Albert Au Yeung

Notes on machine learning and A.I.

Matrix Factorization: A Simple Tutorial and Implementation in Python

Apr 23, 2017

(This is an updated version of the article published on my previous personal Website and quuxlab)

There is probably no need to say that there is too much information on the Web nowadays. Search engines help us a little bit. What is better is to have something interesting recommended to us automatically without asking. Indeed, from as simple as a list of the most popular questions and answers on Quora to some more personalized recommendations we received on Amazon, we are usually offered recommendations on the Web.

Recommendations can be generated by a wide range of algorithms. While user-based or item-based collaborative filtering methods are simple and intuitive, matrix factorization techniques are usually more effective because they allow us to discover the latent features underlying the interactions between users and items. Of course, matrix factorization is simply a mathematical tool for playing around with matrices, and is therefore applicable in many scenarios where one would like to find out something hidden under the data.

In this tutorial, we will go through the basic ideas and the mathematics of matrix factorization, and then we will present a simple implementation in Python. We will proceed with the assumption that we are dealing with user ratings (e.g. an integer score from the range of 1 to 5) of items in a recommendation system.

Basic Idea

Just as its name suggests, matrix factorization is to, obviously, factorize a matrix, i.e. to find out two (or more) matrices such that when you multiply them you will get back the original matrix.

As mentioned above, from an application point of view, matrix factorization can be used to discover latent features underlying the interactions between two different kinds of entities. (Of course, you can consider more than two kinds of entities and you will be dealing with tensor factorization, which would be more complicated.) And one obvious application is to predict ratings in collaborative filtering.

In a recommendation system such as Netflix or MovieLens, there is a group of users and a set of items (movies for the above two systems). Given that each users have rated some items in the system, we would like to predict how the users would rate the items that they have not yet rated, such that we can make recommendations to the users. In this case, all the information we have about the existing ratings can be represented in a matrix. Assume now we have 5 users and 10 items, and ratings are integers ranging from 1 to 5, the matrix may look something like this (a hyphen means that the user has not yet rated the movie):

	D1	**D2**	**D3**	**D4**
U1	5	3	-	1
U2	4	-	-	1
U3	1	1	-	5
U4	1	-	-	4
U5	-	1	5	4

Hence, the task of predicting the missing ratings can be considered as filling in the blanks (the hyphens in the matrix) such that the values would be consistent with the existing ratings in the matrix.

The intuition behind using matrix factorization to solve this problem is that there should be some latent features that determine how a user rates an item. For example, two users would give high ratings to a certain movie if they both like the actors or actresses in the movie, or if the movie is an action movie, which is a genre preferred by both users.

Hence, if we can discover these latent features, we should be able to predict a rating with respect to a certain user and a certain item, because the features associated with the user should match with the features associated with the item.

In trying to discover the different features, we also make the assumption that the **number of features** would be smaller than the **number of users** and the **number of items**. It should not be difficult to understand this assumption because clearly it would not be reasonable to assume that each user is associated with a unique feature (although this is not impossible). And anyway if this is the case there would be no point in making recommendations, because each of these users would not be interested in the items rated by other users. Similarly, the same argument applies to the items.

The Maths of Matrix Factorization

Having discussed the intuition behind matrix factorization, we can now go on to work on the mathematics. Firstly, we have a set U of users, and a set D of items. Let R of size $|U| \times |D|$ be the matrix that contains

all the ratings that the users have assigned to the items. Also, we assume that we would like to discover K latent features. Our task, then, is to find two matrics matrices P (of size $|U| \times |K|$) and Q (of size $|D| \times |K|$) such that their product apprioximates |R|:

$$\mathbf{R} \approx \mathbf{P} \times \mathbf{Q}^T = \hat{\mathbf{R}}$$

In this way, each row of $\mathbf{P}$ would represent the strength of the associations between **a user and the features**. Similarly, each row of $\mathbf{Q}$ would represent the strength of the associations between **an item and the features**. To get the prediction of a rating of an item d_j by u_i, we can calculate the dot product of their vectors:

$$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^{k} p_{ik} q_{kj}$$

Now, we have to find a way to obtain $\mathbf{P}$ and $\mathbf{Q}$. One way to approach this problem is the first intialize the two matrices with some values, calculate how **different** their product is to $\mathbf{M}$, and then try to minimize this difference iteratively. Such a method is called gradient descent, aiming at finding a local minimum of the difference.

The difference here, usually called the **error** between the estimated rating and the real rating, can be calculated by the following equation for each user-item pair:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2$$

Here we consider the **squared error** because the estimated rating can be either higher or lower than the real rating.

To minimize the error, we have to know in which direction we have to modify the values of p_{ik} and q_{kj}. In other words, we need to know the gradient at the current values, and therefore we differentiate the above equation with respect to these two variables separately:

$$\frac{\partial}{\partial p_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(q_{kj}) = -2e_{ij}q_{kj}$$

$$\frac{\partial}{\partial q_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(p_{ik}) = -2e_{ij}p_{ik}$$

Having obtained the gradient, we can now formulate the **update rules** for both p_{ik} and q_{kj}:

$$p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + 2\alpha e_{ij} q_{kj}$$

$$q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + 2\alpha e_{ij} p_{ik}$$

Here, α is a constant whose value determines the **rate of approaching the minimum**. Usually we will choose a small value for α, say 0.0002. This is because if we make too large a step towards the minimum we may run into the risk of missing the minimum and end up oscillating around the minimum.

A question might have come to your mind by now: if we find two matrices $\mathbf{P}$ and $\mathbf{Q}$ such that $\mathbf{P} \times \mathbf{Q}$ approximates $\mathbf{R}$, isn't that our predictions of all the unseen ratings will be zeros? In fact, we are not really trying to come up with $\mathbf{P}$ and $\mathbf{Q}$ such that we can reproduce $\mathbf{R}$ exactly. Instead, we will only try to minimise the errors of the **observed** user-item pairs. In other words, if we let T be a set of tuples, each of which is in the form of (u_i, d_j, r_{ij}), such that T contains all the observed user-item pairs together with the associated ratings, we are only trying to minimise every e_{ij} for $(u_i, d_j, r_{ij}) \in T$. (In other words, T is our set of **training data**.) As for the rest of the unknowns, we will be able to determine their values once the associations between the users, items and features have been learnt.

Using the above update rules, we can then iteratively perform the operation until the error converges to its minimum. We can check the overall error as calculated using the following equation and determine when we should stop the process.

$$E = \sum_{(u_i,d_j,r_{ij}) \in T} e_{ij} = \sum_{(u_i,d_j,r_{ij}) \in T} (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2$$

Regularization

The above algorithm is a very basic algorithm for factorizing a matrix. There are a lot of methods to make things look more complicated. A common extension to this basic algorithm is to introduce **regularization** to avoid **overfitting**. This is done by adding a parameter β and modify the squared error as follows:

$$e_{ij}^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2 + \frac{\beta}{2} \sum_{k=1}^{K} (||P||^2 + ||Q||^2)$$

In other words, the new parameter β is used to control the **magnitudes** of the user-feature and item-feature vectors such that P and Q would give a good approximation of R without having to contain large numbers. In practice, β is set to some values in the order of 0.02. The new update rules for this squared error can be obtained by a procedure similar to the one described above. The new update rules are as follows:

$$p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + \alpha (2 e_{ij} q_{kj} - \beta p_{ik})$$

$$q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + \alpha (2 e_{ij} p_{ik} - \beta q_{kj})$$

Adding Biases

When predicting the ratings of users given to items, it is useful to consider how ratings are generated. In the above discussion, we have assumed that ratings are generated based on matching the users preferences on some latent factors and the items' characteristics on the latent factors. Actually, it may also be helpful to consider additional factors here.

For example, we can assume that when a rating is generated, some **biases** may also contribute to the ratings. In particular, every user may have his or her own bias, meaning that he or she may tend to rate items higher or lower than the others. In movie ratings, if a user is a serious movie watcher, he or she may tend to give lower ratings, when compared to another user who generally enjoys movies as long as they are not too boring. A similar idea can also be applied to the items being rated.

Hence, in the equal of predicting a rating, we can also add these biases in order to better model how a rating is generated:

$$\hat{r}_{ij} = b + bu_i + bd_j + \sum_{k=1}^{k} p_{ik} q_{kj}$$

where b is the global bias (which can be easily estimated by using the mean of all ratings), bu_i is the bias of user i, and bd_j is te bias of item j.

Using the same steps mentioned above, we can derive the update rules for the user biases and item biases easily:

$$bu'_i = bu_i + \alpha \times (e_{ij} - \beta bu_i)$$

$$bd'_j = bd_j + \alpha \times (e_{ij} - \beta bd_j)$$

In practice, the process of factorization will converge faster if biases are included in the model.

A Simple Implementation in Python

Once we have derived the update rules as described above, it actually becomes very straightforward to implement the algorithm. The following is a function that implements the algorithm in Python using the **stochastic gradient descent** algorithm. Note that this implementation requires the **Numpy** module.

```python
import numpy as np

class MF():

    def __init__(self, R, K, alpha, beta, iterations):
        """
        Perform matrix factorization to predict empty
        entries in a matrix.

        Arguments
        - R (ndarray)   : user-item rating matrix
        - K (int)       : number of latent dimensions
        - alpha (float) : learning rate
        - beta (float)  : regularization parameter
        """

        self.R = R
        self.num_users, self.num_items = R.shape
        self.K = K
        self.alpha = alpha
        self.beta = beta
        self.iterations = iterations

    def train(self):
        # Initialize user and item latent feature matrice
        self.P = np.random.normal(scale=1./self.K, size=(self.num_users, sel
        self.Q = np.random.normal(scale=1./self.K, size=(self.num_items, sel

        # Initialize the biases
        self.b_u = np.zeros(self.num_users)
        self.b_i = np.zeros(self.num_items)
        self.b = np.mean(self.R[np.where(self.R != 0)])

        # Create a list of training samples
        self.samples = [
            (i, j, self.R[i, j])
            for i in range(self.num_users)
            for j in range(self.num_items)
            if self.R[i, j] > 0
        ]

        # Perform stochastic gradient descent for number of iterations
        training_process = []
        for i in range(self.iterations):
```

```
            np.random.shuffle(self.samples)
            self.sgd()
            mse = self.mse()
            training_process.append((i, mse))
            if (i+1) % 10 == 0:
                print("Iteration: %d ; error = %.4f" % (i+1, mse))

        return training_process

    def mse(self):
        """
        A function to compute the total mean square error
        """
        xs, ys = self.R.nonzero()
        predicted = self.full_matrix()
        error = 0
        for x, y in zip(xs, ys):
            error += pow(self.R[x, y] - predicted[x, y], 2)
        return np.sqrt(error)

    def sgd(self):
        """
        Perform stochastic graident descent
        """
        for i, j, r in self.samples:
            # Computer prediction and error
            prediction = self.get_rating(i, j)
            e = (r - prediction)

            # Update biases
            self.b_u[i] += self.alpha * (e - self.beta * self.b_u[i])
            self.b_i[j] += self.alpha * (e - self.beta * self.b_i[j])

            # Update user and item latent feature matrices
            self.P[i, :] += self.alpha * (e * self.Q[j, :] - self.beta * sel
            self.Q[j, :] += self.alpha * (e * self.P[i, :] - self.beta * sel

    def get_rating(self, i, j):
        """
        Get the predicted rating of user i and item j
        """
        prediction = self.b + self.b_u[i] + self.b_i[j] + self.P[i, :].dot(s
        return prediction
```

```
    def full_matrix(self):
        """
        Computer the full matrix using the resultant biases, P and Q
        """
        return self.b + self.b_u[:,np.newaxis] + self.b_i[np.newaxis:,] + se
```

We can try to apply it to our example mentioned above and see what we would get. Below is a code snippet in Python for running the example.

```
R = np.array([
    [5, 3, 0, 1],
    [4, 0, 0, 1],
    [1, 1, 0, 5],
    [1, 0, 0, 4],
    [0, 1, 5, 4],
])

mf = MF(R, K=2, alpha=0.1, beta=0.01, iterations=20)
```

And the matrix obtained from the above process would look something like this:

```
[[ 4.99  3.    3.34  1.01]
 [ 4.    3.18  2.98  1.01]
 [ 1.02  0.96  5.54  4.97]
 [ 1.    0.6   4.78  3.97]
 [ 1.53  1.05  4.94  4.03]]
```

We can see that for existing ratings we have the approximations very close to the true values, and we also get some 'predictions' of the unknown values. In this simple example, we can easily see that $U1$ and $U2$ have similar taste and they both rated $D1$ and $D2$ high, while the rest of the users preferred $D3$, $D4$ and $D5$. When the number of features (K in the Python code) is 2, the algorithm is able to associate the users and items to two different features, and the predictions also follow these associations. For example, we can see that the predicted rating of $U4$ on $D3$ is 4.78, because $U4$ and $U5$ both rated $D4$ high.

Further Information

We have discussed the intuitive meaning of the technique of matrix factorization and its use in collaborative filtering. In fact, there are many different extensions to the above technique. An important extension is the requirement that all the elements of the factor matrices $\mathbf{P}$ and $\mathbf{Q}$ in the above example) should be **non-negative**. In this case it is called non-negative matrix factorization (NMF). One advantage of NMF is that it results in intuitive meanings of the resultant matrices. Since no elements are negative, the process of multiplying the resultant matrices to get back the original matrix would not involve subtraction, and can be considered as a process of generating the original data by linear combinations of the latent features.

Source Code

An example can be found at this IPython notebok. It is also available at my Github account in this repository.

References

There have been quite a lot of references on matrix factorization. Below are some of the related papers:

- Gábor Takács et al (2008). Matrix factorization and neighbor based algorithms for the Netflix prize problemIn: Proceedings of the 2008 ACM Conference on Recommender Systems, Lausanne, Switzerland, October 23 - 25, 267-274.
- Patrick Ott (2008). Incremental Matrix Factorization for Collaborative Filtering. Science, Technology and Design 01/2008, Anhalt University of Applied Sciences.
- Daniel D. Lee and H. Sebastian Seung (2001). Algorithms for Non-negative Matrix Factorization. Advances in Neural Information Processing Systems 13: Proceedings of the 2000 Conference. MIT Press. pp. 556–562.
- Daniel D. Lee and H. Sebastian Seung (1999). Learning the parts of objects by non-negative matrix factorization. Nature, Vol. 401, No. 6755. (21 October 1999), pp. 788-791.

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

Filipe Silva • 7 months ago

Hi Albert,

First of all, thumbs up for the way you presented this topic, very well organised where the concepts build seamlessly on previous ones.

Besides that, are not the equations for the biases missing the factor 2?

Ex: bu′i=bui+α×(eij−βbui) -> bu′i=bui+α×(2eij−βbui)

Cheers.

1 • Reply • Share ›

Jing Li ➚ Filipe Silva • 3 months ago

I agree, also I think in the line below:
self.b_u[i] += self.alpha * (e - self.beta * self.b_u[i])
should be
self.b_u[i] += self.alpha * (2*e - self.beta * self.b_u[i])

• Reply • Share ›

VC • 7 months ago

I'm using a larger matrix now and I start getting null values in the 'prediction'. All values in my matrix are >=0. Any tips on how to fix this?

• Reply • Share ›

VC ➚ VC • 7 months ago

Also got the error: RuntimeWarning: overflow encountered in multiply ... Q[j, :] is filled with "inf"

• Reply • Share ›

Albert Au Yeung Mod ➚ VC • 7 months ago

How many iterations are you running? What's the maximum value in your matrix?

• Reply • Share ›

VC ➚ Albert Au Yeung • 7 months ago

Max value in the matrix is 5 ... for testing purposes, I was only using 2 iterations.

• Reply • Share ›

Geoffrey Anderson • 9 months ago

Might try partitioning dataset into training, dev, and test datasets. Within training dataset, we could hide some known ratings from the learning algo. Then we could see what MSE value did

we get on training dataset. Could train more iterations until bias error is low. THen look at dev dataset and tune the regularization parameters on that to reduce variance error. When MSE is low as possible on dev dataset by tuning the parameters, then we can finally estimate error best by using the test dataset. This might be easier not to code directly in python all the way but to use tensorflow lib, with python as host language. It does the calculus automatically and parallelizes since you wanted more speed. I would start speeding it up only after doing the dataset partitioning and model development. Code vectorization in numpy would be a good first step followed by parallelization in that order if you like writing fast python from scratch for your purposes. In any case the dataset partitioning is tough on so few examples, like 12 in this case. Better in the hundreds or thousands of course for the described methodology. THanks for writing this nice article.

Reply • Share ›

IvanHerr • 10 months ago

last line is not debugged, it should read:

return self.b + self.b_u[:,np.newaxis] + self.b_i[np.newaxis,:] + self.P.dot(self.Q.T)

instead of

return mf.b + mf.b_u[:,np.newaxis] + mf.b_i[np.newaxis:,] + mf.P.dot(mf.Q.T)

btw, the implementation is good, as it is very straightforward to understand, but veeery slow. No possibility of getting a more vectorized implementation?

Reply • Share ›

Albert Au Yeung Mod ➚ IvanHerr • 10 months ago

Thanks for noticing the bug. I have already updated it. For sure this is not an efficient implementation. I don't have an idea now as to how to further speed up this specifically though. Will do some research.

Reply • Share ›

Geoffrey Anderson ➚ Albert Au Yeung • 9 months ago

Is vectorization and parallelization of this python code, or tensorflow rewrite, more to your interests? Both can run much faster but your code provides a nice correctness reference probably so it's not wasted at all.

Reply • Share ›

Albert Au Yeung Mod ➚ Geoffrey Anderson • 9 months ago

Thanks for your message. Yes, the article was written to show the basic idea of matrix factorization. I think in recent years people have used neural network and embeddings to generate recommendations. Definitely something that I would like to learn more next.

Reply • Share ›

ALSO ON **ALBERT AU YEUNG**

Deploying Jupyter in Ubuntu with Nginx and Supervisor

13 comments • 2 years ago


Michel Wilhelm — Thanks a lot!

Performing Sequence Labelling using CRF in Python

51 comments • 2 years ago


Abhishek Mamidi — How can we perform Named Entity Recognition using SVM? How to

Back to Top