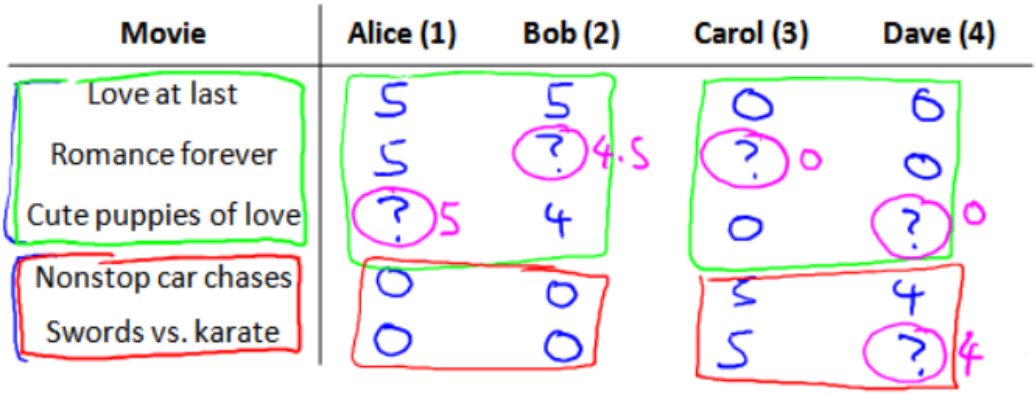
推荐系统(Recommender Systems)

问题形式化

推荐系统是机器学习中的一个重要的应用。如淘宝推荐商品，抖音推荐短视频等等。这些推荐系统，根据浏览你过去买过的东西，或过去评价过什么类型的视频来判断给你推荐他们的服务。

我们从一个例子开始定义推荐系统的问题。

假我们是一个电影供应商，我们有 5 部电影和 4 个用户，我们要求用户为电影打分。



前三部电影是爱情片，后两部则是动作片，我们可以看出**Alice**和**Bob**似乎更倾向与爱情片， 而 **Carol** 和 **Dave** 似乎更倾向与动作片。并且没有一个用户给所有的电影都打过分。我们希望构建一个算法来预测他们每个人可能会给他们没看过的电影打多少分，并以此作为推荐的依据。

基于内容的推荐系统

在电影推荐系统的例子中，我们可以假设每部电影都有两个特征，如x_1代表电影的浪漫程度，x_2 代表电影的动作程度。

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	0.9	0
Romance forever	5	?	?	0	1.0	0.01
Cute puppies of love	?	4	0	?	0.99	0
Nonstop car chases	0	0	5	4	0.1	1.0
Swords vs. karate	0	0	5	?	0	0.9

则每部电影都有一个特征向量，如$x^{(1)}$是第一部电影的特征向量为[0.9 0]。

下面我们要基于这些特征来构建一个推荐系统算法。

下面引入一些标记：

n_u 代表用户的数量

n_m 代表电影的数量

$r(i,j)$ 如果用户 j 给电影 i 评过分则 $r(i,j)=1$

$y^{(i,j)}$ 代表用户 j 给电影i的评分

m_j代表用户 j 评过分的电影的总数

假设我们采用线性回归模型，我们可以针对每一个用户都训练一个线性回归模型，如$\theta^{(1)}$是第一个用户的模型的参数。

$\theta^{(j)}$是用户 j 的参数向量

$x^{(i)}$是电影 i 的特征向量

对于用户 j 和电影 i，我们预测评分为：$(\theta^{(j)})^T x^{(i)}$

代价函数可以按照如下定义

针对用户 j，该线性回归模型的代价为预测误差的平方和，加上正则化项：

$$\min_{\theta(j)} \frac{1}{2} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \left(\theta_k^{(j)} \right)^2$$

其中 $i:r(i,j)$表示我们只计算那些用户 j 评过分的电影。在一般的线性回归模型中，误差项和正则项应该都是乘以$1/2m$，在这里我们将m去掉，这是因为去掉m依然不影响我们对代价取最小值的处理。在这里我们不对方差项θ_0进行正则化处理。

上面的代价函数只是针对一个用户的，为了学习所有用户，我们将所有用户的代价函数求和：

$$\min_{\theta^{(1)},\dots,\theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

如果我们要用梯度下降法来求解最优解，我们计算代价函数的偏导数后得到梯度下降的更新公式为：

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} \quad (\text{for } k = 0)$$

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)} \right) \quad (\text{for } k \neq 0)$$

协同过滤

在之前的基于内容的推荐系统中，对于每一部电影，我们可以使用这些特征训练出了每一个用户的参数。相反地，如果我们拥有用户的参数，我们可以学习得出电影的特征。

$$\min_{x^{(1)},\dots,x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{jr(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

但是如果我们既没有用户的参数，也没有电影的特征，这两种方法都不可行了。协同过滤算法可以同时学习这两者。

我们的优化目标便改为同时针对x和θ进行。

$$J(x^{(1)}, \dots x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)}) = \frac{1}{2} \sum_{(i:j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

对代价函数求偏导数的结果如下：

$$x_k^{(i)} := x_k^{(i)} - \alpha \left(\sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) \theta_k^j + \lambda x_k^{(i)} \right)$$

$$\theta_k^{(i)} := \theta_k^{(i)} - \alpha \left(\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)} \right)$$

协同过滤算法使用步骤如下：

1. 初始 $x^{(1)}, x^{(1)}, \ldots x^{(nm)}$, $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(n_u)}$为一些随机小值
2. 使用梯度下降算法最小化代价函数
3. 在训练完算法后，我们预测$(\theta^{(j)})^T x^{(i)}$为用户 j 给电影 i 的评分

通过这个学习过程获得的特征矩阵包含了有关电影的一些特征数据，这些数据不总是人能读懂的，但是我们可以用这些数据作为给用户推荐电影的依据。

例如，如果一位用户正在观看电影 $x^{(i)}$，我们可以寻找另一部电影$x^{(j)}$，依据两部电影的特征向量之间的距离$\left\| x^{(i)} - x^{(j)} \right\|$的大小。

协同过滤算法

协同过滤优化目标：

给定$x^{(1)}, \ldots, x^{(n_m)}$，估计$\theta^{(1)}, \ldots, \theta^{(n_u)}$：

$$\min_{\theta^{(1)},\ldots,\theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

给定$\theta^{(1)}, \ldots, \theta^{(n_u)}$，估计$x^{(1)}, \ldots, x^{(n_m)}$：

$$min_{x^{(1)},\ldots,x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{jr(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

同时最小化$x^{(1)}, \ldots, x^{(n_m)}$和$\theta^{(1)}, \ldots, \theta^{(n_u)}$：

$$J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

$$\min_{x^{(1)},\ldots,x^{(n_m)}\ \theta^{(1)},\ldots,\theta^{(n_u)}} J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)})$$

向量化：低秩矩阵分解

本小节的内容是协同过滤算法的向量化实现，以及利用该算法可以做的其他事情。

利用协同过滤算法，我们可以做到的事包括：

1. 当给出一件产品时，找到与之相关的其它产品。
2. 一位用户最近看上一件产品，找到其它相关的产品推荐给他。

继续以电影推荐系统为例：

我们有关于五部电影的数据集，我将要做的是，将这些用户的电影评分，进行分组并存到一个矩阵中。

我们有五部电影，以及四位用户，那么 这个矩阵 Y 就是一个5行4列的矩阵，它将这些电影的用户评分数据都存在矩阵里：

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)
Love at last	5	5	0	0
Romance forever	5	?	?	0
Cute puppies of love	?	4	0	?
Nonstop car chases	0	0	5	4
Swords vs. karate	0	0	5	?

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 5 & ? & ? & 0 \\ ? & 4 & 0 & ? \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{bmatrix}$$

对于用户 j 和电影 i，我们预测评分为： $(\theta^{(j)})^T x^{(i)}$

$$\begin{bmatrix} (\theta^{(1)})^T(x^{(1)}) & (\theta^{(2)})^T(x^{(1)}) & \dots & (\theta^{(n_u)})^T(x^{(1)}) \\ (\theta^{(1)})^T(x^{(2)}) & (\theta^{(2)})^T(x^{(2)}) & \dots & (\theta^{(n_u)})^T(x^{(2)}) \\ \vdots & \vdots & \vdots & \vdots \\ (\theta^{(1)})^T(x^{(n_m)}) & (\theta^{(2)})^T(x^{(n_m)}) & \dots & (\theta^{(n_u)})^T(x^{(n_m)}) \end{bmatrix}$$

找到相关影片：

对于每一部电影x_i，我们寻找另一部电影$x^{(j)}$，依据两部电影的特征向量之间的距离$\left\| x^{(i)} - x^{(j)} \right\|$的大小排序。如图所示，选出距离最小的5个作为我们推荐的电影。

For each product i, we learn a feature vector $x^{(i)} \in \mathbb{R}^n$.

→ x_1 = romance, x_2 = action, x_3 = comedy, x_4 = ...

How to find movies j related to movie i?

small $\|x^{(i)} - x^{(j)}\|$ → movie j and i are "similar"

5 most similar movies to movie i:
Find the 5 movies j with the smallest $\|x^{(i)} - x^{(j)}\|$.

现在我们有一个很方便的方法来度量两部电影之间的相似性。例如说：电影 i 有一个特征向量$x^{(i)}$，我们尝试找到一部不同的电影 j，使得两部电影的特征向量之间的距离很小，那就能表明电影i和电影 j 在某种程度上有相似，这样我们就能给用户推荐几部不同的电影了。

均值归一化

让我们来看下面的用户评分数据：

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	Eve (5)
Love at last	5	5	0	0	?
Romance forever	5	?	?	0	?
Cute puppies of love	?	4	0	?	?
Nonstop car chases	0	0	5	4	?
Swords vs. karate	0	0	5	?	?

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{bmatrix}$$

如果我们新增一个用户 **Eve**，并且 **Eve** 没有为任何电影评分，那么我们以什么为依据为**Eve**推荐电影呢?

我们首先需要对结果 Y 矩阵进行均值归一化处理，将每一个用户对某一部电影的评分减去所有用户对该电影评分的平均值：

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{bmatrix} \quad \mu = \begin{bmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix} \rightarrow Y = \begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 & ? \\ 2.5 & ? & ? & -2.5 & ? \\ ? & 2 & -2 & ? & ? \\ -2.25 & -2.25 & 2.75 & 1.75 & ? \\ -1.25 & -1.25 & 3.75 & -1.25 & ? \end{bmatrix}$$

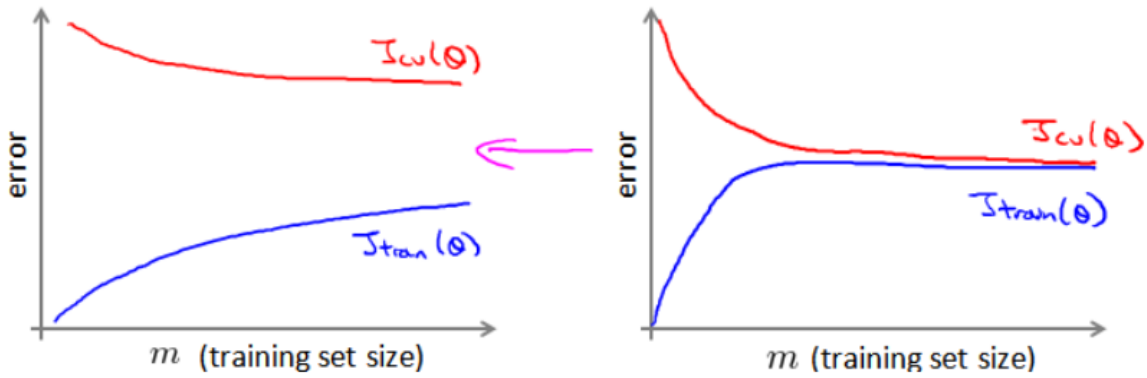
然后我们利用这个新的 Y 矩阵来训练算法。
如果我们要用新训练出的算法来预测评分，则需要将平均值重新加回去，预测$(\theta^{(j)})^T x^{(i)} + \mu_i$，对于**Eve**，我们的新模型会认为她给每部电影的评分都是该电影的平均分。

大规模机器学习(Large Scale Machine Learning)

大型数据集的学习

大型数据集对于机器学习有着巨大的优势，但同时也带来新的挑战。以线性回归模型为例，每一次梯度下降迭代，我们都需要计算训练集的误差的平方和，如果我们的学习算法需要有20次迭代，这便已经是非常大的计算代价。

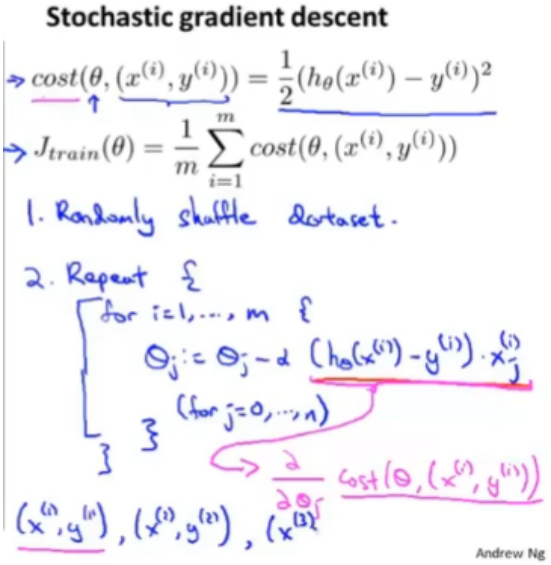
首先应该做的事是去检查一个这么大规模的训练集是否真的必要，也许我们只用1000个训练集也能获得较好的效果，我们可以绘制学习曲线来帮助判断。



随机梯度下降法

如果我们一定需要一个大规模的训练集，我们可以尝试使用随机梯度下降法来代替批量梯度下降法。

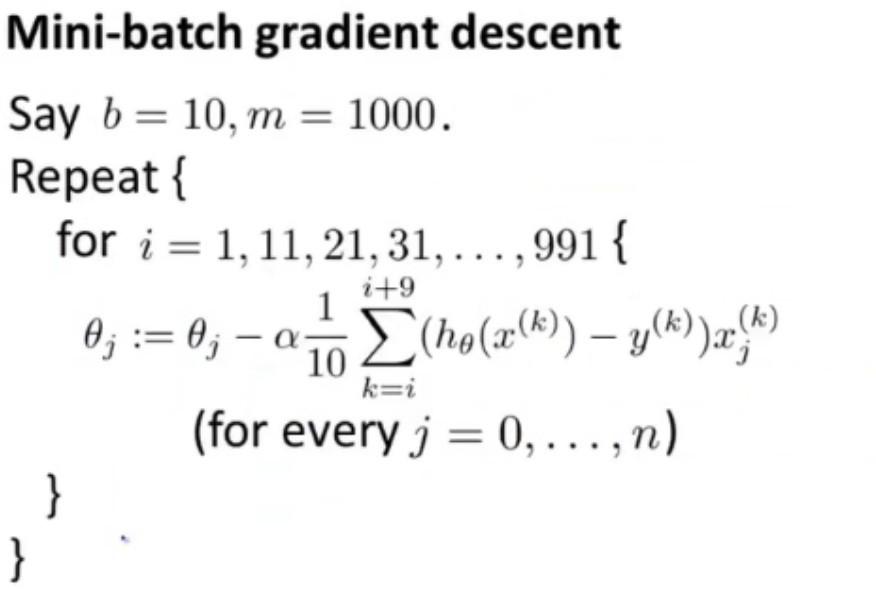
在随机梯度下降法中，我们定义代价函数为一个单一训练实例的代价。**随机**梯度下降算法为：首先对训练集随机“洗牌”，然后在每一次循环中，我们每训练一个样本，我们就把参数稍微修改一下，使其对当前这个样本拟合的效果更好一些。从第一个样本开始，接着是第二个第三个每一次我们对参数做一点小的调整，直到完成所有的训练集。



随机梯度下降算法在每一次计算之后便更新参数 θ，而不需要首先将所有的训练集求和，在梯度下降算法还没有完成一次迭代时，随机梯度下降算法便已经走出了很远。但是这样的算法存在的问题是，不是每一步都是朝着"正确"的方向迈出的。因此算法虽然会逐渐走向全局最小值的位置，但是可能无法站到那个最小值的那一点，而是在最小值点附近徘徊。

小批量梯度下降

小批量梯度下降算法是介于批量梯度下降算法和随机梯度下降算法之间的算法，每计算常数b次训练实例，便更新一次参数 θ。

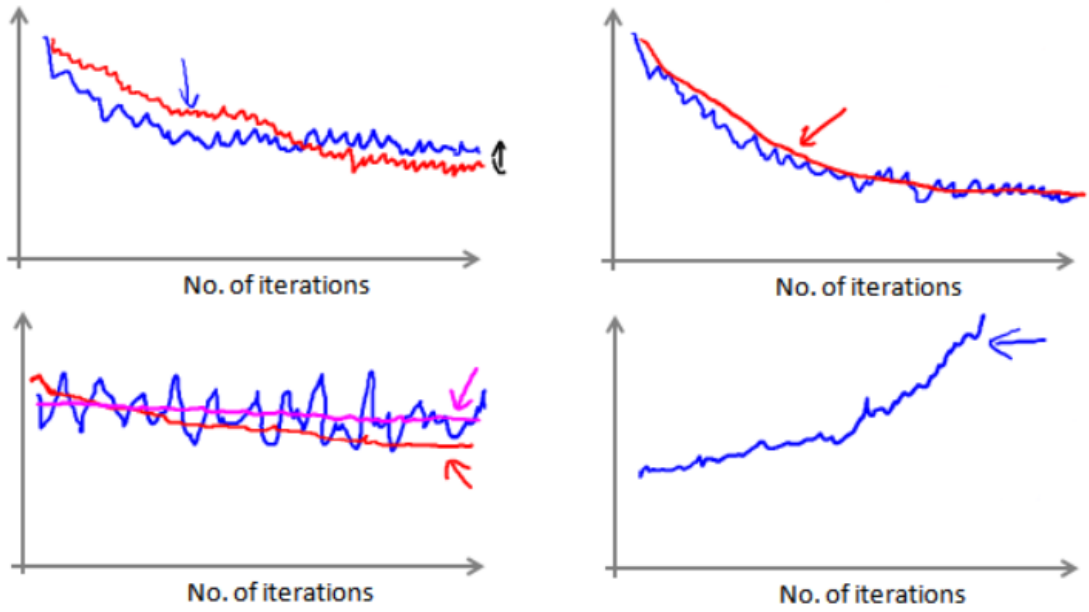


随机梯度下降收敛

现在我们介绍随机梯度下降算法的调试，以及学习率 α 的选取。

在批量梯度下降中，我们可以令代价函数J为迭代次数的函数，绘制图表，根据图表来判断梯度下降是否收敛。但是，在大规模的训练集的情况下，这是不现实的，因为计算代价太大了。

在随机梯度下降中，我们在每一次更新 θ 之前都计算一次代价，然后每x次迭代后，求出这x次训练实例计算代价的平均值，然后绘制这些平均值与x次迭代的次数之间的函数图表。



当我们绘制这样的图表时，可能会得到一个颠簸不平但是不会明显减少的函数图像（如上面左下图中蓝线所示）。我们可以增加α来使得函数更加平缓，也许便能看出下降的趋势了（如上面左下图中红线所示）；或者可能函数图表仍然是颠簸不平且不下降的（如粉红色线所示），那么我们的模型本身可能存在一些错误。

如果我们得到的曲线如上面右下方所示，不断地上升，那么我们可能会需要选择一个较小的学习率α。

我们也可以令学习率随着迭代次数的增加而减小。

$$\alpha = \frac{const1}{iterationNumber + const2}$$

随着我们不断地靠近全局最小值，通过减小学习率，我们迫使算法收敛而非在最小值附近徘徊。
但是通常我们不需要这样做便能有非常好的效果了，对α进行调整所耗费的计算通常不值得

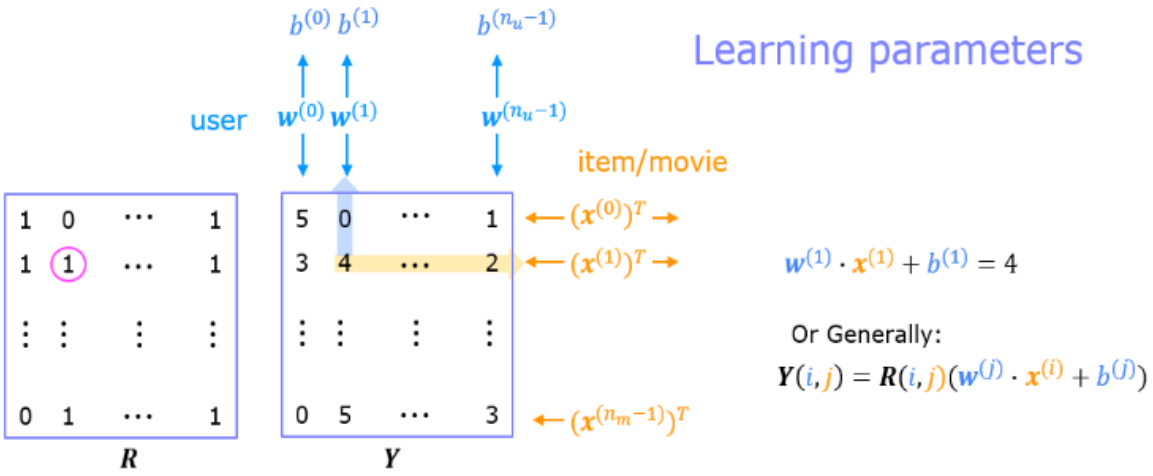
实验（Collaborative Filtering Recommender Systems）

在本次实验中，我们将创建一个基于协同过滤算法的推荐系统来实现电影推荐的功能。

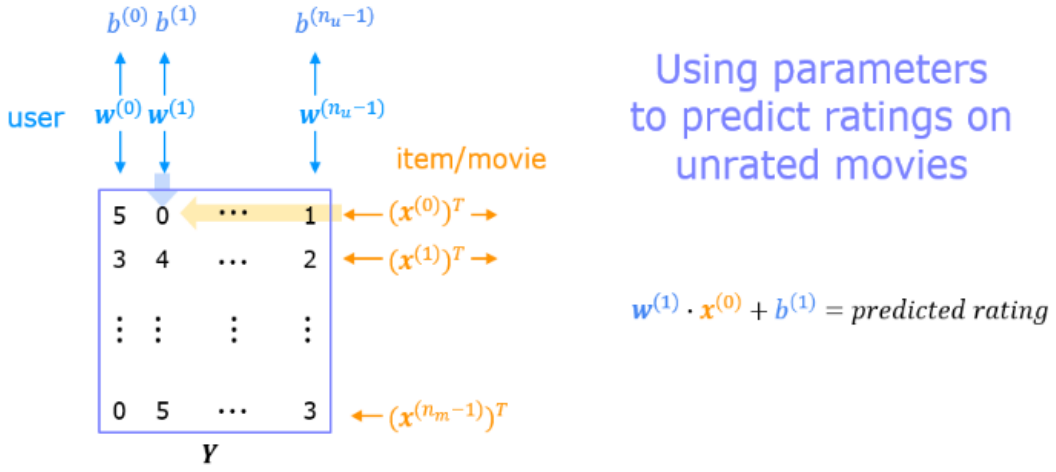
本实验中用到的参数如下：

General Notation	Description	Python
$r(i,j)$	scalar; = 1 if user j rated game i = 0 otherwise	
$y(i,j)$	scalar; = rating given by user j on game i (if r(i,j) = 1 is defined)	
$w^{(j)}$	vector; parameters for user j	
$b^{(j)}$	scalar; parameter for user j	
$x^{(i)}$	vector; feature ratings for movie i	
n_u	number of users	num_users
n_m	number of movies	num_movies
n	number of features	num_features
$\mathbf{X}$	matrix of vectors $x^{(i)}$	X
$\mathbf{W}$	matrix of vectors $w^{(j)}$	W
$\mathbf{b}$	vector of bias parameters $b^{(j)}$	b
$\mathbf{R}$	matrix of elements $r(i,j)$	R

如图所示为我们的推荐系统学习参数的示意图。在协同过滤算法中，我们需要同时学习$w^{(j)}$，$b^{(j)}$和$x^{(i)}$。



但我们获取到了学习后的参数，就可以用来预测某个用户对某部电影的打分。



代码部分

第一步，先编写损失函数的代码。损失函数用J来表示。在计算损失函数的过程中，我们分为两种情况，第一种没有**regularization term**，或者可以理解为λ为0。第二种是添加了**regularization term**的情况。

```
    ### START CODE HERE ###
    for j in range(nu):
        w = W[j,:]
        b_j = b[0,j]
        for i in range(nm):
            x = X[i,:]
            y = Y[i,j]
            r = R[i,j]
            J += r * np.square((np.dot(w,x) + b_j - y ))
    J += (lambda_) * (np.sum(np.square(W)) + np.sum(np.square(X)))
    J = J/2
    ### END CODE HERE ###

    return J
```

第二步，在如图所示的红框中，调用定义好的损失函数来计算J并打印，两种计算J的方法得到的结果与预期的数值相等。

In [7]:
```
# Reduce the data set size so that this runs faster
num_users_r = 4
num_movies_r = 5
num_features_r = 3

X_r = X[:num_movies_r, :num_features_r]
W_r = W[:num_users_r,  :num_features_r]
b_r = b[0, :num_users_r].reshape(1,-1)
Y_r = Y[:num_movies_r, :num_users_r]
R_r = R[:num_movies_r, :num_users_r]

# Evaluate cost function
J = cofi_cost_func(X_r, W_r, b_r, Y_r, R_r, 0);
print(f"Cost: {J:0.2f}")
```

```
Cost: 13.67
```

Expected Output (lambda = 0): 13.67.

In [8]:
```
# Evaluate cost function with regularization
J = cofi_cost_func(X_r, W_r, b_r, Y_r, R_r, 1.5);
print(f"Cost (with regularization): {J:0.2f}")
```

```
Cost (with regularization): 28.09
```

Expected Output:

28.09

第三步，我们使用**Vectorized Implementation**的方法来代替原来的**non-vectorized version**。

使用**Vectorized Implementation**的方法可以得到原来的预期结果。

In [9]:
```
def cofi_cost_func_v(X, W, b, Y, R, lambda_):
    """
    Returns the cost for the content-based filtering
    Vectorized for speed. Uses tensorflow operations to be compatible with custom tr
    Args:
      X (ndarray (num_movies,num_features)): matrix of item features
      W (ndarray (num_users,num_features)) : matrix of user parameters
      b (ndarray (1, num_users)            : vector of user parameters
      Y (ndarray (num_movies,num_users)    : matrix of user ratings of movies
      R (ndarray (num_movies,num_users)    : matrix, where R(i, j) = 1 if the i-th m
      lambda_ (float): regularization parameter
    Returns:
      J (float) : Cost
    """
    j = (tf.linalg.matmul(X, tf.transpose(W)) + b - Y)*R
    J = 0.5 * tf.reduce_sum(j**2) + (lambda_/2) * (tf.reduce_sum(X**2) + tf.reduce_
    return J
```

In [10]:
```
# Evaluate cost function
J = cofi_cost_func_v(X_r, W_r, b_r, Y_r, R_r, 0);
print(f"Cost: {J:0.2f}")

# Evaluate cost function with regularization
J = cofi_cost_func_v(X_r, W_r, b_r, Y_r, R_r, 1.5);
print(f"Cost (with regularization): {J:0.2f}")
```

```
Cost: 13.67
Cost (with regularization): 28.09
```

Expected Output: Cost: 13.67
Cost (with regularization): 28.09

第四步，利用算法来训练模型，以此达到电影推荐的目的。我们先选择一些我们感兴趣的电影，以及一些我们不喜欢的电影，并且分别给出评分。

```
movieList, movieList_df = load_Movie_List_pd()

my_ratings = np.zeros(num_movies)          #  Initialize my ratings

# Check the file small_movie_list.csv for id of each movie in our dataset
# For example, Toy Story 3 (2010) has ID 2700, so to rate it "5", you can set
my_ratings[2700] = 5

#Or suppose you did not enjoy Persuasion (2007), you can set
my_ratings[2609] = 2;

# We have selected a few movies we liked / did not like and the ratings we
# gave are as follows:
my_ratings[929]  = 5   # Lord of the Rings: The Return of the King, The
my_ratings[246]  = 5   # Shrek (2001)
my_ratings[2716] = 3   # Inception
my_ratings[1150] = 5   # Incredibles, The (2004)
my_ratings[382]  = 2   # Amelie (Fabuleux destin d'Amélie Poulain, Le)
my_ratings[366]  = 5   # Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter
my_ratings[622]  = 5   # Harry Potter and the Chamber of Secrets (2002)
my_ratings[988]  = 3   # Eternal Sunshine of the Spotless Mind (2004)
my_ratings[2925] = 1   # Louis Theroux: Law & Disorder (2008)
my_ratings[2937] = 1   # Nothing to Declare (Rien à déclarer)
my_ratings[793]  = 5   # Pirates of the Caribbean: The Curse of the Black Pearl (20(
my_rated = [i for i in range(len(my_ratings)) if my_ratings[i] > 0]

print('\nNew user ratings:\n')
for i in range(len(my_ratings)):
    if my_ratings[i] > 0 :
        print(f'Rated {my_ratings[i]} for  {movieList_df.loc[i,"title"]}');
```

第五步，加载Y和R矩阵，并往其中添加新的评分。同时对数据集做归一化处理。

```
In [12]: # Reload ratings and add new ratings
         Y, R = load_ratings_small()
         Y    = np.c_[my_ratings, Y]
         R    = np.c_[(my_ratings != 0).astype(int), R]

         # Normalize the Dataset
         Ynorm, Ymean = normalizeRatings(Y, R)
```

第六步，初始化参数并选择**Adam**算法。

```
#  Useful Values
num_movies, num_users = Y.shape
num_features = 100

# Set Initial Parameters (W, X), use tf.Variable to track these variables
tf.random.set_seed(1234) # for consistent results
W = tf.Variable(tf.random.normal((num_users,  num_features),dtype=tf.float64),  name
X = tf.Variable(tf.random.normal((num_movies, num_features),dtype=tf.float64),  name
b = tf.Variable(tf.random.normal((1,          num_users),   dtype=tf.float64),  name

# Instantiate an optimizer.
optimizer = keras.optimizers.Adam(learning_rate=1e-1)
```

第七步，开始训练模型。选择迭代的次数为200次

In [14]:
```
iterations = 200
lambda_ = 1
for iter in range(iterations):
    # Use TensorFlow's GradientTape
    # to record the operations used to compute the cost
    with tf.GradientTape() as tape:

        # Compute the cost (forward pass included in cost)
        cost_value = cofi_cost_func_v(X, W, b, Ynorm, R, lambda_)

    # Use the gradient tape to automatically retrieve
    # the gradients of the trainable variables with respect to the loss
    grads = tape.gradient( cost_value, [X,W,b] )

    # Run one step of gradient descent by updating
    # the value of the variables to minimize the loss.
    optimizer.apply_gradients( zip(grads, [X,W,b]) )

    # Log periodically.
    if iter % 20 == 0:
        print(f"Training loss at iteration {iter}: {cost_value:0.1f}")
```

```
Training loss at iteration 0: 2321191.3
Training loss at iteration 20: 136168.7
Training loss at iteration 40: 51863.3
Training loss at iteration 60: 24598.8
Training loss at iteration 80: 13630.4
Training loss at iteration 100: 8487.6
Training loss at iteration 120: 5807.7
Training loss at iteration 140: 4311.6
Training loss at iteration 160: 3435.2
Training loss at iteration 180: 2902.1
```

第八步，计算每一个用户对每一个电影的评分。

In [15]:
```
# Make a prediction using trained weights and biases
p = np.matmul(X.numpy(), np.transpose(W.numpy())) + b.numpy()

#restore the mean
pm = p + Ymean

my_predictions = pm[:,0]

# sort predictions
ix = tf.argsort(my_predictions, direction='DESCENDING')

for i in range(17):
    j = ix[i]
    if j not in my_rated:
        print(f'Predicting rating {my_predictions[j]:0.2f} for movie {movieList[j]}'

print('\n\nOriginal vs Predicted ratings:\n')
for i in range(len(my_ratings)):
    if my_ratings[i] > 0:
        print(f'Original {my_ratings[i]}, Predicted {my_predictions[i]:0.2f} for {mc
```

```
Predicting rating 4.49 for movie My Sassy Girl (Yeopgijeogin geunyeo) (2001)
Predicting rating 4.48 for movie Martin Lawrence Live: Runteldat (2002)
Predicting rating 4.48 for movie Memento (2000)
Predicting rating 4.47 for movie Delirium (2014)
Predicting rating 4.47 for movie Laggies (2014)
Predicting rating 4.47 for movie One I Love, The (2014)
Predicting rating 4.46 for movie Particle Fever (2013)
Predicting rating 4.45 for movie Eichmann (2007)
Predicting rating 4.45 for movie Battle Royale 2: Requiem (Batoru rowaiaru II: Chi
nkonka) (2003)
```