大数据课程第二次项目作业

计算机科学与技术一班 洪一帆 *1811363*
计算机科学与技术一班 丁一凡 *1811348*
计算机卓越班 林铸天 *1811554*

2021 年 6 月 20 日

目录

第 1 章　问题重述和本组实现

本次作业需要利用不同算法来对用户打分进行估计,我们组在合理利用`ItemAttribute.txt`的基础上，分别利用不同算法进行求解。按照作业要求，本次结果我们汇报：

序号	项目主要特点	是否作业要求	本组完成情况
1	实现基于协同过滤的预测算法	√	√
2	实现基于 SVD 的预测算法	√	√
3	算法评价与分析	√	√
4	利用`ItemAttribute.txt` 优化打分预测		√
5	存储模长以及相关性矩阵，加速推理		√
6	实现使用全局平均信息的预测算法		√

表 1.1: 作业要求及本组完成情况

1. 源代码、报告
2. CF 算法的参数文件，存放于 data 目录下
3. 格式化好的结果文件,分别存放于`./result_RSVD.txt`以及`./result_CF.txt`
4. 可执行文件

第 2 章　项目核心框架介绍

本次项目核心代码为如下两个文件`CF.py`、`RSVD.py`。下面介绍文件结构：

- `CF.py`:
  - `calculate_data_bias`: 计算全局 bias 信息。
  - `global_model`: 以不同评分均值预测，包括总体均值，用户评分均值，item 评分均值及综合均值。
  - `collaborative_filtering_bias`: 协同过滤算法主函数。
  - `predict`: 预测算法。
  - `exec`: 封装好的主入口。

- RSVD.py:
  - build_lr: 构造动态下降的学习率 list。
  - gradient_descent: 梯度下降算法，更新矩阵参数。
  - cal_rmse: 计算 rmse。
- ./dataset/: RSVD 算法所需数据以及结果保存目录。
- ./data/: CF 算法所需数据以及结果保存目录。
- result_RSVD.txt: RSVD 算法生成的格式化好的结果文件。（改文件不会更新，更新后的结果存储在相应目录下对应文件，若需要请阅读下文）
- result_CF.txt: CF 算法生成的格式化好的结果文件。（改文件不会更新，更新后的结果存储在相应目录下对应文件，若需要请阅读下文）
- rsvd.exe: RSVD 算法可执行文件。
- cf.exe: CF 算法可执行文件，直接利用参数进行推理。

第 3 章　数据集概览

```
train set
用户数量： 19835
商品数量： 455705
评分数量： 5001507

% 为了能够预测模型的优劣，我们将 train set 按照 9:1 切分为 train data
↪ 和 validate data。
% 并且为了预测的鲁棒性，我们保证了 train data 中包含了所有的品种，使得我
↪ 们的训练结果不会对某种商品是茫然无知的。

validate data
用户数量： 455124
商品数量： 118599
评分数量： 455124

train data
```

```
用户数量： 4546383
商品数量： 455705
评分数量： 4546383

test data
用户数量： 19835
商品数量： 28292
评分数量： 119010

item attribute
商品数量： 624961
商品属性统计信息：
count    624961.000000
mean     234039.891956
std      207878.812544
min           0.000000
25%           0.000000
50%      206339.000000
75%      418894.000000
max      624943.000000
Name: attribute1, dtype: float64
count    624961.000000
mean     221134.948469
std      207181.520174
min           0.000000
25%           0.000000
50%      188235.000000
75%      401338.000000
max      624951.000000
Name: attribute2, dtype: float64
```

第 4 章　项目配置方法

4.1　运行 exe

您可以直接运行根目录下的两个 exe 程序，无需变动项目结构。

输出结果分别存放于./dataset/test1.txt以及./data/result_CF.txt（分别对应 RSVD 和 CF）

4.2 使用 Python3 解释器运行源代码

本部分供教授和助教检查所用。

1. 首先，请您保证两个 python 文件、exe 文件与 data、dataset 文件夹处于同一目录下。您可以用您本地的 python 解释器运行，也可以直接运行我们打包好的 exe 程序。

2. 运行RSVD.py:

   (a) 对于 RSVD 算法，您可以在 main 函数开头调整您所需要的参数，我们已经预设了一些值。

   (b) 您可以运行 RSVD 的 main 函数，并在训练结束时进行打分预测，结果存放于./dataset/test1.txt

3. 运行CF.py:

   (a) 程序入口已经封装在了 exec 函数中，您可以在 CF.py 的 main 函数中进行路径调整。

   (b) 您可以在 exec 函数中选择运行不同的环节，包括训练全局平均模型、训练协同过滤模型、加载预训练参数并直接预测。

   (c) 预测结果存放于 data/result_CF.txt 中。

第 5 章 实验思路及实践路径

5.1 数据预处理

5.1.1 读取数据

根据不同的算法的需求，我们对于读取数据的策略有细节上的不同（并且命名方式有些许差异，但是阅读源码时您一定能理解，因为命名都很直接），但是大致思路如下：

Algorithm 1 读取数据及预处理

Input: filepath

Output: dataframe(RSVD 算法) or dict(CF 算法)

1: **function** LOAD_DATA($self, filepath$)

```
    打开文件
    while readline 不为空 do
        if 识别到了"|" then
            记录用户 id 以及打分个数
        else
            如果是 nan，那么我们特殊处理
            将下面的评分结果以及用户 id 以不同的方式存储起来（dataframe 或者 dict）
        end if
    end while
end function
```

5.1.2 写出格式化结果

实现思路较为简单，这里不再赘述。

关键点是一些特殊值的处理，如防止结果出现小于 0 大于 100 等少数情况。

5.1.3 划分训练集以及验证集

按照 divide_size 定义的比例划分成 train 与 validate 数据集，同时保证划分后 train 中包含所有 item

原因是后续需要计算 item 与 item 之间相似度，如果 train 中不存在则无法计算，影响效果。

5.1.4 预处理 `ItemAttribute`

我们在实现的时候，也是先将数据存储在 dataframe 中，方便对一些特殊值做处理。

之后为了查询效率（dataframe 查询检索真的很慢很慢），我们将其转为 dict 存储，利用底层的 hash 来使得检索复杂度降低到 O(1)。

此外，比较重要的一点是，我们为了加速预测，将很多中间结果存储在了硬盘中。在预处理 `ItemAttribute` 的过程中，我们为了方便地计算 item 之间的相似度，将每个物品的模长提前计算好并且存储到磁盘中，方便下次直接使用。

代码如下：

```
self.item_attributes["norm"] =
  (self.item_attributes["attribute1"] ** 2
  + self.item_attributes["attribute2"] ** 2) ** (1 / 2)
```

5.1.5 计算 bias

为了引入用户的打分偏好以及物品的受欢迎程度，我们需要计算他们与平均评分之间的差值，以对预测结果进行修正。

Algorithm 2 计算 bias

Input: train_data

Output: 全局平均值，dict{用户 id: 打分偏好偏移}，dict{物品 id: 打分偏好偏移}

1: **function** CALCULATE_DATA_BIAS($self$)
2: 求全局平均
3: 求每个用户平均打分与全局平均的差值，存入 dict
4: 求每个物品平均打分与全局平均的差值，存入 dict
5: **end function**

5.2 方法 1：全局模型 +bias

该算法思路较为简单，就是利用之前计算好的各个均值、偏移信息来构建预测结果。

而实现这个算法的出发点也是很朴素的，就是寻找最简单的规律（平均值、偏移平均值）来对未知进行预测。

1. `Global average`：所有评分的均值作为预测结果，即仅有一个预测值。RMSE: 38.13540922065373
2. `User average`：所有评分的均值作为预测结果 + 用户偏移。RMSE: 29.552371104008376
3. `Item average`：所有评分的均值作为预测结果 + 物品偏移。RMSE: 35.82071948493244
4. `Global effects`：所有评分的均值作为预测结果 + 用户偏移 + 物品偏移。RMSE: 30.946452267391507

Average running time: 2.4483330249786377 Seconds

5.3 方法 2：协同过滤 +bias

该算法需要利用到之间计算好的模长数据。

此外，比较重要的一点是，我们会在程序运行中间阶段计算物品间的相似度并存储在磁盘中，如果下次再次碰到计算这两个物品相似度的情况，我们可以快速读取之前计算好的结果。并且由于相似度是相互的，因此为了节省磁盘开销，我们只需要存储对称矩阵的一半即可。

[注]：由于我们在测试的时候是使用自己切割的 validation 集而非 `test.txt`，因此为了方便地处理两种情况,项目中包含了两个函数 `collaborative_filtering_bias`

以及 `predict()`。但是实现原理是相同的，简化起见以下只介绍 `collaborative_filtering_bia`
您在测试的时候将会使用到 predict 函数来和真实的测试集进行比较。

Algorithm 3 协同过滤 +bias

Input: 我们切分的 train_data 以及 validate_data，之前计算的 bias、相似度中间结果
Output: RMSE

function COLLABORATIVE_FILTERING_BIAS($self$)
  for 遍历数据集 **do**
    计算偏置 b_x，包含全局平均信息以及物品偏好偏移，用于修正物品相关性。
    for 与该物品相关的其他物品 **do**
      if 已经计算过相似度 **then** break
      else
        进行相似度计算：
        属性相似度为 cosine 相似度。
        pearson 相似度用到 pearson 相关系数，并且需要保证用户评分数高于 20 避免出现较大偏差。
      end if
    end for
    取最相似的 100 个物品 (If Any)
    for 遍历所有相似物品以求取加权平均 **do**
      更新偏置，加入用户偏好偏移。
      加入到平均值中。
    end for
  end for
  计算 RMSE 误差。
  return RMSE
end function

实现代码：

```
def collaborative_filtering_bias(self):
  # 协同过滤算法
  predict_rate = []
  for index, row in enumerate(self.train_test_data.values):
    user, item_x, pred_score_x = row
    score_x = 0
    b_x = self.bias["deviation_of_item"][item_x] +
     ↪  self.bias["overall_mean"]
```

```
similar_item = {}
# 计算物品相似度
for item_y in self.user_item_train_data[user].keys():
  if item_x in self.similarity_map and item_y in
   ↪  self.similarity_map[item_x]:
    similar_item[item_y] =
     ↪  self.similarity_map[item_x][item_y]
  elif item_y in self.similarity_map and item_x in
   ↪  self.similarity_map[item_y]:
    similar_item[item_y] =
     ↪  self.similarity_map[item_y][item_x]
  else:
    b_y = self.bias["deviation_of_item"][item_y] +
     ↪  self.bias["overall_mean"]
    if self.item_attributes[item_x][2] == 0 or
     ↪  self.item_attributes[item_y][2] == 0:
      attribute_similarity = 0
    else:
      attribute_similarity =
       ↪  (self.item_attributes[item_x][0] *
       ↪  self.item_attributes[item_y][0]
                  + self.item_attributes[item_x][1] *
                   ↪  self.item_attributes[item_y][1]) \
                / (self.item_attributes[item_x][2] *
                 ↪  self.item_attributes[item_y][2])
    norm_x = 0
    norm_y = 0
    pearson_similarity = 0
    count = 0
    for same_user, score in
     ↪  self.item_user_train_data[item_x].items():
      if same_user not in
       ↪  self.item_user_train_data[item_y]:
        continue
      count += 1
```

```
        pearson_similarity +=
         ↪  (self.item_user_train_data[item_x][same_user] -
         ↪  b_x) * (
            self.item_user_train_data[item_y][same_user] -
             ↪  b_y)
        norm_x +=
         ↪  (self.item_user_train_data[item_x][same_user] -
         ↪  b_x) ** 2
        norm_y +=
         ↪  (self.item_user_train_data[item_y][same_user] -
         ↪  b_y) ** 2
      if count < 20:
        pearson_similarity = 0
      if pearson_similarity != 0:
        pearson_similarity /= (norm_x * norm_y) ** (1 / 2)

      similarity = (pearson_similarity +
       ↪  attribute_similarity) / 2

      if item_x not in self.similarity_map:
        self.similarity_map[item_x] = {}
      self.similarity_map[item_x][item_y] = similarity

      similar_item[item_y] = similarity

  similar_item = sorted(similar_item.items(), key=lambda
   ↪  item: item[1], reverse=True)
  b_x = self.bias['overall_mean'] +
   ↪  self.bias['deviation_of_item'][item_x] +
   ↪  self.bias['deviation_of_user'][
    user]
  norm = 0
  for i, (item_y, similarity) in enumerate(similar_item):
    if i > 100:
      break
```

```
      b_y = self.bias['overall_mean'] +
       ↪  self.bias['deviation_of_item'][item_y] + \
        self.bias['deviation_of_user'][user]
      score_x += similarity *
       ↪  (self.item_user_train_data[item_y][user] - b_y)
      norm += similarity
    if norm == 0:
      score_x = 0
    else:
      score_x /= norm
    score_x += b_x
    score_x = score_x if score_x > 0 else 0
    score_x = score_x if score_x < 100 else 100
    predict_rate.append(score_x)
    if index % 500 == 0 and index != 0:
      print(" 已预测", index)
    if index % 5000 == 0 and index != 0:
      print("RMSE: ", cal_RMSE(predict_rate,
       ↪  self.train_test_data['score'][:index + 1]))
    if index % 200000 == 0 and index != 0:
      with open("data/similarity_map.pickle", 'wb') as f:
        pickle.dump(self.similarity_map, f)
  print("RMSE: ", cal_RMSE(predict_rate,
   ↪  self.train_test_data['score']))
```

5.4 方法 3：RSVD 及矩阵初始化策略

5.4.1 初始化策略

为了保证初始化结果能够具有一个可以接受的上界，首先考虑打分一定介于 $[0, 100]$，因此若对于 $\forall u, i$，我们都假设初始化为 $\hat{A}_{ui}^{(0)} = 50$，则一定有：

$$RMSE = \sqrt{\frac{\sum_{dataset}(A - 50)}{n}} <= 50$$

所以我们初始化时已然具有上界 50，不会比它更差了。此时，假设 P, Q 矩阵每个元素都相等且为 a。

$\forall u, i$ 则：

$$\hat{A}_{ui} = P[u, :]Q[:, i] = fa^2 = 50$$

求解得：

$$a = \sqrt{\frac{50}{f}}$$

因此我们的代码为：

```
U = max(max(train_df['user']), max(validation_df['user'])) + 1
I = max(max(train_df['item']), max(validation_df['item'])) + 1
P = np.ones(U, factor)* (np.sqrt(np.mean(train_df['score']) /
 ↪  factor))
Q = np.ones(factor, I)* (np.sqrt(np.mean(train_df['score']) /
 ↪  factor))
```

在验证集中，初始化时就能达到 RMSE=38 的效果。

5.4.2 建模过程

我们定义用户数量为 U，商品数量为 I，factor 定义为 F。我们定义预测矩阵：

$$\hat{A} = P_{U\times F} \times Q_{F\times I}$$

对于评价指标，我们定义：

$$RMSE = \sqrt{\frac{\sum_{dataset}(A-\hat{A})^2}{n}}$$

对于目标函数，我们定义为：

$$L = \sum_{dataset} (A-\hat{A})^2 + \lambda_p \|P[u,:]\| + \lambda_q \|Q[:,i]\|$$

根据梯度下降法，我们操作：

$$Q[:,i] := Q[:,i] - \eta \nabla Q[:,i]$$

$$P[u,:] := P[u,:] - \eta \nabla P[u,:]$$

根据矩阵求导法则，可以进一步推得：

$$Q[:,i] := \eta(e_{ui}P[u,:] - \lambda_q Q[:,i])$$

$$P[u,:] := \eta(e_{ui}Q[:,i] - \lambda_q P[u,:])$$

其中

$$e_{ui} = A_{ui} - \hat{A}_{ui}$$

$$\hat{A}_{ui} = P[u,:] \cdot Q[:,i]$$

以下是具体实现的核心代码：

```
def gradient_descent(P, Q, search_df, lambda_p, lambda_q,
 ↪  lr=0.01):
  ...
  for _, line in enumerate(search_df.values):  #
   ↪  enumerate(search_df.iterrows()):
    sys.stdout.write('\r梯度下降更新进度-->' + str(round(_ /
     ↪  len(search_df), 4)))
    sys.stdout.flush()
    u = line[0]
    i = line[1]
    s = line[2]
    eui = s - np.matmul(P[u, :], Q[:, i])

    Q[:, i] += lr * eui * P[u, :].T - lambda_q * Q[:, i] * lr
    P[u, :] += lr * eui * Q[:, i].T - lambda_p * P[u, :] * lr
  ...

def cal_rmse(ds, P, Q):
  ...
  true_val = np.array(ds['score'])
  pred_val = []
  for _i, line in enumerate(ds.values):
    u = line[0]
    i = line[1]
    pred_val.append(np.matmul(P[u, :], Q[:, i]))

  rmse = (np.sum((np.array(pred_val) - true_val) ** 2) /
   ↪  len(true_val)) ** 0.5
  ...
```

第 6 章　不同算法结果对比

[注]：除了 CF 算法，其余算法均需要先训练再预测，时间之和填写在预测时间一栏。此外，由于前四种算法较为相似，并且本实验重心并不于此，因此我们仅取平均时长作为其时长（实际上每个算法时长也确实差不离）。

算法名称	训练时间	预测时间	RMSE
Global average		2.44s	38.1
User average		2.44s	29.5
Item average		2.44s	35.8
Global effects		2.44s	30.9
CF	10h	342.4s	24.6
RSVD		约 36min	30.2

表 6.1: 不同算法对比

可以看到，利用带 bias 矫正的 CF 算法在我们的训练集和验证集上表现较为良好，并且训练好后推理速度较快。

但是不可忽视的是基于用户平均的算法，简单朴素的思想以及快速的训练推理仍旧能带来较为不错的结果，着实出乎我们的意料。

以下是两种算法预测结果的部分展示，具体结果请查看 `result_CF.txt` 以及 `result_RSVD.txt`。

```
% CF
0|6
208031 78
193714 38
393064 79
207030 91
112040 6
464229 68
1|6
55971 82
583090 74
180171 93
617646 89
175835 95
553890 89
2|6
48916 76
238557 33
61148 0
```

```
378073 32
17863 41
187943 24


% RSVD
0|6
208031 78
193714 69
393064 85
207030 80
112040 69
464229 91
1|6
55971 79
583090 81
180171 87
617646 88
175835 93
553890 87
2|6
48916 58
238557 57
61148 28
378073 67
17863 56
187943 50
```

第 7 章　遇到的问题及解决方法

- 利用 dataframe 进行索引带来的效率低下，后来改用 O(1) 的 dict 解决了这个问题，大大加速了训练。
- 在 SVD 分解中，常常出现由于不同 factor 值的设置导致的 overflow 问题，进而导致 RMSE 为 nan。为此我们选择了较为满意的参数组合来避免这个问题，但是带来的结果就是无法进行 grid search。

第 8 章　总结

本次作业中，小组成员均在代码、实验和报告中有所贡献。

我们利用了不同的算法，从简单到复杂，较为全面地了解了推荐系统的部分主流实现方式。小组成员的科研和工程能力在本次作业中得到了很好地锻炼。同时，存储预处理好的参数来加速程序也是从中获得的处理大数据问题时的经验。