《数据科学与工程算法》项目报告

报告题目：基于矩阵分解的推荐系统实现

姓　　名：温兆和

学　　号：10205501432

完成日期：2023.06.11

摘要 [中文]：

矩阵分解是将原始矩阵拆分为两个或多个矩阵的乘积，原始矩阵是可以有数据缺失的。所以，在推荐系统中，我们也可以把有较多空缺的用户-产品评分矩阵分解为两个矩阵，再把这两个矩阵相乘，得到的新矩阵就带有用户对某个没有评分过的产品的预测评分值。我们可以通过梯度下降的方法对带有缺失值的矩阵进行分解，还可以在基本的梯度下降法的基础上进行优化，如：正则化、随机梯度下降、动量梯度下降等。

Abstract [English]

Matrix factorization (MF) is the process of decomposing an original matrix into the product of two or more matrices, and the original matrix can have missing data. Therefore, in recommendation systems, we can also decompose a user-item rating matrix with many missing values into two matrices, and then multiply these two matrices to obtain a new matrix with predicted rating values for items that have not been rated by users. We can perform matrix factorization on matrices with missing values using gradient descent methods, and we can further optimize the basic gradient descent approach with techniques such as regularization, stochastic gradient descent, and momentum gradient descent.

一、 项目概述（阐明项目的科学价值与相关研究工作，描述项目主要内容）

1.1 项目科学价值

基于矩阵分解(MF)算法，我们可以补全用户-产品评分矩阵中的缺失值来预测用户对某个没有评过分的商品的可能评分，由此向用户推荐其可能会喜欢的产品。

1.2 相关研究工作

我们主要使用梯度下降法最小化重构误差来得到矩阵分解的结果。目前，有很多针对传统梯度下降的优化，这些优化要么降低重构误差，要么缩短训练时间，要么能同时得到以上两种效果。如：随机梯度下降和小批量梯度下降可以在保证重构误差的前提下加快训练速度，正则化可以缓解模型中的过拟合现象。

1.3 项目主要内容

在本次实验中，我们在 kaggle 上[1]找到了一个包含用户-产品评分信息的数据集。我们首先将其中的用户-产品评分信息转化为用户-产品评分矩阵，用梯度下降法分解这个矩阵并对其进行重构。接着，针对原有用户-产品评分矩阵中的空缺项，我们通过重构矩阵中对应项的值预测该用户对相应产品可能评多少分，并由此向用户推荐其可能喜欢但没有评价过的产品。最后，我们用正则化、随机梯度下降、半正定矩阵分解、动量梯度下降等方法试图优化原有算法，比较不同方法间的训练时间和重构误差。

二、 问题定义（提供问题定义的语言描述与数学形式）

2.1 语言描述

给定带有空缺的用户-产品评分矩阵，将其分解为两个矩阵的乘积，并将这两个矩阵相乘来重构用户-产品评分矩阵，使得重构评分矩阵后，原评分矩阵中的非空缺项与重构后的评分矩阵中对应的项差距尽可能小；而对于原评分矩阵中的空缺项，重构后的评分矩阵中的对应项应能够较好地预测用户对相应产品的可能评分。

2.2 数学形式

给定用户集合U和项目集合D，以及由用户对项目的评分所构成的评分矩阵$R \in \mathbb{R}^{|U|\times|D|}$，我们要找到两个矩阵$P \in \mathbb{R}^{K\times|U|}$和$Q \in \mathbb{R}^{K\times|D|}$，使得$R \approx P^T Q = \hat{R}$。其中$K$是潜在特征的维数。

三、 方法（问题解决步骤和实现细节）

3.1 数据集的预处理

由于数据集比较大，我们在读取整个数据集后从中抽取 1000 条用户-产品评分信息，并去掉我们本次实验中并不会用到的列（时间戳）。

```
DF = pd.read_csv("ratings_Electronics.csv")
whole_set = range(len(DF))
not_in_smp = random.sample(whole_set,len(DF)-1000)
```

[1] https://www.kaggle.com/datasets/saurav9786/amazon-product-reviews

```
DF_sampled = DF.drop(index = not_in_smp)
DF_processed = DF_sampled.drop(columns = ['Time_Stamp'])
DF_processed.head()
```

Out[4]:

	User_ID	Product_ID	Rating
375	A33DH1NN4OOKX3	972683275	2
4552	AO2UCX32NHYUM	8899368872	1
5608	A33FFM2YST8X93	9966377727	3
6670	A1FKN6YBRF87RB	9985502809	5
6938	A1VHBE3E5WU754	9985602560	2

我们可以看到，处理之后的数据集中，每一行有用户、产品和该用户对该铲平的评分构成。接着，我们由此生成用户集合、产品集合和用户-产品评分矩阵。在评分矩阵中，空缺项被置为-1。

```
Users = DF_processed['User_ID'].drop_duplicates().values.tolist()
Products =
DF_processed['Product_ID'].drop_duplicates().values.tolist()
user_rating_table = np.full((len(Users), len(Products)), -1, dtype=int)
for i in DF_processed.index.values:

user_rating_table[Users.index(DF_processed['User_ID'][i])][Products.i
ndex(DF_processed['Product_ID'][i])] = DF_processed['Rating'][i]
print(user_rating_table)
```

```
[[ 2 -1 -1 ... -1 -1 -1]
 [-1  1 -1 ... -1 -1 -1]
 [-1 -1  3 ... -1 -1 -1]
 ...
 [-1 -1 -1 ...  4 -1 -1]
 [-1 -1 -1 ... -1  2 -1]
 [-1 -1 -1 ... -1 -1  2]]
```

可以看到，我们得到了一个非常稀疏的评分矩阵。至此，数据集的预处理完成了。

3.2 Basic Matrix Factorization

这里，我们先用最基本的梯度下降来实现一个矩阵分解算法。我们首先随机生成两个相应尺寸的因式矩阵P和Q，再通过一次一次地迭代来最小化损失函数，直到后一次迭代后生成的重构矩阵与前一次迭代生成的重构矩阵之差的 F-范数小于 1 或者迭代次数达到 1500 次。每一次迭代的更新公式如下：

$$p_{uj}^{(t+1)} = p_{uj}^{(t)} + \epsilon \sum_{i:R_{ui}>0} e_{ui}^{(t)} \, q_{ji}^{(t)}$$

$$q_{ji}^{(t+1)} = q_{ji}^{(t)} + \epsilon \sum_{u:R_{ui}>0} e_{ui}^{(t)} \, p_{uj}^{(t)}$$

其中，ϵ是学习率，$e_{ui}^{(t)} = R_{ui}^{(t)} - \hat{R}_{ui}^{(t)}$。

```
def basicMF(K,epsilon,iteration):
    start=time.perf_counter()
    P = np.random.rand(len(Users), K)#.astype(type('float', (float,),
{}))
    Q = np.random.rand(K,len(Products))#.astype(type('float', (float,),
{}))
    RMSE_Seq = []
    MAE_Seq = []
    Prev_U = np.dot(P,Q)
    for I in range (iteration):
        for i in range(len(user_rating_table)):
            for j in range(len(user_rating_table[i])):
                eij=user_rating_table[i][j]-np.dot(P[i,:],Q[:,j])
                for k in range(K):
                    if user_rating_table[i][j]>0:
                        P[i][k]=P[i][k]+epsilon*(2*eij*Q[k][j])
                        Q[k][j]=Q[k][j]+epsilon*(2*eij*P[i][k])
        U = np.dot(P,Q)
        F_value = np.linalg.norm(U-Prev_U)
        Prev_U = U
        RMSE,MAE = RMSEandMAE(user_rating_table,U)
        RMSE_Seq.append(RMSE)
        MAE_Seq.append(MAE)
        print("-------------")
        print("iteration_time")
        print(I)
        print("RMSE")
        print(RMSE)
        print("MAE")
        print(MAE)
        if(F_value<=1):
            break
    end=time.perf_counter()
    training_time = end - start
    return P,Q,RMSE_Seq,MAE_Seq,training_time
```

3.3 MF with normalization

在数据量小而模型容量比较大的情况下，很容易出现过拟合的情况。在迭代更新公式中添加正则化项可以给模型提供一些先验信息以避免过拟合。所以，更新公式改为：

$$p_{uj}^{(t+1)} = p_{uj}^{(t)} + \epsilon\left(\sum_{i:R_{ui}>0} e_{ui}^{(t)}\, q_{ji}^{(t)} - \tau p_{uj}^{(t)}\right)$$

$$q_{ji}^{(t+1)} = q_{ji}^{(t)} + \epsilon\left(\sum_{u:R_{ui}>0} e_{ui}^{(t)}\, p_{uj}^{(t)} - \tau q_{ji}^{(t)}\right)$$

其中，ϵ是学习率，$e_{ui}^{(t)} = R_{ui}^{(t)} - \hat{R}_{ui}^{(t)}$，$\tau$是正则化项的系数。

```
def MF_with_normalization(K,epsilon,iteration,lamda):
    start=time.perf_counter()
    P = np.random.rand(len(Users), K)#.astype(type('float', (float,),
{}))
    Q = np.random.rand(K,len(Products))#.astype(type('float', (float,),
{}))
    RMSE_Seq = []
    MAE_Seq = []
    Prev_U = np.dot(P,Q)
    for I in range (iteration):
        for i in range(len(user_rating_table)):
            for j in range(len(user_rating_table[i])):
                eij=user_rating_table[i][j]-np.dot(P[i,:],Q[:,j])
                for k in range(K):
                    if user_rating_table[i][j]>0:

P[i][k]=P[i][k]+epsilon*(2*eij*Q[k][j]-lamda*P[i][k])

Q[k][j]=Q[k][j]+epsilon*(2*eij*P[i][k]-lamda*Q[k][j])
        U = np.dot(P,Q)
        F_value = np.linalg.norm(U-Prev_U)
        Prev_U = U
        RMSE,MAE = RMSEandMAE(user_rating_table,U)
        RMSE_Seq.append(RMSE)
        MAE_Seq.append(MAE)
        print("-------------")
        print("iteration_time")
        print(I)
        print("RMSE")
        print(RMSE)
        print("MAE")
        print(MAE)
        if(F_value<=1):
```

```
            break
    end=time.perf_counter()
    training_time = end - start
    return P,Q,RMSE_Seq,MAE_Seq,training_time
```

3.3 随机梯度下降

在随机梯度下降中，每次迭代仅选择一个样本放进更新公式，而不是让所有样本都参与迭代。这样可以减少每次迭代中的循环次数，从而缩短训练时间。此外，弱大数定律保证随机梯度下降最终也能收敛到极值点。参数更新公式中，原来的求和符号不见了，因为我们只选择其中一个样本参与更新。

$$p_{uj}^{(t+1)} = p_{uj}^{(t)} + \epsilon(e_{ui}^{(t)} q_{ji}^{(t)} - \tau p_{uj}^{(t)})$$

$$q_{ji}^{(t+1)} = q_{ji}^{(t)} + \epsilon(e_{ui}^{(t)} p_{uj}^{(t)} - \tau q_{ji}^{(t)})$$

其中，ϵ是学习率，$e_{ui}^{(t)} = R_{ui}^{(t)} - \hat{R}_{ui}^{(t)}$，$\tau$是正则化项的系数，第一个公式中的$i$和第二个公式中的$j$是随机选取的。

```
def MF_with_randomization(K,epsilon,iteration,lamda):
    start=time.perf_counter()
    P = np.random.rand(len(Users), K)#.astype(type('float', (float,),
{}))
    Q = np.random.rand(K,len(Products))#.astype(type('float', (float,),
{}))
    RMSE_Seq = []
    MAE_Seq = []
    Prev_U = np.dot(P,Q)
    for I in range (iteration):
        i_rand = random.randint(0,len(user_rating_table)-1)
        j_rand = random.randint(0,len(user_rating_table[0])-1)
        for i in range(len(user_rating_table)):
            for k in range(K):
                if user_rating_table[i][j_rand]>0:

eij=user_rating_table[i][j_rand]-np.dot(P[i,:],Q[:,j_rand])

P[i][k]=P[i][k]+epsilon*(2*eij*Q[k][j_rand]-lamda*P[i][k])
        for j in range(len(user_rating_table[i])):
            for k in range(K):
                if user_rating_table[i_rand][j]>0:

eij=user_rating_table[i_rand][j]-np.dot(P[i_rand,:],Q[:,j])

Q[k][j]=Q[k][j]+epsilon*(2*eij*P[i_rand][k]-lamda*Q[k][j])
        U = np.dot(P,Q)
        F_value = np.linalg.norm(U-Prev_U)
```

```
        Prev_U = U
        RMSE,MAE = RMSEandMAE(user_rating_table,U)
        RMSE_Seq.append(RMSE)
        MAE_Seq.append(MAE)
        print("-------------")
        print("iteration_time")
        print(I)
        print("RMSE")
        print(RMSE)
        print("MAE")
        print(MAE)
        if(F_value<=1):
            break
    end=time.perf_counter()
    training_time = end - start
    return P,Q,RMSE_Seq,MAE_Seq,training_time
```

3.4 半正定矩阵分解

Gábor Takács 等人的论文[1]提出了一个简单的正则化矩阵分解的扩展，可以得到正的和半正的分解。当 U 和 M 中只包含非负值时，我们称之为正矩阵分解；当其中一个包含正值和负值时，我们称之为半正定矩阵分解。在半正定矩阵分解中，迭代更新公式与 3.2 中相同，但如果$p_{uj}^{(t+1)}$和$q_{ji}^{(t+1)}$计算出来是负的，我们就把它赋为 0。

```
def semipositiveMF(K,epsilon,iteration,lamda):
    start=time.perf_counter()
    P = np.random.rand(len(Users), K)#.astype(type('float', (float,),
{}))
    Q = np.random.rand(K,len(Products))#.astype(type('float', (float,),
{}))
    RMSE_Seq = []
    MAE_Seq = []
    Prev_U = np.dot(P,Q)
    for I in range (iteration):
        for i in range(len(user_rating_table)):
            for j in range(len(user_rating_table[i])):
                eij=user_rating_table[i][j]-np.dot(P[i,:],Q[:,j])
                for k in range(K):
                    if user_rating_table[i][j]>0:

P[i][k]=P[i][k]+epsilon*(2*eij*Q[k][j]-lamda*P[i][k])

Q[k][j]=Q[k][j]+epsilon*(2*eij*P[i][k]-lamda*Q[k][j])
```

[1] Gábor Takács. István Pilászy. Bottyán Németh. Domonkos Tikk. Investigation of Various Matrix Factorization Methods for Large Recommender Systems

```
                    if P[i][k]<0:
                        P[i][k]=0
                    if Q[k][j]<0:
                        Q[k][j]=0
        U = np.dot(P,Q)
        F_value = np.linalg.norm(U-Prev_U)
        Prev_U = U
        RMSE,MAE = RMSEandMAE(user_rating_table,U)
        RMSE_Seq.append(RMSE)
        MAE_Seq.append(MAE)
        print("-------------")
        print("iteration_time")
        print(I)
        print("RMSE")
        print(RMSE)
        print("MAE")
        print(MAE)
        if(F_value<=1):
            break
    end=time.perf_counter()
    training_time = end - start
    return P,Q,RMSE_Seq,MAE_Seq,training_time
```

3.5 动量梯度下降

Gábor Takács 等人的论文[1]还提出，我们可以通过对原始学习规则进行小幅修改来应用动量方法。在每个学习步骤中，权重的修改不仅仅是根据当前梯度计算得出，还考虑了上次权重变化的影响。用动量梯度下降进行矩阵分解，参数更新公式就变成：

$$\Delta p_{uj}^{(t+1)} = (1-\sigma)\left(e_{ui}^{(t)} q_{ji}^{(t)} - \tau p_{uj}^{(t)}\right) + \sigma \Delta p_{uj}^{(t)}$$

$$\Delta q_{ji}^{(t+1)} = (1-\sigma)\left(e_{ui}^{(t)} p_{uj}^{(t)} - \tau q_{ji}^{(t)}\right) + \sigma \Delta q_{ji}^{(t)}$$

$$p_{uj}^{(t+1)} = p_{uj}^{(t)} + \sigma \Delta p_{uj}^{(t+1)}$$

$$q_{ji}^{(t+1)} = q_{ji}^{(t)} + \sigma \Delta q_{ji}^{(t+1)}$$

其中，ϵ是学习率，$e_{ui}^{(t)} = R_{ui}^{(t)} - \hat{R}_{ui}^{(t)}$，$\tau$是正则化项的系数，$\sigma$是动量参数。

```
def momentumMF(K,epsilon,iteration,lamda,sigma):
    start=time.perf_counter()
    P = np.random.rand(len(Users), K)#.astype(type('float', (float,),
{}))
    Q = np.random.rand(K,len(Products))#.astype(type('float', (float,),
```

[1] Gábor Takács. István Pilászy. Bottyán Németh. Domonkos Tikk. Investigation of Various Matrix Factorization Methods for Large Recommender Systems

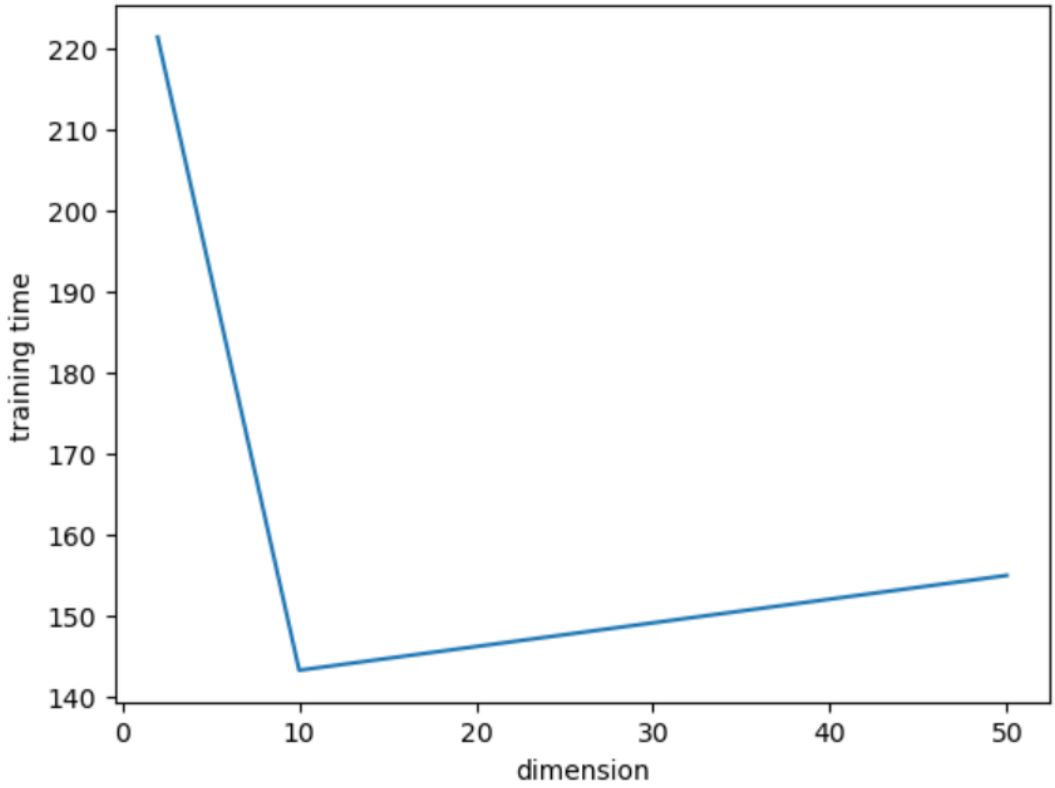
```
{}))
    RMSE_Seq = []
    MAE_Seq = []
    Prev_U = np.dot(P,Q)
    prev_sigma_p = np.zeros((len(user_rating_table),K))
    prev_sigma_q=np.zeros((K,len(user_rating_table[0])))
    for I in range (iteration):
        for i in range(len(user_rating_table)):
            for j in range(len(user_rating_table[i])):
                eij=user_rating_table[i][j]-np.dot(P[i,:],Q[:,j])
                for k in range(K):
                    if user_rating_table[i][j]>0:

prev_sigma_p[i][k]=(1-sigma)*(2*eij*Q[k][j]-lamda*P[i][k])+sigma*prev
_sigma_p[i][k]

prev_sigma_q[k][j]=(1-sigma)*(2*eij*P[i][k]-lamda*Q[k][j])+sigma*prev
_sigma_q[k][j]
                    P[i][k]=P[i][k]+epsilon*prev_sigma_p[i][k]
                    Q[k][j]=Q[k][j]+epsilon*prev_sigma_q[k][j]
        U = np.dot(P,Q)
        F_value = np.linalg.norm(U-Prev_U)
        Prev_U = U
        RMSE,MAE = RMSEandMAE(user_rating_table,U)
        RMSE_Seq.append(RMSE)
        MAE_Seq.append(MAE)
        print("--------------")
        print("iteration_time")
        print(I)
        print("RMSE")
        print(RMSE)
        print("MAE")
        print(MAE)
        if(F_value<=1):
            break
    end=time.perf_counter()
    training_time = end - start
    return P,Q,RMSE_Seq,MAE_Seq,training_time
```

四、 实验结果（验证提出方法的有效性和高效性）

4.1 基本梯度下降前提下 k 取不同值时的结果分析

本次实验中，我们总共设置了 2、10 和 50 这三个不同的 k 值。从运行时间来看，当 k 值取为 10 的时候运行时间最短，k 为 50 的时候稍微长一点，k 为 2 的时候

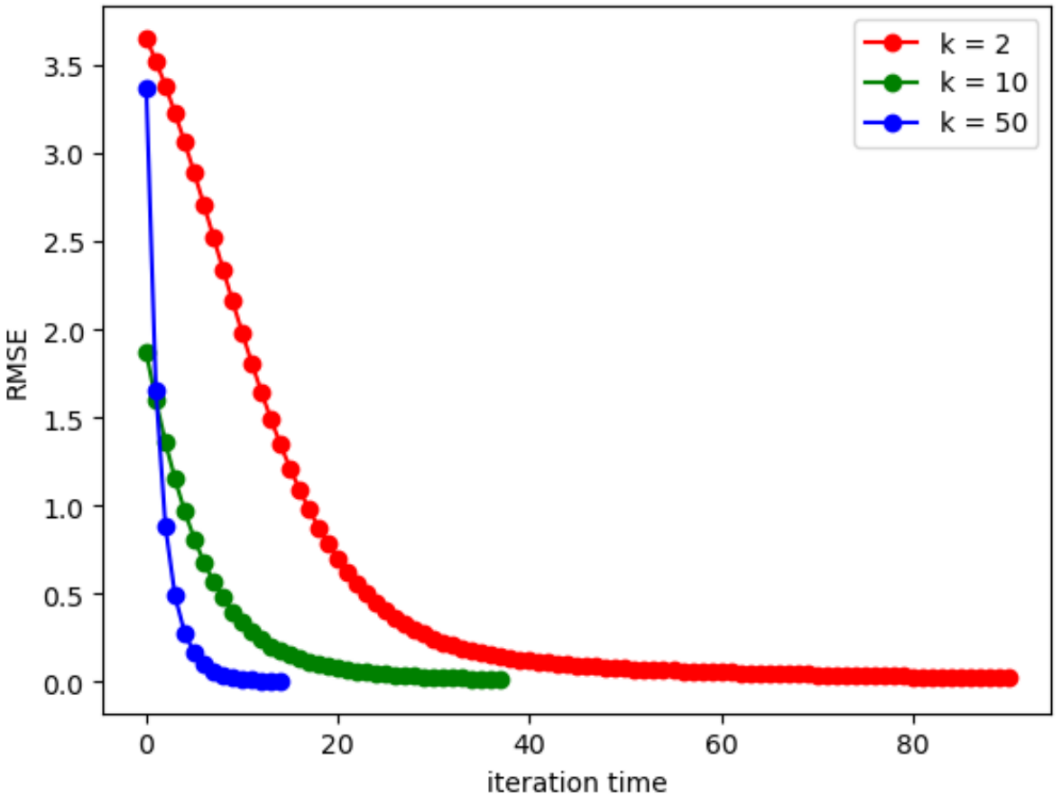
运行时间远长于另外两种情况。究其原因，是因为 k 越小，就需要更多的迭代次数来使梯度下降算法收敛。但当 k 比较大，矩阵的大小变大，训练过程中遍历整个矩阵的时间变长。所以，k 值既不能取得太小，又不能取得太大。

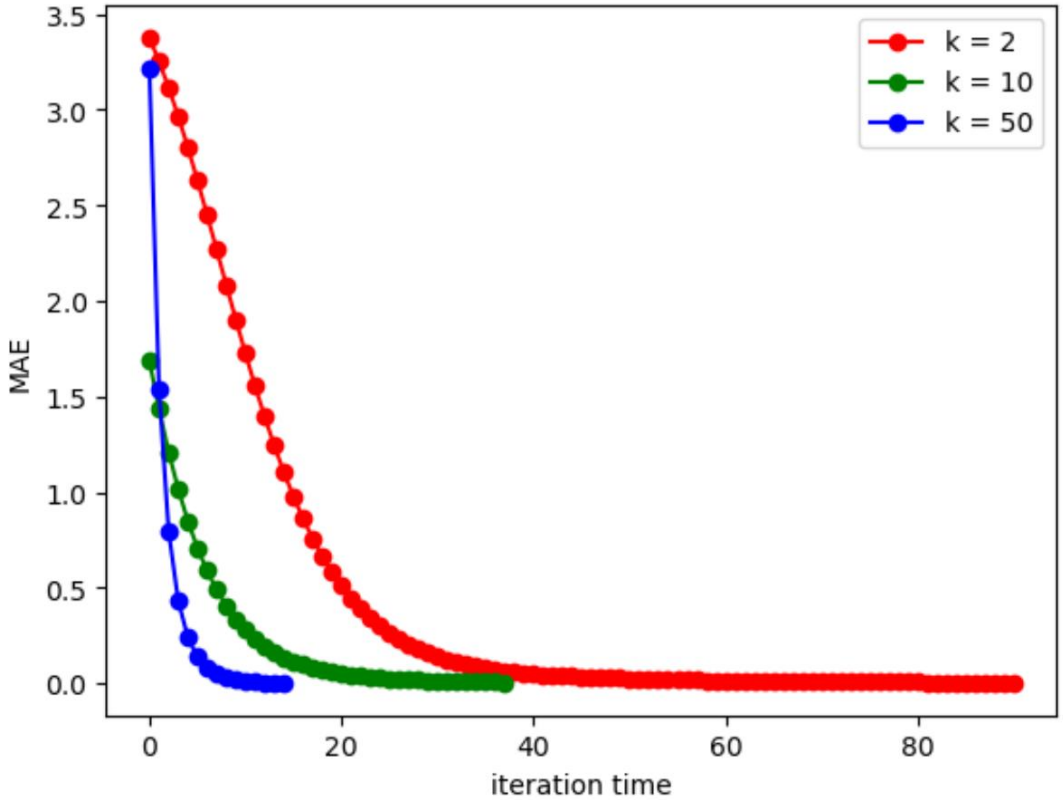


我们用 RMSE 和 MAE 来衡量算法的准确性。其中，RMSE 是重构矩阵与原始矩阵之差的 2-范数，MAE 是重构矩阵与原始矩阵之差的 1-范数。

```
def RMSEandMAE(R,R_hat):
    RMSE = 0
    MAE = 0
    counter = 0
    for i in range(len(R)):
        for j in range(len(R[0])):
            if(R[i][j]>=0):
                counter+=1
                RMSE=RMSE+(R[i][j]-R_hat[i][j])**2
                MAE = MAE+abs(R[i][j]-R_hat[i][j])
    RMSE=RMSE/counter
    MAE=MAE/counter
    RMSE=RMSE**0.5
    return RMSE,MAE
```

从下图我们可以看到，无论是 RMSE 还是 MAE，随着 k 值的增大，算法收敛所需要的迭代次数越少，最终得到的误差也越小。



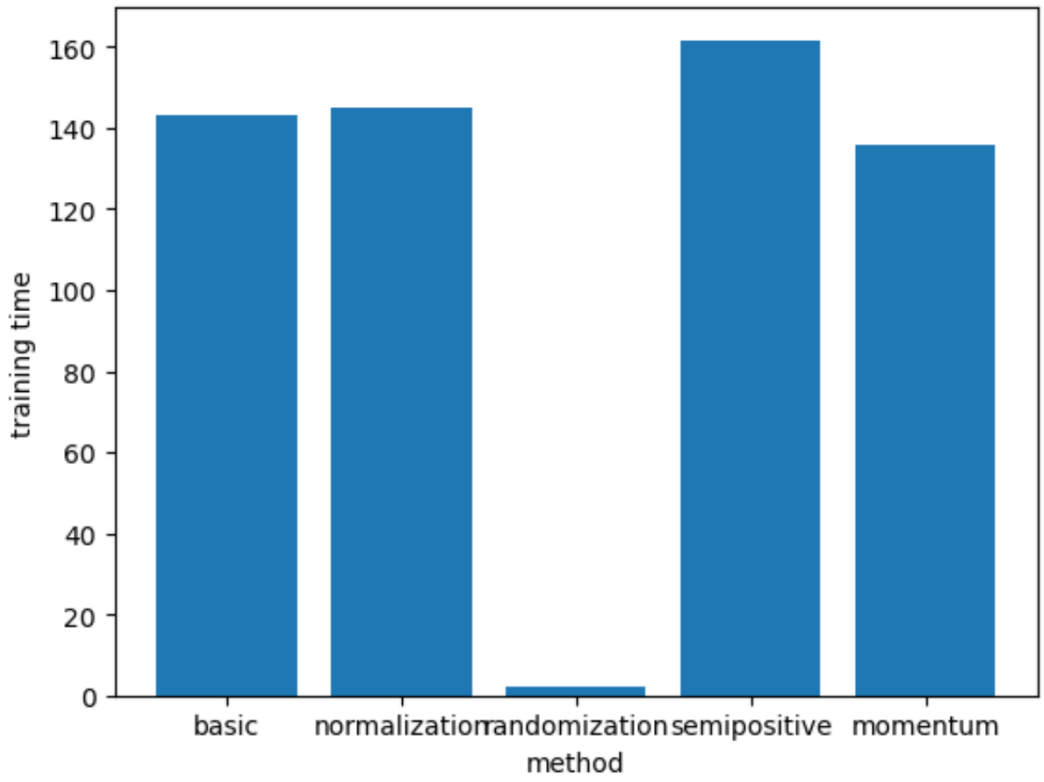


4.2 k 取 10 时用不同方法优化梯度下降得到的结果

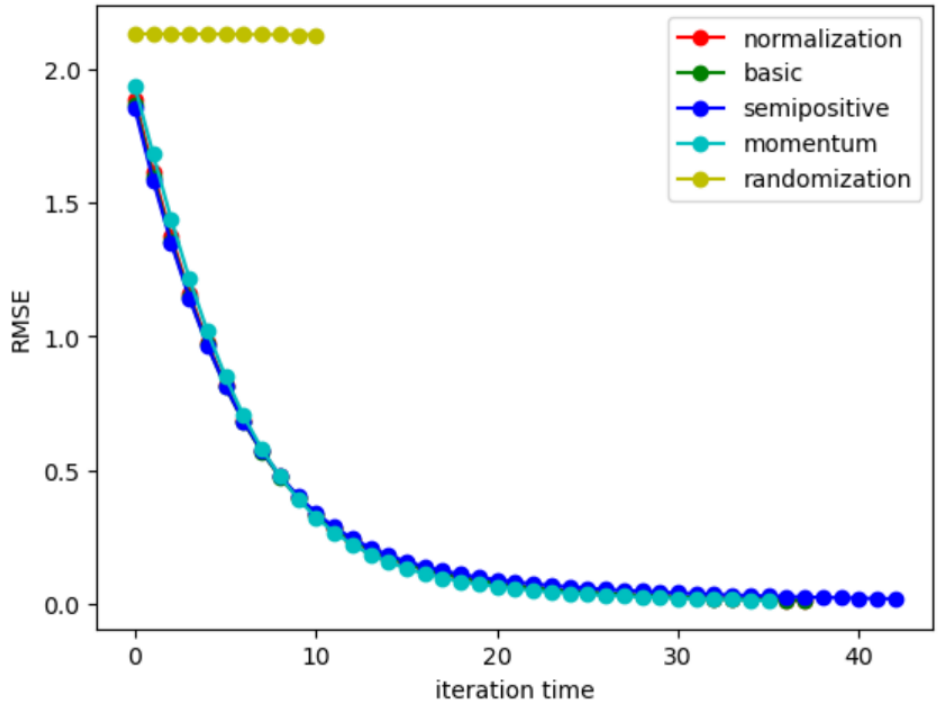
这里，我们把 k 设为 10，最大迭代次数设为 1500，学习率设为 0.01，正则化系数设为 0.00005，动量参数设为 0.3，比较不同优化方法的性能。

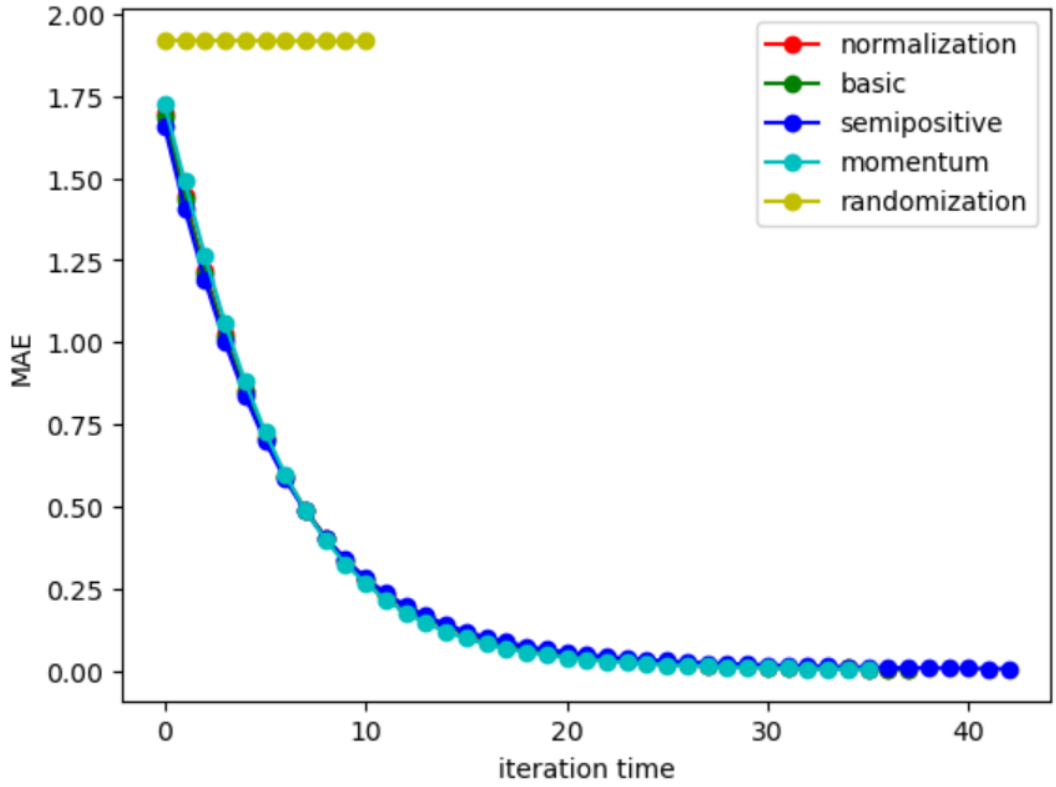
从运行时间来看，随机梯度下降的耗时最少，因为它仅仅迭代了十次就收敛了。

最长的是半正定矩阵分解，因为它在原有正则化梯度下降方法的基础上增加了一条判断语句。单纯的梯度下降和增加了正则化项的梯度下降运行时间相当，与之相比，动量梯度下降的运行时间还要略少一些。

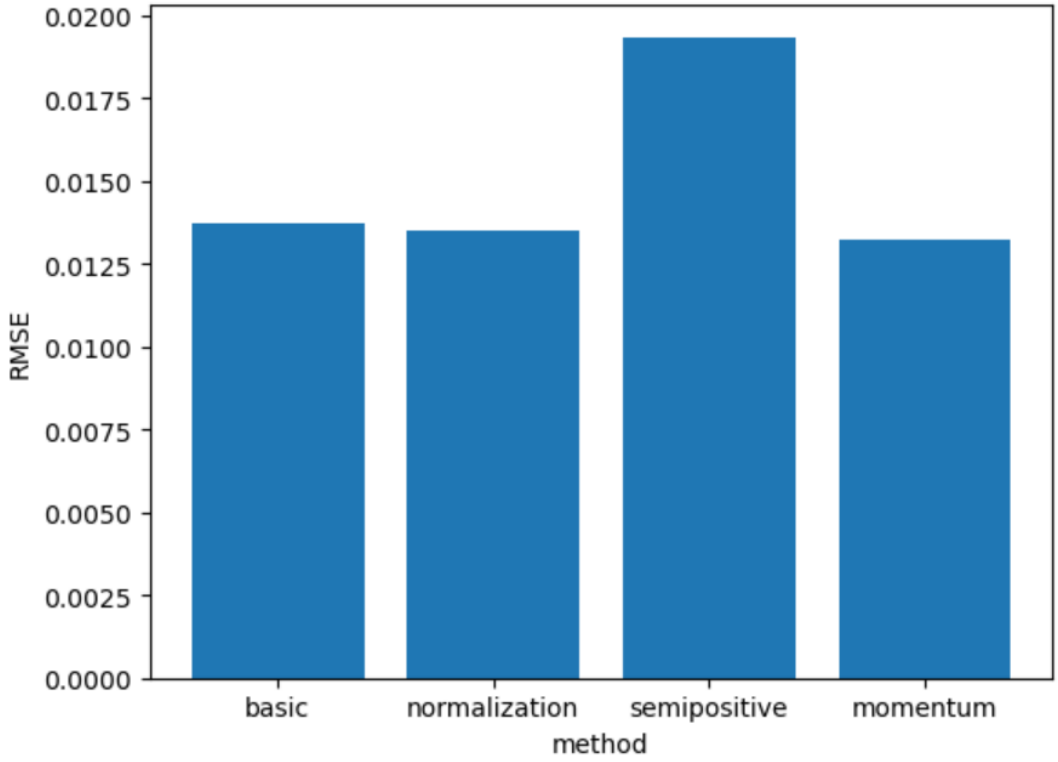


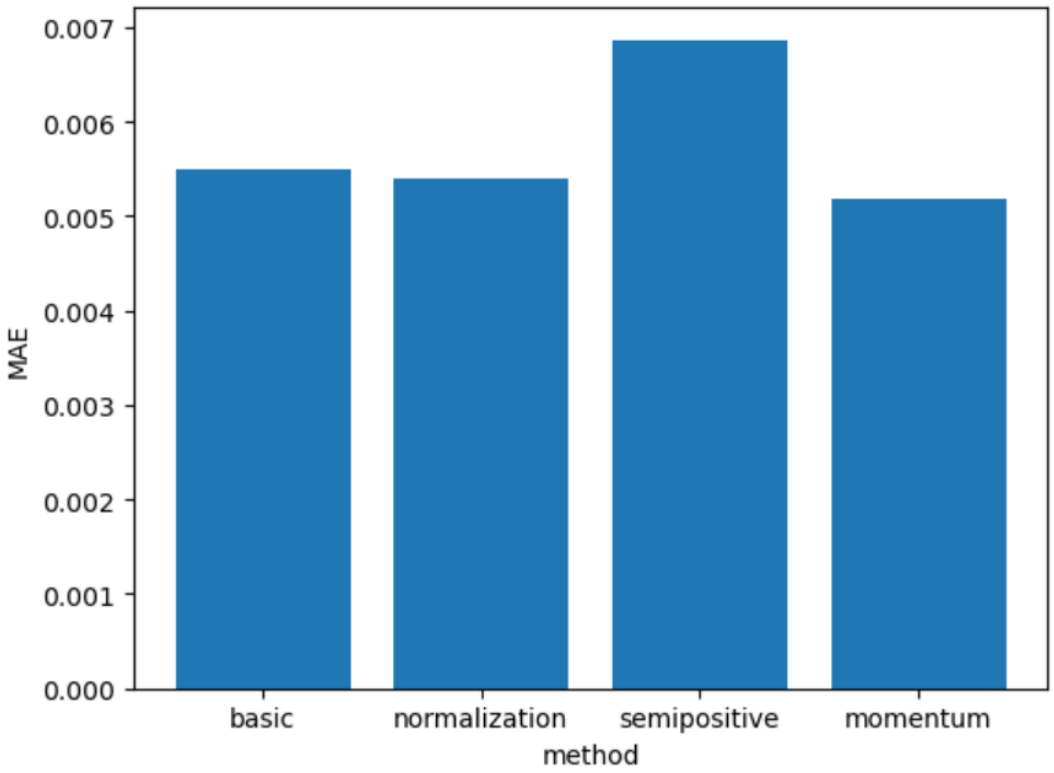
从准确性的角度来看，无论是 RMSE 还是 MAE，随机梯度下降达到收敛所需要的迭代次数最少，但它的误差也远大于其他四种方法。在图中我们可以看到，动量梯度下降的迭代次数比普通梯度下降、正则化梯度下降和半正定梯度下降更少，而且误差也更小。





我们再来更加细致地对比一下除了随机梯度下降以外其他四种算法的准确率。我们可以看到，半正定梯度下降似乎并不适用，因为从某种程度上来说，它使数据失真了，而这个例子中我们并不需要得到的重构矩阵中都是正数。最基本的梯度下降法误差第二大，加入正则化项以后算法的误差略有下降。而在加入正则化项以后，算法的误差又在正则化梯度下降的基础上进一步减小。





4.3 应用推荐

现在，我们就用得到的重构矩阵来给用户推荐商品了。我们先从所有用户中随机选择一位，并在重构评分矩阵中找到属于该用户的那一行，从其原先没有评分过的商品中找出预测评分值最大的前十个商品作为推荐给该用户的商品。

- 基本梯度下降法的推荐结果

[(3.131036726235799, 'B000B9RI14'), (3.1523593863958883, 'B000EUGX70'), (3.1748239626768, 'B000BUK7KW'), (3.1841093693313525, 'B00009R6VS'), (3.1931242312800636, 'B00001P4ZR'), (3.2037378383015453, 'B00004SABB'), (3.2170074510566904, 'B00017LSPI'), (3.2775032530364787, 'B000A6PPOK'), (3.296378624088832, 'B000GPVN32'), (3.601164059865309, 'B000ID6DTG')]

- 正则化梯度下降法的推荐结果

[(3.62406041994071, 'B000AM8SK2'), (3.6326138560959733, 'B000EBK3FW'), (3.6458550744423497, 'B000068034'), (3.6459198556489447, 'B00007M1TZ'), (3.6530622979355445, 'B0007KPRIS'), (3.657500359076782, 'B00004SYNX'), (3.6615375035452975, 'B000GPVN32'), (3.7834021579804413, 'B000F6LXUU'), (3.8102347953730935, 'B000FBK3QK'), (3.827088035414066, 'B00006BBAC')]

- 随机梯度下降法的推荐结果

[(2.5665078721216905, 'B000B60H0G'), (2.583075744310982, 'B000GFHJUS'), (2.6064871484896845, 'B000AA2RCY'), (2.641747799996311, 'B0002BEPU8'), (2.641834379514158, 'B000EWHH7I'), (2.6502046425003867, 'B00005BABT'), (2.6692092137345846, 'B00020M1U0'), (2.7106685751949673, 'B000FIQBL4'), (2.711718704560931, 'B000CR78C4'), (2.787452342341629, 'B000EIZJF4')]

- 半正定矩阵分解的推荐结果

[(3.8534277552067944, 'B0001Y7UAI'), (3.873172773651792, 'B000HPV3RW'), (3.9007690231851138, 'B0002WTK4S'), (3.9205979575620353, 'B0000C3GWU'), (3.952037371020966, 'B0000632H2'), (3.9639512448313322, 'B00006I53W'), (3.98975881114689, 'B000CKVOOY'), (4.085406652211237, 'B000068O16'), (4.103760570841141, 'B00009R8DS'), (4.119182408935584, 'B000EYRLXQ')]

- 动量梯度下降的推荐结果

[(3.055578662118161, 'B00009J5VX'), (3.0725689626842163, 'B000CQPWMS'), (3.086586749880527, 'B000H4WKWK'), (3.095235746760615, 'B0002L596M'), (3.1040452314812135, 'B000FJELQA'), (3.135885741385048, 'B00005T3G0'), (3.1405815287457175, 'B00006BBAC'), (3.173711713829349, 'B000B65Q3Y'), (3.2697672101661066, 'B0001M3MUW'), (3.2808567828247, 'B0000BZL0G')]

五、 结论（对使用的方法可能存在的不足进行分析，以及未来可能的研究方向进行讨论）

5.1 实验结论

- 在使用 MF 算法进行矩阵分解时，我们要适当选取潜在特征维数 k 的值，因为 k 值无论过小过大都会延长训练时间。
- 在基本梯度下降法的基础上增加正则化项，可以在略微延长训练时间的基础上得到稍高的正确率。
- 随机梯度下降法可以大大缩短训练时间，但误差会比使用所有样本进行训练高得多。
- 动量梯度下降在迭代时还会考虑上一次迭代带来的影响，可以在减少迭代次数、缩短训练时间的前提下提高正确率。

5.2 不足与展望

- 本次实验中，我们只设置了三种不同的潜在特征维数 k。在今后的研究中，我们还可以进一步研究如何权衡矩阵维度、迭代次数、准确率等因素，得到最好的 k 值。
- 本次实验中我们只要前一次迭代和本次迭代的重构矩阵之差的 F-范数小于 1 就停止迭代了，这导致随机梯度下降法过快停止迭代。今后，我们还可以调整终止迭代条件，进一步比较随机梯度下降和其他优化方法之间的性能差异。