Project 3: Recommendation Algorithm for Animes

一、背景介绍

随着当今技术的飞速发展，数据量也与日俱增，为了解决**信息过载**的问题，人们提出了推荐系统，区别于搜索引擎，推荐系统更倾向于人们没有明确的目的，推荐系统根据历史行为产生用户可能感兴趣的项目列表。其中长尾理论可以很好地解释推荐系统的存在，推荐系统可以给所有项目提供曝光的机会，不仅仅是曝光率高的项目，以此来挖掘长尾项目的潜在利润。

二、技术原理

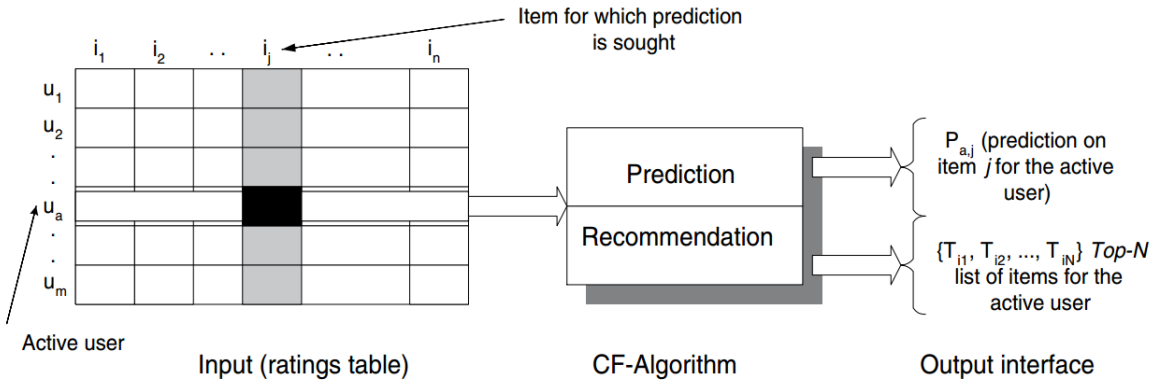
2.1 推荐系统

推荐系统的任务就是解决，当用户无法准确描述自己的需求时，如何去向用户推荐他感兴趣的产品。联系用户和信息，一方面帮助用户发现对自己有价值的信息，另一方面让信息能够展现在对他感兴趣的人群中，从而实现信息提供商与用户的双赢。

2.2 推荐系统算法 -- 协同过滤算法

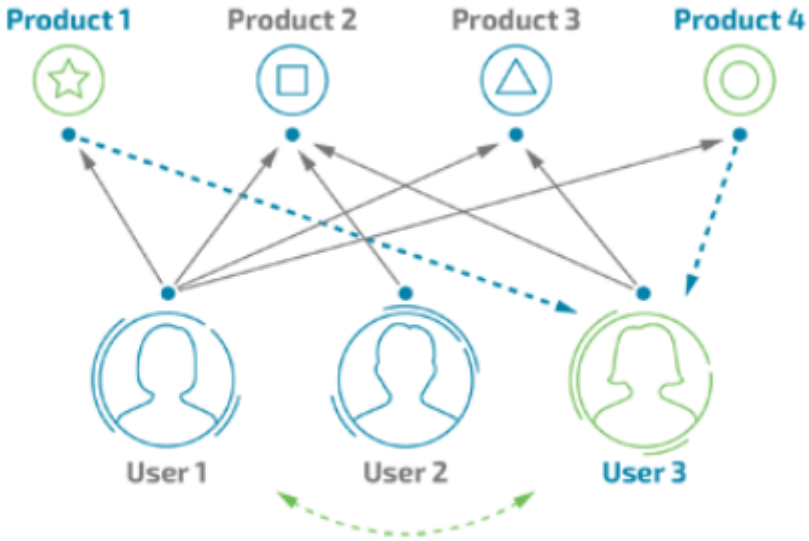
协同过滤推荐算法是诞生最早，并且较为著名的推荐算法。主要的功能是预测和推荐。算法通过对用户历史行为数据的挖掘发现用户的偏好，基于不同的偏好对用户进行群组划分并推荐品味相似的商品。协同过滤推荐算法分为两类，分别是基于用户的协同过滤算法(user-based collaboratIve filtering)，和基于物品的协同过滤算法(item-based collaborative filtering)。

使用协同过滤算法做推荐的流程如下：

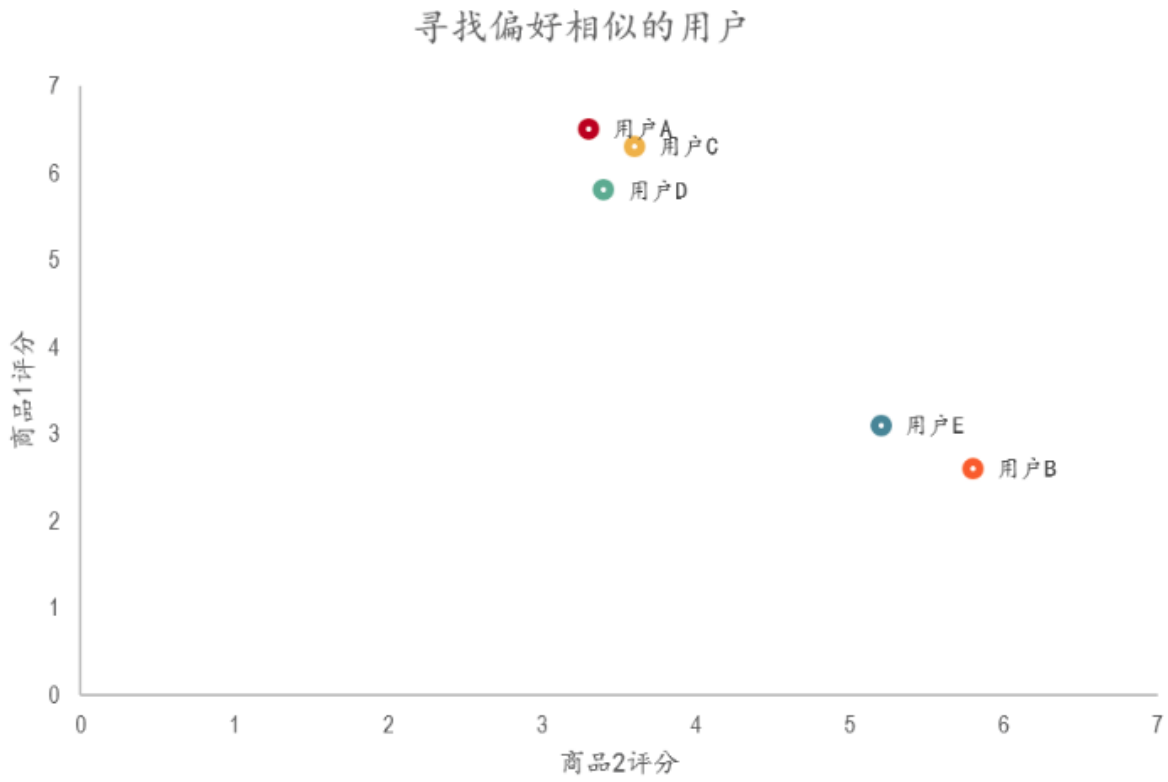


2.2.1 基于用户的协同过滤算法

基于用户的协同过滤算法是通过用户的历史行为数据发现用户对商品或内容的喜欢程度。根据不同用户对相同商品或内容的态度和偏好程度计算用户之间的关系。在有相同喜好的用户间进行商品推荐。简单来讲，它会把跟你相似的人喜欢的东西推荐给你。



一般来讲会使用用户给商品的评分来作为计算用户间相似度的依据：

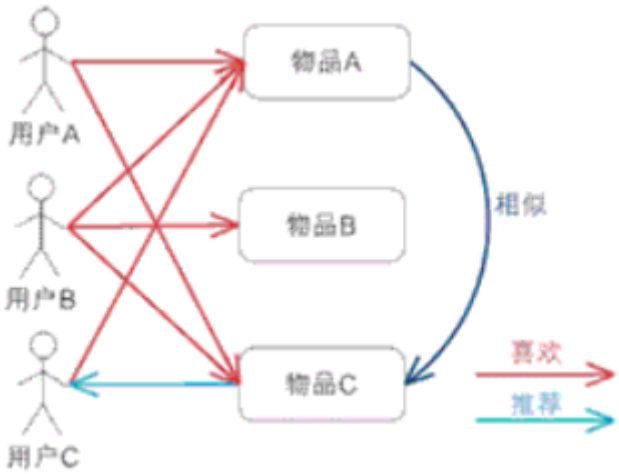


在上图的例子中，我们用用户对商品的评分来表征用户。然后用户可以抽象成高维空间的向量，可以用欧几里德距离，cosine相似度来评价用户之间的相似度。

$$\mathrm{sim}(i,j) = \cos(\vec{i},\vec{j}) = \frac{\vec{i}\cdot\vec{j}}{\|i\|_2 * \|\vec{j}\|_2}$$

2.2.2 基于物品的协同过滤算法

Iterm-based的基本思想是预先根据所有用户的历史偏好数据计算物品之间的相似性，然后把与用户喜欢的物品相类似的物品推荐给用户。举个简单的例子，已经知道物品A和C非常相似，因为喜欢A的用户同时也喜欢C，而用户C喜欢物品A，就可以把物品C推荐给用户A。



2.2.3 基于SVD分解的协同过滤算法

不同于之前的两种算法，基于SVD分解的协同过滤算法是一种基于模型（model-base）的算法。在这个算法中，会用SVD进行降维，得到模型的隐变量，更好的来表征用户和物品。一般在用户和物品的矩阵很稀疏的情况下，效果会比之前两种要好。

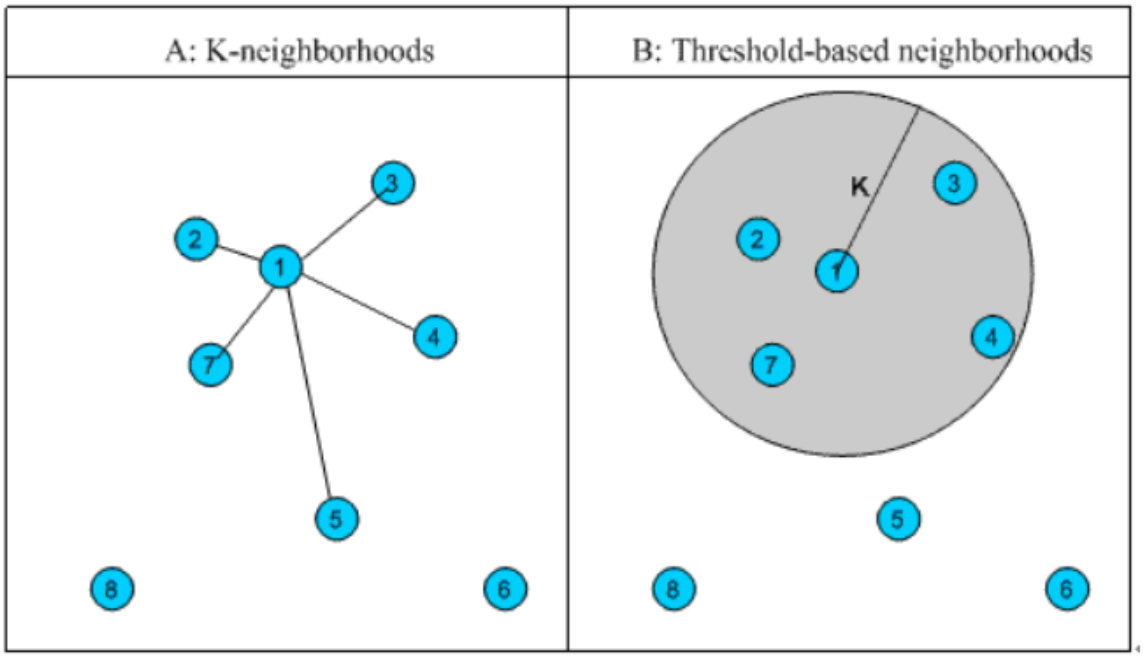
$$A = U\Sigma V^T$$

矩阵A是用户和物品的打分矩阵，分解成用户矩阵U和商品矩阵V。

$$\hat{r}_{ui} = \bar{r}_u + U(user_i) \times \Sigma \times V^T(item_j)$$

每次需要预测的时候，就进行上面的计算得到某用户给某一商品打分的预测值。

为了增加鲁棒性，也可以使用在用户空间中跟他相近的人的评分的加权平均来预测；也可以使用商品空间的类似商品的打分的加权平均值。一般选取neighborhood有以下两种：



基于KNN的公式如下：

$$\hat{r}_{ui} = \frac{\sum_{v \in N_i^k(u)} \mathrm{sim}(u,v) \cdot r_{vi}}{\sum_{v \in N_i^k(u)} \mathrm{sim}(u,v)}$$

在本次实验中，因为用户和物品的矩阵十分稀疏，大于99%以上的地方都是空值。所以，我们选择使用基于SVD的CF算法。在矩阵很稀疏的情况下，SVD可以有更加好的效果。而且，基于SVD的CF更加节省内存空间，不像原本的CF那样需要算一个很大的相似矩阵。

评分矩阵R存在这样一个分解:

$$R_{U\times I} = P_{U\times K}Q_{K\times I}$$

上图中的U表示用户数，I表示商品数。然后就是利用R中的已知评分训练P和Q使得P和Q相乘的结果最好地拟合已知的评分，那么未知的评分也就可以用P的某一行乘上Q的某一列得到了：

$$\hat{r}_{ui} = p_u^T q_i$$

最小化总的误差平方和：

$$\text{SSE} = \sum_{u,i} e_{ui}^2 = \sum_{u,i} \left(r_{ui} - \sum_{k=1}^{K} p_{uk} q_{ki} \right)^2$$

在这里可以使用常见的梯度下降法SGD。之后还可以加入正则项，防止过拟合。

三、问题分析

1.原始数据分析

本次项目的目标是做一个推荐系统，数据源是Animes，这是一个关于动画的数据。我们拥有两个数据文件。

- anime.csv

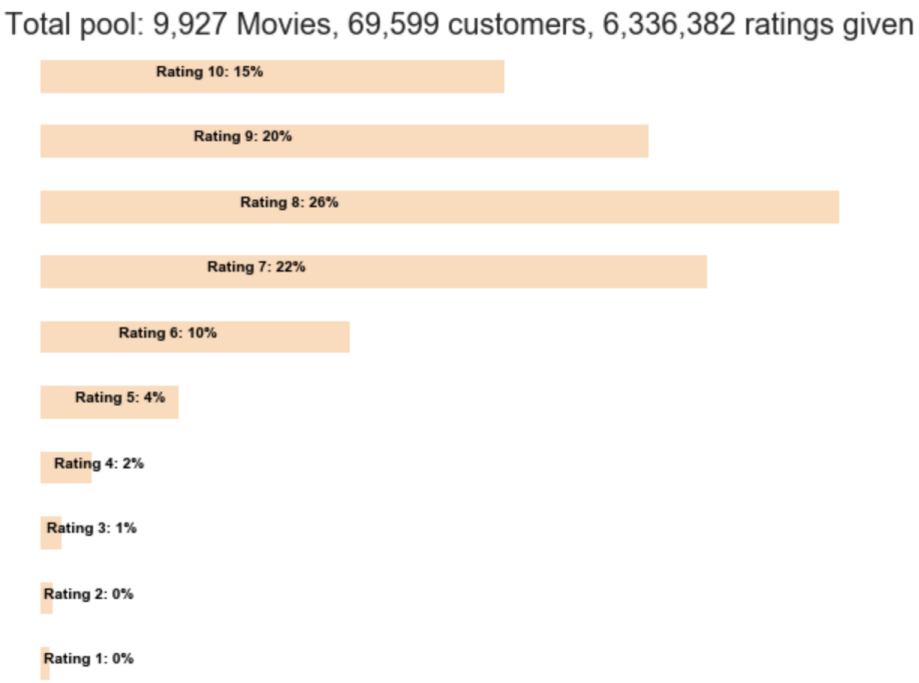
	anime_id	name	genre	type	episodes	rating	members
0	32281	Kimi no Na wa.	Drama, Romance, School, Supernatural	Movie	1	9.37	200630
1	5114	Fullmetal Alchemist: Brotherhood	Action, Adventure, Drama, Fantasy, Magic, Mili...	TV	64	9.26	793665
2	28977	Gintama°	Action, Comedy, Historical, Parody, Samurai, S...	TV	51	9.25	114262
3	9253	Steins;Gate	Sci-Fi, Thriller	TV	24	9.17	673572
4	9969	Gintama'	Action, Comedy, Historical, Parody, Samurai, S...	TV	51	9.16	151266

这份数据主要是每部动画的一些基本信息，如名字、类别、集数、评分、评分人数等。共有12294条数据。

- Rating.csv

	user_id	anime_id	rating
0	1	20	-1
1	1	24	-1
2	1	79	-1
3	1	226	-1
4	1	241	-1

这份数据是用户对每部动画的评价信息，其中评价分数从1-10，-1代表该信息缺失。共有7812689条数据。虽然这份数据看起来很多，但是后续使用协同过滤算法（CF）计算未知的评分时，由于总评价人数和动画数量的众多，共有9927个电影和69599个用户，打分数据的占比实际上是很少的，仅占0.77%。



因此单纯用CF算法，由于有效数据的占比很低，因此得到的效果可能不佳，我们考虑结合两个数据表的数据进行综合分析。

- Merge

简单地将上述两个数据表进行合并，将动画的id作为key合并得到下表，同时新加两列，判断rating是否大于8。

	anime_id	name	genre	type	episodes	rating_x	members	user_id	rating_y	rating_x_bool	rating_y_bool
803251	11771	Kuroko no Basket	Comedy, School, Shounen, Sports	TV	25	8.46	338315	2	10	True	True
7679839	19315	Pupa	Fantasy, Horror, Psychological	TV	12	3.82	83652	3	3	False	False
6356295	1122	Pokemon Advanced Generation: Rekkuu no Houmons...	Action, Adventure, Comedy, Fantasy, Kids, Sci-Fi	Movie	1	7.10	45203	3	7	False	False
3151815	154	Shaman King	Action, Adventure, Comedy, Drama, Shounen, Sup...	TV	64	7.83	169517	3	6	False	False
5381179	341	Spiral: Suiri no Kizuna	Drama, Mystery, Shounen	TV	25	7.38	36187	3	6	False	False
3896451	14345	Btooom!	Action, Psychological, Sci-Fi, Seinen	TV	12	7.68	329561	3	8	False	False
7188213	12671	Pokemon Best Wishes! Season 2: Kyurem vs. Seik...	Action, Adventure, Comedy, Drama, Fantasy, Kids	Movie	1	6.68	14498	3	3	False	False
4832035	6880	Deadman Wonderland	Action, Horror, Sci-Fi	TV	12	7.48	453454	3	6	False	False
117951	199	Sen to Chihiro no Kamikakushi	Adventure, Drama, Supernatural	Movie	1	8.93	466254	3	10	True	True
4001305	23321	Log Horizon 2nd Season	Action, Adventure, Fantasy, Game, Magic, Shounen	TV	25	7.66	215817	3	6	False	False

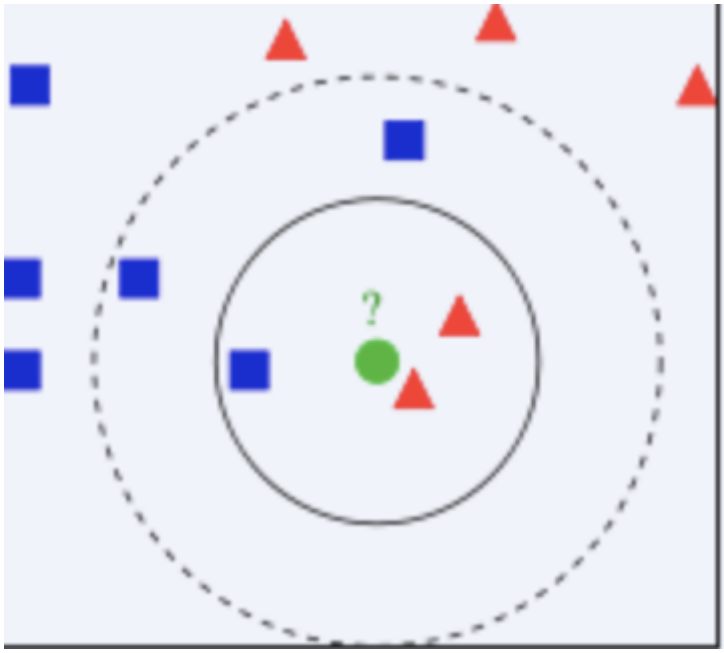
$rating_x$表示平均打分值，$rating_y$表示该用户打的分值，统计两个都为True或False的个数。最终占比大概在67.6%，这么一个朴素的想法，使用平均值去预测的准确率大概也有0.6。

再对合并的数据进行分析：**平均一部动画被打分的人数大概在638部；平均每个人打分的电影数为91次。**

2.KNN算法分析

K最邻近分类算法是数据挖掘分类技术中最简单的方法之一，意思是每个样本都可以用它的K个邻居来代表。KNN通过测量不同特征值之间的距离进行分类。

思路是如果一个样本在特征空间的K个最邻近的样本中的大多数属于某一个类别，则该样本也划分为这个类别。参考下图所示。



在推荐系统中，KNN主要用在相似邻居的计算，可以分为两类：1. 固定数量的邻居K个 2.基于相似度门槛的邻居，落在以当前点为中心，距离为K的区域中。

```
array([[    0,   208,  1494,  1959,    60,   894],
       [    1,   200,   268,   101,   795,   290],
       [    2,     4,     9,    12, 10896,     8],
       ...,
       [12291, 12238, 12237, 12236, 12256, 12235],
       [12292, 12231, 12232, 12230, 12229, 12283],
       [12293,  7426,  8279,  7349,  7335,  7498]])
```

上图展示了K=6时，每个动画的相似邻居编号

```
print_similar_animes(query="Naruto")
```

```
Naruto Shippuuden
Katekyo Hitman Reborn
Bleach
Dragon Ball Z
Boku no Hero Academia
```

上图展示了完全根据特征的相似性做的动画推荐，示例为Naruto的相似动画推荐。

四、Spark实践

在Spark的MLlib中，使用ALS（交替最小二乘法）进行协同过滤。

	v1	v2	v3	v4	v5	v6
u1						3
u2					3	
u3		5				
u4			7			
u5				6		
u6		6	?		5	
u7			6			
u8		1			5	
u9	3					

如上图所示，u表示用户，b表示需要打分的对象，由于并不是每个用户都给每种商品打了分，因此可以假设该ALS矩阵是低秩的，因此可将它化为两个低维矩阵的相乘。

$$A_{m*n} = U_{m*k} * V_{k*n}$$

为什么这种假设是合理的呢？因为用户和商品都包含了一些低维度的隐藏特征。根据这些隐藏特征就能推断出用户的喜好，而无需知道所有详细的信息。

$$\left\|A - UV^T\right\|_F^2 \rightarrow \min \sum_{(i,j)\in R} (a_{ij} - u_i^T v_j)^2$$

这就将协同过滤问题转化成了一个优化问题，由于目标函数中U和V相互耦合，因此需要使用交替最小二乘算法。

参数问题

- 隐藏特征：特征响量维度，越大拟合效果越好，但容易过拟合
- 迭代次数：越大越精确，但也越耗时，一般5-10即可
- 正则化参数：设置很大就可以防止过拟合，小的话训练集拟合效果较好，我们取了0.01

代码

我们依然使用sbt来帮助管理和编译文件，我们的Scala主体代码如下：

```
object CF{
  case class Rating(userId: Int, movieId: Int, rating: Float)
  def parseRating(str: String): Rating = {
    val fields = str.split(",")
    assert(fields.size == 3)
    Rating(fields(0).toInt, fields(1).toInt, fields(2).toFloat)
  }
  case class Testing(userId: Int, movieId: Int)
  def parseTesting(str: String): Testing = {
    val fields = str.split(",")
    assert(fields.size == 2)
    Testing(fields(0).toInt, fields(1).toInt)
  }
  def main(args:Array[String]){
    val n_iter = args(0).toInt
    val RegParam = args(1).toFloat
    val spark = SparkSession
        .builder
```

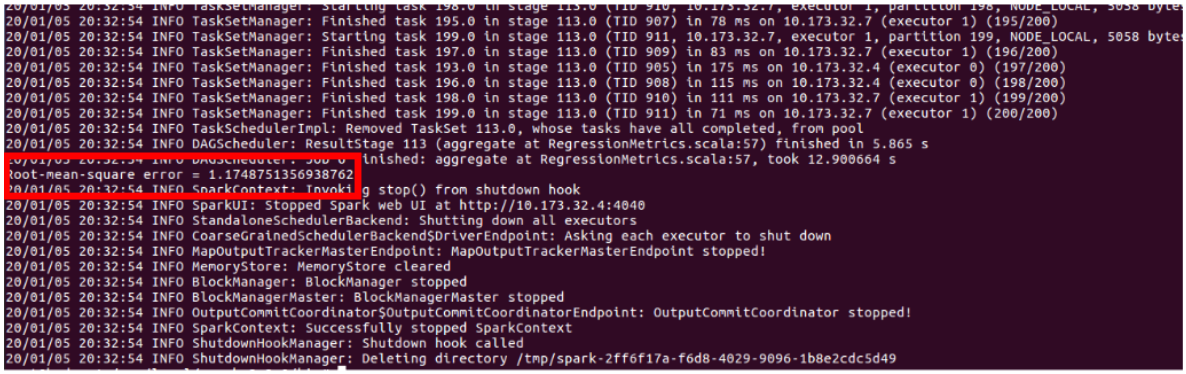
```
        .appName("CF")
        .getOrCreate()
    import spark.implicits._
    val ratings = spark.read.textFile("./Data.txt")
      .map(parseRating)
      .toDF("userId","movieId","rating")

    val testings = spark.read.textFile("./test.txt")
      .map(parseTesting)
      .toDF()
    val Array(training, test) = ratings.randomSplit(Array(0.8, 0.2))

    // Build the recommendation model using ALS on the training data
    val als = new ALS()
      .setMaxIter(n_iter)
      .setRegParam(RegParam)
      .setUserCol("userId")
      .setItemCol("movieId")
      .setRatingCol("rating")

    val model = als.fit(training)
    model.setColdStartStrategy("drop")
    val result = model.transform(testings)
    result.show()
    result.write.format("csv")
      .mode("overwrite")
      .save("result")
    // // Evaluate the model by computing the RMSE on the test data
    // val predictions = model.transform(test)
    // predictions.show()
    // val evaluator = new RegressionEvaluator()
    //   .setMetricName("rmse")
    //   .setLabelCol("rating")
    //   .setPredictionCol("prediction")
    // val rmse = evaluator.evaluate(predictions)
    // println(s"Root-mean-square error = $rmse")
  }
}
```

注释部分在训练时使用，根据QMSE（均方根误差）的大小来调整参数，此部分的代码从Data.txt文件中读取训练数据，转换成Dataframe的形式后，用ml的ALS进行训练，得到模型之后预测testings数据，并保存为csv格式。

结果

我们使用了一个主节点（hadoop1）和两个子节点（hadoop1，hadoop2）进行分布式的计算。

训练时随机划分训练和测试数据集，训练之后得到的均方根误差结果



以及一部分数据的训练结果与真实结果比较

userId	movieId	rating	prediction
29431	148	10.0	7.9803734
10587	148	6.0	7.4133615
51555	148	4.0	4.4443417
57053	148	7.0	6.0316033
52018	148	5.0	6.10683
28932	148	7.0	6.5953627
66563	148	7.0	6.529918
21513	148	6.0	5.913004
65836	148	7.0	5.6276913
2837	148	6.0	3.4598997
57620	148	7.0	8.3916855
15629	148	4.0	7.527989
47758	148	5.0	6.470684
53964	148	8.0	9.052822
73507	148	8.0	6.749236
53516	148	7.0	6.3036003
27233	148	5.0	9.000883
11339	148	6.0	7.166008
58078	148	6.0	6.9210606
47582	148	9.0	9.287761

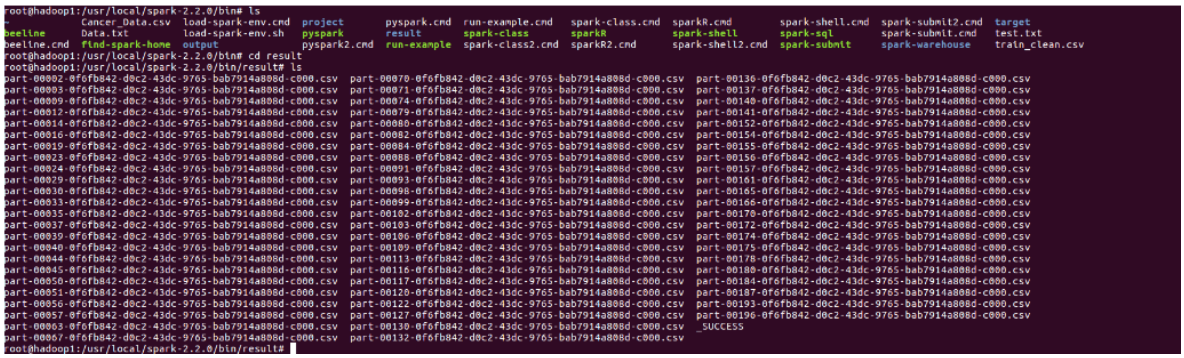
only showing top 20 rows

在预测时，显示一部分预测结果（与上图相比，此部分数据没有ratings标签）

userId	movieId	prediction
3	14075	7.953526
3	17265	7.802073
3	20021	6.9428897
3	27631	7.0030065
3	28497	6.7072835
3	20583	9.096042
3	11111	7.8815503
3	14345	7.4957232
3	1689	7.601284
1	11757	9.964866
3	11757	8.743565
3	19815	8.417049
3	1564	6.909642
3	6880	6.959638
3	14513	8.4374275
3	6178	6.429882
3	9919	8.095105
3	225	7.361757
3	28701	8.450335
3	2404	5.6403875

only showing top 20 rows

以及在运行文件目录下，所有结果保存在result文件夹中，并且分布式地保存在了多个csv文件中。



我们利用python来后续处理这些结果，通过依次读取这些表格并归并，来得到最终的预测结果。

在训练模型式，我们调整了两个参数：迭代次数Iter和正则化参数Regparam，比较了不同参数值下的误差RMSE，如下表所示

Iter	Regparam	RMSE
1	0.01	7.58
10	0.01	1.17
10	0.1	1.16
10	1	1.57
10	0.001	1.22
20	0.01	1.17

于是我们最终训练模型的参数为$Iter = 10, Regparam = 0.1$

五、遇到的问题

1. Spark的mllib读取自定义数据类型时需要引入新的依赖支持

```
val spark = SparkSession
        .builder
        .appName("CF")
        .getOrCreate()
import spark.implicits._
```

2. Spark训练结果一直是NAN

由于我们数据的有效数据量太少，仅为0.7%，因此需要使用冷策略进行训练来保证不会获得NAN

```
model.setColdStartStrategy("drop")
```

3. 将训练结果保存，是一个片段一个片段的分布式的文件

假设有500个测试数据需要给出评分，最终存储的形式是500个单独的文件

我们尝试了设置分片仅为1

```
df.coalesce(1)
```

但是这会导致一个单位的负荷量过大，最终stackoverflow，程序出错。

六、组员分工

胡晨旭：算法搭建以及代码运行

王宇琪：算法模型搭建以及数据处理

张智为：代码运行以及报告整合