# 《大数据计算及应用》

# 【推荐系统】实验报告

小组成员：潘巧巧 1613415、李佳骏1613368

## 【实验相关统计信息】

（1）统计用户数量：19835。

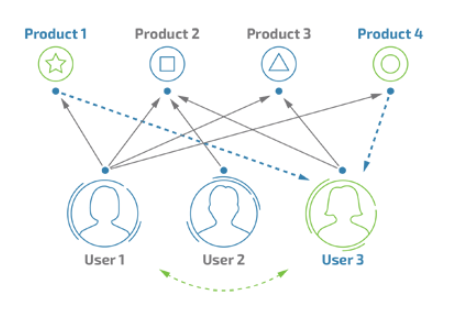
（2）统计所有产品的得分平均值：49.545。

（3）统计每个用户看过的电影数目，在读取数据时将其存入数组int MovieNum[19835];中。  
（4）统计计算Pearson系数需要的数据，存在Pearson矩阵中。

## 【实验原理】

核心算法：**基于用户的协同过滤**

如果已知用户A喜欢《蜘蛛侠》、《奇异博士》、《美国队长》、《绿巨人》等漫威超级英雄系列电影，另外用户B也都喜欢这些电影，此外B还喜欢《钢铁侠》，则A很有可能也喜欢《钢铁侠》，故我们可以预测用户A对于《钢铁侠》的打分会较高。

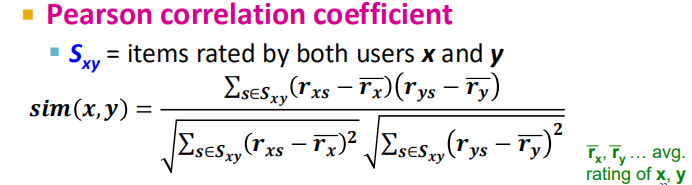


在本次实验中，有两个关键步骤：

1. **找到与目标用户A相似的用户群体**

**皮尔逊相关系数**（Pearson Correlation Coefficient）是余弦相似度在维度值缺失情况下的一种改进，故我们利用了皮尔逊相关系数来衡量两个用户是否相似，由于矩阵较为系数，故仅需当用户B与我们的目标用户之间的Pearson相似度大于0时，用户B对目标电影的打分即成为我们对A打分进行预测时参考的重要指标。

用户x和用户y之间的Pearson相似度的计算公式为：



附：皮尔逊相关系数的分类：

0.8-1.0 极强相关

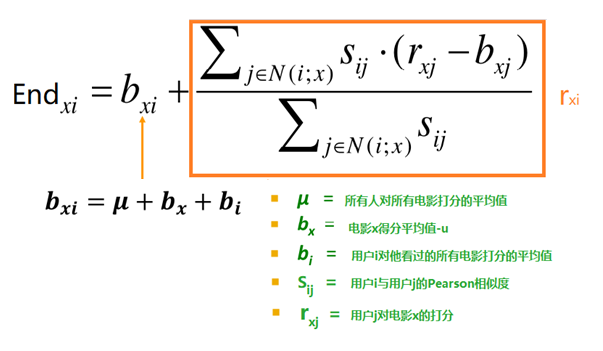
0.6-0.8 强相关

0.4-0.6 中等程度相关

0.2-0.4 弱相关

0.0-0.2 极弱相关

1. **根据相似用户群体对目标电影的打分，对目标用户A给目标电影的打分进行预测。**



bxi为最终结果的基线预测部分。bxi 中bi 可以根据目标用户的个人打分习惯对最终结果进行调整（用户习惯打高分/低分），bx部分可以根据目标电影（产品）与所有产品的质量差距对最终结果进行调整（该电影质量高于/低于所有电影的质量的平均值）。

rxi为调整部分。根据与目标用户的相似用户群体对目标电影的打分以及目标用户与各个相似用户的Pearson相似度对最终预测结果进行调整。

具体实现过程：

1. 前期准备：全局变量
   1. 建立一个struct MovieScore，包括int类型的IDOfMovie（电影编号）和int类型的ScoreOfMovie。
   2. 动态创建MovieScore类型的二维数组Data，行数为用户数量（19835），用户编号与行号一一对应；列数为该行对应的用户看过的电影的数量，每行都不相同。
   3. int MovieNum[19835]，用来存放每个用户观看的电影数量。
   4. 创建一个类型、行数、列数都与Data相同的二维数组PearData。
2. Load()函数中使用getline逐行读取train.txt的数据。
   1. 若该行中存在“|”，说明是一个新的用户，读取他看过的电影数目，存入MovieNum。并动态创建Data中该用户对应行的大小。
   2. 若该行中不存在“|”，则是电影信息。将电影编号和电影分数转成int类型，按读取顺序存入Data中。
3. 调用SortMovieID()函数，对每一用户看过的电影按电影编号从小到大的顺序进行排序，便于后续利用二分查找进行查找工作。
4. 在Pearson()中，利用数组MovieNum对每一行进行动态初始化。用Data的数据对PearData的数据进行相关计算填充。其中电影ID全部相同，PearData[i][j].MovieScore为Data[i][j].MovieScore与“该用户对他看过的所有电影打分的平均值”之差。
5. 计算最终结果中的rxi部分。

遍历所有用户，找到同样看过目标电影的用户，如果该用户与目标用户的Pearson相似度大于0，则进行上图公式中rxi部分的计算。若不存在“看过目标电影且与目标用户相似”的用户，则rxi值返回1000，即一个错误结果。

1. 计算最终结果中的bxi部分。

根据上图公式调用double AvgUser(int UserID)、double AvgItem(int ItemID)等函数进行计算。

1. 计算目标用户对目标电影的预测打分最终结果End。

如果rxi返回值为1000，则以目标用户对他看过的所有电影打分的平均值作为最终结果。如果rxi的返回值有效，则以bxi+rxi作为最终结果。

1. 将预测结果写入text.txt中
2. 读取test.txt中需要测试的数据
3. 将测试数据与测试结果一起按格式写入指定记事本中

## 【关键部分代码解析】

1. 前期准备

struct MovieScore

{

int IDOfMovie;

double ScoreOfMovie;

};

MovieScore \*\*Data = new MovieScore \*[NumOfUser];

MovieScore \*\*PearData = new MovieScore \*[NumOfUser];

int \*MovieNum = new int [NumOfUser];

1. 读取数据

//读取数据到Data矩阵

void load()

{

//打开train.txt

ifstream infile1;

infile1.open("train.txt");

assert(infile1.is\_open()); //若失败,则输出错误消息,并终止程序运行

//向data效用矩阵内存数据

string s; //存放读取行数据

string blank = " ";

string line = "|";

int UserID = -1; //已知数据集中user递增，这里可以直接先赋值为-1

int NumOfMovie = 0; //目前的用户所看过的电影数目

int clock = 0;

while (getline(infile1, s))//逐行读取数据

{

int LocLine = s.find(line); //查找该行中"|"所在位置

if (s == "")

break;

if (LocLine != -1) //"|"存在，新的用户，读取他看过的电影数

{

UserID++;

clock = 0;

int len = s.length();

string ss = s.substr(LocLine + 1, len - LocLine);

NumOfMovie = stoi(ss); //该用户看过的电影数目

MovieNum[UserID] = NumOfMovie; //存入数组中

//动态初始化data数组中该UserID行

Data[UserID] = new MovieScore[NumOfMovie];

}

else //"|"不存在，还是上一个用户,现在s存的是电影号和评分

{

int LocBlack = s.find(blank); //空格所在位置

string IDOfMovie = s.substr(0, LocBlack);

int leng = s.length();

string sco = s.substr(LocBlack + 2, leng - LocBlack);

double score = stof(sco);//当前用户对目前行的电影的评分

int MovieID = stoi(IDOfMovie);//当前电影编号

struct MovieScore MS = { MovieID,score };

//将该行数据存入效用矩阵对应位置

Data[UserID][clock] = MS;

clock++;

}

}

infile1.close();

cout << "train.txt已读取完毕" << endl;

}

1. 将每个用户看过的电影按编号从小到大升序排列，便于后续查找操作中使用二分查找

bool cmp2(MovieScore a, MovieScore b)

{

return a.IDOfMovie < b.IDOfMovie;//按照学号升序排列

}

void SortMovieID()

{

for (int i = 0; i < NumOfUser; i++)

{

sort(Data[i], Data[i] + MovieNum[i], cmp2);

}

}

1. 将Data矩阵的数据转成Pearson系数矩阵

void Pearson()

{

for (int i = 0; i < NumOfUser; i++)//外层遍历所有用户

{

double avguser = AvgUser(i);

PearData[i] = new MovieScore[MovieNum[i]];

for (int j = 0; j < MovieNum[i]; j++) //内层遍历当前用户看过的所有电影

{

PearData[i][j].IDOfMovie = Data[i][j].IDOfMovie;

PearData[i][j].ScoreOfMovie = Data[i][j].ScoreOfMovie - avguser;

}

}

}

1. 求用户A和用户B的Pearson相似度

double PearSimAB(int UserA, int UserB)

{

//求Pearson系数的分子

//求出AB都看过的电影a的编号

//计算：用户A对电影a的评分\*用户B对电影a的评分，存在numerator

double numerator[10000];

int clock = 0;

//如果A看过的电影少于B看过的电影

if (MovieNum[UserA] <= MovieNum[UserB])

{

for (int i = 0; i < MovieNum[UserA]; i++) //遍历A用户看过的电影

{

//利用二分查找，查看当前B是否看过当前A遍历到的电影,此前已将各个用户看过的电影按照电影编号从小到大升序排列

int j = BinSearch(PearData, UserB, MovieNum[UserB], PearData[UserA][i].IDOfMovie);

if (j != -1)

{

numerator[clock] = PearData[UserA][i].ScoreOfMovie \* PearData[UserB][j].ScoreOfMovie;

clock++;

}

}

}

//如果A看过的电影多于B看过的电影

if (MovieNum[UserA] > MovieNum[UserB])

{

for (int i = 0; i < MovieNum[UserB]; i++) //遍历B用户看过的电影

{

int j = BinSearch(PearData, UserA, MovieNum[UserA], PearData[UserB][i].IDOfMovie);

if (j != -1)

{

numerator[clock] = PearData[UserB][i].ScoreOfMovie \* PearData[UserA][j].ScoreOfMovie;

clock++;

}

}

}

}

1. 最后产品得分中的bxi的值

double bxi(int UserID, int ItemID, double u)

{

double bx = AvgUser(UserID) - u;

double bi = AvgItem(ItemID) - u;

double bxi = u + bx + bi;

return bxi;

}

其中调用了函数AvgUser()和AvgItem()，传参进来的u是由AvgAll()求得的总体平均值

//求总体平均值

double AvgAll()

{

double Average = 0;

double Sum = 0;

int count = 0;

for (int i = 0; i < NumOfUser; i++)

{

for (int j = 0; j < MovieNum[i]; j++)

{

Sum = Sum + Data[i][j].ScoreOfMovie;

count++;

}

}

Average = Sum / count;

return Average;

}

//求目标用户对他看过的所有电影打分的平均值

double AvgUser(int UserID)

{

double Average = 0;

double Sum = 0;

for (int i = 0; i < MovieNum[UserID]; i++)

{

Sum = Sum + Data[UserID][i].ScoreOfMovie;

}

Average = Sum / MovieNum[UserID];

return Average;

}

//求目标产品的所有评分的平均值

double AvgItem(int ItemID)

{

double Average = 0;

double Sum = 0;

int count = 0;

for (int i = 0; i < NumOfUser; i++) //外层遍历所有用户

{

int j = BinSearch(Data, i, MovieNum[i], ItemID);

if (j != -1)

{

Sum = Sum + Data[i][j].ScoreOfMovie;

count++;

}

}

Average = Sum / count;

return Average;

}

1. 最后产品得分中的rxi的值

double rxi(int UserID, int ItemID, double u)

{

//①求所有"PearSimAB与目标用户作用大于0、且看过目标电影"的其他用户与目标用户的相似度之和

//②求(看过目标电影的其他用户与目标用户的相似度)\*(看过目标电影的其他用户对目标电影的打分)之和

double SumSij = 0;

double Sum\_Sij\_rjx = 0;

for (int i = 0; i < NumOfUser; i++)//遍历用户

{

int j = BinSearch(Data, i, MovieNum[i], ItemID);

if (j != -1)//如果该用户也看过目标电影

{

double a = PearSimAB(UserID, i);

if (a > 0) //如果两用户是相似用户

{

SumSij += a;

Sum\_Sij\_rjx += a \* (Data[i][j].ScoreOfMovie - bxi(i, Data[i][j].IDOfMovie, u));

}

}

}

//如果计算有效，即存在看过目标电影且与目标用户相似的用户，返回调控值rix

if (SumSij != 0 && Sum\_Sij\_rjx != 0)

{

double rix = Sum\_Sij\_rjx / SumSij;

return rix;

}

//如果计算无效，即不存在看过目标电影且与目标用户相似的用户，返回一个无效值1000

else

{

double rix = 1000;

return rix;

}

}

1. 预测目标用户对目标电影的终极打分

double End(int UserID, int ItemID, double u)

{

double End = 0;

double rxi1 = rxi(UserID, ItemID, u);

if (rxi1 == 1000) //不存在看过目标电影且与目标用户相似的用户

{

//以目标用户打分平均值作为结果

End = AvgUser(UserID);

return End;

}

else

{

End = rxi1 + bxi(UserID, ItemID, u);

//对最终结果的合理性进行适当修改

if (End > 100)

End = 100;

if (End < 0)

End = 0;

return End;

}

}

1. 从test.txt读取要测试的数据，并将测试结果写入text.txt中

void ReadtoBuffer()

{

ifstream infile;

infile.open("test.txt");

assert(infile.is\_open());

string s; //存放读取行数据

string blank = " ";

string line = "|";

int UserID = -1; //user递增，可以直接先赋值为-1

int Row = -1;

int i = 0;

while (getline(infile, s))

{

int LocLine = s.find(line); //查找该行中"|"所在位置

if (s == "")

break;

if (LocLine != -1)

{

Row++;

i = 0;

}

else

{

ReadBuffer[Row][i] = stoi(s);

i++;

}

}

infile.close();

}

void WriteText(double u)

{

fstream file1;

file1.open("Text.txt");

for (int i = 0; i < 19835; i++)

{

for (int j = 0; j < 6; j++)

{

if (!j)

{

file1 << i << "|" << "6" << endl;

file1 << ReadBuffer[i][j] << " " << int(End(i, ReadBuffer[i][j], u)) << endl;

}

else

file1 << ReadBuffer[i][j] << " " << int(End(i, ReadBuffer[i][j], u)) << endl;

}

}

file1.close();

}

1. Main()函数中一些必须执行的步骤

int main()

{

HANDLE hThread;

DWORD ThreadID;

hThread = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)ReadtoBuffer, NULL, 0, &ThreadID);

//创建一个线程，在load()运行的同时同时把test.txt的数据读入

load();

double u = AvgAll();

SortMovieID();

Pearson();

WriteText(u);

return 0;

}

## 【结果分析】

1. 本实验最终测试RMSE结果约为23。
2. 我们选用了基于用户的算法，而User-based算法存在两个问题：
3. 数据稀疏性。本实验测试数据train中不同用户之间观看的电影重叠性较低，导致算法时常无法找到看过目标电影且与目标用户相似的用户，使得调控常数rxi 失效。
4. 算法扩展性。User-based算法的计算量随着用户和物品数量的增加而增加，不适合数据量大的情况使用。

我们仍然选用User-based算法的原因：由train数据集本身特征决定，train数据集中存在“用户ID顺序排列，电影ID随机”的特征，使用基于用户的算法方便处理。

1. 由train数据集本身特性带来的结果偏差。train数据集本身存在一些不合理之处，如0分过多等，给测试结果带来干扰。