$Keras$实现电影推荐系统算法及$Spark-ALS$推荐包应用(下)

1. 什么是 ALS算法

ALS是alternating least squares的缩写 , 意为交替最小二乘法，也就是固定损失函数其中之一个参数，对另一个参数进行迭代更新求解的方法。为了更形象地了解这个算法，我门来看看模型的损失函数：

$$loss = \sum_{i,u}(r_{u,i} - \sum_{k=1}^{K} R_{U*I})^2 + \lambda(\sum^{U} P_U^2 + \sum^{I} Q_I^2)$$

$$R_{U*I} = P_{U*K} \cdot Q_{K*I}$$

这就是官方API中所使用的损失函数，可见其结构与上篇中SVD模型类似，只是增加了基于用户隐形特征和电影隐形特征的L2惩罚项。在训练时，我们固定P_{U*K},对Q_{K*I}求导并对其进行更新，然后我们可以对Q_{K*I}进行同样的操作，直到对参数进行指定次数迭代为止。而在spark中，由于我们可以将P_{U*K}和Q_{K*I}的更新同时并行进行，于是大大加快了模型训练的速度。

下面我们先用spark对数据进行必要的预处理，再运用ALS模块对数据进行训练。

2.数据预处理

In [1]:

```
##加载所需包，导入spark环境
import findspark
findspark.init("/u/cs451/packages/spark")
import random
from pyspark.sql import SparkSession
from pyspark import SparkContext, SparkConf
sc = SparkContext(appName="YourTest", master="local[2]")
```

In [2]:

```
from pyspark.mllib.recommendation import ALS
import math
import numpy as np
```

In [4]:

```
###读取评分数据和电影数据，并对他们的格式进行处理
rawData = sc.textFile('user_movies.csv')
movieData = sc.textFile('hot_movies.csv')
```

In [7]:

```
movieData.take(2)
```

Out[7]:

```
['20645098,8.2,小王子', '26259677,8.3,垫底辣妹']
```

In [8]:

```
movieData = movieData.map(lambda line: line.split(","))
movieData =movieData.map(lambda x: (int(x[0]),(x[1],x[2])))
movieData.take(3)
```

Out[8]:

```
[(20645098, ('8.2', '小王子')),
 (26259677, ('8.3', '垫底辣妹')),
 (11808948, ('7.2', '海绵宝宝'))]
```

In [10]:

```
##评分数据的总条数
rawData.count()
```

Out[10]:

```
1007397
```

In [11]:

```
rawData.take(4)
```

Out[11]:

```
['adamwzw,20645098,4',
 'baka_mono,20645098,3',
 'iRayc,20645098,2',
 'blueandgreen,20645098,3']
```

In [12]:

```
rawRatings = rawData.map(lambda line: line.split(","))
rawRatings.first()
```

Out[12]:

```
['adamwzw', '20645098', '4']
```

注意这里因为ALS默认的输入数据类型必须是数字，所以我们必须重新对用户id进行映射

In [13]:

```
######重新定义user_ID
```

In [14]:

```
user_ID=rawRatings.map(lambda x:x[0]).distinct().zipWithUniqueId().collectAsMap(
)
```

In [15]:

```
ratings =rawRatings.map(lambda x: [int(user_ID[x[0]]),int(x[1]),float(x[2])]).ca
che()
```

In [16]:

```
ratings.take(3)
```

Out[16]:

```
[[0, 20645098, 4.0], [2, 20645098, 3.0], [4, 20645098, 2.0]]
```

3.模型训练

下面我们用处理好的数据来训练模型：

在ALS中主要参数有：

rank*p*：*隐形特征的数量（K值），iterations：更新参数时迭代的次数*iterations，*和 lambda*：L2正则项的惩罚项。

我们下面定义函数来对模型进行cross-validation后的参数调整：（在衡量模型的正确率时，我们使用的是mean square error)

In [25]:

```
def ALS_tuning(rank,iteration,regularization):
    model = ALS.train(train, rank, iterations=iteration,lambda_=regularization)
    predict=model.predictAll(vali.map(lambda x:[x[0],x[1]]))
    predict=predict.map(lambda x: ((x[0], x[1]), x[2]))
    vali_ratings=vali.map(lambda x: ((x[0], x[1]), x[2]))
    join_a=vali_ratings.join(predict)
    result=join_a.map(lambda x: (x[1][0]-x[1][1])**2).mean()
    return result
```

In [44]:

```
rank_p=[3,5]
iterations=[15,20]
regularization=[0.1,0.5]
para=[]
mse=[]

for rank in rank_p:
    for itera in iterations:
        for reg in regularization:
            para.append(str(rank)+'and'+str(itera)+'and'+str(reg))
            print(para)
            mse.append(ALS_tuning(rank,itera,reg))
            print(mse)
```

```
['3and15and0.1']
[1.5653349856128496]
['3and15and0.1', '3and15and0.5']
[1.5653349856128496, 1.5004063384593576]
['3and15and0.1', '3and15and0.5', '3and20and0.1']
[1.5653349856128496, 1.5004063384593576, 1.5669970270925955]
['3and15and0.1', '3and15and0.5', '3and20and0.1', '3and20and0.5']
[1.5653349856128496, 1.5004063384593576, 1.5669970270925955, 1.49990
24473018931]
['3and15and0.1', '3and15and0.5', '3and20and0.1', '3and20and0.5', '5a
nd15and0.1']
[1.5653349856128496, 1.5004063384593576, 1.5669970270925955, 1.49990
24473018931, 1.6182548620673405]
['3and15and0.1', '3and15and0.5', '3and20and0.1', '3and20and0.5', '5a
nd15and0.1', '5and15and0.5']
[1.5653349856128496, 1.5004063384593576, 1.5669970270925955, 1.49990
24473018931, 1.6182548620673405, 1.500303503383629]
['3and15and0.1', '3and15and0.5', '3and20and0.1', '3and20and0.5', '5a
nd15and0.1', '5and15and0.5', '5and20and0.1']
[1.5653349856128496, 1.5004063384593576, 1.5669970270925955, 1.49990
24473018931, 1.6182548620673405, 1.500303503383629, 1.62164055607692
45]
['3and15and0.1', '3and15and0.5', '3and20and0.1', '3and20and0.5', '5a
nd15and0.1', '5and15and0.5', '5and20and0.1', '5and20and0.5']
[1.5653349856128496, 1.5004063384593576, 1.5669970270925955, 1.49990
24473018931, 1.6182548620673405, 1.500303503383629, 1.62164055607692
45, 1.5002263008882624]
```

In [45]:

```
np.asarray([a for a in zip(para,mse)])
```

Out[45]:

```
array([['3and15and0.1', '1.5653349856128496'],
       ['3and15and0.5', '1.5004063384593576'],
       ['3and20and0.1', '1.5669970270925955'],
       ['3and20and0.5', '1.4999024473018931'],
       ['5and15and0.1', '1.6182548620673405'],
       ['5and15and0.5', '1.500303503383629'],
       ['5and20and0.1', '1.6216405560769245'],
       ['5and20and0.5', '1.5002263008882624']],
      dtype='<U18')
```

In [56]:

```
rank_p=[3,4]
iterations=[20]
regularization=[0.2,0.3]
para=[]
mse=[]

for rank in rank_p:
    for itera in iterations:
        for reg in regularization:
            para.append(str(rank)+'and'+str(itera)+'and'+str(reg))
            print(para)
            mse.append(ALS_tuning(rank,itera,reg))
            print(mse)
```

```
['3and20and0.2']
[1.422341798104477]
['3and20and0.2', '3and20and0.3']
[1.422341798104477, 1.3953502701809763]
['3and20and0.2', '3and20and0.3', '4and20and0.2']
[1.422341798104477, 1.3953502701809763, 1.4315015649567486]
['3and20and0.2', '3and20and0.3', '4and20and0.2', '4and20and0.3']
[1.422341798104477, 1.3953502701809763, 1.4315015649567486, 1.406264
61094181]
```

In [58]:

```
np.asarray([a for a in zip(para,mse)])
```

Out[58]:

```
array([['3and20and0.2', '1.422341798104477'],
       ['3and20and0.3', '1.3953502701809763'],
       ['4and20and0.2', '1.4315015649567486'],
       ['4and20and0.3', '1.40626461094181']],
      dtype='<U18')
```

通过上面的参数调整，我们发现在rank_p，隐形特征等于3，迭代次数等于20，和L2正则惩罚项为0.2的情况下， 我们的模型损失值最小。

下面我们用所有的参数在最优的参数情况下训练一次：

In [29]:

```
#####这里将数据后需要先take.再运行后面的代码
#####get complete train data
complete_train=train.union(vali).cache()
```

In [30]:

```
complete_train.take(3)
```

Out[30]:

```
[[2, 20645098, 3.0], [4, 20645098, 2.0], [1, 20645098, 3.0]]
```

In [28]:

```
######train model
```

In [32]:

```
model = ALS.train(complete_train, 3, iterations=20,lambda_=0.2)
```

In [35]:

```
#####predict and accuracy
```

In [36]:

```
predict=model.predictAll(test.map(lambda x:[x[0],x[1]]))
predict=predict.map(lambda x: ((x[0], x[1]), x[2]))
test_ratings=test.map(lambda x: ((x[0], x[1]), x[2]))
join_a=test_ratings.join(predict)
result=join_a.map(lambda x: np.abs(x[1][0]-x[1][1])).mean()
result
```

Out[36]:

```
0.8673831109258564
```

In [36]:

```
#####先用所有的数据和最优参数train model
```

In [29]:

```
from time import time

start_time = time()
model = ALS.train(ratings, 3, iterations=20,lambda_=0.2)
duration = time() - start_time
duration
```

Out[29]:

```
18.88502025604248
```

我们这里还是用absolute mean error 作为衡量test质量的指标。可以发现训练的出的正确率效果很不错，平均仅有0.86的绝

对误差。正确率比上篇中加上用户偏差和电影偏差的模型还要高。从效率的角度来看，ALS更是远远胜过了前面的模型，即使

在训练前面的模型时我们使用的是TPU,速度也远不及这里的分布式环境。

下面然我们来看看ALS模型的推荐出的影片是哪些：

In [31]:

```
####我们先来定义一个推荐电影的方程
def recommend(userid):
    ####用户id的viewed过的电影
    viewed=ratings.filter(lambda x: x[0]==userid).map(lambda x: x[1])
    viewed=viewed.collect()
    ####创造出用户id的Rdd -list
    user_rdd=ratings.map(lambda x:x[1]).distinct().map(lambda x:[userid,x])
    ########predict 结果
    predict=model.predictAll(user_rdd)
    predict_rating=predict.map(lambda x: (x.product, x.rating))
    #######将结果与电影名字join起来,然后map成：电影id,用户评分，豆瓣评分，电影名字
    predictions=predict_rating.join(movieData).map(lambda x:(x[0],x[1][0],x[1][1
][0],x[1][1][1]))
    #######排除掉prediction 中viewed过的电影
    prediction_exclu=predictions.filter(lambda x: x[0] not in viewed)
    ####按评分排序并推荐前30个
    return prediction_exclu.takeOrdered(30, key=lambda x: -x[1])
```

In [ ]:

```
#####for 666
```

In [32]:

```
recommend(666)
```

Out[32]:

```
[(26393561, 4.073942417677685, '8.8', '小萝莉的猴神大叔'),
 (25859495, 3.9477324606170185, '8.2', '思悼'),
 (10533913, 3.944381584066339, '8.8', '头脑特工队'),
 (25766754, 3.8902499233585512, '8.2', '年轻气盛'),
 (25955491, 3.8896251858926467, '8.6', '罪恶之家'),
 (24397586, 3.874895619511557, '8.5', '小羊肖恩'),
 (11520649, 3.863773897052777, '8.2', '麦克法兰'),
 (3592854, 3.8482411994894505, '8.5', '疯狂的麦克斯4：狂暴之路'),
 (26304167, 3.832152195900491, '8.1', '出租车'),
 (24405378, 3.8180418113000987, '8.5', '王牌特工：特工学院'),
 (25855951, 3.795045508174383, '8.3', '贝利叶一家'),
 (19897541, 3.7852999494682456, '9.0', '机动战士高达 THE ORIGIN I 青瞳
的卡斯巴尔'),
 (21937450, 3.7803132850799157, '8.2', '国际市场'),
 (2973079, 3.7481520007902827, '8.2', '霍比特人3：五军之战'),
 (10741643, 3.731546749216214, '8.3', '我的个神啊'),
 (10773239, 3.704494021970995, '8.1', '小男孩'),
 (25823132, 3.7031296608926505, '8.1', '暗杀'),
 (6845667, 3.6928226118148695, '8.0', '秘密特工'),
 (25727048, 3.6895431082854415, '7.8', '福尔摩斯先生'),
 (26270517, 3.6705457773850103, '7.8', '愚人节'),
 (25870236, 3.6608568514571678, '7.8', '可爱的你'),
 (25919385, 3.659415491585847, '7.8', '长寿商会'),
 (3445457, 3.644739677965468, '7.8', '无境之兽'),
 (26252157, 3.6116678294243076, '7.5', '龙三和他的七人党'),
 (26356488, 3.585654376221374, '7.9', '1944'),
 (10792633, 3.580840282409308, '7.8', '金衣女人'),
 (1866473, 3.578334210024443, '7.8', '蚁人'),
 (25922902, 3.5746511172364013, '7.5', '唇上之歌'),
 (10727641, 3.5306911740256357, '7.8', '碟中谍5：神秘国度'),
 (25728010, 3.524641922571302, '7.5', '老手')]
```

In [ ]:

```
####for 6666
```

In [33]:

```
recommend(6666)
```

Out[33]:

```
[(26393561, 3.802002706123595, '8.8', '小萝莉的猴神大叔'),
 (25859495, 3.680035566162225, '8.2', '思悼'),
 (25766754, 3.575823065702563, '8.2', '年轻气盛'),
 (23761370, 3.567120981573659, '8.4', '速度与激情7'),
 (11520649, 3.5494856287382426, '8.2', '麦克法兰'),
 (25870236, 3.483030962015766, '7.8', '可爱的你'),
 (10533913, 3.474314490785445, '8.8', '头脑特工队'),
 (25955491, 3.4693284963143567, '8.6', '罪恶之家'),
 (21937450, 3.4445767107477456, '8.2', '国际市场'),
 (25922902, 3.4427554896816464, '7.5', '唇上之歌'),
 (3592854, 3.4322532097707814, '8.5', '疯狂的麦克斯4：狂暴之路'),
 (24397586, 3.428049178730186, '8.5', '小羊肖恩'),
 (26304167, 3.4023229540888895, '8.1', '出租车'),
 (10741643, 3.3976080418228016, '8.3', '我的个神啊'),
 (25919385, 3.396062495455446, '7.8', '长寿商会'),
 (10773239, 3.3949650598225283, '8.1', '小男孩'),
 (25727048, 3.3897576021409073, '7.8', '福尔摩斯先生'),
 (26252157, 3.3852267274835395, '7.5', '龙三和他的七人党'),
 (25753326, 3.379752771970045, '7.1', '巴霍巴利王(上)'),
 (2973079, 3.369105140024237, '8.2', '霍比特人3：五军之战'),
 (26270517, 3.365113244437646, '7.8', '愚人节'),
 (25823132, 3.36365823909052, '8.1', '暗杀'),
 (24405378, 3.3601199297556836, '8.5', '王牌特工：特工学院'),
 (24719063, 3.3584109641910813, '7.9', '烈日灼心'),
 (25855951, 3.356138710415259, '8.3', '贝利叶一家'),
 (25728010, 3.3337361327144253, '7.5', '老手'),
 (26384515, 3.3263391402070206, '7.6', '这里的黎明静悄悄'),
 (5446197, 3.3214687756883734, '7.2', '铁拳'),
 (26289144, 3.3136699682438424, '7.6', '滚蛋吧！肿瘤君'),
 (6845667, 3.2874868264608645, '8.0', '秘密特工')]
```

从推荐的结果来看，通过ALS预测出的结果似乎在考虑用户的偏好的同时，加强了对同类型高分影片的推荐，我们可以看到ALS也向两位用户都推荐了《小萝莉的猴神大叔》这部电影，但这部电影因为在同类型电影中评分较高，被ALS放到了推荐榜单的首位。

4.模型总结

对比在项目中的所有模型，我们可以总结出从推荐的精准性来讲，由基于SVD的神经网络的模型的推荐结果是最精确的，其次是在spark环境中用ALS搭建的模型，参考用户偏向和电影偏向的RSVD训练出的模型其平均绝对误差也在1以内。但如果我们同时考虑到训练模型的效率，运用分布式的spark绝对是最好的解决方案。

另外，对于新的用户进行推荐，即冷启动问题，我们可以采用向其推荐排行榜的方法，即上篇中的根据用户平均电影评分的排行榜或者根据豆瓣评分的排行榜。亦或我们可以让用户注册账号时选择喜欢的电影种类或者让他给特定电影评分以获得用户的偏好属性。

5.文献及参考资料

[1] Zhou,Y., Wilkinson, D., Schreiber, R., Rong, P. Large-scale Parallel Collaborative Filtering for the Netflix Prize.

https://endymecy.gitbooks.io/spark-ml-source-analysis/content/推荐/papers/Large-scale%20Parallel%20Collaborative%20Filtering%20the%20Netflix%20Prize.pdf (https://endymecy.gitbooks.io/spark-ml-source-analysis/content/推荐/papers/Large-scale%20Parallel%20Collaborative%20Filtering%20the%20Netflix%20Prize.pdf)

[2] Koren,Y., Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model. https://www.cs.rochester.edu/twiki/pub/Main/HarpSeminar/Factorization_Meets_the_Neighborhood-_a_Multifaceted_Collaborative_Filtering_Model.pdf (https://www.cs.rochester.edu/twiki/pub/Main/HarpSeminar/Factorization_Meets_the_Neighborhood-_a_Multifaceted_Collaborative_Filtering_Model.pdf)

知乎相关文章：

https://zhuanlan.zhihu.com/p/24220475 (https://zhuanlan.zhihu.com/p/24220475)

https://colobu.com/2015/11/30/movie-recommendation-for-douban-users-by-spark-mllib/ (https://colobu.com/2015/11/30/movie-recommendation-for-douban-users-by-spark-mllib/)

https://www.zhihu.com/question/31509438/answer/52268608 (https://www.zhihu.com/question/31509438/answer/52268608)

Kaggle文章：https://www.kaggle.com/morrisb/how-to-recommend-anything-deep-recommender (https://www.kaggle.com/morrisb/how-to-recommend-anything-deep-recommender)

CSDN文章：https://blog.csdn.net/zhongkejingwang/article/details/43083603 (https://blog.csdn.net/zhongkejingwang/article/details/43083603)

spark ALS API 官网：https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html (https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html)