Convex Factorization Machines

Mathieu Blondel, Akinori Fujino, and Naonori Ueda

NTT Communication Science Laboratories, Kyoto, Japan

Abstract. Factorization machines are a generic framework which allows to mimic many factorization models simply by feature engineering. In this way, they combine the high predictive accuracy of factorization models with the flexibility of feature engineering. Unfortunately, factorization machines involve a non-convex optimization problem and are thus subject to bad local minima. In this paper, we propose a convex formulation of factorization machines based on the nuclear norm. Our formulation imposes fewer restrictions on the learned model and is thus more general than the original formulation. To solve the corresponding optimization problem, we present an efficient globally-convergent two-block coordinate descent algorithm. Empirically, we demonstrate that our approach achieves comparable or better predictive accuracy than the original factorization machines on 4 recommendation tasks and scales to datasets with 10 million samples.

Keywords: factorization machines, feature interactions, recommender systems, nuclear norm

1 Introduction

Factorization machines [12] [13] are a generic framework which allows to mimic many factorization models simply by feature engineering. Similarly to linear models, factorization machines learn a feature weight vector $\boldsymbol{w} \in \mathbb{R}^d$, where d is the number of features. However, factorization machines also learn a pairwise feature interaction weight matrix $\boldsymbol{Z} \in \mathbb{R}^{d \times d}$. Given a feature vector $\boldsymbol{x} \in \mathbb{R}^d$, factorization machines use $\boldsymbol{w}$ and $\boldsymbol{Z}$ to predict a target $y \in \mathbb{R}$. The main advantage of factorization machines is that they learn the feature interaction weight matrix in factorized form, $\boldsymbol{Z} = \boldsymbol{V}\boldsymbol{V}^{\mathrm{T}}$, where $\boldsymbol{V} \in \mathbb{R}^{d \times k}$ and $k \ll d$ is a rank hyper-parameter. This reduces overfitting, since the number of parameters to estimate is reduced from d^2 to kd, and allows to compute predictions efficiently. Although they can be used for any supervised learning task such as classification and regression, factorization machines are especially useful for recommender systems. As shown in [12][13], factorization machines can mimic many existing factorization models just by choosing an appropriate feature representation for $\boldsymbol{x}$. Examples include standard matrix factorization, SVD++ [8], timeSVD++[9] and PITF (pairwise interaction tensor factorization) [16]. Moreover, it is easy to incorporate auxiliary features such as user and item attributes, contextual information [15] and cross-domain feedback [10]. In [14], it was shown that factorization machines achieve predictive accuracy as good as the best specialized models on the Netflix and KDDcup 2012 challenges. In short, factorization machines are a generic framework which combines the high predictive accuracy of factorization models with the flexibility of feature engineering. Unfortunately, factorization machines have two main drawbacks. First, they involve a non-convex optimization problem. Thus, we can typically only obtain a local solution, the quality of which depends on initialization. Second, factorization machines require the choice of a rank hyper-parameter. In practice, predictive accuracy can be quite sensitive to this choice.

In this paper, we propose a convex formulation of factorization machines based on the nuclear norm. Our formulation is more general than the original one in the sense that it imposes fewer restrictions on the feature interaction weight matrix $\boldsymbol{Z}$. For example, in our formulation, imposing positive semi-definiteness is possible but not necessary. In addition, our formulation does not require choosing any rank hyper-parameter and thus have one less hyper-parameter than the original formulation. For solving the corresponding optimization problem, we propose a globally-convergent two-block coordinate descent algorithm. Our algorithm alternates between estimating the feature weight vector $\boldsymbol{w}$ and a low-rank feature interaction weight matrix $\boldsymbol{Z}$. Estimating $\boldsymbol{w}$ is easy, since the problem reduces to a simple linear model objective. However, estimating $\boldsymbol{Z}$ is challenging, due to the quadratic number of feature interactions. Following a recent line of work [17] [4] [7], we derive a greedy coordinate descent algorithm which breaks down the large problem into smaller sub-problems. By exploiting structure, we can solve these sub-problems efficiently. Furthermore, our algorithm maintains an eigendecomposition of $\boldsymbol{Z}$. Therefore, the entire matrix $\boldsymbol{Z}$ is never materialized and our algorithm can scale to very high-dimensional data. Empirically, we demonstrate that our approach achieves comparable or better predictive accuracy than the original non-convex factorization machines on 4 recommendation tasks and scales to datasets with 10 million samples.

Notation. For arbitrary real matrices, the inner product is defined as $\langle \boldsymbol{A}, \boldsymbol{B} \rangle \coloneqq \mathrm{Tr}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B})$ and the squared Frobenius matrix norm as $\|\boldsymbol{A}\|_F^2 \coloneqq \langle \boldsymbol{A}, \boldsymbol{A} \rangle$. We denote the element-wise product between two vectors $\boldsymbol{a} \in \mathbb{R}^d$ and $\boldsymbol{b} \in \mathbb{R}^d$ by $\boldsymbol{a} \circ \boldsymbol{b} \coloneqq [a_1 b_1, \ldots, a_d b_d]^{\mathrm{T}}$. We denote the Kronecker product between two matrices $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{B} \in \mathbb{R}^{p \times q}$ by $\boldsymbol{A} \otimes \boldsymbol{B} \in \mathbb{R}^{mp \times nq}$. We denote the set of symmetric $d \times d$ matrices by $\mathbb{S}^{d \times d}$. Given $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\mathrm{vec}(\boldsymbol{A}) \in \mathbb{R}^{mn}$ denotes the vector obtained by stacking the columns of $\boldsymbol{A}$. By $[n]$, we denote the set $\{1, \ldots, n\}$. The support of a vector $\boldsymbol{\lambda} \in \mathbb{R}^d$ is defined as $\mathrm{supp}(\boldsymbol{\lambda}) \coloneqq \{j \in [d] : \lambda_j \neq 0\}$.

2 Factorization machines

Factorization machines [12][13] predict the output associated with an input $\boldsymbol{x} = [x_1, \ldots, x_d]^{\mathrm{T}} \in \mathbb{R}^d$ using the following simple equation:

$$\tilde{y}(\boldsymbol{x}|\boldsymbol{w}, \boldsymbol{V}) \coloneqq \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \sum_{j=1}^{d} \sum_{j'=j+1}^{d} (\boldsymbol{V}\boldsymbol{V}^{\mathrm{T}})_{jj'} x_j x_{j'} \tag{1}$$

where $\boldsymbol{w} \in \mathbb{R}^d$, $\boldsymbol{V} \in \mathbb{R}^{d \times k}$ and $k \ll d$ is a hyper-parameter which defines the rank of the factorization. The vector $\boldsymbol{w}$ contains the weights of individual features for predicting y, while the positive semi-definite matrix $\boldsymbol{Z} = \boldsymbol{V}\boldsymbol{V}^{\mathrm{T}} \in \mathbb{S}^{d \times d}$ contains the weights of pairwise feature interactions. Because factorization machines learn $\boldsymbol{Z}$ in factorized form, the number of parameters to estimate is reduced from d^2 to kd. In addition to helping reduce overfitting, this factorization allows to compute predictions efficiently by using

$$\tilde{y}(\boldsymbol{x}|\boldsymbol{w}, \boldsymbol{V}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \frac{1}{2}\Big(\|\boldsymbol{V}^{\mathrm{T}}\boldsymbol{x}\|^2 - \sum_{s=1}^{k} \|\boldsymbol{v}_s \circ \boldsymbol{x}\|^2\Big),$$

where $\boldsymbol{v}_s \in \mathbb{R}^d$ is the s^{th} column of $\boldsymbol{V}$. Thus, computing predictions costs $O(kd)$, instead of $O(d^2)$ when implemented naively. For sparse $\boldsymbol{x}$, the prediction cost reduces to $O(kN_z(\boldsymbol{x}))$, where $N_z(\boldsymbol{x})$ is the number of non-zero features in $\boldsymbol{x}$.

Although they can be used for any supervised learning task such as classification and regression, factorization machines are especially useful for recommender systems. As shown in [12][13], factorization machines can mimic many existing factorization models just by choosing an appropriate feature representation for $\boldsymbol{x}$. For example, consider a record (u, i, y), where $u \in U$ is a user index, $i \in I$ is an item index and $y \in \mathbb{R}$ is a rating given by u to i. Then factorization machines are exactly equivalent to matrix factorization (c.f., Section A in the supplementary material) simply by converting (u, i, y) to $(\boldsymbol{x}, y)$, where $\boldsymbol{x} \in \mathbb{R}^d$ is expressed in the following binary indicator representation with $d = |U| + |I|$

$$\boldsymbol{x} := [\underbrace{0, \dots, 0, \overbrace{\ 1\ }^{u}, 0, \dots, 0}_{|U|}, \underbrace{0, \dots, 0, \overbrace{\ 1\ }^{|U|+i}, 0, \dots, 0}_{|I|}]^{\mathrm{T}}. \tag{2}$$

Using more elaborated feature representations [12] [13], it is possible to mimic many other factorization models, including SVD++ [8], timeSVD++[9] and PITF (pairwise interaction tensor factorization) [16]. Moreover, it is easy to incorporate auxiliary features such as user and item attributes, contextual information [15] and cross-domain feedback [10]. The ability to quickly try many different features ("feature engineering") is very flexible from a practitioner perspective. In addition, since factorization machines behave much like classifiers or regressors, they are easy to integrate in a consistent manner to a machine learning library (see [3] for a discussion on the merits of library design consistency).

Given a training set consisting of n feature vectors $\boldsymbol{X} = [\boldsymbol{x}_1, \dots, \boldsymbol{x}_n]^{\mathrm{T}} \in \mathbb{R}^{n \times d}$ and corresponding targets $[y_1, \dots, y_n]^{\mathrm{T}} \in \mathbb{R}^n$, we can estimate $\boldsymbol{w} \in \mathbb{R}^d$ and $\boldsymbol{V} \in \mathbb{R}^{d \times k}$ using the principle of empirical risk minimization. For example, we can solve the following optimization problem

$$\min_{\boldsymbol{w} \in \mathbb{R}^d, \boldsymbol{V} \in \mathbb{R}^{d \times k}} \sum_{i=1}^{n} \ell\Big(y_i, \tilde{y}(\boldsymbol{x}_i|\boldsymbol{w}, \boldsymbol{V})\Big) + \frac{\alpha}{2}\|\boldsymbol{w}\|^2 + \frac{\beta}{2}\|\boldsymbol{V}\|_F^2, \tag{3}$$

where $\ell(y, \tilde{y})$ is the loss "suffered" when predicting $\tilde{y}$ instead of y. Throughout this paper, we assume ℓ is a twice-differentiable convex function. For instance, for predicting continuous outputs, we can use the squared loss $\ell(y, \tilde{y}) = \frac{1}{2}(\tilde{y} - y)^2$. $\alpha > 0$ and $\beta > 0$ are hyper-parameters which control the trade-off between low loss and low model complexity. In practice, (3) can be solved using the stochastic gradient or coordinate descent methods. Both methods have a runtime complexity of $O(kN_z(\boldsymbol{X}))$ per epoch [13], where $N_z(\boldsymbol{X})$ is the total number of non-zero elements in $\boldsymbol{X}$. Assuming (2) is used, this is the same runtime complexity as for standard matrix factorization. We now state some important properties of the optimization problem (3), which were not mentioned in [12] and [13].

Proposition 1. *The optimization problem* (3) *is i) convex in $\boldsymbol{w}$, ii) non-convex in $\boldsymbol{V}$ and iii) convex in v_{js} (elements of $\boldsymbol{V}$ taken separately). If we replace $\sum_{j'=j+1}^{d}(\boldsymbol{V}\boldsymbol{V}^{\mathrm{T}})_{jj'}x_j x_{j'}$ by $\sum_{j'=j}^{d}(\boldsymbol{V}\boldsymbol{V}^{\mathrm{T}})_{jj'}x_j x_{j'}$ in* (1)*, i.e., if we use diagonal elements of $\boldsymbol{V}\boldsymbol{V}^{\mathrm{T}}$, then* (3) *is iv) non-convex in both $\boldsymbol{V}$ and v_{js}.*

Property ii) means that the stochastic gradient and coordinate descent methods are only guaranteed to reach a local minimum, the quality of which typically depends on the initialization of $\boldsymbol{V}$. Property iii) explains why coordinate descent is a good method for solving (3): it can monotonically decrease the objective (3) until it reaches a local minimum. Property iv) shows that if we use diagonal elements of $\boldsymbol{V}\boldsymbol{V}^{\mathrm{T}}$, (3) becomes a much more challenging optimization problem, possibly subject to more bad local minima. In contrast, our formulation is convex whether or not we use diagonal elements.

3 Convex formulation

We begin by rewriting the prediction equation (1) as

$$\hat{y}(\boldsymbol{x}|\boldsymbol{w}, \boldsymbol{Z}) \coloneqq \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \sum_{j=1}^{d}\sum_{j'=1}^{d} z_{jj'}x_j x_{j'} = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \langle \boldsymbol{Z}, \boldsymbol{x}\boldsymbol{x}^{\mathrm{T}} \rangle,$$

where $z_{jj'}$ denote the entries of the symmetric matrix $\boldsymbol{Z} \in \mathbb{S}^{d \times d}$. Clearly, we need to impose some structure on $\boldsymbol{Z}$ to avoid its $O(d^2)$ memory complexity. We choose to learn a low-rank matrix $\boldsymbol{Z}$, i.e., $\operatorname{rank}(\boldsymbol{Z}) \ll d$. Following recent advances in convex optimization, we can achieve this by regularizing $\boldsymbol{Z}$ with the nuclear norm (a.k.a. trace norm), which is known to be the tightest convex lower bound on matrix rank [11]. Given a symmetric matrix $\boldsymbol{Z} \in \mathbb{S}^{d \times d}$, the nuclear norm is defined as (c.f. supplementary material Section C)

$$\|\boldsymbol{Z}\|_* \coloneqq \operatorname{Tr}\left(\sqrt{\boldsymbol{Z}^2}\right) = \|\boldsymbol{\lambda}\|_1, \tag{4}$$

where $\boldsymbol{\lambda}$ is a vector which gathers the eigenvalues of $\boldsymbol{Z}$. We see that regularizing $\boldsymbol{Z}$ with the nuclear norm is equivalent to regularizing its eigenvalues with the ℓ_1 norm, which is known to promote sparsity. Since $\operatorname{rank}(\boldsymbol{Z}) = \|\boldsymbol{\lambda}\|_0 = |\operatorname{supp}(\boldsymbol{\lambda})|$, the nuclear norm thus promotes low-rank solutions. We therefore propose to learn factorization machines by solving the following optimization problem

$$\min_{\boldsymbol{w} \in \mathbb{R}^d, \boldsymbol{Z} \in \mathbb{S}^{d \times d}} \sum_{i=1}^{n} \ell\Big(y_i, \hat{y}(\boldsymbol{x}_i|\boldsymbol{w}, \boldsymbol{Z})\Big) + \frac{\alpha}{2}\|\boldsymbol{w}\|^2 + \beta\|\boldsymbol{Z}\|_*, \tag{5}$$

where, again, ℓ is a twice-differentiable convex loss function and $\alpha > 0$ and $\beta > 0$ are hyper-parameters. Problem (5) is jointly convex in $\boldsymbol{w}$ and $\boldsymbol{Z}$. In our formulation, there is no rank hyper-parameter (such as k for $\boldsymbol{V}$). Instead, the rank of $\boldsymbol{Z}$ is indirectly controlled by β (the larger β, the lower $\operatorname{rank}(\boldsymbol{Z})$).

Convexity is an important property, since it allows us to derive an efficient algorithm for finding a global solution (i.e., our algorithm is insensitive to initialization). In addition, our convex formulation is more general than the original one in the sense that imposing positive semi-definiteness of $\boldsymbol{Z}$ or ignoring diagonal elements of $\boldsymbol{Z}$ is not necessary (although it is possible, c.f., Section D and Section E in the supplementary material).

Any symmetric matrix $\boldsymbol{Z} \in \mathbb{S}^{d \times d}$ can be written as an eigendecomposition $\boldsymbol{Z} = \boldsymbol{P\Lambda P}^{\mathrm{T}} = \sum_s \lambda_s \boldsymbol{p}_s \boldsymbol{p}_s^{\mathrm{T}}$, where $\boldsymbol{P}$ is an orthogonal matrix with columns $\boldsymbol{p}_s \in \mathbb{R}^d$ and $\boldsymbol{\Lambda} = \operatorname{diag}(\boldsymbol{\lambda})$ is a diagonal matrix with diagonal entries λ_s. Using this decomposition, we can compute predictions efficiently by

$$\hat{y}(\boldsymbol{x}|\boldsymbol{w}, \boldsymbol{P\Lambda P}^{\mathrm{T}}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \langle \boldsymbol{P\Lambda P}^{\mathrm{T}}, \boldsymbol{x}\boldsymbol{x}^{\mathrm{T}} \rangle = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \sum_{s=1}^{k} \lambda_s (\boldsymbol{p}_s^{\mathrm{T}}\boldsymbol{x})^2, \qquad (6)$$

where $k = \operatorname{rank}(\boldsymbol{Z})$. Thus, prediction cost is the same as non-convex factorization machines, i.e., $O(kN_z(\boldsymbol{x}))$. The algorithm we present in Section 4 always maintains such a decomposition. Therefore, $\boldsymbol{Z}$ is never materialized in memory and we can scale to high-dimensional data. Equation (6) also suggests an interesting interpretation of convex factorization machines. Let $\kappa(\boldsymbol{p}, \boldsymbol{x}) = (\boldsymbol{p}^{\mathrm{T}}\boldsymbol{x})^2$, i.e., κ is a homogeneous polynomial kernel of degree 2. Then, (6) can be written as $\hat{y}(\boldsymbol{x}|\boldsymbol{w}, \boldsymbol{P\Lambda P}^{\mathrm{T}}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \sum_{s=1}^{k} \lambda_s \kappa(\boldsymbol{p}_s, \boldsymbol{x})$. Thus, convex factorization machines evaluate the homogeneous polynomial kernel between orthonormal *basis vectors* $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_k$ and $\boldsymbol{x}$. In contrast, kernel ridge regression and other kernel machines compute predictions using $\sum_{i=1}^{n} a_i \kappa(\boldsymbol{x}_i, \boldsymbol{x})$, i.e., the kernel is evaluated between *training instances* and $\boldsymbol{x}$. Thus, the main advantage of convex factorization machines over traditional kernel machines is that the basis vectors are actually *learned* from data.

4 Optimization algorithm

To solve (5), we propose a two-block coordinate descent algorithm. That is, we alternate between minimizing with respect to $\boldsymbol{w}$ and $\boldsymbol{Z}$ until convergence. When the algorithm terminates, it returns $\boldsymbol{w}$ and $\boldsymbol{Z} = \boldsymbol{P\Lambda P}^{\mathrm{T}}$.

4.1 Minimizing with respect to $\boldsymbol{w}$

For minimizing (5) with respect to $\boldsymbol{w}$, we need to solve

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \sum_{i=1}^{n} \ell(y_i, \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + \pi_i) + \frac{\alpha}{2}\|\boldsymbol{w}\|^2, \qquad (7)$$

where $\pi_i = \langle \boldsymbol{Z}, \boldsymbol{x}_i\boldsymbol{x}_i^{\mathrm{T}} \rangle$. This is a standard linear model objective, except that the predictions are shifted by π_i. Thus, we can solve (7) using standard methods.

Algorithm 1 Minimizing (8) w.r.t. $\boldsymbol{Z}$

```
Input: $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, initial $\boldsymbol{Z} = \boldsymbol{P}\,\mathrm{diag}(\boldsymbol{\lambda})\boldsymbol{P}^{\mathrm{T}}$, $\beta > 0$
$\boldsymbol{Z}_{\boldsymbol{\lambda}} := \sum_{s\in\mathrm{supp}(\boldsymbol{\lambda})} \lambda_s \boldsymbol{p}_s \boldsymbol{p}_s^{\mathrm{T}}$
repeat
    Compute $\boldsymbol{p}$ = dominant eigenvector of $\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})$
    Find $\lambda = \mathrm{argmin}_{\lambda\in\mathbb{R}}\, L\Big(\boldsymbol{Z}_{\boldsymbol{\lambda}} + \lambda \boldsymbol{p}\boldsymbol{p}^{\mathrm{T}}\Big) + \beta|\lambda|$
    $\boldsymbol{P} \leftarrow [\boldsymbol{P}\ \boldsymbol{p}] \quad \boldsymbol{\lambda} \leftarrow [\boldsymbol{\lambda}\ \lambda]$
    Diagonal refitting case
    $\bar{\boldsymbol{\lambda}} \leftarrow \boldsymbol{\lambda}$
    $\boldsymbol{\lambda} \leftarrow \mathrm{argmin}_{\boldsymbol{\lambda}\in\mathbb{R}^{\mathrm{supp}(\bar{\boldsymbol{\lambda}})}}\, \tilde{L}(\boldsymbol{\lambda}) + \beta\|\boldsymbol{\lambda}\|_1 = \mathrm{argmin}_{\boldsymbol{\lambda}\in\mathbb{R}^{\mathrm{supp}(\bar{\boldsymbol{\lambda}})}}\, L(\boldsymbol{Z}_{\boldsymbol{\lambda}}) + \beta\|\boldsymbol{\lambda}\|_1$
    Fully-corrective refitting case
    Orthonormalize columns of $\boldsymbol{P}$
    $\boldsymbol{A} = \mathrm{argmin}_{\boldsymbol{A}\in\mathbb{S}^{k\times k}}\, L(\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}) + \beta\|\boldsymbol{A}\|_*$ where $k = \mathrm{rank}(\boldsymbol{Z}_{\boldsymbol{\lambda}})$
    $\boldsymbol{P} \leftarrow \boldsymbol{P}\boldsymbol{Q} \quad \boldsymbol{\lambda} \leftarrow \mathrm{diag}(\boldsymbol{\Sigma})$   where   $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Sigma}\boldsymbol{Q}^{\mathrm{T}}$
until convergence
Output: $\boldsymbol{Z} = \boldsymbol{P}\,\mathrm{diag}(\boldsymbol{\lambda})\boldsymbol{P}^{\mathrm{T}}$
```

4.2 Minimizing with respect to $\boldsymbol{Z}$

For minimizing (5) with respect to $\boldsymbol{Z}$, we need to solve

$$\min_{\boldsymbol{Z}\in\mathbb{S}^{d\times d}} \underbrace{\sum_{i=1}^{n} \ell\Big(y_i, \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + \langle \boldsymbol{Z}, \boldsymbol{x}_i\boldsymbol{x}_i^{\mathrm{T}}\rangle\Big)}_{:=L(\boldsymbol{Z})} + \beta\|\boldsymbol{Z}\|_*. \tag{8}$$

Two standard methods for solving nuclear norm regularized problems are proximal gradient and ADMM. For these methods, the key operation is the proximal operator, which requires an SVD and is thus a bottleneck in scaling to large matrix sizes. In order to address this issue, we adapt greedy coordinate descent algorithms [4] [7] designed for general nuclear norm regularized minimization. The main difference of our algorithm is that we learn an eigendecomposition of $\boldsymbol{Z}$ rather than an SVD, in order to take advantage of the symmetry of $\boldsymbol{Z}$.

Outline. To minimize (8), on each iteration we greedily find the rank-one matrix $\boldsymbol{p}\boldsymbol{p}^{\mathrm{T}}$ that most violates the optimality conditions and add it to $\boldsymbol{Z}$ by $\boldsymbol{Z} \leftarrow \boldsymbol{Z} + \lambda\boldsymbol{p}\boldsymbol{p}^{\mathrm{T}}$, where λ is the optimal weight. Thus, the rank of $\boldsymbol{Z}$ increases by at most 1 on each iteration. In practice, however, we never materialize $\boldsymbol{Z}$ and maintain its eigendecomposition $\boldsymbol{Z} = \boldsymbol{P}\,\mathrm{diag}(\boldsymbol{\lambda})\boldsymbol{P}^{\mathrm{T}}$ instead. To ensure convergence, we refit the eigendecomposition of $\boldsymbol{Z}$ on each iteration using one of two methods: diagonal refitting (update $\boldsymbol{\lambda}$ only) or fully corrective refitting (update both $\boldsymbol{\lambda}$ and $\boldsymbol{P}$). The entire procedure is summarized in Algorithm 1.

Finding λ and $\boldsymbol{p}$. Using (4) and (6), we obtain that (8) is equivalent to

$$\min_{\boldsymbol{\lambda}\in\Theta} \underbrace{\sum_{i=1}^{n} \ell\Big(y_i, \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + \sum_{s\in\mathcal{S}} \lambda_s(\boldsymbol{p}_s^{\mathrm{T}}\boldsymbol{x}_i)^2\Big)}_{:=\tilde{L}(\boldsymbol{\lambda})} + \beta\|\boldsymbol{\lambda}\|_1, \tag{9}$$

where $\mathcal{S}$ is an index set for the elements of the set $\{\boldsymbol{p}\boldsymbol{p}^{\mathrm{T}} : \boldsymbol{p} \in \mathbb{R}^d, \|\boldsymbol{p}\| = 1\}$ and $\Theta \coloneqq \{\boldsymbol{\lambda} \in \mathbb{R}^{\mathcal{S}} : \operatorname{supp}(\boldsymbol{\lambda}) \text{ is finite}\}$. Thus, we converted a problem with respect to $\boldsymbol{Z}$ in the space of symmetric matrices to a problem with respect to $\boldsymbol{\lambda}$ in the space of (normalized) rank-one matrices. This space can be arbitrarily large. However, the number of non-zero elements in $\boldsymbol{\lambda}$ is at most d. Moreover, $\boldsymbol{\lambda}$ will be typically sparse thanks to the regularization term $\beta\|\boldsymbol{\lambda}\|_1$, i.e., $|\operatorname{supp}(\boldsymbol{\lambda})| = \operatorname{rank}(\boldsymbol{Z}) \ll d$. A difference between (9) and past works [17] [4] is that we do not constrain $\boldsymbol{\lambda}$ to be non-negative, since eigenvalues can be negative, unlike singular values. Constraining $\boldsymbol{\lambda}$ to be non-negative corresponds to a positive semi-definite constraint on $\boldsymbol{Z}$, which we cover in Section D of the supplementary material.

According to the Karush-Kuhn-Tucker (KKT) conditions, for any $s \in \mathcal{S}$, the optimality violation of λ_s at $\boldsymbol{\lambda}$ is given by

$$\nu_s = \begin{cases} |\nabla_s \tilde{L}(\boldsymbol{\lambda}) + \beta|, & \text{if } \lambda_s > 0 \\ |\nabla_s \tilde{L}(\boldsymbol{\lambda}) - \beta|, & \text{if } \lambda_s < 0 \\ \max\Big(|\nabla_s \tilde{L}(\boldsymbol{\lambda})| - \beta, 0\Big), & \text{if } \lambda_s = 0, \end{cases}$$

where $\nabla_s \tilde{L}(\boldsymbol{\lambda}) = \frac{\partial \tilde{L}(\boldsymbol{\lambda})}{\partial \lambda_s}$. Using the chain rule, we obtain

$$\nabla_s \tilde{L}(\boldsymbol{\lambda}) = \langle \nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}}), \boldsymbol{p}_s \boldsymbol{p}_s^{\mathrm{T}} \rangle = \boldsymbol{p}_s^{\mathrm{T}} \nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}}) \boldsymbol{p}_s,$$

where $\boldsymbol{Z}_{\boldsymbol{\lambda}} \coloneqq \sum_{s \in \operatorname{supp}(\boldsymbol{\lambda})} \lambda_s \boldsymbol{p}_s \boldsymbol{p}_s^{\mathrm{T}}$ and $\nabla L(\boldsymbol{Z}) \in \mathbb{S}^{d \times d}$ is the gradient of L at $\boldsymbol{Z}$. Intuitively, we would like to find the eigenvector $\boldsymbol{p}_s$ which maximizes ν_s:

$$\operatorname*{argmax}_{s \notin \operatorname{supp}(\boldsymbol{\lambda})} \nu_s = \operatorname*{argmax}_{s \in \mathcal{S}} |\nabla_s \tilde{L}(\boldsymbol{\lambda})| = \operatorname*{argmax}_{s \in \mathcal{S}} |\boldsymbol{p}_s^{\mathrm{T}} \nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}}) \boldsymbol{p}_s|$$

Thus, $\boldsymbol{p}_s$ corresponds to the dominant eigenvector of $\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})$ (eigenvector corresponding to the greatest eigenvalue in absolute value). We can find $\boldsymbol{p}_s$ efficiently using the power iteration method. Since $\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})$ is a $d \times d$ matrix, we cannot afford to store it in memory when d is large. Fortunately, the power iteration method only accesses $\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})$ through matrix-vector products $\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})\boldsymbol{p}$ for some vector $\boldsymbol{p} \in \mathbb{R}^d$. By exploiting the structure of $\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})$, we can compute this product efficiently (c.f., Section 4.3 for the squared loss).

Let $\bar{\boldsymbol{\lambda}}$ be the current iterate of $\boldsymbol{\lambda}$. Once we found $\boldsymbol{p}_s$, we can find λ_s by

$$\lambda_s = \operatorname*{argmin}_{\lambda \in \mathbb{R}} \tilde{L}\Big(\bar{\boldsymbol{\lambda}} + (\lambda - \bar{\lambda}_s)\boldsymbol{e}_s\Big) + \beta|\lambda| = \operatorname*{argmin}_{\lambda \in \mathbb{R}} L\Big(\boldsymbol{Z}_{\bar{\boldsymbol{\lambda}}} + (\lambda - \bar{\lambda}_s)\boldsymbol{p}_s \boldsymbol{p}_s^{\mathrm{T}}\Big) + \beta|\lambda|, \tag{10}$$

where $\boldsymbol{e}_s = [\underbrace{0, \ldots, 0}_{s-1}, 1, 0, \ldots, 0]^{\mathrm{T}}$. For the squared loss, this problem can be solved in closed form (c.f., Section 4.3). For other loss functions, we can solve the problem iteratively.

Diagonal refitting. Similarly to [4], we can refit $\boldsymbol{\lambda}$ restricted to its current support. Let $\bar{\boldsymbol{\lambda}}$ be the current iterate of $\boldsymbol{\lambda}$. Then, we solve

$$\min_{\boldsymbol{\lambda} \in \mathbb{R}^{\operatorname{supp}(\bar{\boldsymbol{\lambda}})}} \tilde{L}(\boldsymbol{\lambda}) + \beta\|\boldsymbol{\lambda}\|_1.$$

This can easily be solved by iteratively using (10) for all $s \in \operatorname{supp}(\bar{\boldsymbol{\lambda}})$ until the sum of violations $\sum_{s \in \operatorname{supp}(\bar{\boldsymbol{\lambda}})} \nu_s$ converges. We call this method "diagonal refitting", since the matrix $\boldsymbol{\Lambda} = \operatorname{diag}(\boldsymbol{\lambda})$ in $\boldsymbol{Z} = \boldsymbol{P \Lambda P}^{\mathrm{T}}$ is diagonal.

Fully-corrective refitting. Any matrix $\boldsymbol{Z} \in \mathbb{S}^{d \times d}$ can be written as $\boldsymbol{PAP}^{\mathrm{T}}$, where $\boldsymbol{P} \in \mathbb{R}^{d \times k}$, $\boldsymbol{A} \in \mathbb{S}^{k \times k}$ ($\boldsymbol{A}$ not necessarily diagonal) and $k = \operatorname{rank}(\boldsymbol{Z})$. Following a similar idea to [17] and [7], injecting $\boldsymbol{Z} = \boldsymbol{PAP}^{\mathrm{T}}$ in (8), we can solve

$$\min_{\boldsymbol{A} \in \mathbb{S}^{k \times k}} L(\boldsymbol{PAP}^{\mathrm{T}}) + \beta \|\boldsymbol{A}\|_*, \tag{11}$$

where we used $\|\boldsymbol{PAP}^{\mathrm{T}}\|_* = \|\boldsymbol{A}\|_*$ if $\boldsymbol{P}$ orthonormal. This problem is similar to (8); only this time, it is $k \times k$ dimensional instead of $d \times d$ dimensional. Once we obtained $\boldsymbol{A}$, we can update $\boldsymbol{P}$ and $\boldsymbol{\lambda}$ by $\boldsymbol{P} \leftarrow \boldsymbol{PQ}$ and $\boldsymbol{\lambda} \leftarrow \operatorname{diag}(\boldsymbol{\Sigma})$, where $\boldsymbol{Q \Sigma Q}^{\mathrm{T}}$ is an eigendecomposition of $\boldsymbol{A}$ (cheap to compute since $\boldsymbol{A}$ is $k \times k$).

We propose to solve (11) by the alternating direction method of multipliers (ADMM). To do so, we consider the following augmented Lagrangian

$$\min_{\boldsymbol{A} \in \mathbb{S}^{k \times k}, \boldsymbol{B} \in \mathbb{S}^{k \times k}} L(\boldsymbol{PAP}^{\mathrm{T}}) + \beta \|\boldsymbol{B}\|_* \text{ s.t. } \boldsymbol{A} - \boldsymbol{B} = 0. \tag{12}$$

ADMM solves (12) using the following iterative procedure:

$$\boldsymbol{A}^{\tau+1} = \operatorname*{argmin}_{\boldsymbol{A} \in \mathbb{S}^{k \times k}} \underbrace{L(\boldsymbol{PAP}^{\mathrm{T}}) + \frac{\rho}{2} \|\boldsymbol{A} - \boldsymbol{B}^{\tau} + \boldsymbol{M}^{\tau}\|^2}_{:= \hat{L}(\boldsymbol{A})} \tag{13}$$

$$\boldsymbol{B}^{\tau+1} = S_{\beta/\rho}\Big(\boldsymbol{A}^{\tau+1} + \boldsymbol{M}^{\tau}\Big) \tag{14}$$

$$\boldsymbol{M}^{\tau+1} = \boldsymbol{M}^{\tau} + \boldsymbol{A}^{\tau+1} - \boldsymbol{B}^{\tau+1},$$

where ρ is a parameter and S_c is the proximal operator (here, shrinkage operator). In practice, a common choice is $\rho = 1$. The procedure converges when $\|\boldsymbol{A}^{\tau} - \boldsymbol{B}^{\tau}\|_F^2 \leq \epsilon$. We now explain how to solve (14) and (13).

Given an eigendecomposition $\boldsymbol{A} = \boldsymbol{Q \Sigma Q}^{\mathrm{T}}$, where $\boldsymbol{\Sigma} = \operatorname{diag}(\sigma_1, \ldots, \sigma_k)$, the shrinkage operator is defined as

$$S_c(\boldsymbol{A}) = \operatorname*{argmin}_{\boldsymbol{B}} \frac{1}{2} \|\boldsymbol{A} - \boldsymbol{B}\|_F^2 + c \|\boldsymbol{B}\|_* = \boldsymbol{Q} \operatorname{diag}(\hat{\sigma}_1, \ldots, \hat{\sigma}_k) \boldsymbol{Q}^{\mathrm{T}}, \tag{15}$$

where $\hat{\sigma}_s = \operatorname{sign}(\sigma_s) \max(|\sigma_s| - c, 0)$. In other words, we apply the soft-thresholding operator to the eigenvalues of $\boldsymbol{A}$. For solving the sub-problem (13), we can afford to use the Newton method, since $k \ll d$. Let $\nabla \hat{L}(\boldsymbol{A}) \in \mathbb{S}^{k \times k}$ and $\nabla^2 \hat{L}(\boldsymbol{A}) \in \mathbb{S}^{k^2 \times k^2}$ be the gradient and Hessian of $\hat{L}$ at $\boldsymbol{A}$. On each iteration, the Newton method updates $\boldsymbol{A}$ by

$$\boldsymbol{A} \leftarrow \boldsymbol{A} - \gamma \boldsymbol{D}$$

where $\boldsymbol{D} \in \mathbb{R}^{k \times k}$ is the solution of the system of linear equations

$$\nabla^2 \hat{L}(\boldsymbol{A}) \operatorname{vec}(\boldsymbol{D}) = \operatorname{vec}\Big(\nabla \hat{L}(\boldsymbol{A})\Big) \tag{16}$$

and γ is adjusted by line search (typically, using the Wolfe conditions). Using the chain rule, we can compute $\nabla \hat{L}(\boldsymbol{A})$ and $\nabla^2 \hat{L}(\boldsymbol{A})$ by

$$\nabla \hat{L}(\boldsymbol{A}) = \boldsymbol{P}^{\mathrm{T}} \Big(\nabla L(\boldsymbol{Z})|_{\boldsymbol{Z}=\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}} \Big) \boldsymbol{P} + \rho(\boldsymbol{A} - \boldsymbol{B}^{\tau} + \boldsymbol{M}^{\tau})$$
$$\nabla^2 \hat{L}(\boldsymbol{A}) = \boldsymbol{P}^{\mathrm{T}} \otimes \boldsymbol{P}^{\mathrm{T}} \Big(\nabla^2 L(\boldsymbol{Z})|_{\boldsymbol{Z}=\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}} \Big) \boldsymbol{P} \otimes \boldsymbol{P} + \rho \boldsymbol{I}$$

To compute $\nabla L(\boldsymbol{Z})|_{\boldsymbol{Z}=\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}}$ and $\nabla^2 L(\boldsymbol{Z})|_{\boldsymbol{Z}=\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}}$, we need to compute the predictions at $\boldsymbol{Z} = \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}$. This can be done efficiently by $\hat{y}(\boldsymbol{x}|\boldsymbol{w}, \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{x}^{\mathrm{T}}(\boldsymbol{P}\boldsymbol{A})(\boldsymbol{P}^{\mathrm{T}}\boldsymbol{x})$.

To solve (16), we can use the conjugate gradient method. This method only accesses the Hessian through Hessian-vector products, i.e., $\nabla^2 \hat{L}(\boldsymbol{A}) \operatorname{vec}(\boldsymbol{D})$. By using the problem structure together with the property $(\boldsymbol{A} \otimes \boldsymbol{B}) \operatorname{vec}(\boldsymbol{D}) = \operatorname{vec}(\boldsymbol{B}\boldsymbol{D}\boldsymbol{A}^{\mathrm{T}})$, we can usually compute these products efficiently.

4.3 Squared loss case

For the case of the squared loss, we obtain very simple expressions and closed-form solutions.

Minimizing with respect to $\boldsymbol{w}$. For the squared loss, (7) becomes

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \frac{1}{2} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{\tau}\|^2 + \frac{\alpha}{2} \|\boldsymbol{w}\|^2,$$

where $\boldsymbol{\tau} \in \mathbb{R}^n$ is a vector with elements $\tau_i = y_i - \langle \boldsymbol{Z}, \boldsymbol{x}_i \boldsymbol{x}_i^{\mathrm{T}} \rangle$. This is a standard ridge regression problem. A closed-form solution can be computed by $\boldsymbol{w} = \boldsymbol{X}^{\mathrm{T}}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}} + \alpha \boldsymbol{I})^{-1}\boldsymbol{\tau}$ in $O(n^3)$ or by $\boldsymbol{w} = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} + \alpha \boldsymbol{I})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\tau}$ in $O(d^3)$. When n and d are both large, we can use an iterative method (e.g., conjugate gradient) instead.

Finding the dominant eigenvector. For finding the dominant eigenvector of $\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})$, we use the power iteration method, which needs to compute matrix-vector products $\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})\boldsymbol{p}$. For the squared loss, the gradient is given by:

$$\nabla L(\boldsymbol{Z}) = \sum_{i=1}^{n} r_i \boldsymbol{x}_i \boldsymbol{x}_i^{\mathrm{T}} = \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}\boldsymbol{X}, \tag{17}$$

where $\boldsymbol{R} = \operatorname{diag}(r_1, \ldots, r_n)$ and $r_i = \hat{y}_i - y_i$ is the residual of $\boldsymbol{x}_i$ at $(\boldsymbol{w}, \boldsymbol{Z})$. Clearly, we can compute $\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})\boldsymbol{p}$ efficiently without ever materializing $\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})$.

Minimizing with respect to $\boldsymbol{\lambda}$. For the squared loss, we obtain that (10) is equivalent to

$$\lambda_s = \operatorname*{argmin}_{\lambda \in \mathbb{R}} \nabla_s \tilde{L}(\bar{\boldsymbol{\lambda}})(\lambda - \bar{\lambda}_s) + \frac{1}{2} \nabla^2_{ss} \tilde{L}(\bar{\boldsymbol{\lambda}})(\lambda - \bar{\lambda}_s)^2 + \beta|\lambda| = \operatorname*{argmin}_{\lambda \in \mathbb{R}} \frac{1}{2}\Big(\lambda - \tilde{\lambda}_s\Big)^2 + c_s|\lambda|$$

where $\tilde{\lambda}_s := \bar{\lambda}_s - \frac{\nabla_s \tilde{L}(\bar{\boldsymbol{\lambda}})}{\nabla^2_{ss} \tilde{L}(\bar{\boldsymbol{\lambda}})}$ and $c_s := \frac{\beta}{\nabla^2_{ss} \tilde{L}(\bar{\boldsymbol{\lambda}})}$. This is the well-known soft-thresholding operator, whose closed-form solution is given by

$$\lambda_s = \operatorname{sign}(\tilde{\lambda}_s) \max(|\tilde{\lambda}_s| - c_s, 0).$$

The first and second derivatives of $\tilde{L}$ with respect to λ_s can be computed efficiently by

$$\nabla_s \tilde{L}(\boldsymbol{\lambda}) = \sum_{i=1}^{n} r_i \langle \boldsymbol{p}_s \boldsymbol{p}_s^{\mathrm{T}}, \boldsymbol{x}_i \boldsymbol{x}_i^{\mathrm{T}} \rangle = \sum_{i=1}^{n} r_i (\boldsymbol{p}_s^{\mathrm{T}} \boldsymbol{x}_i)^2 \tag{18}$$

$$\nabla_{ss}^2 \tilde{L}(\boldsymbol{\lambda}) = \sum_{i=1}^{n} \langle \boldsymbol{p}_s \boldsymbol{p}_s^{\mathrm{T}}, \boldsymbol{x}_i \boldsymbol{x}_i^{\mathrm{T}} \rangle^2 = \sum_{i=1}^{n} (\boldsymbol{p}_s^{\mathrm{T}} \boldsymbol{x}_i)^4, \tag{19}$$

where, again, $r_i = \hat{y}_i - y_i$ is the residual of $\boldsymbol{x}_i$ at $(\boldsymbol{w}, \boldsymbol{Z}_{\boldsymbol{\lambda}})$.

Fully-corrective refitting. For the squared loss, the Newton method gives the exact solution of (13) in one iteration and γ can be set to 1 (i.e., no line search needed). Given an initial guess $\bar{\boldsymbol{A}}$, if we solve the system

$$\nabla^2 \hat{L}(\bar{\boldsymbol{A}}) \operatorname{vec}(\boldsymbol{D}) = \operatorname{vec}\left(\nabla \hat{L}(\bar{\boldsymbol{A}})\right) \tag{20}$$

w.r.t. $\operatorname{vec}(\boldsymbol{D})$, then the optimal solution of (13) is $\boldsymbol{A} = \bar{\boldsymbol{A}} - \boldsymbol{D}$. To solve (20), we use the conjugate gradient method, which accesses the Hessian only through Hessian-vector products. Thus, we never need to materialize the Hessian matrix. The gradient and Hessian-vector product expressions are given by

$$\nabla \hat{L}(\boldsymbol{A}) = \boldsymbol{P}^{\mathrm{T}} \boldsymbol{X}^{\mathrm{T}} \boldsymbol{R} \boldsymbol{X} \boldsymbol{P} + \rho(\boldsymbol{A} - \boldsymbol{B} + \boldsymbol{M}) \tag{21}$$

$$\nabla^2 \hat{L}(\boldsymbol{A}) \operatorname{vec}(\boldsymbol{D}) = \operatorname{vec}(\boldsymbol{P}^{\mathrm{T}} \boldsymbol{X}^{\mathrm{T}} \boldsymbol{\Pi} \boldsymbol{X} \boldsymbol{P}) + \rho \operatorname{vec}(\boldsymbol{D}), \tag{22}$$

where $\boldsymbol{R} = \operatorname{diag}(r_1, \ldots, r_n)$, $r_i = \hat{y}_i - y_i$ is the residual of $\boldsymbol{x}_i$ at $(\boldsymbol{w}, \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}})$, $\boldsymbol{\Pi} = \operatorname{diag}(\pi_1, \ldots, \pi_n)$ and $\pi_i = \langle \boldsymbol{P}\boldsymbol{D}\boldsymbol{P}^{\mathrm{T}}, \boldsymbol{x}_i \boldsymbol{x}_i^{\mathrm{T}} \rangle = \boldsymbol{x}_i^{\mathrm{T}} (\boldsymbol{P}\boldsymbol{D}\boldsymbol{P}^{\mathrm{T}}) \boldsymbol{x}_i$. Note that the Hessian-vector product is independent of $\boldsymbol{A}$.

4.4 Computational complexity

We focus our discussion on minimizing w.r.t. $\boldsymbol{Z}$ when using the squared loss (we assume the implementation techniques described in Section G of the supplementary material are used). For power iteration, the main cost is computing the matrix-vector product $\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})\boldsymbol{p}$. From (17), this costs $O(N_z(\boldsymbol{X}))$. For minimizing with respect to λ_s, the main task consists in computing the first and second derivatives (18) and (19), which costs $O(n)$. For the fully corrective refitting, ADMM alternates between (13) and (14). For (13), the main cost stems from computing the gradient and Hessian-vector product (21) and (22), which takes $O(kN_z(\boldsymbol{X}) + dk^2)$. For (14), the main cost stems from computing the eigendecomposition of a $k \times k$ matrix, which takes $O(k^3)$, where $k \ll d$. If we use the binary indicator representation (2), then convex factorization machines have the same overall runtime cost as convex matrix factorization [7].

4.5 Convergence guarantees

Our method is an instance of block coordinate descent with two blocks, $\boldsymbol{w}$ and $\boldsymbol{Z}$. Past convergence analysis of block coordinate descent typically requires subproblems to have unique solutions [2, Proposition 2.7.1]. However, (5) is convex in $\boldsymbol{Z}$ but not strictly convex. Hence minimization with respect to $\boldsymbol{Z}$ may have multiple optimal solutions. Fortunately, for the case of two blocks, the uniqueness condition is not needed [6]. For minimization with respect to $\boldsymbol{Z}$, our greedy coordinate descent algorithm is an instance of [4] when using diagonal refitting and of [7] when using fully corrective refitting. Both methods asymptotically converge to an optimal solution, even if we find the dominant eigenvector only approximately. Thus, our two-block coordinate descent method asymptotically converges to a global minimum.

5 Experimental results

5.1 Synthetic experiments

We conducted experiments on synthetic data in order to compare the predictive power of different models:

- Convex FM (use diag): $\hat{y} = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \langle \boldsymbol{Z}, \boldsymbol{x}\boldsymbol{x}^{\mathrm{T}} \rangle$
- Convex FM (ignore diag): $\hat{y} = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \langle \boldsymbol{Z}, \boldsymbol{x}\boldsymbol{x}^{\mathrm{T}} - \mathrm{diag}(\boldsymbol{x})^2 \rangle$
- Original FM: $\hat{y} = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \sum_{j=1}^{d} \sum_{j'=j+1}^{d} (\boldsymbol{V}\boldsymbol{V}^{\mathrm{T}})_{jj'} x_j x_{j'}$
- Ridge regression: $\hat{y} = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$
- Kernel ridge regression: $\hat{y} = \sum_{i=1}^{n} a_i \kappa(\boldsymbol{x}_i, \boldsymbol{x})$

For kernel ridge regression, the kernel used was the polynomial kernel of degree 2: $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\gamma + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j)^2$. Due to lack of space, the parameter estimation procedure for Convex FM (ignore diag) is explained in the supplementary material. We compared the above models under various generative assumptions.

Data generation. We generated $\boldsymbol{y} = [y_1, \ldots, y_n]^{\mathrm{T}}$ by $y_i = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + \langle \boldsymbol{Z}, \boldsymbol{x}_i\boldsymbol{x}_i^{\mathrm{T}} \rangle$ (use diagonal case) or by $y_i = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + \langle \boldsymbol{Z}, \boldsymbol{x}_i\boldsymbol{x}_i^{\mathrm{T}} - \mathrm{diag}(\boldsymbol{x}_i)^2 \rangle$ (ignore diagonal case). To generate $\boldsymbol{w} = [w_1, \ldots, w_d]^{\mathrm{T}}$, we used $w_j \sim \mathcal{N}(0,1)\ \forall j \in [d]$ where $\mathcal{N}(0,1)$ is the standard normal distribution. To generate $\boldsymbol{Z} = \boldsymbol{P}\,\mathrm{diag}(\boldsymbol{\lambda})\boldsymbol{P}^{\mathrm{T}}$, we used $p_{js} \sim \mathcal{N}(0,1)\ \forall j \in [d]\ \forall s \in [k]$ and $\lambda_s \in \mathcal{N}(0,1)\ \forall s \in [d]$ (not positive semi-definite [PSD] case) or $\lambda_s \sim \mathcal{U}(0,1)\ \forall s \in [d]$ (positive semi-definite case), where $\mathcal{U}(0,1)$ is the uniform distribution between 0 and 1. For generating $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, we compared two cases. In the dense case, we used $x_{ij} \sim \mathcal{N}(0,1)\ \forall i \in [n]\ \forall j \in [d]$. In the sparse case, we sampled $\bar{d}$ features from a multinomial distribution whose parameters are set uniformly at random. We chose $n = 1000$, $d = 50$, $k = 5$ and $\bar{d} = 5$. We split the data into 75% training and 25% testing and added 1% Gaussian noise to the training targets.

Results. Results (RMSE on test data) are indicated in Table 1. Hyper-parameters of the respective methods were optimized by 5-fold cross-validation. The setting which is most favorable to Original FM is when the matrix $\boldsymbol{Z}$ used for generating synthetic data is PSD and diagonal elements of $\boldsymbol{Z}$ are ignored (2nd and 6th rows in Table 1). In this case, Original FM performed well, although worse than Convex FM (ignore diag). However, in other settings, especially when $\boldsymbol{Z}$ is not PSD, convex FM outperformed the Original FM. For example, for dense data, when $\boldsymbol{Z}$ is not PSD and diagonal elements of $\boldsymbol{Z}$ are ignored, Convex FM

Table 1: Test RMSE of different methods on synthetic data.

Generative process	Convex FM (use diag)	Convex FM (ignore diag)	Original FM	Ridge	Kernel ridge (polynomial kernel)
dense, PSD, use diag	**68.35**	110.18	104.39	104.67	76.77
dense, PSD, ignore diag	27.45	**5.93**	5.97	56.91	31.74
dense, not PSD, use diag	**92.31**	159.47	165.90	223.76	154.12
dense, not PSD, ignore diag	60.74	**21.17**	139.66	208.55	138.17
sparse, PSD, use diag	**23.12**	25.23	23.82	25.45	25.10
sparse, PSD, ignore diag	8.93	**5.10**	5.92	21.41	14.39
sparse, not PSD, use diag	**12.75**	23.13	30.60	36.43	25.17
sparse, not PSD, ignore diag	11.66	**7.91**	27.46	34.62	21.75

(use diag) achieved a test RMSE of 60.74, Convex FM (ignore diag) 21.17 and Original FM 139.66. Ridge regression was the worst method in all settings. This is not surprising since it does not use feature interactions. Kernel ridge regression with a polynomial kernel of degree 2 outperformed ridge regression but was worse than convex FM on all datasets.

5.2 Recommender system experiments

We also conducted experiments on 4 standard recommendation tasks. Datasets used in our experiments are summarized below.

Dataset	n	$d = \lvert U\rvert + \lvert I\rvert$
Movielens 100k	100,000 (ratings)	2,625 = 943 (users) + 1,682 (movies)
Movielens 1m	1,000,209 (ratings)	9,940 = 6,040 (users) + 3,900 (movies)
Movielens 10m	10,000,054 (ratings)	82,248 = 71,567 (users) + 10,681 (movies)
Last.fm	108,437 (tag counts)	24,078 = 12,133 (artists) + 11,945 (tags)

For simplicity, we used the binary indicator representation (2), which results in a design matrix $\boldsymbol{X}$ of size $n \times d$. We split samples uniformly at random between 75% for training and 25% for testing. For Movielens datasets, the task is to predict ratings between 1 and 5 given by users to movies, i.e., $y \in \{1, \ldots, 5\}$. For Last.fm, the task is to predict the number of times a tag was assigned to an artist, i.e., $y \in \mathbb{N}$. In all experiments, we set $\alpha = 10^{-9}$ for convex and original factorization machines, as well as ridge regression. Because we used the binary indicator representation (2), $\boldsymbol{w}$ plays the same role as unpenalized bias terms (c.f., Section A in the supplementary material).

Solver comparison. For minimizing our objective function with respect to $\boldsymbol{Z}$, we compared greedy coordinate descent (GCD) with diagonal refitting and with fully-corrective refitting, the proximal gradient method and ADMM. Minimization with respect to $\boldsymbol{w}$ was carried out using the conjugate gradient method. Results when setting $\beta = 10$ are given in Figure 1. We were only able to run ADMM on Movielens 100K because it needs to materialize $\boldsymbol{Z}$ in memory. Experiments were run on a machine with Intel Xeon X5677 CPU (3.47GHz) and 48 GB memory.

Results. GCD with fully-corrective refitting was consistently the best solver both with respect to objective value and test RMSE. GCD with diagonal refitting converged slower with respect to objective value but was similar with respect to test RMSE, except on Last.fm. The proximal gradient and ADMM methods were an order of magnitude slower than GCD.

Model comparison. We used the same setup as in Section 5.1 except that we replaced kernel ridge regression with support vector regression (we used the implementation in libsvm, which has a kernel cache and scales better than kernel ridge regression w.r.t. n). For hyper-parameter tuning, we used 3-fold cross-validation (CV). For convex and original factorization machines, we chose β from 10 log-spaced values between 10^{-1} and 10^{2}. For original factorization machines, we also chose k from $\{10, 20, 30, 40, 50\}$. For Movielens 10M, we only chose β from 5 log-spaced values and we set $k = 20$ in order to reduce the search space. For SVR, we chose the regularization parameter C from 10 log-spaced values between 10^{-5} and 10^{5}. For convex factorization machines, we made use of warm-start when computing the regularization path in order to accelerate training. For practical reasons, we used early stopping in order to keep rank($\boldsymbol{Z}$) under 50.

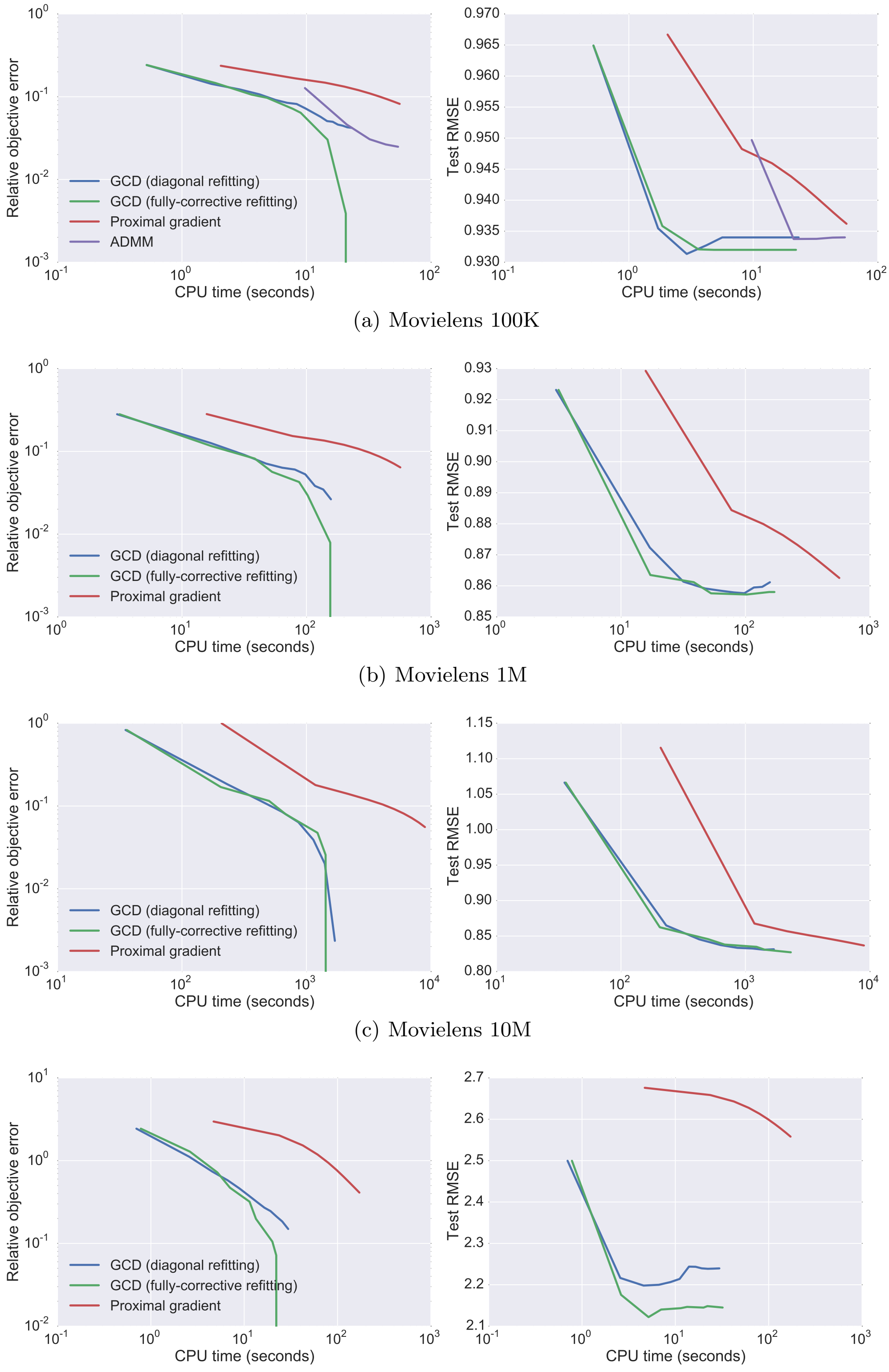
Results. Test RMSE, training time (including hyper-parameter tuning using 3-fold CV) and the rank obtained (when applicable) are indicated in Table 2. Except on Movielens 100k, Convex FM (ignore diag) obtained lower RMSE, was faster to converge and obtained lower rank than Convex FM (use diag). This comes however at the cost of more complicated gradient and Hessian expressions (c.f., Section E in the supplementary material for details). Except on Movielens 10M, Convex FM (ignore diag) obtained lower RMSE than Original FM. Training time was also lower thanks to the reduced number of hyper-parameters to search. Ridge regression (RR) was a surprisingly strong baseline, SVR was worse than RR. This is due to the extreme sparsity of the design matrix when using the binary indicator representation (2). Since features co-occur exactly only once, SVR cannot exploit the feature interactions despite the use of polynomial kernel. In contrast, factorization machines are able to exploit feature interactions despite high sparsity thanks to the parameter sharing induced by the factorization $\boldsymbol{Z} = \boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^{\mathrm{T}}$.

6 Related work

Recently, convex formulations for the estimation of a low-rank matrix have been extensively studied. The key idea [5] is to replace the rank of a matrix, which is non-convex, by the nuclear norm (a.k.a. trace norm), which is known to be the tightest convex lower bound on matrix rank [11]. Nuclear norm regularization has been applied to numerous applications, including multi-task learning and matrix completion [18]. The latter is typically formulated as the following optimization problem. Given a matrix $\boldsymbol{X} \in \mathbb{R}^{|U| \times |I|}$ containing missing values, we solve

$$\min_{\boldsymbol{M} \in \mathbb{R}^{|U| \times |I|}} \frac{1}{2} \|\mathcal{P}_{\Omega}(\boldsymbol{X}) - \mathcal{P}_{\Omega}(\boldsymbol{M})\|_F^2 + \lambda \|\boldsymbol{M}\|_*, \tag{23}$$

where Ω is the set of observed values in $\boldsymbol{X}$ and $(\mathcal{P}_{\Omega}(\boldsymbol{M}))_{i,j} = (\boldsymbol{M})_{i,j}$ if $(i, j) \in \Omega$, 0 otherwise. Extensions to tensor factorization have also been proposed for data



(a) Movielens 100K

(b) Movielens 1M

(c) Movielens 10M

(d) Last.fm

Fig. 1: Solver comparison when using $\alpha = 10^{-9}$ and $\beta = 10$. Left: relative objective error $|(f^t - f^*)/f^*|$, where f^t is the objective value measured on time t and f^* is the optimal objective value. Right: RMSE on test data.

Table 2: Test RMSE, training time (including hyper-parameter tuning using 3-fold cross-validation) and rank of different models on real data. Results are averaged over 3 runs using different train / test splits (rank uses the median).

Dataset		Convex FM (use diag)	Convex FM (ignore diag)	Original FM	Ridge	SVR (polynomial kernel)
Movielens 100k	RMSE	**0.93**	**0.93**	**0.93**	0.95	1.20
	Time	7.09 min	6.72 min	10.05 min	0.28 s	35.30 s
	Rank	23	20	20		
Movielens 1m	RMSE	0.87	**0.85**	0.86	0.91	1.24
	Time	1.07 h	38.74 min	3.93 h	3.14 s	3.68 min
	Rank	27	20	20		
Movielens 10m	RMSE	0.84	0.82	**0.81**	0.87	N/A
	Time	5.02 h	4.29 h	5.84 h	59.35 s	N/A
	Rank	34	17	20		
Last.fm	RMSE	2.21	**2.05**	2.13	2.60	3.24
	Time	7.77 min	6.91 min	14.17 min	0.63 s	36.70 s
	Rank	50	48	40		

with more than two modes (e.g., user, item and time) [19]. However, in (23) and tensor extensions, it is not trivial to incorporate auxiliary features such as user (age, gender, ...) and item (release date, director's name, ...) attributes. The most related work to convex factorization machines is [1], in which a collaborative filtering method which can incorporate additional attributes is proposed. However, their method can only handle two modes (e.g., user and item) and no scalable learning algorithm is proposed. The advantage of convex factorization machines is that it is very easy to engineer features, even for more than two modes (e.g., user, item and context).

7 Conclusion

Factorization machines are a powerful framework that can exploit feature interactions even when features co-occur very rarely. In this paper, we proposed a convex formulation of factorization machines. Our formulation imposes fewer restrictions on the feature interaction weight matrix and is thus more general than the original one. For solving the corresponding optimization problem, we presented an efficient globally-convergent two-block coordinate descent algorithm. Our formulation achieves comparable or lower predictive error on several synthetic and real-world benchmarks. It can also overall be faster to train since it has one less hyper-parameter than the original formulation. As a side contribution, we also clarified the convexity properties (or lack thereof) of the original factorization machine's objective function. Future work includes trying (convex) factorization machines on more data (e.g., genomic data, where feature interactions should be useful) and developing algorithms for out-of-core learning.

References

1. Abernethy, J., Bach, F., Evgeniou, T., Vert, J.P.: A new approach to collaborative filtering: Operator estimation with spectral regularization. J. Mach. Learn. Res. 10, 803–826 (2009)
2. Bertsekas, D.P.: Nonlinear programming. Athena scientific Belmont (1999)
3. Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., Varoquaux, G.: API design for machine learning software: experiences from the scikit-learn project. In: ECML PKDD Workshop: Languages for Data Mining and Machine Learning. pp. 108–122 (2013)
4. Dudik, M., Harchaoui, Z., Malick, J.: Lifted coordinate descent for learning with trace-norm regularization. In: AISTATS. vol. 22, pp. 327–336 (2012)
5. Fazel, M., Hindi, H., Boyd, S.P.: A rank minimization heuristic with application to minimum order system approximation. In: American Control Conference. vol. 6, pp. 4734–4739 (2001)
6. Grippo, L., Sciandrone, M.: On the convergence of the block nonlinear gauss–seidel method under convex constraints. Operations Research Letters 26(3), 127–136 (2000)
7. Hsieh, C.J., Olsen, P.: Nuclear norm minimization via active subspace selection. In: ICML. pp. 575–583 (2014)
8. Koren, Y.: Factorization meets the neighborhood: a multifaceted collaborative filtering model. In: KDD. pp. 426–434 (2008)
9. Koren, Y.: Collaborative filtering with temporal dynamics. Communications of the ACM 53(4), 89–97 (2010)
10. Loni, B., Shi, Y., Larson, M., Hanjalic, A.: Cross-domain collaborative filtering with factorization machines. In: Advances in Information Retrieval, pp. 656–661. Springer (2014)
11. Recht, B., Fazel, M., Parrilo, P.A.: Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization. SIAM review 52(3), 471–501 (2010)
12. Rendle, S.: Factorization machines. In: ICDM. pp. 995–1000. IEEE (2010)
13. Rendle, S.: Factorization machines with libfm. ACM Transactions on Intelligent Systems and Technology (TIST) 3(3), 57–78 (2012)
14. Rendle, S.: Scaling factorization machines to relational data. In: VLDB. vol. 6, pp. 337–348 (2013)
15. Rendle, S., Gantner, Z., Freudenthaler, C., Schmidt-Thieme, L.: Fast context-aware recommendations with factorization machines. In: SIGIR. pp. 635–644 (2011)
16. Rendle, S., Schmidt-Thieme, L.: Pairwise interaction tensor factorization for personalized tag recommendation. In: WSDM. pp. 81–90. ACM (2010)
17. Shalev-Shwartz, S., Gonen, A., Shamir, O.: Large-scale convex minimization with a low-rank constraint. In: ICML. pp. 329–336 (2011)
18. Srebro, N., Rennie, J., Jaakkola, T.S.: Maximum-margin matrix factorization. In: Advances in neural information processing systems. pp. 1329–1336 (2004)
19. Tomioka, R., Hayashi, K., Kashima, H.: Estimation of low-rank tensors via convex optimization. arXiv preprint arXiv:1010.0789 (2010)

Supplementary material

A Equivalence with matrix factorization

Let U be a set of users and I a set of items. Matrix factorization models typically predict missing values using

$$\hat{y}_{ui} \coloneqq \boldsymbol{A}_u^{\mathrm{T}} \boldsymbol{B}_i + a_u + b_i \quad \forall u \in U \quad \forall i \in I,$$

where $\boldsymbol{A} \in \mathbb{R}^{|U| \times k}$, $\boldsymbol{B} \in \mathbb{R}^{|I| \times k}$, $\boldsymbol{a} \in \mathbb{R}^{|U|}$ and $\boldsymbol{b} \in \mathbb{R}^{|I|}$. If we use the binary indicator representation (2), then factorization machines are exactly equivalent to matrix factorization. To see why, choose

$$\begin{aligned} \boldsymbol{V} &\coloneqq \begin{pmatrix} \boldsymbol{A} \\ \boldsymbol{B} \end{pmatrix} \in \mathbb{R}^{d \times k} \quad \text{where} \quad d = |U| + |I| \\ \boldsymbol{w} &\coloneqq \begin{pmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{pmatrix} \in \mathbb{R}^{d}. \end{aligned}$$

Then

$$\begin{aligned} \hat{y}_{ui} &= \boldsymbol{V}_u^{\mathrm{T}} \boldsymbol{V}_j \overbrace{x_u x_j}^{=1} + w_u \overbrace{x_u}^{=1} + w_j \overbrace{x_j}^{=1} \quad \text{where} \quad j = |U| + i \\ &= \boldsymbol{A}_u^{\mathrm{T}} \boldsymbol{B}_i + a_u + b_i. \end{aligned}$$

Furthermore, we have

$$\boldsymbol{V}\boldsymbol{V}^{\mathrm{T}} = \begin{pmatrix} \boldsymbol{A} \\ \boldsymbol{B} \end{pmatrix} \begin{pmatrix} \boldsymbol{A}^{\mathrm{T}} & \boldsymbol{B}^{\mathrm{T}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}\boldsymbol{A}^{\mathrm{T}} & \boldsymbol{A}\boldsymbol{B}^{\mathrm{T}} \\ \boldsymbol{B}\boldsymbol{A}^{\mathrm{T}} & \boldsymbol{B}\boldsymbol{B}^{\mathrm{T}} \end{pmatrix}.$$

Since $j' > j$ in (1), if we use the binary indicator representation (2), factorization machines only use the upper-right block. Note, however, that even though they are not used, other blocks will not be zero.

The equivalence with matrix factorization also holds for convex factorization machines as long as we ignore the diagonal elements of $\boldsymbol{Z}$ and we constrain $\boldsymbol{Z}$ to be positive semi-definite (and thus, there exists $\boldsymbol{V}$ such that $\boldsymbol{Z} = \boldsymbol{V}\boldsymbol{V}^{\mathrm{T}}$). If we use diagonal elements or do not use a positive semi-definite constraint, convex factorization machines may learn a slightly different model.

B Proof of Proposition 1

We want to minimize

$$F = \sum_{i=1}^{n} \ell_i + \frac{\alpha}{2} ||\boldsymbol{w}||^2 + \frac{\beta}{2} \|\boldsymbol{V}\|_F^2,$$

where

$$\ell_i = \ell(y_i, \tilde{y}_i) \quad \text{and} \quad \tilde{y}_i = \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_i + \frac{1}{2} \sum_{s=1}^{k} \Big[\Big(\sum_{j=1}^{d} v_{js} x_{ij} \Big)^2 - \sum_{j=1}^{d} v_{js}^2 x_{ij}^2 \Big].$$

The first and second derivatives of F w.r.t. v_{js} are given by

$$\begin{aligned}
\frac{\partial F}{\partial v_{js}} &= \sum_{i=1}^{n} \frac{\partial \ell_i}{\partial \tilde{y}_i} \frac{\partial \tilde{y}_i}{\partial v_{js}} + \beta v_{js} \\
\frac{\partial^2 F}{\partial v_{js}^2} &= \sum_{i=1}^{n} \left[\frac{\partial^2 \ell_i}{\partial v_{js} \partial \tilde{y}_i} \frac{\partial \tilde{y}_i}{\partial v_{js}} + \frac{\partial \ell_i}{\partial \tilde{y}_i} \frac{\partial^2 \tilde{y}_i}{\partial v_{js}^2} \right] + \beta \\
&= \sum_{i=1}^{n} \left[\frac{\partial^2 \ell_i}{\partial \tilde{y}_i^2} \left(\frac{\partial \tilde{y}_i}{\partial v_{js}} \right)^2 \right] + \beta \\
&\geq 0,
\end{aligned}$$

where we used

$$\frac{\partial \tilde{y}_i}{\partial v_{js}} = x_{ij} \Big(\sum_{j'=1}^{d} v_{j's} x_{ij'} - v_{js} x_{ij} \Big), \quad \frac{\partial^2 \tilde{y}_i}{\partial v_{js}^2} = 0, \quad \text{and} \quad \frac{\partial^2 \ell_i}{\partial v_{js} \partial \tilde{y}_i} = \frac{\partial^2 \ell_i}{\partial \tilde{y}_i^2} \frac{\partial \tilde{y}_i}{\partial v_{js}}.$$

Therefore, F is convex w.r.t. v_{js} if ℓ is twice differentiable convex, and strictly convex if, in addition, $\beta > 0$. If we use the diagonal elements of $\boldsymbol{V}\boldsymbol{V}^{\mathrm{T}}$, then $\frac{\partial^2 \tilde{y}_i}{\partial v_{js}^2} \neq 0$ and F becomes non-convex w.r.t. v_{js}. Non-convexity w.r.t. $\boldsymbol{V}$ (whether we use diagonal elements of $\boldsymbol{V}\boldsymbol{V}^{\mathrm{T}}$ or not) can easily be verified by finding counter-examples (checked either visually or numerically).

C Nuclear norm of a symmetric matrix

For any symmetric matrix $\boldsymbol{Z} \in \mathbb{S}^{d \times d}$, the nuclear norm is defined as

$$\begin{aligned}
\|\boldsymbol{Z}\|_* &= \mathrm{Tr}\left((\boldsymbol{Z}^2)^{\frac{1}{2}} \right) \\
&= \mathrm{Tr}\left((\boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^{\mathrm{T}}\boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^{\mathrm{T}})^{\frac{1}{2}} \right) && \text{eigendecomposition of } \boldsymbol{Z} \\
&= \mathrm{Tr}\left((\boldsymbol{P}\boldsymbol{\Lambda}^2\boldsymbol{P}^{\mathrm{T}})^{\frac{1}{2}} \right) && \text{orthogonality of } \boldsymbol{P} \\
&= \mathrm{Tr}\left(\boldsymbol{P}(\boldsymbol{\Lambda}^2)^{\frac{1}{2}}\boldsymbol{P}^{\mathrm{T}} \right) && \text{square root of an eigendecomposition} \\
&= \mathrm{Tr}\left((\boldsymbol{\Lambda}^2)^{\frac{1}{2}} \right) && \text{trace of a PSD matrix} \\
&= \mathrm{Tr}\left(|\boldsymbol{\Lambda}| \right) \\
&= \|\boldsymbol{\lambda}\|_1 && \boldsymbol{\Lambda} = \mathrm{diag}(\boldsymbol{\lambda}).
\end{aligned}$$

D Positive semi-definite constraint

To constrain $\boldsymbol{Z}$ to be positive semi-definite (PSD), we only need to make a few straightforward modifications. We need to impose a non-negativity constraint on $\boldsymbol{\lambda}$. Thus, the optimality violation with respect to λ_s becomes

$$\nu_s = \begin{cases} |\nabla_s \tilde{L}(\boldsymbol{\lambda}) + \beta|, & \text{if } \lambda_s > 0 \\ |\min(\nabla_s \tilde{L}(\boldsymbol{\lambda}) + \beta, 0)|, & \text{if } \lambda_s = 0. \end{cases}$$

For finding $\boldsymbol{p}_s$ with greatest violation, we need to solve

$$\operatorname*{argmin}_{s \in \mathcal{S}} \nabla_s \tilde{L}(\boldsymbol{\lambda}) = \operatorname*{argmin}_{s \in \mathcal{S}} \boldsymbol{p}_s^{\mathrm{T}} \nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}}) \boldsymbol{p}_s = \operatorname*{argmax}_{s \in \mathcal{S}} \boldsymbol{p}_s^{\mathrm{T}} \Big(-\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}}) \Big) \boldsymbol{p}_s.$$

Thus, we need to find the eigenvector of $-\nabla L(\boldsymbol{Z}_{\boldsymbol{\lambda}})$ with largest eigenvalue. The closed-form solution for updating λ_s (squared loss case) becomes

$$\lambda_s = \max(\tilde{\lambda}_s - c_s, 0).$$

where, again, $\tilde{\lambda}_s := \bar{\lambda}_s - \frac{\nabla_s \tilde{L}(\bar{\boldsymbol{\lambda}})}{\nabla^2_{ss} \tilde{L}(\bar{\boldsymbol{\lambda}})}$ and $c_s := \frac{\beta}{\nabla^2_{ss} \tilde{L}(\bar{\boldsymbol{\lambda}})}$. Given an eigendecomposition $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Sigma}\boldsymbol{Q}^{\mathrm{T}}$, where $\boldsymbol{\Sigma} = \operatorname{diag}(\sigma_1, \ldots, \sigma_k)$, the shrinkage operator becomes

$$S_c(\boldsymbol{A}) = \operatorname*{argmin}_{\boldsymbol{B} \succeq 0} \frac{1}{2} \|\boldsymbol{A} - \boldsymbol{B}\|_F^2 + c\|\boldsymbol{B}\|_* = \boldsymbol{Q} \operatorname{diag}(\hat{\sigma}_1, \ldots, \hat{\sigma}_k) \boldsymbol{Q}^{\mathrm{T}},$$

where $\hat{\sigma}_s = \max(\sigma_s - c, 0)$. Note that if $\boldsymbol{A} \succeq 0$, then $\boldsymbol{Z} = \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}} \succeq 0$ as well.

E Ignoring diagonal elements of $\boldsymbol{Z}$

We can ignore diagonal elements of $\boldsymbol{Z}$ like in the original factorization machines, albeit at the cost of slightly more complicated expressions. The prediction function becomes

$$\begin{aligned} \hat{y}(\boldsymbol{x}|\boldsymbol{w}, \boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^{\mathrm{T}}) &:= \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \langle \boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^{\mathrm{T}}, \boldsymbol{x}\boldsymbol{x}^{\mathrm{T}} - \operatorname{diag}(\boldsymbol{x})^2 \rangle \\ &= \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \sum_{s \in \operatorname{supp}(\boldsymbol{\lambda})} \lambda_s \Big[(\boldsymbol{p}_s^{\mathrm{T}}\boldsymbol{x})^2 - \|\boldsymbol{p}_s \circ \boldsymbol{x}\|^2 \Big]. \end{aligned}$$

We now focus on the squared loss. The gradient of L becomes

$$\nabla L(\boldsymbol{Z}) = \sum_{i=1}^n r_i (\boldsymbol{x}_i \boldsymbol{x}_i^{\mathrm{T}} - \operatorname{diag}(\boldsymbol{x}_i)^2) = \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}\boldsymbol{X} - \boldsymbol{\Delta},$$

where $\boldsymbol{R} = \operatorname{diag}(r_1, \ldots, r_n)$, $r_i = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + \langle \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}, \boldsymbol{x}_i, \boldsymbol{x}_i^{\mathrm{T}} - \operatorname{diag}(\boldsymbol{x}_i)^2 \rangle$ and $\boldsymbol{\Delta} = \sum_{i=1}^n r_i \operatorname{diag}(\boldsymbol{x}_i)^2$. The first and second derivatives of $\tilde{L}$ with respect to λ_s can be computed efficiently as follows

$$\begin{aligned} \nabla_s \tilde{L}(\boldsymbol{\lambda}) &= \sum_{i=1}^n r_i \langle \boldsymbol{p}_s \boldsymbol{p}_s^{\mathrm{T}}, \boldsymbol{x}_i \boldsymbol{x}_i^{\mathrm{T}} - \operatorname{diag}(\boldsymbol{x}_i)^2 \rangle = \sum_{i=1}^n r_i \Big[(\boldsymbol{p}_s^{\mathrm{T}}\boldsymbol{x}_i)^2 - \|\boldsymbol{p}_s \circ \boldsymbol{x}_i\|^2 \Big] \\ \nabla^2_{ss} \tilde{L}(\boldsymbol{\lambda}) &= \sum_{i=1}^n \langle \boldsymbol{p}_s \boldsymbol{p}_s^{\mathrm{T}}, \boldsymbol{x}_i \boldsymbol{x}_i^{\mathrm{T}} - \operatorname{diag}(\boldsymbol{x}_i)^2 \rangle^2 = \sum_{i=1}^n \Big[(\boldsymbol{p}_s^{\mathrm{T}}\boldsymbol{x}_i)^2 - \|\boldsymbol{p}_s \circ \boldsymbol{x}_i\|^2 \Big]^2. \end{aligned}$$

The gradient and Hessian-vector product expressions of $\hat{L}$ with respect to $\boldsymbol{A}$ become

$$\begin{aligned}\nabla \hat{L}(\boldsymbol{A}) &= \boldsymbol{P}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}\boldsymbol{X}\boldsymbol{P} - \boldsymbol{P}^{\mathrm{T}}\boldsymbol{\Delta}\boldsymbol{P} + \rho(\boldsymbol{A} - \boldsymbol{B} + \boldsymbol{M}) \\ \nabla^2 \hat{L}(\boldsymbol{A})\operatorname{vec}(\boldsymbol{D}) &= \operatorname{vec}(\boldsymbol{P}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{\Pi}\boldsymbol{X}\boldsymbol{P}) - \operatorname{vec}(\boldsymbol{P}^{\mathrm{T}}\tilde{\boldsymbol{\Delta}}\boldsymbol{P}) + \rho\operatorname{vec}(\boldsymbol{D}),\end{aligned}$$

where $\boldsymbol{\Pi} = \operatorname{diag}(\pi_1, \dots, \pi_n)$, $\pi_i = \langle \boldsymbol{P}\boldsymbol{D}\boldsymbol{P}^{\mathrm{T}}, \boldsymbol{x}_i\boldsymbol{x}_i^{\mathrm{T}} - \operatorname{diag}(\boldsymbol{x}_i)^2\rangle$ and $\tilde{\boldsymbol{\Delta}} = \sum_{i=1}^n \pi_i \operatorname{diag}(\boldsymbol{x}_i)^2$.

F Proximal gradient method

On each iteration, the proximal gradient method updates $\boldsymbol{Z}$ using

$$\boldsymbol{Z}^{t+1} = S_{\beta/\mu}\Big(\boldsymbol{Z}^t - \frac{1}{\mu}\nabla L(\boldsymbol{Z}^t)\Big),$$

where μ is a step-size parameter and S_c is the shrinkage operator (15). For computing the eigendecomposition of $\boldsymbol{Z}^t - \frac{1}{\mu}\nabla L(\boldsymbol{Z}^t)$, we can use the Lanczos method, which only needs $\boldsymbol{Z}^t - \frac{1}{\mu}\nabla L(\boldsymbol{Z}^t)$ through matrix-vector products. Thus as long as we maintain $\boldsymbol{Z}$ in factorized form, the proximal gradient method does not need to materialize $\boldsymbol{Z}$. Our implementation is based on the `eigsh` function available in SciPy's `scipy.sparse.linalg` module. Matrix-vector products can be implemented using the `LinearOperator` class available from the same module.

G Implementation details

For expressions such as (17), (18), (19), (21) and (22), we only need to iterate over non-zero features. Thus, we store the design matrix $\boldsymbol{X}$ in sparse row-major format. For the diagonal refitting case, we only change one λ_s at a time. We can store the current model predictions $\hat{y}_1, \dots, \hat{y}_n$. From (6), when λ_s is modified by $\lambda_s \leftarrow \lambda_s + \delta$, we simply need to modify $\hat{y}_i$ by $\hat{y}_i \leftarrow \hat{y}_i + \delta(\boldsymbol{p}_s^{\mathrm{T}}\boldsymbol{x}_i)^2$. We can cache $\boldsymbol{p}_s^{\mathrm{T}}\boldsymbol{x}_i$ for all $i \in [n]$ and for all $s \in \operatorname{supp}(\boldsymbol{\lambda})$. Additionally, for the squared loss, we can cache $\nabla^2_{ss}\tilde{L}(\boldsymbol{\lambda})$ for all $s \in \operatorname{supp}(\boldsymbol{\lambda})$, since it is independent of $\boldsymbol{\lambda}$; see (19). For the fully corrective case, we maintain $\boldsymbol{Z}$ in the form $\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}$ throughout the course of the algorithm. We also make extensive use of warm start. When minimizing with respect to $\boldsymbol{w}$ or $\boldsymbol{Z}$, we use the current estimate as initialization. Warm start can also be used to compute a regularization path. A simple technique for improving convergence speed is to perform refitting only periodically. Finally, while our algorithm does not strictly require a rank parameter, we can define a maximum rank "budget" parameter. Because our algorithm is greedy, we can stop it when this budget is reached (early stopping).