修订日志：

|  |  |  |
| --- | --- | --- |
| 修改日期 | 修改人 | 备注 |
| 2019/11/16 | 刘晨旭 | 添加本人负责部分 |
| 11/18 | 季昕 | 添加本人负责部分 |
| 11/18 | 李玫 | 增加es算法部分 |
|  |  |  |
|  |  |  |

**电影相似度算法**

本项目推荐算法主要基于三个方面：无用户登录时的冷启动推荐，有用户登录时用户的协同过滤推荐，点击电影详情页时的相似电影推荐。

1. **无用户登录时的冷启动推荐**

输入：一千部电影的评分人数，上映时间，评分

输出：冷启动时推荐的48部（不足则全部推荐）的电影。

首先寻找评分人数最多的48部电影，而后根据评分高低推荐其中的24部电影；此外根据上映时间，寻找24部最新的电影，将48部电影作为冷启动时的推荐电影。

**2. 点击电影详情页时的相似电影推荐**

此部分主要算法是对电影之间相似度的计算，得到相似度后对每一个电影寻找和它最相似的8部电影进行推荐。接下来介绍相似度计算的算法。

输入：一千条包含名称、导演、主演、简介等信息的电影基本信息以及十万多条用户对这些电影的的评分、评论信息

输出：一千部电影两两之间的相似度

**算法设计**

我们先根据电影本身的信息做相似度计算，再根据用户对电影的评价基于协同过滤思想计算相似度，最后对这两个相似度加权求和。

电影的详细信息我们将其分为几类：文本类信息（电影名称，电影简介等）、标签类信息（导演等）、集合类信息（主演，编剧等）、总评分信息（总评分），用户评分信息。我们需要衡量每个信息的相似度，再求得总的相似度，下面介绍每类信息相似度计算方法。

1. **文本类信息**

文本类信息的计算方式如下。设该文本为，对分词得到

我们使用词嵌入的方式对词进行表示，word2vec词向量模型得到每个词的词向量, , .., ）

然后使用词向量表示

的相似度使用的余弦距离衡量，由于余弦距离的范围为-1到1，为了统一相似度结果在0到1之间，我们对其做了调整。

+ 1

这样就得到了文本类信息的相似度。

1. **标签类信息**

而标签类信息如导演等，其相似度我们认为两个标签相等是相似的，而不相等则不相似，可以使用一个01函数去衡量，设标签为*G*，两个电影标签的相似度为

,

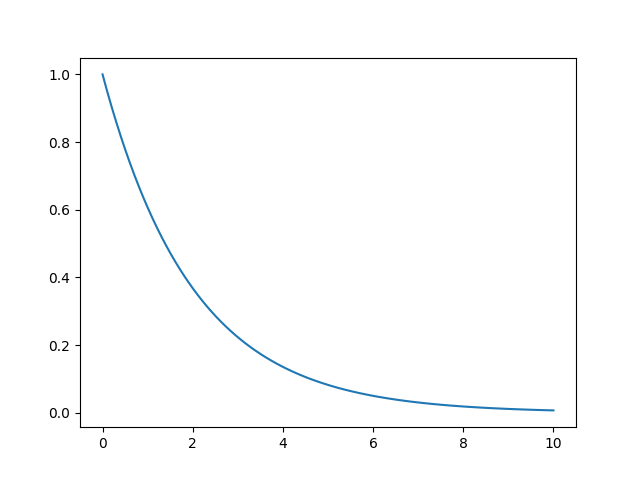
1. **集合类信息**

集合类信息如电影主演，编剧等信息，我们认为其交集越大越相似。设集合为D, 则其相似度可描述为：

1. **总评分类信息**

两个电影之间的评分相似度取决于评分的差值，差值越小相似度越大。设电影的评分为，这个相似度是一个关于的降序函数，满足定义域为0处函数值为1，二阶导函数大于0。我们定义评分信息相似度函数为

其中控制函数的陡峭程度，在本次作业中设置为2。



1. **用户评分信息**

电影自身信息的相似度均可以得到了，下面介绍基于协同过滤思想的电影相似度计算。同时喜欢两部电影的人占分别喜欢两部电影总人数的比例越高，说明两个电影相似度越高。设对两部电影都有评价的用户集合为，喜欢每部电影的用户集合为,，则这两个电影的相似度函数可以表示为

各个信息之间的相似度均可以得到，设电影为M，文本类信息为T，电影标签类信息为*G*，电影集合类信息为*D*，总评分信息为，用户评价信息为

因此电影整体之间的相似度为个信息的加权求和。

其中，，为权重，使结果归一化。

以上为计算两个电影之间的相似度，我们利用此算法，对所有电影两两之间执行此计算过程，得到任意两两电影之间的相似度，保存起来供下游推荐任务使用。

**算法实现**

我们的电影相似度算法通过pyspark实现，读取mysql上的电影数据，计算出相似度，存储到mysql中。pyspark是spark的python接口实现，方便灵活，便于快速编写spark分布式程序。

首先编写计算每种信息相似度的函数，通过pyspark.sql.functions中的udf封装为udf供后续使用。通过sqlContext接口访问mysql，获取电影数据，使用dataFrame数据结构保存。对文本信息电影名称、电影简介，使用jieba分词工具进行分词，利用pyspark.ml.feature.Word2Vec训练word2vec词向量，将电影名称、电影简介转换为向量表征，然后对向量做2阶标准化。接下来对我们的电影数据dataframe做join操作，以求得电影两两之间的相似度。为避免重复Join的条件为item1.id<item2.id。然后利用封装好的udf求相似度，最后加权求和求得总的相似度存入mysql数据库中。

1. **有用户登录时用户的协同过滤推荐**

输入：十万多条用户对电影的评分信息

输出：对每个用户推荐一定数目的电影。

**算法设计**

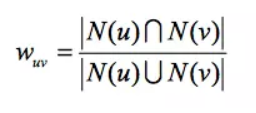
基于用户的协同过滤推荐算法先使用统计技术寻找与目标用户有相同喜好电影的邻居，然后根据目标用户的邻居的喜好产生向目标用户的电影推荐。基本原理就是利用用户访问行为的相似性来互相推荐用户可能感兴趣的电影。

算法主要包含两个步骤：

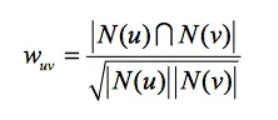
1. **找到和目标用户兴趣相似的用户群；**

这个步骤最关键的就是计算两个用户之间的相似度，给定用户, ，令表示用户u曾经有过正反馈的电影集合，令表示用户曾经有过正反馈的电影集合。

可以通过Jaccard公式衡量



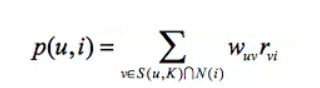
也可以通过余弦相似度衡量



找到这个用户群中用户喜欢的，而目标用户没有看过(听过\点击过)的物品推荐给该用户。

1. **推荐给目标用户感兴趣的物品。**

求得用户两两之间相似度之后，算法会给用户推荐和他兴趣最相似的K个用户喜欢的物品，以下公式度量了算法中用户u对物品i的感兴趣程度



**算法实现**

我们同样使用pyspark实现基于用户的协同过滤算法。首先使用SQLContext从mysql中读取所有从豆瓣爬取的十万多条评论，我们利用pyspark.ml.recommendation中的ALS模型进行训练，设置迭代次数为5，regParam为0.1，coldStartStrategy为drop方式。训练之后使用模型为每个用户用户推荐三十部电影，存到Hbase数据库中的movie\_sim\_1表中。

# 搜索算法设计

## 分类搜索

一部电影通常可能包含多个分类，在mysql中需要多条信息存储。而在es中，可以用一条doc就保存下来。Es对每一个字段都建立了索引，因此能够在大数据集上获得很高的速度。

在存储时，我们把类型字段作为keyword进行存储，因此是作为不分词情况存储的。在搜索的过程中，使用term query查询，term是代表完全匹配，也就是精确查询，搜索前不会再对搜索词进行分词。因为es默认只返回10条数据，因此需要指定返回结果的最大数量为1000以保证返回完整的结果。设置结果排序方法为：按照评分高低倒序排列电影。

在初次尝试时，每次获取到的结果速度都很慢，后来发现kibana中took时间只有几毫秒，实际返回时间有2s左右，后来发现是返回数据量过大，网络传输成为了瓶颈。因此，将返回的field设定为仅返回name。（id信息一定会返回）

## 综合搜索

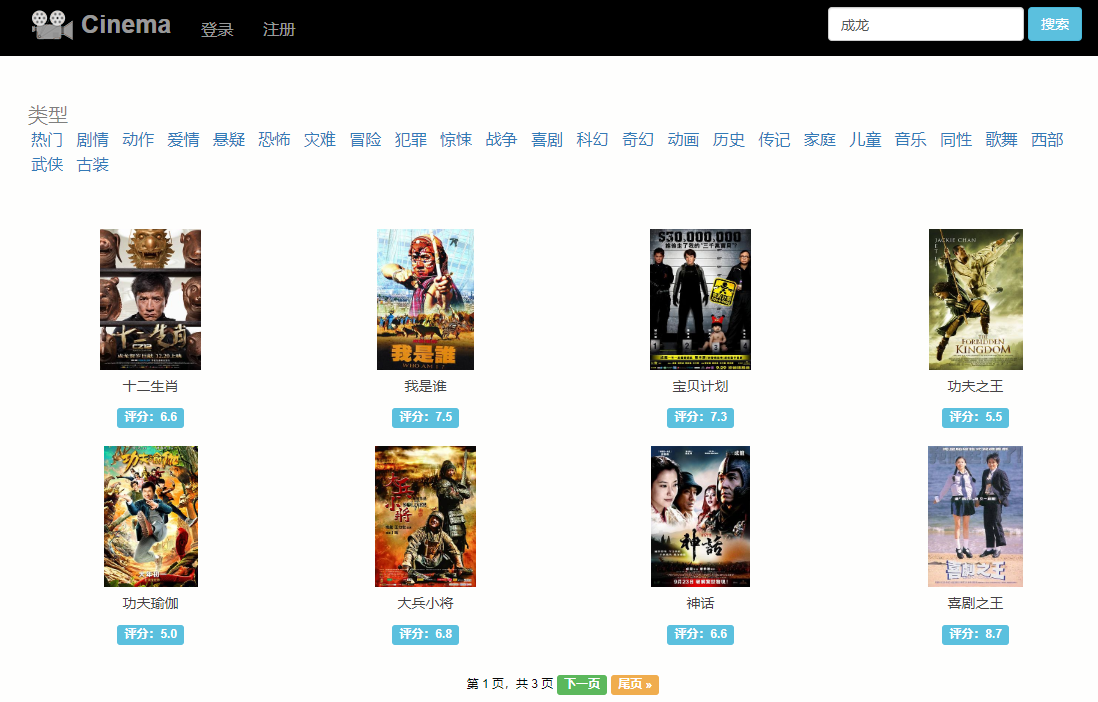
Es对比起传统的关系型数据库，对全文搜索有着显著的优势，同样能够实现分词搜索，让返回的结果能够更加智能准确。而在设计搜索框的搜索算法时，为了提高用户的体验，满足用户的各种需求，需要综合考量多种因素。

* + 1. function\_score

首先，es的搜索结果是根据的内置的默认评分来决定的，优先返回\_score高的结果。为了实现我们自定义评分算法，使用fuction\_score方法来对算法的每个部分进行分数的计算。

* + 1. 支持演员名、导演、编剧名的完全匹配

有的时候用户想要通过查看某一个演员、某一个导演或者是某一个编剧的相关电影，可以直接通过主页的搜索框进行搜索，免去了其他复杂的操作步骤。例如，想看成龙的相关电影，直接搜索成龙即可出现其相关的电影内容：



为了实现term的完全匹配，采用了bool复合查询，对几个子查询的结果分数叠加起来。计算出总共的分数。

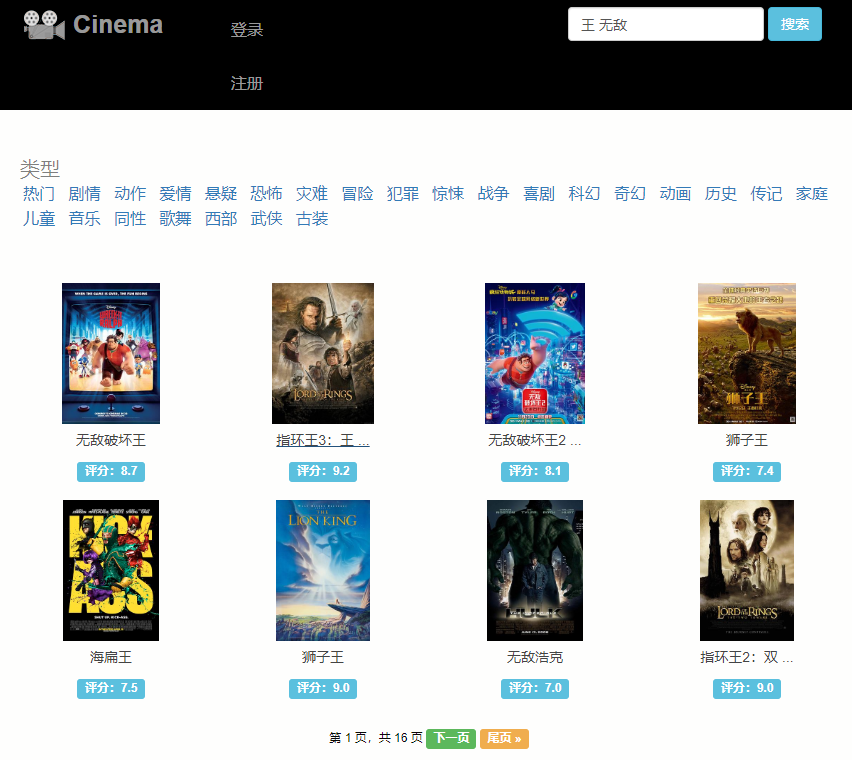
总共三个子查询，都是should查询，因为我们无法确定用户输入的内容是在演员、编剧还是导演字段，所以将三个子查询都不是必须的内容，匹配上加分，不匹配则不加分。

用于人名是一个特有的词，不需要被分词。在数据库中我们将这三个字段都存储为了keyword，因此可以让每一个子查询都使用term完全匹配查询。

minimum\_should\_match设置为1，意味至少需要匹配一个子查询的结果才会被返回。

* + 1. 支持电影名、电影别名、电影内容的分词搜索

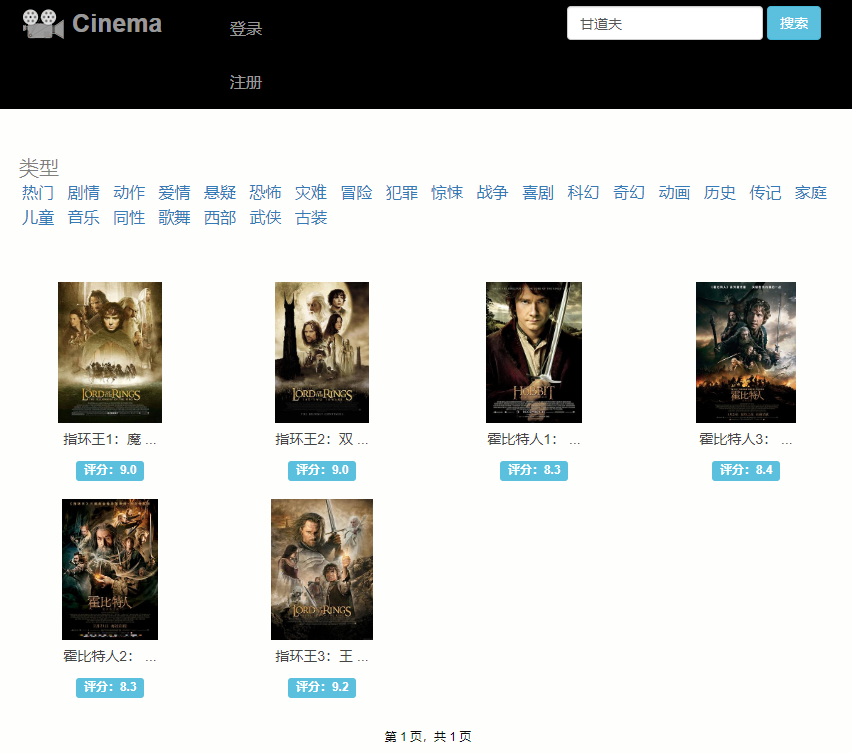
最通常出现的情况是对电影名或者电影的别名的搜索，但是很多时候用户不能够准确地记住电影的名字/别名，因此采用match的方式，对用户输入的内容进行分词后，再到倒排索引的库内进行词项的匹配，进行打分来获得最有可能的结果。



由于不知道用户输入的是电影的名字还是别名，因此使用multi\_match查询来同时对多个字段进行搜索。

使用“best\_fields”搜索类型，即对每个字段都进行搜索，但是最终返回的分数仅由最匹配的字段所决定，因为用户搜索的内容显然更可能是来自一个字段的。但是同时为了综合考虑同时出现在多个字段的匹配情况，使用tie\_breaker=0.3来进行调整。即将其他query的分数，乘以tie\_breaker，然后与最高匹配的query的分数综合在一起进行计算。

同时，可能出现用户并不记得电影名字或者演员名，只记得电影里的一些特定内容或者角色的名字，这个时候就需要用到简介字段。用户搜索的内容中只要包含简介中的角色名或者内容，都可以增加对应的电影分数。要实现这一点同样只要将brief字段加入到multi\_match的fields中，让其参与分数的计算。



由于用户搜索名字的概率比别名的概率更大，因此将name的权重设置为8，alias的权重设置为2，brief的权重设置为1。设置整体的minimum\_should\_match为30%，防止一些过少的匹配量对整体分数的影响。

* + 1. 综合查询

综合以上三点就可得到最终的综合查询结果。将3.2.2和3.2.3中的算法分别作为两个子查询，使用bool查询中的should查询将两个查询连接起来，即最终的总的分数将由两个查询的分数叠加得到。

由于完全匹配的term查询的匹配程度要高于部分匹配match查询，因此对以上3.2.2和3.2.3的两个子查询分别设置boost为0.8和0.2，使得完全匹配的结果优先出现。