基于 Spark 的音乐推荐算法实现

凌浩东
复旦大学大数据学院

日期：2022 年 6 月 8 日

摘　　要

本文是 DATA130015.01，复旦大学大规模分布式课程的期末项目报告。本项目基于听歌历史，给用户进行音乐推荐。首先，基于任务要求，本项目实现了基于用户和基于物品的协同过滤算法。然后，考虑到本任务属于隐式反馈推荐问题，实现了有权重的交替最小平方（ALS）模型。最后，本项目设置不同的并行度，对算法的可拓展性进行了讨论。代码可以访问 github::DATA130015.01_final_pj。核心代码和实现的截图见附录。

关键词： 协同过滤，隐式反馈，交替最小平方，spark

1　问题概述

本任务的目的是：利用用户听音乐历史数据，实现协同过滤算法，返回给定用户可能喜爱的音乐。本任务是典型的隐式反馈（Implicit Feedback）推荐问题。本任务中，用户没有对音乐进行打分，无法判断用户对歌的好恶程度，因此在基于用户和基于物品的协同过滤算法中，将“听过”近似代表为“喜欢”。同时，我们也实现了Hu et al. (2008) 中的算法来解决隐式反馈问题。

2　数据集

数据集来源于kaggle::WSDM2018 (The 11th ACM International Conference on Web Search and Data Mining)，由 KKBOX 提供，数据中包含用户的听歌记录，特别的是，数据指明了用户在一个月内是否再次听了这首歌。

3　算法

3.1　算法概述

3.1.1　基于用户的协同过滤

基于用户的协同过滤大概过程如下：

1. 计算用户之间的余弦相似度。假设目标用户 u 听过的歌集为 I_u，用户 v 听过的歌集为 I_v，$I_u \cap I_v = I$，则两者的余弦相似度为

$$w_{u,v} = \frac{\sum_{k\in I} r_{u,k} r_{v,k}}{\sqrt{\sum_{i\in I_u} r_{u,i}^2}\sqrt{\sum_{j\in I_j} r_{v,j}^2}}$$

在本任务中，由于用户并没有打分，因此所有的 $r_{u,i}$ 都为 1。

2. 找到与目标用户 u 最接近的 k 个用户，记为用户 u 的邻居 G_u。将 G_u 中用户听过的所有歌曲，剔除掉 u 听过的歌曲，得到候选的歌曲目录 C。
3. 对于 C 中的每首歌，在 G_u 中预测 u 对他的喜好程度。在本任务中，对于某一首候选歌曲，我们简单地将 G_u 中所有听过他的用户与 u 的相似度求和，选择最高的 N 个。

3.1.2 基于物品的协同过滤

基于物品的协同过滤大概过程如下：

1. 计算歌曲之间的余弦相似度。
2. 对于目标用户 u，对于他听过的任意一首歌 $i \in I_u$，找到 k 首与之最相近的歌曲，加入候选目录 C。最后剔除在 I_u 中的歌。
3. 对于 C 中的每首歌 c，计算与 I_u 的相似度：

$$w_{c,I_n} = \sum_{i \in I_n} w_{c,i}$$

按照相似度由大到小排列，取前 N 首。

3.1.3 处理隐式反馈

在之前的两种方法中，我们隐式地做了一个假设：如果一个用户没有听过某首歌，那他对这首歌的打分为 0。然而，用户没听过一首歌，可能只是不知道有这首歌的存在。同样，用户听过一首歌，也有可能只是偶然刷到，其实对这首歌没什么感觉。也就是说，我们不能直接从用户听歌历史中直接得到用户对某一首歌的喜好程度，只能借助用户的听歌历史来间接推断用户对该歌曲的喜好程度。这就是隐式反馈与显式反馈推荐问题的区别。

- 显式反馈：知道用户对于音乐的打分，可以从打分中直观地得到用户对于音乐的喜好程度。
- 隐式反馈：只能从用户行为中推测用户对于音乐的喜好程度。

根据Hu et al. (2008)，我们将 $r_{u,i}$ 称作一个观察：$r_{u,i}$ 记录了观察到的用户听这首歌的次数。然后，我们设定一个 0-1 变量 $p_{u,i}$，称作偏好：

$$p_{u,i} = \begin{cases} 1 & r_{u,i} > 0 \\ 0 & r_{u,i} = 0 \end{cases}$$

，

偏好变量只是我们对于用户喜好的推测值，而对于这个偏好的置信度为 $c_{u,i}$：

$$c_{u,i} = 1 + \alpha r_{u,i}$$

通常 α 被设为 40。从置信度的表达式中我们可以看出，如果用户重复听了某一首歌，那么我们对偏好变量 $p_{u,i} = 1$ 的信心就强；即使用户没有听过某首歌，我们对他有可能喜欢这首歌的置信度也不是 0。

然后，我们求解以下优化问题即可：

$$\min_{x_*,y_*} \sum_{u,i} c_{u,i}(p_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

这也相当于是带权重的交替最小平方（alternating least square, ALS）协同过滤算法。

3.2 算法实现

对于基于用户和基于物品的协同过滤算法，我们主要使用`spark sql` 实现。对于带权重的 ALS 模型，在`spark ml` 库中已经有了成熟的实现，我们调用即可。

3.2.1 离线计算与数据采样

以基于用户的方法为例，共有 7377418 条左右的记录，其中约有 30000 个用户，每个用户平均有 200 条听歌记录。为某一个用户计算时，

- 计算相似度：需要约 $200 \times 7000000 \times 4 = O(10^9)Bytes = 1GB$ 的内存。
- Top-N 推荐：假设选择最相近的 20 个用户，需要约 $20 \times 200 \times 4 = O(10^4)Bytes = 10KB$ 的内存。

因此计算相似度花费的内存过大。我们采用离线计算的方法：提前计算用户、歌曲之间的相似度并存储起来。在为某个用户做推荐时，直接调用。

在实测过程中，离线计算所需的存储空间超过了计算资源所能承受。粗略计算，计算相似度需要 $7000000 \times 7000000 \times 4 = O(10^{13})Bytes = O(10^4)GB$，超过计算资源。因此，我们选择随机抽取 10% 的数据作为训练集。

3.3 算法评估

以用户 42 为例，使用三种推荐方法得到的结果如下图：

将全部的用户听歌作为 Ground Truth，用 10% 用户记录训练模型。用准确率来评估模型效果。

$$\text{准确率} = \frac{TP}{TP + FP}$$

其中，TP 代表真正例的数目，FP 代表假正例的数目。

设置随机种子 42，随机选择 100 位用户进行推荐，计算出基于用户的协同过滤准确度为 0.21，基于物品的准确度为 0.03，带权重的 ALS 模型的准确度为 0.38。考虑隐性反馈问题后，模型的准确率有了明显的提升。

4 并行度分析

分布式系统的优势在于具有良好的可扩展性，在面对大数据问题，能将数据和任务分配给多个子结点，从而提升整体的效率。通过合理调整并行度，可以让集群得到充分的利用。当使用`DataFrame` 时，创建的分区数等于 `spark.sql.shuffle.partitions` 参数，默认值为 200。当

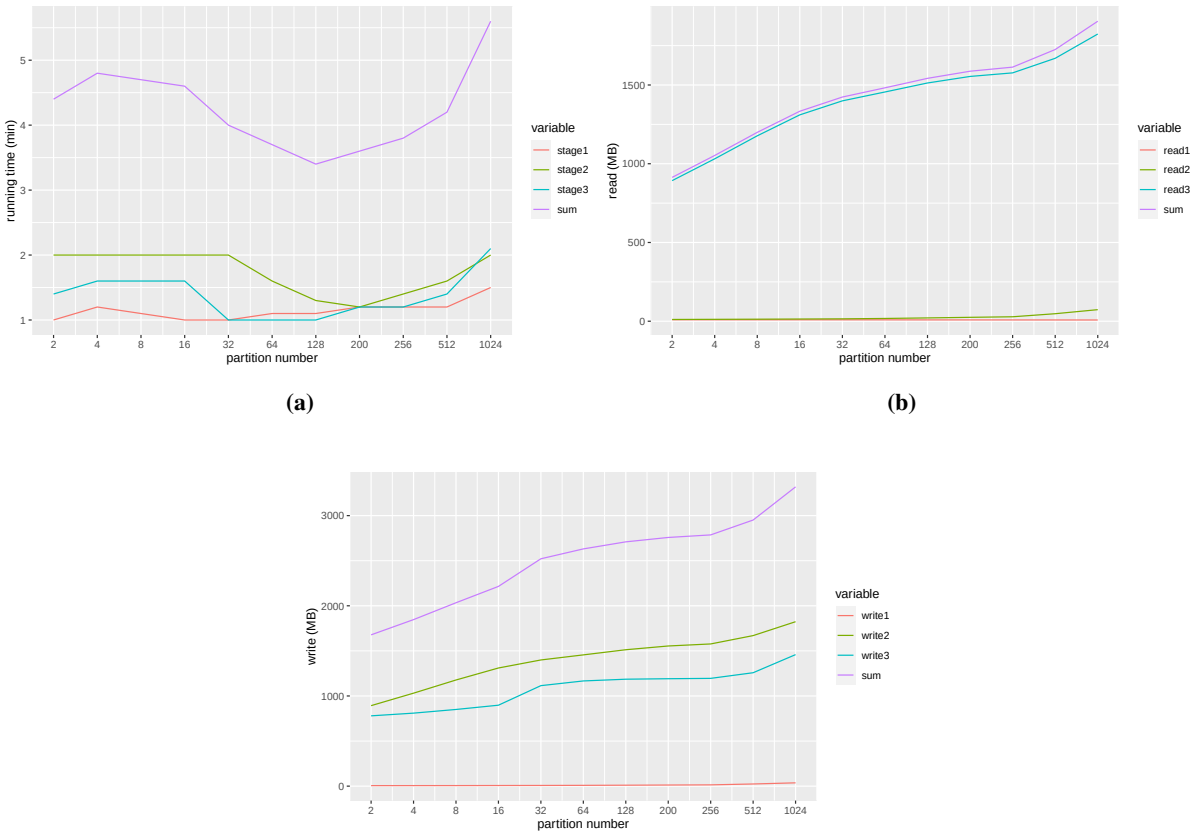
数据集较小时，如果分区数相对较大，就会导致大量调度上的开销；当数据集很大时，如果分区数较小，就不能有效利用集群的计算资源。

根据Xu (2020)，分区数的设置有如下原则：

- 数据量较小，集群规模有限：将分区的数量设置为 core 数量的 1 倍或 2 倍。(每个分区应小于 200MB)。
- 集群的大小有限，但要处理较大数据量：将分区的数量设置为输入数据大小/分区大小（每个分区小于 200MB)。
- 集群很大：将 shuffle 分区的数量设置为 core 数量的 1 倍或 2 倍。

在本任务中，输入的数据为 80MB 左右，在进行`join`操作时，预测数据量在 6400MB 以下，分区数应当调整为 $6400/200 = 32$ 左右。以计算用户相似度为例，我们对不同大小的分区数进行测试。

由于使用了四核的 cpu，因此我们将测试的分区数设置为 4 的倍数。将任务数设置为 2，4，8，16，32，64，128，200，256，512，1000，运行所用时间、读写大小如下图所示，图中只展示了三个耗时较长的阶段及总和：



(a) **(b)** **(c)**

图 1: 设置不同分区数时，计算用户相似度耗时、读写的变化。(a) 运算耗时；(b) 读耗费空间；(c) 写耗费空间。

实测的结果显示，在分区数为 128 时，程序的总运算用时最少。分区数较小时，随着分区数增大，读写花费的空间会更大，程序的运算更快。当分区数过大时，程序会花费很多时间在调度上，运行也会变慢。

在部署大规模集群时，也应该按照数据量和处理器数量合理设置并行度。

5 未来工作

本项目基于 spark 实现了三种协同过滤算法，并讨论了并行度对算法性能的影响。本项目仍然存在许多不足：

- 目前的代码方法依赖于对 `Dataframe` 的 `join` 操作，会消耗大量存储。根据Mishra (2019)，可能的解决方向有：
  1. 数据偏度 (`data skew`)。不同用户听歌数目存在很大差异。
  2. 垃圾回收 (`garbage allocation`)。`join` 过程中会产生大量中间结果。

  在 `spark3.0.0` 版本后，已经实现了对这两点的优化，详见spark::tuning。
- 对算法优劣的评价。目前比较武断地将听歌历史作为正确集，这存在以下问题：
  1. 任务属于隐式反馈问题，听歌历史不代表喜爱。
  2. KKBOX 的推荐算法也基于协同过滤。即使我们推荐的结果与真实听歌历史相似，但与用户真正的喜好可能存在偏差。

参考文献

GRIFFO U, 2021. Apache spark - best practices and tuning[EB/OL]. https://umbertogriffo.gitbook.io/apache-spark-best-practices-and-tuning/parallelism/untitled.

HU Y, KOREN Y, VOLINSKY C, 2008. Collaborative filtering for implicit feedback datasets[C]//2008 Eighth IEEE international conference on data mining. Ieee: 263-272.

MISHRA R, 2019. Why your spark apps are slow or failing, part ii: Data skew and garbage collection[EB/OL]. https://dzone.com/articles/why-your-spark-apps-are-slow-or-failing-part-ii-da.

SU X, KHOSHGOFTAAR T M, 2009. A survey of collaborative filtering techniques[J]. Advances in artificial intelligence, 2009.

XU X, 2020. Databricks spark jobs optimization techniques: Shuffle partition technique (part 1)[EB/OL]. https://nealanalytics.com/blog/databricks-spark-jobs-optimization-techniques-shuffle-partition-technique-part-1/.

A 核心代码

A.1 代码结构

代码文件中，`userBasedCFModel`、`itemBasedCFModel`、`implicitFeedback` 文件夹中分别包含了对基于用户、基于物品、带权重的 ALS 模型的实现。在 `evaluation.py` 和 `evaluation_for_als.py` 中分别对基于物品和用户、带权重的 ALS 的效果进行计算。

A.2 基于用户的协同过滤

A.2.1 计算用户相似度

```
@pandas_udf("user_index int, song_index int, obs float", PandasUDFType.GROUPED_
    MAP)
def L2normalize(pdf):
  obs = pdf.obs
  return pdf.assign(obs=obs / LA.norm(obs))

# caculate L2 norm
df = df.groupby("user_index").apply(L2normalize)

# calculate similarity by matrix multiplication
rate_df = df.alias("df1").\
  join(df.alias("df2"), col("df1.song_index") == col("df2.song_index")).\
  groupby(col("df1.user_index"), col("df2.user_index")).\
  agg(F.sum(col("df1.obs") * col("df2.obs"))).\
  toDF("user1", "user2", "similarity")

windowDept = Window.partitionBy("user1").orderBy(col("similarity").desc())
rate_df = rate_df.withColumn("row", row_number().over(windowDept)).\
  filter(col("row") <= 30)
```

A.2.2 推荐模型

```
class UserBasedCFModel():
  """
  Implementation of user-based Collabrative Filtering Model
  """
  def __init__(self, rate_df, sim_df, song_extra_info):
    """
    Args:
      rate_df: the user listening history dataframe
          sim_df: the similarity dataframe
      song_extra_info: dataframe of song_id and song name
    """
    self.rate_df = rate_df.select(["user_index", "song_index"])
    self.rate_df = self.rate_df.\
      withColumn("user_index", self.rate_df["user_index"].cast(IntegerType())).
          \
      withColumn("song_index", self.rate_df["song_index"].cast(IntegerType()))
    self.sim_df = sim_df
    self.song_extra_info = song_extra_info.select(["song_id", "name"])

  def findKNN(self, user_index, k = 20):
    """
    find k nearest neighbor of user
    [Suspended] In this version, we calculate kNN in advance to save time
```

```
        Args:
            user_index: the index of user we care about
            k: the number of the nearest neighbor
    Returns:
      A dataframe containing kNN
    """
    neighbor_df = self.sim_df.filter(self.sim_df.user1 == user_index).\
        drop("user1")
    neighbor_df = neighbor_df.filter(neighbor_df.user2 != user_index).\
                orderBy(col("similarity").desc()).\
        limit(k)

    return neighbor_df

  def topNRecommend(self, user_index, N = 20, k = 20):
    """
    Args:
      user_index: the user waiting for recommendation
      N: number of sone recommendation
      k: number of neighbor
    Returns:
      A list of index of songs
    """

    # get the user-neigbhor-similarity dataframe
    neighbor_sim_df = self.sim_df.filter(self.sim_df.user1 == user_index)

    # get the neighbor's song from rate_df
    song_sim_df = neighbor_sim_df.join(self.rate_df, neighbor_sim_df.user2 ==
        self.rate_df.user_index)

    # get the song list user listened
    user_listen_df = self.rate_df.filter(self.rate_df.user_index == user_index)

    # exclude the user-listened-song
    song_sim_df = song_sim_df.join(
      user_listen_df,
      song_sim_df.song_index == user_listen_df.song_index,
      "leftanti"
    )

    # get the song-similarity dataframe
    song_sim_df = song_sim_df.\
      groupby("song_index").\
      agg(F.sum("similarity")).\
      toDF("song_index", "sumOfSimilarity").\
      orderBy(col("sumOfSimilarity").desc()).\
```

```
      limit(N)

    return song_sim_df
```

A.3 基于物品的协同过滤

A.3.1 计算物品相似度

```
@pandas_udf("user_index int, song_index int, obs float", PandasUDFType.GROUPED_
    MAP)
def L2normalize(pdf):
  obs = pdf.obs
  return pdf.assign(obs=obs / LA.norm(obs))

# L2 normalize for same song
df = df.groupBy("song_index").apply(L2normalize)

# calculate similarity by matrix multiplication
rate_df = df.alias("df1").\
  join(df.alias("df2"), col("df1.user_index") == col("df2.user_index")).\
  groupBy(col("df1.song_index"), col("df2.song_index")).\
  agg(F.sum(col("df1.obs") * col("df2.obs"))).\
  toDF("song1", "song2", "similarity")


windowDept = Window.partitionBy("song1").orderBy(col("similarity").desc())
rate_df = rate_df.withColumn("row", row_number().over(windowDept)).\
  filter(col("row") <= 20)
```

A.3.2 推荐模型

```
class ItemBasedCFModel():
  """
  Implement the item-based Collabrative Filtering Model
  """

  def __init__(self, rate_df, sim_df, song_extra_info):
    """
    Args:
      rate_df: the user-song-rate dataframe
      sim_df: the song-song-similarity dataframe
      song_extra_info: the id-name dataframe
    """
    self.rate_df = rate_df.select(["user_index", "song_index"])
    self.rate_df = self.rate_df.\
```

```
      withColumn("user_index", self.rate_df["user_index"].cast(IntegerType())).
          \
      withColumn("song_index", self.rate_df["song_index"].cast(IntegerType()))

    self.sim_df = sim_df
    self.song_extra_info = song_extra_info.select(["song_id", "name"])

  def topNRecommend(self, user_index, N = 20):
    # get the user-song df
    user_listen_df = self.rate_df.filter(self.rate_df.user_index == user_index)

    # join with song-neighbor-sim df
    song_sim_df = user_listen_df.join(
      self.sim_df, user_listen_df.song_index == self.sim_df.song1
      ).\
      groupBy("song2").\
      agg(F.sum("similarity")).\
      toDF("song_index", "sumOfSimilarity")

    # delete those songs listened by user
    song_sim_df = song_sim_df.join(
      user_listen_df,\
      song_sim_df.song_index == user_listen_df.song_index,
      "leftanti"
    )

    song_sim_df = song_sim_df.\
      orderBy(col("sumOfSimilarity").desc()).\
      limit(N)

    return song_sim_df
```

A.4 带权重的 ALS 模型

```
class ALSCFModel:
  def __init__(self, obs_df, song_extra_info, spark, retrain = False):
    self.obs_df = obs_df.select(["user_index", "song_index", "obs"])
    self.obs_df = self.obs_df.\
      withColumn("user_index", self.obs_df["user_index"].cast(IntegerType())).\
      withColumn("song_index", self.obs_df["song_index"].cast(IntegerType())).\
      withColumn("obs", self.obs_df["obs"].cast(FloatType()))
    self.song_extra_info = song_extra_info
    self.spark = spark

    if retrain == True:
      als = ALS(maxIter=5, regParam=0.01, implicitPrefs=True,
```

```
      userCol="user_index", itemCol="song_index", ratingCol="obs")
      model = als.fit(obs_df)
      model.save("file:///home/hadoop/Desktop/distributed_system/final_pj/model
          /als")

    self.model = ALSModel.load("file:///home/hadoop/Desktop/distributed_system/
        final_pj/model/als")

  def topNRecommend(self, user_index, N=20):
    user_subset = self.obs_df.where(self.obs_df.user_index == user_index)
    song_sim_df = self.spark.createDataFrame(
      self.model.recommendForUserSubset(user_subset, 20).collect()[0][1])
    return song_sim_df
```

B 系统实现截图

(a) **(b)** **(c)**

图 2: 为用户 42 做推荐。(a) 基于用户，正确率为 0.95 ; (b) 基于物品，正确率为 0; (c) 带权重的 ALS，正确率为 1。