

一、实验过程

1. MapReduce 热门商品统计

热门商品和商家的统计与 Word Count 逻辑非常类似，此处直接使用 Word Count 作业中定义的 IntSumReducer、IntWritableDecreaseComparator、OrderRecordWriter 和 OrderOutputFormat 类来实现 Reduce 和格式化输出，并重新定义 ItemFollowedMapper 和 MerchantFollowedMapper 类来实现对符合要求的商品与商家的过滤。

在 ItemFollowedMapper 类的 map() 函数中，对 time_stamp 和 action_type 做判断，将符合要求的 item_id 发出。

```
1. public void map(Object key, Text value, Context context) throws IOException,
   InterruptedException {
2.     String line = value.toString();
3.     String[] tokens = line.split(",");
4.     if (tokens[0].equals("user_id")) return;
5.     if (!tokens[5].equals("1111")) return;
6.     if (tokens[6].equals("0")) return;
7.     keyIntWritable.set(Integer.parseInt(tokens[1]));
8.     context.write(keyIntWritable, valueIntWritable);
9. }
```

在 MerchantFollowedMapper 类的 setup() 函数中，读取 info 文件，将年龄小于 30 的用户的 user_id 加入集合 ageU30Set。

```
1. public void setup(Context context) throws IOException, InterruptedException
   {
2.     conf = context.getConfiguration();
3.     BufferedReader fis = new BufferedReader(new FileReader("user_info_format
   1.csv"));
4.     String line = null;
5.     while ((line = fis.readLine()) != null) {
6.         String[] tokens = line.split(",");
7.         if (tokens.length < 2) continue;
8.         if (tokens[0].equals("user_id")) continue;
9.         if (!(tokens[1].equals("1") || tokens[1].equals("2") || tokens[1].eq
   uals("3"))) continue;
10.        ageU30Set.add(Integer.parseInt(tokens[0]));
11.    }
12. }
```

在 map() 函数中，对 time_stamp 和 action_type 做判断，并且判断 user_id 是否在 ageU30Set 之中，将符合要求的 merchant_id 发出。

```

1. public void map(Object key, Text value, Context context) throws IOException,
   InterruptedException {
2.     String line = value.toString();
3.     String[] tokens = line.split(",");
4.     if (tokens[0].equals("user_id")) return;
5.     if (!tokens[5].equals("1111")) return;
6.     if (tokens[6].equals("0")) return;
7.     if (!ageU30Set.contains(Integer.parseInt(tokens[0]))) return;
8.     keyIntWritable.set(Integer.parseInt(tokens[3]));
9.     context.write(keyIntWritable, valueIntWritable);
10. }

```

在 main()函数中依次分别执行以 ItemFollowedMapper 和 MerchantFollowedMapper 类为 Mapper、IntSumReducer 类为 Reducer 的 job，再对两者的中间结果分别执行以 InverseMapper 为 Mapper、IntWritableDecreaseComparator 为 SortComparator 的 sortJob。

MapReduce 程序即编写完成。在本地集群中运行结果如下。

```

root@h01: /usr/local/hadoop
root@h01: /usr/local/hadoop# bin/hadoop fs -cat temp/output_0/* | head
1: 191499, 2494
2: 353560, 2250
3: 1059899, 1917
4: 713695, 1754
5: 655904, 1674
6: 67897, 1572
7: 221663, 1547
8: 1039919, 1511
9: 454937, 1387
10: 81360, 1361
root@h01: /usr/local/hadoop# bin/hadoop fs -cat temp/output_0/* | tail
91: 982357, 580
92: 293244, 577
93: 419724, 571
94: 256896, 559
95: 893999, 554
96: 870470, 549
97: 1042707, 549
98: 82431, 548
99: 1093758, 545
100: 1112049, 543
root@h01: /usr/local/hadoop#

```

```

root@h01: /usr/local/hadoop
root@h01: /usr/local/hadoop# bin/hadoop fs -cat temp/output_1/* | head
1: 4044, 7278
2: 3491, 3661
3: 1102, 3588
4: 3828, 3434
5: 4173, 3348
6: 3734, 3303
7: 2385, 3214
8: 4976, 3064
9: 798, 2997
10: 422, 2893
root@h01: /usr/local/hadoop# bin/hadoop fs -cat temp/output_1/* | tail
91: 3163, 885
92: 3623, 884
93: 4287, 875
94: 1867, 868
95: 3173, 867
96: 2318, 865
97: 2677, 865
98: 4427, 864
99: 786, 839
100: 643, 827
root@h01: /usr/local/hadoop#

```

2. Spark 热门商品统计

采用 IntelliJ IDEA、Scala 语言和 sbt 构建工具作为开发环境，采用本地 Docker 集群中的 Spark 2.4.0 作为运行环境。由于 Spark 2.4.0 依赖于 Hadoop 2.7.0，而先前实验中

所用的 Hadoop 版本为 3.2.1, 两者无法兼容, 因此需要重新配置安装集群环境, 步骤较为繁琐, 不在此赘述。

在 IDEA 中, 安装 Scala 插件后即可创建 Scala 工程。在 build.sbt 文件中增加 Spark 相关依赖项即可在源代码中导入、调用 Spark 相关库。

```
1. libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.0"
```

新建 MostPopular 类, 并在其中定义 main() 函数, 作为程序主入口。首先读取 info 文件为 RDD, 通过 map() 分割数据行, 通过 filter() 过滤出年龄小于 30 的用户, 通过 map() 提取 user_id 列, 最后通过 collect().toSet 将 RDD 转换为集合, 赋值给 userU30。

```
1. val infoFile = "file:///usr/local/spark-2.4.0/input/user_info_format1.csv"
2. val conf = new SparkConf().setAppName("Most Popular Item");
3. val sc = new SparkContext(conf)
4. val info = sc.textFile(infoFile).cache()
5. val userU30 = info
6.   .map(line => line.split(","))
7.   .filter(line => line.size >= 2 && !line(0).equals("user_id") && (line(1).equals("1") || line(1).equals("2") || line(1).equals("3")))
8.   .map(line => line(0))
9.   .collect()
10.  .toSet
```

而后读取 input 文件为 RDD, 通过 map() 分割数据行, 通过 filter() 过滤出在双十一有加购物车、购买或收藏操作的用户, 通过 map() 提取 item_id 列, 通过 reduceByKey() 以 item_id 为分组计数, 通过 sortBy() 排序, 最后通过 take() 提取结果的前 100 项, 赋值给 itemTop100。

```
1. val inputFile = "file:///usr/local/spark-2.4.0/input/user_log_format1.csv"
2. val input = sc.textFile(inputFile).cache()
3. val itemTop100 = input
4.   .map(line => line.split(","))
5.   .filter(line => !line(0).equals("user_id") && line(5).equals("1111") && !line(6).equals("0"))
6.   .map(line => (line(1), 1))
7.   .reduceByKey { case (x, y) => x + y }
8.   .sortBy(_._2, false)
9.   .take(100)
```

以类似方式处理得到 merchantTop100, 仅在 filter() 函数的参数中增加对 user_id 是否在 userU30 集合中的判断。

```
1. val merchantTop100 = input
2.   .map(line => line.split(","))
```

```

3. .filter(line => !line(0).equals("user_id") && line(5).equals("1111") && !line(6).equals("0") && userU30.contains(line(0)))
4. .map(line => (line(3), 1))
5. .reduceByKey { case (x, y) => x + y }
6. .sortBy(_._2, false)
7. .take(100)

```

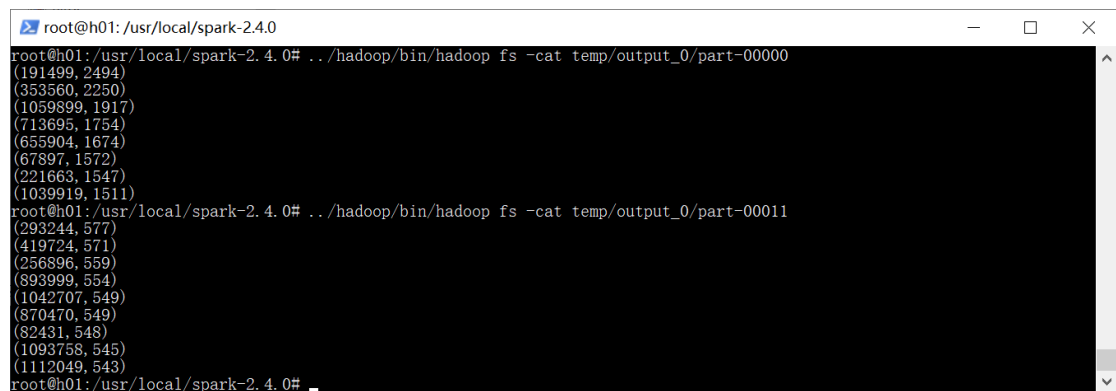
需要注意，经过 take()函数处理后的 RDD 不再是 RDD，而是 Array，需要通过 sc.parallelize()将其重新转化为 RDD 后通过 saveAsTextFile()保存到文件。

```

1. sc.parallelize(itemTop100).saveAsTextFile("temp/output_0")
2. sc.parallelize(merchantTop100).saveAsTextFile("temp/output_1")

```

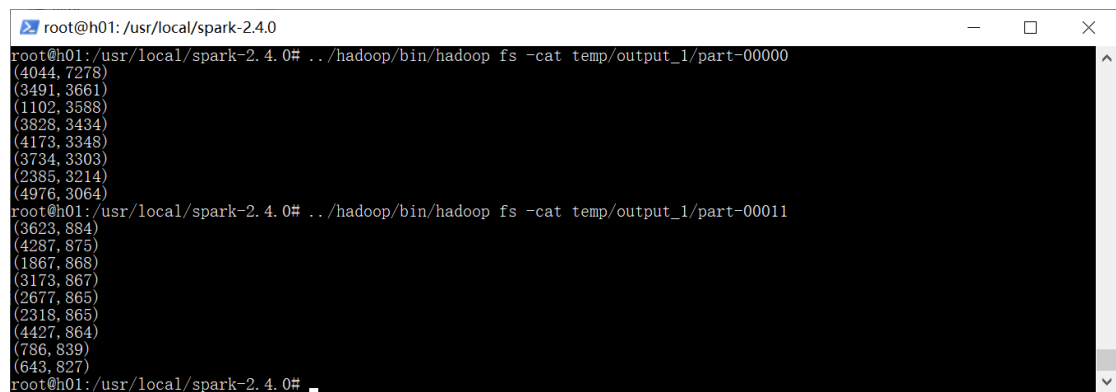
Spark 程序即编写完成。在本地集群中运行结果如下。



```

root@h01: /usr/local/spark-2.4.0
root@h01: /usr/local/spark-2.4.0# ./hadoop/bin/hadoop fs -cat temp/output_0/part-00000
(191499, 2494)
(353560, 2250)
(1059899, 1917)
(713695, 1754)
(655904, 1674)
(67897, 1572)
(221663, 1547)
(1039919, 1511)
root@h01: /usr/local/spark-2.4.0# ./hadoop/bin/hadoop fs -cat temp/output_0/part-00011
(293244, 577)
(419724, 571)
(256896, 559)
(893999, 554)
(1042707, 549)
(870470, 549)
(82431, 548)
(1093758, 545)
(1112049, 543)
root@h01: /usr/local/spark-2.4.0#

```



```

root@h01: /usr/local/spark-2.4.0
root@h01: /usr/local/spark-2.4.0# ./hadoop/bin/hadoop fs -cat temp/output_1/part-00000
(4044, 7278)
(3491, 3661)
(1102, 3588)
(3828, 3434)
(4173, 3348)
(3734, 3303)
(2385, 3214)
(4976, 3064)
root@h01: /usr/local/spark-2.4.0# ./hadoop/bin/hadoop fs -cat temp/output_1/part-00011
(3623, 884)
(4287, 875)
(1867, 868)
(3173, 867)
(2677, 865)
(2318, 865)
(4427, 864)
(786, 839)
(643, 827)
root@h01: /usr/local/spark-2.4.0#

```

3. Spark 用户画像统计

新建 BuyerInfo 类，并在其中定义 main()函数，作为程序主入口。首先读取 input 和 info 文件为 RDD。

```

1. val inputFile = "file:///usr/local/spark-2.4.0/input/user_log_format1.csv"
2. val infoFile = "file:///usr/local/spark-2.4.0/input/user_info_format1.csv"
3. val conf = new SparkConf().setAppName("User Info")
4. val sc = new SparkContext(conf)
5. val input = sc.textFile(inputFile).cache()

```

```
6. val info = sc.textFile(infoFile).cache()
```

对 input 通过 map()分割数据行, 通过 filter()过滤出在双十一有购买操作的用户, 通过 map()提取 user_id 列, 最后通过 collect().toSet 将 RDD 转换为集合并去重, 赋值给 buyer。

```
1. val buyer = input
2.   .map(line => line.split(","))
3.   .filter(line => line(5).equals("1111") && line(6).equals("2"))
4.   .map(line => line(0))
5.   .collect()
6.   .toSet
```

对 info 通过 map()分割数据行, 通过 filter()过滤出在双十一有购买操作的用户, 通过 map()将缺失值设为“未知”, 通过 map()提取 gender 列, 通过 reduceByKey()以 gender 为分组计数, 最后通过 sortBy()排序, 赋值给 gender。

```
1. val gender = info
2.   .map(line => line.split(","))
3.   .filter(line => buyer.contains(line(0)))
4.   .map(line => if (line.size < 3) Array[String](line(0), "0", "2") else line
   )
5.   .map(line => (line(2), 1))
6.   .reduceByKey { case (x, y) => x + y }
7.   .sortBy(x => x._2, false)
```

对 gender 通过 map()提取 count 列, 通过 reduce()求和得到在双十一有购买操作的用户总数, 赋值给 buyerSum。

```
1. val buyerSum = gender
2.   .map(line => line._2)
3.   .reduce(_ + _)
```

对 gender 通过 map()计算分组数量和总数之比, 得到比例, 赋值给 genderProp。

```
1. val genderProp = gender
2.   .map(line => (line._1, line._2.toDouble/buyerSum.toDouble))
```

以类似方式对 info 处理得到年龄段比例, 赋值给 ageProp。

```
1. val age = info
2.   .map(line => line.split(","))
3.   .filter(line => buyer.contains(line(0)))
4.   .map(line => if (line.size < 2 || (line.size >= 2 && line(1).equals("")))
   Array[String](line(0), "0") else line)
```

```

5.  .map(line => (line(1), 1))
6.  .reduceByKey { case (x, y) => x + y }
7.  .sortBy(x => x._2, false)
8.  val ageProp = age
9.  .map(line => (line._1, line._2.toDouble/buyerSum.toDouble))

```

最后保存到文件。

```

1. genderProp.saveAsTextFile("temp/output_0")
2. ageProp.saveAsTextFile("temp/output_1")

```

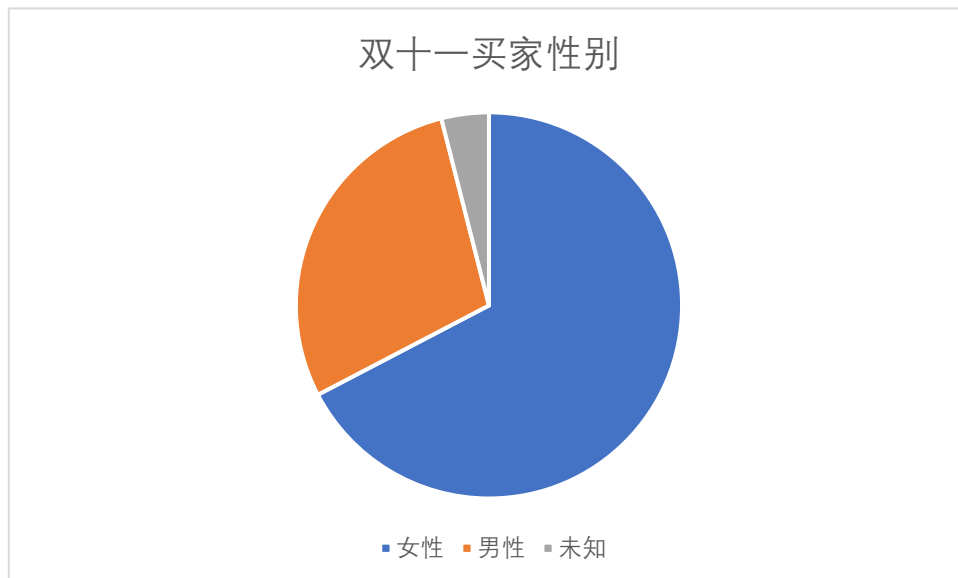
Spark 程序即编写完成。在本地集群中运行结果如下。

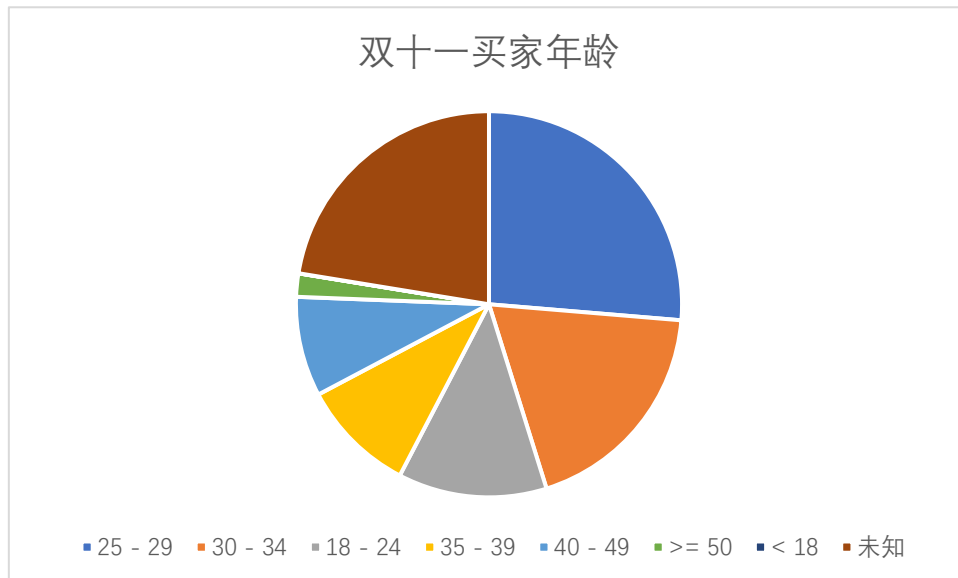
```

root@h01: /usr/local/spark-2.4.0
root@h01:/usr/local/spark-2.4.0# ../hadoop/bin/hadoop fs -cat temp/output_0/*
(0.0.6734045312021124)
(1.0.2868425395478228)
(2.0.03975292925006483)
root@h01:/usr/local/spark-2.4.0# ../hadoop/bin/hadoop fs -cat temp/output_1/*
(3.0.26322936558455334)
(0.0.22427564419925974)
(4.0.18858240799679374)
(2.0.12464577881509772)
(5.0.09613362566895349)
(6.0.08360798736355707)
(7.0.016483956904071482)
(8.0.0029846523799420045)
(1.5.6581087771412404E-5)
root@h01:/usr/local/spark-2.4.0#

```

可视化结果如下。





4. Spark SQL 用户画像统计

在 build.sbt 文件中增加 Spark SQL 所需依赖项。

```
1. libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.4.0"
```

定义 SparkSession。

```
1. val spark = SparkSession
2.   .builder()
3.   .appName("Spark SQL Buyer Info")
4.   .config("spark.some.config.option", "some-value")
5.   .getOrCreate()
6. import spark.implicits._
```

以 csv 格式读取 input 和 info 文件为 DataFrame。

```
1. val inputDF = spark
2.   .read.format("csv")
3.   .option("inferSchema", "true")
4.   .option("header", "true")
5.   .load("file:///usr/local/spark-2.4.0/input/user_log_format1.csv")
6. val infoDF = spark
7.   .read.format("csv")
8.   .option("inferSchema", "true")
9.   .option("header", "true")
10.  .load("file:///usr/local/spark-2.4.0/input/user_info_format1.csv")
```

对 inputDF 通过 filter() 过滤出在双十一有购买操作的用户，通过 select() 提取 user_id 列，通过 distinct() 去重，通过 join() 连接 infoDF 得到带有上述用户 gender 和 age_range

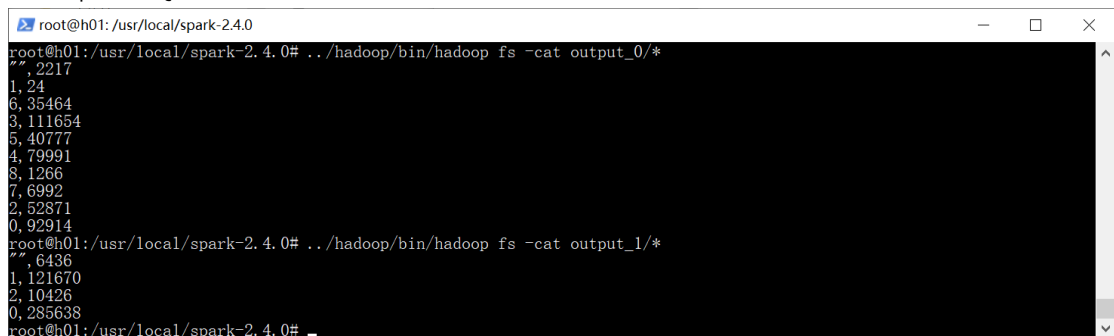
的 DataFrame，赋值给 buyer1111。

```
1. val buyer1111 = inputDF
2.   .filter($"time_stamp" === 1111 && $"action_type" === 2)
3.   .select("user_id")
4.   .distinct()
5.   .join(infoDF, "user_id")
```

对 buyer1111 通过 groupBy().count()按列分组计数，最后通过 write()保存到文件。

```
1. buyer1111
2.   .groupBy("age_range")
3.   .count()
4.   .write
5.   .format("csv")
6.   .mode("overwrite")
7.   .save("output_0")
8. buyer1111
9.   .groupBy("gender")
10.  .count()
11.  .write
12.  .format("csv")
13.  .mode("overwrite")
14.  .save("output_1")
```

Spark SQL 程序即编写完成。在本地集群中运行结果如下。



```
root@h01: /usr/local/spark-2.4.0
root@h01: /usr/local/spark-2.4.0# ../hadoop/bin/hadoop fs -cat output_0/*
,2217
1,24
6,35464
3,111654
5,40777
4,79991
8,1266
7,6992
2,52871
0,92914
root@h01: /usr/local/spark-2.4.0# ../hadoop/bin/hadoop fs -cat output_1/*
,6436
1,121670
2,10426
0,285638
root@h01: /usr/local/spark-2.4.0#
```

此处得到的结果为不同性别、年龄段买家的计数，简单处理即可得到对应的比例，结果与 Spark 程序结果相同，不再展示可视化结果。

本地集群运行时，发现尽管使用 DataFrame 的 Spark SQL 代码更加简洁，但运行时间远远长于使用 RDD 的 Spark 程序。猜测可能是因为 Spark SQL 中 join 操作的复杂度较高。

5. Spark ML 复购预测

首先进行特征选取。

通过观察数据集和用户行为日志，发现在训练集和测试集中存在的用户是且仅是在

双十一当天首次购买过某商家商品的用户，其在双十一之前可以点击、加购物车或者收藏该商家的商品，但不能没有买过该商家的商品，或在双十一之前买过该商家的商品。对于符合上述要求的用户，其双十一之前六个月的所有行为都被收录在用户行为日志中（包括其在其他商家的行为），但仅有符合上述要求的用户-商家对才存在于训练集和测试集中，即预测只需考虑存在新客关系的用户-商家对，但符合要求的用户在其他商家的行为也被提供以供参考。

基于以上发现，可见用户行为日志中收录的行为对于用户而言是完整的，是该用户在双十一及之前六个月的全部行为记录，但对于商家或商品是不完整的，仅包含与符合上述要求的用户相关的记录。

因此基于待预测的用户-商家对的特征和基于用户的特征有相对完整的语义，而一些基于时间或基于商家的特征可能没有相对完整的语义。

基于以上考虑，共选取 36 个特征如下。

用户在该商家的点击/加购/购买/收藏次数
用户在该商家的点击/加购/购买/收藏商品数量
用户在该商家的点击/加购/购买/收藏品类数量
用户在该商家的点击/加购/购买/收藏次数最多的品类
用户在该商家的点击/加购/收藏天数
用户在该商家的点击/加购/购买/收藏次数
用户的总点击/加购/购买/收藏商品数量
用户的总点击/加购/购买/收藏品类数量
用户的总点击/加购/购买/收藏次数最多的品类
用户的总点击/加购/收藏天数
用户的性别/年龄

接着通过 Spark SQL 提取特征。

先以 csv 格式将相关文件读取为 DataFrame。

```
1. val log = spark
2.   .read.format("csv")
3.   .option("inferSchema", "true")
4.   .option("header", "true")
5.   .load("file:///usr/local/spark-2.4.0/input/user_log_format1.csv")
```

对于次数、数量等特征，对用户行为日志 log 通过 filter() 过滤出符合要求的条目，而后通过 groupBy() 分组，再通过 count() 或 agg(countDistinct()) 计数即可。

```
1. val merchantViewCount = log
2.   .filter($"action_type" === 0)
3.   .groupBy("user_id", "merchant_id")
4.   .count()
5.   .withColumnRenamed("count", "merchantViewCount")
```

```

6.
7. val merchantViewItemNum = log
8.   .filter($"action_type" === 0)
9.   .groupBy("user_id", "merchant_id")
10.  .agg(countDistinct("item_id"))
11.  .withColumnRenamed("count(DISTINCT item_id)", "merchantViewItemNum")

```

对于出现次数最多的品类这一特征，首先通过 filter() 过滤出符合要求的条目，通过 groupBy() 分组，注意需要将 cat_id 加入分组的 key 中，再通过 count() 计数得到用户-商家-品类组的计数信息，存为 DataFrame。

而后通过 groupBy() 对上述 DataFrame 重新按用户-商家对分组，通过 agg(max()) 得到该用户-商家对中品类的最高出现次数，存为 DataFrame。

再将上述两个 DataFrame 按用户-商家对连接在一起，得到一个既有每个品类出现次数，又有品类最高出现次数的 DataFrame，注意应使用左连接。

最后对上述 DataFrame 通过 filter() 过滤出品类出现次数等于最高出现次数的条目，即出现最多次的品类所在条目，通过 select() 去除无用的次数列，通过 dropDuplicates() 去除可能存在的用户-商家对的重复行（如果有多个品类出现次数相同且都为最高）。

出现次数最多的品类即提取完成。

```

1. val merchantViewCatCount = log
2.   .filter($"action_type" === 0)
3.   .groupBy("user_id", "merchant_id", "cat_id")
4.   .count()
5.
6. val merchantViewCatMost = merchantViewCatCount
7.   .join(merchantViewCatCount
8.     .groupBy("user_id", "merchant_id")
9.     .agg(max("count").alias("maxCount")),
10.    Seq[String]("user_id", "merchant_id"), "left")
11.  .filter($"count" === $"maxCount")
12.  .select("user_id", "merchant_id", "cat_id")
13.  .dropDuplicates("user_id", "merchant_id")
14.  .withColumnRenamed("cat_id", "merchantViewCatMost")

```

将以上方法提取得到的特征全部通过左连接与训练集 train 相连，再与用户个人画像 info 相连，即得到了完整的特征。

由于上述任务过大，将其编程为一个 object 来运行会使 Spark 崩溃，因此我将其分为四个 object 分别处理全部特征中的四部分，最后再通过一个单独的 object 将四个部分的特征连接在一起，得到完整的向量化训练集。

最后通过 Spark ML 训练和预测。

先以 csv 格式将向量化训练集读取为 DataFrame，而后通过 VectorAssembler() 将 36 个特征列整合为一个列，接着通过 randomSplit() 划分训练集和测试集。

```

1. val featuresTrain = spark
2.   .read.format("csv")
3.   .option("inferSchema", "true")
4.   .option("header", "true")
5.   .load("file:///usr/local/spark-2.4.0/features.csv")
6.
7. val assembler = new VectorAssembler()
8.   .setInputCols(Array(
9.     "age_range", "gender", "merchantViewCount", "merchantAddCount", "merchan
    tBuyCount", "merchantLikeCount",
10.    "merchantViewItemNum", "merchantAddItemNum", "merchantBuyItemNum", "merc
    hantLikeItemNum", "merchantViewCatNum",
11.    "merchantAddCatNum", "merchantBuyCatNum", "merchantLikeCatNum", "merchan
    tViewCatMost", "merchantAddCatMost",
12.    "merchantBuyCatMost", "merchantLikeCatMost", "merchantActionDays", "user
    ViewCount", "userAddCount",
13.    "userBuyCount", "userLikeCount", "userViewItemNum", "userAddItemNum", "u
    serBuyItemNum", "userLikeItemNum",
14.    "userViewCatNum", "userAddCatNum", "userBuyCatNum", "userLikeCatNum", "u
    serViewCatMost", "userAddCatMost",
15.    "userBuyCatMost", "userLikeCatMost", "userActionDays"))
16.   .setOutputCol("features")
17.
18. val assemblerTrain = assembler.transform(featuresTrain)
19. val Array(train, test) = assemblerTrain.randomSplit(Array(0.8, 0.2))

```

再定义随机森林模型，训练并预测，最后通过 BinaryClassificationEvaluator()计算测试集上的 auc 值，以供评估学习效果。

```

1. val rfc = new RandomForestClassifier()
2.   .setFeaturesCol("features")
3.   .setLabelCol("label")
4.
5. val rfcModel = rfc.fit(train)
6. val trainPredict = rfcModel.transform(train)
7. val testPredict = rfcModel.transform(test)
8. val auc = new BinaryClassificationEvaluator()
9.   .setMetricName("areaUnderROC")
10.  .evaluate(testPredict)

```

程序在 Spark 中运行结果如下。

```
root@h01: /usr/local/spark-2.4.0
2021-01-11 14:56:50 INFO TaskSetManager:54 - Finished task 5.0 in stage 29.0 (TID 155) in 16 ms on localhost (executor driver) (2/8)
2021-01-11 14:56:50 INFO Executor:54 - Finished task 4.0 in stage 29.0 (TID 154). 834 bytes result sent to driver
2021-01-11 14:56:50 INFO Executor:54 - Finished task 3.0 in stage 29.0 (TID 153). 834 bytes result sent to driver
2021-01-11 14:56:50 INFO TaskSetManager:54 - Finished task 7.0 in stage 29.0 (TID 157) in 16 ms on localhost (executor driver) (3/8)
2021-01-11 14:56:50 INFO Executor:54 - Finished task 1.0 in stage 29.0 (TID 151). 877 bytes result sent to driver
2021-01-11 14:56:50 INFO TaskSetManager:54 - Finished task 3.0 in stage 29.0 (TID 153) in 17 ms on localhost (executor driver) (4/8)
2021-01-11 14:56:50 INFO Executor:54 - Finished task 6.0 in stage 29.0 (TID 156). 834 bytes result sent to driver
2021-01-11 14:56:50 INFO Executor:54 - Finished task 2.0 in stage 29.0 (TID 152). 834 bytes result sent to driver
2021-01-11 14:56:50 INFO TaskSetManager:54 - Finished task 4.0 in stage 29.0 (TID 154) in 18 ms on localhost (executor driver) (5/8)
2021-01-11 14:56:50 INFO TaskSetManager:54 - Finished task 1.0 in stage 29.0 (TID 151) in 20 ms on localhost (executor driver) (6/8)
2021-01-11 14:56:50 INFO TaskSetManager:54 - Finished task 6.0 in stage 29.0 (TID 156) in 17 ms on localhost (executor driver) (7/8)
2021-01-11 14:56:50 INFO TaskSetManager:54 - Finished task 2.0 in stage 29.0 (TID 152) in 18 ms on localhost (executor driver) (8/8)
2021-01-11 14:56:50 INFO TaskSchedulerImpl:54 - Removed TaskSet 29.0, whose tasks have all completed, from pool
2021-01-11 14:56:50 INFO DAGScheduler:54 - ResultStage 29 (aggregate at AreaUnderCurve.scala:45) finished in 0.024 s
2021-01-11 14:56:50 INFO DAGScheduler:54 - Job 15 finished: aggregate at AreaUnderCurve.scala:45, took 0.026563 s
2021-01-11 14:56:50 INFO MapPartitionsRDD:54 - Removing RDD 59 from persistence list
2021-01-11 14:56:50 INFO BlockManager:54 - Removing RDD 59
----- TEST AUC: 0.6245682095920998 -----
2021-01-11 14:56:50 INFO SparkContext:54 - Invoking stop() from shutdown hook
2021-01-11 14:56:50 INFO AbstractConnector:318 - Stopped Spark@29d2d081[HTTP/1.1, [http/1.1]] {0.0.0.0:4040}
2021-01-11 14:56:50 INFO SparkUI:54 - Stopped Spark web UI at http://h01:4040
2021-01-11 14:56:50 INFO MapOutputTrackerMasterEndpoint:54 - MapOutputTrackerMasterEndpoint stopped!
2021-01-11 14:56:50 INFO MemoryStore:54 - MemoryStore cleared
2021-01-11 14:56:50 INFO BlockManager:54 - BlockManager stopped
2021-01-11 14:56:50 INFO BlockManagerMaster:54 - BlockManagerMaster stopped
2021-01-11 14:56:50 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint:54 - OutputCommitCoordinator stopped!
2021-01-11 14:56:50 INFO SparkContext:54 - Successfully stopped SparkContext
2021-01-11 14:56:50 INFO ShutdownHookManager:54 - Shutdown hook called
2021-01-11 14:56:50 INFO ShutdownHookManager:54 - Deleting directory /tmp/spark-973f376a-cd98-4e96-8369-b26e2a20d6cc
2021-01-11 14:56:50 INFO ShutdownHookManager:54 - Deleting directory /tmp/spark-c85d76fc-5c46-4bc2-84ef-c603d0226f30
root@h01: /usr/local/spark-2.4.0#
```

可见程序功能正常，测试集上的 AUC 为 0.625。

二、问题总结、解决方案及其他思考

实验过程中的所有问题及其解决方案已在实验过程中叙述，此处不再重复。

本次实验首先需要安装配置 Spark 环境，有了 Hadoop 安装配置的经验后，该项任务不算困难，但仍会花去很多时间。

由于 Spark 推荐使用 Scala 语言编程，且听说 Scala 语言程序较为简洁，本次实验中我采用了 Scala 语言。Scala 语言需要一定时间上手，但在习惯其匿名表达式的用法、且理解 RDD 或 DataFrame 的转化过程后，可发现 Scala 语言确实非常高效，编写程序的过程类似于构造 Pipeline 的过程，非常流畅。

在机器学习任务中，特征工程的复杂性和重要性远比想象中要高，花费的时间和精力也是最多的。由于提取的特征数量多，Spark 程序往往会在运行过程中崩溃，而一次运行又需要耗费非常多时间，因此程序调试起来比较困难。对于一个任务能否在某一资源条件下的 Spark 上顺利运行似乎并没有提前知晓的方法，只能一次次尝试直到其能够正常运行。