

金融大数据第四次实验

目录

1.环境准备	2
1.1 安装 Spark3.5.3	2
1.2 安装 Scala	3
1.3 安装 Py4J	3
1.4 配置环境变量	4
2.任务 1: Spark RDD 编程	4
2.1 作业一	4
2.1.1 任务描述	4
2.1.2 实现思路	5
2.1.3 遇到的问题及解决方案	6
2.1.4 运行结果部分截图	6
2.2 作业二	6
2.2.1 任务描述	7
2.2.2 实现思路	7
2.2.3 运行部分结果截图	8
3.任务 2: Spark SQL 编程	8
3.1 作业一	8
3.1.1 任务描述	8
3.1.2 实现思路	8
3.1.3 实现结果截图	9
3.2 作业二	9
3.2.1 任务描述	9
3.2.2 实现思路	9
3.2.3 实现结果截图	11
4.任务 3: Spark ML 编程	11
4.1 任务描述	11
4.2 实现思路	11
4.3 遇到的问题及解决方案	14
4.4 实现结果	14


在线文档:

https://hy5kne0un7.feishu.cn/docx/NG7tdqtC4oOf7zxtXVucOhYhnhh?from=from_copylink

1.环境准备

1.1 安装 Spark3.5.3

在官方下载地址：[Download Apache Spark™](#)，选择安装 Spark3.5.3。

Download Libraries Documentation Examples Community Developers GitHub

Download Apache Spark™

1. Choose a Spark release:
2. Choose a package type:
3. Download Spark: [spark-3.5.3-bin-without-hadoop.tgz](#)
4. Verify this release using the 3.5.3 [signatures](#), [checksums](#) and [project release KEYS](#) by following these [procedures](#).

Note that Spark 3 is pre-built with Scala 2.12 in general and Spark 3.2+ provides additional pre-built distribution with Scala 2.13.

Link with Spark

Spark artifacts are [hosted in Maven Central](#). You can add a Maven dependency with the following coordinates:

```
groupId: org.apache.spark
artifactId: spark-core_2.12
version: 3.5.3
```

通过下面的命令将下载的压缩包解压缩，并且存储到/usr/local/spark 目录下：

SQL

```
sudo tar -zxf ~/Downloads/spark-3.5.3-bin-without-hadoop.tgz -C /usr/local/
cd /usr/local
sudo mv ./spark-3.5.3-bin-without-hadoop/ ./spark
sudo chown -R zhangke ./spark
```

安装成功后，修改 Spark 的配置文件 spark-env.sh，在第一行添加以下配置信息：

bash

```
export SPARK_DIST_CLASSPATH=$(/usr/local/hadoop/bin/hadoop classpath)
```

```
export SPARK_DIST_CLASSPATH=$(/usr/local/hadoop/bin/hadoop classpath)
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements. See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License. You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the license for the specific language governing permissions and
# limitations under the License.
#
# This file is sourced when running various Spark programs.
# Copy it as spark-env.sh and edit that to configure Spark for your site.
#
# Options read when launching programs locally with
# ./bin/run-example or ./bin/spark-submit
# - HADOOP_CONF_DIR, to point Spark towards Hadoop configuration files
# - SPARK_LOCAL_IP, to set the IP address Spark binds to on this node
# - SPARK_PUBLIC_DNS, to set the public dns name of the driver program
#
# Options read by executors and drivers running inside the cluster
# - SPARK_LOCAL_IP, to set the IP address Spark binds to on this node
# - SPARK_PUBLIC_DNS, to set the public dns name of the driver program
# - SPARK_LOCAL_DIRS, storage directories to use on this node for shuffle and RDD data
# - MESOS_NATIVE_JAVA_LIBRARY, to point to your libmesos.so if you use Mesos
```

配置完成后就可以直接使用，不需要像 Hadoop 运行启动命令。并且可以通过运行 Spark 自带的示例，验证 Spark 安装成功。

```
zhangke@zhangke-VMware-Virtual-Platform: /usr/local/spark$ bin/run-example SparkPi
Pi is roughly 3.139435697178486
zhangke@zhangke-VMware-Virtual-Platform: /usr/local/spark$
```

1.2 安装 Scala

通过以下命令进行 Scala 的安装即可。

```
Bash
wget https://downloads.lightbend.com/scala/2.12.8/scala-2.12.8.deb
sudo dpkg -i scala-2.12.8.deb
```

最后输入 `scala -version` 以验证 Scala 是否安装成功，见下图：

```
zhangke@zhangke-VMware-Virtual-Platform:~$ wget https://downloads.lightbend.com/scala/2.12.8/scala-2.12.8.deb
--2024-12-09 15:49:49-- https://downloads.lightbend.com/scala/2.12.8/scala-2.12.8.deb
Resolving downloads.lightbend.com (downloads.lightbend.com)... 3.166.244.40, 3.166.244.45, 3.166.244.109, ...
Connecting to downloads.lightbend.com (downloads.lightbend.com)|3.166.244.40|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 151418626 (144M) [application/octet-stream]
Saving to: 'scala-2.12.8.deb'

scala-2.12.8.deb                               100%[=====>] 144.40M  12.0MB/s   in 13s

2024-12-09 15:50:03 (11.1 MB/s) - 'scala-2.12.8.deb' saved [151418626/151418626]

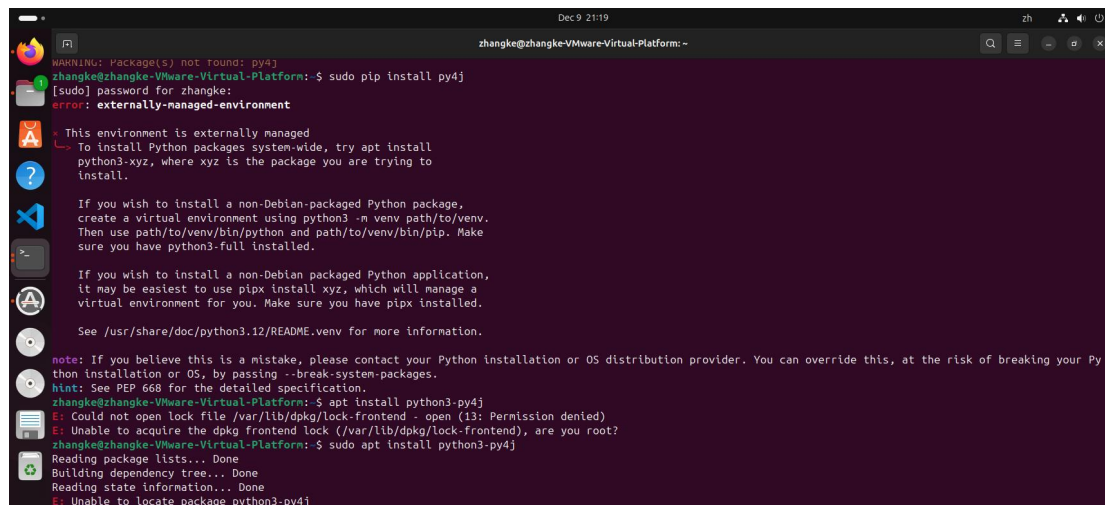
zhangke@zhangke-VMware-Virtual-Platform:~$ sudo dpkg -i scala-2.12.8.deb
[sudo] password for zhangke:
Sorry, try again.
[sudo] password for zhangke:
Selecting previously unselected package scala.
(Reading database ... 198106 files and directories currently installed.)
Preparing to unpack scala-2.12.8.deb ...
Unpacking scala (2.12.8-4b0) ...
Setting up scala (2.12.8-4b0) ...
Creating system group: scala
Creating system user: scala with scala daemon-user and shell /bin/false
Processing triggers for man-db (2.12.0-4build2) ...
zhangke@zhangke-VMware-Virtual-Platform:~$ scala -version
Scala code runner version 2.12.8 -- Copyright 2002-2018, LAMP/EPFL and Lightbend, Inc.
zhangke@zhangke-VMware-Virtual-Platform:~$
```

1.3 安装 Py4J

Py4J 在驱动程序上用于 Python 和 Java SparkContext 对象之间的本地通信，其安装命令如下：

```
Bash
sudo pip install py4j
```

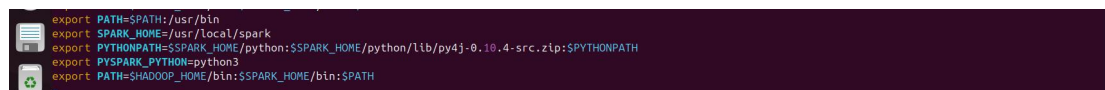
但是使用 pip 出现报错，解决方案，尝试的第一个是先运行 `sudo apt install python3-pip` 再安装 py4j，但是还是出现下面报错

A terminal window showing the process of installing py4j. It starts with a warning that the package is not found, followed by an attempt to install it using pip. This results in an error: 'externally-managed-environment'. The terminal then displays a message explaining that the environment is externally managed and suggests using apt to install python3-xyz. It also provides instructions for installing non-Debian packaged Python packages using venv or pipx. Finally, it shows an attempt to install python3-py4j using apt, which fails with the error: 'Unable to locate package python3-py4j'.

究其原因是因为系统使用了受管理的 Python 环境，禁止通过 pip 直接安装全局包。所以我选择在虚拟环境中安装 py4j 从而避免冲突，具体步骤主要有两步：一是创建虚拟环境并激活；二是通过 `pip install py4j` 安装 py4j。

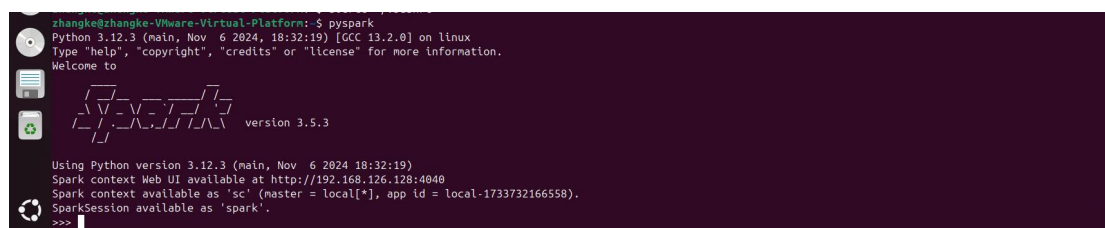
1.4 配置环境变量

在通过 `vim ~/.bashrc` 命令在环境变量中加入下面的内容：

A terminal window showing the configuration of environment variables in the ~/.bashrc file. The variables being set are: export PATH=\$PATH:/usr/bin, export SPARK_HOME=/usr/local/spark, export PYTHONPATH=\$SPARK_HOME/python/lib/py4j-0.10.4-src.zip:\$PYTHONPATH, export PYSARK_PYTHON=python3, and export PATH=\$HADOOP_HOME/bin:\$SPARK_HOME/bin:\$PATH.

然后运行 `source ~/.bashrc` 是环境变量生效。

最后在终端直接输入 pyspark 就可以启动 pyspark。

A terminal window showing the output of the pyspark command. It displays the Python version (3.12.3), the Spark version (3.5.3), and the Spark context available as 'sc'. It also shows the SparkSession available as 'spark'.

2.任务 1： Spark RDD 编程

2.1 作业一

2.1.1 任务描述

查询特定日期的资金流入和流出情况： 使用 `user_balance_table`，计算出所有用

户在每一

天的总资金流入和总资金流出量。

2.1.2 实现思路

首先因为我已经将文件上传到 hdfs 上，所以可以使用 `sc.textFile()` 方法读取 CSV 文件，得到一个包含每一行数据的 RDD，具体代码如下：

```
Python
data_file = "/input/user_balance_table.csv"
rdd = sc.textFile(data_file)
```

然后，因为我们的任务是计算出所有用户在每一天的总资金流入和流出量，所以所需字段为日期、购买金额、赎回金额三个。因此通过解析每一行数据，根据相应的索引获取对应的值，其中数值如果遇到空字段，就使用默认值 0。具体代码如下：

```
Python
def parse_line(line):
    fields = line.split(",") #CSV 以逗号作为间隔符
    try:
        report_date = fields[1]
        total_purchase_amt = int(fields[4]) if fields[4] else 0
        #total_purchase_amt 的索引是 4
        total_redeem_amt = int(fields[8]) if fields[8] else 0
        #total_redeem_amt 的索引是 8
        return (report_date, total_purchase_amt, total_redeem_amt)
    #最后以元组的形式返回
    except IndexError:
        return None
```

除此之外，为了避免数据异常影响程序运行，我对一些特殊情况进行了处理：

```
Python
# 解析数据，去掉表头
header = rdd.first()
rdd = rdd.filter(lambda line: line != header)

# 解析数据并过滤掉无效行
parsed_rdd = rdd.map(parse_line).filter(lambda x: x is not None)
```

获取完目标数据以后，根据任务要求，我将数据按日期进行聚合，相同日期的资金流入和流出分别加总来得到所需的每一天资金总流入和总流出。主要是思路如下：首先使用 `map()` 将数据转化为一个键值对形式（日期，（购买金额，赎回金额））。然

后使用 `reduceByKey()` 对每个日期的购买金额和赎回金额进行聚合。`reduceByKey` 会将相同日期的数据合并在一起，累加购买和赎回金额。

Python

```
aggregated_rdd = parsed_rdd.map(lambda x: (x[0], (x[1],  
x[2]))).reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
```

最后就是格式化输出结果，并将其保存到文件中。

Python

```
result_rdd = sorted_rdd.map(lambda x: f"{x[0]} {x[1][0]}  
{x[1][1]}")  
  
result_rdd.saveAsTextFile("fund_flow")
```

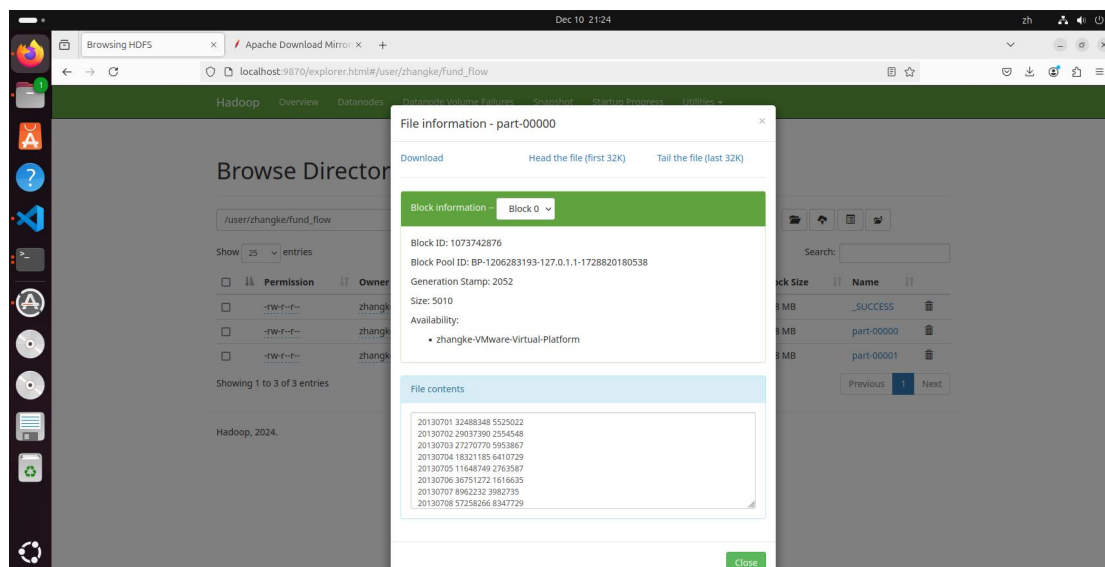
2.1.3 遇到的问题及解决方案

第一次运行代码得到的输出是乱序的，我发现这与 RDD 的性质有关。正是因为 RDD 更注重灵活性和性能，所以它不会像 MapReduce 一样自动排序。但我为了结果的正确性更好验证，采取 `sortByKey()` 对数据进行全局排序：

Python

```
sorted_rdd = aggregated_rdd.sortByKey()
```

2.1.4 运行结果部分截图



2.2 作业二

2.2.1 任务描述

使用 `user_balance_table`，定义活跃用户为在指定月份内有至少 5 天记录的用户，统计 2014 年 8 月的活跃用户总数。

2.2.2 实现思路

和作业一相同还是从 hdfs 上读取 `user_balance_table.csv`，然后将其加载为一个 RDD，以及解析每行数据和作业一也一致。然后我们需要筛选出 2014 年 8 月的数据，定义 `filter_august_data(record)` 函数用于检查日期是否属于 2014 年 8 月（即日期以 201408 开头），再通过 `filter()` 操作来筛选出符合条件的数据记录，返回新的 RDD `august_data`，使该 RDD 只包含 2014 年 8 月的用户数据。

```
Python
def filter_august_data(record):
    report_date = record[1]
    return report_date.startswith("201408")

august_data = parsed_data.filter(filter_august_data)
```

接着就是计算每个用户在 2014 年 8 月的活跃天数，根据用户 ID 聚合，利用 `count_active_days()` 函数将每个用户的日期列表转换成一个集合 (set)，从而去重，得到用户在 2014 年 8 月活跃的独立天数，最后让 `map(count_active_days)` 返回 RDD 为 `(user_id, active_days_count)`：

```
Python
def count_active_days(records):
    user_id = records[0]
    dates = records[1] # 获取该用户的所有日期列表
    unique_days = set(dates) # 使用 set 去重日期
    return (user_id, len(unique_days)) # 返回用户 ID 和去重后的活跃天数

user_activity = august_data.groupByKey().mapValues(list) #
groupByKey 聚合相同用户的所有日期
user_active_days = user_activity.map(count_active_days) # 计算每个用户的活跃天数
```

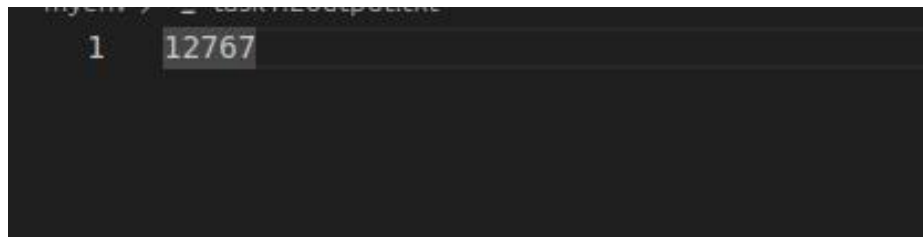
然后，通过 `filter()` 函数筛选出活跃天数大于或等于 5 的用户：

```
Python
active_users = user_active_days.filter(lambda x: x[1] >= 5)
```


最后，只要利用 `count()` 返回 `active_users` RDD 中的元素个数，也就是 2014 年 8 月活跃用户的总数即可。

```
Python
active_users_count = active_users.count()
```

2.2.3 运行部分结果截图



3.任务 2： Spark SQL 编程

3.1 作业一

3.1.1 任务描述

计算每个城市在 2014 年 3 月 1 日的用户平均余额 (`tBalance`)，按平均余额降序排列。

3.1.2 实现思路

文件读取以及数据加载和任务一是一致的，所以在此就不加赘述了。由于任务是要求计算每个城市在 20140301 的用户平均余额，所以第一步就是从 `user_balance_table.csv` 中筛选出所有用户在 2014 年 3 月 1 日的余额，其他对应列不改变。筛选出来的结果存储到 `filtered_balance_df` 中：

```
Python
filtered_balance_df = user_balance_df.filter(col("report_date") == 20140301)
```

其次因为是每个城市的平均余额，所以毫无疑问我们需要按照城市分组，这一步通过 `groupby` 实现。然后在计算每组的平均余额，计算平均数可以直接使用 Spark SQL 中的 `avg` 函数，主要代码如下：

```
Python
city_avg_balance_df =
```



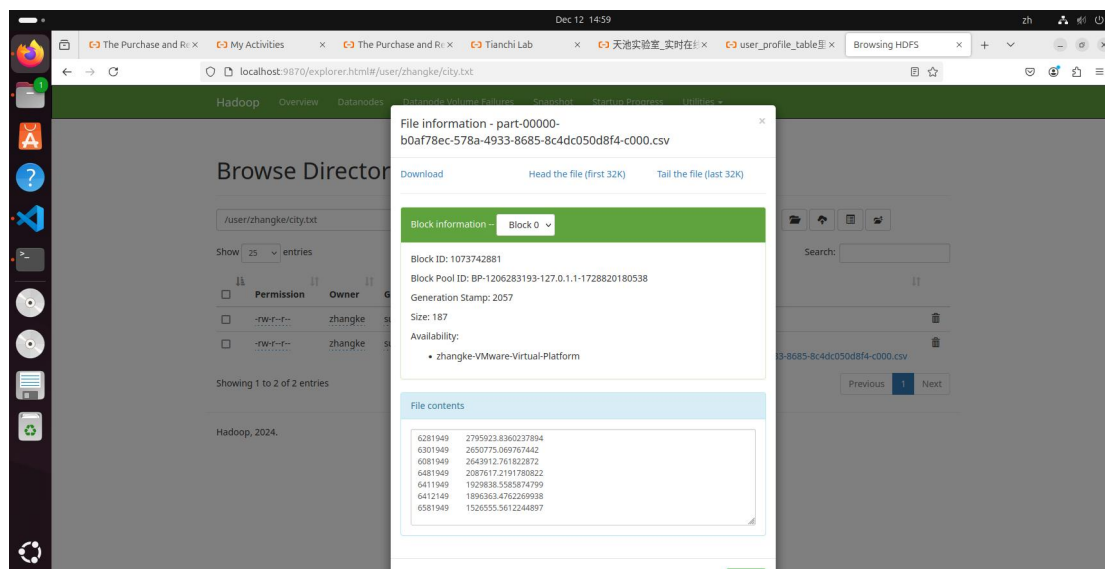
```
joined_df.groupBy("city").agg(avg("tBalance").alias("avg_balance"))
)
```

到此为止计算部分基本上就已经完成了，接下来我们将结果按每个城市的平均余额降序排列，并且根据规定格式保存到输出文件中（输出文件未保留表头）：

Python

```
sorted_city_avg_balance_df =
city_avg_balance_df.orderBy(col("avg_balance").desc())
# 将结果写入到文本文件，使用 tab 作为分隔符
sorted_city_avg_balance_df.write.option("header",
"false").option("delimiter", "\t").csv("city.txt")
```

3.1.3 实现结果截图



3.2 作业二

3.2.1 任务描述

统计每个城市中每个用户在 2014 年 8 月的总流量（定义为 total_purchase_amt + total_redeem_amt），并输出每个城市总流量排名前三的用户 ID 及其总流量。

3.2.2 实现思路

首先筛选出每个用户在 2014 年 8 月的数据，和实验三一样还是采用正则表达式匹配：

Python

```
user_balance_df_filtered =  
user_balance_df.filter(col('report_date').rlike('^201408[0-9]{2}$'))
```

然后计算每个用户每天的总流量，在原来的 **Balance** 表中加入一列来保存总流量，即申购+赎回：

Python

```
user_balance_df_filtered = user_balance_df_filtered.withColumn(  
    'total_flow',  
    col('total_purchase_amt').cast("int") +  
    col('total_redeem_amt').cast("int")  
)
```

紧接着就是计算每个用户在 2014 年 8 月一整个月的总流量，将表格中的 **total_flow** 通过 **sum** 加总即可：

Python

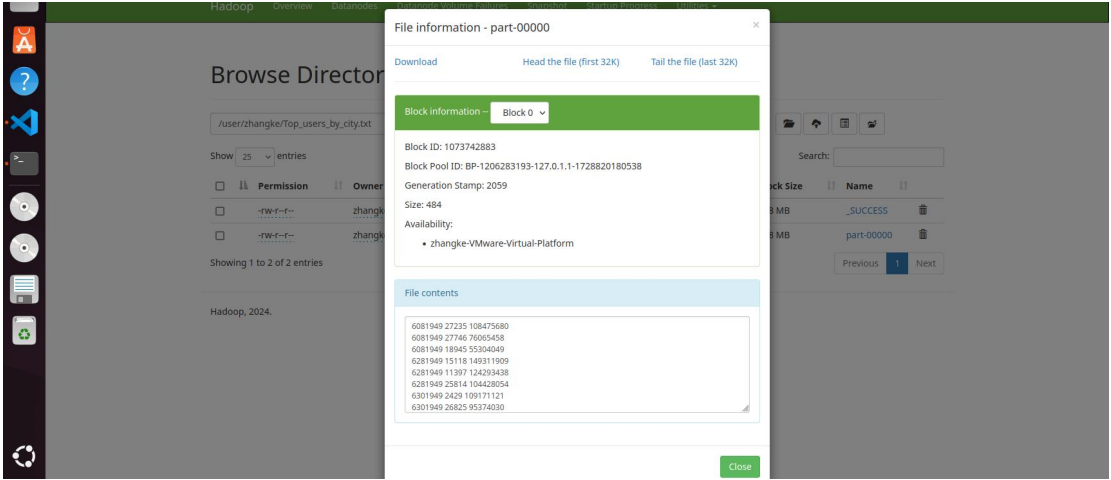
```
user_total_flow_df =  
user_balance_df_filtered.groupBy('user_id').agg(  
    sum('total_flow').alias('total_flow')  
)
```

然后将每个用户与其所在城市进行一个匹配，因为城市信息在 **profile** 表格中，所以需要 **Balance** 和 **profile** 进行合并。合并后，按照城市分组，给各个城市的每个用户总流量进行排序。主要思路就是，先按城市分组，组内根据总流量大小降序排列，然后增加 **rank** 列，作为该用户总流量在其城市的排名，最后只要输出 **rank**≤3 的用户数据：

Python

```
user_data_df = user_total_flow_df.join(user_profile_df,  
on='user_id', how='inner')  
# 排名每个城市中的用户总流量  
window_spec =  
Window.partitionBy('city').orderBy(col('total_flow').desc())  
user_data_ranked_df = user_data_df.withColumn('rank',  
rank().over(window_spec))  
  
# 选择排名前三的用户  
top_users_df = user_data_ranked_df.filter(col('rank') <= 3)
```

3.2.3 实现结果截图



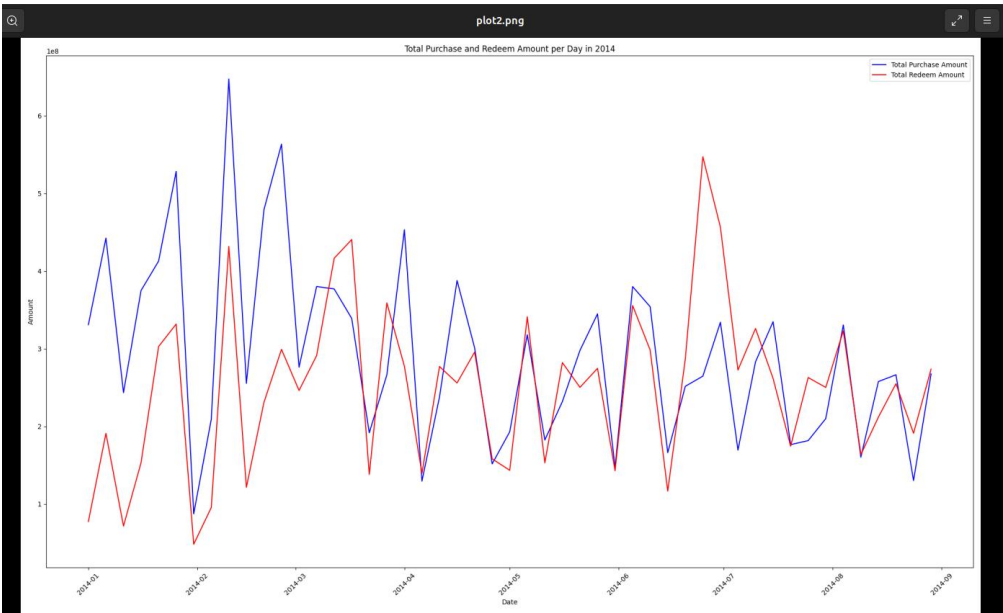
4.任务 3： Spark ML 编程

4.1 任务描述

根据已给的数据，使用 Spark MLlib 提供的机器学习模型，预测 2014 年 9 月每天的申购与赎回总额。

4.2 实现思路

为了避免数据的滞后性，同时也为了减少数据集，我决定只选取 2014 年的数据，为了进一步观察这些数据的可用性，将 2014 年的申购和赎回的数据绘制成折线图（见下图），从下图中观察到 1 月末到二月初出现急剧下降，所以最后决定选择 2014 年 2 月 5 日以后的数据进行预测。



选择使用随机森林回归进行预测，选取的数据范围是 2014-02-05 至 2014-08-31，计算出每天所有用户的总申购和总赎回（这也就是后续模型中所要用到的全部数据了）：

```
Python
df_filtered = df.filter((col("report_date") >= '2014-02-05') &
    (col("report_date") <= '2014-08-31'))
# 按日期进行聚合，计算每日的总申购和总赎回
daily_totals = df_filtered.groupBy("report_date").agg(
    _sum("total_purchase_amt").alias("total_purchase_amt"),
    _sum("total_redeem_amt").alias("total_redeem_amt")
)
```

接下来就是为随机森林做准备，先在数据中加入一些特征，主要是与日期有关的，毕竟我们是需要按时间进行预测，所以加入日期的年份、月份、星期、节假日作为其特征，利用 `VectorAssembler` 将多个这五个输入特征列合并成一个单一的向量列，作为随机森林回归的输入：

```
Python
# 设定节假日列表，并将日期格式转换为字符串
holidays = ['2014-02-14', '2014-04-05', '2014-05-01', '2014-06-02',
    '2014-08-02', '2014-09-01', '2014-09-08']
holidays = [pd.to_datetime(date).strftime('%Y-%m-%d') for date in
    holidays] # 转换为字符串列表

daily_totals_pd['year'] = daily_totals_pd.index.year
daily_totals_pd['month'] = daily_totals_pd.index.month
daily_totals_pd['day'] = daily_totals_pd.index.day
daily_totals_pd['day_of_week'] = daily_totals_pd.index.dayofweek
daily_totals_pd['is_holiday'] =
    daily_totals_pd.index.isin(holidays).astype(int)

# 特征列
feature_cols = ['year', 'month', 'day', 'day_of_week',
    'is_holiday']
assembler = VectorAssembler(inputCols=feature_cols,
    outputCol="features")
```

然后利用 `RandomForestRegressor` 初始化两个随机森林回归模型，一个用来预测申购量一个用来预测赎回，经过不断地测试，最后选定随机种子为 42，其预测结果是相对较好的。

```
Python
```

```
purchase_rf = RandomForestRegressor(featuresCol="features",
labelCol="total_purchase_amt",seed=42)
redeem_rf = RandomForestRegressor(featuresCol="features",
labelCol="total_redeem_amt",seed=42)
```

初始化完模型后，就可以开始训练模型了。通过构建两个机器学习的流水线，为了简化模型的训练和预测过程，我们利用 `pipeline` 将特征汇聚器和模型估算器组合在一起，随后通过 `fit()` 函数将数据集传入 `pipeline` 中进行模型训练：

Python

```
# 构建 Pipeline
pipeline_purchase = Pipeline(stages=[assembler, purchase_rf])
pipeline_redeem = Pipeline(stages=[assembler, redeem_rf])

# 训练模型
model_purchase = pipeline_purchase.fit(spark_df)
model_redeem = pipeline_redeem.fit(spark_df)
```

训练好模型后，我们需要生成预测日期的特征，方法和前面生成数据集中日期特征一致，在此就不再赘述了。随后将预测日期的相关数据传入前面已经训练好的模型从而进行预测，最后只需从预测结果中提取所需信息，按照规定格式写入输出表格即可。

Python

```
# 使用模型进行预测
purchase_predictions = model_purchase.transform(forecast_spark_df)
redeem_predictions = model_redeem.transform(forecast_spark_df)

# 提取预测结果
purchase_forecast = purchase_predictions.select("forecast_date",
"prediction").withColumnRenamed("prediction",
"predicted_purchase_amt")
redeem_forecast = redeem_predictions.select("forecast_date",
"prediction").withColumnRenamed("prediction",
"predicted_redeem_amt")
```

除此之外，通过观察市场中的交易情况可知，节假日的申购量和赎回量会偏高，所以对于节假日的申购和赎回我们另乘 1.2，代码中主要是利用了 `isin` 函数判断当前日期是节假日。

Python

```
forecast_result = forecast_result.withColumn(
    "predicted_purchase_amt",
```

```

        (col("predicted_purchase_amt") * (1 +
        (col("forecast_date").cast("string").isin(holidays).cast("int") *
        0.2))).cast("int")
    )

forecast_result = forecast_result.withColumn(
    "predicted_redeem_amt",
    (col("predicted_redeem_amt") * (1 +
    (col("forecast_date").cast("string").isin(holidays).cast("int") *
    0.2))).cast("int")
)

```

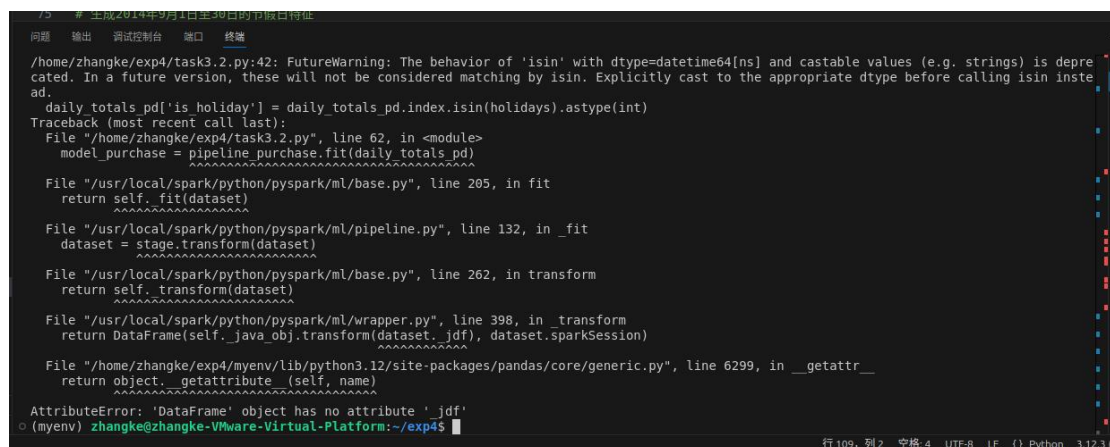
4.3 遇到的问题及解决方案

在一次运行过程中，出现如下图的报错。我根据报错的位置找到出现该报错的原因是，fit 函数一般接受的输入需要是 Spark DataFrame（也就是 PySpark 中的数据结构），而不能是简单的 Pandas DataFrame，因为 fit 函数处理过程中涉及到的一些属性 Pandas DataFrame 中不存在，所以会出现报错。解决方法很简单，只需将 Pandas DataFrame 转换成 Spark DataFrame 即可，于是我在代码中加入了以下语句：

```

Python
spark_df = spark.createDataFrame(daily_totals_pd)

```

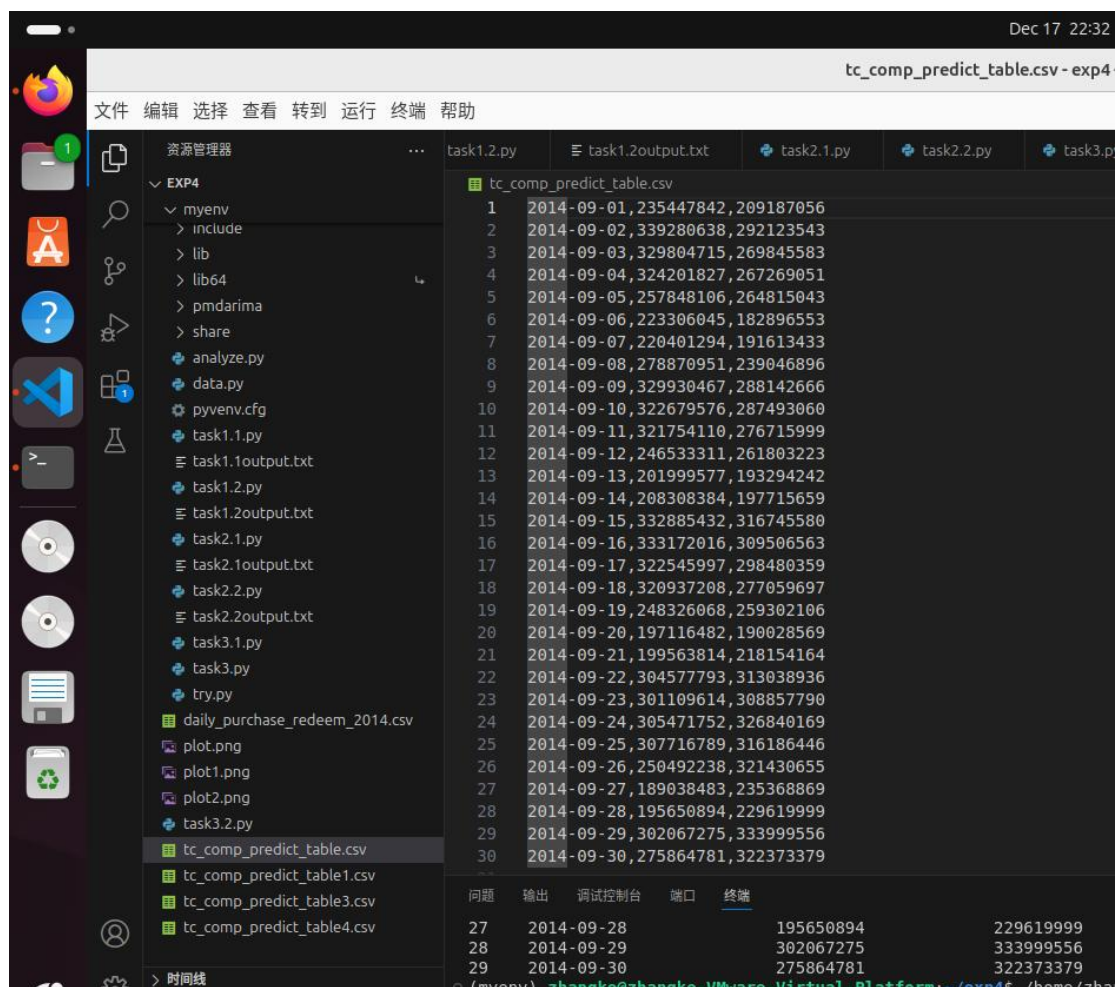


```

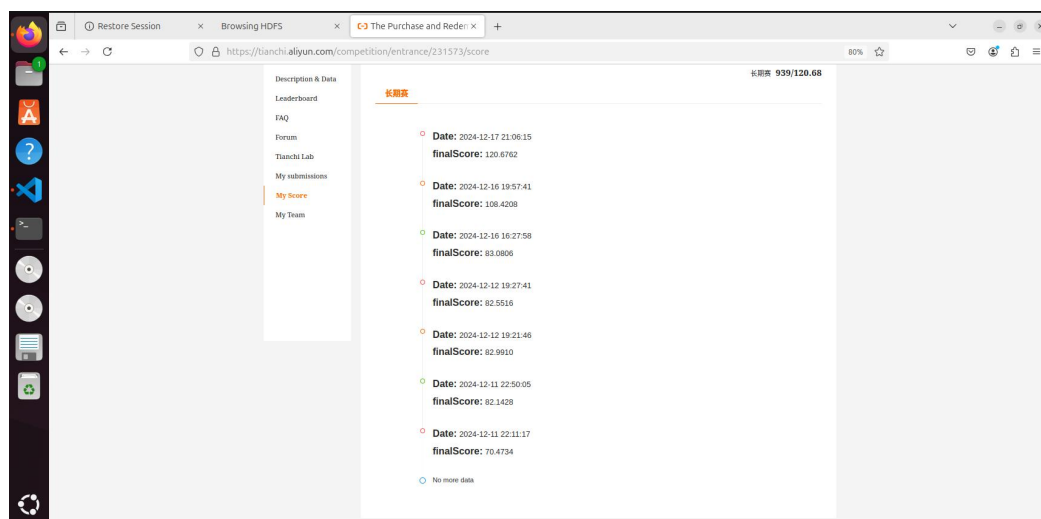
75 # 生成2014年9月1日至30日的节假日特征
问题 输出 调试控制台 端口 终端
/home/zhangke/exp4/task3.2.py:42: FutureWarning: The behavior of 'isin' with dtype=datetime64[ns] and castable values (e.g. strings) is deprecated. In a future version, these will not be considered matching by isin. Explicitly cast to the appropriate dtype before calling isin instead.
daily_totals_pd['is_holiday'] = daily_totals_pd.index.isin(holidays).astype(int)
Traceback (most recent call last):
  File "/home/zhangke/exp4/task3.2.py", line 62, in <module>
    model_purchase = pipeline_purchase.fit(daily_totals_pd)
    ~~~~~
  File "/usr/local/spark/python/pyspark/ml/base.py", line 205, in fit
    return self._fit(dataset)
    ~~~~~
  File "/usr/local/spark/python/pyspark/ml/pipeline.py", line 132, in _fit
    dataset = stage.transform(dataset)
    ~~~~~
  File "/usr/local/spark/python/pyspark/ml/base.py", line 262, in transform
    return self._transform(dataset)
    ~~~~~
  File "/usr/local/spark/python/pyspark/ml/wrapper.py", line 398, in _transform
    return DataFrame(self._java_obj.transform(dataset._jdf), dataset.sparkSession)
    ~~~~~
  File "/home/zhangke/exp4/myenv/lib/python3.12/site-packages/pandas/core/generic.py", line 6299, in __getattr__
    return object.__getattr__(self, name)
    ~~~~~
AttributeError: 'DataFrame' object has no attribute 'jdf'
(myenv) zhangke@zhangke-VMware-Virtual-Platform:~/exp4$

```

4.4 实现结果



预测结果



得分截图

经不断测试发现，随机森林回归预测的效果要优于 ARIMA 模型（图片中一开始的几个分数均是采用 ARIMA 模型的结果）。究其原因可能是，ARIMA 主要考虑的是时间序列中的直接滞后关系（AR）和滑动平均项（MA），对季节性、趋势等依赖较强的复杂时序关系建模能力有限。而随机森林能够通过树的结构捕捉到多层次、多维度

的依赖关系，尤其适合于那些时间序列数据中存在多重时序效应或复杂模式的情况。这使得它能够更好地应对具有较强季节性或趋势性的申购和赎回量数据。