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

Recently, 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. 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, sub-sampled 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), 24 pages.

1. INTRODUCTION

Binary classification has been an important technique for many practical applications. Assume that the binary result and the observations of an event are described by a label \( y \in \{-1, 1\} \) and an \( n \)-dimensional feature vector \( x \), respectively. The model is a mapping from the feature space to a real number, \( y : \mathbb{R}^n \rightarrow \mathbb{R} \), while the output label is the sign of the output value. Among many binary classification 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,

\[(y_1, x_1), \ldots, (y_l, x_l),\]

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 104R8782 and 105R7872.

Authors' email: \{d01944006,r03944049,b01902037\}@csie.ntu.edu.tw
Author's addresses: W.-S. Chin, Graduate Institute of Networking and Multimedia, National Taiwan University; B.-W. Yuan, Y.-M. Yuan, and C.-J. 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

ACM Transactions on Intelligent Systems and Technology, Vol. 0, No. 0, Article 0, Publication date: 0.
where $y_i$ and $x_i$ are the label and the feature vector of the $i$th training instance, respectively. For an instance $(y, x)$, the output value of FM is defined by

$$
\hat{y}(x) = w^T x + \sum_{j=1}^{n} u_j u_{j'} x_j x_{j'},
$$

where $w \in \mathbb{R}^n$ and $U = [u_1, \ldots, u_n] \in \mathbb{R}^{d \times n}$ are the parameters of FM. The $j$th column of $U$, $u_j$, is the $j$th feature latent representation in a latent space with the dimensionality $d$ specified by the user. In (1), the variable $w$ 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. For FM, the coefficient of the conjunction between $x_j$ and $x_{j'}$ is determined by the inner product of $u_j$ and $u_{j'}$. To determine FM's parameters, we solve the following optimization problem.

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

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 the sign of $\hat{y}(x)$ and the actual label $y$; that is, the model is expected to output larger prediction values for positive instances against those negatives. Among possible losses, we consider the logistic loss $\xi(\delta) = \log(1 + e^{-\delta})$, which is common in binary classification. Note that if $U$ is not considered, (2) is reduced to regularized logistic regression.

Problem (2) is difficult to solve because of its non-convex objective function. All existing approaches for training FM with the logistic loss are stochastic methods; for example, a Markov Chain Monte Carlo method in LIBFM [Rendle 2012] and a stochastic gradient method (ADAGRAD) in LIBFFM [Juan et al. 2016]. If the squared loss is used, then coordinate descent methods (CD) have also been applied to FM [Rendle 2012, Blondel et al. 2016]. However, for the logistic loss it has been known that for the linear case of (2), i.e., logistic regression, CD is less suitable because of needing a relatively large number of exponential/logarithmic operations [Yuan et al. 2012a]. In contrast, [Lin et al. 2008] and [Yuan et al. 2012a] showed that classic optimization approaches such as Newton methods require fewer exponential/logarithmic operations and are competitive in training. Because we aim at training FM with the logistic loss, an interesting question is whether Newton methods can be adopted here.

Unfortunately, the use of Newton methods for (2) is not trivial because of the non-convex loss function. In this work, we consider a variant of FM that is multi-block convex and then propose an alternating Newton method (ANT). Some novel optimization techniques are introduced to reduce the running time. We empirically demonstrate the extraordinary efficiency of the proposed algorithm against stochastic gradient methods. The parallelism of our method is also investigated for the acceleration in multi-threading environments.

The paper is organized as follows. The proposed algorithm, ANT, for learning FM is developed in Section 2. Section 3 investigates some algorithmic issues and implementation techniques. Then, we study the parallelization in Section 4. We discuss related optimization methods for FM in Section 5. The experimental results in Section 6 confirm the effectiveness of the proposed method. Finally, Section 7 concludes our work.
2. ALTERNATING NEWTON METHODS FOR LEARNING FACTORIZATION MACHINES

Because (2) is non-convex, developing a Newton is not straightforward. 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 Newton methods solve a quadratic problem in one iteration.

Unfortunately, FM is not in a multi-block convex form so that each block contains many variables. The reason is that every $u_j^T u_j'$ in (1) is a non-convex term with respect to $u_j$ and $u_j'$. Then, we can only get a small convex sub-problem by fixing all but one $u_j$.

Recently, Blondel et al. [2016] proposed modifying the output function by introducing some new variables, $v_1, \ldots, v_n$, so that finding larger blocks becomes easier. Instead of (1), the output value is changed to

$$\hat{y}(x) = w^T x + \frac{1}{2} \sum_{j=1}^{n} \sum_{j'=1}^{n} u_j^T v_{jj'} x_j x_{j'}$$

(3)

where $V = [v_1, \ldots, v_n] \in \mathbb{R}^{d \times n}$. Note that Blondel et al. [2016] do not consider the 1/2 coefficient of the interaction term. We added it because the new formulation nearly doubles the number of feature conjunctions in (1). The new optimization problem is clearly formed by three blocks of variables $w$, $U$, and $V$:

$$\min_{w,U,V} F(w, U, V),$$

(4)

where

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

(5)

It has been mentioned in [Blondel et al. 2016] that (5) is a multi-block convex function of $w$, $U$, and $V$. That is, (5) becomes a convex sub-problem if two of $w$, $U$, 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 (4) of using the new model in (3) 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.

$$\min_w F(w, U, V),$$

(6)

$$\min_U F(w, U, V),$$

(7)

$$\min_V F(w, U, V).$$

(8)
Algorithm 1 Solving the modified FM problem (4) via alternating minimization.

1: Given an initial solution \((w, U, V)\)
2: while stopping condition is not satisfied do
3: \(w \leftarrow \arg \min_w F(w, U, V)\)
4: \(U \leftarrow \arg \min_U F(w, U, V)\)
5: \(V \leftarrow \arg \min_V F(w, U, V)\)
6: end while

The overall procedure to minimize (4) is summarized in Algorithm 1. In the rest of this section, we discuss how to apply Newton 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 regularized logistic regression.

2.1. Relation Between (4)’s Sub-problems and Regularized Logistic Regression

Although (6)-(8) are different optimization problems, we show that they are all equivalent to logistic regression by the change of variables. Given \(l\) training instances, a logistic regression problem can be written as

\[
\min_{\tilde{w} \in \mathbb{R}^{\tilde{n}}} \tilde{f}(\tilde{w}),
\]

where

\[
\tilde{f}(\tilde{w}) = \frac{\tilde{\lambda}}{2} \|\tilde{w}\|^2 + \sum_{i=1}^{l} \xi(y_i \tilde{y}_i),
\]

\(\tilde{y}_i = \tilde{w}^T \tilde{x}_i + \tilde{c}_i\),

\(\tilde{x}_i\) is the feature vector of the \(i\)th instance, and \(\tilde{c}_i\) is a constant.

Before showing some reformulations of (6)-(8), we note that (3) = \(w^T x + \frac{1}{2} p^T q\),

where

\(p = U x\) and \(q = V x\).

(10)

It is straightforward to see that (6) is in a form of (9) when

\(\tilde{\lambda} = \lambda, \; \tilde{w} = w, \; \tilde{x}_i = x_i,\) and \(\tilde{c}_i = \frac{1}{2} p_i^T q_i\).

Note that constants such as \(\|U\|_F^2\) and \(\|V\|_F^2\) are not considered when updating \(w\). To write (7) in a form of (9), we note that when \(w\) and \(V\) are fixed, (10) can be reformulated as an affine function of \(U\),

\[
\tilde{y}_i = \frac{1}{2} (U x_i)^T q_i + w^T x_i = \text{vec}(U)^T (\frac{1}{2} (x_i \otimes q_i)) + w^T x_i,
\]

where the vectorization operator outputs a column vector by stacking the input matrix’s columns and “\(\otimes\)" denotes the Kronecker product. With \(\|\text{vec}(U)\|_F^2 = \|U\|_F^2\), (7) is in the form of (9) by

\[
\tilde{\lambda} = \lambda', \; \tilde{w} = \text{vec}(U), \; \tilde{x}_i = \frac{1}{2} (x_i \otimes q_i),\) and \(\tilde{c}_i = w^T x_i.
\]

In standard logistic regression, \(\tilde{c}_i = 0\). Note that we do not consider a bias term in the output function.
An Efficient Alternating Newton Method for Learning Factorization Machines

Similarly, (9) becomes (8) if

\[ \tilde{\lambda} = \lambda'', \quad \tilde{\mathbf{w}} = \text{vec}(V), \quad \tilde{x}_i = \frac{1}{2} (x_i \otimes p_i), \quad \text{and} \quad \tilde{c}_i = \mathbf{w}^T x_i. \]  

(13)

We can conclude that (9) can be used to represent all sub-problems in Algorithm 1, so the same optimization algorithm can be used to solve them. We give the gradient and the Hessian-matrix of (9) below for applying optimization algorithms.

\[ \nabla \tilde{f}(\tilde{\mathbf{w}}) = \tilde{\lambda} \tilde{\mathbf{w}} + \tilde{\mathbf{X}}^T \mathbf{b}, \quad (14) \]

\[ \nabla^2 \tilde{f}(\tilde{\mathbf{w}}) = \tilde{\lambda} \mathbf{I} + \tilde{\mathbf{X}}^T \mathbf{D} \tilde{\mathbf{X}}, \quad (15) \]

where \( \tilde{\mathbf{X}} = [\tilde{x}_1, \ldots, \tilde{x}_l]^T \in \mathbb{R}^{l \times \tilde{n}} \) is the data matrix, \( \mathbf{b} = [\xi'(y_1\tilde{y}_1)y_1, \ldots, \xi'(y_l\tilde{y}_l)y_l]^T \in \mathbb{R}^l \), and \( \mathbf{D} \) is a diagonal matrix whose \( i \)th diagonal element is \( \xi''(y_i\tilde{y}_i) \). Note that \( \xi'(\delta) = -\frac{1}{1 + e^\delta} \) and \( \xi''(\delta) = \frac{e^\delta}{(1 + e^\delta)^2} \).

With \( \tilde{\lambda} > 0 \), it is clear that (15) is positive definite, so the optimal solution of (9) is unique.

2.2. Truncated Newton Methods for Solving (9)

We discuss a truncated Newton method to solve (9). At the \( k \)th iteration, Newton method obtains an update direction by minimizing the second-order Taylor expansion at \( \tilde{f}(\tilde{\mathbf{w}}^k) \).

\[ \min_{\tilde{s} \in \mathbb{R}^{\tilde{n}}} \quad \tilde{\mathbf{g}}^T \tilde{s} + \frac{1}{2} \tilde{s}^T \tilde{H} \tilde{s}, \quad (16) \]

where \( \tilde{\mathbf{w}}^k \) is the solution to be improved, \( \tilde{H} = \nabla^2 \tilde{f}(\tilde{\mathbf{w}}^k) \), and \( \tilde{\mathbf{g}} = \nabla \tilde{f}(\tilde{\mathbf{w}}^k) \). Because \( \tilde{H} \) is positive definite, (16) is equivalent to a linear system,

\[ \tilde{H} \tilde{s} = -\tilde{\mathbf{g}}, \quad (17) \]

In practice, (17) may not need to be exactly solved, especially in early iterations. Therefore, truncated Newton methods have been introduced to approximately solve (17) while maintain the convergence [Nash 2000]. Iterative methods such as conjugate gradient method (CG) are often used for approximately solving (17), so at each iteration an inner iterative process is involved. Another difficulty to solve (17) is that storing the Hessian matrix \( \tilde{H} \) in \( \mathcal{O}(n^2) \) space is not possible if \( \tilde{n} \) is large. To overcome the memory difficulty, Komarek and Moore [2005] and Keerthi and DeCoste [2005] showed that for problems like (9), the conjugate gradient method that mainly requires a sequence of Hessian-vector products can be performed without explicitly forming the Hessian matrix:

\[ \tilde{H} \tilde{s} = \lambda \tilde{s} + \tilde{\mathbf{X}}^T (\mathbf{D}(\tilde{\mathbf{X}} \tilde{s})), \quad (18) \]

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.

In Newton or many other optimization methods, after a direction is found, we must decide the step size taken along that direction. The purpose is to ensure the sufficient decrease of the function value and hence the convergence. Two major methods are trust region and line search. Here we consider a standard back-track line-search procedure and discuss a common trick for efficient computation. Specifically, we find the largest \( \theta \in \{1, \beta, \beta^2, \ldots\} \) such that

\[ \tilde{f}(\tilde{\mathbf{w}}^k + \theta \tilde{s}) \leq \tilde{f}(\tilde{\mathbf{w}}^k) + \theta \nu \tilde{g}^T \tilde{s}, \quad (19) \]
where $\nu \in (0,1)$ and $\beta \in (0,1)$ are pre-specified constants. Recalculating the function value at each $\hat{w}^k + \theta \hat{s}$ is expensive, where the main cost is on calculating $(\hat{w} + \theta \hat{s})^T \hat{x}_i$, $\forall i$. However, for logistic regression, the following trick in [Yuan et al. 2010] can be employed. Assume that 

$$\hat{y}_i = (\hat{w}^k)^T \hat{x}_i + \hat{c}_i$$

and

$$\hat{\Delta}_i = \hat{s}^T \hat{x}_i, \quad i = 1, \ldots, l$$

are available. At an arbitrary $\theta$, we can calculate

$$(\hat{w}^k + \theta \hat{s})^T \hat{x}_i + \hat{c}_i = \hat{y}_i + \theta \hat{\Delta}_i$$

(21)

to get the new function value. By further maintaining

$$(\hat{w}^k)^T \hat{s}, \quad \|\hat{s}\|^2, \text{ and } \hat{g}^T \hat{s},$$

(22)

the total cost of checking the condition of (19) is merely $O(l)$ because

$$\hat{f}(\hat{w}^k + \theta \hat{s}) - \hat{f}(\hat{w}^k) = \frac{\lambda}{2} \left(2\theta(\hat{w}^k)^T \hat{s} + \theta^2 \|\hat{s}\|^2\right) + \sum_{i=1}^l \log \left(1 + e^{-y_i(\hat{y}_i + \theta \hat{\Delta}_i)}\right)$$

$$- \sum_{i=1}^l \log \left(1 + e^{-y_i \hat{y}_i}\right).$$

(23)

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

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

and the last value obtained in (21) can be can be used as the new $\hat{y}_i$ in the next iteration; see (20). Notice that when $\hat{y}_i, \forall i$ are available, we can calculate $e^{y_i \hat{y}_i}, \forall i$ and obtain elements of $d$ and $D$ respectively by

$$b_i = \frac{-y_i}{1 + e^{y_i \hat{y}_i}} \quad \text{and} \quad D_{ii} = \frac{e^{y_i \hat{y}_i}}{(1 + e^{y_i \hat{y}_i})^2}.$$  

(24)

Algorithm 2 summarizes the above procedure. The computational complexity of one iteration in Algorithm 2 can be estimated by

$$(\text{cost of } \hat{f} \text{ and } \hat{g}) + (\text{# of CG iterations}) \times (\text{cost of } \hat{H} \hat{s} + \|\hat{g}\|) + (\text{# of line search iterations}) \times O(l).$$

(25)

Note that the above line-search procedure is also responsible for calculating the next function value, so in (25), except the $O(l)$ cost for calculating (21) at each line-search step, we consider all initialization tasks such as calculating $\Delta_i, \forall i$ in (20) and (22) as the cost of the function evaluation.

Although we do not exactly solve (17), Algorithm 2 asymptotically converges to the optimal point if $\hat{s}$ satisfies

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

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 2 can be used for solving the three sub-problems because they are all in the form of (9). However, without taking their problem structures into account, the implementation may be inefficient. In particular, (7) and (8) significantly differ from traditional logistic regression because they are matrix-variable problems. In Section 2.3, we will discuss some techniques for an efficient implementation of the truncated Newton method.
Algorithm 2 A CG-based truncated Newton method with line search for solving (9).

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

2.3. An Efficient Implementation of Algorithm 2 for Solving Sub-problems (7)-(8)

Among the three sub-problems, (6) is very close to standard logistic regression, so the implementation is similar to past works. However, (7) and (8) are more different because their variables are matrices. Here, we discuss details of solving (7), because the situation for (8) is similar.

First, we check the function evaluation, in which the main task is on calculating $\tilde{y}_i$, $\forall i$. From (10)-(12),

$$\tilde{y}_i = \text{vec}(U)^T \tilde{x}_i + \tilde{c}_i$$

$$= \frac{1}{2} \text{vec}(U)^T (x_i \otimes q_i) + \tilde{c}_i$$

$$= \frac{1}{2} q_i^T U x_i + \tilde{c}_i, \quad i = 1, \ldots, l,$$

or equivalently

$$\tilde{y} = \tilde{X} \text{vec}(U) + \tilde{c}$$

$$= \frac{1}{2} \left( Q^T \ast (XU^T) \right) 1_{d \times 1} + \tilde{c},$$

where $Q = [q_1, \ldots, q_l] \in \mathbb{R}^{d \times l}$, "$\ast$" stands for the element-wise product of two matrices, and $1_{d \times 1}$ is the vector of ones. We can calculate and store $XU^T$ as a dense matrix regardless of whether $X$ is sparse or not; the cost is $O(d \times (\# \text{nnz}))$, where $(\# \text{nnz})$
is the total number of non-zeros in $X$. Similar to (27), $\tilde{\Delta}$ used in line search can be computed via

$$\tilde{\Delta}_i = \frac{1}{2} q_i^T S x_i, \ \forall i \text{ or equivalently } \tilde{\Delta} = \tilde{X} \tilde{s} = \frac{1}{2} (Q^T \ast (XS^T)) 1_{d \times 1},$$

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

Then, we investigate how to compute the gradient of (7). By some reformulations, we have

$$\nabla \tilde{f}(\text{vec}(U)) = \lambda' \text{vec}(U) + \tilde{X}^T b$$

$$= \lambda' \text{vec}(U) + \frac{1}{2} \sum_{i=1}^{l} (b_i x_i \otimes q_i)$$

$$= \lambda' \text{vec}(U) + \frac{1}{2} \sum_{i=1}^{l} \text{vec}(b_i q_i x_i^T)$$

$$= \lambda' \text{vec}(U) + \frac{1}{2} \text{vec} \left( \sum_{i=1}^{l} (q_i b_i x_i^T) \right)$$

$$= \lambda' \text{vec}(U) + \frac{1}{2} \text{vec}(Q \text{diag}(b) X),$$

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

Next we discuss Hessian-vector products in CG. From (18) and $\tilde{s} = \text{vec}(S)$,

$$\hat{H} \tilde{s} = \lambda' \text{vec}(S) + \tilde{X}^T (D(\tilde{X} \text{vec}(S)))$$

$$= \lambda' \text{vec}(S) + \tilde{X}^T (Dz)$$

$$= \lambda' \text{vec}(S) + \sum_{i=1}^{l} (Dz)_i \tilde{x}_i$$

$$= \lambda' \text{vec}(S) + \sum_{i=1}^{l} (Dz)_i (x_i \otimes q_i)$$

$$= \lambda' \text{vec}(S) + \frac{1}{2} \sum_{i=1}^{l} \text{vec}(q_i (Dz)_i x_i^T)$$

$$= \lambda' \text{vec}(S) + \frac{1}{2} \text{vec}(Q \text{diag}(Dz) X),$$

where

$$z_i = \tilde{x}_i^T \tilde{s} = \frac{1}{2} (x_i^T \otimes q_i^T) \text{vec}(S) = \frac{1}{2} q_i^T S x_i, \ i = 1, \ldots, l$$

or equivalently

$$z = \tilde{X} \text{vec}(S) = \frac{1}{2} (Q^T \ast (XS^T)) 1_{d \times 1}$$

by following the same calculation in (28). Note that $Dz$ is the product between a diagonal matrix $D$ and a vector $z$. We investigate the complexity of the Hessian-vector
The first term in (30) is a standard vector scaling that costs $O(nd)$. For the second term in (30), we separately consider two major steps:

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

In (34), we have seen that the first step can be done by the same way to compute (28). For the second step, we can refer to the calculation of $\tilde{X}^T b$ in (29). Therefore, following earlier discussion, the cost of one Hessian-vector product is $O(d \times (\# \text{nnz}))$.

A disadvantage of the above setting is that the data matrix $X$ is accessed twice in the two different steps. From (31) and (33), the Hessian-vector product can be represented as

$$\lambda' \text{vec}(S) + \frac{1}{4} \sum_{i=1}^{l} \text{vec} (q_i D_i (q_i^T S x_i) x_i^T).$$

From this instance-wise representation, the second term can be computed by a single loop over $x_1, \ldots, x_l$. While $x_i$ is still used twice: $Sx_i$ and $q_i z_i$, between them no other $x_j$, $j \neq i$ are accessed and therefore $x_i$ tends to remain at a higher layer of the memory hierarchy. Such a setting of getting better data locality has been considered for training logistic regression [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 (34). Therefore, the decision of using (35) or not may rely on whether $X$ is a sparse matrix.

In line search, besides the calculation of $\tilde{\Delta}$ in (28), from (22) we also need

$$\langle U, S \rangle, \|S\|_F^2,$$

where $G$ is the matrix form of (7)'s gradient and $\langle \cdot, \cdot \rangle$ is the inner product of two matrices.

Algorithm 3 summarizes the details in our Newton procedure for solving (7). Matrix representations are used, so we never have to reshape $S$ to vec$(S)$ or $U$ to 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 3, the complexity per Newton iteration is

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

in which the term $2 \times O(d \times (\# \text{nnz}))$ is from the costs of (28) and (29) for function and gradient evaluation. Note that the $O(nd)$ cost for the quantities in (36) and other places is not listed here because it is no more than $O(d \times (\# \text{nnz}))$.

2.4. Convergence Guarantee and Stopping Conditions

Algorithm 1 is under the framework of block coordinate descent methods, which sequentially update one block of variables while fixing all remaining blocks. 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]. Each of our sub-problems (6)-(8) 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 (4).

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

$$\|\nabla F(w, U, V)\| \leq \epsilon \|\nabla F(w_{\text{init}}, U_{\text{init}}, V_{\text{init}})\|,$$

where $0 < \epsilon < 1$ is a small stopping tolerance and $(w_{\text{init}}, U_{\text{init}}, V_{\text{init}})$ indicates the initial point of the model. The use of this type of criteria includes, for example, [Lin 2007] for non-negative matrix factorization.
Algorithm 3

An implementation of Algorithm 2 for solving (7) by operations on matrix variables without vectorizing them.

1: Given $0 < \tilde{\epsilon} < 1$ and the current $(w, U, V)$.
2: $Q \leftarrow VX^T$
3: Compute and cache $	ilde{y} = \frac{1}{2} (Q^T \ast (UX^T)) 1_{d \times 1} + Xw$.
4: $f \leftarrow \frac{\lambda'}{2} \|U\|_F^2 + \sum_{i=1}^l \log \left(1 + e^{-y_i \tilde{y}_i}\right)$.
5: for $k \leftarrow \{0, 1, \ldots \}$ do
6: Calculate and store $e^{y_i \tilde{y}_i}$ and then obtain $b_i$ by (24), $\forall i$.
7: $G \leftarrow \lambda' U + \frac{1}{2} Q \text{diag}(b)$
8: if $k = 0$ then
9: $\|G^0\|_F \leftarrow \|G\|_F$
10: end if
11: if $\|G\|_F \leq \tilde{\epsilon} \|G^0\|_F$ then
12: Output $U$ as the solution of (7).
13: break
14: end if
15: Compute $D_{ii}$ via (24), $\forall i$.
16: Solve (17) approximately via CG to get an update direction $S$.
17: Prepare variables listed in (36) and $\tilde{\Delta} = \frac{1}{2} (Q^T \ast (XS^T)) 1_{d \times 1}$
18: for $\theta \leftarrow \{1, \beta, \beta^2, \ldots \}$ do
19: $\delta \leftarrow \frac{\lambda'}{2} (2\theta \langle U, S \rangle + \theta^2 \|S\|_F^2) + \sum_{i=1}^l \log \left(1 + e^{-y_i (\tilde{y}_i + \theta \tilde{\Delta}_i)}\right) - \sum_{i=1}^l \sum_{i=1}^l \log \left(1 + e^{-y_i \tilde{y}_i}\right)$.
20: if $\delta \leq \theta \nu \langle G, S \rangle$ then
21: $U \leftarrow U + \theta S$
22: $f \leftarrow f + \delta$
23: $\tilde{y} \leftarrow \tilde{y} + \theta \tilde{\Delta}$
24: break
25: end if
26: end for
27: end for

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 (38) may never be reached. In Section 2.2, we consider a relative stopping condition, $\|\nabla \tilde{f} (\tilde{w})\| \leq \tilde{\epsilon} \|\nabla \tilde{f} (\tilde{w}^0)\|$, (39)

where $\tilde{w}^0$ is the initial point in solving the sub-problem. Take the sub-problem of $U$ as an example. The gradient norm of (4) with respect to $U$ is equivalent to the gradient norm of $U$'s sub-problem: $\|\nabla_U f(w, U, V)\|_F = \|\nabla \tilde{f} (\text{vec}(U))\|$. If $\|\nabla_U f(w, U, V)\|_F > 0$, Algorithm 3 conducts at least one Newton step. The reason is that (39) is violated if we do not change $U$. In this case, the function value of $F(w, U, V)$ is decreased. Therefore, unless $\|\nabla_w f(w, U, V)\| = \|\nabla_U f(w, U, V)\|_F = \|\nabla_V f(w, U, V)\|_F = 0$, one of the three blocks must be updated. That is, if the condition in (38) is not satisfied yet, our alternating minimization procedure must continue to update $(w, U, V)$ rather than stay at the same point.
3. TECHNIQUES TO ACCELERATE TRUNCATED NEWTON METHODS FOR SOLVING THE SUB-PROBLEMS

From (37), 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.

3.1. Preconditioned Conjugate Gradient Methods

The first technique aims to reduce the number of CG iterations for solving (17) 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

$$\hat{H} \hat{s} = \hat{g},$$

(40)

where $\hat{H} = \tilde{M}^{-1} \tilde{H} \tilde{M}^{-T}$ and $\hat{g} = \tilde{M}^{-1} \tilde{g}$. Once $\hat{s}$ is found, the solution of (17) can be recovered by $\tilde{s} = \tilde{M}^{-T} \hat{s}$. If $H$ is perfectly factorized (i.e., $\hat{H} = \tilde{M} \tilde{M}^T$), (40) 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 logistic regression in [Lin et al. 2008], the use of preconditioned conjugate gradient methods (PCG) has been studied. They point out that the implicit use of $H$ 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 $H$ 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 logistic regression, 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 (4) rather than a single logistic regression 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 (17) as an example.

$$\tilde{M} = \tilde{M}^T = \text{diag}(\tilde{h}),$$

(41)

where

$$\tilde{h} = \sqrt{\lambda 1_{\tilde{h} \times 1} + \sum_{i=1}^{l} D_{ii} (\tilde{x}_i \cdot \tilde{x}_i)}.$$  

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-preconditioned linear system is

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

(42)

To illustrate the computational details of PCG, we consider (7)’s Newton linear system as an example. First, we discuss the construction of the preconditioner. Consider the
sub-problem (7) as an example. From
\[ \tilde{x}_i = \frac{1}{2} (x_i \otimes q_i) = \frac{1}{2} \text{vec}(q_i x_i^T), \]
we have
\[ \sum_{i=1}^{l} D_{ii} (\tilde{x}_i \cdot \tilde{x}_i) = \frac{1}{4} \sum_{i=1}^{l} D_{ii} \text{vec} ((q_i \cdot q_i) (x_i \cdot x_i)^T) \]
(43)
\[ = \frac{1}{4} \sum_{i=1}^{l} \text{vec} ((q_i \cdot q_i) D_{ii} (x_i \cdot x_i)^T) \]
\[ = \frac{1}{4} \text{vec} ((Q \cdot Q) D (X \cdot X)). \]
Thus, the preconditioner of \( U \)'s sub-problem can be obtained via
\[ \tilde{M} = \tilde{M}^T = \text{diag} \left( \sqrt{\lambda'_{1 \times 1}} + \frac{1}{4} \text{vec} ((Q \cdot Q) D (X \cdot X)) \right) \]
or without vectorization
\[ M = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix} + \frac{1}{4} (Q \cdot Q) D (X \cdot X) \in \mathcal{R}^{d \times n}. \] (44)
We point out that the cost to obtain (44) is \( O(d \times (\# \text{nnz})) \).

Next, we look into the computation of (42). Let \( \hat{S} \in \mathcal{R}^{d \times n} \) be the matrix form of the vector \( \hat{s} \) such that \( \hat{s} = \text{vec}(\hat{S}) \). It is possible to present (42) in terms of \( \hat{S} \) to avoid vectorizations. First, the matrix form of \( \tilde{M}^{-1} \hat{s} \) is \( \hat{S} / M \), where “/” denotes the element-wise division. By substituting \( S = \hat{S} / M \) into (32) and then element-wisely dividing the resulting matrix by \( M \) again, (42) can be written as
\[ \left( \lambda' \left( \hat{S} / M \right) + \frac{1}{2} Q \text{diag} (D \hat{z}) X \right) / M, \]
(45)
where
\[ \hat{z} = \frac{1}{2} \left( Q^T \cdot \left( X \left( \hat{S} / M \right)^T \right) \right) 1_{d \times 1}. \]
For the computational complexity, besides \( O(d \times (\# \text{nnz})) \) from what we discussed in (32), the newly introduced divisions associated with \( M \) require \( O(nd) \) operations. Unless the data matrix is very sparse, from \( n \leq (\# \text{nnz}) \), 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 4 for a PCG procedure of using matrix variables and operations for solving the Newton system of \( U \)'s sub-problem.

3.2. 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
Algorithm 4 A preconditioned conjugate gradient method for solving (17) by operations on matrix variables. Linear systems in the sub-problem of \( U \) are considered.

1: Given \( 0 < \eta < 1 \) and \( G \), the gradient matrix of the sub-problem (i.e., the matrix form of (29)). Let \( \hat{S} = 0_{d \times n} \).
2: Compute \( M \) via (44).
3: Calculate \( R = -G \cdot M, \hat{D} = R, \) and \( \gamma^0 = \gamma = \|R\|_F^2 \).
4: \textbf{while} \( \sqrt{\gamma} > \eta \sqrt{\gamma^0} \) \textbf{do}
   5: \( \hat{D}_h \leftarrow \hat{D} \cdot M \)
   6: \( \hat{z} \leftarrow (Q^T \cdot (X \hat{D}_h)^T) 1_{d \times 1} \)
   7: \( \hat{D}_h \leftarrow (\lambda' \hat{D}_h + Q \text{diag}(D\hat{z}) X) / M \)
   8: \( \alpha \leftarrow \gamma / \langle \hat{D}, \hat{D}_h \rangle \)
   9: \( \hat{S} \leftarrow \hat{S} + \alpha \hat{D} \)
  10: \( R \leftarrow R - \alpha D_h \)
  11: \( \gamma_{\text{new}} \leftarrow \|R\|_F^2 \)
  12: \( \beta \leftarrow \gamma_{\text{new}} / \gamma \)
  13: \( \hat{D} \leftarrow R + \beta \hat{D} \)
  14: \( \gamma \leftarrow \gamma_{\text{new}} \)
5: \textbf{end while}
16: Output \( S = \hat{S} / M \) as the solution.

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 \( \text{Pr}(y, \tilde{x}) \).

Then, (15) can be interpreted as an empirical estimation of

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

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

\[
\tilde{H}_{\text{sub}} = \tilde{\lambda} I + \frac{l}{|L|} \sum_{i \in L} D_{ii} \tilde{x}_i \tilde{x}_i^T,
\]

\[
= \tilde{\lambda} I + \frac{l}{|L|} \tilde{X}_{L,:}^T D_{L,L} \tilde{X}_{L,:}, \tag{46}
\]

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

\[
\tilde{H}_{\text{sub}} \hat{s} = -\tilde{g}. \tag{47}
\]

Consider the sub-problem (7) as an example. From (32) and (35), 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 (47) can be conducted by

\[
\lambda' S + \frac{l}{2|L|} Q_{:,L} \text{diag}(D_{L,L} z_{L}) X_{L,:},
\]
where $Q_{:,L} \in \mathbb{R}^{d \times |L|}$ is a sub-matrix of $Q$ containing $q_i, i \in L$ and
\[
z_L = \frac{1}{2} (Q_{:,L} \cdot (X_{L,:} S^T)) 1_{d \times 1}. \tag{48}\]
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
\[
O \left( \frac{|L| d}{l} \times (\# \text{nnz}) \right),
\]
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 (47). 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.

4. 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 2.1 that each sub-problem is equivalent to a logistic regression problem, our approach follows past developments of parallel Newton methods for logistic regression (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 $x_1, \ldots, x_l$ are used without the sub-sampling technique in Section 3.2. 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 $z$ in Section 3.2. From (27) and (29), the parallelized forms of the output values and the gradient respectively are
\[
\tilde{y} = \begin{bmatrix}
\frac{1}{2} (Q^T_{:,L_1} \cdot (X_{L_1,:} U^T)) 1_{d \times 1} + X_{L_1,:} w \\
\vdots \\
\frac{1}{2} (Q^T_{:,L_s} \cdot (X_{L_s,:} U^T)) 1_{d \times 1} + X_{L_s,:} w
\end{bmatrix}, \tag{49}
\]
\[
\nabla \tilde{f}(U) = \lambda' U + \frac{1}{4} \sum_{r=1}^{s} (Q_{:,L_r} \text{diag}(b_{L_r}) X_{L_r,:}), \tag{50}
\]
where tasks indexed by $L_r$ is assigned to the $r$th thread. Form (32), parallel Hessian-vector product can be done by considering
\[
\lambda' S + \frac{1}{4} \sum_{r=1}^{s} (Q_{:,L_r} \text{diag}(D_{L_r} z_{L_r}) X_{L_r,:}). \tag{51}
\]
For PCG, the diagonal preconditioner can be obtained via
\[
M = \sqrt{\lambda'} \begin{bmatrix} 1 & \ldots & 1 \\ 0 & \ddots & 0 \\ 0 & \ldots & 1 \end{bmatrix} + \frac{1}{4} \sum_{r=1}^{s} (Q_{:,L_r} \cdot Q_{:,L_r} D_{L_r} (X_{L_r,:} \cdot X_{L_r,:}) \in \mathbb{R}^{d \times n}. \tag{52}
\]
Besides (49), for completing any of (50), (51), and (52), 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 (51), we can compute

$$B_1 = Q_{:,L_1} \text{diag}(D_{L_1}z_{L_1})X_{L_1,:} \quad \text{and} \quad B_2 = Q_{:,L_2} \text{diag}(D_{L_2}z_{L_2})X_{L_2,:}.$$  

(53)

in parallel, and sum up the two matrices to obtain the second term of (51). Notice that from (33) and (51), the $r$th worker only maintains $Q_{:,L_r}$ and $z_{:,L_r}$ instead of the whole $Q$ and $z$.

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

5. OTHER OPTIMIZATION METHODS: STOCHASTIC GRADIENT METHODS AND THEIR PARALLELIZATION

It is mentioned in Section 1 that currently stochastic methods are the major way to train FM. In this section, we study an effective implementation with its parallelization in order to conduct a comparison with our proposed method. We adopt ADAGRAD proposed by Duchi et al. [2011]. Its use for FM includes, for example, [Juan et al. 2016] and [Ta 2015].

Consider $l$ training instances $(y_i, x_i)$, $i = 1, \ldots, l$ again. ADAGRAD can be used to solve optimization problems in the following form.

$$\min_{\hat{w} \in \mathbb{R}^\tilde{n}} \sum_{i=1}^{l} f(\hat{w}; y_i, x_i),$$  

(54)

where $\hat{w}$ is the unknown parameter and $f(\hat{w}; y_i, x_i)$ is the cost function that the $i$th instance incurs. At the $k$th iteration, an instance $(y_i, x_i)$ is sampled from the $l$ training instances to calculate the stochastic gradient, $\hat{g}^k = \nabla f(\hat{w}^k; y_i, x_i)$. Then, the current solution is updated via

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

where $\eta_0 > 0$ is a pre-specified constant and $\tilde{G}^{-1/2}$ is the inverse of the root of a diagonal matrix defined by

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

The complete procedure that ADAGRAD solves (54) is shown in Algorithm 5.

**Algorithm 5 ADAGRAD.**

1. Given initial solution $\hat{w}^0$, number of iterations $\tilde{k}$, and $\eta_0 > 0$.
2. Initialization $\tilde{G}$ with zeros.
3. for $k \leftarrow 1, \ldots, \tilde{k}$ do
   4. Draw an index $i \in \{1, \ldots, l\}$
   5. $\hat{g} \leftarrow \nabla f(\hat{w}^k; y_i, x_i)$
   6. $\tilde{G} \leftarrow \tilde{G} + \text{diag}\left(\hat{g}\right)^2$
   7. $\hat{w}^{k+1} \leftarrow \hat{w}^k - \eta_0 \tilde{G}^{-1/2} \hat{g}$
4. end for
To apply ADAGRAD to learn the modified FM, we need to rewrite (4) into a form of (54). Let
\[
\tilde{w} = [w^T, u_1^T, \ldots, u_n^T, v_1^T, \ldots, v_n^T]^T \in \mathbb{R}^{n+2nd}.
\] (55)
We argue that if
\[
f(\tilde{w}; y_i, x_i) = \sum_{j \in N_i} \left( \lambda \frac{2}{|\Omega_j|} w_j^2 + \lambda' \frac{2}{|\Omega_j|} ||u_j||^2 + \lambda'' \frac{2}{|\Omega_j|} ||v_j||^2 \right) + \xi(y_i, \hat{y}(x_i)),
\] (56)
then (54) is equivalent to (4), where \(N_i\) is the index set of the \(i\)th instance’s non-zero features, \(\Omega_j\) is the indexes set of the instances whose \(j\)th features are not zero, and \(|\cdot|\) returns the size when the input is a set. Because the summation of the loss terms in (56) over \(i = 1, \ldots, l\) is obviously the loss term in (4), we only check if the regularization terms in (56) lead to the same regularization terms in (5). First, for \(w\)’s regularizaitoin, we have
\[
\sum_{i=1}^l \sum_{j \in N_i} \frac{\lambda}{2|\Omega_j|} w_j^2 = \frac{\lambda}{2} \sum_{j=1}^n \sum_{i \in \Omega_j} \frac{1}{|\Omega_j|} w_j^2 = \frac{\lambda}{2} \sum_{j=1}^n w_j^2 = \frac{\lambda}{2} ||w||^2.
\]
By an analogous derivation, inversely scaling \(\|u_j\|^2\) and \(\|v_j\|^2\) by \(|\Omega_j|\) generates the desired regularization in (5). The equivalence between (56) and (5) is therefore verified.

From (56) and the definition of \(\tilde{w}\) in (55), we deduce the stochastic gradient with respect to the model parameters in each iteration. For \(w\), we have
\[
\frac{\partial f(\tilde{w}; y_i, x_i)}{\partial w_j} = \begin{cases} \lambda \frac{2}{|\Omega_j|} w_j + \xi'(y_i, \hat{y}(x_i)) y_i x_{ij} & \text{if } j \in N_i, \\ 0 & \text{otherwise.} \end{cases}
\] (57)
Similarly, taking the partial derivative of (56) with respect to \(u_j\) and \(v_j\) yields the needed equations below for the other coordinates.
\[
\frac{\partial f(\tilde{w}; y_i, x_i)}{\partial u_j} = \begin{cases} \lambda' \frac{2}{|\Omega_j|} u_j + \frac{1}{2} \xi'(y_i, \hat{y}(x_i)) y_i q_j x_{ij} & \text{if } j \in N_i, \\ 0_{d \times 1} & \text{otherwise.} \end{cases}
\] (58)
\[
\frac{\partial f(\tilde{w}; y_i, x_i)}{\partial v_j} = \begin{cases} \lambda'' \frac{2}{|\Omega_j|} v_j + \frac{1}{2} \xi'(y_i, \hat{y}(x_i)) y_i p_j x_{ij} & \text{if } j \in N_i, \\ 0_{d \times 1} & \text{otherwise.} \end{cases}
\]
Substituting (57) and (58) into the step that calculates the stochastic gradient in Algorithm 5, we get the procedure to solve (4) by ADAGRAD.

Next, we analyze the computational complexity of one for-loop iteration in Algorithm 5. If instance \((y_i, x_i)\) is drawn, we spend \(O(\|N_i|d)\) operations to calculate the output value \(\hat{y}_i\) and use (57), (58) to compute the \((\|N_i| + 2|N_i|)\) non-zero values of \(\hat{y}\). Then, the diagonal elements of \(\hat{G}\) correspond to the non-zero coordinates in \(\hat{y}\) are updated. Finally, the coordinates of \(\tilde{w}\) corresponding to the non-zero coordinates in \(\hat{g}\) are adjusted. The total complexity in one iteration is therefore \(O(|N_i|d)\) and we expect \(O(d \times (\# \text{nnz}))\) operations to process an epoch (i.e., \(l\) instances). Recalling (37), we note that one CG iteration costs nearly as much as processing one ADAGRAD epoch.

Parallel stochastic gradient methods have received much attention in large-scale machine learning. Our implementation follows [Niu et al. 2011] to launch several threads to execute the for-loop in Algorithm 5 without the synchronization between them.
Table I: Data statistics and parameters used in experiments. Density is the average number of non-zero features per instance.

<table>
<thead>
<tr>
<th>Data set</th>
<th>$l$</th>
<th>$n$</th>
<th>density</th>
<th>$\lambda$</th>
<th>$\lambda'$</th>
<th>$\lambda''$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a9a</td>
<td>26,049</td>
<td>122</td>
<td>11.367%</td>
<td>64</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>webspam</td>
<td>280,000</td>
<td>246</td>
<td>34.599%</td>
<td>0.25</td>
<td>0.0625</td>
<td>0.0625</td>
</tr>
<tr>
<td>kddb</td>
<td>9,464,836</td>
<td>651,166</td>
<td>0.000%</td>
<td>1</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>news20</td>
<td>16,009</td>
<td>1,355,191</td>
<td>0.000%</td>
<td>0.0625</td>
<td>0.0625</td>
<td>0.0625</td>
</tr>
<tr>
<td>rcv1</td>
<td>677,399</td>
<td>47,236</td>
<td>0.001%</td>
<td>0.0625</td>
<td>0.0625</td>
<td>1</td>
</tr>
<tr>
<td>url</td>
<td>1,916,904</td>
<td>3,231,961</td>
<td>0.000%</td>
<td>16</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

6. EXPERIMENTS

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 3. The running time comparison on ANT and a stochastic gradient implementation using multiple threads is presented in Section 6.5.

6.1. Data Sets and Parameters

We consider several real-world data sets listed in Table I. All data sets but kddb can be downloaded at LIBSVM data sets page. For webspam we consider the uni-gram version. For kddb, we generate it from the raw data of the “bridge to algebra” track in KDD-Cup 2010 by following [Juan et al. 2016]. Note that the kddb set used here is different from the version on LIBSVM page. 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. 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.

In problem (4), 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 $\{0.5, 1, 4, 16, 64\}$ as the search range of each regularization parameter and check all the combinations. Values leading to the lowest logistic loss on the validation are selected. Table I gives the parameters identified by the process. Furthermore, we set $w_{init} = 0$ and choose every element of $U_{init}$ and $V_{init}$ uniformly random from $[-1/\sqrt{d}, 1/\sqrt{d}]$. For any comparison between different settings or algorithms, we ensure that the same initial point is used. For the stopping condition of PCG, we set $\eta = 0.3$ so Algorithm 4 terminates once

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

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.

---

2LIBSVM data sets page is available at [www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets](http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets).

3The raw data set can be downloaded at [http://pslcdatashop.web.cmu.edu/KDDCup/downloads.jsp](http://pslcdatashop.web.cmu.edu/KDDCup/downloads.jsp). The script to generate the data used here is available at [www.csie.ntu.edu.tw/~cjlin/fm/exps](http://www.csie.ntu.edu.tw/~cjlin/fm/exps).
All the algorithms are implemented in C++ for a fair comparison. For the storage 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 4. 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].

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

To see the effect of different stopping tolerances in Algorithm 2, we consider \( \bar{\epsilon} = 0.01, 0.1, \text{ and } 0.8 \), 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 the log-scaled distance between the current solution and a local optimal function value.

\[
\frac{F(w,U,V) - F^*}{F^*},
\]

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 \( \epsilon \) 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 \( w \) 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 5]. Our relative stopping condition in (39) automatically achieves that; although \( \bar{\epsilon} \) is fixed in the entire procedure, (39) is a strict condition in the final stage of the optimization process because of a small \( \| \nabla f(\bar{w}) \| \).

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., \( \bar{\epsilon} = 0.01 \)), 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 \( \bar{\epsilon} = 0.8 \) and extract the three convex sub-problems (6)-(8) 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 \( \bar{\epsilon} = 0.8 \), 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.

[www.csie.ntu.edu.tw/~cjlin/papers/logistic/pcgnote.pdf]
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 (59).

6.4. Effects of the Use of Sub-sampled Hessian Matrix
In Section 3, 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 and kddb. 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.

6.5. A Comparison on ANT and Stochastic Gradient Algorithms
We compare the running time of ANT-P and the stochastic gradient method described in Section 5 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 set $\tilde{\epsilon} = 0.8$ as the stopping tolerance of solving sub-problems and use 10% of data for constructing the sub-Hessian matrix. For the parameter $\eta_0$ in the stochastic gradient method detailed in Algorithm 5, we use 0.1. Following the format of Figure 1, we draw the convergence speed with different numbers of threads in Figure 4. We observe that the parallelization of both methods is effective. Their training time decreases as the number of
threads increases. Further, in either single- or multi-thread setting, ANT-P is always significantly faster than the stochastic gradient method.

6.6. Test Performances
From a practical perspective, it is important to examine if the modified FM problem (4) can perform as well as the original FM problem (2). To this end, for each data set, we compare their test accuracies and logistic losses in Tables II-III. To see how FM improves over a linear model, we include results of logistic regression. For a fair comparison, we use $d = 40$ for (1) and $d = 20$ for (3), so they have the same number of model parameters. For the solvers, we consider the truncated Newton method in Algorithm 2 for logistic regression, the stochastic gradient method in Section 5 for (2) and (4), and ANT-P for (4). As shown in Tables II-III, the performance difference between problems (2) and (4) is not significant, so the modified FM formulation in (4) seems to give similar learning capability. Regarding the comparison between FM and logistic regression, FM gives a much higher test score than logistic regression on webspam. This data set has less features than its instances, so feature conjunction seems to be useful. For other data sets, the performance difference is smaller. For 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 four 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 logistic loss.
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 (59).

Table II: Test logistic losses of logistic regression (LR) and the two FM formulations (2) and (4).

<table>
<thead>
<tr>
<th>Model</th>
<th>Solver</th>
<th>Data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>Algorithm 2</td>
<td>a9a 0.3238 webspam 0.2085 news20 0.1194 kddb 0.2852 url 0.0315 rcv1 0.0740</td>
</tr>
<tr>
<td>FM: formulation (2)</td>
<td>SG</td>
<td>a9a 0.3200 webspam 0.1151 news20 0.1019 kddb 0.2652 url 0.0144 rcv1 0.0623</td>
</tr>
<tr>
<td>FM: formulation (4)</td>
<td>SG</td>
<td>a9a 0.3203 webspam 0.0904 news20 0.0884 kddb 0.2742 url 0.0156 rcv1 0.0595</td>
</tr>
<tr>
<td>ANT-P</td>
<td></td>
<td>a9a 0.3206 webspam 0.0583 news20 0.0862 kddb 0.2755 url 0.0155 rcv1 0.0584</td>
</tr>
</tbody>
</table>

Table III: Test accuracies of logistic regression (LR) and the two FM formulations (2) and (4).

<table>
<thead>
<tr>
<th>Model</th>
<th>Solver</th>
<th>Data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>Algorithm 2</td>
<td>a9a 85.03% webspam 92.63% news20 96.43% kddb 88.87% url 98.87% rcv1 97.59%</td>
</tr>
<tr>
<td>FM: formulation (2)</td>
<td>SG</td>
<td>a9a 85.26% webspam 95.93% news20 95.98% kddb 89.43% url 99.53% rcv1 98.05%</td>
</tr>
<tr>
<td>FM: formulation (4)</td>
<td>SG</td>
<td>a9a 85.16% webspam 97.00% news20 96.38% kddb 89.29% url 99.51% rcv1 98.03%</td>
</tr>
<tr>
<td>ANT-P</td>
<td></td>
<td>a9a 85.12% webspam 98.34% news20 96.91% kddb 89.26% url 99.50% rcv1 98.10%</td>
</tr>
</tbody>
</table>

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 regression. 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. 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


ACM Transactions on Intelligent Systems and Technology, Vol. 0, No. 0, Article 0, Publication date: 0.
An Efficient Alternating Newton Method for Learning Factorization Machines


Received; revised; accepted