Parallel News-Article Traffic Forecasting with ADMM

Stratis Ioannidis[*]
ECE, Northeastern University
Boston, MA
ioannidis@ece.neu.edu

Yunjiang Jiang[*]
Google
Mountain View, CA
jyj@google.com

Saeed Amizadeh[*]
Microsoft
Redmond, WA
saamizad@microsoft.com

Nikolay Laptev
Yahoo Research
Sunnyvale, CA
nlaptev@yahoo-inc.com

ABSTRACT

Predicting the traffic of an article, as measured by page views, is of great importance to content providers. Articles with increased traffic can improve advertising revenue and expand a provider's user base. We propose a broadly applicable methodology incorporating meta-data and joint forecasting across articles, that involves solving a large optimization problem through the Alternating Directions Method of Multipliers (ADMM). We implement our solution using Spark, and evaluate it over a large corpus of articles and forecasting models. Our results demonstrate that our feature-based forecasting is both scalable as well as highly accurate, significantly improving forecasting predictions compared to traditional forecasting models.

1. INTRODUCTION

Predicting the traffic of an article, as measured by page views, is of great importance to content providers. Articles with increased traffic can improve advertising revenue and expand a provider's user base. Traffic provenance is also important: user referrals from social media or search engines may indicate that an article is going viral. Predicting future traffic from such sources can therefore aid in identifying viral articles. This, in turn, can aid decisions regarding, e.g., which articles a provider should promote on its home page, on its official Twitter account, or through display and search ads. To that end, this paper aims at devising a methodology for forecasting traffic, categorized by provenance. In particular, we wish to predict the traffic an article will receive from social media and search, as both indicate the public's interest in an article.

The above forecasting objective poses several challenges. First, although a provider may only be interested in, e.g., social traffic, the dimensions of the time series are highly correlated. For example, an increase in social traffic may lead to an increase in search traffic and vice-versa. This is something that we should try to exploit during forecasting; hence, a multi-dimensional forecasting approach is necessary.

Second, traditional forecasting techniques like, e.g., autoregressive models [3], involve basing forecasts only on past time series data: under such techniques, future traffic is predicted using past traffic spanning, e.g., the past few hours. Such forecasting methods are clearly at a disadvantage when an article is young, and there is little traffic evidence to be used to forecast its future behavior. This is very important in our case, as the first few hours of an article's life are the most crucial: due to the "24-hour news cycle" phenomenon, most articles are either picked up early on or not at all, and very few receive significant traffic after 36 hours following their appearance. In contrast to traditional forecasting in domains such as finance or weather, where the preponderance of historical data can be taken for granted, our forecasting techniques are most needed precisely when the article is nascent, and has recently been released.

Third, forecasting each article's traffic individually is ill-advised in the context of our problem because it ignores topicality. Observing, for example, that an article about an artist or a world event is gaining traction in the social media can be used in forecasting the traffic of an article with the same topic. In short, a feature-based approach utilizing article similarity seems appropriate.

To address the above issues, we propose a forecasting methodology built on both traffic traces and article features. The latter indeed allows us to implement a feature-based approach to traffic forecasting: we acheive this by regressing the parameters of traditional forecasting models over an article's features. In particular, we make the following contributions:

- We propose a broadly applicable methodology incorporating features and joint forecasting across articles. In short, our methodology can extend any forecasting method that can be expressed as a parametrized multi-dimensional time series fitting problem, to jointly training across multiple articles.
- Training jointly across thousands of articles, each associated with meta-data comprising thousands of features, amounts to a large optimization problem. We show that this problem can be solved through the Alternating Directions Method of Multipliers (ADMM), recently advocated as an exemplary method for parallel learning [5]. Our solution is highly parallelizable, and scales to thousands of machines, articles, and features.
- We implement our solution in a fully parallel manner using Spark [28]. Our implementation is highly modular and extensible: a programmer wishing to incorporate a new forecasting model needs to only implement a penalized fitting function on a single article traffic trace: our framework immediately scales this code to jointly training among multiple articles and machines using Spark.
- We extensively evaluate our method over a variety of

[*]Work done while authors were at Yahoo Research.

different forecasting models on a dataset comprising 2K articles spanning 16K features. Our results indicate that, when predicting peak traffic, our methodology yields predictions between 5 and 100 times better than traditional forecasting methods.

The measured quantities required to implement our forecasting method are readily available to content providers: page views are routinely monitored and scraped from webserver logs, and website content features are routinely extracted for advertisement purposes. In addition, traffic provenance is logged by webservers through so-called *referral* events: the page containing the link a person clicked in order to reach an article is routinely tracked.

The remainder of this paper is organized as follows. Related work is presented in Section 2, while we formally state the problem we study in Section 3. A parallel solution through ADMM is presented in Section 4. The set of traditional forecasting models we incorporate in our framework are introduced in Section 5, and our implementation over Spark is described in Section 6. Section 7 contains our experiments, and we conclude in Section 8.

2. RELATED WORK

Broadly speaking, time series forecasting techniques can be grouped into traditional machine learning approaches, such as support vector machines [23, 9] and neural networks [4, 22], specifically applied to time series forecasting, as well as approaches tailored to time series, like autoregressive models [3, 12]. Although all these models are very different in terms of their underlying assumptions, parameter estimation from trace data is often formulated as an optimization problem, e.g., through the minimization of sum of squared errors or through maximum likelihood estimation. As such, the training phase for many of these models can be incorporated into our proposed ADMM-based framework: we illustrate this in our work (c.f. Section 5), showing how several traditional models can be extended to leverage metadata by solving, in parallel, an optimization problem.

A series of recent papers have focused on modeling popularity in social media. Focusing on mentions of "memes" (short phrases and hashtags), Yang et al. [25] identify 6 canonical shapes that characterize most time series of memes in Twitter, while Chang et al. [10] propose a clustering method aimed at identifying spikes. Matsubara et al. [20] propose a parametrized time series model that spans the 6 canonical patterns detected by [25], and fits actual meme time series quite well. All of the above works observe diurnal patterns, as well as the growth and eventual decay of a meme, which is also observed in other online content [14]. Article consumption is subject to very similar patterns; we exploit this by incorporating variants of the above models in our forecasting experiments.

Though its formal guarantees have been known since the early 70's, ADMM [5] has only recently received considerable attention due to its ability to scale problems over parallel architectures. In the past few years, it has been applied with great success to a variety of large-scale problems, including matrix factorization [27], multi-dimensional anisotropic total variation problems [26], rank-SVM training [11], as well as click prediction [2]. Implementations on Hadoop [1] and Spark [16] have also been recently publicly released. Our work departs by focusing on a time series forecasting, but also on producing a nested implementation through Spark that is highly modular supporting a variety of different models, as discussed in Section 6, whereas prior implementations focused on a single learning task (e.g., logistic regression). In addition, our implementation has a distributed consensus step, whereas other implementations (e.g., [16], implemented in Scala) collect and broadcast all consensus variables from a single processor; this does not scale with the number of variables. Compared to Hadoop-based approaches such as [1], Spark resilience ensures that relevant data structures stay in memory, and are not read repeatedly from disk at every step of ADMM, leading to improved performance. We have open-sourced our ADMM implementation,[1] written in Python over Spark.

It should be noted that there are other approaches to distributed optimization than our Spark-ADMM methodology. In terms of the algorithm, Stochastic Gradient Descent (SGD) [29] and Proximal Gradient Method (PGM) [21] are two alternatives to ADMM that can be used for distributed optimization. In these methods, each processor locally performs a tiny gradient computation. In contrast, in ADMM, each processor solves a local optimization problem over a subset of the data. This is a computationally intensive task, that leads to a greater advance towards the optimal solution per iteration. In turn, this decreases the number of iterations, as well as the communication traffic among processors.

In terms of the platform, Hadoop as well as the different flavors of parameter servers [18, 24, 13, 17] are some common alternatives to Spark. Compared to Hadoop, Spark has lower overhead as data is cached in memory, and has built-in fault-tolerance, recovering computation from failed processors automatically. It is also a far easier programming framework to work with, as it seamlessly integrates with high-level languages like Python and Scala. Parameter servers support asynchronous consistency models [24] in contrast to the synchronous Spark framework described here. While asynchrony can boost parallelism, it is specifically effective for algorithms like SGD and PGM where the task units and therefore the parameter updates are small and frequent; this is not the case for ADMM, in which updates are computed through bulk tasks. In addition, though synchronous, ADMM provides asymptotic convergence rate guarantees with fewer assumptions.

3. PROBLEM STATEMENT

In this section, we give a formal description of the problem we address, namely, joint traffic forecasting among a collection of articles. We begin by describing the traffic traces typically at the disposal of a provider, and briefly review how such data alone can be used for forecasting purposes. Finally, we describe the approach we take to enhance this prediction process using article features.

3.1 Trace Description

For our purposes, a *traffic trace* is a multi-dimensional time series. Each point in this time series captures the traffic received by an article within a specific timeslot. The dimensions of the traffic indicate the different *types* of traffic that an article receives. In our case, we categorize traffic by *referral type*; that is, we consider:

[1]http://github.com/yahoo/SparkADMM

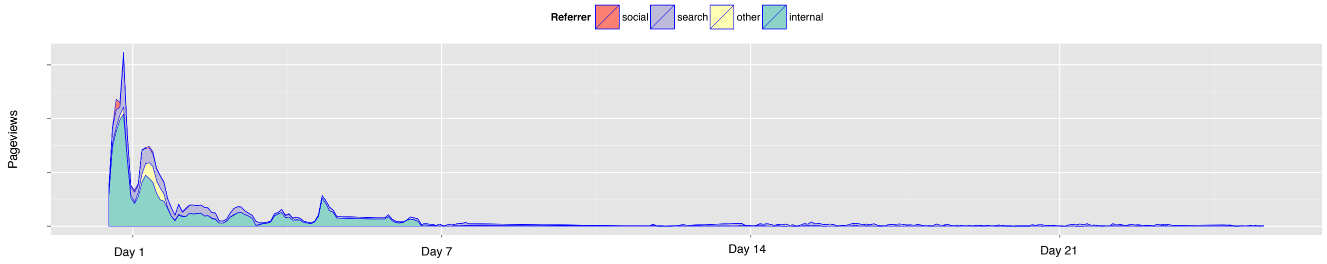

Figure 1: Example of an article's traffic trace. Labels are as follows: "internal" indicates a referral to this article from the same domain, "social" captures referrals from social media like Twitter and Facebook, "search" indicates traffic from a search engines, and remaining traffic is labeled as "other".

- "Internal" traffic; this time series measures the number of article views at a given timeslot by users previously visiting another article within the same domain.
- "Social" traffic; this time series measures the number of article views by users previously visiting social media like Twitter, Facebook, LinkedIn, Weibo, etc.
- "Search" traffic; this time series measures the number of article views by users that reach the article by clicking at a link provided by a search engine, like Google, Bing, Yahoo search, Baidu, etc.

We focus on the above categories, though other traffic categorizations, e.g., by viewer device, are also possible. Both our analysis, as well our code, are agnostic to the categories used. An example of an anonymized traffic trace can be found in Figure 1. Most articles exhibit this "gamma distribution"-like shape: traffic peaks within a few hours, and then dies out.

We assume that each article is monitored from the time it is published for at a total of T timeslots, the *monitoring duration*. We denote by P the number of referral types (in our case, $P = 3$), and by $\boldsymbol{y}(t) \in \mathbb{R}^P$, for $t = 1, \dots, T$ the traffic of an article at timeslot t. We also denote by $\mathcal{Y}^t = \{\boldsymbol{y}(\tau)\}_{\tau=0}^{t}$ the *trace history* up to and including time t; as such, the trace history $\mathcal{Y}^T$ contains the entire trace.

3.2 Forecasting Models

A *parametrized forecasting model* is a mapping f that takes as input:

- the trace history up to time t, i.e., $\mathcal{Y}^{t-1}$ and
- a parameter vector $\boldsymbol{\beta} \in \mathbb{R}^D$,

and outputs a prediction $\hat{\boldsymbol{y}}(t)$ of the traffic at time t. Formally, the prediction is given by

$$\hat{\boldsymbol{y}}(t) = f(\boldsymbol{\beta}; \mathcal{Y}^{t-1}),$$

where $f : \mathbb{R}^D \times \bigcup_{\tau=0}^{\infty} [\mathbb{R}^P]^\tau \to \mathbb{R}^P$. Clearly, such a model predicts traffic at time t using the trace history up to t; in can also predict traffic at time $T + 1$ (i.e., outside the observed trace).

The parameters of the model can be trained by fitting predictions to the existing trace. E.g., they can be computed by minimizing the square error of predictions to actual values:

$$F(\boldsymbol{\beta}; \mathcal{Y}^T) \equiv \textstyle\sum_{t=1}^{T} \|\boldsymbol{y}(t) - f(\boldsymbol{\beta}; \mathcal{Y}^{t-1})\|_2^2, \quad (1)$$

where $\|\cdot\|_2$ is the L_2 norm. For the sake of illustration, we give an example of such a parametrized model below; Section 5 contains an exhaustive presentation of all models we use.

Example: Linear Autoregressive Model. In a linear autoregressive model [7], the traffic at time t is expressed as a linear function of past traffic, i.e., $\hat{\boldsymbol{y}}(t) = \beta_0 + \sum_{\tau=1}^{\tau_0} B_\tau \boldsymbol{y}(t-\tau)$. The parameters of the model are the vector $\beta_0 \in \mathbb{R}^P$ and the matrices $B_\tau \in \mathbb{R}^{P\times P}$, $\tau = 1, \dots, \tau_0$. Intuitively, τ_0 captures how far back in the past one needs to go to make a prediction.

We note that, if the square error function F in (1) is convex in $\boldsymbol{\beta}$, there exist well-known techniques for finding the optimal $\boldsymbol{\beta}$ that minimizes F (see, e.g., [6]). This is the case, e.g., for the square error function resulting from the simple autoregressive model described above.

3.3 Using Article Features

As discussed in the introduction, using a parametrized prediction model has certain disadvantages. First, there is a "cold-start" problem: there may be insufficient traffic for an article that has just been published to train the model and give meaningful predictions. Broadly speaking, parametrized predictive models perform best when T is large and the historical data suffice to train the parameters $\boldsymbol{\beta}$. A second disadvantage is that this type of forecasting treats articles in isolation, and does not leverage article similarities. For example, this approach does not exploit the fact that all articles mentioning a certain artist or world event seem to be generating considerable traffic.

A natural way to address these issues is to link model parameters to article features. In particular, suppose that we have a training dataset comprising N traffic traces $\mathcal{Y}_i^T$, $i \in \mathcal{N} \equiv \{1, \dots, N\}$. Suppose also that each article $i \in \mathcal{N}$ is associated with M features, describing its content. We denote by $\boldsymbol{x}_i \in \mathbb{R}^M$ the feature vector of article i. Typically, although the number of features M may be large, each $\boldsymbol{x}_i$ has sparse support, i.e., for $\mathcal{M} \equiv \{1, \dots, M\}$,

$$|\,\mathsf{supp}(\boldsymbol{x}_i)| = |\{j \in \mathcal{M} : \boldsymbol{x}_{ij} \neq 0\}| \ll M \text{ for all } i \in \mathcal{N}. \quad (2)$$

Our approach is to regress the parameters of the traffic model from these features. In particular, we assume that the parameters $\boldsymbol{\beta}_i$ to be plugged in the fitting function (1) are given by a linear transformation of these features, i.e.,

$$\boldsymbol{\beta}_i = Z\boldsymbol{x}_i, \quad \text{for all } i \in \mathcal{N} \quad (3)$$

where $Z \in \mathbb{R}^{D\times M}$ is an unknown matrix to be trained from the data. In particular, Z can be trained from the dataset $\{\mathcal{Y}_i^T, \boldsymbol{x}_i\}_{i=1}^N$ by solving (1), subject to the set of linear constraints (3). However, in the presence of a large number of features, it is natural to expect that matrix Z is sparse, and only few features are relevant to forecasting. For this reason, we incorporate feature selection by solving instead a regularized minimization problem:

$$\textsc{GlobalOptimization}$$
$$\min_{Z\in\mathbb{R}^{D\times M}} \textstyle\sum_{i=1}^N F(Z\boldsymbol{x}_i; \mathcal{Y}_i^T) + \lambda \|Z\|_1, \tag{4}$$

where F as in (1), $\|Z\|_1$ is the sum of absolute values of the elements of Z (the entry-wise L_1 matrix norm), and $\lambda > 0$ is a regularization parameter. Note that, if F is convex in $\boldsymbol{\beta}$, the problem (4) remains convex, and can thus be solved with standard techniques.

Our approach indeed addresses the two problems outlined in the beginning of this section. First, the cold-start problem is addressed, as the parameters of a new article can be estimated from Z and the features $\boldsymbol{x}_i$ even in the absence of any traffic, using (3). Second, Z is jointly trained across multiple articles: indeed, articles that have similar features are predicted to have similar behavior—this is again an implication of (3). One can train Z infrequently, e.g., once or twice a day: we discuss different ways of using a trained Z to make frequent (e.g., hourly) predictions in Section 7.

4. A PARALLEL SOLUTION

Training models jointly across traces, by learning matrix Z through (4), is quite appealing for the reasons we outlined above. However, it also raises a significant scalability challenge. Even if the prediction model is linear, solving the lasso regression (4) involves jointly minimizing an objective comprising a total of $N \times T \times P$ square terms, over $D \times M$ unknown variables. When the number of websites N and the size of the traces span several days, this number can be prohibitive. In fact, the entire data set of traces and features may not even fit in memory.

This calls for parallelizing the solution of (4). We outline how this can be done using the Alternating Directions Method of Multipliers (ADMM) [5]. We focus here on how (4) can be solved in parallel through this approach, as well as in formally describing both the architecture and the execution of each step of this iterative process.

4.1 Technical Preliminary

For completeness, we give a brief overview of ADMM; the exposition below follows [5], which is an excellent reference for both the method as well as its applications. ADMM solves arbitrary convex optimization problems with a separable objective and linear equality constraints, such as:

$$\text{Minimize:} \quad g_1(\boldsymbol{x}) + g_2(\boldsymbol{z}) \tag{5a}$$
$$\text{subject to:} \quad A\boldsymbol{x} + B\boldsymbol{z} = c \tag{5b}$$

for $\boldsymbol{x} \in \mathbb{R}^d$, $\boldsymbol{z} \in \mathbb{R}^{d'}$, $A \in \mathbb{R}^{d''\times d}, B \in \mathbb{R}^{d''\times d'}$, $c \in \mathbb{R}^{d''}$, and $g_1 : \mathbb{R}^d \to \mathbb{R}$, $g_2 : \mathbb{R}^{d'} \to \mathbb{R}$ convex functions. For $\rho > 0$, consider the following *augmented Lagrangian* of (5)

$$\begin{aligned} L_\rho(\boldsymbol{x},\boldsymbol{z},\boldsymbol{\xi}) = g_1(\boldsymbol{x}) + g_2(\boldsymbol{z}) + \boldsymbol{\xi}^\top(A\boldsymbol{x} + B\boldsymbol{z} - c) \\ + \frac{\rho}{2}\|A\boldsymbol{x} + B\boldsymbol{z} - c\|_2^2, \end{aligned} \tag{6}$$

where $\boldsymbol{\xi} \in \mathbb{R}^{d''}$ are the dual variables (i.e., Lagrange multipliers) corresponding to the constraints (5b). Compared to the standard Lagrangian, (6) contains an additional quadratic term $\frac{\rho}{2}\|A\boldsymbol{x} + B\boldsymbol{z} - c\|_2^2$, which vanishes for $\boldsymbol{x}, \boldsymbol{z}$ feasible.

ADMM solves (5) through *dual ascent*, iteratively minimizing the augmented Lagrangian (6) as follows:

$$\boldsymbol{x}^{k+1} = \arg\min_{\boldsymbol{x}\in\mathbb{R}^d} L_\rho(\boldsymbol{x}, \boldsymbol{z}^k, \boldsymbol{\xi}^k) \tag{7a}$$
$$\boldsymbol{z}^{k+1} = \arg\min_{\boldsymbol{z}\in\mathbb{R}^{d'}} L_\rho(\boldsymbol{x}^{k+1}, \boldsymbol{z}, \boldsymbol{\xi}^k) \tag{7b}$$
$$\boldsymbol{\xi}^{k+1} = \boldsymbol{\xi}^k + \rho\nabla_{\boldsymbol{\xi}} L_\rho(\boldsymbol{x}^{k+1}, \boldsymbol{z}^{k+1}, \boldsymbol{\xi}^k) \tag{7c}$$

where $k \in \mathbb{N}$. This iterative procedure is guaranteed to converge to a solution of (5) under mild conditions, namely, that g_1 and g_2 are convex and that the augmented Lagrangian has a saddle point [5]. Most importantly, when f and g are separable, the iterations (7) are parallelizable. This is precisely the property we exploit in this paper.

4.2 Solving GlobalOptimization through ADMM

We now discuss how ADMM can be used to parallelize the solution of GlobalOptimization. We do so by casting GlobalOptimization as a *generalized consensus problem* (see Chapter 7 of [5]). For $\mathcal{L} \equiv \{1, \ldots, L\}$, let $\{I_\ell\}_{\ell\in\mathcal{L}}$ be a partition of the set of articles $\mathcal{N}$, i.e., $\bigcup_{\ell\in\mathcal{L}} I_\ell = \mathcal{N}$, and $I_\ell \cap I_{\ell'} = \emptyset$, for $\ell \neq \ell'$. For $\mathcal{M} = \{1, \ldots, M\}$ the set of features, consider the bipartite graph $G(\mathcal{L}, \mathcal{M}, E)$, where

$$E = \{(\ell, j) : j \in \textstyle\bigcup_{i\in I_\ell} \mathsf{supp}(\boldsymbol{x}_i)\}, \tag{8}$$

where $\mathsf{supp}(\boldsymbol{x}_i)$ the support of $\boldsymbol{x}_i$, as in (2). An example of such a graph can be shown in Figure 2(a). Intuitively, the partition $\{I_\ell\}_{\ell\in\mathcal{L}}$ splits the dataset of articles into L pieces. The two sets of nodes of the bipartite graph G correspond to partitions in $\mathcal{L}$ and features in $\mathcal{M}$, respectively. An edge between a partition node $\ell \in \mathcal{L}$ and a feature node $j \in \mathcal{M}$ exists if and only if ℓ "covers" j, i.e., there exists a feature vector $\boldsymbol{x}_i$, $i \in I_\ell$, whose j-th coordinate is non-zero. Note that graph G is sparse when $L = \Theta(N)$, by the sparse support assumption (2), .

For each node $\ell \in \mathcal{L}$, we denote by

$$\Gamma(\ell) = \{j \in \mathcal{M} : (\ell, j) \in E\} \overset{(8)}{=} \textstyle\bigcup_{i\in I_\ell} \mathsf{supp}(\boldsymbol{x}_i) \tag{9}$$

the neighborhood of ℓ in G; we define $\Gamma(j)$, $j \in \mathcal{M}$, similarly. For $\boldsymbol{x} \in \mathbb{R}^M$ and $A \subset \mathcal{M}$ we denote by $[\boldsymbol{x}]_A \in \mathbb{R}^{|A|}$ the projection of $\boldsymbol{x}$ to the coordinates spanned by A. Similarly, for $Z \in \mathbb{R}^{D\times M}$, we define $[Z]_{\cdot A} \in \mathbb{R}^{D\times|A|}$ to be the corresponding projection of the columns of Z.

We then consider this equivalent formulation of (4):

$$\text{Minimize:} \quad \sum_{\ell\in\mathcal{L}}\sum_{i\in I_\ell} F(Z_\ell[\boldsymbol{x}_i]_{\Gamma(\ell)}; \mathcal{Y}_i^T) + \lambda\|Z\|_1, \tag{10a}$$
$$\text{subject to:} \quad Z_\ell = [Z]_{\cdot\Gamma(\ell)}, \qquad \ell = 1, \ldots, L. \tag{10b}$$

In this formulation, $Z \in \mathbb{R}^{D\times M}$ is a *consensus value*, while $Z_\ell \in \mathbb{R}^{D\times|\Gamma(\ell)|}$, $\ell \in \mathcal{L}$, are auxiliary variables "local" to each partition ℓ; through (10b), they coincide with the columns of the consensus value Z in $\Gamma(\ell)$. Problem (10) is equivalent to (4): to see this, observe that $Z_\ell[\boldsymbol{x}_i]_{\Gamma(\ell)} = Z\boldsymbol{x}_i$ for any Z such that $Z_\ell = [Z]_{\cdot\Gamma(\ell)}$, by (9).

This is also a classic example of a problem that can be solved through ADMM. Indeed, (10) is a special case of (5),

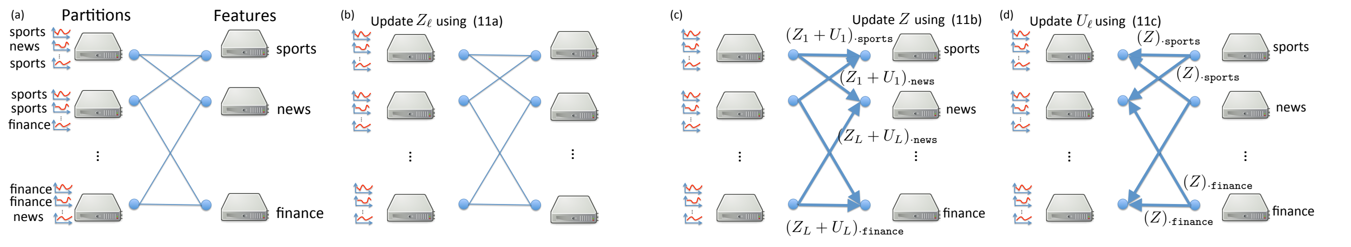
(a)
Partitions
Features
sports
news
sports
sports
sports
finance
finance
finance
news
sports
news
finance
(b)
Update Z_ℓ using (11a)
(c)
Update Z using (11b)
$(Z_1+U_1)_{\cdot\texttt{sports}}$
$(Z_1+U_1)_{\cdot\texttt{news}}$
$(Z_L+U_L)_{\cdot\texttt{news}}$
$(Z_L+U_L)_{\cdot\texttt{finance}}$
(d)
Update U_ℓ using (11c)
$(Z)_{\cdot\texttt{sports}}$
$(Z)_{\cdot\texttt{sports}}$
$(Z)_{\cdot\texttt{finance}}$
$(Z)_{\cdot\texttt{finance}}$

Figure 2: Distributed Implementation of ADMM solving (4). Figure (a) illustrates the bipartite graph G when articles are described by M binary features, indicating whether an article can be included in a certain category (sports, news, etc.). Figures (b)-(d) describe the parallel ADMM algorithm. In (b), each partition processor fits a local solution Z_ℓ to its data, penalized by a "proximal" quadratic term, forcing the solution to be close to the consensus value. In (c) the resulting local solutions, minus the dual variables, are sent to the feature processors, that averages these value and apply soft-thresholding, producing thus a new consensus value. In (d), the feature processors send the updated consensus value columns to the partition processors, which use this to update their dual variables, and the process is repeated.

as it is minimizing the sum of:

$$g_1(Z_1,\ldots,Z_L)=\sum_{\ell\in\mathcal{L}}\sum_{i\in I_\ell}F(Z_\ell[\boldsymbol{x}_i]_{\Gamma(\ell)};\mathcal{Y}_i^T),\ \text{and}\ g_2(Z)=\lambda\|Z\|_1,$$

subject to the equality constraints (10b). The ADMM iterations (7) take the following form:

$$Z_\ell^{k+1}=\operatorname*{arg\,min}_{Z_\ell\in\mathbb{R}^{D\times M}}\sum_{i\in I_\ell}F(Z_\ell[\boldsymbol{x}_i]_{\Gamma(\ell)};\mathcal{Y}_i^T)+\ldots$$

$$+\frac{\rho}{2}\|Z_\ell-[Z^k]_{\cdot\Gamma(\ell)}+U_\ell^k\|_2^2 \tag{11a}$$

$$[Z^{k+1}]_{dj}=S_{\frac{\lambda}{|\Gamma(j)|\rho}}\Big(\frac{1}{|\Gamma(j)|}\sum_{\ell\in\Gamma(j)}[Z_\ell^{k+1}+U_\ell^k]_{dj}\Big),\ \forall d,j \tag{11b}$$

$$U_\ell^{k+1}=U_\ell^k+Z_\ell^{k+1}-[Z^{k+1}]_{\cdot\Gamma(\ell)} \tag{11c}$$

where $S_\kappa:\mathbb{R}\to\mathbb{R}$ is the *soft thresholding operator*:

$$S_\kappa(a)=\begin{cases}a-\kappa, & \text{if } a>k,\\ 0, & \text{if } |a|\le\kappa, \text{and}\\ a+\kappa, & \text{if } a<-\kappa,\end{cases} \tag{12}$$

which is applied to every element of the consensus matrix Z, and matrices $U_\ell\in\mathbb{R}^{D\times|\Gamma(\ell)|}$ are the dual variables of the equality constraints. The steps outlined above can be parallelized across L processors; we discuss how this parallelization takes place below.

4.3 Parallel Architecture

In a parallel implementation of (11), both the organization of parallel processors and the communication pattern between them are determined by the bipartite graph G described in Section 4.2. In particular, computation occurs over L "partition" processors and M "feature" processors, connected to each other according to G. The consensus value Z is partitioned among the M feature processors in $\mathcal{M}$, with each $j\in\mathcal{M}$ storing the j-th column of the consensus Z. This is in contrast to previous consensus ADMM implementations, such as [1, 16], where the consensus step (11b) is centralized and becomes a bottleneck for large M.

The trace and feature vectors dataset $\{\boldsymbol{x}_i;\mathcal{Y}_i^T\}_{i=1}^N$ is partitioned among the L processors in $\mathcal{L}$, so that processor ℓ stores the data indexed by I_ℓ[2]. At each iteration $k\in\mathbb{N}$, a partition processor ℓ updates the primal and dual variables $Z_\ell,U_\ell\in\mathbb{R}^{D\times|\Gamma(\ell)|}$, while a feature processor updates its corresponding column of Z. In more detail, matrices Z and U_ℓ are initialized to zero, and the distributed execution of (11) proceeds as follows at each step $k\in\mathbb{N}$:

1. Each partition processor ℓ computes, in parallel, a local solution Z_ℓ, fitting this to its local data through (11a).
2. Each partition processor ℓ sends each column of $Z_\ell+U_\ell$ to the corresponding feature processor in $\Gamma(\ell)$.
3. Each feature processor j computes a new consensus value Z from the received values through (11b): that is, it averages the reported vector values, and then applies the soft threshold (12) entry-wise.
4. Each feature processor j sends its new consensus value Z to the partition processors in $\Gamma(j)$, that update their dual variables U_ℓ through (11c), and the process is repeated.

The above steps are outlined in Figure 2. The local optimization solved by partition processors amounts to performing a local fit on the data they store, with an additional quadratic penalty. Intuitively, this additional "proximal" penalty term in (11a) "forces" the local solution to be closer to the consensus value reported in the previous step by the feature processors, thereby leading to convergence. The sparsity of G implies that this solution scales well w.r.t. to memory and communication costs, in *both* the number of articles N *as well as* the number of features M. In particular, if G is sparse, both the data stored by partition processors ℓ and the messages they exchange are $O(\Gamma(\ell))$. Moreover, no single processor is required to store all of Z, or any variable of the order of M or N, at any time.

In our implementation, the same processors play the role of both "partition" and "feature" processors, while message passing over G happens through appropriate map, reduce, and join operations. We elaborate on this in Section 6.

4.4 Nested ADMM

Note that, in the distributed implementation of ADMM described above, a slave processor $\ell\in\mathcal{L}$, solves at each iteration the optimization problem determined by (11a). Interestingly, this too can be solved through ADMM. To see this, note that (11a) is equivalent to:

[2]For space efficiency reasons, processor ℓ in fact stores $[\boldsymbol{x}_i]_{\Gamma(\ell)}$, for $i\in I_\ell$, or some other compact representation of the sparse vector $\boldsymbol{x}_i$.

LocalOptimization

$$\text{Minimize} \quad \sum_{i\in I_\ell} F(\boldsymbol{\beta}_i;\mathcal{Y}_i^T) + \tfrac{\rho}{2}\|Z_\ell - \hat{Z}_\ell\|_2^2, \tag{13a}$$

$$\text{subject to:} \quad \boldsymbol{\beta}_i = Z_\ell[x_i]_{\Gamma(\ell)}, \qquad i \in I_\ell \tag{13b}$$

where $\hat{Z}_\ell = [Z]_{\cdot\Gamma(\ell)} - U_\ell$ is the "target" value that the proximal penalty forces the optimization to agree with. As a separable objective with linear constraints, this also admits a solution through ADMM. In particular, (7) reduces to:

$$\boldsymbol{\beta}_i^{k+1} \leftarrow \arg\min_{\boldsymbol{\beta}_i} F(\boldsymbol{\beta};\mathcal{Y}_i^T) + \frac{\rho'}{2}\left\|\boldsymbol{\beta}_i - Z_\ell^k[\boldsymbol{x}_i]_{\Gamma(\ell)} + \boldsymbol{u}_i^k\right\|_2^2 \tag{14a}$$

$$Z_\ell^{k+1} \leftarrow \arg\min_{Z_\ell} \frac{\rho}{2}\|Z_\ell - \hat{Z}_\ell\|_2^2 + \frac{\rho'}{2}\sum_{i=1}^{n}\|\boldsymbol{\beta}_i - Z_\ell[\boldsymbol{x}_i]_{\Gamma(\ell)} + \boldsymbol{u}_i\|_2^2 \tag{14b}$$

$$\boldsymbol{u}_i^{k+1} \leftarrow \boldsymbol{u}_i^k + \boldsymbol{\beta}_i^{k+1} - Z_\ell^{k+1}[\boldsymbol{x}_i]_{\Gamma(\ell)} \tag{14c}$$

The iterative steps (14) ensure that no more than one trace is at any time loaded in memory. Moreover, solving the local problem through (14) implies that it is relatively easy to incorporate, and train, a variety of different traditional time series forecasting models. This is because (14a) involves performing a traditional fit over *a single trace*, with an additional proximal quadratic penalty term. With this minor modification to existing code for traditional forecasting models, a developer can exploit the above pipeline to train jointly across multiple traces, while regressing parameters from features. We elaborate on this in Section 6.

5. TRAFFIC MODELS

In this section, we describe the parametric models $f(\boldsymbol{\beta};\mathcal{Y}^t)$ we use in our analysis.

Constant Model (`CONST`). As a baseline, we consider a constant model that predicts that traffic at time t equals the traffic at time $t-1$, i.e, $\hat{\boldsymbol{y}}(t) = \boldsymbol{y}(t-1)$. In ARIMA terminology [12], this is an ARIMA(0,1,0) model.

Vector Auto Regressive Model (`VAR`). We also consider the Auto Regressive Model of order k (VAR(k)) [19, 3], described in Example 1. Recall that under this model the traffic at time t is a linear combination of past traffic: $\hat{\boldsymbol{y}}(t) = \beta_0 + \sum_{\tau=1}^{\tau_0} B_\tau \boldsymbol{y}(t-\tau)$ where, $\beta_0 \in \mathbb{R}^P$ and $\{B_\tau \in \mathbb{R}^{P\times P}\}_{\tau=1}^{\tau_0}$ are the parameters of the model, and τ_0 is the length of history used for auto-regression. In ARIMA terminology, this is an ARIMA(τ_0,0,0) model.

Constant Auto Regressive Model (`CAR`). `CAR` is a combination of the `CONST` and `VAR` models, that fits a `VAR` model to the error $\Delta\boldsymbol{y}(t) = \boldsymbol{y}(t) - \boldsymbol{y}(t-1)$ of the `CONST` model. Equivalently, a prediction is given by: $\hat{\boldsymbol{y}}(t) = \boldsymbol{y}(t-1) + \beta_0 + \sum_{\tau=1}^{\tau_0} B_\tau \Delta\boldsymbol{y}(t-\tau)$, and this is an ARIMA($\tau_0$,1,0) model.

Simple Infection-Based Model (`SI`). Beyond the above generic time series models, we also consider a model motivated by meme-virality, that has been used in the past to describe mentions of blog topics and hashtags [20]. The intuition behind it is that social traffic acts like an infection: someone visiting a website from Facebook or Twitter will, with some probability, post this website on their Facebook page, tweet about it, or otherwise share it with their friends. Moreover, the "virality" of a pageview decays with time: for example, a post containing a url is more likely to generate additional page views when it is first posted, and becomes less likely to do so as it moves down on a person's feed. These ideas are formalized by the following model adapted from Matsubara et al. [20]:

$$\hat{\boldsymbol{y}}(t) = \sum_{\tau=1}^{t} B\boldsymbol{y}(t-\tau)s(t-\tau) + \delta(t)$$

where:

- $B \in \mathbb{R}^{P\times P}$ is a matrix whose elements are positive, and captures the "infectiousness" across different types of traffic, and s is a dampening factor, given by $s(t) = \frac{1}{(1+t)^\alpha}$ for a parameter $\alpha > 0$. This is set to 1.5 in our experiments, as recommended by Matsubara et al. [20].
- $\delta(t) \in \mathbb{R}^P$ is a vector whose coordinate p takes a non-zero value for a single time $\tau_p \in \mathbb{N}$.

Intuitively, the dampening factor forces "virality" to decay with time: page views that happen in the distant past (i.e., for large τ) have little effect present traffic, while $\delta(t)$ allows for a single "spike" in each coordinate. In our experiments, the spike at coordinate i is set to occur at the first non-zero time of the trace. Under these constraints, fitting `SI` through (1) becomes a convex optimization problem.[3]

Ordinary Differential Equation Model (`ODE`). Recall that our traffic traces, exhibit a growth-and-decay patern, as observed in Figure 1. One commonly used model to describe such behavior is an Ordinary Differential Equation model (`ODE`) [8]. In particular, `ODE` assumes $\boldsymbol{y}(t)$ satisfies:

$$\sum_{\tau=0}^{k} \beta_{p,\tau} y_p^{(\tau)}(t) = 0, 1 \le p \le P \tag{15}$$

where $y_p^{(\tau)}(t) = \frac{d^\tau}{dt^\tau} y_p(t)$, k is the order of the ODE and $\boldsymbol{\beta} = \{\boldsymbol{\beta}_p = [\beta_{p,0},\ldots,\beta_{p,k}]\}_{p=1}^P$ are the parameters of the model. Without loss of generality, we set $\beta_{p,k} = 1, 1 \le p \le P$. Given $\boldsymbol{\beta}$ is known, one can solve the ODE (explicitly or numerically) to forecast $\hat{\boldsymbol{y}}(t)$. Estimating $\boldsymbol{\beta}$ from the traffic traces amounts to computing the derivatives of the training trace $\{y_p^{(\tau)}(t)\}_{\tau=0}^k$ and regressing the highest order derivative $y_p^{(k)}(t)$ on the rest so that estimating $\boldsymbol{\beta}$ again reduces to a convex least squares minimization problem. In our experiments, we estimate derivatives using second order central differences in the interior and first order differences on the boundary of the trace.

6. IMPLEMENTATION

In this section, we describe how we implemented the parallel, nested ADMM through Spark.

Spark. Apache Spark is a programming framework for parallel cluster computing [28]. It is built around the concept of Resilient Distributed Datasets (RDD), well suited for distributed iterative algorithms like ADMM. RDDs are distributed data structures that support map/reduce operations, as well as *persistence*: outcomes of operations are stored in memory, and can be accessed directly by later operations. Spark also provides a seamless interface to several popular programming languages, including Python, which we used. The main challenge in our setting is the fully parallel implementation of the distributed ADMM operations over the bipartite graph outlined in Section 4.3, through map, reduce, and join operations over RDDs. We outline the design of our implementation below.

RDDs and Related Map-Reduce Operations. In our implementation, computations are carried out on sev-

[3]We note that the original model presented in [20] is non-convex, and `SI` is a simplification.

Algorithm 1 Pseudocode for SparkADMM

```
Input: data: RDD with tuples of the form (partitionid,datavalue), where datavalue is a list of (metadata,trace) pairs.
    index: RDD with tuples of the form (coordinate,partitionid), storing the edges of graph $G$.
Output: Z: RDD with tuples of the form (coordinate, value), containing trained matrix
 1: Distribute data and index in $L$ machines, ensuring that each machine gets a single pair (partitionid,datavalue).
 2: Initialize RDD Z with tuples of the form (coordinate, 0.0), and distribute it over the $L$ machines.
 3: Initialize RDDs Uloc and Zloc with tuples of the form (partitionid, [ (coordinate1,0.0), (coordinate2,0.0),...]),
    where the list of coordinate/value pairs contains coordinates relevant to this partitionid. Distribute them over the $L$ machines
 4: for $k$ in [1:$K$] do
 5:     Zdistr = Z.join(index).map(lambda (coordinate,(value,partitionid)): (partitionid, (coordinate,value)).groupByKey()
 6:     Uloc = data.join(Zdistr).join(Uloc).join(Zloc).map(ApplyUlocUpdate)   \* Eq. (11c) *\
 7:     Zloc = data.join(Zdistr).join(Uloc).join(Zloc).map(ApplyZlocUpdate)   \* Eq. (11a) *\
 8:     ZlplUl = Zloc.join(Uloc).map(addPairLists).flatMap(lambda (partition, pairList): pairList)
 9:     Z = ZlplUl.map(lambda (coordinate:value):(coordinate,(value,1.0)))
            .reduceByKey(lambda $((v_0, c_0), (v_1, c_1))$:$(v_0 + v_1, c_0 + c_1)$).map(ApplyTresholdingOperator)   \* Eq. (11b) *\
10: end for
```

eral RDDs indexed by one of two types of keys: *partition keys* and *feature keys*. Operations on RDDs indexed by the former correspond to the left-hand side of the bipartite graph G, while operations on RDDs indexed by feature keys correspond to the right-hand side.

In more detail, feature vectors and traces of articles are stored together in an RDD called `data`, partitioned over articles across multiple machines. Similarly, the bipartite graph G is stored as an RDD called `index`. The global Z matrix is also stored as an RDD called `Z`, indexed by its coordinates, and reconstructed at each iteration of ADMM. All RDDs are distributed and stored over *the same* L machines.

These RRDs consist of `(key,value)` pairs. RDD `data` comprises pairs of the form (partition-id,data-value), where partition-id is a number between 1 and L and the data-value is a list of trace and feature vector pairs describing the articles in each partition. RDD `Z` comprises pairs of the form (coordinate, value), corresponding to numerical values associated with each coordinate of the matrix Z. Coordinates are represented internally as (feature,parameter) pairs, representing columns and rows of Z. In turn, `index` contains pairs of the form (coordinate, partition-id), corresponding to the edges in graph G: if feature j connects to partition ℓ, all coordinates (i.e., the entire column) of `Z` corresponding to this feature are represented in `index` through an appropriate pair linking them to this partition-id.

The implementation then proceeds according to the following alternating steps, outlined also in Algorithm 1:

- The RDDs `data`, `Z`, and `index` are distributed across L processors, with values in `Z` initialized to zero.
- For each partition-id $\ell \in \mathcal{L}$, sub-matrices Z_ℓ and U_ℓ are also constructed as RDDs. These comprise tuples of the form (partition-id, list), where list contains the elements of these matrices represented as (coordinate, value) pairs.
- At each iteration of ADMM, the columns of `Z` are scattered across the L machines. This happens by first joining `Z` with `index`, thus linking coordinates to partition-ids they need to be replicated to. A map reorders data so that the partition-id becomes the new key; subsequently, a groupByKey() operation gathers all pairs corresponding to a partition-id to a list, creating thus a list representation of $(Z)_{\cdot\Gamma(\ell)}$, for every partition-id ℓ.
- This data structure, termed `Zdistr`, is joined with `data`, `Zloc`, and `Uloc` in terms of their common keys (namely, partition-ids). A map operation allows to compute the new values of `Uloc` and `Zloc`, by applying code that implements (11c) and (11a) on the three joined values.
- Finally, through a join and a map, the local sub-matrices `Zloc` and `Uloc` are added together. To construct the new consensus value, the lists representing the added matrices are flattened into an RDD comprising (coordinate, value) pairs. A reduce operation groups values corresponding to the same coordinates together, sums them, and counts the size of $\Gamma(j)$. The sums and counts can subsequently be filtered through a map that computes the average and passes it through the soft thresholding operator (12), producing the new consensus value `Z`.

We stress here that, through appropriate use of so-called Spark *partitioners*, extra care is taken so that `data` (i.e., features and traces of articles), which is the largest RDD, is loaded into memory once; its contents are never shuffled or re-transmitted across processors. Maps, reduces with alternating keys (either features or partition-ids), and joins are used only to shuffle the resulting Z_ℓ, U_ℓ and Z around. Moreover, the memory load at each processor is of the order $O(\frac{N}{L} + \frac{M}{L})$, thus scaling w.r.t. both articles and features.

Modular Implementation. There is an additional implementation advantage acheived through the nested technique discussed in Section 4.4. In particular, the distribution of datasets to machines (handled by Spark), the Lasso/ℓ_1 regularization, incorporating features into prediction, are all coded in a manner completely agnostic to what model is being used to fit parameters to the data. The model appears only in step (14a) of the nested implementation. As such, our code for all of the above operations is written independently of actual prediction model used, and is thus highly modular. Plugging in a new model reduces to simply implementing (14a). To that end, we define an abstract Python class called `AbstractModel`, determining the interface a developer needs to instantiate to run ADMM: beyond I/O methods, all a developer needs to specify is (14a), which amounts to fitting a model to *a single trace*, in the presence of a quadratic regularization term. This allows us to incorporate (and scale!) multiple existing parametric forecasting models very quickly into our code.

7. EVALUATION

7.1 Methodology

We test the performance of our predictions on a dataset described in detail in Table 1. The dataset includes traffic traces for 2K articles. Associated features are extracted from article content, leading to a total of 16K unique fea-

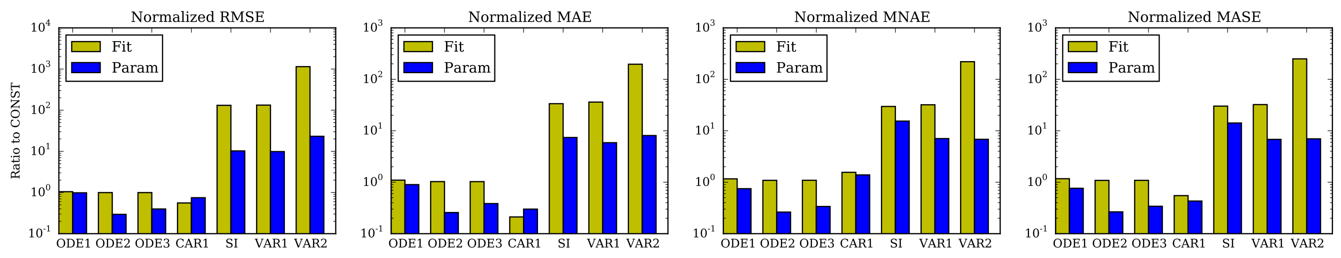

Figure 3: Prediction performance at peak traffic across RMSE, MAE, MNAE, and MASE, for different parametric forecasting models, normalized to `CONST` performance. 'Fit' predicts the traffic at time t^{peak} by fitting parameters over the trace up till $t^{\text{peak}} - 1$, while 'Param' uses parameters regressed over features. Using regression parameters increases prediction performance almost universally, across all metrics and all forecasting models. Moreover, `VAR`k models, that assume stationarity, perform worse than `ODE`k or `CAR`k models, that instead assume transient behavior.

Table 1: Dataset Description

p (types)	N (articles)	M (features)
3	2376	16420
T (mean)	Peak Time (mean)	Features per article
69.27	7.94h	36.6

tures. Traffic traces are broken down by "internal","social", and "search" traffic, based on referrer provenance; each time series is constructed by binning corresponding traffic every hour. The average peak time among articles occurs at approximately 8 hours.

The prediction performance of each method is evaluated through 3-fold cross validation. That is, we train a consensus matrix Z through ADMM using 2/3 of the dataset (the training set), and use it to predict traffic on 1/3 of the dataset (the test set); this is repeated 3 times, with all tree folds serving as test sets. Reported prediction performance metrics are averaged across all three folds. On each test set, we predict traffic for $t = 1, \ldots, 12$ hours: that is, we predict "internal","social", and "search' traffic for time t using only (a) the trace $\mathcal{Y}^{t-1}$ collected so far (b) the article's features, and (c) the matrix Z (learned on the training set). By default, we set the prediction at time $t = 0$ to 0.0. Unless otherwise mentioned, ADMM is performed over 200 executors with $\rho = 0.2$, $\lambda = 10^3$, 50 outer iterations of (11), and local solutions to (13) computed through (14) until primal and dual residuals norms are below 0.01.

Prediction Performance Metrics. We use the following metrics to estimate the performance of different forecasting methods. Given a traffic vector $\boldsymbol{y}_i(t)$ and a prediction $\hat{\boldsymbol{y}}_i(t)$ for article $i = 1, \ldots, N$, we compute the usual root mean square error (RMSE) and the mean absolute error (MAE) as :

$$\text{RMSE}(t) = \sqrt{\frac{1}{N} \textstyle\sum_{i=1}^{N} \|\boldsymbol{y}_i(t) - \hat{\boldsymbol{y}}_i(t)\|_2^2}, \text{ and}$$

$$\text{MAE}(t) = \frac{1}{N} \textstyle\sum_{i=1}^{N} \|\boldsymbol{y}_i(t) - \hat{\boldsymbol{y}}_i(t)\|_1.$$

We also compute two normalized errors, the Mean Normalized Absolute Error (MNAE):

$$\text{MNAE}(t) = \frac{1}{N} \textstyle\sum_{i=1}^{N} \frac{\|\boldsymbol{y}_i(t) - \hat{\boldsymbol{y}}_i(t)\|_1}{\|\boldsymbol{y}_i(t)\|_1}.$$

as well as the so-called Mean Absolute Scaled Error (MASE), that uses the one-step variance as a scaling factor [15]:

$$\text{MASE}(t) = \frac{1}{N} \textstyle\sum_{i=1}^{N} \frac{\|\boldsymbol{y}_i(t) - \hat{\boldsymbol{y}}_i(t)\|_1}{\frac{1}{T-1}\sum_{\tau=2}^{T} \|\boldsymbol{y}_i(\tau) - \boldsymbol{y}_i(\tau-1)\|_1}.$$

Forecasting models. We evaluate the prediction performance of several forecasting methods outlined in Section 5. In particular, we consider the following methods: `ODE` (with degree $k = 1, 2, 3$), `VAR` (with parameter $k = 1, 2$), `CAR` (with $k = 1$), and `SI`, all as defined in Section 5.

For all of the above methods, we evaluate the prediction performance w.r.t. RMSE, MAE, MNAE, and MASE using the following three variants of the forecasting methods.

1. **Fitting.** The first variant is standard fitting. That is, to predict the traffic at time t for article i, we fit the best possible parameters $\boldsymbol{\beta}$ using the trace $\mathcal{Y}_i^{t-1}$ by solving (1). That is, we obtain

$$\boldsymbol{\beta}_{\texttt{FIT}} = \operatorname*{arg\,min}_{\boldsymbol{\beta} \in \mathbb{R}^d} \sum_{\tau=1}^{t-1} \|\boldsymbol{y}(\tau) - f(\boldsymbol{\beta}; \mathcal{Y}_i^{\tau-1})\|_2^2$$

   and predict $\hat{\boldsymbol{y}}_i(t) = f(\boldsymbol{\beta}_{\texttt{FIT}}; \mathcal{Y}^\tau - 1)$, where f is the function describing the parametrized forecasting model.
2. **Using Regressed Parameters**. The second variant leverages learning through ADMM as well as article metadata. For $Z \in \mathbb{R}^{d \times M}$ the matrix learned from the training set, and $\boldsymbol{x}_i \in \mathbb{R}^M$ the features of article i, we predict $\hat{\boldsymbol{y}}_i(t) = f(Z\boldsymbol{x}_i; \mathcal{Y}_i^{\tau-1})$.
3. **Regularized Fit.** The final variant interpolates between the the above two variants. In particular, given a $\rho \geq 0$, we fit a parameter vector by adding a proximal penalty term

$$\boldsymbol{\beta}_\rho = \operatorname*{arg\,min}_{\boldsymbol{\beta} \in \mathbb{R}^d} \sum_{\tau=1}^{t-1} \|\boldsymbol{y}(\tau) - f(\boldsymbol{\beta}; \mathcal{Y}_i^{\tau-1})\|_2^2 + \rho \|\boldsymbol{\beta} - Z\boldsymbol{x}_i\|_2^2, \tag{16}$$

   and subsequently predict $\hat{\boldsymbol{y}}_i(t) = f(\boldsymbol{\beta}_\rho; \mathcal{Y}_i^\tau - 1)$.

Throughout our analysis, we use the constant method `CONST` as a baseline method.

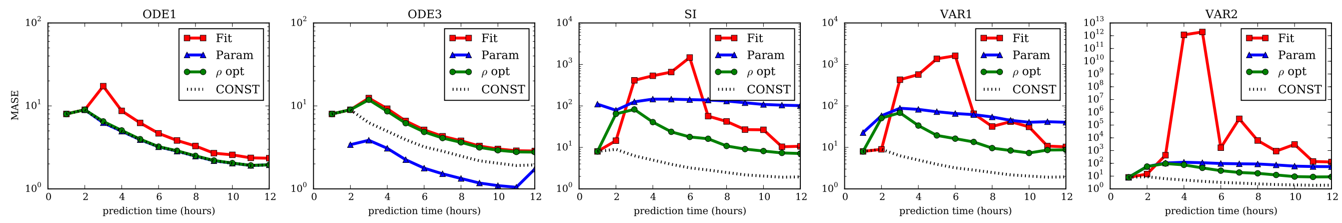

Figure 4: MASE Prediction performance, for different parametric forecasting models, as a function of time. 'Fit' predicts the traffic at time t by fitting parameters over the trace up till $t-1$, while 'Param' uses parameters regressed from features. The optimal interpolation between the two through (16) is indicated by ρ-opt. Feature-extracted parameters increase prediction performance almost universally, but its improvement wanes as the size of the available trace increases. Again, models assuming stationarity perform worse than CONST, while non-stationary models outperform it.

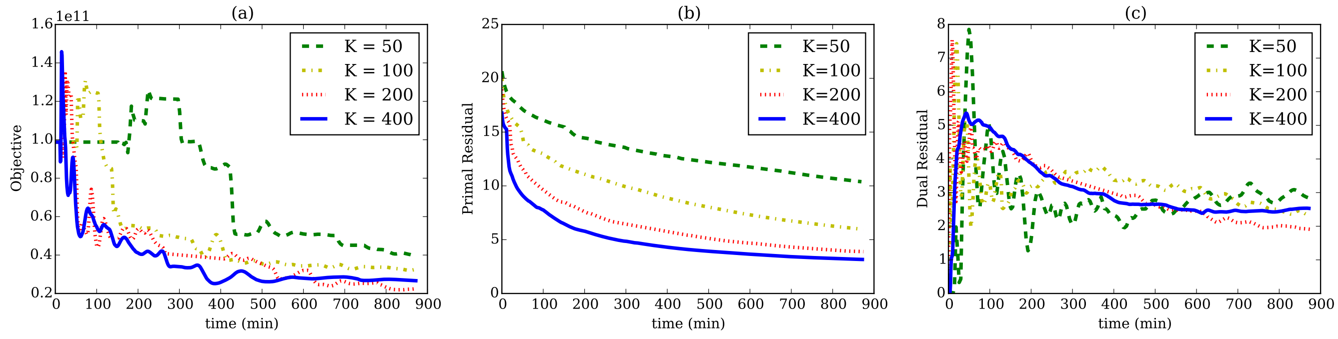

Figure 5: Convergence performance. Subfigure (a) shows the trajectory of the objective function for the `ODE` model of degree 3, for four different values of parallelism K. The primal and dual residuals are shown in subfigures (b) and (c), respectively. Increasing the number of partitions leads to speedier consensus and higher stability.

7.2 Prediction Performance

We evaluated the performance of prediction at the time of peak traffic. For an article i, we define its peak traffic time t_i^{peak} as:

$$t_i^{\mathsf{peak}} = \arg\max_{\tau} \|\boldsymbol{y}_i(\tau)\|_1$$

The performance of our prediction methods at peak traffic (i.e., at $t = t_i^{\mathsf{peak}}$) is reported in Figure 3. For each method and error metric, we report a normalized error: we divide by the prediction error under the baseline prediction method `CONST`.

Predicting peak traffic accurately is important for a provider, as this indicates whether an article will be a good performer or not (and, thus, should be promoted or advertized). Across metrics and forecasting methods, using parameters regressed from features overwhelmingly outperforms prediction through traditional fitting a forecasting model. Indeed, as discussed in the introduction, and as evidenced by Table 1, articles peak early on. As a result, simple fitting yields worse prediction performance than regressing. We also observe that models that assume stationarity like `VAR`k, perform worse than models that do not, like `ODE`k or `CAR`k. In fact, although in all cases regressed parameters outperform fitting, stationary models perform even worse than `CONST`. This is expected, as stationary models do not span the growth-and-decay shape of the trace curves exemplified by Figure 1.

Similar observations can be drawn by studying these metrics at an arbitrary prediction time t. Figure 4 shows how MASE evolves with the prediction time for several different methods. We again see that parameters regressed from features outperform fitted parameters, though the effect diminishes as more trace datapoints become available. In this plot, we also show prediction performance using an optimal value of ρ. We computed this as follows: for each t, we compute the MASE at t using ρ in 10^j, for j ranging from -4 to $+9$, and plot the minimal value among all rhos. For models assuming stationarity, such interpolation outperforms both classic and feature-regressed estimation of parameters.

7.3 Convergence Performance

We next turn our attention to how ADMM scales with the number of partitions K. We focus on training an `ODE` model with degree $k = 3$ over the entire dataset. We train the model using 50, 100, 200, and 400 partitions, each assigned to its own Spark executor. During these experiments, we set $\lambda = 1000$ and $\rho = 2.0$. Each executor solves the local problem (13) through (14). In each of the four experiments, we execute the nested solver (14) until convergence (norms of primal and dual residuals below 0.01), with a maximum number of iterations of (14) set to 600. This ensures that a large fraction (close to 90%) of local solvers terminate early and never reach this limit (*c.f.* the last row of the Table 2).

A summary of statistics for the four experiments can be found in Table 2, while the trajectories of the objective of

Table 2: Summary statistics across partitions.

	$K = 50$	$K = 100$	$K = 200$	$K = 400$
Traces per partition				
Mean	47.52	23.76	11.88	5.94
Max	48	25	11	5
Min	46	23	13	7
Features per partition				
Mean	907.94	502.75	277.67	153.92
Max	1042	666	171	73
Min	800	388	392	243
Local Solver (14) Statistics				
Avg Time (sec)	275.83	160.37	140.01	130.12
Avg # of steps	170.50	105.74	83.90	65.03
Convergence	88.28%	94.14%	97.90%	98.39%

(4), as well as the norms of its primal and dual residuals, are shown in Figure 5. Overall, increasing the number of partitions speeds-up convergence. As indicated in Table 2, increasing the partitions clearly reduces the time per iteration, as well as the number of inner-loop steps required for convergence of (14). However, setup overhead and the cost of aggregating results lead to diminishing returns, with 400 partitions not leading to a significant improvement over 200 partitions. Rather surprisingly, the behavior of the primal residuals in Figure 5 indicates that, despite the large number of partitions, the fact that local iterations terminate quickly allows high-partition experiments to reach consensus faster than experiments with few partitions. Interestingly, the dual residuals also indicate that increasing the number of partitions also reduces oscillations: a small number of partitions, each one solving a large optimization problem, yields to large, oscillating changes in Z from one iteration to the next.

8. CONCLUSIONS

Our analysis indicates that training web traffic forecasting models jointly can significantly improve performance of traditional forecasting models. Crucially, such joint training is amenable to a highly parallelizable implementation through ADMM. More broadly speaking, the above approach can be used to arbitrary convex fitting problems that may benefit from a "collaborative filtering" approach. Investigating such applications of our framework beyond timeseries forecasting, as well as in broader classes of applications beyond web traffic, remains an interesting open problem.

9. REFERENCES

[1] D. Agarwal, B.-C. Chen, B. Long, and L. Zhang. ADMM-based large scale logistic regression. https://github.com/linkedin/ml-ease, 2014.

[2] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang. Laser: a scalable response prediction platform for online advertising. In *WSDM*, 2014.

[3] H. Akaike. Fitting autoregressive models for prediction. *Annals of the Institute of Statistical Mathematics*, 21(1):243–247, 1969.

[4] E. M. Azoff. *Neural Network Time Series Forecasting of Financial Markets.* John Wiley & Sons, Inc., 1994.

[5] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

[6] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge university press, 2009.

[7] P. J. Brockwell and R. A. Davis. *Introduction to Timeseries and Forecasting*, volume 1. Taylor & Francis, 2002.

[8] W. L. Brogan. *Modern Control Theory.* Pearson Education India, 1985.

[9] L. Cao. Support vector machines experts for time series forecasting. *Neurocomputing*, 51:321–339, 2003.

[10] Y. Chang, M. Yamada, A. Ortega, and Y. Liu. Ups and downs in buzzes: Life cycle modeling for temporal pattern discovery. In *ICDM*, 2014.

[11] K. Duh, J. Suzuki, and M. Nagata. Distributed learning-to-rank on streaming data using alternating direction method of multipliers. In *NIPS Big Learning Workshop*, 2011.

[12] B. George. *Time Series Analysis: Forecasting & Control, 3/e.* Pearson Education India, 1994.

[13] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[14] L. Hong, B. Dom, S. Gurumurthy, and K. Tsioutsiouliklis. A time-dependent topic model for multiple text streams. In *KDD*, 2011.

[15] R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4):679–688, 2006.

[16] D. Lawson. Alternating direction method of multipliers implementation using Apache Spark. https://github.com/dieterichlawson/admm, 2014.

[17] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[18] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola. Parameter server for distributed machine learning. In *NIPS Big Learning Workshop*, 2013.

[19] H. Lütkepohl. *Vector Autoregressive Models.* Springer, 2011.

[20] Y. Matsubara, Y. Sakurai, B. A. Prakash, L. Li, and C. Faloutsos. Rise and fall patterns of information diffusion: model and implications. In *KDD*, 2012.

[21] N. Parikh and S. Boyd. Proximal algorithms. *Foundations and Trends in Optimization*, 1(3):123–231, 2013.

[22] Z. Tang, C. de Almeida, and P. A. Fishwick. Time series forecasting using neural networks vs. Box-Jenkins methodology. *Simulation*, 57(5):303–310, 1991.

[23] F. E. Tay and L. Cao. Application of support vector machines in financial time series forecasting. *Omega*, 29(4):309–317, 2001.

[24] J. Wei, W. Dai, A. Kumar, X. Zheng, Q. Ho, and E. P. Xing. Consistent bounded-asynchronous parameter servers for distributed ML. *arXiv preprint arXiv:1312.7869*, 2013.

[25] J. Yang and J. Leskovec. Patterns of temporal variation in online media. In *WSDM*, 2011.

[26] S. Yang, J. Wang, W. Fan, X. Zhang, P. Wonka, and J. Ye. An efficient admm algorithm for multidimensional anisotropic total variation regularization problems. In *KDD*, 2013.

[27] Z.-Q. Yu, X.-J. Shi, L. Yan, and W.-J. Li. Distributed stochastic admm for matrix factorization. In *KDD*, 2014.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud*, 2010.

[29] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *NIPS*, 2010.