推荐系统课程实验报告

实验数据集分析

霍超凡

1 数据集规模

表 1 和表 2 给出了一些数据集的规模和稀疏度，一些模型在这些数据集上面跑是十分耗时的，需要进行优化，优化过程无非是空间和时间上的权衡，关于优化我放在后面。原本打算在多个数据集测试算法，后来发现工作量太大，计划选取其中两个具有代表性的数据集 MovieLens 和 Netflix。

表 1：不同规模的 MovieLens 数据集

数据集	时间跨度	用户个数	物品个数	评分记录个数	稀疏度
MovieLens-100K	1997-1998	943	1682	100,000	0.937
MovieLens-1M	In 2000	6,040	3,900	1,000,209	0.957
MovieLens-10M	-	71,567	10,681	10,000,054	0.987
MovieLens-20M	1995-2015	138,493	27,278	20,000,263	0.994
MovieLens-25M	1995-2019	162,541	62,423	25,000,095	0.997

表 2：其它数据集

数据集	数据集类别	用户个数	物品个数	评分记录个数	稀疏度
Book-Crossing	书籍	278,858	271,379	1,149,780	0.999
Jester	笑话	73,421	100	4,100,000	0.441
Personality	电影	1,834	-	1,028,752	-
Netflix	电影	480,000	17,000	100,000,000	0.987

2 数据集用户、物品标签信息

表 3:

数据集	用户标签			电影标签
	年龄	性别	职业	类别
MovieLens-100K	√	√	√	√
MovieLens-1M	√	√	√	√
MovieLens-10M	-	-	-	√
MovieLens-20M	-	-	-	√

MovieLens-25M	-	-	-	√

表 4:

数据集	用户描述	物品标签
Book-Crossing	年龄、地理位置信息	-
Jester	-	-
Personality	性格开放度、情绪敏感性、外向性、自律性等	-
Netflix	-	-

3 数据集评分统计

数据集	分值范围	时间信息			
MovieLens-100K	0-5	√			
MovieLens-1M	0-5	√			
MovieLens-10M					
MovieLens-20M					
MovieLens-25M					

数据集	分值范围	时间信息		
MovieLens	0-5	√		
Book-Crossing	0-10	-		
Jester	-10-10	-		
Personality	0-5	√		
Netflix	0-5	√		

4 数据集的访问与存储

在后续的编程中，越发感受到数据集规模的庞大，这些数据集虽不像图像数据集那样，动辄几十个 G，但是这种数据集条目非常多，有的高达千万条，即使按顺序逐个访问一遍也非常耗时，我们的模型的性能很大程度上受到数据集访问效率的影响，需要在数据集上花费点心思。

4.1 数据集的访问

数据集的访问可分成三种情况：

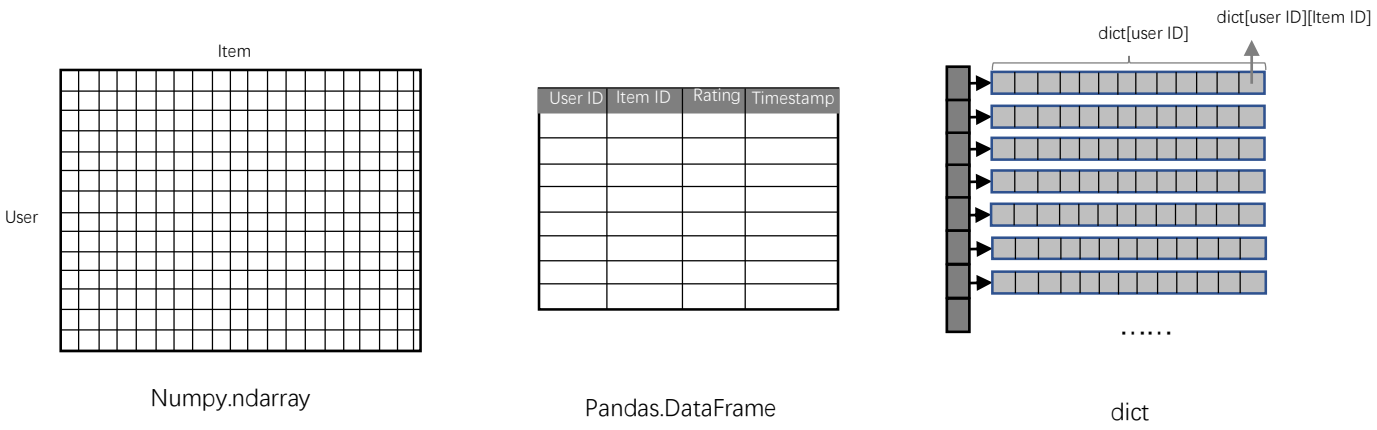
1. 给定用户 ID 和物品 ID，返回用户对物品的评分。
2. 给定用户的 ID，返回用户给每一个物品的评分。
3. 给定物品的 ID，返回所有用户给这个物品的评分。

这三种情况分别对应于，访问矩阵中的某个元素、访问行向量、访问列向量。

4.2 数据集的存储

实际上，数据的存储和访问是分不开的，什么样的访问方式决定了怎样的存储，针对上一小节的三种访问方式，最简单并且最高效的存储方式是将评分矩阵按照二维矩阵存储，访问时可以在常量的时间内得到数据的地址，但是这种存储方式是不现实的，在第一节已经看到数据集的稀疏度几乎都在 90%以上，有的能够达到 99%，这么大的二维表在内存中是存不下的，这种方式只适用于存储规模较小的数据集（如 MovieLens-100K、MovieLens-1M），虽说这种存储方式不推荐，但是我在实验中大量采用了这种存储方式，使用这种存储方式可以快速的得到模型的结果，适合于在调节模型的参数的时候使用。

另外一种存储方式是使用稀疏矩阵的存储方式，只存储矩阵中有值的元素。DataFrame 为我们提供了相关的方法，我在实验时对 DataFrame 的了解不够，数据访问时效率非常低，我又重新写了一种利用 python 中的字典数据结构存储的数据矩阵，以上三种存储方式如图所示，



4.3 数据集存储空间与访问效率

之前提到的三种存储方式在 MovieLens 数据集上的内存占用空间大小和数据集加载时间的对比如表 5，横向对比，ndarray 的占用空间最大，DataFrame 占用空间出奇的小，我是使用 DataFrame.memory_usage()方法计算空间大小的；DataFrame 只需要从磁盘中读入二维表，所以加载时间少，ndarray 加载数据集时需要逐个往数组内填入数据，dict 在加载数据集时要创建键值对应关系，加载较慢；纵向对比，加载时间和占用内存空间近乎与数据集大小成线性关系。

表 5：不同存储方式在 MovieLens 数据上所占用空间大小和数据集加载时间

Dataset	Numpy.ndarray		Pandas.DataFrame		dict	
	Size (MB)	Load time (s)	Size (MB)	Load time (s)	Size (MB)	Load time (s)
100K	12	3.597	1	0.241	5	4.222
1M	182	30.314	9	7.277	83	32.673
10M	-	-	95	82.119	829	329.714

20M	-	-	191	84.536	1572	613.724
25M	-	-	286	105.986	1936	783.325

表6给出了这些存储方式的访问效率，表中AET表示访问单个元素所耗费的时间（Access Element Time），ART是访问一行评分所消耗的时间（Access Row Time），ACT是访问一列元素所耗费的时间（Access Column Time），表中数据仅供参考，数据的准确性受机器状态的影响较大，存在波动，但是数据级别准确。ndarray不受数据集规模的限制，访问元素均为常量时间，对DataFrame中行列（即User ID和Item ID）分别建立了索引，它的访问效率大体上是和数据集的规模成$O(\log n)$的关系，索引可能是树结构，dict是散列形式存储键值的，dictAET为常量时间，但是ART和ACT却是$\mathrm{O(n)}$的复杂度。注意到表1中用户个数要比物品个数多，所以ACT访问比ART满。综合发现，这三种存储方式均有优劣，但没有一个能够完整满足我们的要求，事实上，考虑到ID是连续且不重复的整数，在DataFrame的基础上修改，将树结构索引改成散列表索引，可以是访问效率将达到$\mathrm{O(1)}$，进一步的工作没有继续做下去。

表 6：不同存储方式在 MovieLens 数据集上的访问效率（单位 ms）

Dataset	Numpy.ndarray			Pandas.DataFrame			dict		
	AET	**ART**	**ACT**	**AET**	**ART**	**ACT**	**AET**	**ART**	**ACT**
100K	0.0007	0.0009	0.0009	0.0783	0.4937	0.4966	0.0003	0.0019	0.0767
1M	0.0006	0.0019	0.0010	0.0655	0.4697	2.1143	0.0004	0.0079	0.4936
10M				0.2119	0.9165	2.0016	0.0004	0.0488	8.8872
20M				0.3530	0.9853	3.4082	0.0057	0.6951	20.726
25M				0.4681	0.9863	4.2287	0.0006	2.3504	-

4.4 存储方式对实际推荐模型的影响

使用不同的数据存储类型构造User-based模型，表7中Loading Time是模型加载时间，即计算用户之间相似度的时间，在计算用户间相似度时需要频繁按行访问，Recommend Time是构建完成模型之后，为用户推荐物品列表所消耗的时间，这段时间的差别也主要体现在按行访问数据，注意和表6中第一行的数据对比，数据访问效率基本可以体现模型在实际工作中的效率。

表 7：User-Based 模型在数据集 MovieLens-100K 上的时间效率

	Numpy.ndarray	Pandas.DataFrame	dict
Loading Time（s）	16.30	954.96	57.88
Recommend Time（s）	0.082	4.162	0.182