《金融科技导论与实践》

实验3

推荐系统

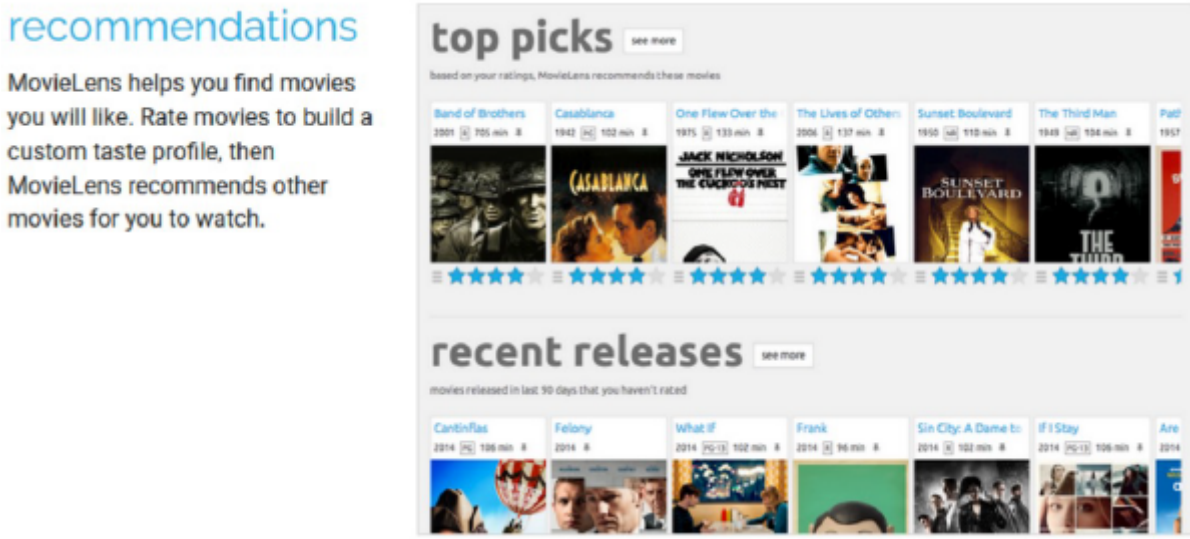
姓名： 周炜

学号： 32010103790

实验目的

1. 掌握矩阵分解的数学推导过程
2. 利用 python numpy 等软件包，实现基于矩阵分解技术下的推荐系统

本实验主要实现的是电影推荐



环境要求

所有的环境要求我都写了一份 `requirement.txt` (已经定义换源)放在各自的工程文件中, 以下我列出了一些重要的配置，并且给出了 `conda` 配置环境的 `shell` 命令：

```
conda create -n fintech python=3.8
conda activate fintech
python -m pip install -r requirements.txt
```

实验步骤

导入数据

在两个 `.csv` 的文件中，包括了 `userId` 表示用户， `movieId` 表示电影 `rating` 表示评分，`timestamp` （时间戳）表示时间，其中，时间戳并没有用到

```
train_rating = pd.read_csv("data/ratings_train_v1.csv")
test_rating = pd.read_csv("data/ratings_test_v1.csv")
train_rating.head()
```

	userId	movieId	rating	timestamp
0	1	804	4	964980499
1	1	1210	5	964980499
2	1	2628	4	964980523
3	1	2826	4	964980523
4	1	2018	5	964980523

得到 rating 矩阵

下列代码将评分矩阵`rating_mat`中的特定位置（由`user_id`和`item_id`确定）设置为当前的评分值`score`

`mse`表示均方误差（Mean Squared Error）

`rmse`表示均方根误差（Root Mean Squared Error）

```
for i in range(len(train_rating)):
    user = int(train_rating.iloc[i]['userId'])
    item = int(train_rating.iloc[i]['movieId'])
    score = float(train_rating.iloc[i]['rating'])
    user_id = np.where(all_user == user)[0][0]
    item_id = np.where(all_item == item)[0][0]
    rating_mat[user_id][item_id] = float(score)
```

实现矩阵分解的梯度下降算法

按照下述伪代码，在梯度下降函数中实现代码如下(使用`np.dot`加速运算)

The optimization scheme of Matrix Factorization with regularization term is given as:

Algorithm 1: Matrix Factorization with regularization

Input: $\boldsymbol{R}$: user/item rating matrix; λ: hyper parameters; γ: learning rate.
Output: $\boldsymbol{P}$: User embeddings; $\boldsymbol{Q}$: Item embeddings.
For $\text{epoch} = 1$ to T

$$\text{Update } \boldsymbol{p}_i \leftarrow \boldsymbol{p}_i + \gamma \left[\sum_{j \in S} (r_{ij} - \boldsymbol{p}_i^\top \boldsymbol{q}_j) \boldsymbol{q}_j - \lambda \boldsymbol{p}_i \right].$$

$$\text{Update } \boldsymbol{q}_j \leftarrow \boldsymbol{q}_j + \gamma \left[\sum_{i \in S} (r_{ij} - \boldsymbol{p}_i^\top \boldsymbol{q}_j) \boldsymbol{p}_i - \lambda \boldsymbol{q}_j \right].$$

End For

```
def get_loss(rating_mat, embed_dim, lamda, P, Q):
    pred_mat = np.dot(P, Q.T)
    idx = np.where(rating_mat > 0)
    diff = rating_mat[idx] - pred_mat[idx]
    error = np.sum(diff ** 2)
    error += lamda * (np.sum(P[idx[0], :] ** 2) + np.sum(Q[idx[1], :] ** 2))
    error /= rating_mat.shape[0]
    return error
```

记录矩阵分解梯度下降过程中loss下降的趋势

在给定`P`和`Q`的情况下， 按照下述式子进行计算：

$$\min_{P,Q} \ell = \sum_{i,j \in S} (r_{ij} - \hat{r}_{ij})^2 = \sum_{i,j \in S} (r_{ij} - \boldsymbol{p}_i^\top \boldsymbol{q}_j)^2 + \underbrace{\lambda \left(\sum_{i \in S} ||\boldsymbol{p}_i||^2 + \sum_{j \in S} ||\boldsymbol{q}_j||^2 \right)}_{\text{regularization}}$$

```
def matrix_factorization(rating_mat, embed_dim, gamma, lamda, steps):
    m, n = rating_mat.shape
    P = np.random.rand(m, embed_dim)
    Q = np.random.rand(embed_dim, n)
    loss = []
    for s in range(steps):
        for i in range(m):
            for j in range(n):
                if rating_mat[i][j] > 0:
                    rij = rating_mat[i][j] - np.dot(P[i,:], Q[:,j])
                    P[i] += gamma * (rij * Q[:,j] - lamda * P[i])
                    Q[:,j] += gamma * (rij * P[i] - lamda * Q[:,j])
        loss.append(get_loss(rating_mat, embed_dim, lamda, P, Q.T))
        print("Training step:  {}  Loss on the training datasets:
{:.10f}".format(s, loss[s]))
    return P, Q, loss
```

> 我尝试过一次运算出结果以提高效率
>
> ```
>  for i in range(steps):
>      pred_mat = np.dot(P, Q.T)
>      idx = np.where(rating_mat > 0)
>      eij = rating_mat[idx] - pred_mat[idx]
>      P[idx[0]] = np.maximum(0, P[idx[0]] + gamma * (np.outer(eij, Q[idx[1]])
> - lamda * P[idx[0]]))
>      Q[idx[1]] = np.maximum(0, Q[idx[1]] + gamma * (np.outer(eij, P[idx[0]])
> - lamda * Q[idx[1]]))
>      error = get_loss(rating_mat, embed_dim, lamda, P, Q)
>      error_list.append(error)
> ```
>
> 但是这会导致开的数组太大，超出计算机的内存范围

将训练好的用户和商品 embedding 进行测试评估

```
def test(test_rating, P, Q, all_user, all_item):
    mse = 0
    rmse = 0
    for i in range(len(test_rating)):
        user = int(test_rating.iloc[i]['userId'])
        item = int(test_rating.iloc[i]['movieId'])
        score = float(test_rating.iloc[i]['rating'])
        user_id = np.where(all_user == user)[0][0]
        item_id = np.where(all_item == item)[0][0]
        prediction = np.dot(P[user_id, :], Q[item_id, :])
```

```
            loss = score - prediction
            mse += loss ** 2
            rmse += loss ** 2
        mse /= len(test_rating)
        rmse = np.sqrt(rmse / len(test_rating))
        return mse, rmse
```

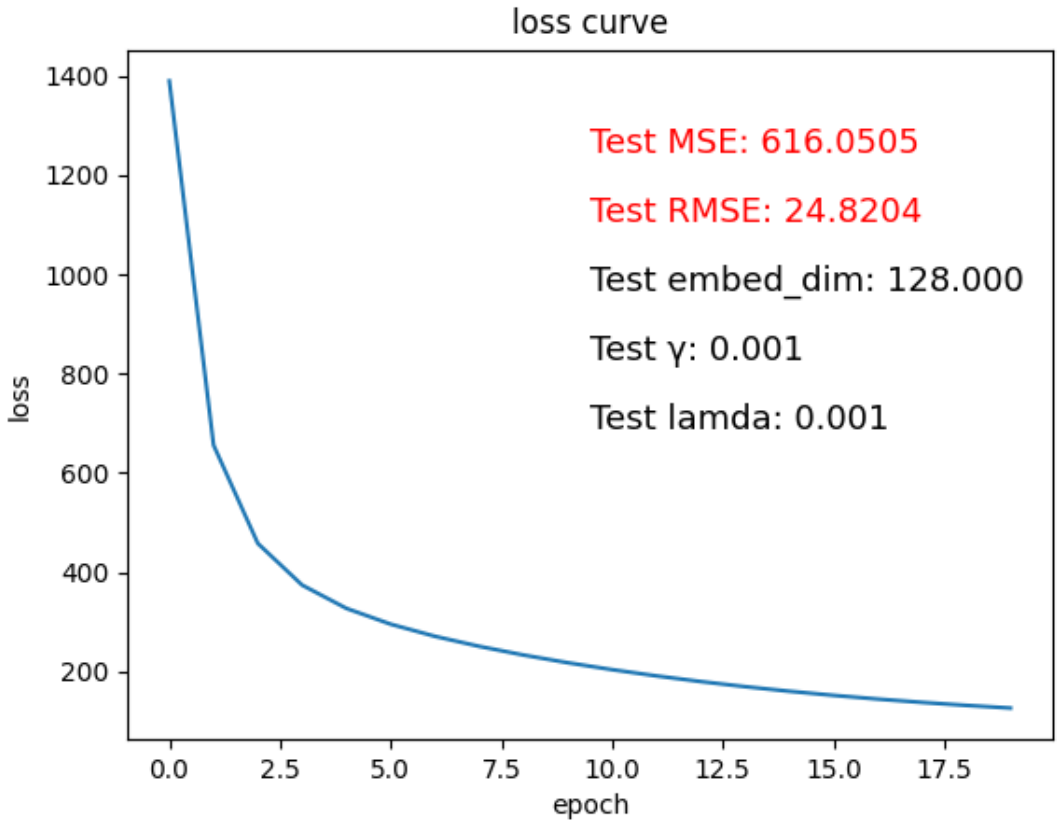
调试不同的λ和学习率γ

我修改了`draw`和`train`这两个函数，使得`MSE`和`RMSE`以及`embed_dim`，`gamma`，`lambda`能够反映在图上

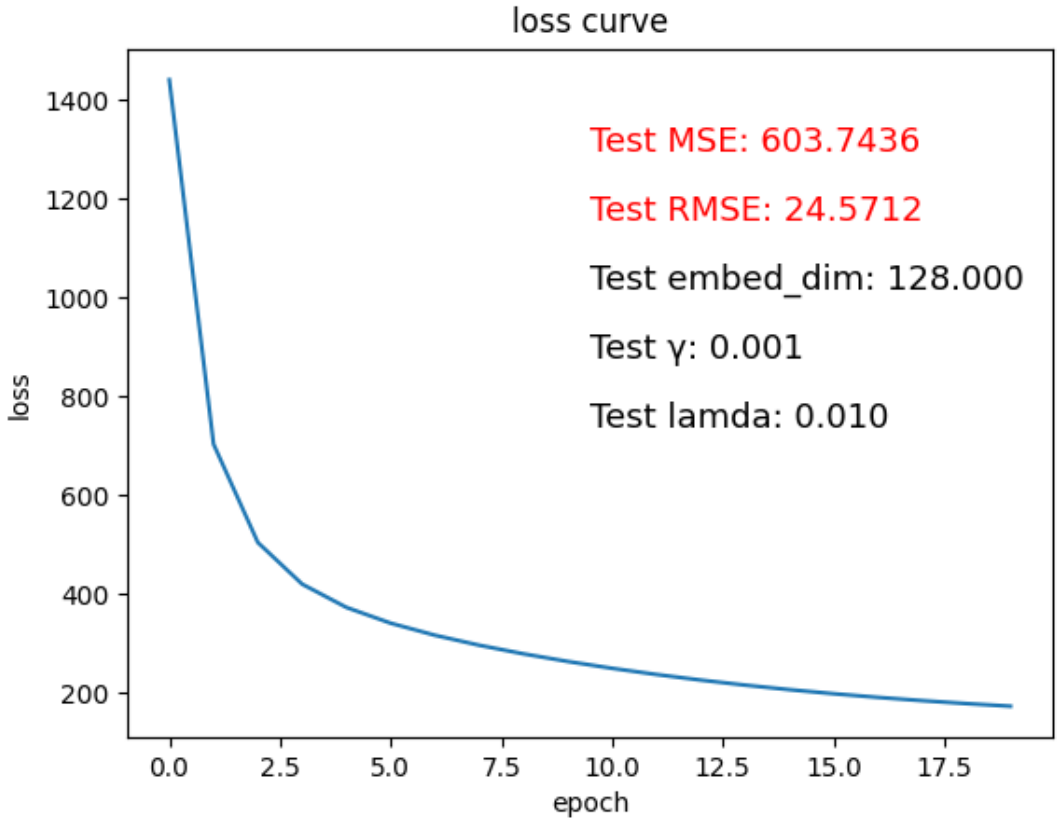
```
def draw(error_list, mse, rmse, embed_dim, gamma, lamda):
    plt.plot(range(len(error_list)), error_list)
    plt.xlabel("epoch")
    plt.ylabel("loss")
    plt.title("loss curve")
    plt.text(0.5, 0.85, f"Test MSE: {mse:.4f}", transform=plt.gca().transAxes,
fontsize=13, color='red')
    plt.text(0.5, 0.75, f"Test RMSE: {rmse:.4f}", transform=plt.gca().transAxes,
fontsize=13, color='red')
    plt.text(0.5, 0.65, f"Test embed_dim: {embed_dim:.3f}",
transform=plt.gca().transAxes, fontsize=13)
    plt.text(0.5, 0.55, f"Test γ: {gamma:.3f}", transform=plt.gca().transAxes,
fontsize=13)
    plt.text(0.5, 0.45, f"Test lamda: {lamda:.3f}",
transform=plt.gca().transAxes, fontsize=13)
    plt.savefig("loss curve.png")

def train(rating_mat, all_user, all_item, embed_dim, gamma, lamda, steps):
    P, Q, loss = matrix_factorization(rating_mat, embed_dim, gamma, lamda, steps)
    mse, rmse = test(test_rating, P, Q.T, all_user, all_item)
    print('Test MSE: ', mse, 'Test RMSE: ', rmse)
    draw(loss, mse, rmse, embed_dim, gamma, lamda)
```

使得`embed_dim = 32`，`gamma = 0.001`，改变`lamda`

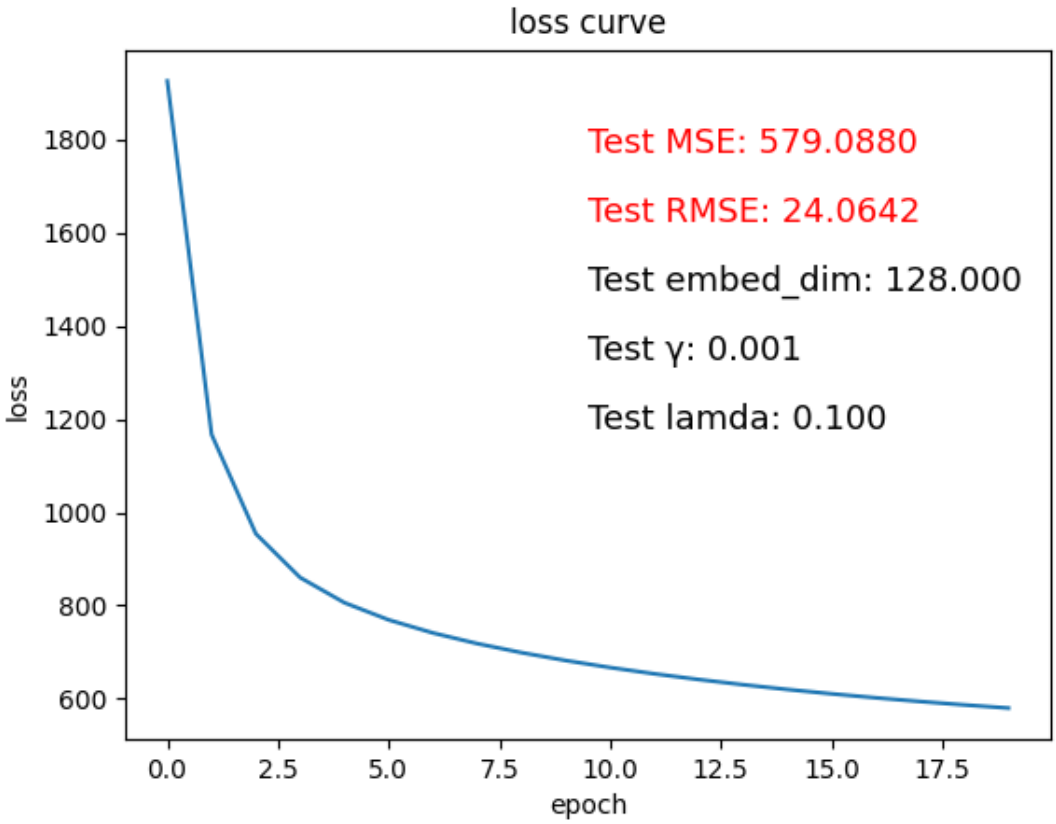
`lamda = 0.001`



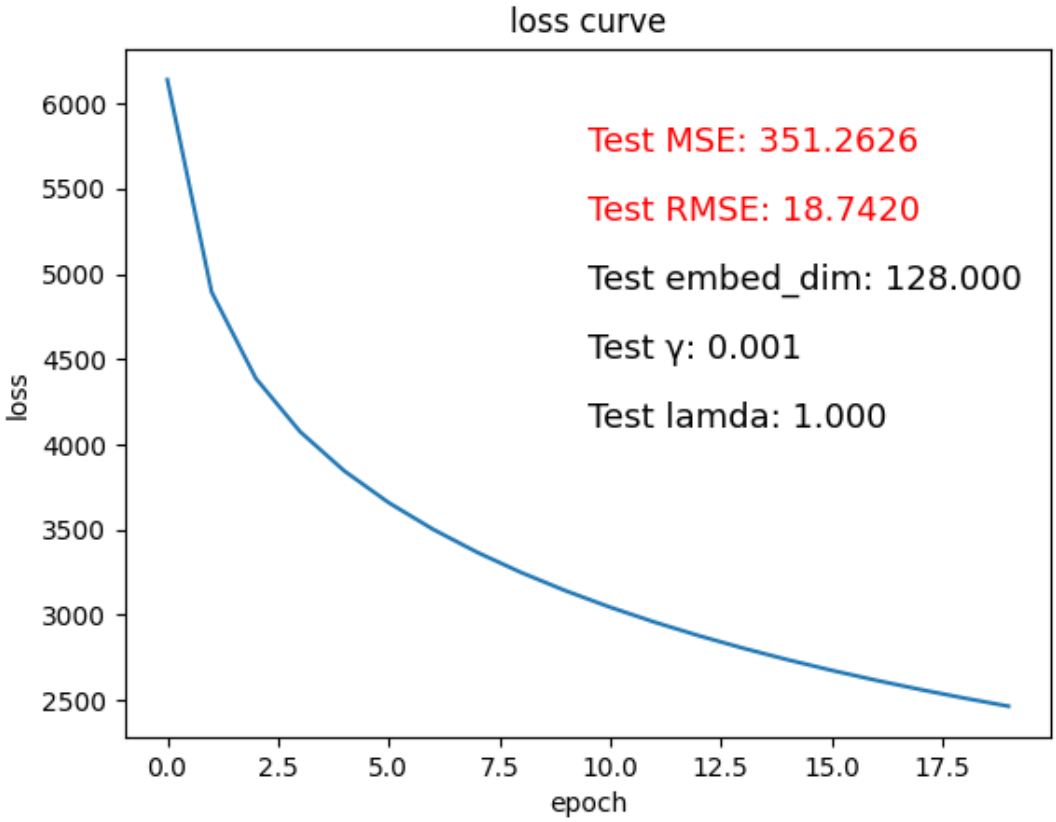
`lamda = 0.01`



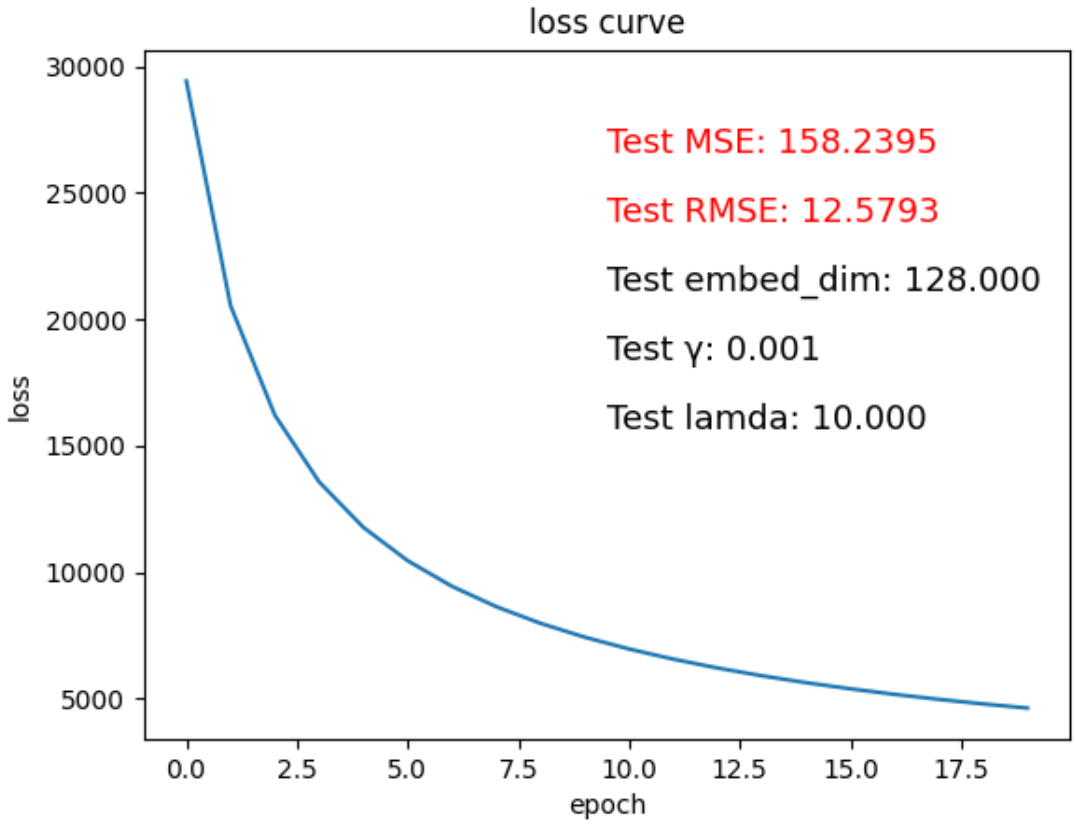
`lamda = 0.1`



`lamda = 1`



`lamda = 10`



可以看到，随着`lamda`增大，训练`loss`增大，但是测试集的`MSE`却在逐渐变小

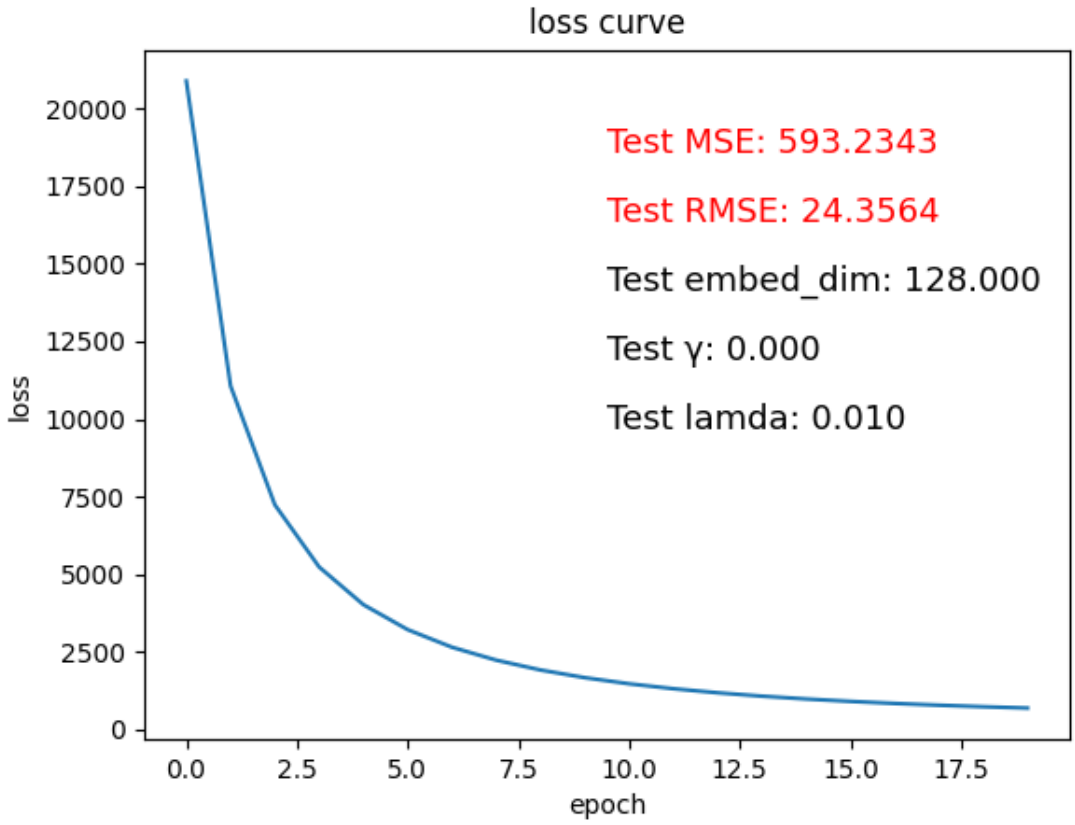
在矩阵分解过程中，`lamda`是正则化参数，用于控制模型的复杂度。增大`lamda`会对模型的参数进行更强的约束，限制其取值范围。这样可以减小过拟合的风险，提高模型的泛化能力

增大`lamda`会引入更多的正则化项，使得模型对训练数据的拟合程度减弱，因此训练误差会增加,导致训练`loss`增大

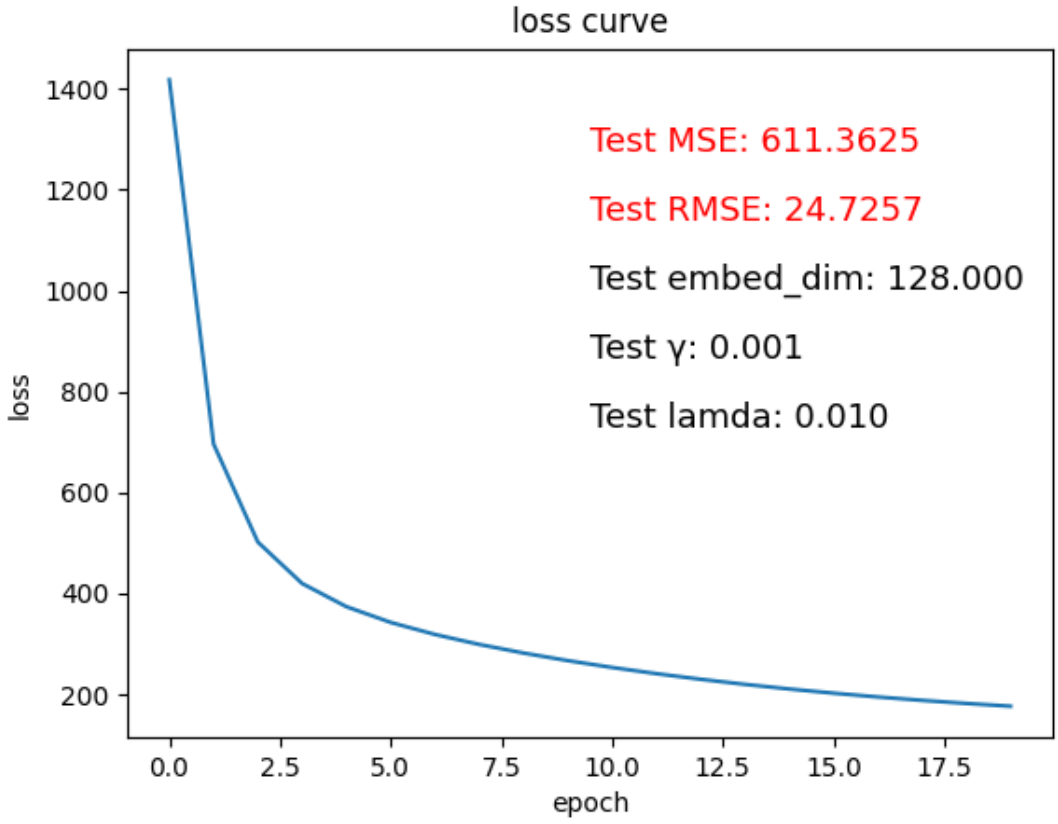
增大`lamda`可以有效地控制模型的复杂度，防止过拟合，使得模型在未见过的测试数据上表现更好，因此测试集的`MSE`逐渐变小

使得`embed_dim = 32`，`lamda = 0.01`，改变`gamma`

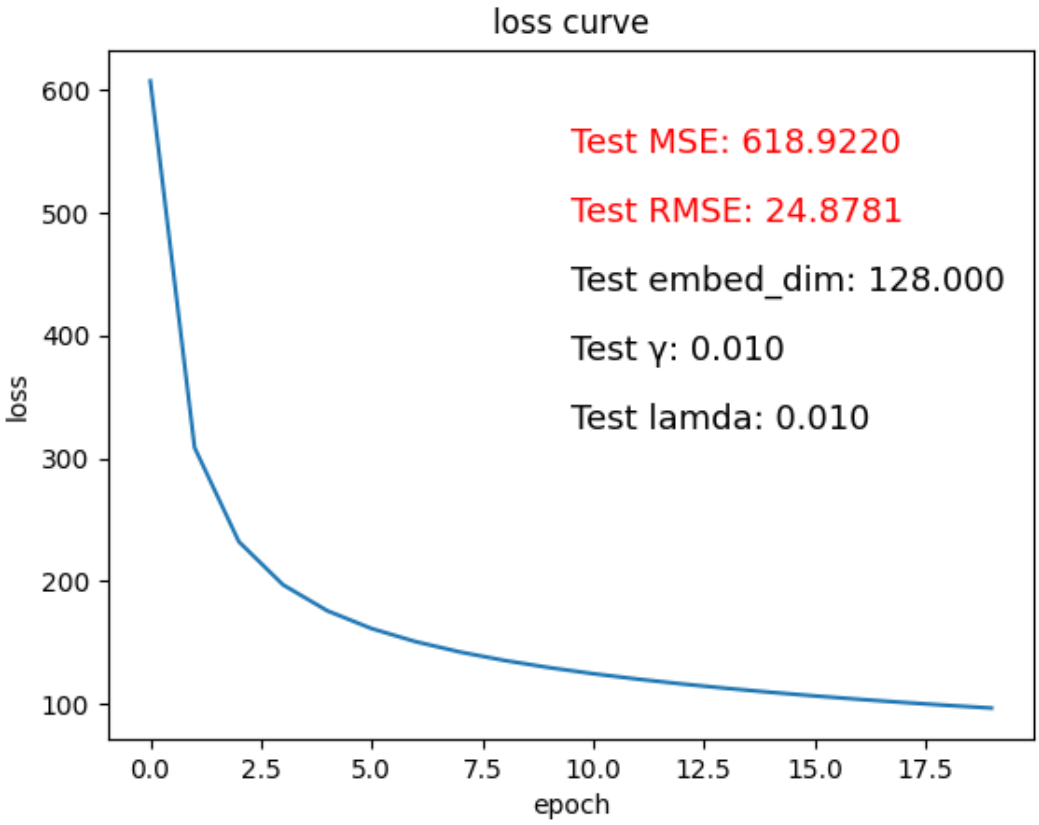
`gamma = 0.0001`



`gamma = 0.001`



`gamma = 0.01`



可以看到，随着`gamma`的增大，训练`loss`变小，但测试`MSE`在增大

推测这可能是由两个原因造成的

1. **过拟合问题**：`gamma`是学习率，它控制了每次迭代中参数更新的步长。当`gamma`较大时，每次更新的幅度也会增大。这可能导致模型在训练集上更快地收敛，训练集上的误差减小。然而，如果`gamma`设置得过大，模型可能会过于迅速地适应训练集的噪声和细节，导致过拟合。过拟合表示模型在训练集上表现良好，但在未见过的测试集上表现较差。因此，当`gamma`增大时，模型在训练集上的误差减小，但在测试集上的`MSE`可能会增大，因为模型过度拟合了训练集的特定特征和噪声。
2. **泛化能力下降**：增大`gamma`可能会导致模型在训练集上更快地收敛，但在测试集上的泛化能力可能会下降。泛化能力是指模型在未见过的数据上的表现能力。当`gamma`较大时，模型可能会过于专注于训练集中的细节和噪声，而忽略了更一般的模式和趋势。这种情况下，模型可能无法很好地适应测试集中的数据，导致测试集的`MSE`增大

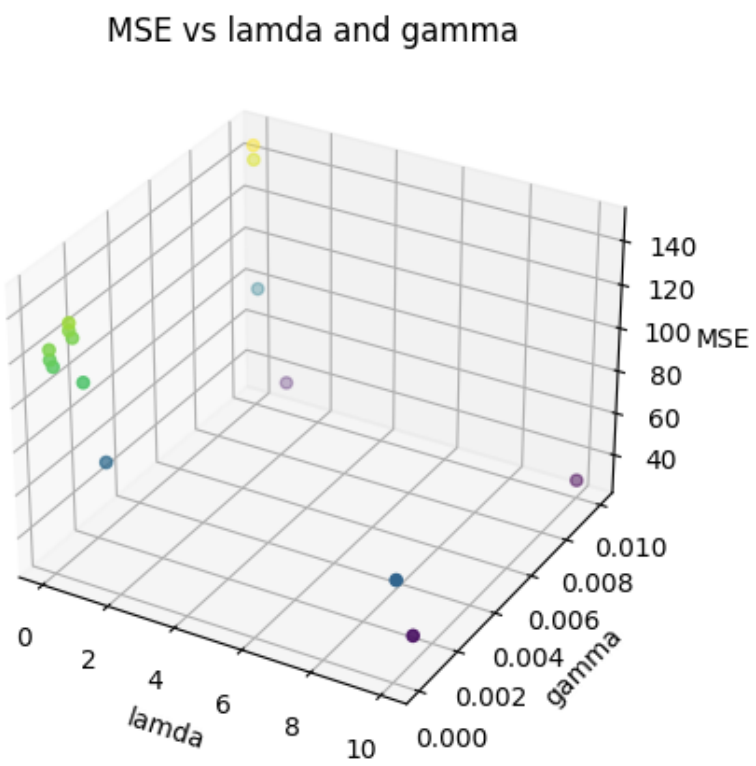
绘制三维关系图

我还使用了`seaborn`绘制`MSE`与`lamda`和`gamma`的关系三维图

```
def threeD():
    lamda_values = [0.001, 0.01, 0.1, 1, 10]
    gamma_values = [0.0001, 0.001, 0.01]
    mse_results = []
    for lamda in lamda_values:
        for gamma in gamma_values:
            P, Q, loss = matrix_factorization(rating_mat, embed_dim=64,
gamma=gamma, lamda=lamda, steps=20)
            mse, _ = test(test_rating, P, Q.T, all_user, all_item)
            mse_results.append((lamda, gamma, mse))
    mse_df = pd.DataFrame(mse_results, columns=["lamda", "gamma", "mse"])
    fig = plt.figure()
```

```
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(mse_df['lamda'], mse_df['gamma'], mse_df['mse'], c=mse_df['mse'],
cmap='viridis')
    ax.set_xlabel('lamda')
    ax.set_ylabel('gamma')
    ax.set_zlabel('MSE')
    plt.title("MSE vs lamda and gamma")
    plt.savefig("MSE_vs_lamda_gamma.png")
```

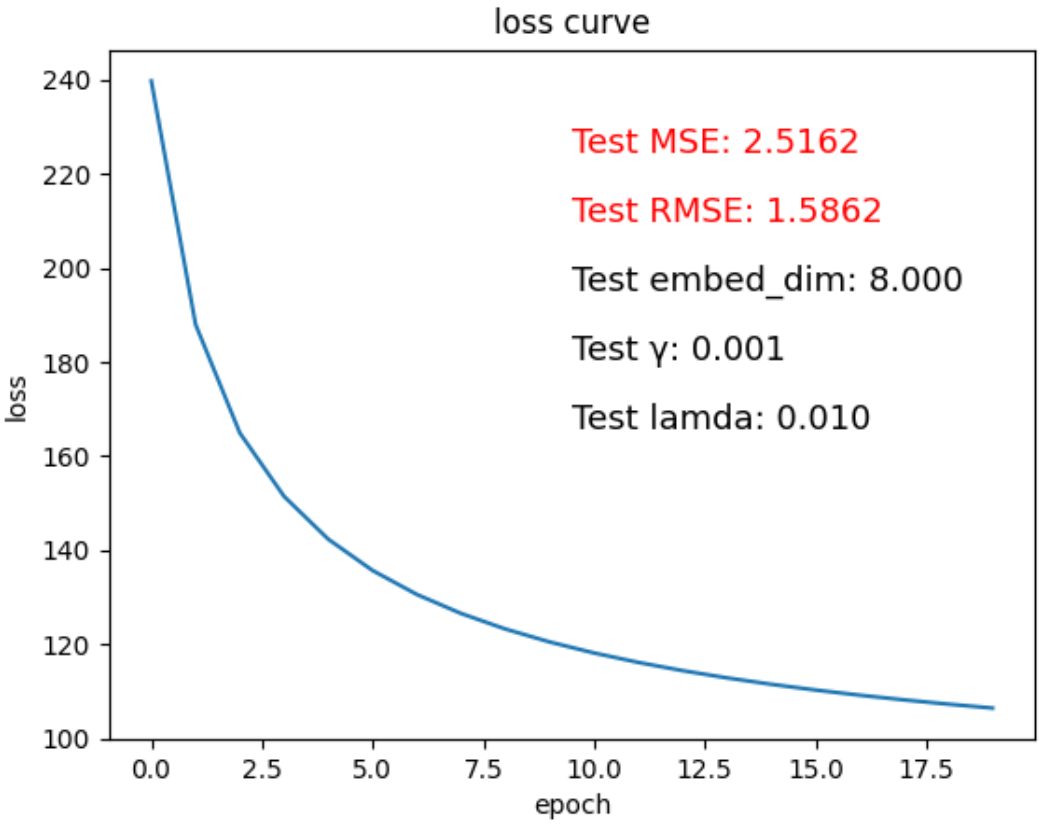
输出结果为：



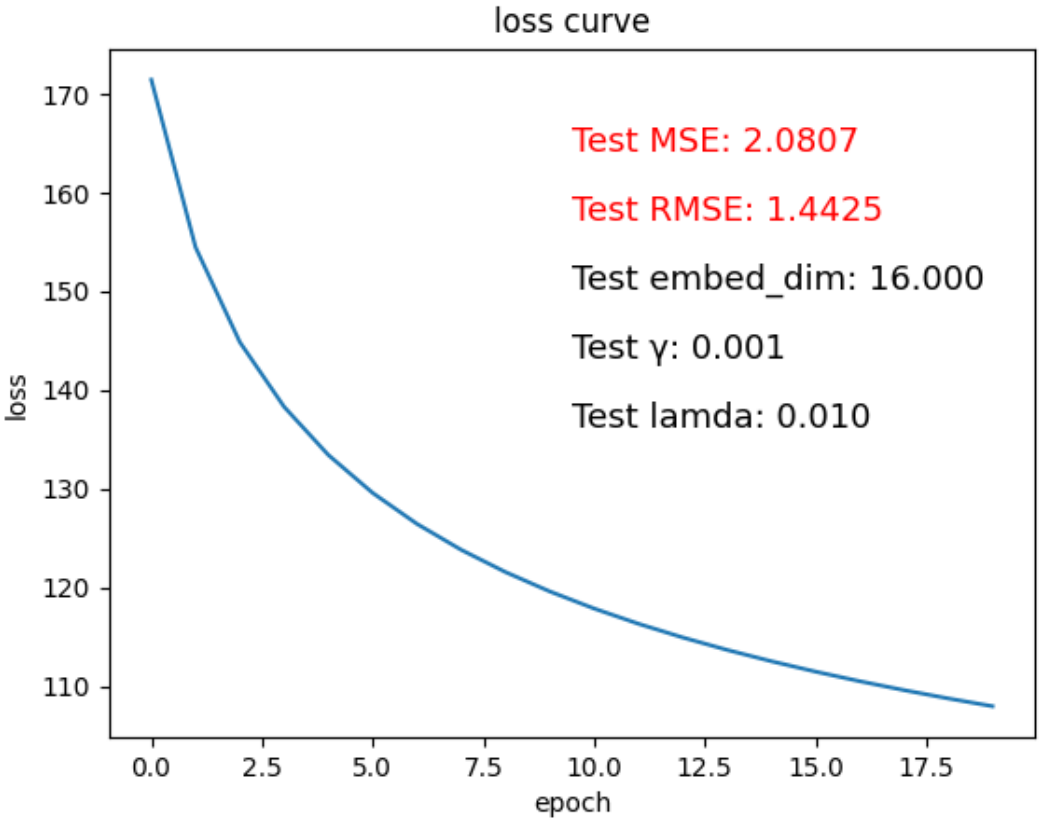
调试不同的用户商品嵌入维度embed_dim

使得 `lamda = 0.01`，`gamma = 0.001` 改变 `embed_dim`

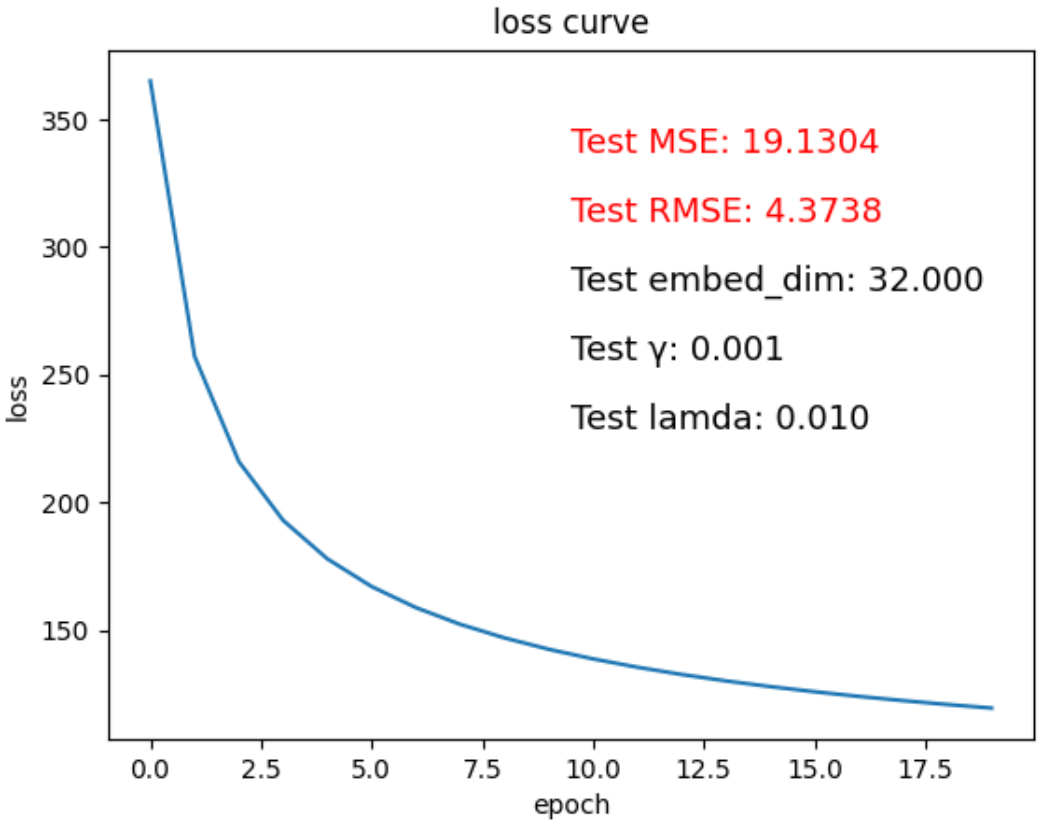
`embed_dim = 8`



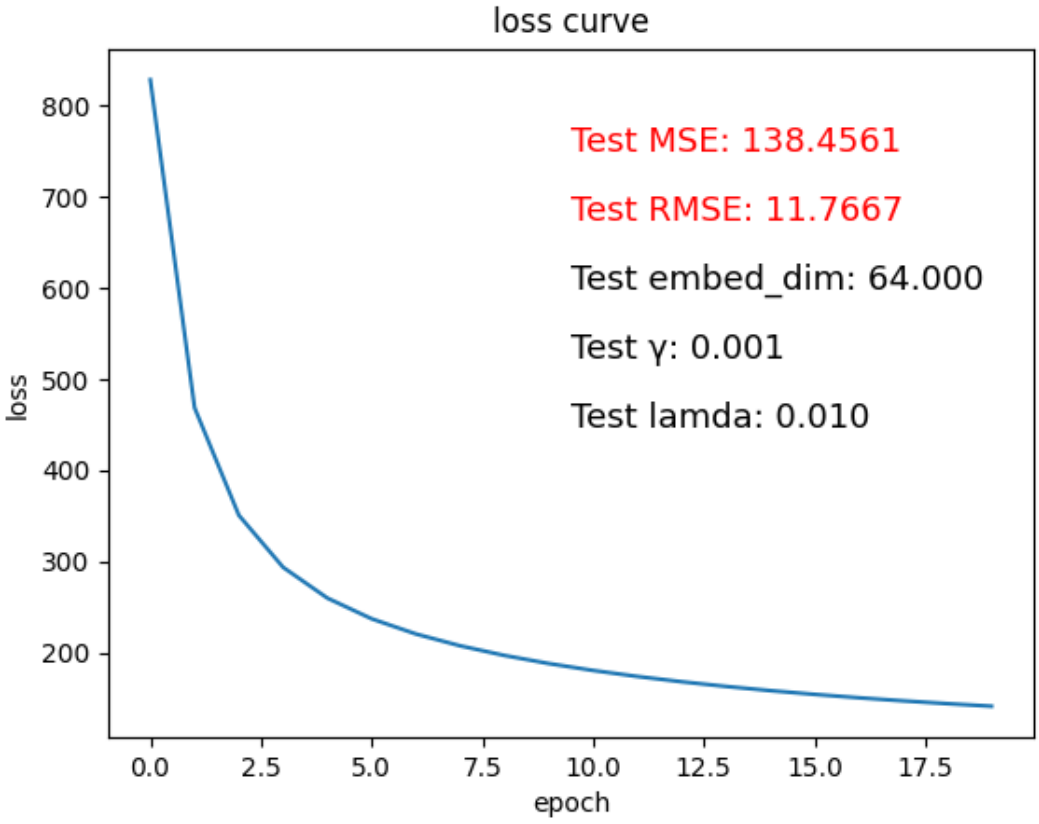
`embed_dim = 16`



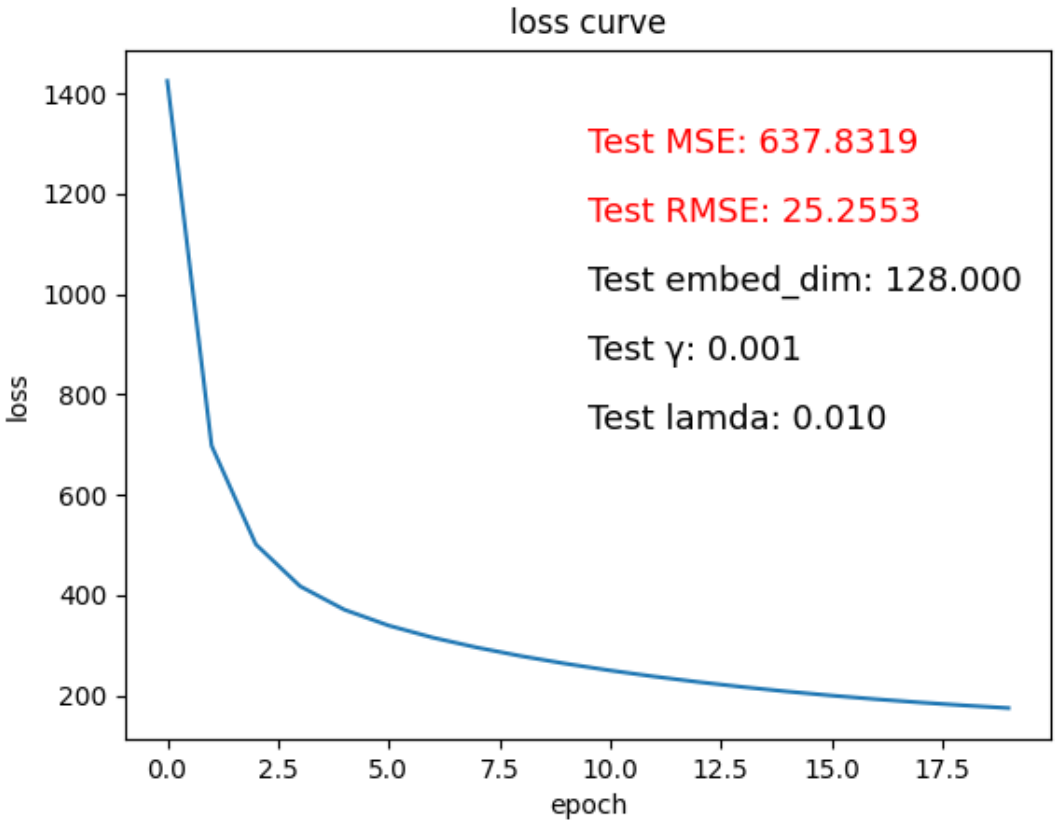
`embed_dim = 32`



`embed_dim = 64`



`embed_dim = 128`



当`embed_dim`较小的时候结果较好，当特征数量太大时，模型效果会剧烈下降

当`embed_dim`较小时，模型的特征维度较低，意味着每个用户和物品的表示会被压缩到一个较低维度的向量空间中。这种压缩使得模型更加简洁和紧凑，减少了参数的数量和复杂性。由于特征维度较小，模型的学习能力相对有限，可能无法捕捉到数据中的所有细节和复杂关系。然而，当数据量较小或特征之间的相关性较低时，较小的`embed_dim`可以防止模型过度拟合训练数据，并在测试数据上表现较好。这是因为较小的特征维度可以强制模型学习到更加通用和共享的特征表示，从而提高泛化能力

加上非负约束，实现矩阵分解建模

$$s.t. \quad p_{ij} \geq 0, q_{ij} \geq 0$$

上式矩阵分解的函数可以更新为（即最内存循环的`p[i]`,`q[:,j]`的更新需要与0比较再取其中的较大值）

$$p_i \leftarrow \max(0, p_i + \gamma \left[\sum_{j \in S} (r_{ij} - {p_i}^T q_j) q_j - \lambda p_i \right])$$

$$q_j \leftarrow \max(0, q_j + \gamma \left[\sum_{j \in S} (r_{ij} - {p_i}^T q_j) p_i - \lambda q_j \right])$$

编写函数：

```
def non_negative_matrix_factorization(rating_mat, embed_dim, gamma, lamda,
steps):
    m, n = rating_mat.shape
    P = np.random.rand(m, embed_dim)
    Q = np.random.rand(embed_dim, n)
    loss = []
    for s in range(steps):
        for i in range(m):
```

```
            for j in range(n):
                if rating_mat[i][j] > 0:
                    rij = rating_mat[i][j] - np.dot(P[i,:], Q[:,j])
                    P[i] = np.maximum(np.zeros(embed_dim), P[i] + gamma * (rij *
Q[:,j] - lamda * P[i]))
                    Q[:,j] = np.maximum(np.zeros(embed_dim), Q[:,j] + gamma *
(rij * P[i] - lamda * Q[:,j]))
        loss.append(get_loss(rating_mat, embed_dim, lamda, P, Q.T))
        print("Training step:  {}  Loss on the training datasets:
{:.10f}".format(s, loss[s]))
    return P, Q, loss
```