个性化推荐

潘祖江*
清华大学计算机科学与技术系软件所†

17/12/5

1 问题描述

个性化推荐问题是大数据的一个典型应用。个性化推荐，就是利用已知的用户浏览历史推荐新的信息：给定用户行为矩阵 X_{m*n} ，其中 m 是用户数，n 是需要推荐的内容数量，X 中的元素 X_ij 表示用户对某个电影的打分。因此，所谓的推荐任务就转化成，当我们已知 X 中的一部分值时，如何对未知值进行预测。本次 Project，大家的任务是利用视频推荐网站 Netflix 的数据集完成推荐任务。

2 数据集

我们使用的是 Netflix 推荐竞赛的一个子集，包含 10000 个用户和 10000 个电影。用户行为数据包含用户对电影的打分，分数的取值范围是 1-5. 我们选取行为数据的 80% 作为训练集，其余的 20% 作为测试集。具体的文件格式如下：

1. 用户列表 users.txt
   文件有 10000 行，每行一个整数，表示用户的 id，文件对应本次 Project 的所有用户。

*潘祖江 2017210817 pzj17@mails.tsinghua.edu.cn
†地址：北京市海淀区清华大学东主楼

2. 训练集 netflix_train.txt
   文件包含 689 万条用户打分，每行为一次打分，对应的格式为：用户 id 电影 id 分数打分日期其中用户 id 均出现在 users.txt 中，电影 id 为 1 到 10000 的整数。各项之间用空格分开

3. 测试集 netflix_test.txt
   文件包含约 172 万条用户打分，格式与训练集相同。

4. 电影名称 movie_titles.txt
   文件对应每部电影的年份和名称，格式为：电影 id，年份，名称

3 实验内容

本文在 8G 内存，Intel(R) Core(TM) i5 处理器，2.30GHZ 主频，64 位操作系统的 MacBook Pro 上进行实验。

3.1 数据预处理

在大数据处理过程中不得不经历一个步骤：数据清洗及格式化。对应到我们的问题，就是需要将我们的输入文件整理成维度为用户 * 电影的矩阵 X，其中 X_{ij} 对应用户 i 对电影 j 的打分。对于分数未知的项，可以采取一些特殊的处理方法，如全定为 0 或另建一个矩阵进行记录哪些已知哪些未知。这一步的输出为两个矩阵，$X_{train}X_{test}$，分别对应训练集和测试集。本次实验代码均由 Python 实现。
在实验过程中，我所处理的是 10000*10000 的矩阵，即将所有的训练集载入 X_{train} 矩阵。

1. 载入用户 ID

```
def load_users_txt():
    'load txt file in a numpy array, which shape is (10000,)'
    users = np.loadtxt("/Users/datamining/Desktop/hw2/Project2-data/
                                        users.txt")
    # users.reshape(users.shape[0], 1)
    print(users.shape)
    return users
```

2. 载入用户训练数据集

```
def load_netflix_train_txt():
    'load netflix_train.txt as a numpy array,which shape is (
                                        record_number, 3)'
    record_number = 6897746
    netflix_train = np.zeros((record_number, 3))
    with open("/Users/datamining/Desktop/hw2/Project2-data/
                                        netflix_train.txt", "r") as
                                        f:
        i=0
        while True:
            lines = f.readline()
            if not lines:
                break
            strs = lines.split()
            netflix_train[i][0] = int(strs[0])
            netflix_train[i][1] = int(strs[1])
            netflix_train[i][2] = int(strs[2])
            i += 1
    return netflix_train
```

3. 构造 X_{train}

```
def generate_matrix_from_train_set():
    score_matrix = np.zeros((10000,10000))
    netflix_train = load_netflix_train_txt()
    print(netflix_train.shape[0])
    users = load_users_txt()
    previous = None
    index = 0
    for i in range(netflix_train.shape[0]):
        userID = netflix_train[i][0]
        if userID != previous:
            index = int(get_index_by_userID(userID, users))
        score_matrix[index][int(netflix_train[i][1]) - 1] =
                                            netflix_train[i][2]
        previous = userID
        # print(userID, int(netflix_train[i][1]), netflix_train[i][2])
    return score_matrix
```

测试集构造方法如同训练集构造方法, 在此不做赘述。

3.2 协同过滤

协同过滤 (Collaborative Filtering) 是最经典的推荐算法之一，包含基于 use 的协同过滤和基于 item 的协同过滤两种策略。本文所实现的是基于用户的协同过滤算法。算法的思路非常简单，当我们需要判断用户 i 是否喜欢电影 j，只要看与 i 相似的用户，看他们是否喜欢电影 j，并根据相似度对他们的打分进行加权平均。用公式表达，就是:

$$\text{score(i,j)} = \frac{\sum_k \text{sim}\left(X\left(i\right), X\left(k\right)\right) \cdot \text{score}\left(k, j\right)}{\sum_k \text{sim}\left(X\left(i\right), X\left(k\right)\right)}$$

其中，$X(i)$ 为用户 i 对所有 di 电影的打分，对应到我们的问题中，就是 X 矩阵中第 i 行对应的 10000 维向量。$sim(X(i), X(k))$ 表示用户 i 和用户 k 对电影打分的相似度，此处我们采用向量的余弦相似度来表示:

$$\cos\text{(x,y)} = \frac{x \cdot y}{|x| \cdot |y|}$$

计算余弦相似度代码如下所示:

```
def cosine_simi_matrix(score_matrix):
    product = np.dot(score_matrix, score_matrix.T)
    norms = np.sqrt(np.diag(product))
    norms = np.mat(norms)
    norms = np.array(norms)
    normProduct = np.dot(norms.T, norms)
    cosine_simi = product/normProduct
    return cosine_simi
```

我们采用 RMSE（root mean square error，均方根误差）来作为评价指标，计算公式为:

$$\text{RMSE} = \sqrt{\frac{1}{n}\left(\sum_{<i,j>\in Test}\left(X_{ij} - \tilde{X_{ij}}\right)^2\right)}$$

其中 Test 为所有测试样本组成的集合，X_{ij} 为实际值 $\tilde{X_{ij}}$ 为预测值。获得预测矩阵以及计算 RMSE 的代码为:

```
def get_predict_matrix(simi_matrix, score_matrix):
    predict = np.dot(simi_matrix, score_matrix)/np.dot(simi_matrix,(
                                        score_matrix != 0))
    return predict
def final_compute_rmse(predict_matrix, test_score_matrix):
```

```
    A_test = test_score_matrix != 0
    sparse_A_predict = A_test * predict_matrix
    part1 = LA.norm(sparse_A_predict - test_score_matrix,"fro")
    rmse = np.sqrt(part1*part1/1719466)
    return rmse
```

最后由协同过滤算法计算得出的 rmse 值为 1.01836903941,在 129.2543728351593s 内跑完。其中包括数据清洗以及协同过滤的运算时间。

3.3 基于梯度下降的矩阵分解算法

矩阵分解在推荐算法中有很大的应用, 对给定的行为矩阵 X，我们将其分解为 U，V 两个矩阵的乘积，使 UV 的乘积在已知值部分逼近 X, 即: $X_{m\cdot n} \approx U_{m\cdot k}V^T_{n\cdot k}$，其中 k 为隐空间的维度, 为隐空间的维度，是算法的参数。基于行为矩阵的低秩假设，我们可以认为 U 和 V 是用户和电影在隐空间的特征表达，它们的乘积矩阵可以用来预测 X 的未知部分。本实验，我们使用梯度下降法优化求解这个问题。我们的推荐算法的目标函数是:

$$J = \frac{1}{2}\left\|A \circ \left(X - UV^T\right)\right\|_F^2 + \lambda\left\|U\right\|_F^2 + \lambda\left\|V\right\|_F^2$$

其中 A 是指示矩阵，A_{ij} 为 1 意味着 X_{ij} 已知，反之亦然。$\circ$ 是阿达马积（矩阵逐元素相乘）。$\|\cdot\|_F$ 表示矩阵的 Frobenius 范数。计算公式 $\|F\|_F = \sqrt{\sum_i \sum_j a_{ij}^2}$。在目标函数 J 中，第一项为已知值部分，UV 的乘积逼近 X。后面的两项是为了防止过拟合而加入的正则化项,λ 是控制正则化项大小的超参数，由我们自己设置。

当目标函数获得最小值时，算法得到最优解。首先我们队 U 和 V 分别偏导 j 结果如下:

$$\frac{\partial J}{\partial U} = \left(A \circ \left(UV^T - X\right)\right)V + 2\lambda U$$
$$\frac{\partial J}{\partial V} = \left(A \circ \left(UV^T - X\right)\right)^T U + 2\lambda V$$

之后我们对 U 和 V 进行梯度下降，具体算法如下:

算法中 α 为学习率，通常根据具体情况选择 0.0001 到 0.1 之间的值，算法的收敛条件可以选择损失函数 J 的变化量小于某个阈值，至此我们就完成了在给定的 k 和 λ 下，对目标函数 J 进行优化求解的全过程。

求解损失函数的代码如下，此处简单的调用 API 即可，也不需自行推导向量化公式:

Algorithm 1 MATRIX_FACTORIZATION_GRADIENT

Initialize U and V(very small value)

loop until converge:

$$U = U - \alpha \frac{\partial J}{\partial U}$$
$$V = V - \alpha \frac{\partial J}{\partial V}$$

end loop

```
def compute_cost_function(score_matrix, U, V, lambd, A):
    part1 = LA.norm(A * (score_matrix - np.dot(U, V.T)),"fro")
    part2 = LA.norm(U, "fro")
    part3 = LA.norm(V, "fro")
    cost = part1 * part1 / 2 + lambd * part2 *part2 + lambd * part3 *part3
    # print(cost)
    return cost
```

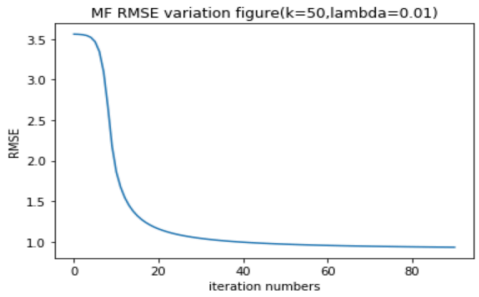
其中 score_matrix 即为训练集，norm 函数是 numpy 下的一个求范数的函数。

求解梯度下降的代码如下：

```
def gradient_descent(X, U, V, lambd, learning_rate, A):
    previous_cost = 0
    # cnt = 0
    while True:
        temp = compute_cost_function(X, U, V, lambd, A)
        print(temp)
        if abs(temp - previous_cost) < 3000:
            break
        previous_cost = temp
        derivative_U = np.dot(A * (np.dot(U, V.T) - X), V) + 2 * lambd * U
        derivative_V = np.dot((A * (np.dot(U, V.T) - X)).T, U) + 2 * lambd *
                                                V
        U = U - learning_rate * derivative_U
        V = V - learning_rate * derivative_V
    return np.dot(U, V.T)
```

其中我们设置为当损失函数的变化值小于 3000 时，算法停止迭代，函数的返回值是预测矩阵。最后算出来的 RMSE 是 0.931651588415，消耗时间是 398.10307002067566s。给定 k=50， = 0.01 的情况，迭代过程中目标函数值和测试集上 RMSE 的变化如图 1 所示，由图可知刚开始 RMSE 值大于

图 1: 基于梯度下降算法 RMSE 迭代变化情况



3.5，开始时 rmse 降低速度较慢，在第 5 次到第 30 次迭代之间 RMSE 下降速度很快，而当 RMSE 接近于 1 时，迭代过程中 RMSE 的变化明显变小。

表 1: 不同的 k 以及 λ 对 RMSE 的影响

k	λ	RMSE
50	0.001	0.931564069223
50	0.01	0.931651588415
50	0.1	0.932065789951
20	0.001	0.932039832809
20	0.01	0.93204399725
20	0.1	0.932085719556

由表 1 可以看出，对 k 以及 λ 以及 k 进行变动对 RMSE 的影响很小，最后都会收敛到一个近似的值，而基于梯度下降的矩阵分解的方法明显是优于基于协同过滤的推荐算法。这是由于基于梯度下降的算法有更多的参数可以调节。

4 总结

本文研究了个性化推荐的实现方法，分别实现了基于用户的协同过滤的方法以及基于梯度下降的矩阵分解方法，基于用户的协同过滤方法时间效率更高，但是效果差于基于梯度下降的矩阵分解方法。