Collaborative filtering algorithms comparsion

这次项目完成了四种算法的实现和evulation，分别是memory-based的user-based, item-based，新方法tendency-based以及model-based中的slope方法。具体使用的数据集是movielens的100K的数据集（100,000 ratings from 1000 users on 1700 movies），具体的evulation方法是首先在原始数据集中提取出来10%－90%的tranining数据集以及10%的evulation数据集，然后将这个tranining数据集作为输入，分别在这个数据集上运行不同算法，然后将evulation的数据用来测试。

Mapreduce的常见实现思路：

输入文件 each line string->token for each line->key value collector->reduce->key array<collector>

下面分别来介绍下四种算法的核心思想和实现；

item-based

输入数据

userID movieID rating date

首先第一步先调用collect来收集用户数据，运用mapreduce的key,value属性，将每一个用户所打的分数的信息收集起来，

userID movieID rating date

map输入：each line string from the original file

map输出：userID, movierating类的一系列collector (mid, rating)

reduce输入：userID, movierating类的一系列collector

reduce输出：userID, array<movierating>(通过迭代器收集map的输出中的一系列movierating，放入当前key(userID)对应的list value中) (按照mid进行排序的结果)

－》map->collector: key userID value (mID rating) ->reduce-> userID (mID rating) list

after collect:

uid {(m1,r1), (m2,r2), (m3,r3)…}

第二步是预处理，运用collect的输出结果，再次运用mapreduce

map输入：each user and their ratings

map输出：movieID, postingUser类的一系列collector(uid, uavg, ratingforthis movie)

reduce输入：movieID, postingUser类的一系列collector

reduce输出: movieID, array<postingUser>(按照uid进行排序)

after preprocess:

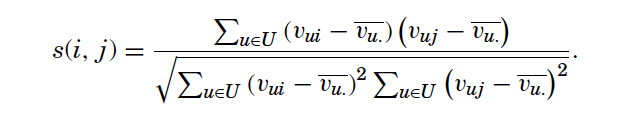
m1 {(u1, u1avg, ur1), (u2, u2avg, u2)…}

m2 {(u2, u2avg, ur2), (u3, u3avg, u3)…}

KNN:

首先我们吧所有preprocess所得到的结果存入一个hashmap中，然运用一个二重循环，计算这个hashmap中所有movie两两的weight，要计算这个weight，我们需要四个参数：midi, midj, milist, mjlist

由于这里每个movie所拥有的list已经根据用户id进行排序了，所以在两个movie的list中寻找userid相同的项是很容易的，找出两个movie的list中userid相同的那些项目（这个user同时对moviei和moviej进行了评分），运用公式



就能计算出任意两个movie的相似度（只要他们有公共的用户，即有用户同时对他们进行评分），最后按照相似度进行排序，并且将key(moviei) value(movie 的所有邻居moviej, wij)作为map的collector的输出，作为整个preprocess的输出。

Query：

最关键的一个步骤，我们根据原始数据集的10%的比例生成了evulation dataset，将evulation set中的userid和movieid作为两个输入，要预测这个userid对这个movieid的分数。首先现在knn的结果中找到待预测的movieid的所有邻居节点，将结果放入本地的cache 中

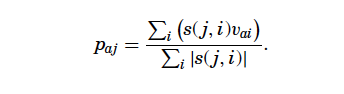
localHash: 待rate的movie的所有邻居的信息。

然后我们在给定的training set中寻找等待预测的那个用户的id，将带预测的用户所有打过分的movie信息存入localUserRating中

localUserRating:待打分用户所有已经打过分的电影信息。（如果发现已经包含等待打分的电影id，则不需要预测了）

我们在预测的时候，如果发现localHash为空，说明等待打分的电影没有邻居（不存在任何一个用户同时对这部电影以及其他任何电影同时打分），如果localUserRating为空，说明等待打分的用户没有对任何一部电影打过分。这两类情况都是非法的，它可能发生，它也是KNN item-based算法的不足之处，这种情况下，我们没有办法使用item-based对其做出预测，这是training set信息不足所造成的。然而这两者全部不为null的时候，如果等待测试的movie的所有邻居中没有一个是等待测试的user已经rate过的（这是有可能的，我们拥有一些数据，待测movie拥有neighbour，并且待测user也打过一系列的电影，只是这些rated过的电影没有一个是movie的neighbour，可以看出还是一个sparse所造成的问题，user打分还不够多，这些事knn的常见问题。）

如果一切正常，就是当localHash和localUserRating都不是空的时候，我们能够对其进行预测。对于localHash中的邻居节点（相似度从高到低，选10个），如果user对这个邻居节点评过分，用这个评分以及和待测电影的相似度，就能够计算得到最终的预测分数。



User-based

和item－based一样，user-based的第一步也是通过collect求得training set（所有已知用户的评分）中用户的评分。然后用preprocess，其输出是

user array<mid, uAVG, rating>，根据mid进行排序。

KNN

和item-based类似，首先我们要将preprocess的结果全部存入本地hash中，然后计算hash中任意两个key的相似度（不同于item-based，这里的key是userid），要计算任意两个user的相似度，我们同样需要四个变量，useriID, userjID, useri的value list, userj的value list，在具体的计算过程中，同样的，由于mid是sorted的，所以我们能够很容易地定位到value中id相同的两个movie（ui和uj同时对某一个movie进行打分，这里可以看到，在item-based中，我们寻找的是同时对mi和mi打分的那个user，而这里我们寻找的是ui和uj同时打过分mid）,找到它们，从而计算出ui和uj的相似度，并且和item-based，类似的，生成ui的邻居串。

下面来重点说一下query的过程。首先，我们先要从knn的结果中将需要预测的那个user的所有邻居（uid, weight）存入本地hash中。同时，在原始数据中寻找，同样先判断是否有必要预测（给定user尚未对给定movie进行评分）。然后，如果发现给定user没有对给定movie进行评分，则需要进行核心的预测操作。

首先判断localhash是否为空，对于给定的user，如果它的neighbour为空，说明任何已知的其他用户和它没有共同的打分电影（可能是其他用户打分的数量不足够，也或者是用户数量不够），这也是KNN的信息不足（sparse density所造成的问题）。如果localhash存在，说明该用户拥有neighbour，则我们可以选取这些users中和待测user相似度较高的那些，获取它们的评分。

然后判断localUserRating是否为空，如果这都为空，我们无法搜索neighbour中任何的userid是否对给定movie所打的分数，这种情况也是不合法的。

一旦发现这两个集合都没问题（数据量相对可以，user拥有自己的neighbour，并且本地user集合也存在，能够所寻neighbour）,这时候我们就需要在本地user集合中所寻neigubhour，找到neighbour中对movie打过分的前10个rating，因为这些user和原始user的similarity相对比较高，能够运用这些rating的平均值估算出user对movie的 rating。和item-based相似的，这里我们还需要注意，如果遍历完发现没找到，说明所有的neighbour都没有对给定的movie打过分，这是可能的（某些user打得分数还不够多，这也是sparse所造成的问题）

memory-based 总结

我们可以看出，无论是item-based还是user-based，都是依赖相似度所进行地运算，在item-based中，我们寻找的是和待测movie相似度高的那些movie，找到待测user对它们打的分数，运用这些分数来估算待测movie的评分。而在user-based中，我们找的是和待测user相似度高的那些user，通过这些user对待测movie的打分直接估算待测user对待测movie的打分。

Tendency

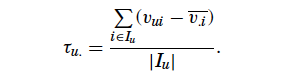
和常见的knn memory-based寻找相似度不同的是，这种计算tendency的方式是基于difference所进行的运算。

Tendency的预处理需要同时运用item-based和user-based的预处理过程，也是创建movieid array<uid, avgrating, rating>的串以及userid array<mid, avgrating, rating>的串。

然后是tendency的query过程，和item-based和user-based类似，这里首先需要将远程HDFS的预处理结果文件存入本地cache中，item-based的预处理结果存入movieinfo中，而user-based的预处理结果存入userinfo中，完成后就能够将开辟的用来读取HDFS的变量释放掉。

User-tendency:

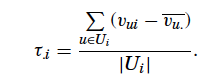
然后所要做的是判断待预测user是否存在与userinfo中（这里判断本地文件是否为空可以省略，因为这里assume输入的文件是有内容的，当然，加入也是不可以。）如果存在，从userinfo中将这个user所打分信息提取出来，我们需要的是这个user对i的rating以及这个i的平均被rating值。要获取这个被rating值，我们需要从movieinfo中找出这个movie的被rating信息，从而计算这个平均值。最后，运用这个公式



计算user-tendency。

Item-tendency

和user-tendency类似，要计算这个，我们需要首先从movieinfo中提取出这个movie的被rating信息，如果这个不存在，则我们无法计算tendency。在我们提取出这个movie的信息后，所要做的是获取这个movie被特定useri 所打的rating以及那个user的平均打分值。要获取后者，类似的，我们同样需要获取userinfo中信息，然后利用



计算出结果。

可以看出，这里利用training set计算待测user和movie的rating的时候，我们只是依赖数据集中是否有user和movie，而不需要关注是否有公共的关系，比如user-based中neighbour是否也对movie进行过打分，这样对数据集的spase要求就比传统的knn要低得多，能够评估的范围也大了很多。

在计算完user tendency和item tendency之后，我们好需要做的一件事是获取待测user的平均rating和待测movie的平均被rating，这很容易，这里就不赘述了。

最后根据tendency分析结果，分别为两者tendency都为positive，两者都为negative以及两者一个positive一个negative进行分类讨论，得出最后的结果。

Slopeone: