Binary Matrix Factorization

Ignacio Ramírez

March 22, 2018

1 introduction

We consider the problem of approximating a binary matrix $X \in \{0,1\}^{m \times n}$ as the product of two other binary matrices $U \in \{0,1\}^{m \times k}$ and $V \in \{0,1\}^{n \times p}$ plus a small residual,

$$X = UV^{\intercal} + E. \tag{1}$$

Matrix Factorization The problem (**??**) has been addressed using various methods from different fields for over a hundred years. Two such fields are Signal Processing and Machine Learning under the so-called "Matrix Factorization" (MF) methods, (with non-negative matrix factorization (NMF) a very popular particular case,) which is very mature and has been developed largely for the case of real matrices.

Dictionary Learning A particular case of MF is the so-called "Dictionary Learning" problem which is particularly useful when $m \gg n$. In this case, contrary to the more general MF approach, the roles of U and V are quite different and, accordingly, both are estimated in quite different ways. More specifically, U is called a dictionary (and usually assigned the letter D), whereas $V^{\intercal} = A$ is a matrix of "linear combination coefficients". The columns of the matrix D are called "atoms" and are supposed to embody typical patterns observed throughout the columns of X, while the columns of A, usually assumed sparse, specify the linear combination of columns of D that better approximates the corresponding columns of X.

Data Mining On the other hand, there is a large body of work from the Data Mining community on the subject of extracting general "patterns" from large matrices, in particular binary matrices. Some of those works are based on matrix factorization concepts, for example the Proximus algorithm [**?**],although most of them are quite heuristic.

Other ideas Initially I was interested in connecting the above techniques with other learning frameworks aimed at binary data. In particular, the Bidirectional Auto-associative Memory (BAM) model [?] was an early binary recurrent neural network.

Our approach The idea here is very simple: to apply a Dictionary Learning approach to the problem of binary matrix factorization for those cases where the matrix X is significantly "fat" $m \gg n$ (or, naturally, apply it to $X^\intercal$ when it is "tall", $m \ll n$).

2 The Dictionary Learning approach

The problem (**??**) is generally non-convex. There are very special cases in which it can be solved exactly, or it reduces to a tractable convex problem under particular conditions. Most DL approaches fall into the general setting where (**??**) is NP-hard and only local convergence (to a stationary point) can be guaranteed. This is usually achieved through *alternate (block) minimization* on D and A,

$$\begin{aligned} A^{(k+1)} &= \quad \arg\min_A\{f(D^{(k)} - A) + g(A)\} && (2) \\ D^{(k+1)} &= \arg\min_D\{f(D - A^{(k+1)}) + g(A^{(k+1)})\}, && (3) \end{aligned}$$

where $f(\cdot)$ and $g(\cdot)$ are *fitting* and *regularization* functions respectively. The first one is typically the squared ℓ_2 norm, $\|\cdot\|^2$, and the second one is an ℓ_p norm such as $\|\cdot\|_1$, with $0 \leq p \leq 2$ (when $p < 1$ it is not a norm). For that case, this algorithm is known as *Method of Directions* or MOD [?]:

$$\begin{aligned} A_j^{(k+1)} &= \arg\min_{a\in\mathbb{R}^p}\{\|x_j - D^{(k)}a\|_2^2 + \|a\|_1,\ j = 1,\ldots,n\} && (4) \\ D_r^{(k+1)} &= u_j/\|u_j\|_2,\ u_j = X(A^{(k+1)})^\intercal \left(A^{(k+1)}(A^{(k+1)})^\intercal\right)^{-1}\}, && (5) \end{aligned}$$

where A_j and D_r are the j-th and r-th columns of A and D respectively. The first step corresponds to an ℓ_1-regularized least squares regression problem on each column of A, also known as LASSO [?]. In the second step, each atom in the dictionary corresponds to a normalized down version of the least squares solution u; note that here it is customary as well to apply some sort of regularization so that $AA^\intercal$ is invertible (non-singular).

K-SVD Another popular approach to the dictionary learning problem is the K-SVD method [?]. In this case, $g(\cdot)$ corresponds to the ℓ_0 pseudonorm, which counts the number of non-zeros in a vector. The updates on A are similar to MOD, but use a greedy method known as OMP (Orthogonal Matching Pursuit) [?] to obtain an approximate solution,

```
**Data**: vector to encode $x$, dictionary $D$, maximum residual norm $\epsilon$
**Result**: Optimal coefficients for $x$, $a$
Set iteration $k = 0$, residual $r^{(0)} = x$, coefficients $a^{(0)} = 0$;
**while** $\left\| r^{(k)} \right\| \geq \epsilon$ **do**
    $i = \arg\max \left\{ D_i^\intercal r^{(k)} \right\}$ ;
    $a_i \leftarrow D_i^\intercal r^{(k)}$ ;
    $r^{(k+1)} \leftarrow r^{(k)} - a_i D_i$;
    $k \leftarrow k + 1$ ;
**end**
```

What OMP does at each iteration is to project the residual onto the atom that is most correlated to it, and then remove the projection from the residual. For this to work well, the atoms must be normalized to have ℓ_2 norm 1.

The seconds stage, instead of being a block descent on D, updates both D and A (again) using rank-one updates as follows. For each atom D_j to be updated, a residual is first computed which does not take the contribution of D_j into account:

$$E_j^{(k+1)} = X - DA + D_j A^j, \tag{6}$$

where A^j is the j-th *row* of A. Then, both D_j and A^j are updated using the first pair of left and right eigenvectors of the SVD decomposition of $E_j^{(k+1)}$.

$$D_j = U_1, \quad A^j = V^1, \quad E_j^{(k+1)} = U \Sigma V, \tag{7}$$

The K-SVD dictionary step is significantly more costly than that of MOD, but usually requires significantly less iterations. MOD, however, is better suited for online dictionary adaptation, as fast approximations of the statistics $AA^\intercal$ (which can be thought of as the Hessian associated to minimizing D) and $XA^\intercal$ can be efficiently updated on a sample to sample basis.

3 Binary Dictionary Learning

The straightforward approach here is to adapt methods such as MOD or K-SVD to the case of binary matrices. One possibility is to simply use those methods as they are, possibly imposing constraints that enforce the entries in the results to be between 0 and 1. An alternative, more akin to works such as those done in Data Mining with the PROXIMUS algorithn, is to work directly on binary operands, using binary operations.

3.1 Coefficients update

First of all, the ℓ_0 norm and the ℓ_1 norms are the same for binary vectors, so both dictionary update methods from MOD and K-SVD can be treated together. (Actually, the ℓ_0 norm is nothing else than the *Hamming weight* or simply *weight* of a binary vector $h(x)$, so we may call it by that name hereafter). We thus consider the problem of minimizing $h(x_j - Da_j)$. Following the OMP idea of removing the closest atom at each iteration. The problem here is that we do not have normalized vectors, so dot product is not a good idea. Instead, we simply search for the atom that is closest in Hamming distance, and remove the candidate atom from the residual using modulo-2 arithmetic (we use $\oplus$ to denote modulo-2 or XOR addition). The algorithm goes as follows:

```
**Data**: vector to encode $x$, dictionary $D$
**Result**: Optimal coefficients for $x$, $a$
Set iteration $k = 0$, residual $r^{(0)} = x$, coefficients $a^{(0)} = 0$;
**while** $hr^{(k)} \geq \epsilon$ **do**
    $h_{\min} \leftarrow\leftarrow \min h(D_i, r^{(k)})$ ;
    **if** $h_{\min} > h(r^{(k)})$ **then**
        break
    **end**
    $i \leftarrow \arg\min h(D_i, r^{(k)})$ ;
    $a_i \leftarrow a_i \oplus 1$ ;
    $r^{(k+1)} \leftarrow r^{(k)} \oplus D_i$;
**end**
```

Besides the differences mentioned, the algorithm also stops when the Hamming distance between the atom that is closest to the current residual is larger than the Hamming weight of the residual itself, as in such case there is no possible improvement.

3.2 Dictionary update – MOD-like case

Here we want to update each atom so that the Hamming weight of the total residual matrix $E = X \oplus DA$ is minimum (note that minus and plus are the same thing in modulo-2 arithmetic). Say we want to update the r-th atom at iteration k. Clearly, the affected columns will only be those for which the coefficients in A corresponding to that atom are non-zero, that is $\{j : A_{rj} \neq 0\}$; let us call this set J_r and n_r its size. What we want is the update Δ that, when added to D_r, minimizes the weight of the residual E in those colums affected by D_r,

$$\Delta = \arg\min_d \textstyle\sum_{j \in J_r} h(E_j^{(k)} \oplus d) \tag{8}$$

$$= \arg\min_d \textstyle\sum_{j \in J_r} (\sum_i E_j^{(k)} + n_r d_i). \tag{9}$$

(note that the summation symbols are carried on using normal addition). The above objective is trivially separable in the elements of d,

$$\begin{aligned} \Delta_i = &\quad \arg\min_{u \in \{0,1\}} \textstyle\sum_j E_{ij}^{(k)} \oplus d_i & (10) \\ = &\ \arg\min_{u \in \{0,1\}} \textstyle\sum_{j \in J_r} E_{ij}^{(k)} + n_r u. & (11) \end{aligned}$$

From (**??**) we cleary obtain

$$\Delta_i = \begin{cases} 0, n_r \leq \sum_j E_{ij}^{(k)} \\ 1, n_r > \sum_j E_{ij}^{(k)} \end{cases} \tag{12}$$

In other words, the i-th element of D_r should be 0 if most of the elements in the rows of E affected by it are 0, and 1 otherwise. This is clearly an optimum solution (note that there could be many optima when n_r is even, in which case we simply choose one).

3.3 Dictionary update – K-SVD-like case

In this case, following the K-SVD concept, we want to obtain the best rank-one approximation to the residual E obtained after removing the contribution of D_r,

$$(D_r, A_r) = \arg\min_{u,v} h(E \oplus uv^\intercal) \tag{13}$$

where

$$E = X_{J_r} \oplus D^{(k)} A_{J_r}^{(k)} \oplus D_r^{(k)} (A_{J_r}^{(k)})^r, \tag{14}$$

where X_{J_r} and A_{J_r} contain only the colums in $J_r = \{j : A_{rj} = 1\}$.

The problem (**??**) is, again, NP-hard, so an approximate solution must be sought. The idea here is to use alternate updates on D_r and A_r, in which case the updates have a simple closed form solution. This is exactly the solution proposed at each step of the PROXIMUS [**?**] algorithm. Say we want to minimize $h(E \oplus uv^\intercal)$ using alternate updates on u and v. For fixed u, we have

$$\sum_{i,j} E \oplus uv^\intercal = \sum_{j : v_j = 1} h(E_j \oplus u).$$

We can solve this separately for each v_j simply by setting $v_j = 1$ is $h(E_j \oplus u) < h(E_j)$ and $v_j = 0$ otherwise. The update on u is identical. In both cases, the objective function can only decrease, the function is bounded and the domain is compact, so that the algorithm must converge to a local minimum. Because the set is finite, convergence is attained in a finite number of iterations. Usually this number is quite small.

4 Initialization

The above algorithms, which I will now call BDL and K-PROX (this is not a definitive name), are quite quite non-convex and nasty. Therefore, a clever initialization is always welcome. Here are some recipes, a few of them taken from the PROXIMUS paper [**?**]. There are issues however with the type of data that we are handling here; when the matrix is very fat such as in typical dictionary learning methods, things such as *graph grow* don't even make a lot of sense[1]. Overall, the best seems to be *neighbors* (not to be confused with nearest neighbors), although partition seems to work quite well with MNIST (it does much better than with neighbors on the demo included in the root directory of the project).

neighbors Each atom d_k is initialized using a randomly drawn sample from X, c_k. Then it d_k is computed as the Hamming average of all samples x so that $c_k^\intercal x > 0$, that is, so that they have at least one dimension where both c_k and x are 1. [2]

graph grow This is actually a variant of the original graph grow algorithm. Given K as the number of atoms to initialize, the method starts by drawing K initial (different) samples from the data set X and associates a *weight* vector w to it, which is initialized to the value of the corresponding random sample. Then it randomly draws one previously unused sample x from X and adds it to the weight vector w^* which satisfies

$$(w^*)^\intercal x / \|w^*\|_1 \geq w^\intercal x / \|w\|_1.$$

This procedure continues until all data samples in X have been used. The final k-th atom is the Hamming average of all the samples added to its corresponding weight vector, that is, $d_k[i] = \lfloor w_k[i]/n_k \rceil$, where n_k is the number of samples from X which were added to w_k. This is a *soft* version of the original graph grow algorithm, where the criterion used to assign a sample to a given atom k is based on the Hamming distance between w and a sample x.

partition Works only when $K \leq M$. This implementation ranks the dimensions of the data matrix X (that is, the number of columns of X) in descending weight order. Then, each atom k is initialized as the Hamming average of all samples in X which have a value of 1 in the k-th ranked dimension.

[1] It would actually be nice to do a serious formal analysis of why such methods fail in this scenario

[2] This again is a lame method for very fat matrices, as there will be so many of such samples, and if noise is present. Some more intelligent criterion must be devised for the dictionary learning case.

random samples D. Given a number K of atoms, each data sample is randomly assigned to one atom. The K atoms are then the average (element-wise majority vote) of all the samples to which they were assigned.[3]

5 Learning variants

I implemented several variants of the above algorithms, most of them just change the order in which some updates are made, or switch the roles of dictionary and coefficients alternatively, so that the method can be used on a broader class of data.

1. Traditional: traditional alternate descent until local convergence
2. Role-switching I: at each iteration, the role of A and D are switched
3. Role-switched learning II: after local convergence (the same as in the first case), the role of A and D are switched and the traditional model is applied again
4. Role switched learning III: like type I but only the dictionary update step is applied (for use with Proximus)

6 Model selection-powered learning variants

The following two methods are actually meant to choose the best from an ensemble of candidate dictionaries learned using one of the previous four methods (referred to as "inner learning method" in the implementation). MDL stands for Minimum Description Length, and is a criterion for model selection (that is, choosing the best model out of a set of competing models for describing some particular data) where the criterion used is, essentially, "which of the models produces a more succint description of the data"; in other words, which model compresses the most.

1. MDL/forward selection: Here begin with an initial dictionary size K_0, train a dictionary, add a few more atoms to obtain K_1, and keep doing this until no further improvement is obtained, that is, until the overall codelength does not decrease by adding new atoms to the dictionary.
2. MDL/backward selection: We begin with a maximum dictionary size K_0, train the dictionary, and then remove the worst atom using some criterion (for example, remove one atom at a time and drop the one which results in the largest decrease in the overall codelength). If

[3] Now that I write this down, I don't think this kind of initialization makes ANY sense!!!!

removing an atom does not result in a better overall codelength, the method stops.

3. MDL/full search: in both forward and backward selection, all or a part of the dictionary used in one iteration is kept and re-adapted after adding or removing atoms from it. In this case, for a range of values of K, a whole new dictionary is trained, and the best one is chosen among them. This method is much slower than the previous two ones (the fastest is forward selection), but tends to provide better results; This tradeoff is open for research.

7 Applications

Here we describe some classical applications from the Dictionary Learning literature, with some references to some of the best results in each case.

7.1 Denoising

Main reference [?]

Denoising is a special case of the more general *signal restoration* problem (other examples are zooming, deblurring, or inpainting – a.k.a. filling erased regions). Here we assume that we want to recover a signal sample $x \in \mathbb{R}^m$ from a noisy observation $z \in \mathbb{R}^m$, both related by

$$z = x + n,$$

where $n \in \mathbb{R}^m$ is a vector of i.i.d. samples from some known distribution, e.g. Gaussian or Poisson. For instance, in the context of Image Processing, z and x are usually vectorized versions of square $\sqrt{m} \times \sqrt{m}$ patches of an image, and what one desires to recover is the whole image I from its corresponding noisy version J.

The method used for denoising an image in the Dictionary Learning framework usually proceeds as follows:

- An initial dictionary $D_0 \in \mathbb{R}^{m \times p}$ is available; this dictionary is learned to efficiently represent a *large* set of patches taken from some public dataset of natural images, usually comprising thousands of images.

- The image to be denoised, $J \in \mathbb{R}^{M \times N}$, is decomposed into $n = (M - \sqrt{m} + 1) \times (N - \sqrt{m} + 1)$ overlapping patches $(z_j : j = 1, \ldots, n)$ that we will arrange for convenience as columns in a matrix $Z \in \mathbb{R}^{m \times n}$

- The samples Z are used to further adapt the initial dictionary and the accompanying sparse coefficients $A \in \mathbb{R}^{p \times n}$; the sparse coding variant during the coefficient update step of this dictionary learning process is the *denoising* formulation, given by

$$a_j^{(t+1)} = \arg\min_a \|a\|_r \quad \text{s.t.} \quad \|z_j - D^{(t)}a\|_2 \leq C\sigma,$$

where σ is the variance of the noise C is a positive constant usually close to 1 and $0 \leq r \leq 1$ is a sparsity-inducing (pseudo)-norm. The dictionary update is the same as described before in (**??**)

- Upon convergence, the *clean image patches* x_j are estimated from the solution (D^*, A^*) as
$$x_j = D^* a_j^*.$$

- Finally, the estimated clean image is obtained by stitching back the estimated clean patches x_j into their respective locations, averaging pixels where intersections between patches occur.

7.2 Inpainting – missing data

Here the task is to fill in missing samples. As we only have binary matrices around, we need an auxiliary mask matrix $H \in \mathbb{R}^{m \times n}$ which will tell us which samples are to be considered missing (value 0) in the data matrix X, regardless of their value in X. The known values in X are indicated with a 1 in the corresponding place in H. The task is to fill in the missing values.

If we have a dictionary D trained to samples similar to X, the idea is the following:

- For each sample x with missing samples specified in a corresponding mask sample h,

- Take the subset of rows from D and x, D_h and x_h for which the corresponding row in h is 1.

- Encode x_h using D_h using the usual encoding scheme (that is, add atoms until no further decrease in the error is obtained); we obtain a vector of coefficients a

- Fill in the subvector of missing values, $x_{\bar{h}}$ ($\bar{h} = 1 - h$) as,

$$x_{\bar{h}} = D_{\bar{h}} a$$

7.3 Classification

Here again we discuss the case of image patches classification, including the special case where the patches are not part of a larger image but pre-cropped and aligned handwritten characters taken from the public datasets MNIST and USPS typically used in character recognition benchmarks.

There are several variants in this case. The one we'll mention here [**?**], which seems quite natural, is a generative one where a dictionary D_c is adapted to efficiently represent samples from each class $c\, in\mathcal{C}$. A new sample is then classified into class c' if its representation under the dictionary $D_{c'}$ is better in some sense than that obtained using the dictionaries trained for the other classes. The method can be summarized as follows:

- A dictionary D_c is adapted to a set of samples X_c known to belong to class c using some known method; this is done for each $c \in \mathcal{C}$.
- In order to classifiy a new sample x, it is sparsely encoded using each of the dictionaries $D_c, c \in \mathcal{C}$, and the corresponding optimum coding cost is used as a score l_c. Any sparse coding variant could be used in principle, but typical choices here are the "basis pursuit" variant,

  $$l_c(x) = \min_a \|z_j - D^{(t)}a\|_2 \quad \text{s.t.} \quad \|a\|_r \leq \tau,$$

  and the Lagrangian variant,

  $$l_c(x) = \min_a \|z_j - D^{(t)}a\|_2 + \tau\|a\|_r.$$

- The sample x is then declared to belong to the class c so that l_c is the smallest. More so than in denoising, the success in this application depends largely on the (critical) parameters τ, the r-norm to be used, and the size of the dictionary p. These critical issues are treated in [**?**].