

# Topics in Applied Optimization (IIIT, Hyderabad, India)

## Jupyter Notebook Assignment-1

**Instructor:** Dr. Pawan Kumar (IIIT, H)  
[\(https://faculty.iiit.ac.in/~pawan.kumar/\)](https://faculty.iiit.ac.in/~pawan.kumar/)

If you are not familiar with Jupyter notebook, before proceeding further, please go and watch this video:

<https://www.youtube.com/watch?v=HW29067qVWk>

### Regarding assignments

**Deadline: 13 October 2023**

- all the assignment is to be done in this notebook itself
- any proof etc can be done on paper, and image is to be inserted in this notebook
- save this notebook with your roll number and upload it in moodle
- basic familiarity with python is required, brush up if necessary
- you are not allowed to use any existing library for gradient methods; this defeats the purpose of this assignment
- sample output is in the zip file of assignment
- if there are any doubt, then raise it in course moodle site, it may help others
- if you find typo, raise this issue in moodle promptly!
- please avoid copying from others, there may be oral exam to test your knowledge

### Implementation of Recommender System Using Latent Factor Model in Python

Please refer to the class slides for more detail. Also, for full detail, please refer the following main reference:

**[1] Y. Koren, R. Bell, and C. Volinsky, Matrix Factorization Techniques for Recommender Systems, Computer Archive, Volume 42, Issue 8, 2009**

We first load the necessary libraries:

1. matplotlib: needed for plotting figures and

2. numpy needed for doing math

```
In [ ]: %matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
```

## Now we write the main functions

We write everything inside a class LF, which stands for latent factor. We need the following functions:

1. main
2. train
3. mse
4. sgd
5. predict
6. full\_matrix

## Short Tutorial on Recommender Systems

Please see lecture slides for more detail.

### Latent Factor Model

In a recommender systems, we are given an incomplete ratings matrix say  $R$ . The  $(i, j)$  entry of this matrix is denoted by  $r_{ij}$ .

- We believe that there are dependencies in the matrix, i.e., the matrix is low rank, hence it can be written as a product of low rank matrices.
- Even if the matrix is not strictly low rank, we believe that certain features are more important than others. In other words, if the ratings matrix were full, then after doing SVD, we get

$$R = USV^T$$

some of the singular values were very small, hence the matrix can be approximated well with truncated SVD. Let us consider the first  $k$  largest singular vectors, then the truncated SVD is

$$R \approx U_k S_k V_k.$$

- See IPSC notes for more detail on how to compute SVD or truncated SVD

- For a recommendation problem, for example, movie recommendation or news recommendation or product recommendation, etc, we believe that certain latent features are most important and play a major role in ratings.
- For movie recommendation, for example, latent factors could be comedy, adventure, horror, etc. These features are called latent, because the given data, which is ratings matrix does not explicitly tell about these.
- For recommendation systems, we cannot do SVD, because the matrix  $R$  is incomplete! Obviously, we can't treat the missing entries to be zero! Hence, instead of doing SVD, we will consider an error function that considers only the ratings that are given. Let us assume that the ratings matrix can be modelled as a latent factors, i.e., suppose that

$$R = PQ, \quad P \in \mathbb{R}^{n \times k}, Q \in \mathbb{R}^{k \times n},$$

then the given rating

$$r_{ij} = p_{i*}q_{*j},$$

where  $p_{i*}$  denotes the  $i$ th row of  $P$ , and  $q_{*j}$  denotes the  $j$ th column of  $Q$ .

## Optimization Model

- As mentioned above, we cannot directly do SVD, however, we can indeed create a **loss function** as follows:

$$\mathcal{L}(p_{1*}, p_{2*}, \dots, p_{n*}, q_{*1}, q_{*2}, \dots, q_{*n}) = \sum_{(i,j) \in K} (r_{ij} - p_{i*}q_{*j}).$$

- Note above that we see loss function as a function of the variables  $p_{1*}, p_{2*}, \dots, p_{n*}$ , which are row vectors of  $P$ , and as the column vectors of  $Q$ , which are  $q_{*1}, q_{*2}, \dots, q_{*n}$ . This allows us to **vectorize**. Why vectorize?
- In machine learning the weights and their combinations with the given data creates a model. In this case, the model is  $PQ$ . Since the  $k$ , is a hyperparameter, it is likely that one may choose  $P$  and  $Q$  to have too many columns, i.e., too many weights are used, this will lead to a large model, and hence leading to **overfitting**. If you are not familiar with overfitting, then pause here, and see the video here:

<https://www.youtube.com/watch?v=u73PU6QwI1I>

- What did you learn from video?
- What is termed as model?
- What is the meaning of model being large?
- What is the difference between linear and logistic regression?
- What is underfitting?
- What is a bias?
- How do you characterize bias and variance using terms overfitting or underfitting?

- A well known way to avoid overfitting is to do regularizations, by penalizing **large model size**. Let us modify our loss function as follows:

$$\mathcal{L}(\cdot) = \sum_{(i,j) \in K} (r_{ij} - p_{i*}q_{*j})^2 + \gamma/2(\|P\|_F^2 + \|Q\|_F^2)$$

- We have added the term  $\gamma/2(\|P\|_F^2 + \|Q\|_F^2)$ . It is called **regularization term**
- The parameter  $\gamma$  is called the **regularization parameter**
- High  $\gamma$  would mean that we don't want big model size; this may help prevent overfitting
- Let  $e_{ij}$  denote the error corresponding to each  $(ij)$  term,

$$e_{ij} = r_{ij} - \sum_{s=1}^k p_{is}q_{sj}$$

then let us define the new error term  $\tilde{e}$

$$\tilde{e}_{ij} = e_{ij}^2 + \gamma/2(\|P\|_F^2 + \|Q\|_F^2),$$

that is,

$$\tilde{e}_{ij} = (r_{ij} - \sum_{s=1}^k p_{is}q_{sj})^2 + \gamma/2(\|P\|_F^2 + \|Q\|_F^2)$$

- Note that

$$\mathcal{L}(\cdot) = \sum_{(i,j) \in \mathcal{K}} \tilde{e}_{ij},$$

where  $\mathcal{K}$  is the set of all indices for which ratings  $r_{ij}$  are available.

## Gradient Method Versus Stochastic Gradient Methods

Following notes are taken from [https://en.wikipedia.org/wiki/Stochastic\\_gradient\\_descent](https://en.wikipedia.org/wiki/Stochastic_gradient_descent)

For large dataset,  $k$  and  $n$  tend to be large, and computing gradient fully becomes too demanding. In machine learning or statistical estimation, we are usually required to find minimum of the loss functions of the form

$$Q(w) = 1/n \sum_{j=1}^n Q_j(w)$$

That is loss function as "additive" decomposition. In other words, the loss function is a sum of the loss function where the sum is over the data samples. If  $n$  is large, then computing full gradient is computationally demanding. However, if we indeed decide to run the gradient method. Then we recall the following steps:

1. Take initial random weight (possibly random and normally distributed between 0 and 1):

$$w^0$$

$$2. w^{i+1} = w^i - \alpha \nabla Q(w^i), \quad i = 0, \dots$$

Here the gradient  $\nabla Q$  is the full gradient in the sense that

$$\nabla Q(w) = \sum_{i=1}^n \nabla_w Q_i(w). \quad (1)$$

Here

$$\nabla_w Q_i(w) = \begin{bmatrix} \frac{\partial Q_i(w)}{\partial w_1} \\ \frac{\partial Q_i(w)}{\partial w_2} \\ \vdots \\ \frac{\partial Q_i(w)}{\partial w_n} \end{bmatrix} \quad (2)$$

## Stochastic gradient descent method

Stochastic gradient method, in short, SGD is a method (not necessarily descent due to stochasticity, but in expectation is usually a descent), where the gradient is computed partially. For example, instead of summing for all  $i = 1 \dots n$ , we can take a random sample, say  $j_1$ , and do the update only for this sample as follows:

$$\nabla Q(w) = \nabla_w Q_{j_1}(w),$$

that is we have picked just one of the term i.e.,  $j_1$ th term from (1). Then the SGD update is:

$$w^{i+1} = w^i - \alpha \nabla Q_{j_1}(w^i), \quad i = 0, \dots$$

Note that we could have taken few more terms in the sum, and would have updated for a batch. This is called, batched stochastic gradient descent. It sounds incorrect, isn't it? Why such method may converge? With some basic assumption on the loss function, we can prove some convergence in expectation. More on convergence theory in class!

**Question-0:** Compute the full gradient of the loss function for recommendation system

**Answer:**

In the following, we derive a stochastic gradient for the loss function for recommender loss.

## Gradient Method for Optimization Problem

- We want to use gradient method to minimize the error. An update in gradient method is given as follows

$$w^{i+1} = w^i - \nabla \mathcal{L}(w^i)$$

- To use gradient method, we need to compute gradient. If we see  $\mathcal{L}$  as a function of  $p'_{ik}$ s and  $q'_{ki}$ s, then

$$\frac{\partial \mathcal{L}}{\partial p_{is}} = 2e_{is} \frac{\partial e_{is}}{\partial p_{is}} + \gamma p_{is} = -2e_{is}q_{sj} + \gamma p_{is}$$

Similarly,

$$\frac{\partial \mathcal{L}}{\partial q_{sj}} = 2e_{sj} \frac{\partial e_{sj}}{\partial q_{sj}} + \gamma q_{sj} = -2e_{sj}p_{js} + \gamma q_{sj},$$

- The gradient vector  $\nabla \mathcal{L}$  is given by

$$\nabla \mathcal{L} = \left( \frac{\partial \mathcal{L}}{\partial p_{11}}, \dots, \frac{\partial \mathcal{L}}{\partial p_{nk}}, \frac{\partial \mathcal{L}}{\partial q_{11}}, \dots, \frac{\partial \mathcal{L}}{\partial p_{kn}} \right)$$

- The weights  $p_{is}$  can be updated as

$$p_{is} = p_{is} + \alpha(2e_{ij}q_{sj} - \gamma p_{is})$$

- The weights  $p_{sj}$  can be updated as

$$q_{sj} = q_{sj} + \alpha(2e_{ij}p_{js} - \gamma q_{sj})$$

- We can vectorize  $s$ . Finally, the vectorized form of the update looks like:

$$(p_{i+1*}, q_{*j+1})^T = (p_{i*}, q_{*j})^T + \alpha((2e_{ij}q_{*j} - \gamma p_{i*}), (2e_{ij}p_{j*} - \gamma q_{*j}))^T$$

- The error at  $(p_{i*}, q_{*j})^T$  needed above can be computed by

$$e_{ij} = r_{ij} - \text{prediction at } (i, j)$$

Here prediction at  $(i, j)$  can be computed by making a call to predict() function below.

**Question-1:** Why the error is computed this way?

**Answer:** Either use latex (preferred) or write it on paper, take pic, and insert it here. See the video on jupyter above on how to paste images.

**Question-2:** Are the updates for  $p_{i*}$  and  $q_{*j}$  stochastic? Justify.

**Answer:** Write your answer here.

## Adding Bias

It is often observed in practice that adding a bias term helps. Let us add the bias term  $b$  to the prediction function. The bias term is computed as the mean of the ratings matrix. See the predict function below to see how bias is added.

**Question-2b:** Read the main reference paper and justify why bias is added. Note in this notebook we add global bias. Which other bias term was suggested in the paper?

**Answer:**

In [ ]: `class LF():`

```
    def __init__(self, R, K, alpha, gamma, iterations):
        """
        Parameters:
        - R (ndarray) : user-item rating matrix
        - K (int)      : number of latent dimensions
        - alpha (float): learning rate
        - gamma (float): regularization parameter
        """

        self.R = R
        self.num_users, self.num_items = R.shape
        self.K = K
        self.alpha = alpha
        self.gamma = gamma
        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, self.K))
        self.Q = np.random.normal(scale=1./self.K, size=(self.num_items, self.K))

        # Compute the global bias
        # For Question-6, you may want to initialize additional bias terms here
        self.b = np.mean(self.R[np.where(self.R != 0)])
        #self.b_i = TODO
        #self.b_u = TODO

        # 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("SGD Iteration: %d ; error = %.4f" % (i+1, mse))

    return training_process

def mse(self):
    """
    MSE: Compute 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):
    """
    SGD: Stochastic Gradient Descent
    """
    for i, j, r in self.samples:
        # Compute prediction and error
        prediction = TODO
        e = TODO

        # Keep a copy of row of P. It will be needed for update
        P_i = self.P[i, :][:]

        # Update user and item latent feature matrices
        self.P[i, :] += TODO
        self.Q[j, :] += TODO

def gd(self):
    """
    GD: Gradient Descent
    Replace self.sample below to full batch for gradient descent
    Feel free to remove everything, and implement from scratch
    """
    for i, j, r in TODO:
        # Compute prediction and error
        prediction = TODO
        e = TODO

        # Create copy of row of P since we need to update it but use older value
        P_i = self.P[i, :][:]

        # Update user and item latent feature matrices
        self.P[i, :] += TODO
        self.Q[j, :] += TODO

def predict(self, i, j):
    """
    Prediction: Predicted the rating of user i and item j
    """
    prediction = self.b + self.P[i, :].dot(self.Q[j, :].T)

```

```

    return prediction

def full_matrix(self):
    """
    The rating matrix using the biases P and Q
    """
    return lf.b + lf.P.dot(lf.Q.T)

```

```

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

lf = LF(R, K=2, alpha=0.1, gamma=0.01, iterations=100)
training_process = lf.train()
print()
print("P x Q:")
print(lf.full_matrix())
print()
print("Global bias:")
print(lf.b)
print()

```

## More questions

**Question-3:** Implement the full gradient descent method

**Answer:** Complete gd function above

**Question-4:** Vectorize the updates of p and q, by vectorizing s (detailed above)

**Answer:** Fill the TODO in code above

**Using the code below, answer the following:**

```

In [ ]: x = [x for x, y in training_process]
y = [y for x, y in training_process]
plt.figure(figsize=(16,4))
plt.plot(x, y)
plt.xticks(x, x)
plt.xlabel("Iterations")
plt.ylabel("Mean Square Error")
plt.grid(axis="y")

```

**Question-5:** Plot the MSE versus iterations for  $\alpha = 1, \gamma = 0.01$  and 100 iterations of SGD

**Answer:** Put your figure here

**Question-6:** Show the plot for 50 iterations,  $\alpha = 1, \gamma = 0.01$  of SGD without bias, i.e.,  $b = 0$

**Answer:** Paste your output and figures here or below this cell

**Question-7:** Repeat above with bias

**Answer:** Paste your output and figures here or below this cell

**Question-8:** Show plots with values of regularization parameters to be 1, 0.1, 0.01, 0.001 and for 50 iterations, and  $\alpha = 1, \gamma = 0.01$

**Answer:** Paste all your plots here or below this cell.

**Question-9:** Prove that the loss function is not convex. Is the loss function differentiable?

**Answer:** Write your answer here.

## Bonus Question

**Question-10:** In the reference paper [1] above, additional bias terms are recommended, implement it

**Answer:** Put your modified function here or below this cell.

**Question-11:** Download the netflix ratings matrix, and run your algorithm.

**Answer:** Put your MSE versus iterations here.

## Thats All Folks! Further Readings or Infomation

<https://www.coursera.org/learn/networks-illustrated/lecture/8GPZT/netflix-timeline>

# Keep Recommending!