 74.84 |
| 1 | 10 | 100 | 10 | 74.77 |
| 1 | 10 | 100 | 100 | 74.42 |
| 1 | 10 | 100 | 1000 | 73.53 |
| 1 | 10 | 1000 | 1 | 72.88 |
| 1 | 10 | 1000 | 10 | 72.23 |
| 1 | 10 | 1000 | 100 | 72.07 |
| 1 | 10 | 1000 | 1000 | 71.16 |

| $\lambda_0$ | $\nu$ | $\omega$ | $\tau$ | Accuracy (%) |
|---|---|---|---|---|
| 1 | 100 | 1 | 1 | 75.39 |
| 1 | 100 | 1 | 10 | 74.95 |
| 1 | 100 | 1 | 100 | 74.42 |
| 1 | 100 | 1 | 1000 | 74.00 |
| 1 | 100 | 10 | 1 | 75.58 |
| 1 | 100 | 10 | 10 | 75.14 |
| 1 | 100 | 10 | 100 | 74.92 |
| 1 | 100 | 10 | 1000 | 73.90 |
| 1 | 100 | 100 | 1 | 74.87 |
| 1 | 100 | 100 | 10 | 74.72 |
| 1 | 100 | 100 | 100 | 74.09 |
| 1 | 100 | 100 | 1000 | 72.95 |
| 1 | 100 | 1000 | 1 | 73.08 |
| 1 | 100 | 1000 | 10 | 72.15 |
| 1 | 100 | 1000 | 100 | 72.47 |
| 1 | 100 | 1000 | 1000 | 70.72 |
| 1 | 1000 | 1 | 1 | 75.47 |
| 1 | 1000 | 1 | 10 | 75.37 |
| 1 | 1000 | 1 | 100 | 74.73 |
| 1 | 1000 | 1 | 1000 | 74.49 |
| 1 | 1000 | 10 | 1 | 75.78 |
| 1 | 1000 | 10 | 10 | 74.73 |
| 1 | 1000 | 10 | 100 | 74.87 |
| 1 | 1000 | 10 | 1000 | 74.40 |
| 1 | 1000 | 100 | 1 | 74.97 |
| 1 | 1000 | 100 | 10 | 75.06 |
| 1 | 1000 | 100 | 100 | 73.68 |
| 1 | 1000 | 100 | 1000 | 72.51 |
| 1 | 1000 | 1000 | 1 | 72.29 |
| 1 | 1000 | 1000 | 10 | 72.02 |
| 1 | 1000 | 1000 | 100 | 71.94 |
| 1 | 1000 | 1000 | 1000 | 70.34 |

Table 10: Tuning the hyperparameters $\lambda_0$, $\nu$, $\omega$ and $\tau$ using grid search.

| $\lambda_0$ | $\nu$ | $\omega$ | $\tau$ | Accuracy (%) |
|---|---|---|---|---|
| 10 | 1 | 1 | 1 | 65.93 |
| 10 | 1 | 1 | 10 | 67.67 |
| 10 | 1 | 1 | 100 | 63.99 |
| 10 | 1 | 1 | 1000 | 64.67 |
| 10 | 1 | 10 | 1 | 69.97 |
| 10 | 1 | 10 | 10 | 74.92 |
| 10 | 1 | 10 | 100 | 75.13 |
| 10 | 1 | 10 | 1000 | 74.52 |
| 10 | 1 | 100 | 1 | 61.19 |
| 10 | 1 | 100 | 10 | 75.85 |
| 10 | 1 | 100 | 100 | 75.44 |
| 10 | 1 | 100 | 1000 | 75.30 |
| 10 | 1 | 1000 | 1 | 68.35 |
| 10 | 1 | 1000 | 10 | 74.20 |
| 10 | 1 | 1000 | 100 | 74.45 |
| 10 | 1 | 1000 | 1000 | 73.90 |
| 10 | 10 | 1 | 1 | 64.50 |
| 10 | 10 | 1 | 10 | 66.59 |
| 10 | 10 | 1 | 100 | 64.90 |
| 10 | 10 | 1 | 1000 | 61.98 |
| 10 | 10 | 10 | 1 | 69.95 |
| 10 | 10 | 10 | 10 | 75.61 |
| 10 | 10 | 10 | 100 | 75.02 |
| 10 | 10 | 10 | 1000 | 74.63 |
| 10 | 10 | 100 | 1 | 73.26 |
| 10 | 10 | 100 | 10 | 75.19 |
| 10 | 10 | 100 | 100 | 74.77 |
| 10 | 10 | 100 | 1000 | 74.68 |
| 10 | 10 | 1000 | 1 | 58.03 |
| 10 | 10 | 1000 | 10 | 74.83 |
| 10 | 10 | 1000 | 100 | 74.95 |
| 10 | 10 | 1000 | 1000 | 73.83 |

| $\lambda_0$ | $\nu$ | $\omega$ | $\tau$ | Accuracy (%) |
|---|---|---|---|---|
| 10 | 100 | 1 | 1 | 66.44 |
| 10 | 100 | 1 | 10 | 64.54 |
| 10 | 100 | 1 | 100 | 64.98 |
| 10 | 100 | 1 | 1000 | 62.97 |
| 10 | 100 | 10 | 1 | 68.87 |
| 10 | 100 | 10 | 10 | 75.84 |
| 10 | 100 | 10 | 100 | 75.09 |
| 10 | 100 | 10 | 1000 | 75.41 |
| 10 | 100 | 100 | 1 | 69.45 |
| 10 | 100 | 100 | 10 | 75.41 |
| 10 | 100 | 100 | 100 | 75.74 |
| 10 | 100 | 100 | 1000 | 75.30 |
| 10 | 100 | 1000 | 1 | 72.66 |
| 10 | 100 | 1000 | 10 | 75.19 |
| 10 | 100 | 1000 | 100 | 74.30 |
| 10 | 100 | 1000 | 1000 | 74.03 |
| 10 | 1000 | 1 | 1 | 67.33 |
| 10 | 1000 | 1 | 10 | 63.73 |
| 10 | 1000 | 1 | 100 | 61.84 |
| 10 | 1000 | 1 | 1000 | 60.44 |
| 10 | 1000 | 10 | 1 | 67.40 |
| 10 | 1000 | 10 | 10 | 75.60 |
| 10 | 1000 | 10 | 100 | 75.26 |
| 10 | 1000 | 10 | 1000 | 75.30 |
| 10 | 1000 | 100 | 1 | 72.45 |
| 10 | 1000 | 100 | 10 | 75.44 |
| 10 | 1000 | 100 | 100 | 75.30 |
| 10 | 1000 | 100 | 1000 | 75.03 |
| 10 | 1000 | 1000 | 1 | 66.71 |
| 10 | 1000 | 1000 | 10 | 75.00 |
| 10 | 1000 | 1000 | 100 | 75.07 |
| 10 | 1000 | 1000 | 1000 | 74.42 |

Table 11: Tuning the hyperparameters $\lambda_0$, $\nu$, $\omega$ and $\tau$ using grid search.

| $\lambda_0$ | $\nu$ | $\omega$ | $\tau$ | Accuracy (%) |
|---|---|---|---|---|
| 100 | 1 | 1 | 1 | 67.36 |
| 100 | 1 | 1 | 10 | 66.39 |
| 100 | 1 | 1 | 100 | 63.79 |
| 100 | 1 | 1 | 1000 | 65.24 |
| 100 | 1 | 10 | 1 | 65.46 |
| 100 | 1 | 10 | 10 | 67.84 |
| 100 | 1 | 10 | 100 | 64.63 |
| 100 | 1 | 10 | 1000 | 60.97 |
| 100 | 1 | 100 | 1 | 72.27 |
| 100 | 1 | 100 | 10 | 56.05 |
| 100 | 1 | 100 | 100 | 66.44 |
| 100 | 1 | 100 | 1000 | 70.18 |
| 100 | 1 | 1000 | 1 | 52.14 |
| 100 | 1 | 1000 | 10 | 71.03 |
| 100 | 1 | 1000 | 100 | 65.30 |
| 100 | 1 | 1000 | 1000 | 73.16 |
| 100 | 10 | 1 | 1 | 65.87 |
| 100 | 10 | 1 | 10 | 65.94 |
| 100 | 10 | 1 | 100 | 67.53 |
| 100 | 10 | 1 | 1000 | 64.05 |
| 100 | 10 | 10 | 1 | 69.18 |
| 100 | 10 | 10 | 10 | 62.42 |
| 100 | 10 | 10 | 100 | 62.87 |
| 100 | 10 | 10 | 1000 | 63.63 |
| 100 | 10 | 100 | 1 | 56.48 |
| 100 | 10 | 100 | 10 | 66.43 |
| 100 | 10 | 100 | 100 | 75.18 |
| 100 | 10 | 100 | 1000 | 75.57 |
| 100 | 10 | 1000 | 1 | 53.74 |
| 100 | 10 | 1000 | 10 | 69.14 |
| 100 | 10 | 1000 | 100 | 75.5 |
| 100 | 10 | 1000 | 1000 | 75.24 |

| $\lambda_0$ | $\nu$ | $\omega$ | $\tau$ | Accuracy (%) |
|---|---|---|---|---|
| 100 | 100 | 1 | 1 | 64.43 |
| 100 | 100 | 1 | 10 | 65.58 |
| 100 | 100 | 1 | 100 | 65.06 |
| 100 | 100 | 1 | 1000 | 63.42 |
| 100 | 100 | 10 | 1 | 57.59 |
| 100 | 100 | 10 | 10 | 65.14 |
| 100 | 100 | 10 | 100 | 67.08 |
| 100 | 100 | 10 | 1000 | 68.03 |
| 100 | 100 | 100 | 1 | 72.68 |
| 100 | 100 | 100 | 10 | 65.14 |
| 100 | 100 | 100 | 100 | 75.01 |
| 100 | 100 | 100 | 1000 | 75.26 |
| 100 | 100 | 1000 | 1 | 64.70 |
| 100 | 100 | 1000 | 10 | 55.27 |
| 100 | 100 | 1000 | 100 | 75.24 |
| 100 | 100 | 1000 | 1000 | 74.79 |
| 100 | 1000 | 1 | 1 | 65.79 |
| 100 | 1000 | 1 | 10 | 65.15 |
| 100 | 1000 | 1 | 100 | 60.47 |
| 100 | 1000 | 1 | 1000 | 61.25 |
| 100 | 1000 | 10 | 1 | 67.20 |
| 100 | 1000 | 10 | 10 | 63.50 |
| 100 | 1000 | 10 | 100 | 66.72 |
| 100 | 1000 | 10 | 1000 | 61.48 |
| 100 | 1000 | 100 | 1 | 67.65 |
| 100 | 1000 | 100 | 10 | 66.88 |
| 100 | 1000 | 100 | 100 | 75.71 |
| 100 | 1000 | 100 | 1000 | 75.40 |
| 100 | 1000 | 1000 | 1 | 57.90 |
| 100 | 1000 | 1000 | 10 | 69.47 |
| 100 | 1000 | 1000 | 100 | 75.91 |
| 100 | 1000 | 1000 | 1000 | 75.49 |

Table 12: Tuning the hyperparameters $\lambda_0$, $\nu$, $\omega$ and $\tau$ using grid search.

| $\lambda_0$ | $\nu$ | $\omega$ | $\tau$ | Accuracy (%) |
|---|---|---|---|---|
| 1000 | 1 | 1 | 1 | 64.42 |
| 1000 | 1 | 1 | 10 | 62.57 |
| 1000 | 1 | 1 | 100 | 65.63 |
| 1000 | 1 | 1 | 1000 | 62.00 |
| 1000 | 1 | 10 | 1 | 64.62 |
| 1000 | 1 | 10 | 10 | 65.87 |
| 1000 | 1 | 10 | 100 | 65.36 |
| 1000 | 1 | 10 | 1000 | 65.51 |
| 1000 | 1 | 100 | 1 | 71.64 |
| 1000 | 1 | 100 | 10 | 65.88 |
| 1000 | 1 | 100 | 100 | 61.27 |
| 1000 | 1 | 100 | 1000 | 71.41 |
| 1000 | 1 | 1000 | 1 | 66.75 |
| 1000 | 1 | 1000 | 10 | 74.07 |
| 1000 | 1 | 1000 | 100 | 54.60 |
| 1000 | 1 | 1000 | 1000 | 71.88 |
| 1000 | 10 | 1 | 1 | 63.93 |
| 1000 | 10 | 1 | 10 | 63.35 |
| 1000 | 10 | 1 | 100 | 63.68 |
| 1000 | 10 | 1 | 1000 | 64.02 |
| 1000 | 10 | 10 | 1 | 71.02 |
| 1000 | 10 | 10 | 10 | 65.37 |
| 1000 | 10 | 10 | 100 | 66.78 |
| 1000 | 10 | 10 | 1000 | 64.47 |
| 1000 | 10 | 100 | 1 | 56.41 |
| 1000 | 10 | 100 | 10 | 60.10 |
| 1000 | 10 | 100 | 100 | 68.24 |
| 1000 | 10 | 100 | 1000 | 61.10 |
| 1000 | 10 | 1000 | 1 | 55.50 |
| 1000 | 10 | 1000 | 10 | 66.23 |
| 1000 | 10 | 1000 | 100 | 65.71 |
| 1000 | 10 | 1000 | 1000 | 72.19 |

| $\lambda_0$ | $\nu$ | $\omega$ | $\tau$ | Accuracy (%) |
|---|---|---|---|---|
| 1000 | 100 | 1 | 1 | 63.52 |
| 1000 | 100 | 1 | 10 | 66.93 |
| 1000 | 100 | 1 | 100 | 65.36 |
| 1000 | 100 | 1 | 1000 | 64.37 |
| 1000 | 100 | 10 | 1 | 55.54 |
| 1000 | 100 | 10 | 10 | 62.08 |
| 1000 | 100 | 10 | 100 | 64.24 |
| 1000 | 100 | 10 | 1000 | 62.66 |
| 1000 | 100 | 100 | 1 | 63.72 |
| 1000 | 100 | 100 | 10 | 69.01 |
| 1000 | 100 | 100 | 100 | 61.42 |
| 1000 | 100 | 100 | 1000 | 64.31 |
| 1000 | 100 | 1000 | 1 | 60.21 |
| 1000 | 100 | 1000 | 10 | 71.92 |
| 1000 | 100 | 1000 | 100 | 67.46 |
| 1000 | 100 | 1000 | 1000 | 75.28 |
| 1000 | 1000 | 1 | 1 | 63.09 |
| 1000 | 1000 | 1 | 10 | 68.08 |
| 1000 | 1000 | 1 | 100 | 63.58 |
| 1000 | 1000 | 1 | 1000 | 59.18 |
| 1000 | 1000 | 10 | 1 | 69.81 |
| 1000 | 1000 | 10 | 10 | 68.01 |
| 1000 | 1000 | 10 | 100 | 67.49 |
| 1000 | 1000 | 10 | 1000 | 66.57 |
| 1000 | 1000 | 100 | 1 | 59.24 |
| 1000 | 1000 | 100 | 10 | 61.69 |
| 1000 | 1000 | 100 | 100 | 66.04 |
| 1000 | 1000 | 100 | 1000 | 61.54 |
| 1000 | 1000 | 1000 | 1 | 71.65 |
| 1000 | 1000 | 1000 | 10 | 58.39 |
| 1000 | 1000 | 1000 | 100 | 71.65 |
| 1000 | 1000 | 1000 | 1000 | 75.59 |

Table 13: Tuning the hyperparameters $\lambda_0$, $\nu$, $\omega$ and $\tau$ using grid search.

Edward Bortnikov, Ari Frank, Eshcar Hillel, and Sriram Rao. Predicting execution bottlenecks in map-reduce clusters. In *HotCloud*, 2012.

Rich Caruana. Multitask learning: A knowledge-based source of inductive bias. In *ICML*, 1993.

Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2011.

Yanpei Chen, Sara Alspaugh, and Randy H. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *PVLDB*, 5(12), 2012.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3), 1995.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ASPLOS*, 2014.

David L Donoho. For most large underdetermined systems of linear equations the minimal l1-norm solution is also the sparsest solution. *Communications on pure and applied mathematics*, 59(6), 2006.

Theodoros Evgeniou and Massimiliano Pontil. Regularized multi–task learning. In *KDD*, 2004.

Theodoros Evgeniou, Charles A Micchelli, Massimiliano Pontil, and John Shawe-Taylor. Learning multiple tasks with kernel methods. *JMLR*, 6(4), 2005.

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, 2008.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, 2003.

Shekhar Gupta, Christian Fritz, Bob Price, Roger Hoover, Johan Dekleer, and Cees Witteveen. Throughputscheduler: Learning to schedule on heterogeneous hadoop clusters. In *ICAC*, 2013.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

Laurent Jacob, Jean philippe Vert, and Francis R. Bach. Clustered multi-task learning: A convex formulation. In *NIPS*. 2009.

Marius Kloft, Ulf Brefeld, Sören Sonnenburg, and Alexander Zien. lp-norm multiple kernel learning. *J. Mach. Learn. Res.*, 12:953–997, July 2011. ISSN 1532-4435.

YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *SIGMOD*, 2012.

Henry Li. *Introducing Windows Azure*. Apress, Berkely, CA, USA, 2009. ISBN 143022469X, 9781430224693.

Shibin Parameswaran and Kilian Q. Weinberger. Large margin multi-task metric learning. In *NIPS*. 2010.

John Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3), 1999.

Ting Kei Pong, Paul Tseng, Shuiwang Ji, and Jieping Ye. Trace norm regularization: reformulations, algorithms, and multi-task learning. *SIAM Journal on Optimization*, 20 (6), 2010.

Ariadna Quattoni, Michael Collins, and Trevor Darrell. Transfer learning for image classification with sparse prototype representations. In *CVPR*, 2008.

Bernardino Romera-Paredes, Hane Aung, Nadia Bianchi-Berthouze, and Massimiliano Pontil. Multilinear multitask learning. In *ICML*, 2013.

M Signoretto, R Langone, M Pontil, and J Suykens. Graph based regularization for multilinear multitask learning. 2014.

Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 607–614, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0632-4. doi: 10.1145/1963405.1963491. URL `http://doi.acm.org/10.1145/1963405.1963491`.

Sebastian Thrun. Is learning the n-th thing any easier than learning the first? *NIPS*, 1996.

Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

M. Varma and D. Ray. Learning the discriminative power-invariance trade-off. In *ICCV*, October 2007.

Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596521979, 9780596521974.

Christian Widmer, Yasemin Altun, and Nora C. Toussaint. Multitask multiple kernel learning (mt-mkl), 2010.

Kishan Wimalawarne, Masashi Sugiyama, and Ryota Tomioka. Multitask learning meets tensor factorization: task imputation via convex optimization. In *NIPS*, 2014.

Ya Xue, Xuejun Liao, Lawrence Carin, and Balaji Krishnapuram. Multi-task learning for classification with dirichlet process priors. *JMLR*, 8, 2007.

Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 26:1–26:14, New York, NY, USA, 2014. ACM.

Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Eurosys*, 2010.