实验二：矩阵分解

（大数据分析课程报告）

姓　　名：肖文韬

学　　号：2020214245

二〇二〇年十二月十四日

目 录

目录.....I
第 1 章 数据整理..... 1
第 2 章 任务一：协同过滤算法..... 3
第 3 章 任务二：矩阵分解算法..... 6
参考文献..... 7

第 1 章　数据整理

数据集每一行为 `<user_id> <movie_id> <rate>` 的格式：

1. user_id: 用户唯一 id，用户数量为 10000，不过用户 id 并不在 $[0, 10000]$ 内，需要自己创建一个 user2id 的映射。
2. movie_id: 电影唯一 id，电影数量为 10000，并且电影 id $\in [1, 10001]$。
3. rate: 电影打分，为 $[0, 5]$ 的整数，其中 0 表示未打分。

观察分析数据后，有以下结论：

1. 训练集和测试集中的数据并不是规整的按电影来划分的，也就是说对于某个电影，训练集和测试集都只有一部分用户打分。并且打分是不重叠的，也就是说如果用户在训练集中打了分，在测试集中就是没打分的，反之亦然。
2. 因为数据规模不是很大，矩阵虽然是 10000×10000 大小的，不过比较稀疏（训练集稀疏度 6.89%，测试集 1.72%），所以可以使用稀疏矩阵来加快速度。

总之，该数据集不算很大，合理使用稀疏矩阵，使用矩阵运算（并行）来代替循环，在显卡上进行矩阵运算，都可以大大的提高处理速度。

故在实验中，采用全量数据。

数据读取部分的代码实现：

```
def read_data(filename):
  user2id = {}
  st = time()
  X = np.zeros((10000, 10000))
  with open(filename) as f:
    for l in f.readlines():
      user, movie, rate, _ = l.split()
      movie = int(movie) - 1
      rate = int(rate)
      if user not in user2id:
        user2id[user] = len(user2id)
      user = user2id[user]
      X[user, movie] = rate
  print(f'Done in {time() - st}s.')
  return X

X_train = read_data('netflix_train.txt')
X_test = read_data('netflix_test.txt')')
```

第 2 章　任务一：协同过滤算法

协同过滤 (Collaborative Filtering) 是一种经典的且被工业界广泛使用推荐算法。狭义的协同过滤定义为：通过收集大量其他**相似**用户的偏好或品味信息（**协同**）来自动地对预测（**过滤**）当前用户的兴趣。广义的协同过滤则定义为：使用多个主体，观点，数据源进行合作来过滤信息或者模式的过程。

协同过滤算法分为两类：

1. 基于用户的协同过滤：构建一个 user-item 矩阵，然后寻找该用户的相似用户在 user-item 矩阵中的信息来预测该用户的偏好结果。
2. 基于目标项目（item）的协同过滤：构建一个 item-item 矩阵来衡量不同项目之间的关联度，然后使用该矩阵来推导用户的偏好。

这里主要是实现一个基于用户的协同过滤算法，基于用户的协同过滤算法的一般定义为（用户 i 对项目 m 的打分）：

$$score(i,m) = \mathrm{agg}_{j \in N}(score(j,m)) \tag{2-1}$$

其中 N 表示 i 的某个邻域（一般以相似度作为度量），agg 用户对该邻域内所有打分结果进行聚合运算（i.e., 加权平均），包括一些预处理（归一化，收缩，等）。

本节将要实现的算法的核心公式为（以预测用户 i 对电影 m 的偏好度为例）：

$$score(i,m) = \frac{\sum_{k \in N(i,m)} sim(X(i), X(k)) \times score(k,m)}{\sum_{k \in N(i,m)} sim(X(i), X(k))} \tag{2-2}$$

其中 $X(i)$ 代表用户 i 对所有电影的打分（当然要预测的电影的打分是为 0 的）向量，$N(i,m)$ 在本例中简单选用为所有给电影 m 打了分的用户，相似度量采用**余弦距离（cosine distance）**：

$$sim(X(i), X(k)) = \frac{\langle X(i), X(k) \rangle}{\|X(i)\| \|X(k)\|} \tag{2-3}$$

公式 2-2 可以转换为矩阵运算的形式：

$$\hat{X} = \frac{S \cdot X}{S \cdot A} \tag{2-4}$$

其中 A 为 X 的指示矩阵，有 $A_{ij} = \mathrm{sgn}(X_{ij})$。

在 Python 上使用 numpy 实现的代码为：

```python
def pairwise_cos_distance(X):
  X = X.T / X.sum(axis=1)
  return X.T @ X

def coll_filter_sp(train: np.ndarray):
  sim = pairwise_cos_distance(train)
  score = sim @ train
  train = train.astype(np.bool)
  Z = sim @ train
  return score_pred
```

虽然转换成矩阵运算之后，相对于简单粗暴的 for 循环实现，速度上已经有了很明显的提升。不过既然是矩阵运算，理说应当更加适合在显卡上进行运算。以下是使用 cupy[①] 实现的版本（是的，基本上就是把 np 改成了 cp）。

```python
def coll_filter_cuda(train: cp.ndarray):
  cp.cuda.Device(3).use()
  X_train_cuda = cp.array(train)
  X_test_cuda = cp.array(test)
  sim = pairwise_cos_distance(X_train_cuda)
  score = sim @ X_train_cuda
  Z = sim @ X_train_cuda.astype(np.bool)
  score_pred = score / Z
  return score_pred
```

RMSE 因为测试样本数不是 $10000 * 10000$ 而是 170 多万，所以最好不要使用 numpy 提供的 RMSE，我们可以自己简单实现一下：

① 一个类似于 numpy 的矩阵运算库，不过所有运算都是在 GPU 上进行的。

```
def rmse(X, X_hat):
  test_point = X.astype(np.bool)
  loss = np.sqrt(np.power(
    X_hat[test_point] - X[test_point], 2
  ).mean())
  return loss
```

表 2.1　协同过滤算法不同实现对比

实现	RMSE	速度（s）
CPU	1.02821	12.94
CPU (k 取所有)	2.60669	7.94
GPU	1.02821	11.79
GPU (k 取所有)	2.60669	7.88

最终计算速度和最终的 RMSE 对比如表 2.1所示。可以看出，GPU 竟然只比 CPU 快一点点，这样不行呀，至于原因我认为还是因为这里面涉及到一些对 GPU 不太友好的运算（例如归一化用到的两次逐位除法），并且 numpy 优化得比较好了，=,= 总之，在第二个任务，矩阵分解任务，我们就能看到 GPU 运算的魅力啦

备注：开头说了数据集比较稀疏，可是为什么我最终没有使用稀疏矩阵运算呢？因为我发现，这个矩阵还是不够大，也不够稀疏，还不如全量矩阵直接用显卡或者 Intel CPU 提供的 AVX 指令集优化地并行加速高效。使用 scipy 提供的 csr 模式的稀疏矩阵运算大概需要耗时 120s。

第 3 章　任务二：矩阵分解算法

参考文献