An Efficient Alternating Newton Method for Learning Factorization Machines

WEI-SHENG CHIN, National Taiwan University
BO-WEN YUAN, National Taiwan University
MENG-YUAN YANG, National Taiwan University
CHIH-JEN LIN, National Taiwan University

To date, factorization machines (FM) have emerged as a powerful model in many applications. In this work, we study the training of FM with the logistic loss for binary classification, which is a non-linear extension of the linear model with the logistic loss (i.e., logistic regression). For the training of large-scale logistic regression, Newton methods have been shown to be an effective approach, but it is difficult to apply such methods to FM because of the non-convexity. We consider a modification of FM that is multi-block convex and propose an alternating minimization algorithm based on Newton methods. Some novel optimization techniques are introduced to reduce the running time. Our experiments demonstrate that the proposed algorithm is more efficient than stochastic gradient algorithms and coordinate descent methods. The parallelism of our method is also investigated for the acceleration in multi-threading environments.

CCS Concepts: •**Computing methodologies** → **Factor analysis; Learning latent representations; Supervised learning by classification;** •**Mathematics of computing** → **Nonconvex optimization;** •**Information systems** → **Collaborative filtering;** •**Theory of computation** → *Shared memory algorithms;*

Additional Key Words and Phrases: Newton methods, preconditioned conjugate gradient methods, subsampled Hessian matrix

ACM Reference Format:
Wei-Sheng Chin, Bo-Wen Yuan, Meng-Yuan Yang, and Chih-Jen Lin, 2016. An Efficient Alternating Newton Method for Learning Factorization Machines. *ACM Trans. Intell. Syst. Technol.* 0, 0, Article 0 (0), 31 pages.
`DOI:` 0000001.0000001

1. INTRODUCTION

Binary classification has been an important technique for many practical applications. Assume that the observations and the scalar result of an event are respectively described by an n-dimensional feature vector $\boldsymbol{x}$ and a label $y \in \{-1, 1\}$. The model is an output function from the feature space to a real number, $\hat{y} : \mathcal{R}^n \rightarrow \mathcal{R}$, where the output label is the sign of the output value. Among many existing models, we are interested in factorization machines (FM), which have been shown to be useful for high dimensional and sparse data sets [Rendle 2010]. Assume that we have l training events,

$$(y_1, \boldsymbol{x}_1), \ldots, (y_l, \boldsymbol{x}_l),$$

This work is supported by MOST of Taiwan via grants 104-2622-E-002-012-CC2 and 104-2221-E-002-047-MY3 and MOE of Taiwan via grants 104R7872 and 105R7872.

Authors' email: {d01944006,f03944049,b01902037,cjlin}@csie.ntu.edu.tw
Author's addresses: Wei-Sheng Chin, Graduate Institute of Networking and Multimedia, National Taiwan University; Bo-Wen Yuan, Meng-Yuan Yang, and Chih-Jen Lin, Department of Computer Science, National Taiwan University

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 0 ACM. 2157-6904/0/-ART0 $15.00
`DOI:` 0000001.0000001

where y_i and $\boldsymbol{x}_i$ are the label and the feature vector of the ith training instance, respectively. For an instance $(y, \boldsymbol{x})$, the output value of FM is defined by

$$\hat{y}(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + \sum_{j=1}^{n}\sum_{j'=j+1}^{n} \boldsymbol{u}_j^T\boldsymbol{u}_{j'}x_jx_{j'}, \tag{1}$$

where $\boldsymbol{w} \in \mathcal{R}^n$ and $U = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n] \in \mathcal{R}^{d\times n}$ are the parameters of FM. The jth column of U, $\boldsymbol{u}_j$, can be viewed as the jth feature's representation in an latent space with the dimensionality d specified by the user. In (1), the variable $\boldsymbol{w}$ describes the linear relation between the input features and the output label, while the second term of (1) includes all feature conjunctions that capture the information carried by feature co-occurrences (also called feature interactions or feature conjunctions). Moreover, the coefficient of the conjunction between x_j and $x_{j'}$ is determined by the inner product of $\boldsymbol{u}_j$ and $\boldsymbol{u}_{j'}$. To determine FM's parameters, we solve the following optimization problem.

$$\min_{\boldsymbol{w}, U} \quad \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{\lambda'}{2}\|U\|_F^2 + \sum_{i=1}^{l}\xi(\hat{y}(\boldsymbol{x}_i); y_i). \tag{2}$$

With regularization parameters λ and λ', FM's objective function consists of the squared sum of all model parameters for preventing over-fitting and the sum of l loss values. The loss function $\xi(\cdot)$ encourages the consistency between $\hat{y}(\boldsymbol{x})$ and the actual label y. For practical classification problems, one can use, for example, the logistic loss or the squared loss:

$$\xi_{\text{log}}(\hat{y}; y) = \log\left(1 + e^{-y\hat{y}}\right) \tag{3}$$

$$\xi_{\text{sq}}(\hat{y}; y) = \frac{1}{2}(\hat{y} - y)^2. \tag{4}$$

Note that the regression-oriented loss (4) can be adopted here for classification because it panellizes the distance between the predicted value and the true label. Furthermore, if U is not considered, using any of (3) and (4) leads to logistic regression (or LR for short) and linear regression, respectively.

By following the concept of computing the conjunction coefficient of a feature pair from their latent representations, some FM variants have been proposed. For example, Blondel et al. [2016] proposed modifying the output function by introducing some new variables, $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$, and change output value in (1) to

$$\begin{aligned}\hat{y}(\boldsymbol{x}) &= \boldsymbol{w}^T\boldsymbol{x} + \frac{1}{2}\sum_{j=1}^{n}\sum_{j'=1}^{n} \boldsymbol{u}_j^T\boldsymbol{v}_{j'}x_jx_{j'} \\ &= \boldsymbol{w}^T\boldsymbol{x} + \frac{1}{2}\underbrace{(U\boldsymbol{x})^T(V\boldsymbol{x})}_{=\boldsymbol{x}^TU^TV\boldsymbol{x}},\end{aligned} \tag{5}$$

where $V = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n] \in \mathcal{R}^{d\times n}$ encodes the feature representations in another latent space. They replace U^TV with $\frac{1}{2}(U^TV + V^TU)$ in (5), but both of the output value and the optimization problem remain unchanged. In addition to the appearance of another latent space associated with V, (5) differs from (1) in the following aspects. First, the multiplications $x_jx_{j'}$ and $x_{j'}x_j$ are treated as different feature conjunctions, so they are characterized by different coefficients $\boldsymbol{u}_j^T\boldsymbol{v}_{j'}$ and $\boldsymbol{u}_{j'}^T\boldsymbol{v}_j$, respectively. Second, a feature index can occur twice in one conjunction; that is, self-interaction is allowed. Resulting from these two changes, the summation in (5) must iterate through all indexes in $\{(j, j') \mid j, j' = 1, \ldots, n\}$. Besides, Blondel et al. [2016] do not consider the

Table I: FM solvers which have been studied.

<table>
<tr><th>Loss
Solver</th><th>FM type</th><th>Squared loss</th><th>Logistic loss</th></tr>
<tr><td rowspan="2">SG</td><td>(2)</td><td>[Rendle 2010]</td><td>[Rendle 2010; Ta 2015; Juan et al. 2016]</td></tr>
<tr><td>(6)</td><td colspan="2">[Blondel et al. 2016]</td></tr>
<tr><td>MCMC</td><td>(2)</td><td colspan="2">[Rendle 2012]</td></tr>
<tr><td rowspan="2">CD</td><td>(2)</td><td>[Rendle 2012]</td><td>[Blondel et al. 2016]</td></tr>
<tr><td>(6)</td><td colspan="2">[Blondel et al. 2016]</td></tr>
<tr><td>L-BFGS</td><td>(2), (6)</td><td>[Blondel et al. 2016]</td><td></td></tr>
</table>

$1/2$ coefficient of the non-linear term in (5). We added it because the new formulation nearly doubles the number of feature conjunctions in (1). Finally, the new optimization problem associated with (5) is clearly formed by three blocks of variables $\boldsymbol{w}$, U, and V:

$$\min_{\boldsymbol{w},U,V} \quad F(\boldsymbol{w},U,V), \tag{6}$$

where

$$F(\boldsymbol{w},U,V) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{\lambda'}{2}\|U\|_F^2 + \frac{\lambda''}{2}\|V\|_F^2 + \sum_{i=1}^{l} \xi\left(y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + \frac{1}{2}(U\boldsymbol{x}_i)^T(V\boldsymbol{x}_i)\right)\right). \tag{7}$$

In this paper, we consider (5) because some of its properties are useful in developing our method. While general convex losses are included in the discussion we focus more on the logistic loss (3), which is commonly used for classification applications such as CTR (click-through rate) predictions.

Problems (2) and (5) are not easy to solve because of their non-convex objective functions. When the squared loss (4) is used, stochastic gradient methods (SG) and coordinate descent methods (CD) are two of the most effective techniques to train a model [Rendle 2012; Blondel et al. 2016]. If the logistic loss is applied, most existing FM packages use stochastic methods. For example, a Markov Chain Monte Carlo method (MCMC) and a classical stochastic gradient method are implemented in LIBFM [Rendle 2012] while LIBFFM [Juan et al. 2016] considers ADAGRAD, an advanced SG by Duchi et al. [2011]. Ta [2015] also reported the effectiveness of a variant of ADAGRAD. On the other hand, in designing a CD method for FM problem (6) with the squared loss, Blondel et al. [2016] mentioned that with some minor modifications, their method can solve FM problems with the logistic loss. Unfortunately no experimental result was provided. We give a summary in Table I for existing FM algorithms and the loss functions they have been implemented for. One original motivation of this study was to compare the two state-of-the-art algorithms CD and SG. However, in applying CD to solve FM with the logistic loss, we realize that CD suffers from a relatively large number of exponential/logarithmic operations. The CD procedure may not be competitive because each exponential/logarithmic operation costs much more than a basic operation like addition or multiplication. This issue has been well studied in the training of logistic regression [Yuan et al. 2012a]. They show that CD's ratio of $\exp / \log$ operations to standard arithmetic operations is $\mathcal{O}(1)$ for logistic regression. We point out in Section 2 that CD for FM has an equally high ratio. Yuan et al. [2012a] also show that Newton-type methods require a relatively small number of exponential/logarithmic operations. Therefore we decide to explore the use of Newton's methods in training FM. Our main results are summarized below:

- Because (7) is multi-block convex, we point out that if all but one block of variables are fixed, the optimization problem is reduced to be an LR-like sub-problem. We then develop an efficient Newton's method to solve the sub-problem.
- We explore in detail the use of Newton methods in solving each sub-problem of one block of variables. Some techniques are proposed to significantly reduce the running time.
- The parallelism of our method is investigated for the acceleration in multi-threading environments.
- We empirically demonstrate the extraordinary efficiency of the proposed algorithm against stochastic gradient methods and coordinate descent methods, targeting (6) with the logistic loss.

In addition to Newton methods, L-BFGS is a quasi Newton method that has been popular in training logistic regression. It has been used in Blondel et al. [2016] for training (6) with the squared loss. However, they directly apply L-BFGS to solve the non-convex problem (6) rather than convex sub-problems. Hence their scenario is different from ours. In their experiments. L-BFGS is inferior to CD and SG. Because for logistic regression Lin et al. [2008] have shown that Newton is competitive with L-BFGS, in this study we do not consider L-BFGS for solving our sub-problems.

The paper is organized as follows. Section 2 reviews existing state of the art FM solvers. We develop an efficient algorithm based on Newton methods in Section 3. Section 4 discusses some algorithmic issues and implementation techniques for our method. Then, we study its parallelization in Section 5. The experimental results in Section 6 confirm the effectiveness of the proposed method. Finally, Section 7 concludes our work.

2. STATE OF THE ART METHODS FOR TRAINING FACTORIZATION MACHINES

In this section, we review two state of the art FM solvers in detail, CD and SG. We focus on the FM problem (6) with the loss functions defined in (3) and (4). The reason of not considering (2) is that some of (6)'s nice properties are required in developing our method.

2.1. Stochastic Gradient Methods

It is mentioned in Section 1 that stochastic gradient methods are popular in training FM. Among existing SG approaches we describe one using ADAGRAD because its superiority over standard SG implemented in LIBFM has been established in [Juan et al. 2016].

Consider the aforementioned l training instances $(y_i, \boldsymbol{x}_i)$, $i = 1, \ldots, l$ again. ADAGRAD solves optimization problems in the following form.

$$\min_{\tilde{\boldsymbol{w}} \in \mathcal{R}^{\tilde{n}}} \quad \sum_{i=1}^{l} f(\tilde{\boldsymbol{w}}; y_i, \boldsymbol{x}_i), \tag{8}$$

where $\tilde{\boldsymbol{w}}$ is the collection of all variables. At the kth iteration, an instance $(y_i, \boldsymbol{x}_i)$ is sampled from the l training instances to calculate the stochastic gradient, $\tilde{\boldsymbol{g}}^k = \nabla f(\tilde{\boldsymbol{w}}^k; y_i, \boldsymbol{x}_i)$. Then, the current solution is updated via

$$\tilde{\boldsymbol{w}}^{k+1} \leftarrow \tilde{\boldsymbol{w}}^k - \eta_0 \tilde{G}^{-\frac{1}{2}} \tilde{\boldsymbol{g}}^k,$$

where $\eta_0 > 0$ is a pre-specified constant and $\tilde{G}$ is a diagonal matrix defined by

$$\tilde{G} = \sum_{k'=1}^{k} \operatorname{diag}\left(\tilde{\boldsymbol{g}}^{k'}\right)^2.$$

The complete procedure that ADAGRAD solves (8) is shown in Algorithm 1.

Algorithm 1 ADAGRAD.

```
Given initial solution $\tilde{\boldsymbol{w}}^0$, number of iterations $\bar{k}$, and $\eta_0 > 0$.
Initialize $\tilde{G}$ with zeros.
for $k \leftarrow 1, \ldots, \bar{k}$ do
    Draw an index $i \in \{1, \ldots, l\}$
    $\tilde{\boldsymbol{g}} \leftarrow \nabla f(\tilde{\boldsymbol{w}}^k; y_i, \boldsymbol{x}_i)$
    $\tilde{G} \leftarrow \tilde{G} + \operatorname{diag}(\tilde{\boldsymbol{g}})^2$
    $\tilde{\boldsymbol{w}}^{k+1} \leftarrow \tilde{\boldsymbol{w}}^k - \eta_0 \tilde{G}^{-1/2} \tilde{\boldsymbol{g}}$
end for
```

To apply ADAGRAD to learn the considered FM, we begin with rewriting (6) into a form of (8). Let

$$\tilde{\boldsymbol{w}} = \left[\boldsymbol{w}^T, \boldsymbol{u}_1^T, \ldots, \boldsymbol{u}_n^T, \boldsymbol{v}_1^T, \ldots, \boldsymbol{v}_n^T\right]^T \in \mathcal{R}^{n+2dn}. \tag{9}$$

We argue that if

$$f(\tilde{\boldsymbol{w}}; y_i, \boldsymbol{x}_i) = \sum_{j \in N_i} \left(\frac{\lambda}{2|\Omega_j|} w_j^2 + \frac{\lambda'}{2|\Omega_j|} \|\boldsymbol{u}_j\|^2 + \frac{\lambda''}{2|\Omega_j|} \|\boldsymbol{v}_j\|^2 \right) + \xi(\hat{y}(\boldsymbol{x}_i); y_i), \tag{10}$$

then (8) is equivalent to (6), where N_i is the index set of the ith instance's non-zero features, Ω_j is the indexes set of the instances whose jth features are not zero, and $|\cdot|$ returns the size when the input is a set. Because the summation of the loss terms in (10) over $i = 1, \ldots, l$ is obviously the loss term in (7), we only check if the regularization terms in (10) lead to the same regularization terms in (7). First, for $\boldsymbol{w}$'s regularization, we have

$$\sum_{i=1}^{l} \sum_{j \in N_i} \frac{\lambda}{2|\Omega_j|} w_j^2 = \frac{\lambda}{2} \sum_{j=1}^{n} \sum_{i \in \Omega_j} \frac{1}{|\Omega_j|} w_j^2 = \frac{\lambda}{2} \sum_{j=1}^{n} w_j^2 = \frac{\lambda}{2} \|\boldsymbol{w}\|^2.$$

By an analogous derivation, inversely scaling $\|\boldsymbol{u}_j\|^2$ and $\|\boldsymbol{v}_j\|^2$ by $|\Omega_j|$ produces the desired regularization in (7). The equivalence between (10) and (7) is therefore verified.

From (10) and the definition of $\tilde{\boldsymbol{w}}$ in (9), we deduce the stochastic gradient with respect to the model parameters in each iteration. For $\boldsymbol{w}$, we have

$$\frac{\partial f(\tilde{\boldsymbol{w}}; y_i, \boldsymbol{x}_i)}{\partial w_j} = \begin{cases} \frac{\lambda}{|\Omega_j|} w_j + \xi'(\hat{y}(\boldsymbol{x}_i); y_i) x_{ij} & \text{if } j \in N_i, \\ 0 & \text{otherwise,} \end{cases} \tag{11}$$

where $\xi'(\hat{y}; y)$ is the derivative of the selected loss function with respect to $\hat{y}$. Note that

$$\begin{aligned} \xi'_{\text{log}}(\hat{y}; y) &= \frac{-y}{1 + e^{y\hat{y}}} \\ \xi'_{\text{sq}}(\hat{y}; y) &= \hat{y} - y. \end{aligned} \tag{12}$$

To simplify subsequent notations, we define $\boldsymbol{p}_i = U\boldsymbol{x}_i$ and $\boldsymbol{q}_i = V\boldsymbol{x}_i$. Taking the partial derivative of (10) with respect to $\boldsymbol{u}_j$ and $\boldsymbol{v}_j$ yields the derivatives along other coordinates.

$$
\begin{aligned}
\frac{\partial f(\tilde{\boldsymbol{w}}; y_i, \boldsymbol{x}_i)}{\partial \boldsymbol{u}_j} &= \begin{cases} \frac{\lambda'}{|\Omega_j|}\boldsymbol{u}_j + \frac{1}{2}\xi'(\hat{y}(\boldsymbol{x}_i); y_i)\boldsymbol{q}_i x_{ij} & \text{if } j \in N_i, \\ \boldsymbol{0}_{d\times 1} & \text{otherwise.} \end{cases} \\
\frac{\partial f(\tilde{\boldsymbol{w}}; y_i, \boldsymbol{x}_i)}{\partial \boldsymbol{v}_j} &= \begin{cases} \frac{\lambda''}{|\Omega_j|}\boldsymbol{v}_j + \frac{1}{2}\xi'(\hat{y}(\boldsymbol{x}_i); y_i)\boldsymbol{p}_i x_{ij} & \text{if } j \in N_i, \\ \boldsymbol{0}_{d\times 1} & \text{otherwise.} \end{cases}
\end{aligned}
\tag{13}
$$

Substituting (11) and (13) into the step that calculates the stochastic gradient in Algorithm 1, we get the procedure to solve (6) by ADAGRAD.

Next, we analyze required computational complexity in one for-loop iteration in Algorithm 1. Taking a closer look at (12), one may quickly realize that computing the derivative of (3) is much more slower than calculating the other components. The chief reason is that exponential and logarithmic function evaluations are much more expensive than standard arithmetic operations. In contrast, (4)'s evaluation is not especially high-priced. Therefore, the amounts of loss-related function calls and other operations are separately discussed. If the instance $(y_i, \boldsymbol{x}_i)$ is drawn, we first spend $\mathcal{O}(|N_i|d)$ operations to calculate and store $\boldsymbol{p}_i$, $\boldsymbol{q}_i$ and then the output value $\hat{y}_i$ via

$$
(5) = \boldsymbol{w}^T\boldsymbol{x}_i + \frac{1}{2}\boldsymbol{p}_i^T\boldsymbol{q}_i. \tag{14}
$$

The following step is substituting $\hat{y}_i$ into $\xi'(\hat{y}_i; y)$ and caching its output. Next, we use (11)-(13) to compute the $(|N_i|+2|N_i|d)$ non-zero values of $\tilde{\boldsymbol{g}}$. Then, the diagonal elements of $\tilde{G}$ corresponding to the non-zero coordinates in $\tilde{\boldsymbol{g}}$ are updated. Finally, new values are computed and assigned to the coordinates of $\tilde{\boldsymbol{w}}$ linked to the non-zero coordinates in $\tilde{\boldsymbol{g}}$. The total arithmetic complexity in one iteration is therefore $\mathcal{O}(|N_i|d)$ with only one call to the loss function's derivative. Let (# nnz) denote the total number of non-zeros in X. To process one epoch (i.e., l instances), we expect $\mathcal{O}(d \times (\text{\# nnz}))$ operations and l loss-related function calls.

From a practical perspective, selecting proper storage formats for the used variables is important in obtaining an efficient implementation. A well-known principle is to use one leading to shorter physical distance between two consecutively-accessed variables. In Algorithm 1, the major matrices are U, V, and

$$
X = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_l]^T \in \mathcal{R}^{l\times n}.
$$

For each of them, we can use row-major or column-major formats. Since ADAGRAD accesses X instance-by-instance, it should be stored in a row-major format (e.g., compressed row format because X is usually sparse in large-scale problems). Because the coordinates in $\boldsymbol{u}_j$, $j \in N_i$ are sequentially accessed when computing (13), a column-major format is suggested for U. Similarly, V is stored in a column-major format. It follows that the matrices of accumulated gradients are also in column-major.

2.2. Coordinate Descent Methods

Coordinate descent methods, a family of iterative procedures, have gained great successes in solving large-scale minimization problems. At each iteration, only one variable is updated by solving an associated sub-problem while other variables are fixed. For the update sequence, here we choose a cyclic one by following [Blondel et al. 2016]. That is, we cyclically go through

$$
w_1, \ldots, w_n, \underbrace{U_{11}, U_{12}, \ldots, U_{1n}}_{\text{the 1st row of } U}, \underbrace{V_{11}, V_{12}, \ldots, V_{1n}}_{\text{the 1st row of } V}, \ldots, \underbrace{U_{d1}, U_{d2}, \ldots, U_{dn}}_{\text{the } d\text{th row of } U}, \underbrace{V_{d1}, V_{d2}, \ldots, V_{dn}}_{\text{the } d\text{th row of } V}. \tag{15}
$$

For forming a sub-problem and obtaining its solution, we leverage the concept in state of the art methods developed for LR and support vector machine [Chang et al. 2008; Huang et al. 2010]. The situations for updating the coordinates of $\boldsymbol{w}$, U, and V are quite similar. Let us first outline the update of U_{cj}, the value at the cth row and the jth column of U, and then extend the result to other cases. By fixing all variables but U_{cj} in (6), we obtain U_{cj}'s sub-problem,

$$\min_{U_{cj} \in \mathcal{R}} \quad F^U_{cj}(U_{cj}), \tag{16}$$

where

$$F^U_{cj}(U_{cj}) = \frac{\lambda'}{2} U^2_{cj} + \sum_{i \in \Omega_j} \xi(y_i; \hat{y}_i).$$

For solving (16), a single-variable Newton method is employed; that is, if the current solution is U_{cj}, the update direction would be

$$s = -g/h, \tag{17}$$

where

$$\begin{aligned} g &= \frac{\partial F^U_{cj}}{\partial U_{cj}} = \lambda' U_{cj} + \frac{1}{2} \sum_{i \in \Omega_j} \xi'(\hat{y}_i; y_i)(\boldsymbol{q}_i)_c(\boldsymbol{x}_i)_j \\ h &= \frac{\partial^2 F^U_{cj}}{\partial^2 U_{cj}} = \lambda' + \frac{1}{4} \sum_{i \in \Omega_j} \xi''(\hat{y}_i; y_i)(\boldsymbol{q}_i)^2_c(\boldsymbol{x}_i)^2_j. \end{aligned} \tag{18}$$

For the second-order derivatives, we note

$$\begin{aligned} \xi''_{\text{log}}(\hat{y}; y) &= \frac{e^{y\hat{y}}}{(1 + e^{y\hat{y}})^2} \\ \xi''_{\text{sq}}(\hat{y}; y) &= 1. \end{aligned} \tag{19}$$

If

$$\begin{aligned} P &= [\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_l] \in \mathcal{R}^{d \times l} \\ Q &= [\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_l] \in \mathcal{R}^{d \times l} \end{aligned}$$

are available, $\mathcal{O}(|\Omega_j|)$ arithmetic and loss-related operations are involved in the evaluation of (18). Then, to guarantee a sufficient objective function reduction and hence the convergence, a back-tracking line search is lunched to find the largest step size $\theta \in \{1, \beta, \beta^2, \ldots\}$ satisfying Armijo rule,

$$F^U_{cj}(U_{cj} + \theta s) - F^U_{cj}(U_{cj}) \leq \theta \nu g s, \tag{20}$$

where $\nu \in (0, 1)$ and $\beta \in (0, 1)$ are pre-specified constants. If $\hat{y}_i$, $i \in \Omega_j$ are available, computing the left-hand side in (20) can be finished within $\mathcal{O}(|\Omega_j|)$ arithmetic operations and loss-related function calls. Precisely,

$$\begin{aligned} F^U_{cj}(U_{cj} + \theta s) - F^U_{cj}(U_{cj}) = &\frac{\lambda'}{2} \left(2\theta U_{cj} s + \theta^2 s^2\right) + \sum_{i \in \Omega_j} \xi(\hat{y}_i + \theta \Delta_i; y_i) \\ &- \sum_{i \in \Omega_j} \xi(\hat{y}_i; y_i), \end{aligned} \tag{21}$$

Algorithm 2 CD for solving (16).

```
Given iteration number $\bar{k}_{\text{inner}}$, $\lambda' > 0$, $0 < \nu < 1$, $0 < \beta < 1$, the current $U_{cj}$, $\hat{y}_i$, $(\boldsymbol{p}_i)_c$,
and $(\boldsymbol{q}_i)_c$, $i = 1, \ldots, l$.
for $k \leftarrow \{1, \ldots, \bar{k}_{\text{inner}}\}$ do
    Evaluate (17) to obtain $s$
    $\Delta_i \leftarrow s(\boldsymbol{q}_i)_c(\boldsymbol{x}_i)_j/2, \forall i \in \Omega_j$
    for $\theta \leftarrow \{1, \beta, \beta^2, \ldots\}$ do
        if (21) $\leq \theta \nu g s$ then
            $U_{cj} \leftarrow U_{cj} + \theta s$
            Maintain variables via (22), $\forall i \in \Omega_j$.
            break
        end if
    end for
end for
Output $U_{cj}$ as the solution of (16).
```

where $\Delta_i = 0.5 \times s(\boldsymbol{q}_i)_c(\boldsymbol{x}_i)_j$ because in (5),

$$(U\boldsymbol{x})^T(V\boldsymbol{x}) = \boldsymbol{q}^T U\boldsymbol{x} = \sum_{c=1}^{d}\sum_{j=1}^{n} q_c U_{cj} x_j$$

and only U_{cj} is being adjusted along s. In addition, Δ_i, $i \in \Omega_j$ can be pre-computed in the beginning and reused in checking every θ. Once θ is found, the variable U_{cj} is adjusted via

$$U_{cj} \leftarrow U_{cj} + \theta s.$$

It is known that for an efficient CD implementation, $\boldsymbol{p}_i$, $\boldsymbol{q}_i$, and $\hat{y}_i$, $\forall i$ should be cached and carefully maintained through iterations to save computation [Blondel et al. 2016]. The update of U_{cj} causes that the cth coordinate at $\boldsymbol{p}_i$ and $\hat{y}_i$, $i \in \Omega_j$ are changed. Recomputing $\boldsymbol{p}_i$ and $\hat{y}_i$ for each $i \in \Omega_j$ may need $\mathcal{O}(|N_i|d)$ operations, but all necessary changes can be done via an $\mathcal{O}(|\Omega_j|)$ rule below.

$$\begin{array}{r}(\boldsymbol{p}_i)_c \leftarrow (\boldsymbol{p}_i)_c + \theta s(\boldsymbol{x}_i)_j \\ \hat{y}_i \leftarrow \hat{y}_i + \theta\Delta_i\end{array}, \ \forall i \in \Omega_j. \tag{22}$$

Algorithm 2 sketches our CD procedure of solving (16). Besides, if $\xi(\hat{y}; y)$ is μ-smooth, letting

$$h = \lambda' + \frac{\mu}{4}\sum_{i \in \Omega_j}(\boldsymbol{q}_i)_c^2(\boldsymbol{x}_i)_j^2 \tag{23}$$

can guarantee the monotonic decrease of the objective function with $\theta = 1$. It leads to the CD procedure without line search in [Blondel et al. 2016]. Nevertheless, replacing the calculation of h in (18) with (23) does not change the algorithm's computation complexity, so we use (18) for more accurate second-order information following a state of the art LR solver [Huang et al. 2010]. All iterative algorithms need a stopping condition. For Algorithm 2, a fixed number of iterations $\bar{k}_{\text{inner}}$ is set. Concerning (18), (21), and (22), Algorithm 2 yields a computational cost at

$$\mathcal{O}\left(\bar{k}_{\text{inner}} \times (\text{\# of line search iterations} + 2) \times |\Omega_j|\right), \tag{24}$$

Algorithm 3 Solving (6) via CD.

```
Given an initial solution $(\boldsymbol{w}, U, V)$ and number of outer iterations $\bar{k}_{\text{outer}}$.
Calculate and cache $\boldsymbol{p}_i$, $\boldsymbol{q}_i$, and $\hat{y}_i = \boldsymbol{w}^T\boldsymbol{x}_i + \frac{1}{2}\boldsymbol{p}_i^T\boldsymbol{q}_i$, $i = 1, \ldots, l$.
for $k \leftarrow \{1, \ldots, \bar{k}_{\text{outer}}\}$ do
    for $j \leftarrow \{1, \ldots, n\}$ do
        Update $w_j$ via Algorithm 2 by replacing $(\lambda', U_{cj}, (\boldsymbol{q}_i)_c)$ with $(\lambda, w_j, 2)$ and discarding the first rule in (22).
    end for
    for $c \leftarrow \{1, \ldots, d\}$ do
        for $j \leftarrow \{1, \ldots, n\}$ do
            Update $U_{cj}$ via Algorithm 2.
        end for
        for $j \leftarrow \{1, \ldots, n\}$ do
            Update $V_{cj}$ via Algorithm 2 by replacing $(\lambda', U_{cj}, (\boldsymbol{p}_i)_c)$ with $(\lambda'', V_{cj}, (\boldsymbol{q}_i)_c)$.
        end for
    end for
end for
```

where the coefficient “2” accounts for the evaluation of (18) and (22). If all coordinates of U are updated once, the induced complexity is

$$\mathcal{O}\left(d\bar{k}_{\text{inner}} \times (\text{\# of expected line search iterations} + 2) \times (\text{\# nnz})\right). \tag{25}$$

As none of loss-related operations is present in (22), the total number of loss-related function calls in Algorithm 2 and in the process of updating U once can be simply obtained by changing the coefficient “2” in (24) and (25) to “1,” respectively.

With some minor changes, Algorithm 2 can be used to update the other coordinates. For V_{cj}, we just need to replace $(\lambda', U_{cj}, (\boldsymbol{p}_i)_c, (\boldsymbol{q}_i)_c)$ with $(\lambda'', V_{cj}, (\boldsymbol{q}_i)_c, (\boldsymbol{p}_i)_c)$. For w_j, the strategy is to replace $(\lambda', U_{cj}, (\boldsymbol{p}_i)_c, (\boldsymbol{q}_i)_c)$ with $(\lambda, w_j, 2, 2)$ and disable the adjustment of $(\boldsymbol{p}_i)_c$ in (22). All combined, Algorithm 3 demonstrates the final two-layer CD framework for solving (6). Because the adjustment of U and V costs more than the update of $\boldsymbol{w}$, the complexity of updating all coordinates once (called an outer iteration) is (25) with loss-related function calls at the same level. Besides, in Algorithm 3, the initialization step requires $\mathcal{O}(d \times (\text{\# nnz}))$ extra operations to prepare $\boldsymbol{p}_i$, $\boldsymbol{q}_i$, and $\hat{y}_i$, $i = 1, \ldots, l$.

Finally, we discuss which of row- and column-major orders should be obeyed when storing the elements in X, U, V, P, and Q. Recall the three major steps in Algorithm 2: (18), the line search procedure (including the preparation of Δ_i, $\forall i \in \Omega_j$), and (22). For each step, we observe that aligning elements in Q in row-major order may increase memory continuity when accessing $(\boldsymbol{q}_i)_c$, resulting from its summations over the instance indexes in Ω_j. This argument also holds for P when updating V. Thus, both of P and Q should be stored in row-major format. Because we are always looping through $(\boldsymbol{x}_i)_j$, $i \in \Omega_j$, X should be in column-major format. From the update sequence specified by (15), row-major format is recommended for both of U and V.

3. ALTERNATING NEWTON METHODS FOR LEARNING FACTORIZATION MACHINES

From Sections 2.1-2.2, we know that CD's and SG's ratios of loss-related function calls to standard arithmetic operations are $\mathcal{O}(1)$ and $\mathcal{O}\left(\frac{l}{d \times \text{\# nnz}}\right)$, respectively. It implies that when the loss function is composed of some expensive operations, the running time of CD can be heavily prolonged. For Newton-type methods, solving the direction-

Algorithm 4 Solving the modified FM problem (6) via alternating minimization.

```
Given an initial solution $(\boldsymbol{w}, U, V)$
while stopping condition is not satisfied do
    $\boldsymbol{w} \leftarrow \arg\min_{\boldsymbol{w}} F(\boldsymbol{w}, U, V)$
    $U \leftarrow \arg\min_{U} F(\boldsymbol{w}, U, V)$
    $V \leftarrow \arg\min_{V} F(\boldsymbol{w}, U, V)$
end while
```

finding sub-problem is usually the heaviest step in each iteration and, fortunately, costs no loss-related operation. Thus, it may be a more suitable method than CD for tackling the logistic loss. For logistic regression, the superiority of Newton over CD has been confirmed in Yuan et al. [2012a]. However, the non-convexity of (2) and (6) brings some difficulties in developing a Newton-type FM solver. The Hessian matrix (the second-order derivative) is not always positive definite, so a direct application of the Newton method can fail to find a descent direction. We may consider a positive-definite approximation of the Hessian matrix such as the Gauss-Newton matrix [Schraudolph 2002] used in deep learning [Martens 2010], but such an approach is generally less effective than Newton methods for convex optimization problems. On the other hand, for problems related to matrix factorization, the objective function is multi-block convex, so alternating Newton methods have been widely applied. That is, sequentially a convex sub-problem of one block of variables is solved by the Newton method while other variables are fixed. In fact, if the squared loss is used, then the alternating Newton method is reduced to the alternating least square method (e.g., [Zhou et al. 2008]) because the Newton method solves a quadratic problem in one iteration.

Unfortunately, the original FM problem (2) is not in a multi-block convex form so that each block contains many variables. The reason is that every $\boldsymbol{u}_j^T \boldsymbol{u}_{j'}$ in (1) is a non-convex term with respect to $\boldsymbol{u}_j$ and $\boldsymbol{u}_{j'}$. Then, we can only get a small convex sub-problem by fixing all but one $\boldsymbol{u}_j$. On the other hand, it has been mentioned in [Blondel et al. 2016] that (7) is a multi-block convex function of $\boldsymbol{w}$, U, and V. That is, (7) becomes a convex sub-problem if two of $\boldsymbol{w}$, U, and V are fixed. Then the idea of alternating minimization by using any convex optimization algorithm to solve each sub-problem can be applied. In the rest of this paper we consider the optimization problem (6) of using the new model in (5) rather than the original FM optimization problem.

For the update sequence of the block minimization, it is possible to have a dynamic scheme, but for the sake of simplicity we focus on the cyclic one. Iteratively the following three optimization sub-problems are solved.

$$\min_{\boldsymbol{w}} \quad F(\boldsymbol{w}, U, V), \tag{26}$$

$$\min_{U} \quad F(\boldsymbol{w}, U, V), \tag{27}$$

$$\min_{V} \quad F(\boldsymbol{w}, U, V). \tag{28}$$

The overall procedure to minimize (6) is summarized in Algorithm 4. In the rest of this section, we discuss how to apply Newton-type methods to solve the sub-problems. The resulting procedure is called ANT (alternating Newton method). We begin with showing that each sub-problem is in a form similar to a linear classification problem.

We emphasize that our method is especially designed to alleviate CD's weakness induced by using the logistic loss, though the discussion is applicable to computationally cheaper losses such as the squared loss.

3.1. Relation Between (6)'s Sub-problems and Linear Classification

Although (26)-(28) are different optimization problems, we show that they are all linear classification problems by the change of variables. Given l training instances, a linear classification problem can be written as

$$\min_{\tilde{\boldsymbol{w}} \in \mathcal{R}^{\tilde{n}}} \quad \tilde{f}(\tilde{\boldsymbol{w}}), \tag{29}$$

where

$$\begin{aligned} \tilde{f}(\tilde{\boldsymbol{w}}) &= \frac{\tilde{\lambda}}{2}\|\tilde{\boldsymbol{w}}\|^2 + \sum_{i=1}^{l} \xi(\tilde{y}_i; y_i), \\ \tilde{y}_i &= \tilde{\boldsymbol{w}}^T \tilde{\boldsymbol{x}}_i + \tilde{c}_i, \end{aligned} \tag{30}$$

$\tilde{\boldsymbol{x}}_i$ is the feature vector of the ith instance, and $\tilde{c}_i$ is a constant.[1]

From (14), it is straightforward to see that (26) is in a form of (29) when

$$\tilde{\lambda} = \lambda, \ \tilde{\boldsymbol{w}} = \boldsymbol{w}, \ \tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i, \text{ and } \tilde{c}_i = \frac{1}{2}\boldsymbol{p}_i^T \boldsymbol{q}_i.$$

Note that constants such as $\|U\|_F^2$ and $\|V\|_F^2$ are not considered when updating $\boldsymbol{w}$. To write (27) in a form of (29), we note that when $\boldsymbol{w}$ and V are fixed, (14) can be reformulated as an affine function of U,

$$\tilde{y}_i = \frac{1}{2}(U\boldsymbol{x}_i)^T \boldsymbol{q}_i + \boldsymbol{w}^T \boldsymbol{x}_i = \operatorname{vec}(U)^T (\frac{1}{2}(\boldsymbol{x}_i \otimes \boldsymbol{q}_i)) + \boldsymbol{w}^T \boldsymbol{x}_i,$$

where the vectorization operator outputs a column vector by stacking the input matrix's columns and "$\otimes$" denotes the Kronecker product. With $\|\operatorname{vec}(U)\|^2 = \|U\|_F^2$, (27) is in the form of (29) by

$$\tilde{\lambda} = \lambda', \ \tilde{\boldsymbol{w}} = \operatorname{vec}(U), \ \tilde{\boldsymbol{x}}_i = \frac{1}{2}(\boldsymbol{x}_i \otimes \boldsymbol{q}_i), \text{ and } \tilde{c}_i = \boldsymbol{w}^T \boldsymbol{x}_i. \tag{31}$$

Similarly, (29) becomes (28) if

$$\tilde{\lambda} = \lambda'', \ \tilde{\boldsymbol{w}} = \operatorname{vec}(V), \ \tilde{\boldsymbol{x}}_i = \frac{1}{2}(\boldsymbol{x}_i \otimes \boldsymbol{p}_i), \text{ and } \tilde{c}_i = \boldsymbol{w}^T \boldsymbol{x}_i. \tag{32}$$

We can conclude that (29) can be used to represent all sub-problems in Algorithm 4, so the same optimization algorithm can be used to solve them. We give the gradient and the Hessian-matrix of (29) below for applying optimization algorithms.

$$\nabla \tilde{f}(\tilde{\boldsymbol{w}}) = \tilde{\lambda}\tilde{\boldsymbol{w}} + \tilde{X}^T \boldsymbol{b} \tag{33}$$

$$\nabla^2 \tilde{f}(\tilde{\boldsymbol{w}}) = \tilde{\lambda} I + \tilde{X}^T D \tilde{X}, \tag{34}$$

where $\tilde{X} = [\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_l]^T \in \mathcal{R}^{l \times \tilde{n}}$ is the data matrix, $\boldsymbol{b} = [\xi'(\tilde{y}_1; y_1), \ldots, \xi'(\tilde{y}_l; y_l)]^T \in \mathcal{R}^l$, and D is a diagonal matrix whose ith diagonal element is $\xi''(\tilde{y}_i; y_i)$. See (12) and (19) for $\xi'(\tilde{y}; y)$'s and $\xi''(\tilde{y}; y)$'s definitions. With $\tilde{\lambda} > 0$, it is clear that (34) is positive definite, so the optimal solution of (29) is unique.

---

[1] In standard linear classification problems, $\tilde{c}_i = 0$. Note that we do not consider a bias term in the output function.

3.2. Truncated Newton Methods for Solving (29)

We discuss a truncated Newton method to solve (29). At the kth iteration, Newton method obtains an update direction by minimizing the second-order Taylor expansion at $\tilde{f}(\tilde{\boldsymbol{w}}^k)$.

$$\min_{\tilde{\boldsymbol{s}} \in \mathcal{R}^{\tilde{n}}} \quad \tilde{\boldsymbol{g}}^T \tilde{\boldsymbol{s}} + \frac{1}{2} \tilde{\boldsymbol{s}}^T \tilde{H} \tilde{\boldsymbol{s}}, \tag{35}$$

where $\tilde{\boldsymbol{w}}^k$ is the solution to be improved, $\tilde{H} = \nabla^2 \tilde{f}(\tilde{\boldsymbol{w}}^k)$, and $\tilde{\boldsymbol{g}} = \nabla \tilde{f}(\tilde{\boldsymbol{w}}^k)$. Because $\tilde{H}$ is positive definite, (35) is equivalent to a linear system,

$$\tilde{H} \tilde{\boldsymbol{s}} = -\tilde{\boldsymbol{g}}. \tag{36}$$

In practice, (36) may not need to be exactly solved, especially in early iterations. Therefore, truncated Newton methods have been introduced to approximately solve (36) while maintain the convergence [Nash 2000]. Iterative methods such as conjugate gradient method (CG) are often used for approximately solving (36), so at each iteration an inner iterative process is involved. Another difficulty to solve (36) is that storing the Hessian matrix $\tilde{H}$ in $\mathcal{O}\left(\tilde{n}^2\right)$ space is not possible if $\tilde{n}$ is large. To overcome the memory issue, Komarek and Moore [2005] and Keerthi and DeCoste [2005] showed that for problems like (29), the conjugate gradient method that mainly requires a sequence of Hessian-vector products can be performed without explicitly forming the Hessian matrix:

$$\tilde{H} \tilde{\boldsymbol{s}} = \lambda \tilde{\boldsymbol{s}} + \tilde{X}^T (D(\tilde{X} \tilde{\boldsymbol{s}})), \tag{37}$$

where $\tilde{s}$ is the iterate in the CG procedure. In [Lin et al. 2008], this idea is further incorporated into a truncated Newton framework.

Like coordinate descent methods and many other optimization algorithms, after a direction is found in our Newton method, we must decide the step size taken along that direction. Here we consider the back-tracking line-search procedure mentioned in Section 2.2 and discuss a trick sharing the same spirit with (21) for efficient computation. Concerning (29) and the $\tilde{n}$-dimensional search direction, the line search's stopping condition specified in (20) becomes

$$\tilde{f}(\tilde{\boldsymbol{w}}^k + \theta \tilde{\boldsymbol{s}}) - \tilde{f}(\tilde{\boldsymbol{w}}^k) \leq \theta \nu \tilde{\boldsymbol{g}}^T \tilde{\boldsymbol{s}}. \tag{38}$$

Recalculating the function value at each $\tilde{\boldsymbol{w}}^k + \theta \tilde{\boldsymbol{s}}$ is expensive, where the main cost is on calculating $(\tilde{\boldsymbol{w}} + \theta \tilde{\boldsymbol{s}})^T \tilde{\boldsymbol{x}}_i, \forall i$. However, for linear models, the following trick in [Yuan et al. 2010] can be employed. Assume that

$$\tilde{y}_i = (\tilde{\boldsymbol{w}}^k)^T \tilde{\boldsymbol{x}}_i + \tilde{c}_i \text{ and } \tilde{\Delta}_i = \tilde{\boldsymbol{s}}^T \tilde{\boldsymbol{x}}_i, \ i = 1, \ldots, l \tag{39}$$

are available. At an arbitrary θ, we can calculate

$$\left(\tilde{\boldsymbol{w}}^k + \theta \tilde{\boldsymbol{s}}\right)^T \tilde{\boldsymbol{x}}_i + \tilde{c}_i = \tilde{y}_i + \theta \tilde{\Delta}_i \tag{40}$$

to get the new output value. By further maintaining

$$(\tilde{\boldsymbol{w}}^k)^T \tilde{\boldsymbol{s}}, \ \|\tilde{s}\|^2, \text{ and } \tilde{\boldsymbol{g}}^T \tilde{\boldsymbol{s}}, \tag{41}$$

the total cost of checking the condition of (38) is merely $\mathcal{O}\left(l\right)$ because

$$\begin{aligned} \tilde{f}(\tilde{\boldsymbol{w}}^k + \theta \tilde{\boldsymbol{s}}) - \tilde{f}(\tilde{\boldsymbol{w}}^k) = & \frac{\tilde{\lambda}}{2} \left(2\theta (\tilde{\boldsymbol{w}}^k)^T \tilde{\boldsymbol{s}} + \theta^2 \|\tilde{s}\|^2\right) + \sum_{i=1}^{l} \xi(\tilde{y}_i + \theta \Delta_i; y_i) \\ & - \sum_{i=1}^{l} \xi(\tilde{y}_i; y_i). \end{aligned} \tag{42}$$

Algorithm 5 A CG-based truncated Newton method with line search for solving (29).

```
1: Given initial value of $\tilde{\boldsymbol{w}}^0$ and $0 < \tilde{\epsilon} < 1$.
2: Compute and cache $\tilde{y}_i$, $\forall i$.
3: $f^0 \leftarrow \tilde{f}(\tilde{\boldsymbol{w}}^0)$
4: for $k \leftarrow \{0, 1, \dots\}$ do
5:     Calculate and store $b_i$ (and $e^{y_i \tilde{y}_i}$ if (3) is the loss) using (12), $\forall i$.
6:     $\tilde{\boldsymbol{g}} \leftarrow \tilde{\lambda}\tilde{\boldsymbol{w}} + \tilde{X}^T\boldsymbol{b}$
7:     if $k = 0$ then
8:         $\|\tilde{\boldsymbol{g}}^0\| \leftarrow \|\tilde{\boldsymbol{g}}\|$
9:     end if
10:    if $\|\tilde{\boldsymbol{g}}\| \leq \tilde{\epsilon}\|\tilde{\boldsymbol{g}}^0\|$ then
11:        Output $\tilde{\boldsymbol{w}}^k$ as the solution.
12:        break
13:    end if
14:    Compute $D_{ii}$ via (19), $\forall i$.
15:    Solve (36) approximately via CG to get an update direction $\tilde{\boldsymbol{s}}$.
16:    Calculate variables listed in (41) and $\tilde{\Delta}_i$, $\forall i$.
17:    for $\theta \leftarrow \{1, \beta, \beta^2, \dots\}$ do
18:        Compute $\delta = f(\tilde{\boldsymbol{w}}^k + \theta\tilde{\boldsymbol{s}}) - f^k$ via (42).
19:        if $\delta \leq \nu\theta\tilde{\boldsymbol{g}}^T\tilde{\boldsymbol{s}}$ then
20:            $\tilde{\boldsymbol{w}}^{k+1} \leftarrow \tilde{\boldsymbol{w}}^k + \theta\tilde{\boldsymbol{s}}$
21:            $f^{k+1} \leftarrow f^k + \delta$
22:            Let the last value calculated in (40) be the next $\tilde{y}_i$, $\forall i$
23:            break
24:        end if
25:    end for
26: end for
```

In the end of the line-search procedure, the model is updated by

$$\tilde{\boldsymbol{w}}^{k+1} \leftarrow \tilde{\boldsymbol{w}}^k + \theta\tilde{\boldsymbol{s}},$$

and the last value obtained in (40) can be can be used as the new $\tilde{y}_i$ in the next iteration; see (39). When $\tilde{y}_i$, $\forall i$ are available, we immediately obtain the elements in $\boldsymbol{b}$ and D according to (12) and (19), respectively. Notice that if the logistic loss is considered, a temporal variable $e^{y_i \tilde{y}_i}$ which participates in the computation of b_i can be cached and reused in the computation of D_{ii} to save some exponential operations.

Algorithm 5 summarizes the above procedure. The computational complexity of one iteration in Algorithm 5 can be estimated by

$$(\text{cost of } \tilde{f} \text{ and } \tilde{\boldsymbol{g}}) + (\text{\# of CG iterations}) \times (\text{cost of } \tilde{H}\tilde{\boldsymbol{s}}) + (\text{\# of line search iterations}) \times \mathcal{O}(l). \tag{43}$$

Note that the above line-search procedure is also responsible for calculating the next function value, so in (43), except the $\mathcal{O}(l)$ cost for calculating (40) at each line-search step, we consider all initialization tasks such as calculating $\tilde{\Delta}_i$, $\forall i$ in (39) and (41) as the cost of evaluating the objective function. The complexity of evaluating the objective function also absorbs the cost of calculating b_i and D_{ii}, $i = 1, \dots, l$. Since there are $\mathcal{O}(l)$ at each of Lines 5, 14, and 18, the number of total loss-related function calls in one iteration is

$$\mathcal{O}((\text{\# of line search iterations} + 2) \times l). \tag{44}$$

Although we do not exactly solve (36), Algorithm 5 asymptotically converges to the optimal point if $\tilde{\boldsymbol{s}}$ satisfies

$$\|\tilde{H}\tilde{\boldsymbol{s}} + \tilde{\boldsymbol{g}}\| \leq \eta\|\tilde{\boldsymbol{g}}\|,$$

where $0 < \eta < 1$ can be either a pre-specified constant or a number controlled by a dynamic strategy [Eisenstat and Walker 1994; Nash and Sofer 1996].

Based on our discussion, Algorithm 5 is able to solve the three sub-problems because they are all in the form of (29). However, without taking their problem structures into account, the implementation may be inefficient. In particular, (27) and (28) significantly differ from traditional linear classification problems because they are matrix-variable problems. In Section 3.3, we will discuss some techniques for an efficient implementation of the truncated Newton method.

3.3. An Efficient Implementation of Algorithm 5 for Solving Sub-problems (27)-(28)

Among the three sub-problems, (26) is very close to standard linear classification problem, so the implementation is similar to past works. However, (27) and (28) are more different because their variables are matrices. Here, we only discuss details of solving (27) because the situation for (28) is similar.

To begin, we check the function evaluation, in which the main task is on calculating $\tilde{y}_i, \forall i$. From (30) and (31),

$$\begin{aligned}\tilde{y}_i &= \text{vec}(U)^T\tilde{\boldsymbol{x}}_i + \tilde{c}_i\\ &= \frac{1}{2}\,\text{vec}(U)^T(\boldsymbol{x}_i \otimes \boldsymbol{q}_i) + \tilde{c}_i\\ &= \frac{1}{2}\boldsymbol{q}_i^T U\boldsymbol{x}_i + \tilde{c}_i,\ i = 1, \ldots, l,\end{aligned}$$

or equivalently

$$\begin{aligned}\tilde{\boldsymbol{y}} = \begin{bmatrix}\tilde{y}_1\\ \vdots\\ \tilde{y}_l\end{bmatrix} &= \tilde{X}\,\text{vec}(U) + \tilde{\boldsymbol{c}}\\ &= \frac{1}{2}\left(Q^T \,.\!*\left(XU^T\right)\right)\mathbf{1}_{d\times 1} + \tilde{\boldsymbol{c}},\end{aligned} \tag{45}$$

where $Q = [\boldsymbol{q}_1, \ldots, \boldsymbol{q}_l] \in \mathcal{R}^{d\times l}$, "$.*$" stands for the element-wise product of two matrices, and $\mathbf{1}_{d\times 1}$ is the vector of ones. For (45), we can calculate and store XU^T as a dense matrix regardless of whether X is sparse or not; the cost is $\mathcal{O}\left(d \times (\#\ \text{nnz})\right)$. Similar to (45), $\tilde{\boldsymbol{\Delta}}$ used in line search can be computed via

$$\tilde{\Delta}_i = \frac{1}{2}\boldsymbol{q}_i^T S\boldsymbol{x}_i,\ \forall i \text{ or equivalently } \tilde{\boldsymbol{\Delta}} = \tilde{X}\tilde{\boldsymbol{s}} = \frac{1}{2}\left(Q^T \,.\!*\left(XS^T\right)\right)\mathbf{1}_{d\times 1}, \tag{46}$$

where $\tilde{\boldsymbol{s}} = \text{vec}(S)$ with $S \in \mathcal{R}^{d\times n}$ is the search direction.

Then, we investigate how to compute the gradient of (27). By some reformulations, we have

$$\begin{aligned}\nabla \tilde{f}(\mathrm{vec}(U)) &= \lambda' \,\mathrm{vec}(U) + \tilde{X}^T \boldsymbol{b} \\ &= \lambda' \,\mathrm{vec}(U) + \frac{1}{2}\sum_{i=1}^{l}(b_i \boldsymbol{x}_i \otimes \boldsymbol{q}_i) \\ &= \lambda' \,\mathrm{vec}(U) + \frac{1}{2}\sum_{i=1}^{l}\mathrm{vec}(b_i \boldsymbol{q}_i \boldsymbol{x}_i^T) \\ &= \lambda' \,\mathrm{vec}(U) + \frac{1}{2}\mathrm{vec}\left(\sum_{i=1}^{l}(\boldsymbol{q}_i b_i \boldsymbol{x}_i^T)\right) \\ &= \lambda' \,\mathrm{vec}(U) + \frac{1}{2}\mathrm{vec}(Q\,\mathrm{diag}(\boldsymbol{b})X),\end{aligned} \tag{47}$$

where $\mathrm{diag}(\boldsymbol{b})$ is a diagonal matrix in which the ith diagonal element is b_i. Depending on the size of Q and X, or whether X is sparse, we can decide to calculate $Q\,\mathrm{diag}(\boldsymbol{b})$ or $\mathrm{diag}(\boldsymbol{b})X$ first. Then, the main operation for gradient evaluation is a matrix-matrix product that costs $\mathcal{O}(d \times (\#\ \mathrm{nnz}))$.

Next we discuss Hessian-vector products in CG. From (31), (37), and $\tilde{\boldsymbol{s}} = \mathrm{vec}(S)$,

$$\tilde{H}\tilde{\boldsymbol{s}} = \lambda' \,\mathrm{vec}(S) + \tilde{X}^T(D(\tilde{X}\,\mathrm{vec}(S))) \tag{48}$$

$$\begin{aligned} &= \lambda' \,\mathrm{vec}(S) + \tilde{X}^T(D\boldsymbol{z}) \\ &= \lambda' \,\mathrm{vec}(S) + \sum_{i=1}^{l}(D\boldsymbol{z})_i \tilde{\boldsymbol{x}}_i \\ &= \lambda' \,\mathrm{vec}(S) + \sum_{i=1}^{l}(D\boldsymbol{z})_i(\boldsymbol{x}_i \otimes \boldsymbol{q}_i) \end{aligned}$$

$$= \lambda' \,\mathrm{vec}(S) + \frac{1}{2}\sum_{i=1}^{l}\mathrm{vec}(\boldsymbol{q}_i(D\boldsymbol{z})_i\boldsymbol{x}_i^T) \tag{49}$$

$$= \lambda' \,\mathrm{vec}(S) + \frac{1}{2}\mathrm{vec}(Q\,\mathrm{diag}(D\boldsymbol{z})X), \tag{50}$$

where

$$z_i = \tilde{\boldsymbol{x}}_i^T\tilde{\boldsymbol{s}} = \frac{1}{2}(\boldsymbol{x}_i^T \otimes \boldsymbol{q}_i^T)\,\mathrm{vec}(S) = \frac{1}{2}\boldsymbol{q}_i^T S \boldsymbol{x}_i,\ i = 1,\ldots,l \tag{51}$$

or equivalently

$$\boldsymbol{z} = \tilde{X}\,\mathrm{vec}(S) = \frac{1}{2}\left(Q^T .* \left(XS^T\right)\right)\mathbf{1}_{d\times 1} \tag{52}$$

by following the same calculation in (46). Note that $D\boldsymbol{z}$ is the product between a diagonal matrix D and a vector $\boldsymbol{z}$. We investigate the complexity of the Hessian-vector product. The first term in (48) is a standard vector scaling that costs $\mathcal{O}(nd)$. For the second term in (48), we separately consider two major steps:

1. The computation of $\boldsymbol{z} = \tilde{X}\,\mathrm{vec}(S)$.
2. The product between $\tilde{X}^T$ and a dense vector $D\boldsymbol{z}$.

In (52), we have seen that the first step can be done by the same way to compute (46). For the second step, we can refer to the calculation of $\tilde{X}^T\boldsymbol{b}$ in (47). Therefore, following earlier discussion, the cost of one Hessian-vector product is

$$\mathcal{O}\left(d \times (\text{\# nnz})\right). \tag{53}$$

A disadvantage of the above setting is that the data matrix X is accessed twice in the two different steps. From (49) and (51), the Hessian-vector product can be represented as

$$\lambda' \operatorname{vec}(S) + \frac{1}{4}\sum_{i=1}^{l} \operatorname{vec}\left(\boldsymbol{q}_i D_{ii} (\boldsymbol{q}_i^T S \boldsymbol{x}_i) \boldsymbol{x}_i^T\right). \tag{54}$$

From this instance-wise representation, the second term can be computed by a single loop over $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l$. While $\boldsymbol{x}_i$ is still used twice: $S\boldsymbol{x}_i$ and $\boldsymbol{q}_i z_i \boldsymbol{x}_i^T$, between them no other $\boldsymbol{x}_j$, $j \neq i$ are accessed and therefore $\boldsymbol{x}_i$ tends to remain at a higher layer of the memory hierarchy. Such a setting of getting better data locality has been considered for training LR [Zhuang et al. 2015]. However, because of the instance-wise setting, we can not benefit from optimized dense matrix operations (e.g., optimized BLAS) when computing, for example, SX^T in (52). Therefore, the decision of using (54) or not may hinge on whether X is a sparse matrix.

In line search, besides the calculation of $\tilde{\boldsymbol{\Delta}}$ in (46), from (41) we also need

$$\langle U, S\rangle,\ \|S\|_F^2, \text{ and } \langle G, S\rangle, \tag{55}$$

where G is the matrix form of (27)'s gradient and $\langle \cdot, \cdot \rangle$ is the inner product of two matrices.

Algorithm 6 summarizes the details in our Newton procedure for solving (27). Matrix representations are used, so we never have to reshape S to $\operatorname{vec}(S)$ or U to $\operatorname{vec}(U)$. The latest U in ANT is used as the initial point, so we can continue to improve upon the current solution. In Algorithm 6, the computational complexity per Newton iteration is

$$(\text{\# of CG iterations} + 2) \times \mathcal{O}\left(d \times (\text{\# nnz})\right) + (\text{\# of line search iterations}) \times \mathcal{O}\left(l\right), \tag{56}$$

in which the term $2 \times \mathcal{O}\left(d \times (\text{\# nnz})\right)$ is from the costs of (46) and (47) for function and gradient evaluation. Note that the $\mathcal{O}\left(nd\right)$ cost for the quantities in (55) and other places is not listed here because it is no more than $\mathcal{O}\left(d \times (\text{\# nnz})\right)$. Let

$$\bar{n} = \frac{\text{\# nnz}}{l}$$

denote the average non-zero features per instance. By (44) and (56), there are around

$$\frac{(\text{\# of CG iteration} + 2) \times \mathcal{O}\left(d\bar{n}\right) + (\text{\# of line search iterations})}{(\text{\# of line search iterations} + 2)}$$

standard read, write, or arithmetic operations per loss-related function call. Because the number of line-search iterations is usually small, the ratio of loss-related operations to other standard operations in ANT is $\mathcal{O}\left(1\right)$. Thus the major drawback in CD is alleviated considerably.

3.4. Convergence Guarantee and Stopping Conditions

Algorithm 4 is under the framework of block coordinate descent method (BCD), which sequentially update one block of variables while keeping all remaining blocks unchanged. BCD is a generalization of CD, in which each block contains only a single variable. See [Bertsekas 1999] for a comprehensive introduction about BCD. It is known that if each sub-problem of a block has a unique optimal solution and is exactly solved, then the procedure converges to a stationary point [Bertsekas 1999, Proposition 2.7.1].

Algorithm 6 An implementation of Algorithm 5 for solving (27) by operations on matrix veriables without vectorizing them.

```
Given $0 < \tilde{\epsilon} < 1$ and the current $(\boldsymbol{w}, U, V)$.
$Q \leftarrow VX^T$
Compute and cache $\tilde{\boldsymbol{y}} = \frac{1}{2}\left(Q^T .* \left(XU^T\right)\right)\mathbf{1}_{d\times 1} + X\boldsymbol{w}$.
$f \leftarrow \frac{\lambda'}{2}\|U\|_F^2 + \sum_{i=1}^{l}\xi(\tilde{y}_i; y_i)$.
for $k \leftarrow \{0, 1, \dots\}$ do
    Calculate and store $\tilde{y}_i$ (and $e^{y_i\tilde{y}_i}$ if (3) is used) and then obtain $b_i$ by (12), $\forall i$.
    $G \leftarrow \lambda' U + \frac{1}{2}Q\,\mathrm{diag}(\boldsymbol{b})X$
    if $k = 0$ then
        $\|G^0\|_F \leftarrow \|G\|_F$
    end if
    if $\|G\|_F \le \tilde{\epsilon}\|G^0\|_F$ then
        Output $U$ as the solution of (27).
        break
    end if
    Compute $D_{ii}$ via (19), $\forall i$.
    Solve (36) approximately via CG to get an update direction $S$.
    Parpare variables listed in (55) and $\tilde{\boldsymbol{\Delta}} = \frac{1}{2}\left(Q^T .* \left(XS^T\right)\right)\mathbf{1}_{d\times 1}$
    for $\theta \leftarrow \{1, \beta, \beta^2, \dots\}$ do
        $\delta \leftarrow \frac{\lambda'}{2}\left(2\theta\langle U, S\rangle + \theta^2\|S\|_F^2\right) + \sum_{i=1}^{l}\xi(\tilde{y}_i + \theta\tilde{\Delta}_i; y_i) - \sum_{i=1}^{l}\xi(\tilde{y}_i; y_i)$
        if $\delta \le \theta\nu\langle G, S\rangle$ then
            $U \leftarrow U + \theta S$
            $f \leftarrow f + \delta$
            $\tilde{\boldsymbol{y}} \leftarrow \tilde{\boldsymbol{y}} + \theta\tilde{\boldsymbol{\Delta}}$
            break
        end if
    end for
end for
```

Each of our sub-problems (26)-(28) possesses a unique optimal solution because of the strongly convex regularization term. Therefore, if we exactly solve every sub-problem, then our method is guaranteed to converge to a stationary point of (6).

In practice, any optimization method needs a stopping condition. Here, we consider a relative setting

$$\|\nabla F(\boldsymbol{w}, U, V)\| \le \epsilon\|\nabla F(\boldsymbol{w}_{\text{init}}, U_{\text{init}}, V_{\text{init}})\|, \tag{57}$$

where $0 < \epsilon < 1$ is a small stopping tolerance and $(\boldsymbol{w}_{\text{init}}, U_{\text{init}}, V_{\text{init}})$ indicates the initial point of the model. The use of this type of criteria includes, for example, [Lin 2007] for non-negative matrix factorization.

Each sub-problem in the alternating minimization procedure requires a stopping condition as well. A careful design is needed as otherwise the global stopping condition in (57) may never be reached. In Section 3.2, we consider a relative stopping condition,

$$\|\nabla\tilde{f}\left(\tilde{\boldsymbol{w}}\right)\| \le \tilde{\epsilon}\|\nabla\tilde{f}\left(\tilde{\boldsymbol{w}}^0\right)\|, \tag{58}$$

where $\tilde{\boldsymbol{w}}^0$ is the initial point in solving the sub-problem. Take the sub-problem of U as an example. The gradient norm of (6) with respect to U is identical to the gradient norm of U's sub-problem:

$$\|\nabla_U f(\boldsymbol{w}, U, V)\|_F = \|\nabla\tilde{f}(\mathrm{vec}(U))\|.$$

If $\|\nabla_U f(\boldsymbol{w}, U, V)\|_F > 0$, Algorithm 6 conducts at least one Newton step and the function value of $F(\boldsymbol{w}, U, V)$ must be decreased. The reason is that (58) is violated if we do not change U. In other words, any block with a non-vanished gradient would be adjusted. Therefore, unless

$$\|\nabla_{\boldsymbol{w}} f(\boldsymbol{w}, U, V)\| = \|\nabla_U f(\boldsymbol{w}, U, V)\|_F = \|\nabla_V f(\boldsymbol{w}, U, V)\|_F = 0,$$

one of the three blocks must be updated. That is, if the condition in (57) is not satisfied yet, our alternating minimization procedure must continue to update $(\boldsymbol{w}, U, V)$ rather than stay at the same point.

4. TECHNIQUES TO ACCELERATE TRUNCATED NEWTON METHODS FOR SOLVING THE SUB-PROBLEMS

From (56), the computational bottleneck in our truncated Newton method is the CG procedure, so we consider two popular techniques, preconditioning and sub-sampled Hessian, for its acceleration.

4.1. Preconditioned Conjugate Gradient Methods

The first technique aims to reduce the number of CG iterations for solving (36) by considering an equivalent but better conditioned linear system. Precisely, we consider a preconditioner $\tilde{M}$ to approximately factorize $\tilde{H}$ such that $\tilde{H} \approx \tilde{M}\tilde{M}^T$, and then use CG to solve

$$\hat{H}\hat{\boldsymbol{s}} = \hat{\boldsymbol{g}}, \tag{59}$$

where $\hat{H} = \tilde{M}^{-1}\tilde{H}\tilde{M}^{-T}$ and $\hat{\boldsymbol{g}} = \tilde{M}^{-1}\tilde{\boldsymbol{g}}$. Once $\hat{\boldsymbol{s}}$ is found, the solution of (36) can be recovered by $\tilde{\boldsymbol{s}} = \tilde{M}^{-T}\hat{\boldsymbol{s}}$. If $\tilde{H}$ is perfectly factorized (i.e., $\tilde{H} = \tilde{M}\tilde{M}^T$), (59) can be solved in no time because $\hat{H}$ is an identity matrix.

Many preconditioners have been proposed, e.g., diagonal preconditioner, incomplete Cholesky factorization, and polynomial preconditioners [Golub and Van Loan 2012]. However, finding a suitable preconditioner is not easy because first each CG iteration becomes more expensive and second the decrease of the number of CG iterations is not theoretically guaranteed. For a CG-based truncated Newton method for LR in [Lin et al. 2008], the use of preconditioned conjugate gradient methods (PCG) has been studied. They point out that the implicit use of $\tilde{H}$ further increases the difficulty to implement a preconditioner and finally choose the simple diagonal preconditioner for the following two reasons. First, the diagonal elements of $\tilde{H}$ can be constructed cheaply. Second, the extra computation introduced by the diagonal preconditioner in each CG iteration is relatively small. Experiments in [Lin et al. 2008] show that for LR, using diagonal preconditioners may not be always effective in decreasing the total number of CG iterations and the running time. Nevertheless, we think it is worth trying preconditioned CG here because of some differences between their settings and ours. In the alternating minimization procedure, we solve a sequence of optimization sub-problems of (6) rather than a single linear classification problem in [Lin et al. 2008]. In the early stage of the ANT, we only loosely solve the sub-problems and the effect of PCG may vary depending on the strictness of the stopping condition.

To see the details of using PCG in ANT, we consider the following diagonal preconditioner for solving (36) as an example.

$$\tilde{M} = \tilde{M}^T = \operatorname{diag}(\tilde{\boldsymbol{h}}), \tag{60}$$

where

$$\tilde{\boldsymbol{h}} = \sqrt{\tilde{\lambda}\mathbf{1}_{\tilde{n}\times 1} + \sum_{i=1}^{l} D_{ii}\left(\tilde{\boldsymbol{x}}_i \,.\!* \tilde{\boldsymbol{x}}_i\right)}.$$

Note that "$\sqrt{\cdot}$" element-wisely performs the square-root operation if the input argument is a vector or a matrix. With $\tilde{M} = \tilde{M}^T$, the Hessian-vector product in CG to solve the diagonally-preconditiond linear system is

$$\hat{H}\hat{\boldsymbol{s}} = \tilde{M}^{-1}(\tilde{H}(\tilde{M}^{-T}\hat{\boldsymbol{s}})) = \tilde{M}^{-1}(\tilde{H}(\tilde{M}^{-1}\hat{\boldsymbol{s}})). \tag{61}$$

To illustrate the computational details of PCG, we consider (27)'s Newton linear system as an example. First, we discuss the construction of the preconditioner. Consider the sub-problem (27) as an example. From

$$\tilde{\boldsymbol{x}}_i = \frac{1}{2}(\boldsymbol{x}_i \otimes \boldsymbol{q}_i) = \frac{1}{2}\operatorname{vec}(\boldsymbol{q}_i\boldsymbol{x}_i^T),$$

we have

$$\begin{aligned}
&\sum_{i=1}^{l} D_{ii}\left(\tilde{\boldsymbol{x}}_i \,.\!* \tilde{\boldsymbol{x}}_i\right)\\
=&\frac{1}{4}\sum_{i=1}^{l} D_{ii}\operatorname{vec}\left(\left(\boldsymbol{q}_i \,.\!* \boldsymbol{q}_i\right)\left(\boldsymbol{x}_i \,.\!* \boldsymbol{x}_i\right)^T\right)\\
=&\frac{1}{4}\sum_{i=1}^{l}\operatorname{vec}\left(\left(\boldsymbol{q}_i \,.\!* \boldsymbol{q}_i\right) D_{ii}\left(\boldsymbol{x}_i \,.\!* \boldsymbol{x}_i\right)^T\right)\\
=&\frac{1}{4}\operatorname{vec}\left(\left(Q \,.\!* Q\right) D\left(X \,.\!* X\right)\right).
\end{aligned} \tag{62}$$

Thus, the preconditioner of U's sub-problem can be obtained via

$$\tilde{M} = \tilde{M}^T = \operatorname{diag}\left(\sqrt{\lambda'\mathbf{1}_{nd\times 1} + \frac{1}{4}\operatorname{vec}\left(\left(Q \,.\!* Q\right) D\left(X \,.\!* X\right)\right)}\right)$$

or without vectorization

$$M = \sqrt{\lambda'\begin{bmatrix}1 & \dots & 1\\ \vdots & \ddots & \vdots\\ 1 & \dots & 1\end{bmatrix} + \frac{1}{4}\left(Q \,.\!* Q\right) D\left(X \,.\!* X\right)} \in \mathcal{R}^{d\times n}. \tag{63}$$

We point out that the cost to obtain (63) is $\mathcal{O}\left(d \times (\#\ \text{nnz})\right)$.

Next, we look into the computation of (61). Let $\hat{S} \in \mathcal{R}^{d\times n}$ be the matrix form of the vector $\hat{\boldsymbol{s}}$ such that $\hat{\boldsymbol{s}} = \operatorname{vec}(\hat{S})$. It is possible to present (61) in terms of $\hat{S}$ to avoid vectorizations. First, the matrix form of $\tilde{M}^{-1}\hat{\boldsymbol{s}}$ is $\hat{S} \,./M$, where "$./$" denotes the element-wise division. By substituting $S = \hat{S} \,./M$ into (50) and then element-wisely dividing the resulting matrix by M again, (61) can be written as

$$\left(\lambda'\left(\hat{S} \,./\, M\right) + \frac{1}{2}Q\operatorname{diag}\left(D\hat{\boldsymbol{z}}\right)X\right)./M, \tag{64}$$

Algorithm 7 A preconditioned conjugate gradient method for solving (36) by operations on matrix variables. Linear systems in the sub-problem of U are considered.

```
Given $0 < \eta < 1$ and $G$, the gradient matrix of the sub-problem (i.e., the matrix form of (47)). Let $\hat{S} = \mathbf{0}_{d\times n}$.
Compute $M$ via (63).
Calculate $R = -G \,./M$, $\hat{D} = R$, and $\gamma^0 = \gamma = \|R\|_F^2$.
while $\sqrt{\gamma} > \eta\sqrt{\gamma^0}$ do
    $\hat{D}_h \leftarrow \hat{D} \,./M$
    $\hat{\boldsymbol{z}} \leftarrow \left(Q^T .* \left(X\hat{D}_h^T\right)\right)\mathbf{1}_{d\times 1}$
    $\hat{D}_h \leftarrow \left(\lambda'\hat{D}_h + Q\,\mathrm{diag}\left(D\hat{\boldsymbol{z}}\right)X\right) ./M$
    $\alpha \leftarrow \gamma / \left\langle \hat{D}, \hat{D}_h \right\rangle$
    $\hat{S} \leftarrow \hat{S} + \alpha\hat{D}$
    $R \leftarrow R - \alpha\hat{D}_h$
    $\gamma^{\text{new}} \leftarrow \|R\|_F^2$
    $\beta \leftarrow \gamma^{\text{new}}/\gamma$
    $\hat{D} \leftarrow R + \beta\hat{D}$
    $\gamma \leftarrow \gamma^{\text{new}}$
end while
Output $S = \hat{S} \,./M$ as the solution.
```

where

$$\hat{\boldsymbol{z}} = \frac{1}{2}\left(Q^T .* \left(X\left(\hat{S} \,./\, M\right)^T\right)\right)\mathbf{1}_{d\times 1}.$$

For the computational complexity, besides $\mathcal{O}(d \times (\#\ \text{nnz}))$ from what we discussed in (50), the newly introduced divisions associated with M require $\mathcal{O}(nd)$ operations. Unless the data matrix is very sparse, from $n \leq \#$ nnz, each PCG iteration does not cost significantly more than that without preconditioning. Note that we only compute M once in the beginning of PCG and then reuse it in each Hessian-vector product. See Algorithm 7 for a PCG procedure of using matrix variables and operations for solving the Newton system of U's sub-problem.

4.2. Sub-sampled Hessian Matrix

Some recent works have demonstrated that we can use sub-sampled Hessian to accelerate the Hessian-vector product in truncated Newton methods for empirical risk minimization [Byrd et al. 2011; Byrd et al. 2012; Wang et al. 2015]. From a statistical perspective, the training instances are drawn from an underlying distribution $\Pr(y, \tilde{\boldsymbol{x}})$. Then, (34) can be interpreted as an empirical estimation of

$$\tilde{\lambda}I + lE\left[\xi''(\tilde{y}; y)\tilde{\boldsymbol{x}}\tilde{\boldsymbol{x}}^T\right].$$

If L is a set of $|L|$ instances randomly drawn from $\{1, \ldots, l\}$, then the sub-sampled Hessian matrix,

$$\begin{aligned} \tilde{H}_{\text{sub}} &= \tilde{\lambda}I + \frac{l}{|L|}\sum_{i\in L} D_{ii}\tilde{\boldsymbol{x}}_i\tilde{\boldsymbol{x}}_i^T, \\ &= \tilde{\lambda}I + \frac{l}{|L|}\tilde{X}_{L,:}^T D_{L,L}\tilde{X}_{L,:}, \end{aligned} \tag{65}$$

is an unbiased estimation of the original Hessian matrix. Note that the rows of $\tilde{X}_{L,:} \in \mathcal{R}^{|L|\times\tilde{n}}$ are the sampled feature vectors and $D_{L,L} \in \mathcal{R}^{|L|\times|L|}$ is the corresponding sub-matrix of D. After replacing $\tilde{H}$ with $\tilde{H}_{\text{sub}}$, the linear system solved by CG becomes

$$\tilde{H}_{\text{sub}}\tilde{\boldsymbol{s}} = -\tilde{\boldsymbol{g}}. \tag{66}$$

Consider the sub-problem (27) as an example. From (50) and (54), if a subset of the rows in the data matrix X can be easily extracted, then the Hessian-vector product (in matrix form) for solving (66) can be conducted by

$$\lambda' S + \frac{l}{2|L|} Q_{:,L} \operatorname{diag}(D_{L,L}\boldsymbol{z}_L) X_{L,:},$$

where $Q_{:,L} \in \mathcal{R}^{d\times|L|}$ is a sub-matrix of Q containing $\boldsymbol{q}_i$, $i \in L$ and

$$\boldsymbol{z}_L = \frac{1}{2}\left(Q_{:,L}^T .* \left(X_{L,:}S^T\right)\right)\mathbf{1}_{d\times1}. \tag{67}$$

Therefore, a row-wise format should be used in storing X. In comparison with using all training instances, the cost of each CG iteration can be reduced to

$$\mathcal{O}\left(\frac{|L|d}{l} \times (\text{\# nnz})\right),$$

when, for example, the sub-problem of U is considered. Although a small L can largely speed up the Hessian-vector product, the information loss caused by dropping instances can have a negative impact on the update direction found by solving (66). In the extreme case that L is an empty set, our truncated Newton method is reduced to a gradient descent method, which needs much more iterations. It may not be easy to decide the size of the subset L, but in Section 6.4 we will examine the running time under different choices.

5. PARALLELIZATION OF TRUNCATED NEWTON METHODS FOR SOLVING THE SUB-PROBLEMS

When solving each sub-problem, we should parallelize the main computational bottleneck: the calculations of the objective function, the gradient, and the Hessian-vector product. Because we have shown in Section 3.1 that each sub-problem is equivalent to a linear classification problem, our approach follows past developments of parallel Newton methods for regularized empirical risk minimization (e.g., [Zhuang et al. 2015] for distributed environments and [Lee et al. 2015] for multi-core environments). Here we focus on an implementation for multi-core machines by assuming that X can be instance-wisely accessed by multiple threads. To illustrate the details, we consider the sub-problem of U as an example. For easy description, we assume that all $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l$ are used without the sub-sampling technique in Section 4.2. If there are s threads available, we divide $\{1, \ldots, l\}$ into s disjoint subsets, $L_1, \ldots, L_s$, for our task distribution. Recall the sub-matrix notations of X, Q, D, and $\boldsymbol{z}$ in Section 4.2. From (45) and (47), the parallelized forms of the output values and the gradient respectively are

$$\tilde{\boldsymbol{y}} = \begin{bmatrix} \frac{1}{2}\left(Q_{:,L_1}^T .* \left(X_{L_1,:}U^T\right)\right)\mathbf{1}_{d\times1} + X_{L_1,:}\boldsymbol{w} \\ \vdots \\ \frac{1}{2}\left(Q_{:,L_s}^T .* \left(X_{L_s,:}U^T\right)\right)\mathbf{1}_{d\times1} + X_{L_s,:}\boldsymbol{w} \end{bmatrix} \tag{68}$$

$$\nabla\tilde{f}(U) = \lambda' U + \frac{1}{2}\sum_{r=1}^{s}(Q_{:,L_r}\operatorname{diag}(\boldsymbol{b}_{L_r})X_{L_r,:}), \tag{69}$$

where tasks indexed by L_r is assigned to the rth thread. Form (50), parallel Hessian-vector product can be done by considering

$$\lambda' S + \frac{1}{4}\sum_{r=1}^{s} \left(Q_{:,L_r} \operatorname{diag}(D_{L_r}\boldsymbol{z}_{L_r}) X_{L_r,:}\right). \tag{70}$$

For PCG, the diagonal preconditioner can be obtained via

$$M = \sqrt{\lambda' \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix} + \frac{1}{4}\sum_{r=1}^{s} \left(Q_{:,L_r} .* Q_{:,L_r}\right) D_{L_r} \left(X_{L_r,:} .* X_{L_r,:}\right)} \in \mathcal{R}^{d\times n}. \tag{71}$$

Besides (68), for completing any of (69), (70), and (71), we must collect the results from the used threads by a series of d-by-n matrix additions (called a dn-dimensional reduction) after the parallelized sections are finished. For example, if two threads are used together to compute (70), we can compute

$$B_1 = Q_{:,L_1} \operatorname{diag}(D_{L_1}\boldsymbol{z}_{L_1}) X_{L_1,:} \text{ and } B_2 = Q_{:,L_2} \operatorname{diag}(D_{L_2}\boldsymbol{z}_{L_2}) X_{L_2,:},$$

in parallel, and sum up the two matrices to obtain the second term of (70). Notice that from (51) and (70), the rth worker only maintains $Q_{:,L_r}$ and $\boldsymbol{z}_{:,L_r}$ instead of the whole Q and $\boldsymbol{z}$.

The parallelization of solving V's sub-problem is very similar.

6. EXPERIMENTS

In some previous studies on linear regression and classification, coordinate descent methods beat Newton-type methods when the loss-related evaluations are not expensive [Chang et al. 2008; Ho and Lin 2012]. Because our sub-problems are similar to logistic regression problems, ANT may not be the best strategy if losses like (4) are used. Thus, we concentrate on the logistic loss in the subsequent discussion and recommend using CD (or SG) otherwise.

To examine the effectiveness of ANT, we conduct a series of experiments on some real-world data sets. Section 6.1 gives the data statistics and discusses our experimental settings including the parameter selection. The environment for our experiments is described in Section 6.2. Sections 6.3-6.4 demonstrate the usefulness of the two techniques discussed in Section 4. The running time comparison on ANT against ADAGRAD and CD is presented in Section 6.5.

6.1. Data Sets and Parameters

We select several real-world data sets listed in Table II. All data sets can be downloaded at LIBSVM data sets page.[2] For `webspam` we consider the uni-gram version. For `kddb`, we choose the raw version of "bridge to algebra" track in KDD-Cup 2010 following [Juan et al. 2016]. To select proper parameters and calculate a test score, we need training, validation, and test sets for each problem. If a test set is available from the source, we directly use it.[3] Otherwise, we randomly select 20% instances from the whole data set for testing. Then, the training and validation sets are created by a 80-20 split of instances not used for testing. Note that we swap `rcv1`'s training and test sets on LIBSVM page to make the training set bigger.

---

[2] LIBSVM data sets are available at www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets.

[3] The labels in the test sets of `avazu-app`, `avazu-site`, and `criteo` are not meaningful, so they are treated as if they do not have test sets.

Table II: Data statistics and parameters used in experiments.

Data set	l	n	nnz	λ	λ'	λ''
`a9a`	26,049	122	451,592	64	1	1
`webspam`	280,000	246	23,817,071	0.25	0.0625	0.0625
`kddb`	9,464,836	651,166	173,376,873	1	16	16
`news20`	16,009	1,355,191	7,303,887	0.0625	0.0625	0.0625
`rcv1`	677,399	47,236	39,627,321	0.0625	1	1
`url`	1,916,904	3,231,961	2,216,68,617	16	1	1
`avazu-app`	14,596,137	1,000,000	189,632,716	4	4	4
`avazu-site`	25,832,830	1,000,000	353,517,436	1	4	4
`criteo`	45,840,617	1,000,000	1,552,191,209	0.25	16	16

In problem (6), four parameters must be tuned. To avoid the expensive cost of searching the best setting, we fix $d = 20$ and $\lambda' = \lambda''$ throughout our experiments. For the regularization coefficients of each data set, we consider $\{0.0625, 0.25, 1, 4, 16, 64\}$ as the search range of each regularization parameter and check all the combinations. Values leading to the lowest logistic loss on the validation set are used. Table II gives the parameters identified by the process. Furthermore, we set $\boldsymbol{w}_{\text{init}} = \boldsymbol{0}$ and randomly choose every element of U_{init} and V_{init} uniformly from $[-1/\sqrt{d}, 1/\sqrt{d}]$. For any comparison between different settings or algorithms, we ensure that they start with the same initial point. For the stopping condition of PCG, we set $\eta = 0.3$ so Algorithm 7 terminates once

$$\|\hat{H}\hat{\boldsymbol{s}} + \hat{\boldsymbol{g}}\| \leq \eta\|\hat{\boldsymbol{g}}\|.$$

Moreover, the two parameters of line search are $\beta = 0.5$ and $\nu = 0.01$.

6.2. Environment and Implementation

The experiments in Sections 6.3-6.4 are conducted on a Linux machine with one Intel Core i7-6700 CPU 3.40GHz and 32 GB memory. For the experiments on multi-core implementations in Section 6.5, we migrate to another Linux machine with two Intel Xeon E5-2620 2.0GHz processors (12 physical cores in total) and 128 GB memory.

All the algorithms are implemented in C++ for a fair comparison. For the types of the variables, we use double-precision floating point for all real numbers and unsigned long integer for all indexes. We use `parallel-for` in `OpenMP` to implement all parallel computations discussed in Section 5. If the result generated by the threads is a scalar, the built-in scalar reduction can be used to automatically aggregate the results from different threads. For vector results, we manually merge them by sequential vector additions. This strategy has been confirmed to be effective in the parallelization of a truncated Newton method in [Lee et al. 2015]. For the parallelization of ADAGRAD, our implementation follows [Niu et al. 2011] to implement the for-loop in Algorithm 1 via `parallel-for` without the synchronization between them. The parallelization of CD is more difficult and beyond the scope of this work. Thus, we only implement a single-thread version.

6.3. Stopping Tolerance of Sub-problems and the Effect of Preconditioning

To see the effect of different stopping tolerances in Algorithm 5, we consider $\tilde{\epsilon} = 0.01$, 0.1, and 0.8, and examine the convergence speed of ANT. We also provide the results of ANT with PCG (denoted as ANT-P) to check the effectiveness of preconditioning. We present in Figure 1 the relation of running time to log-scaled distance between the current solution and a local optimal function value.

$$\frac{F(\boldsymbol{w}, U, V) - F^*}{F^*}, \tag{72}$$

where F^* is the lowest function value reached among all settings.

We begin with analyzing the results without preconditioning. From Figure 1, a larger inner stopping tolerance $\tilde{\epsilon}$ leads to shorter overall training time. This result indicates that in the early stage of the alternating minimization procedure there is no need to waste time for accurately solving the sub-problems. For example, if neither V nor $\boldsymbol{w}$ is close to the optimum, in the sub-problem of U, getting an accurate solution is not very useful. Some earlier alternating minimization procedures have had similar observations and made the inner stopping condition from a loose one in the beginning to a tight one in the end; see, for example, [Lin 2007, Section 6]. Our relative stopping condition in (58) automatically achieves that; although $\tilde{\epsilon}$ is fixed in the entire procedure, (58) is a strict condition in the final stage of the optimization process because of a small $\|\nabla \tilde{f}(\tilde{\boldsymbol{w}})\|$.

By comparing the ANT and ANT-P at different stopping conditions in Figure 1, we see that preconditioning is almost useless when the tolerance is tight (e.g., $\tilde{\epsilon} = 0.01$), but ANT-P converges faster than or at least comparable to ANT in the cases with a larger tolerance. To get more details, we investigate the convergence of the Newton method with/without preconditioning in solving sub-problems. We run ANT with $\tilde{\epsilon} = 0.8$ and extract the three convex sub-problems (26)-(28) at a specific iteration. Then for each sub-problem we run Newton methods with/without preconditioning (denoted as Newton-PCG and Newton-CG, respectively), and present in Figure 2 the relation between the function-value reduction (of the sub-problem) and the number of CG iterations used. We have the following observations. First, in the beginning of solving each sub-problem, Newton-PCG performs as good as or even better than Newton-CG. Second, as the number of CG iterations increases, the performance gap between Newton-CG and Newton-PCG gradually shrinks and preconditioning may even be harmful. Consequently, because we always terminate our truncated Newton method by a loose stopping of using $\tilde{\epsilon} = 0.8$, preconditioning is helpful or at least not harmful in ANT. A note that conducts more detailed analysis on the results in [Lin et al. 2008] has a similar conclusion.[4]

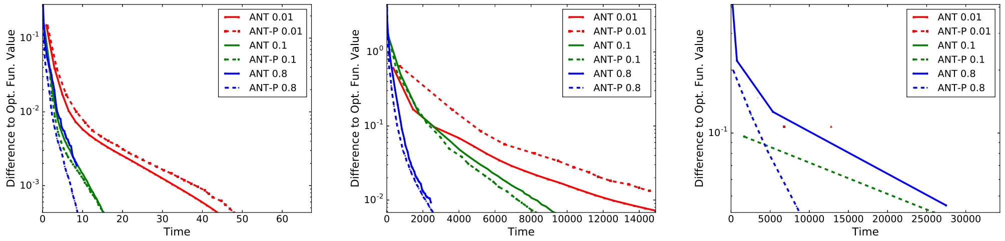
6.4. Effects of the Use of Sub-sampled Hessian Matrix

In Section 4, we have learned that sub-sampled Hessian can save some operations in Hessian-vector products. However, the overall convergence may not be faster because of less accurate directions. To have a good understanding, we compare the results using the full Hessian matrix, a 10% sub-sampled Hessian matrix, and a 1% sub-sampled one by using the best setting in Section 6.3 (i.e., ANT-P with $\tilde{\epsilon} = 0.8$). In Figure 3, for each sampling ratio, we show the change of objective values versus time. Comparing to using the full set, we observe that using 10% instances generally leads to similar or shorter training time. The difference is significant for data sets `a9a`, `kddb`, `avazu-app`, `avazu-site`, and `criteo`. These sets satisfy $l \gg n$, so some instances may carry redundant information and can be dropped. However, when only 1% instances are used, ANT often spends more time to achieve a specified objective value than that of using the full set.

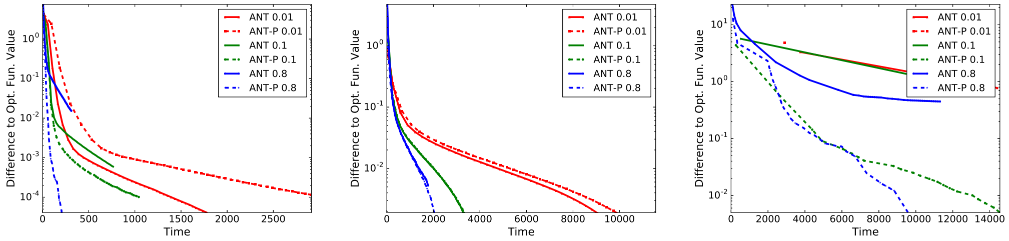
We conclude that by selecting a suitable amount of instances, the approach of using sub-sampled Hessian is useful for ANT. However, it is important to avoid choosing a too small subset.

---

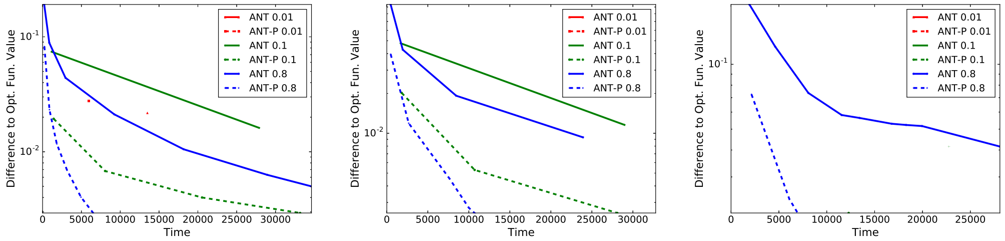
[4] www.csie.ntu.edu.tw/~cjlin/papers/logistic/pcgnote.pdf



(a) a9a (b) webspam (c) kddb



(d) news20 (e) rcv1 (f) url

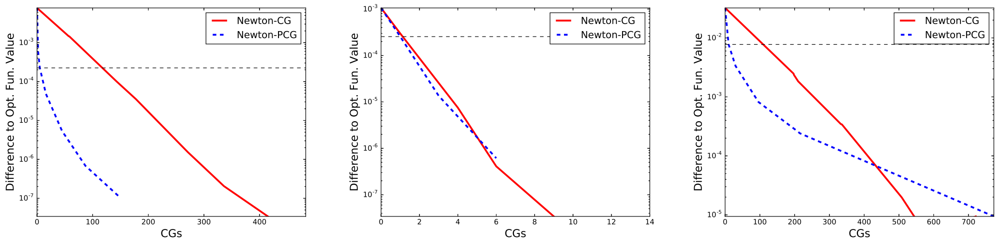


(g) avazu-app (h) avazu-site (i) criteo

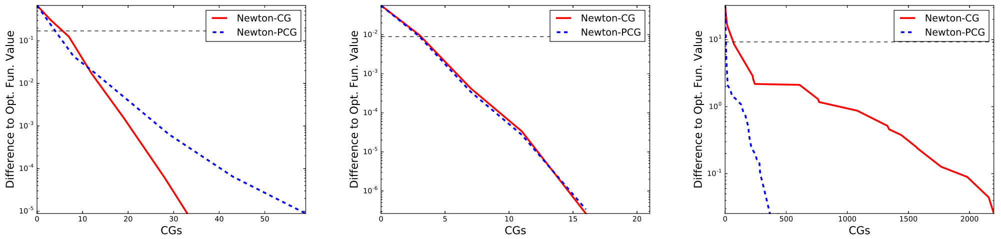
Fig. 1: Effects of the stopping condition and preconditioning for solving the sub-problems. The number after ANT(-P) indicates the used $\tilde{\epsilon}$. Time is in seconds. The y-axis is the log-scaled distance to a local optimal objective value; see (72). We terminate the training process if it does not produce any point within eight hours, so for larger data sets like `criteo`, some settings may have no line.

6.5. A Comparison on ANT and Other State of the Art Methods

We compare the running time of ANT-P against ADAGRAD and CD described in Section 2 under both single-thread and multi-thread settings. The parameters of ANT-P is determined by our discussion in Sections 6.3 and 6.4; we choose $\tilde{\epsilon} = 0.8$ as the stopping condition of solving sub-problems and use 10% of data for constructing the sub-Hessian matrix. For the parameter η_0 in the stochastic gradient method detailed in Algorithm 1, we use 0.1. For CD's number of inner iterations, we set $\bar{k}_{\text{inner}} = 1$. Following the format of Figure 1, we draw these methods' running performance at different numbers of threads in Figure 4. To provide a clearer view of Figure 4, we also explicitly list those solvers' running time required to reach a fixed objective value (largest of all approaches' final values) in Table IV. Note that the parallelization of CD is not easy, so we only implement a single-thread version. From Figure 4, we see that CD



(a) Solving (26) of `kddb`@5. (b) Solving (27) of `kddb`@5. (c) Solving (28) of `kddb`@5.



(d) Solving (26) of `url`@2. (e) Solving (27) of `url`@2. (f) Solving (28) of `url`@2.

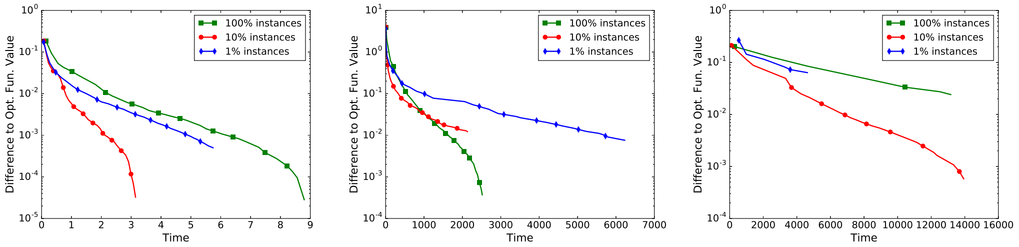
Fig. 2: Analysis of Newton methods with/without preconditioning in solving the three sub-problems in an ANT iteration indicated by the integer after @. The y-axis is the log-scaled distance to the global optimal function value of the sub-problem. The horizontal line indicates the first time that (58) with $\epsilon = 0.8$ is satisfied during the optimization process. That is, in ANT (or ANT-P) the sub-problem is approximately solved until reaching the horizontal line.

Table III: Portion of time spent on $\exp$ / $\log$ operations.

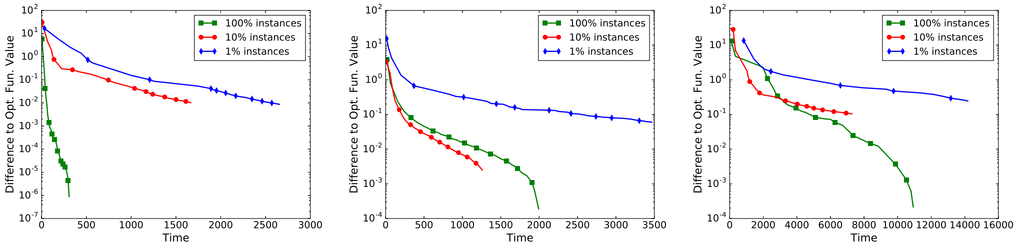
Data set / Solver	`a9a`	`news20`
ADAGRAD	0.64%	0.06%
CD	81.50%	65.05%
ANT	3.52%	0.07%

is largely outperformed by ADAGRAD and ANT-P. From Sections 2.1, 2.2, and 3.3, a possible reason is that CD costs much more expensive loss-related operations (i.e., $\exp$ / $\log$ evaluations) than its counterparts. To support our argument, Table III gives the time portion of executing $\exp$/$\log$ functions in the first ANT-P iteration, the first epoch of ADAGRAD, and the first outer iteration of CD on `a9a` and `news20`. Results on ANT-P and CD are consistent with those in an earlier study [Yuan et al. 2010], an observation explaining the slowness of CD. When comparing ANT-P and ADAGRAD, the time spent on $\exp$ and $\log$ can be ignored because most of the training time (typically more than 96%) are spent on standard arithmetic operations. Notice that in contrast to ANT and CD, ADAGRAD is not a descent method, so it may increase the objective value during the training procedure; for example, see the data set `url`'s curves in Figure 4.

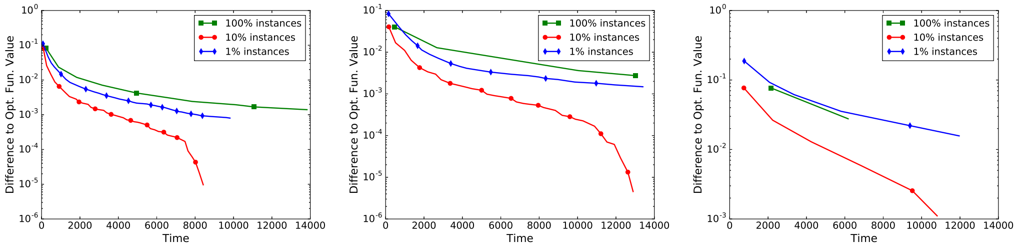
Furthermore, we observe in Figure 4 that the parallelization of ANT-P and ADAGRAD is effective. Their training time decreases as the number of threads increases.



(a) `a9a` (b) `webspam` (c) `kddb`



(d) `news20` (e) `rcv1` (f) `url`



(g) `avazu-app` (h) `avazu-site` (i) `criteo`

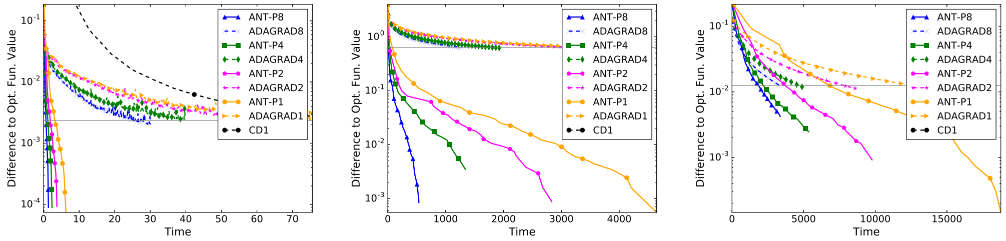
Fig. 3: Effects of ANT-P with different rates of using sub-sampled Hessian. Time is in seconds. The y-axis is the log-scaled distance to a local optimal objective value; see (72).

On the other hand, in either single- and multi-thread settings, ANT-P always produces significantly smaller objective values in comparison with ADAGRAD and CD. Two possible reasons are given. One is that ADAGRAD may eventually reach the same values but its convergence is very slow. The other one is that the FM problem is not convex, so ADAGRAD and ANT may reach different solutions.

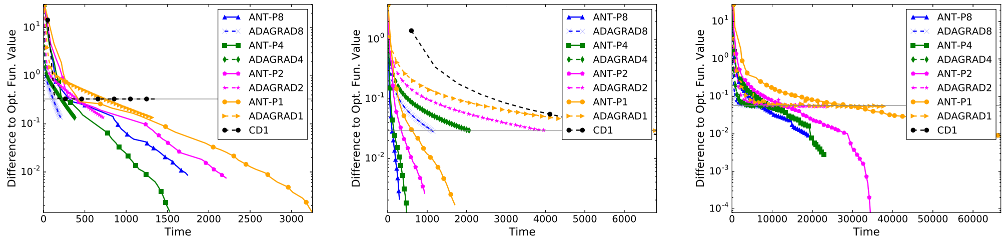
6.6. Comparison on Test Performance with Other Formulations

From a practical perspective, it is important to examine if the modified FM problem (6) can perform as well as the original FM problem (2). To this end, we consider two evaluation criteria, the accuracy and the log loss defined as

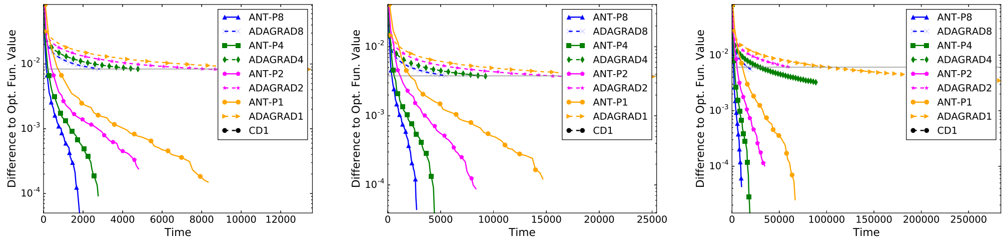
$$\text{Log loss} = \frac{1}{l}\sum_{i=1}^{l}\log(1+\exp(-y_i\hat{y}_i))$$



(a) `a9a` (b) `webspam` (c) `kddb`



(d) `news20` (e) `rcv1` (f) `url`



(g) `avazu-app` (h) `avazu-site` (i) `criteo`

Fig. 4: A comparison between ANT-P, CD, and ADAGRAD. The number in the legend indicates the number of threads. Time is in seconds. The y-axis is the log-scaled distance to a local optimal objective value; see (72). We terminate the training process if it does not produce any point within three hundred thousand seconds. Thus for some data sets there is no curve for CD. For each data set, the horizontal line indicates the lowest object value that can be reached by all our settings.

For each data set, we compare their test accuracies and log losses in Table V. To see how FM improves over a linear model, we include results of LR. For a fair comparison, we use $d = 40$ for (2) and $d = 20$ for (6) so that they have the same number of model parameters. For the solvers, we consider the truncated Newton method in Algorithm 5 for LR, the stochastic gradient method in Section 2.1 for (2) and (6), and CD and ANT-P for (6).

As shown in Table V, the performance gap between problems (2) and (6) is not significant, so the modified FM formulation in (6) seems to give similar learning capability. Regarding the comparison between FM and LR, FM gives a much higher test score than LR on `webspam`. This data set has less features than its instances, so feature conjunction seems to be useful. For other data sets, the performance gap is smaller. For

# of threads	1	2	4	8	1	2	4	8	1
Solver / Data set	ANT-P				ADAGRAD				CD
`a9a`	3.3	1.9	1.2	0.7	73.7	59.9	39.9	24.9	87.4
`webspam`	113.1	40.4	19.3	15.7	3581.6	3168.8	1836.3	1270.8	NA
`news20`	438.6	406.7	328.3	357.0	741.0	403.6	214.2	115.4	1345.1
`rcv1`	640.9	350.4	158.1	110.4	6821.1	3974.8	2073.0	1127.7	6299.7
`url`	28500.4	12533.6	8971.3	7610.1	13861.0	8718.2	4793.3	2277.8	NA
`kddb`	6844.9	3757.6	2216.5	1887.8	12801.4	7915.2	4700.8	3336.6	NA
`avazu-app`	905.5	509.7	277.2	183.5	12582.4	8236.6	4655.0	2851.0	NA
`avazu-site`	2171.4	1197.3	631.6	384.9	22892.4	15614.3	9006.8	5557.5	NA
`criteo`	11400.8	6022.1	2830.3	1502.0	100963.0	60580.1	29531.9	16594.7	NA

Table IV: A comparison on training time in seconds between ANT-P, CD and ADAGRAD. For each data set and each approach, we list the time needed to reach the horizontal line in Figure 4. The symbol "NA" indicates that the CD does not finish an outer iteration within three hundred thousand seconds.

`a9a`, it is known that even highly non-linear Gaussian-kernel SVM gives only similar accuracy to a linear classifier [Huang and Lin 2016]. For the seven sparse data sets, three of them are document data, and generally linear classifiers are powerful enough [Yuan et al. 2012b]. However, overall we still see that FM slightly boosts the test accuracy and the log loss.

7. CONCLUSIONS

In this paper, we propose an effective alternating Newton method for solving a multi-block convex reformulation of factorization machines. A sequence of convex sub-problems are solved, where we show that each sub-problem can be written in a form equivalent to regularized logistic/linear regression depending on the chosen loss function. Because variables in a sub-problem may be a matrix rather than a vector, we carefully design a truncated Newton method that employs fast matrix operations. Further, some advanced techniques such as preconditioned conjugate gradient methods and sub-sampled Hessian technique are incorporated into our framework. Through experiments in this paper, we establish the superiority over existing stochastic gradient algorithms and coordinate descent methods when the logistic loss is considered. We also show the effectiveness of the parallelization of our algorithm.

In summary, we have successfully developed a useful algorithm and implementation for training factorization machines. A MATLAB package of the proposed method and programs used for experiments are available at

http://www.csie.ntu.edu.tw/~cjlin/papers/fm.

With the built-in matrix operations, our MATLAB package is as efficient as our C++ implementation but the MATLAB training and prediction modules only need around 150 lines of code. It means that our algorithm can be easily ported into any existing service as long as a good matrix library is available.

REFERENCES

Dimitri P. Bertsekas. 1999. *Nonlinear Programming* (second ed.). Athena Scientific, Belmont, MA 02178-9998.

Table V: Test log losses and accuracies of LR and the two FM formulations (2) and (6). The symbol “NA” indicates that CD is too slow to finish one outer iteration within fifteen thousands seconds, so we do not calculate its test score.

	Model	LR	FM: (2)	FM: (6)		
	Solver	Algorithm 5	ADAGRAD	ADAGRAD	CD	ANT-P
Data set	`a9a`	0.3238	0.3200	0.3200	0.3206	0.3204
	`webspam`	0.2085	0.1151	0.0904	NA	0.0583
	`kddb`	0.2852	0.2652	0.2647	NA	0.2673
	`news20`	0.1194	0.1019	0.0879	0.0824	0.0844
	`rcv1`	0.0740	0.0623	0.0517	0.0506	0.0505
	`url`	0.0315	0.0144	0.0155	NA	0.0157
	`avazu-app`	0.3392	0.3407	0.3300	NA	0.3344
	`avazu-site`	0.4460	0.4377	0.4354	NA	0.4381
	`criteo`	0.4616	0.4474	0.4489	NA	0.4513

(a) Test log losses

	Model	LR	FM: (2)	FM: (6)		
	Solver	Algorithm 5	ADAGRAD	ADAGRAD	CD	ANT-P
Data set	`a9a`	85.03%	85.26%	85.24%	85.21%	85.18%
	`webspam`	92.63%	95.93%	97.00%	NA	98.34%
	`kddb`	88.87%	89.43%	89.62%	NA	89.52%
	`news20`	96.43%	95.98%	96.38%	96.78%	96.61%
	`rcv1`	97.59%	98.05%	98.39%	98.38%	98.39%
	`url`	98.87%	99.53%	99.51%	NA	99.50%
	`avazu-app`	87.09%	87.11%	87.13%	NA	87.06%
	`avazu-site`	80.44%	80.61%	80.67%	NA	80.58%
	`criteo`	78.37%	79.13%	79.04%	NA	78.91%

(b) Test accuracies

Mathieu Blondel, Masakazu Ishihata, Akinori Fujino, and Naonori Ueda. 2016. Polynomial networks and factorization machines: new insights and efficient training algorithms. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*.

Richard H. Byrd, Gillian M. Chin, Will Neveitt, and Jorge Nocedal. 2011. On the use of stochastic Hessian information in optimization methods for machine learning. *SIAM Journal on Optimization* 21, 3 (2011), 977–995.

Richard H. Byrd, Gillian M. Chin, Jorge Nocedal, and Yuchen Wu. 2012. Sample size selection in optimization methods for machine learning. *Mathematical Programming* 134, 1 (2012), 127–155.

Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. 2008. Coordinate descent method for large-scale L2-loss linear SVM. *Journal of Machine Learning Research* 9 (2008), 1369–1398. http://www.csie.ntu.edu.tw/~cjlin/papers/cdl2.pdf

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12 (2011), 2121–2159.

Stanley C. Eisenstat and Homer F. Walker. 1994. Globally convergent inexact newton methods. *SIAM Journal on Optimization* 4, 2 (1994), 393–422.

Gene H. Golub and Charles. F. Van Loan. 2012. *Matrix Computations* (fourth ed.). The Johns Hopkins University Press.

Chia-Hua Ho and Chih-Jen Lin. 2012. Large-scale linear support vector regression. *Journal of Machine Learning Research* 13 (2012), 3323–3348. http://www.csie.ntu.edu.tw/~cjlin/papers/linear-svr.pdf

Fang-Lan Huang, Cho-Jui Hsieh, Kai-Wei Chang, and Chih-Jen Lin. 2010. Iterative scaling and coordinate descent methods for maximum entropy. *Journal of Machine Learning Research* 11 (2010), 815–848. http://www.csie.ntu.edu.tw/~cjlin/papers/maxent_journal.pdf

Hsin-Yuan Huang and Chih-Jen Lin. 2016. Linear and kernel classification: When to use which?. In *Proceedings of SIAM International Conference on Data Mining (SDM)*. http://www.csie.ntu.edu.tw/~cjlin/papers/kernel-check/kcheck.pdf

Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. 2016. Field-aware factorization machines for CTR prediction. In *Proceedings of the ACM Recommender Systems Conference (RecSys)*. http://www.csie.ntu.edu.tw/~cjlin/papers/ffm.pdf

S. Sathiya Keerthi and Dennis DeCoste. 2005. A modified finite Newton method for fast solution of large scale linear SVMs. *Journal of Machine Learning Research* 6 (2005), 341–361.

Paul Komarek and Andrew W. Moore. 2005. *Making Logistic Regression A Core Data Mining Tool: A Practical Investigation of Accuracy, Speed, and Simplicity*. Technical Report TR-05-27. Robotics Institute, Carnegie Mellon University.

Mu-Chu Lee, Wei-Lin Chiang, and Chih-Jen Lin. 2015. Fast matrix-vector multiplications for large-scale logistic regression on shared-memory systems. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*. http://www.csie.ntu.edu.tw/~cjlin/papers/multicore_liblinear_icdm.pdf

Chih-Jen Lin. 2007. Projected gradient methods for non-negative matrix factorization. *Neural Computation* 19 (2007), 2756–2779. http://www.csie.ntu.edu.tw/~cjlin/papers/pgradnmf.pdf

Chih-Jen Lin, Ruby C. Weng, and S. Sathiya Keerthi. 2008. Trust region Newton method for large-scale logistic regression. *Journal of Machine Learning Research* 9 (2008), 627–650. http://www.csie.ntu.edu.tw/~cjlin/papers/logistic.pdf

James Martens. 2010. Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*.

Stephen G. Nash. 2000. A survey of truncated-Newton methods. *Journal of Computational and Applied Mathematics* 124, 1–2 (2000), 45–59.

Stephen G. Nash and Ariefla Sofer. 1996. *Linear and Nonlinear Programming*. McGraw-Hill.

Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. 2011. HOGWILD!: a lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R.S. Zemel, P. Bartlett, F.C.N. Pereira, and K.Q. Weinberger (Eds.). 693–701.

Steffen Rendle. 2010. Factorization machines. In *Proceedings of IEEE International Conference on Data Mining (ICDM)*. 995–1000.

Steffen Rendle. 2012. Factorization machines with libFM. *ACM Transactions on Intelligent Systems and Technology (TIST)* 3, 3 (2012), 57.

Nicol N Schraudolph. 2002. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation* 14, 7 (2002), 1723–1738.

Anh-Phuong Ta. 2015. Factorization machines with follow-the-regularized-leader for CTR prediction in display advertising. In *Proceedings of the IEEE International Conference on Big Data*.

Chien-Chih Wang, Chun-Heng Huang, and Chih-Jen Lin. 2015. Subsampled Hessian Newton methods for supervised learning. *Neural Computation* 27 (2015), 1766–1795. http://www.csie.ntu.edu.tw/~cjlin/papers/sub_hessian/sample_hessian.pdf

Guo-Xun Yuan, Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. 2010. A comparison of optimization methods and software for large-scale l1-regularized linear classification. *Journal of Machine Learning Research* 11 (2010), 3183–3234. http://www.csie.ntu.edu.tw/~cjlin/papers/l1.pdf

Guo-Xun Yuan, Chia-Hua Ho, and Chih-Jen Lin. 2012a. An improved GLMNET for l1-regularized logistic regression. *Journal of Machine Learning Research* 13 (2012), 1999–2030. http://www.csie.ntu.edu.tw/~cjlin/papers/l1_glmnet/long-glmnet.pdf

Guo-Xun Yuan, Chia-Hua Ho, and Chih-Jen Lin. 2012b. Recent advances of large-scale linear classification. *Proc. IEEE* 100, 9 (2012), 2584–2603. http://www.csie.ntu.edu.tw/~cjlin/papers/survey-linear.pdf

Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-scale parallel collaborative filtering for the Netflix prize. In *Proceedings of the Fourth International Conference on Algorithmic Aspects in Information and Management*. 337–348.

Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. 2015. Distributed Newton method for regularized logistic regression. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*.

Received ; revised ; accepted