Recommender System

李易非 李灿晨 熊苗

一、 数据初探

本次项目中，我们的训练集有 `ratings.csv` 与 `animes.csv` 两部分， `ratings.csv` 包括用户 ID，动漫 ID，以及用户对相应动漫的打分，为 -1 的打分值无意义； `animes.csv` 中包含动漫的种种信息，动漫 ID、名字、类别、级数、平均分、观看人数等等。

需要注意的是，本次训练集中存在大量没有任何评分信息的用户与没有任何用户评分信息的动漫，这就带来了冷启动问题，我们将在冷启动问题部分专门叙述。整体来说，该数据集有如下的一些数字特征：

```
num of ratings: 6336428
num of movies: 12294
num of users: 73514
num of animes with true rating: 9926
num of users with true rating: 69599
ratings per user: 86.19348695486573
ratings per movie: 515.4081665853262
user with max ratings: 3747
user with min ratings: 1
movie with max ratings: 34219
movie with min ratings: 1
recommendation count: 5868149
unrecommendation count: 468279
```

从上述数据中，我们可以看到，

- 有效记录共有 6336428 条，将大于等于 6 的评分视为推荐，小于 6 的评分视为不推荐，在有效用户记录中共有 5868149 条推荐记录，共有 468279 条不推荐记录；
- 不同 user 共计 73514 个，拥有有效评分的用户共计 69599 个；
- 不同的动漫总计 12294 个，拥有有效用户评分的动漫共计 9926 个；
- 在拥有有效评分的用户集合中，每个用户平均对 86 个动漫有评分记录，打分最多的用户共计打分 3737 部动漫，打分最少的用户打分 1 部动漫；
- 在拥有有效评分的动漫集合中，每个动漫平均收到 515 个评分记录，最多收到 34219 条评分记录，最少收到 1 条评分；

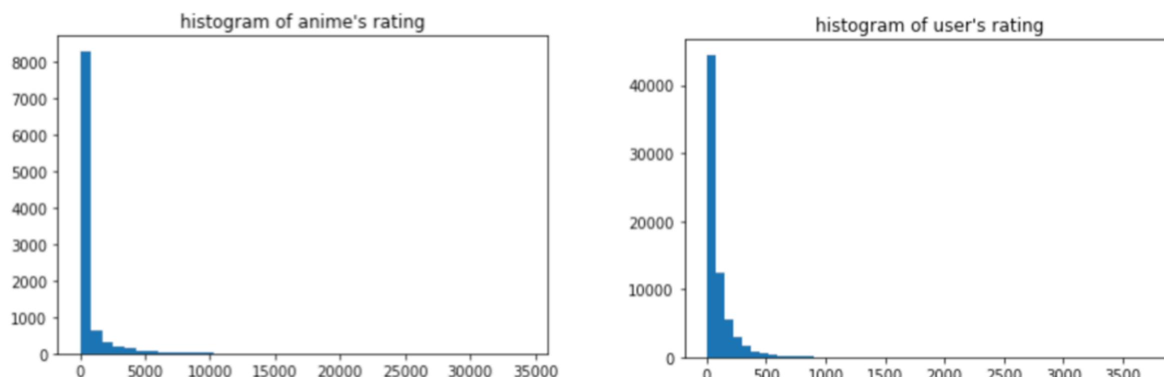
  进一步的，我们将以上特点总结为四点

极度稀疏性

用户共计 73514 个，动漫共计 12294 个，最理想的情况下，应有 73514 * 12294 = 903,781,116 条评分记录，而我们仅有 6336482 条有效记录，6336482 / 903781116 约等于 0.7%，稀缺性异常严重；

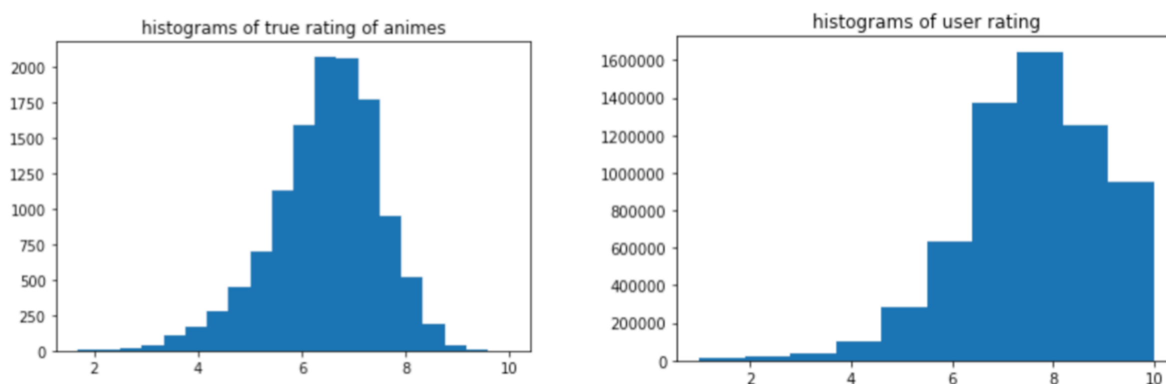
长尾性

绝大部分用户打分次数较少；绝大部份动漫收到的打分较少；



极度偏差性

下面两幅图中，左图为 `animes.csv` 中动漫总体评分的直方图，右图代表 `ratings.csv` 中出现的有效评分的直方图；从两幅图的对比中我们可以看到，真实评分趋于 6.5 附近，而我们的数据集中评分整体偏高，集中在 8 附近；有效记录共有 6336428 条，将大于等于 6 的评分视为推荐，小于 6 的评分视为不推荐，在有效用户记录中共有 5,868,149 条推荐记录，共有 468,279 条不推荐记录；二者整整 **相差一个数量级**，这也为我们本次的模型建立带来了挑战与困难。



二、模型建立

推荐系统在我们的生活中无处不在，百度令人作呕的广告推荐、淘宝购物的猜你喜欢、今日头条的新闻推荐都利用了多种推荐算法来完成各自的目的。一般来说，推荐系统分析用户过去的浏览、点击、购买等行为，建立合理的模型（通常是混合模型），并最终做出推荐。

推荐系统要面临的挑战与问题也大同小异，数据稀疏性、长尾性、时间性等。通常来说，推荐系统可以分为以下几个大类：

1. **content-based (CB)** 基于内容推荐，该方法根据用户历史观看的记录得到用户对应的特征，将与

用户兴趣特征相似的产品进行推荐。

2. **collaborative filtering (CF)** 协同过滤，协同过滤不关注产品的实际特征，而是通过用户-物品评分对发掘用户与产品之间的相互作用。通常，CF 方法又分为 model based 基于模型与 Neighborhood 基于临域的两种方法，Neighborhood 方法也可分为 item-based 与 user-based。一般情况下，一个系统中的 user 远多于 item 数量，通过 item-based 的临域模型可以得到更加可靠的推荐结果；
3. **基于聚类的方法**，聚类方法通常利用 k-means 或 fuzzy c-means 模糊聚类方法将用户或物品聚集为不同的群体，基于群体特征进行推荐；

1、 协同过滤算法

协同过滤方法假设如果某个用户过去喜欢某种产品，那么TA 现在仍喜欢与此产品相似的产品，我们就可以分析已经收集到的用户-产品评分对中所呈现的用户与产品的相互作用，根据这些相互作用把和用户之前喜欢的相似的产品推荐给她。

它最大的一个优势是它只依赖于用户过去的行为，不需要预处理产品或用户的特征，不依赖于应用中的特有领域，并且能克服 CBF 方法内在的推荐缺陷。

但协同过滤方法也有一些困难：

- 冷启动：
  - 对于一个新用户，我们缺乏 TA 对产品的评分，CF 无法提供可靠的产品推荐
  - 对于一种新的产品，类似的困难同样存在，CF 无法确定该把它推荐给哪些用户
- 可扩展性
  - CF 方法中可能涉及到数以百万计的用户为成千上万种产品提供的评分。
  - 传统的 CF 推荐算法（如 kNN）通常需要计算每对用户或产品之间的相似度，这中间产生的开销将会非常大。

以上这两种问题在我们处理的时候都有遇到，我们之后会详细介绍我们对此的解决方案。

基准预测模型 Baseline Predictor

CF 模型尝试捕获用户和电影之间的交互。但是大多数观察到的评级值也同时受是用户或电影本身的影响，这与交互无关。比如某些用户表现出比其他用户打分更高的倾向，以及一些电影容易获得比其他电影获得更高的评级。
我们将在 baseline predictor 中封装那些不涉及交互的影响因素。通过基准预测模型可以隔离那些偏差因素，真正代表用户电影交互的部分。
我们用 μ 表示电影的总体平均得分，值得一提的是，我们在这里尝试过两种计算方式，一种是使用我们已知的用户-电影评分，一种是使用 label 里给出的关于 anime 的全局评分。关于电影的全局作用我们用 b_i表示，关于用户的全局作用我们用b_u 表示。

$$b_{u,i} = \mu + b_u + b_i$$

对于全局作用的实现，我们主要使用SQL语句实现。

- 计算每部电影的平均得分 μ

```
this.mu = sparkSess
  .sql(s"SELECT AVG($labelCol) FROM data")
  .collect()(0) // 选出一行返回  Row 类型
  .getDouble(0)
println(s"mu: $mu")
```

- 计算 b_i， 为了提高精度，我们通过 λ_2 进行压缩惩罚

$$b_i = \frac{\sum_u (r_{ui} - \mu)}{\lambda_2 + |R(i)|}$$

我们把这个式子的计算拆分成分子分母的计算过程，利用 SQL 语句的 group by 特性进行同一部电影的所有用户的评分的加和，并计算出 $R(i)$

然后再利用 SQL 语句进行列运算，得到 b_i

```
val itemTemp = sparkSess
      .sql(
        s"SELECT $itemCol, SUM($labelCol), COUNT($labelCol) FROM data GROUP BY $itemCol")
      .toDF(itemCol, "sumRating", "countRating")

itemEffect = sparkSess
      .sql(
        s"SELECT $itemCol, (sumRating - countRating * $mu) / ($lambda_2 + countRating) FROM items"
      )
      .toDF(itemCol, "itemEffect")
```

- 计算 b_u

$$b_u = \frac{\sum_i (r_{ui} - \mu - b_i)}{\lambda_3 + |R(u)|}$$

b_u 的计算过程和 b_i 相同，只是需要多减去 b_i， 这里需要把 r_{ui} 和 b_i 的求和一同计算，利用 INNER JOIN 这个操作把数据表和上面求出来的 itemEffect 的表。

step 1： 得到 userID + user 看过的电影的全局作用 的表格

```
SELECT data.$userCol, SUM(itemEffect.itemEffect)
FROM data INNER JOIN itemEffect ON itemEffect.$itemCol = data.$itemCol
GROUP BY data.$userCol
```

step 2: 计算看过用户看过的每部电影的评分和 以及 电影的全局效应和

```
SELECT users.$userCol, sumRatingUser, countRatingUser, sumItemEffect
FROM users INNER JOIN user2 on users.$userCol = user2.$userCol


SELECT $userCol, (sumRatingUser - countRatingUser * $mu - sumItemEffect) / ($lambda_3 + countRatingUser)
from join1

```

step 3: 求出用户的全局作用

```
SELECT $userCol, (sumRatingUser - countRatingUser * $mu - sumItemEffect) / ($lambda_3 + countRatingUser)
from join1
```

仅仅通过上述流程做出的 baseline predictor，可以达到以下的效果

	precision	recall	f1	accuracy
正样本	94.4%	98.3%	96.3%	93.1%
负样本	56.3%	27.1%	36.6%	

对于 baselinePredictor 的改进

传统的 baselinePredictor 的应用场景多在于物品综合评分不存在的情形，但在我们的项目中， `animes.csv` 已经给定了几乎所有电影的综合评分（仍有230部空缺，用均值填充），所以在 baselinePredictor 的计算过程中，

$$b_{u,i} = \mu + b_u + b_i$$

μ 可以直接用真实评分的均值代替，也就是 6.48，之前的平均分为 7.8，这也能一定程度上改善模型偏斜的问题，相当于添加了一个强有力的先验。令电影综合评分为 r_i ，则此时物品效应 b_i 直接为 $r_i - \mu$；用户效应的计算公式保持不变。进一步看，上述公式退化成了

$$b_{u,i} = r_i + b_u$$

利用改进的 baseline Predictor 还有一个显著的好处在于冷启动问题的解决，之前的 baseline 仅仅利用了 `ratings.csv` 的评分信息，而在我们的样本中，仍存在大量没有评分信息的用户与电影，在这里我们将没有评分信息的用户的 b_u 设置为0，不采取任何先验的假设，

也就是，综合评分再加上用户效应作为我们的 baseline Predictor，单纯利用改进的 baseline Predictor 预测结果如下

	precision	recall	f1	accuracy
正样本	94.4%	98.3%	96.3%	93.1%
负样本	56.3%	27.1%	36.6%	

可以看到，综合评分信息是一个强有力的信息量，他改善了原先 baselinePredictor 预测的各项指标，且在一定程度上减缓了数据集极大偏斜性的问题。

接下来我们介绍我们实现的协同过滤方法，有基于模型的MF矩阵分解，基于邻域的 KNN 。

基于邻域的方法 KNN

基于邻域的协同过滤方法有基于用户和基于产品两者，因为用户的数量远远多于产品，所以基于产品的 KNN 通常可以获得更好的预测精度和可扩展性[1]，所以我们采取了基于产品的 KNN 方法。同时处于简单的考虑，我们先实现了传统的 KNN 模型。

计算相似度

我们采取 Pearson correlation 公式作为相似度的定义。

$$s^p_{mn} = \frac{\sum_v (r_{v,m} - \bar{r}^m)(r_{v,n} - \bar{r}^n)}{\sqrt{\sum_v (r_{v,m} - \bar{r}^m)^2 \sum_v (r_{v,n} - \bar{r}^n)^2}}$$

其中 $\bar{r}_m$ 和 $\bar{r}_n$ 分别表示电影 m 和 n 获得的评分平均值。

实现

我们在 Spark 通过 Dataframe API调用 SQL 语句来实现 $S_{m,n}$ 的计算。

step 1: 计算模型的baseline $r_{u,m}$

```
SELECT data.$userCol, data.$itemCol, $mu + userEffect.userEffect + itemEffect.itemEffect FROM data, userEffect, itemEffect
WHERE data.$userCol = userEffect.$userCol AND data.$itemCol = itemEffect.$itemCol
```

step 2: 计算 $r_{u,m} - \bar{r}^m$

```
SELECT data.$userCol, data.$itemCol, (data.$labelCol - baselinePredictor)
FROM data, baseline
WHERE data.$userCol = baseline.$userCol AND data.$itemCol = baseline.$itemCol
```

step 3: 得到 （电影 u） X （电影 m ）及其相似度的表格

```
SELECT substract1.$itemCol, substract1.substract, substract2.$itemCol, substract2.substract
FROM substract AS substract1, substract AS substract2
WHERE substract1.$itemCol < substract2.$itemCol AND substract1.$userCol = substract2.$userCol
```

step 4: 得到分子分母各个元素的求和，使用 group by

```
SELECT item1Col, item2Col, COUNT(item1Substract), SUM(item1Substract * item2Substract), SUM(POW(item1Substract, 2)), SUM(POW(item2Substract, 2))
FROM mergeTable
GROUP BY item1Col, item2Col
```

step 5: 列运算得到相似度表格

```
1   SELECT item1, item2, (((count - 1) * numer) / ((count - 1 + $lambda_8) * POW(deno_1
    * deno_2, 0.5))) AS similarity
2   FROM mergeTable
```

选取邻居

为了预测用户 u 对电影 m 的评分值，我们首先从 Pu 中选取与电影 m 有最
高相似度的特定数量的电影，这些电影形成 u-m 对的邻居(neighborhood)，
记为N(m;u)。

由于Spark没有官方实现KNN，所以我们对于 KNN 的实现主要是基于其 Dataframe API 和内嵌的 SQL 语法。想要选取用户看过的所有电影中与待预测电影最相似的 K 部电影，但 SQL 语法里没有对分组取前 K 个的操作，我们用了一种比较机械的办法，即分成 K次，每次取每个用户-电影对(u, m)里面的和电影 m 相似度最高的一部电影，再在下次选取的时候把这次选出的集合给 except 掉，这样就可以得到 K对最相似的电影。

伪代码实现

```
1   1. 先通过 JOIN 得到一张表 A: $userCol,$itemCol,userSimItem(用户看过的和目标电影有相似度的电影), similarity
2   2. for(循环 K次):
3     res1 =  每次根据 (u, m) 对分组，求出A中和 userSimItem 最相近的电影，需要排除掉之前已经求出的集合 res0
4     res0 和上一次求出的最相近的电影求并集
```

计算预测值

利用以下公式求出 p_{ij}

$$\hat{\rho_{ij}} = \frac{\sum_u (r_{ui} - bui)(r_{uj} - b_{uj}))}{\sqrt{\sum_u (r_{ui} - b_{ui})^2 \sum_u (r_{uj} - b_{uj})^2}}$$

利用以下公式进行压缩相似度改进预测精度：

$$s_i j = \frac{n_{ij} - 1}{n_{ij} - 1 + \lambda_8} \rho_{ij}$$

这个过程也是利用 SQL 语句的 Group By 和 SUM POW 等运算功能，分别求出分子分母，然后再执行一次列运算。具体的实现过程和上述代码相似，故不在此赘述。

整体框架搭建

在这一步我们仿照 Spark 官方API自己搭建了 KNN算法的API和一整套 pipeline 的流程。

```
1  val knn = new KNN()
2        .setUserCol("userId")
3        .setItemCol("animeId")
4        .setLabelCol("rating")
5        .setK(5)
6        .setLambda2(50)
7        .setLambda3(10)
```

```
8           .setLambda9(10)
9           .setSimilarityThreshold(0.3)
10          .setMu(6.473902)
11
12        val yeevaluator = new Yeevaluator()
13          .setMetricName("f1_p")
14          .setLabelCol("rating")
15          .setPredictionCol("prediction")
16          .setThreshold(6.0)
17        // 设置基本参数
18
19        val predictions = knn.transform(test)
```

算法优化

我们在取 前K个相似邻居的时候发现计算过程非常缓慢，而且这个还不仅仅是训练过程缓慢，整个预测过程都非常缓慢，所以我们尝试了一些优化。

- threshold filtering
  - 我们观察到有些电影之间的相似度甚至是负值，这说明用户倾向于给 A电影打高分，但倾向于给B电影打低分，所以两部电影极度不相似，我们可以在选 K个邻居的过程里就应该先把这些相似性非常小的电影筛除，因为即使他们进入选择过程，也只是增加了表格的大小，对结果没有太多助益。
  - 于是我们只把相似度大于某个阈值的邻居全被囊括进来
- 缩小计算表格的体量
  - 为了进一步减少 SQL 语句执行的工作量，我们决定把表格的容量缩小。如果不缩小的话，再计算相似度的时候第一个并起来的大表约有700 M，计算量非常大。
  - 我们减少的过程是先看测试集，把测试集需要的用户电影对选择进来，其他不会用到的信息先删除掉，这样就可以大大缩小表格的容量。

  最终，KNN 在 k = 5 时预测结果如下，可以看到，KNN 方法负样本的 recall 过于低，负样本整体表现很差。

	precision	recall	f1	accuracy
正样本	95.2%	98.5%	96.8%	94.0%
负样本	58.1%	29.8%	39.4%	

基于模型的推荐算法 MF

MF (Matrix Factorization) 算法的主要思想是通过矩阵分解得到用户的特征矩阵和电影的特征矩阵；

它的主要过程为如下步骤：

- 将用户和和所有电影一同组成一个大的矩阵
- 进行矩阵分解（SVD），分解为用户矩阵和内容矩阵。
  - 用户矩阵的列相当于 feature, 行为用户

    - 内容矩阵的列相当于电影，行为 feature
    - 任何一个用户的一行点乘电影的一列，就得到了一个评分
- 考虑到有大量缺失值，需要进行参数惩罚

值得庆幸的是，spark 官方实现了基于 MF 的推荐算法，通过调用 `ALS` 方法，我们就可以获得 MF 方法的实现，在 `ALS` 方法中，我们也可以设定诸多可调参数，譬如交替最小二乘过程中的惩罚系数，因子个数（即为分解矩阵后用户矩阵的行数与动漫矩阵的列数），迭代次数等等；一个典型的调用过程如下

```
1   val als = new ALS()
2          .setMaxIter(5)
3          .setUserCol("userId")
4          .setItemCol("animeId")
5          .setRatingCol("rating")
6          .setColdStartStrategy("drop")
7          .setRegParam(0.05)
8          .setRank(5)
9   val alsModel = als.fit(train)
10  val predictions = alsModel.transform(test)
```

需要特别注意的是，`setColdStartStrategy` 专门用来设定冷启动处理策略，在 Spark 中仅支持 `drop` 方法来忽略冷启动问题，否则就会在 `predictions` 中出现 NA 项；我们采取了自己的冷启动策略，会在冷启动部分详细叙述。

同时，我们还可以利用 Spark 提供的 `CrossValidator` API 来进行相关参数的选择，一个典型的调用过程如下

```
1   val als = new ALS()
2          .setMaxIter(5)
3          .setUserCol("userId")
4          .setItemCol("animeId")
5          .setRatingCol("rating")
6          .setColdStartStrategy("drop")
7          .setRegParam(0.05)
8          .setRank(5)
9   val regressionEvaluator = new RegressionEvaluator()
10             .setMetricName("rmse")
11             .setLabelCol("rating")
12             .setPredictionCol("prediction")
13
14   val paramGrid = new ParamGridBuilder()
15             .addGrid(als.regParam, Array(0.05, 0.1, 0.15))
16             .addGrid(als.rank, Array(5, 10, 15))
17             .build()
18
19  val cv = new CrossValidator()
20           .setEstimator(als)
21           .setEvaluator(regressionEvaluator)
22           .setEstimatorParamMaps(paramGrid)
23           .setNumFolds(5)
24           .setParallelism(3)
25
26  val cvModel = cv.fit(train)
```

```
27    val predictions = cvModel.transform(test)
```

在上述过程中， `ALS` 作为我们的 estimator， `regressionEvaluator` 作为 evaluator， `paramGrid` 构建参数网格； `CrossValidator` 模型可以设定 `NumFolds` 参数，在这里我们进行 5-fold cross validation 来选择最优参数；

有一点不足在于，本次项目的着眼点不在于 rmse，而在于正样本的各项指标，是一个分类而非回归问题，虽然 Spark 的 `BinaryClassificationEvaluator` 可以处理 0，1分类问题，但是 `als` 方法输出的值并不是0、1，在 `CrossValidator` 模型调用中过程中是一个相对闭合的流程，我们无法先将 als 输出结果转化为二值再传给 `BinaryClassificationEvaluator` ，为了解决以上的需求，我们继承 `Eevaluator` 类，完成了 `Yeevaluator` 的封装；

`Yeevaluator` 自动将接受到的连续值按照给定的 threshold 转化为二值变量，可设定的 metricName 有 `precision_p` ， `recall_p` ， `f1_p` ， `f1_n` ， `accuracy` 用来进行不同目的的模型选择，以下为简化的 API

```
1   class Yeevaluator(override val uid: String)
2       extends Evaluator
3       with DefaultParamsWritable {
4           val metricNameArray = Array("f1_p", "recall_p", "precision_p", "accuracy",
    "f1_n")
5       override def evaluate(data: Dataset[_]): Double = {}
6           def score(data: Dataset[_]): Array[Double] = {}
7           override def copy(extra: ParamMap): Yeevaluator = defaultCopy(extra)
8           private def convertToBinary(value: Double, threshold: Double): Double = {}
9     }
```

通过上述实现，我们就可以在 crossvalidation 中，加入自定义的 evaluator, 完成特定工作。

参数选择

经过交叉验证，我们最终选定 `rank= 5` ， `regParam = 0.05` 两个参数，单纯利用 MF 模型，能够得到如下的结果（惰性处理冷启动：

	precision	recall	f1	accuracy
正样本	95.7%	96.2%	96.0%	92.5%
负样本	49.5%	46.0%	47.7%	

可以看到，正样本表现很好，但是负样本表现很差，这也源于我们之前提到的极度严重的数据偏斜，整个模型都偏向于打较高的分数，但 MF 模型整体表现是优于 KNN 模型的。

2、基于内容的推荐算法

相比于协同过滤算法，基于内容的推荐算法作为推荐算法本身来说并不占优太多的优势：其基于用户历史记录的特性注定了基于内容的推荐算法无法有效地发掘用户的兴趣爱好，而只能是在用户已有的兴趣爱好中为用户推荐更多的相关内容，局限用户的视野，同时也无法很好地应对冷启动问题。因此，在实际的使用中，基于内容的推荐算法很少用于主要的推荐系统中，通常来说仅仅是对于其他推荐算法的补

充作用。但是，在本次实现的推荐系统中，由于最终的评分指标是给出具体用户是否会喜欢具体电影的结果，因此，我们认为基于内容的推荐算法对于本次项目也许是合适的选择。

数据特征请求

由于采用基于内容的推荐算法，动画的特征对算法的影响是至关重要的。通常来说，一个平庸的模型配上大量的数据比先进的模型配上不足的数据有更好的结果。因此，我们根据本次项目 Slide 中给出的相应参考资料，尝试对 AniList GraphQL API 进行了数据请求，以扩充本次给出的 `anime.csv` 文件。

AniList 所给出的 API 是十分完整的，其中不仅有关于用户和动漫作品的全部数据，还拥有关于出版商、评论、个人播放列表等一系列信息，这个庞大的关系型数据库也向我们说明虽然用户个人偏好的相关的信息就藏在数据库的点点滴滴中，但如果要使用所有“可以利用”的信息去构建一个推荐算法是一件困难度远超过想象的事，我们所做的仅仅是通过其中的一小部分尝试预测用户的个人偏好。

通过对文档的阅读，我们最终确定下如下的额外信息用于扩充关于动漫的特征信息文件。

- `startDate.year` ：相应动漫的开始年份。
- `endDate.year` ：相应动漫停止更新的年份。
- `popularity` ：相应动漫的观众数量
- `trending` ：相应动漫在排行榜中的位置
- `tags.name` ：相应动漫具有的标签的名称
- `tags.category` ：相应动漫具有的标签所属的标签目录
- `isAdult` ：相应动漫是否为成人动漫

值得注意的是，我们在一开始尝试使用动漫的 `anime_id` 对 API 中 `Media` 类型的 `id` 域进行索引以确定相应的动漫，通过对前几个动漫的尝试也说明这种做法是正确的。但是，当我们花费了较长时间请求了较多数据之后，通过再一次的抽查，我们发现我们所获取的数据有很多名称与给出的动画名称并不相符。我们在一次检查了数据，发现给出的训练数据与 API 中所请求到的数据的一些差异。

- 给出的 `anime_id` 既可能是 API 中 `Media` 的 `id` ，也可能是 `idMal`
- 并非所有的动画都能通过其 `anime_id` 在 API 中找到
- 给出的动画名称一栏英文和日文音译混杂，不能仅通过日文名称判断动画是否符合

因此，我们为相应的请求代码做了如下的提升：针对一个动漫条目，依照其 `anime_id` 对 `id` 和 `idMal` 进行两个独立的请求，请求中同时获取相应动画的英文名和日文音译名，再使用正则表达式挑选首个单词与训练数据中动漫条目相等的条目取用，如不相等则认为此数据为空。

相应的关键请求及判断代码如下所示。

```python
def name_match(response, row, identifier):
    romaji = response['data'][identifier]['title']['romaji']
    english = response['data'][identifier]['title']['english']
    if romaji != None and re.findall('[a-zA-Z]*', romaji)[0].lower() in name_sequence[row].lower():
        return True
    if english != None and re.findall('[a-zA-Z]*', english)[0].lower() in name_sequence[row].lower():
        return True
    return False

query = """
query($id: Int) { id: Media(id: $id) {  title {   romaji   english  }  startDate {   year  }  endDate {   year  }  popularity  trending  tags {   name   category  }  isAdult } idMal: Media(idMal: $id) {  title {   romaji   english  }  startDate {   year  }  endDate {   year  }  popularity  trending  tags {   name   category  }  isAdult }}
"""
```

> 我们也尝试在 **AniDB** 使用爬虫获取关于用户的信息，但由于 AniDB 对于爬虫的方法十分严格，传统的访问几乎无法获取数据。我们使用了 `selenium` 进行了模拟浏览，但当我们数据获取过多时，我们收到了 AniDB 的人机身份验证，并且在一次验证之后使用同一账号和 IP 地址登陆的所有尝试都需要进行验证，这让我们放弃了从 AniDB 获取用户数据的想法。

特征预处理

根据我们所获取的特征，我们进行了一下预处理工作。

- 将请求得到的数据与原先给定的 `anime.csv` 使用动画的 id 一列的值进行 concate
- 除去 `tag_category` 一列的重复
- 将 `genre` 域种不同的标签使用分号隔开，同时除去引号以及此列存在的空值
- 生成训练集和测试集，其中测试集要求相拥用户的打分不能为 -1
- 对数据进行独热处理

在独热处理中，我们选用了 genre、type、start year、end year、tag_name、tag_category、is adult 几个指标。而对于剩下的包括动画的评分、受欢迎度、排行等，我们希望将这些信息在之后的处理中作为对动画受欢迎的的衡量指标，用于改善我们的结果。

我们将衡量年份的指标根据年份区间离散化，将其余的指标依据其标签进行离散化，最终得到了 342 维的独热特征向量。对于缺失的指标，我们并没有在特征预处理阶段进行过多的处理，而是使用我们之后所定义的判定算法进行对缺失值的处理。

近邻推荐

在基于内容的推荐算法中，在考虑到用户可能拥有较多的喜好内容和厌恶内容的方面，而这些方面根据经验能够使用用户曾经的观看内容进行简单的衡量。因此，我们也使用了近邻推荐的策略，并采用一个近邻作为判断准则，使用按位与方法简单地定义独热码的距离。

$d_{ij} = i \ \& \ j$

相应算法的伪代码表示如下。

```
given user_id and anime_id
positive_samples = get_user_posscored_aid(user_id)
negative_samples = get_user_negscored_aid(user_id)
pos_belief = max(common_entries(anime_id, _ for _ in positive_samples))
neg_belief = max(common_entries(anime_id, _ for _ in negative_samples))
if cannot_get(pos_belief or neg_belief) or pos_belief == neg_belief:
  return Recommend
else if pos_belief > neg_belief:
  return Recommend
else
  return Not Recommend
```

模型分析

我们在选出的 1000 个样本的测试集上运行了上述算法，并实现了以下结果。

	Liked	Not Liked
Recommended	883	32
Not Recommended	63	22

依据此数据，我们计算了相应的 precision 、 recall 以及成功的 f 指标。

```
positive_precision: 0.9334
positive_recall: 0.9650
negative_precision: 0.2588
negative_recall: 0.4074
positive_f: 0.9489
```

考虑到在算法的实现中，针对与用户偏好 / 不偏好的距离相等的距离以及评分列表中出现的无法在动画列表中找到的 `anime_id` 的情况，我们使用了动画列表中的平均分作为评分标准。我们在算法中对使用平均分做评分标准的样本数量进行了统计，结果显示使用现有的均分作为评分标准的样本数量占到了测试集总量的 18.9%。这不禁让我们开始思考一个问题：如果我们对所有的样本都使用相应动画的均值进行是否推荐的预测，会有怎样的结果？

3、冷启动问题的解决

我们要解决什么样的冷启动

- 首先，我们最终要完成的目标是，给定一个 user 和 anime, 给出 user 是否该推荐它
  - 保证 user 一定在给定训练集中，保证 anime 一定在给定训练集中
  - 但是，在数据探索中我们发现，很多电影**没有被任何一个用户评过分**，很多用户，**没有给任何一个电影评分**，所以，我们还是会遇到冷启动问题，且这个冷启动是多方面的
    - 一个评分很少的用户，如何推荐?
    - 一个没有评分的用户，如何推荐?
    - 一个没有评分的电影，如何推荐?
- 首先定义，我们语境下的没有出现过的用户以及没有出现过的电影
  - 没有出现过的用户：是指在训练集中，完全没有任何评分信息的用户

    - 没有出现过的电影：是指在训练集中，完全没有任何评分信息的电影
        - 特别恶劣：且缺少 **全局评分值**
- 其次看看目前决定的能够使用的方法：KNN, MF, CB，从原始模型考虑
    - kNN，在算相似度时，没得算（评分电影过少），或者相似度都差的很远，没有出现过的用户：
        - 没有出现过的用户　　：如果该电影评分很高，推荐之；若电影没有评分，考虑到整个数据集的偏斜，直接推荐即可；
        - 没有出现过的电影　　：baseline predictor；利用 CB，看看和他的兴趣是否相似；若这个动漫评分很高，直接推荐；
        - 特别恶劣的没有出现过的电影　：baseline predictor；利用 CB，看看和他的兴趣是否相似；
        - 计算相似度时，与目标电影相似的太少，或者用户本身评分过少　：baseline predictor；利用 CB，看看目标电影是否与他的兴趣相似；评分很高的动画，推荐之
    - MF，没有出现过的用户，没有出现过的电影：
        - 没有出现过的用户　　：对于评分很高的动画，推荐之
        - 没有出现过的电影　　：利用 CB，找到和他兴趣类似的，推荐之；评分很高的动漫，推荐之；

最终采取的冷启动方案

综合模型简化的角度考虑，我们最终采取了一种较为简单的解决方案。baseline Predictor 由用户与动漫两方面构成，结合真实平均得分得到的 baseline predictor 可以得到数据集中所有 user-anime 对的评分预测，所以我们将造成冷启动的问题的 user-anime 对直接赋给 baseline predictor 的值，该方法封装在 `KNN` 的 `coldFilling` 方法中，简化代码如下：

```
1  def coldFilling(testData: Dataset[_], predictions: Dataset[_], isLabel: Boolean)
   : Dataset[_] = {
2
3      val coldData = testData.select(userCol, itemCol)
4                         .except(predictions.select(userCol, itemCol))
5
6      val coldBaseLine = sparkSess
7        .sql(
8          s"SELECT coldData.$userCol, coldData.$itemCol, $mu +
   userEffect.userEffect + itemEffect.itemEffect FROM coldData, userEffect,
   itemEffect WHERE coldData.$userCol = userEffect.$userCol AND coldData.$itemCol =
   itemEffect.$itemCol")
9        .toDF(userCol, itemCol, "prediction")
10
11     val filledPredictions = predictions.union(coldBaseLine)
12   return filledPredictions
13
14   }
```

通过上述方法，我们可以较为简单高效的解决 KNN / MF 方法的冷启动问题，且基本不影响各个方法本来的准确率、召回率、f1值。

4、混合模型

MF 与 KNN 混合

鉴于时间原因，我们仅仅尝试了 MF 与 KNN 方法的线性组合，从二者结果中可以大致看到，二者对于正样本的预测精度相差无几，但 KNN 方法负样本 precision 较高，MF 方法负样本 recall 较高，通过合理的结合，希望结合二者的优点，在负样本上有所改善；从另一个方面进行思考，MF 方法着眼于全局，发掘隐藏因子，而 KNN 方法包含更多的局部信息量，二者相结合也可以互补局部与全局的信息。

```
1   val mfPredictions = knn.coldFilling(mfModel.transform(test))
2   val knnPredictions = knn.transform(test)
3   val weight = 0.85
4   val finalPrediction = spark
5             .sql(s"SELECT mf.userId, mf.animeId, mf.rating, $weight * mf.prediction
    + (1 - $weight) * knn.prediction AS prediction FROM mf INNER JOIN knn ON
    mf.userId = knn.userId AND mf.animeId = knn.animeId")
6             .toDF("userId", "animeId", "rating", "prediction")
7   val Array(positivePrecision,
8             positiveRecall,
9             f1_p,
10            negativePrecision,
11            negativeRecall,
12            f1_n, accuracy) = yeevaluator.score(finalPrediction)
```

最终我们选取了 0.85 作为二者结合的参数，MF 模型占据主导，KNN 模型作为辅助，baseLine Predictor 作为冷启动解决策略，达到了如下的预测精度，可以看到，负样本的表现较着单模型有了一定程度的提升。

weight: 0.8	precision	recall	f1	accuracy
正样本	95.6%	96.5%	96.0%	92.6%
负样本	50.4%	44.6%	47.4%	

weight: 0.85	precision	recall	f1	accuracy
正样本	95.6%	96.5%	96.0%	92.6%
负样本	50.2%	44.6%	47.2%	

weight: 0.9	precision	recall	f1	accuracy
正样本	95.6%	96.6%	96.0%	92.7%
负样本	49.4%	44.1%	47.3%	

混合模型可调参数总结

λ_1: MF 方法中的惩罚因子；

$rank$: MF 方法中需要提取的隐因子个数；

K: KNN 方法中的邻居个数

λ_2: baseline predictor 计算 item_effect 的惩罚因子；

λ_3: baseline predictor 计算 user_effect 的惩罚因子；

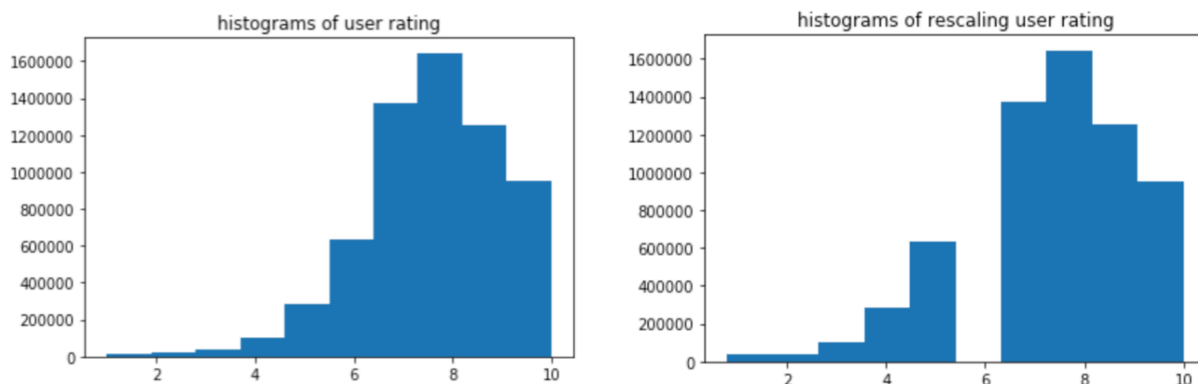
λ_4: 计算相似度时的惩罚因子；

λ_5: KNN 计算预测评分的惩罚因子；

$weight$: MF 与 KNN 混合模型中，MF 所占比重；

5、样本偏斜严重问题的解决探讨

这一部分将探讨我们思考过的偏斜问题解决方案，观察用户打分的分布,在有效用户记录中共有 5,868,149 条推荐记录，共有 468,279 条不推荐记录；二者整整 **相差一个数量级**，而在本就少的可怜的负样本中，还基本都聚集在 6 附近，这就进一步为算法带来了难度，考虑到这一点，我们对负样本做了 rescale，将负样本的得分乘以一个惩罚系数，旨在拉开负样本与正样本的区分度，提升负样本的表现



	precision	recall	f1	accuracy
正样本	89.9%	93.9%	91.9%	86.3%
负样本	63.5%	50.4%	56.2%	

可以看到，负样本情况有了较为明显的改善，但是正样本的 precision 损失惨重，值得一提的是，负样本的 recall 几乎没有任何改善，一定程度的 rescale 仍然不足以将一部分预测正确拉回 0 的区域。

在改进的 baseline Predictor 中，我们使用了均值更小的 μ ，但是 baseline Predictor 的预测结果显示，负样本 recall 的情况仍旧格外差，如下是我们思考的原因：

- 首先产生这种现象的直接原因在于，大部分为 0 的评分值被预测为了 1，也就是综合评分不低的动漫，因为用户的个人原因选为了 0；
- 综合评分平均分为 6.4，大部分评分集中在推荐与不推荐的界限之间，这部分数据本身就难以区分，仅仅利用 baseline Predictor 不足以充分建模用户的偏好；
- 我们整体数据集是偏斜的，集中了数据库中评分较高的一批用户的数据，评分较低的动画相关的低评分在数据集中出现较少，导致 baseline Predictor 很大一部分信息被浪费。

再者，如果能够良好的结合 CB 与 CF 模型，负样本的情况应该也会得到改善，鉴于时间有限，本次项目中我们没有将 CB 模型加入最终的混合模型中，也就损失了相当大一部分信息量，这一部分信息也许正是解决负样本问题的关键。

三、感想与总结

- SQL 语句的优化至关重要，利用 Spark Dataframe API 之后，我们可以通过更加简单的语句实现更加复杂的功能。在数据库中我们曾经学过 SQL 语句的优化，在这次 KNN 模型的 coding 过程中，我们遇到了很多很多难题，其中很大一部分在于 SQL 语句的优化，不合理的 SQL 语句会导致模型完全 untractable，甚至预测时间都不可预估；
- 多看官方 API 与源码，一开始在做 MF 模型的交叉验证时苦于没有合适的 Evaluator 使用，阅读源码后发现 evaluator 的实现比较简单，于是我们自己继承 Evaluator 类，实现了 `Yeevaluator` ，完成了本次项目中所有评估任务；
- 通过与使用平均分进行推荐的算法相比，我们也知道了准确率在推荐算法中并不能够成为唯一的指标：如果仅仅使用公众打分最高的内容作为对用户的推荐，这样虽然能够在冷启动时吸引用户，也能够使用户喜欢平台所推荐的大多数内容，但同时也造成了完全没有个性化的存在，大家看到的内容千篇一律，评分低的作品几乎被完全剥夺生存空间等。因此，根据用户偏好进行个性化定制的模型尽管会犯错，却依然能够依靠其对用户的个性化赢得用户的喜爱。这些经验也启发我们在实际应用中，以评分作为基准的准确率衡量虽然是便捷的，但更为复杂的推荐算法性能指标的衡量却可能极为复杂，可能包括用户在特定页面的滞留时间、对应用使用率的变化等，甚至需要对用户进行主观的用户调研以进行推荐算法的评价，获取对已有的推荐算法的性能优劣评估在实际场景下可能是一件长期并且成本巨大的工作；
- 在本次数据检索中，我们也发现自身从互联网搜索数据的水平实在不足，例如花费了过长的时间写了一个被 AniDB 封掉的爬虫等。我们也希望大数据应用强化训练课程能够在之后的培养方案中加入数据检索的内容，如介绍优质的开放数据来源、服务端 API 调用方法以及爬虫使用等；

三、future work

- 将可调参数用 `Param` 类型进行封装，就可以利用 `CrossValidator` API 进行交叉验证，本次项目中 KNN 的参数没有进行交叉验证调参，仅仅通过经验与论文建议选择了参数。
- 继续优化 KNN 模型的 SQL 语句，本次项目时间较紧，我们只做了比较粗糙的优化，肯定还有更大的优化空间；
- 鉴于时间原因，本次项目中，我们的 Content-based 模型没有很好的与 CF 结合，最终的混合模型遗失了相当大一部分信息量，改善负样本的关键也在于利用更多更广的信息，我们期望可以在假期继续将 CB 做完善，将其与 CF 模型结合，看看能不能从本质上提升负样本各项指标；

四、References

[1] Recommender Systems Handbook

[2] Bell, Robert M., and Yehuda Koren. "Lessons from the Netflix prize challenge." *Acm Sigkdd Explorations Newsletter* 9.2 (2007): 75-79.

[3] 吴金龙, "Netflix Prize 中的协同过滤算法" *北京大学博士研究生论文* (2010)

[4] Aslanian, Ehsan, Mohammadreza Radmanesh, and Mahdi Jalili. "Hybrid recommender systems based on content feature relationship." *IEEE Transactions on Industrial Informatics* (2016).