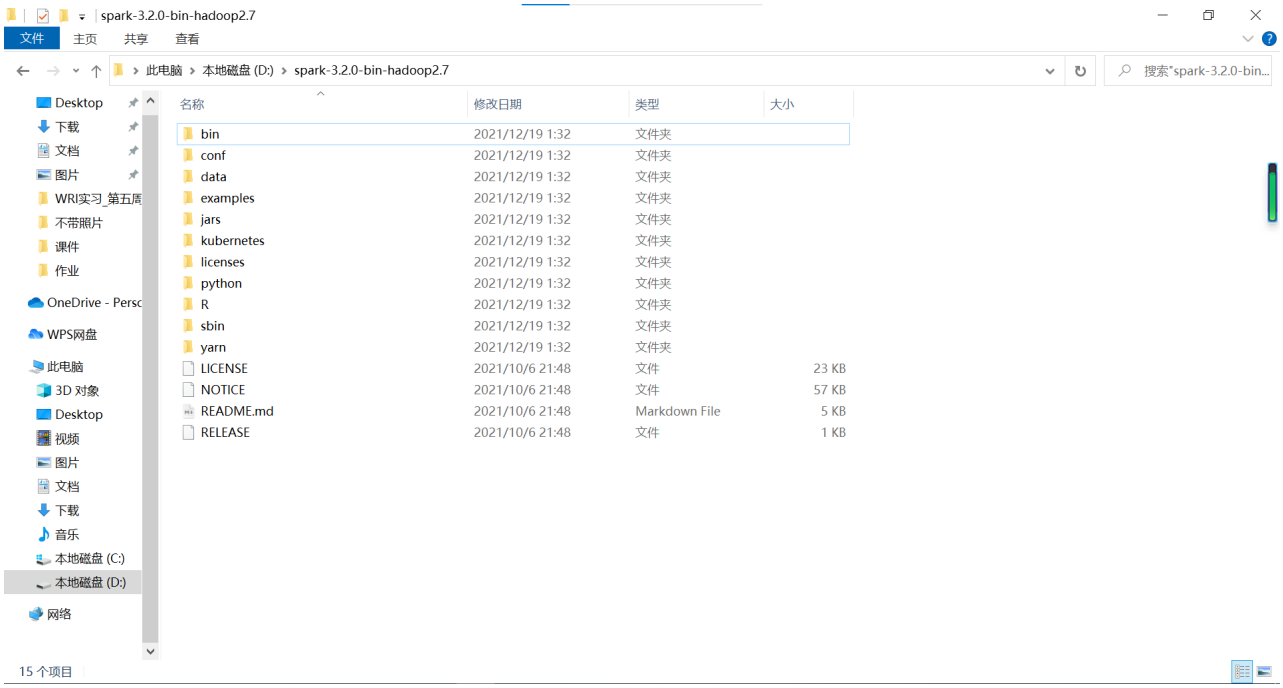
191098323 张峻珲 实验四

一. 配置环境

1. jdk（已有）

```
C:\Users\DELL>java -version
java version "1.8.0_152"
Java(TM) SE Runtime Environment (build 1.8.0_152-b16)
Java HotSpot(TM) 64-Bit Server VM (build 25.152-b16, mixed mode)
```

2. spark



但是测试时显示不成功，后查找发现需要安装winutils.exe修改权限

```
C:\WINDOWS\system32>cd C:\winutils\bin

C:\winutils\bin>winutils.exe chmod 777 \tmp\hive
ChangeFileModeByMask error (2): ???????????


C:\winutils\bin>winutils.exe chmod 777 \tmp\hive

C:\winutils\bin>
```

成功了

```
C:\Users\DELL>spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
21/12/19 01:53:05 ERROR SparkContext: Error initializing SparkContext.
java.lang.reflect.InvocationTargetException
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
        at java.lang.reflect.Constructor.newInstance(Constructor.java:423)
        at org.apache.spark.executor.Executor.addReplClassLoaderIfNeeded(Executor.scala:909)
        at org.apache.spark.executor.Executor.<init>(Executor.scala:160)
        at org.apache.spark.scheduler.local.LocalEndpoint.<init>(LocalSchedulerBackend.scala:64)
        at org.apache.spark.scheduler.local.LocalSchedulerBackend.start(LocalSchedulerBackend.scala:132)
        at org.apache.spark.scheduler.TaskSchedulerImpl.start(TaskSchedulerImpl.scala:220)
        at org.apache.spark.SparkContext.<init>(SparkContext.scala:581)
        at org.apache.spark.SparkContext$.getOrCreate(SparkContext.scala:2690)
        at org.apache.spark.sql.SparkSession$Builder.$anonfun$getOrCreate$2(SparkSession.scala:949)
        at scala.Option.getOrElse(Option.scala:189)
        at org.apache.spark.sql.SparkSession$Builder.getOrCreate(SparkSession.scala:943)
        at org.apache.spark.repl.Main$.createSparkSession(Main.scala:106)
        at $line3.$read$$iw$$iw.<init>(<console>:15)
        at $line3.$read$$iw.<init>(<console>:42)
        at $line3.$read.<init>(<console>:44)
        at $line3.$read$.<init>(<console>:48)
        at $line3.$read$.<clinit>(<console>)
```

3. anaconda（已有）

4. pyspark

   将Spark\python路径下的pyspark文件夹复制粘贴到安装的Anaconda3下的lib下的site-packages下

```
C:\WINDOWS\system32\cmd.exe - pyspark
Microsoft Windows [版本 10.0.19042.1415]
(c) Microsoft Corporation。保留所有权利。

C:\Users\DELL>pyspark
Python 3.8.10 | packaged by conda-forge | (default, May 11 2021, 06:25:23) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
C:\Users\DELL\anaconda3\lib\site-packages\numpy\__init__.py:143: UserWarning: mkl-service package failed to import, ther
efore Intel(R) MKL initialization ensuring its correct out-of-the box operation under condition when Gnu OpenMP had alre
ady been loaded by Python process is not assured. Please install mkl-service package, see http://github.com/IntelPython/
mkl-service
  from . import _distributor_init
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.2.0
      /_/

Using Python version 3.8.10 (default, May 11 2021 06:25:23)
Spark context Web UI available at http://DESKTOP-KGT58BA:4040
Spark context available as 'sc' (master = local[*], app id = local-1639855202619).
SparkSession available as 'spark'.
>>> 21/12/19 03:20:12 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of Proc
essTree metrics is stopped
```

5. 尝试运行

   发现报错，安装py4j包

```
sc = SparkContext("local", "count app")
words = sc.parallelize(
    ["scala",
     "java",

~\anaconda3\lib\site-packages\pyspark\__init__.py in <module>
     51
     52 from pyspark.conf import SparkConf
---> 53 from pyspark.rdd import RDD, RDDBarrier
     54 from pyspark.files import SparkFiles
     55 from pyspark.status import StatusTracker, SparkJobInfo, SparkStageInfo

~\anaconda3\lib\site-packages\pyspark\rdd.py in <module>
     32 from math import sqrt, log, isinf, isnan, pow, ceil
     33
---> 34 from pyspark.java_gateway import local_connect_and_auth
     35 from pyspark.serializers import AutoBatchedSerializer, BatchedSerializer, NoOpSerializer, \
     36     CartesianDeserializer, CloudPickleSerializer, PairDeserializer, PickleSerializer, \

~\anaconda3\lib\site-packages\pyspark\java_gateway.py in <module>
     27 from subprocess import Popen, PIPE
     28
---> 29 from py4j.java_gateway import java_import, JavaGateway, JavaObject, GatewayParameters
     30 from py4j.clientserver import ClientServer, JavaParameters, PythonParameters
     31 from pyspark.find_spark_home import _find_spark_home

ModuleNotFoundError: No module named 'py4j'
```

成功

```
In [1]: from pyspark import SparkContext
        sc = SparkContext("local", "count app")

In [ ]:
```

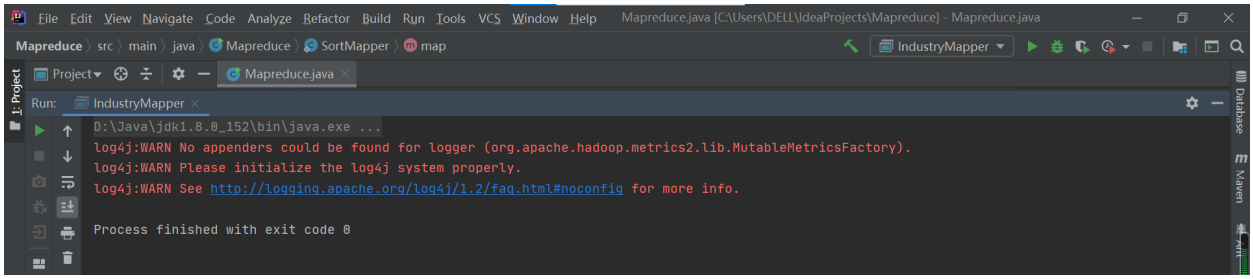
二. 编程

任务1

1. 思路
   - 先提取出被统计对象

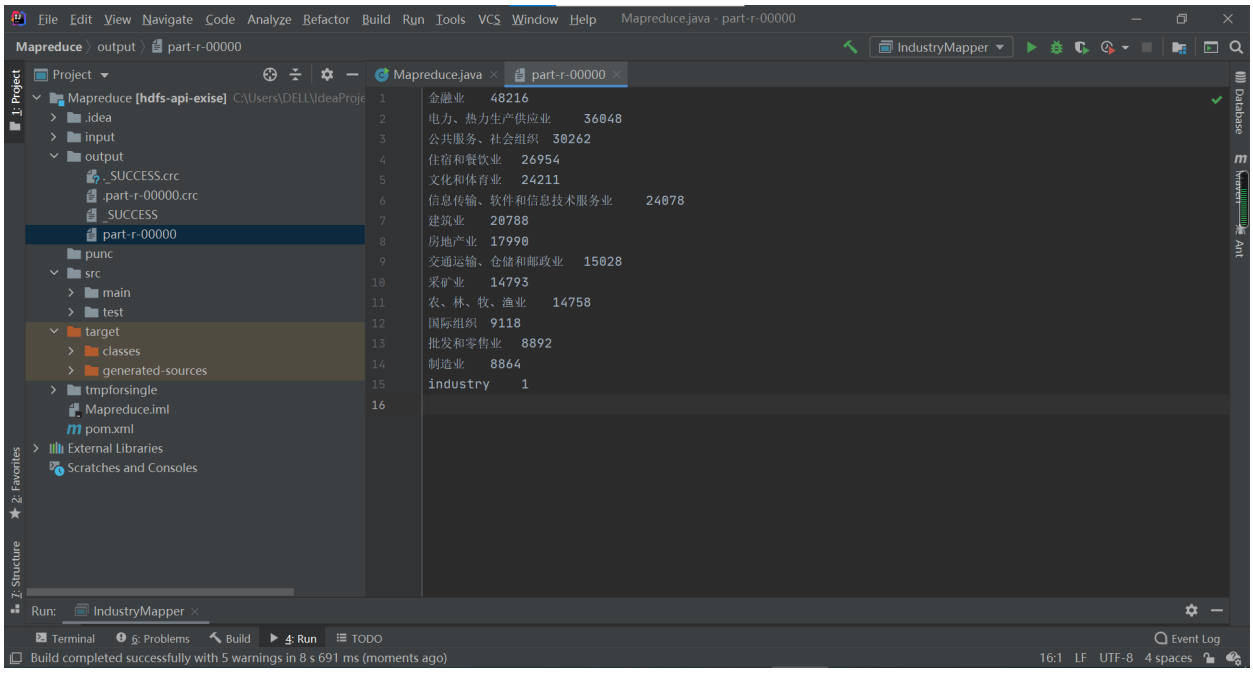
```
private static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>
{
    private final static IntWritable one = new IntWritable( value: 1);
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException
    {
        String line[] = value.toString().split( regex: ",");
        // 提取出 Industry 列
        context.write(new Text(line[10]), one);
    }
}
```

   - 第一个job（word count）进行词频统计，结果存入中间过程
   - 以中间过程为输入，第二个job（sort）进行降序排序
2. 运行结果
   - 运行成功



- 运行中间过程（词频统计结果）

- 最终结果（排序结果）



任务2

1. 导入和初始化
2. 读取数据

   按照教程使用load函数，但是报错，貌似废除了这种操作了

```
In [3]: # 读取数据
        df = sqlContext.load(source="train_data .csv", header="true")
```

```
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_13488/930479792.py in <module>
      1 # 读取数据
----> 2 df = sqlContext.load(source="train_data .csv", header="true")

AttributeError: 'SQLContext' object has no attribute 'load'
```

改用read方法

```
In [6]: # 读取数据
        df = sqlContext.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load("train_data .csv")
```

```
In [ ]:
```

3. 统计total_loan列的情况，确定最大的区间应该是(40000, 41000)

```
In [22]: # 统计total_loan列的情况
         df.select(df.total_loan).describe().show()
```

```
+-------+------------------+
|summary|        total_loan|
+-------+------------------+
|  count|            300000|
|   mean|       14399.20875|
| stddev|8709.179953286359|
|    min|             500.0|
|    max|           40000.0|
+-------+------------------+
```

4. 使用filter函数进行统计，但是filter函数只能给出一个条件（尝试了几次复杂的条件都在报错），所以先filter得到大于等于的部分、再filter得到小于的部分，再取交集，最后统计。

```
In [63]: # 确定了最大区间为(40000, 41000)后，进行统计
         for i in range(0, 41):
             low = i * 1000
             high = low + 1000
             print('(' + str(low) + ', ' + str(high) + '), ' + str(df.filter(df.total_loan >= low).intersect(df.filter(df.total_loan < high)).count()))
```

```
(0, 1000), 2
(1000, 2000), 4043
(2000, 3000), 6341
(3000, 4000), 9317
(4000, 5000), 10071
(5000, 6000), 16514
(6000, 7000), 15961
(7000, 8000), 12789
(8000, 9000), 16384
(9000, 10000), 10458
(10000, 11000), 27170
(11000, 12000), 7472
(12000, 13000), 20513
(13000, 14000), 5928
(14000, 15000), 8888
(15000, 16000), 18612
(16000, 17000), 11277
(17000, 18000), 4388
(18000, 19000), 9342
(19000, 20000), 4077
(20000, 21000), 17612
(21000, 22000), 5507
```

不过这样做的效率过低，稍微改进一下思路，只统计大于等于的部分的数量，存入一个list，然后用前面的减去后面的的差值，就是这个区间的数量（这么做效率可以提高一倍，不过写起来就没那么好看了）

In [64]:
```
# 另一种统计
num = []
for i in range(0, 41):
    low = i * 1000
    num_tem = df.filter(df.total_loan >= low).count()
    num.append(num_tem)
for i in range(0, 40):
    low = i * 1000
    high = low + 1000
    print('(' + str(low) + ', ' + str(high) + '), ' + str(num[i] - num[i + 1]))
print('(' + '40000' + ', ' + '41000' + '), ' + str(num[40]))
```

```
(0, 1000), 2
(1000, 2000), 4043
(2000, 3000), 6341
(3000, 4000), 9317
(4000, 5000), 10071
(5000, 6000), 16514
(6000, 7000), 15961
(7000, 8000), 12789
(8000, 9000), 16384
(9000, 10000), 10458
(10000, 11000), 27170
(11000, 12000), 7472
(12000, 13000), 20513
(13000, 14000), 5928
(14000, 15000), 8888
(15000, 16000), 18612
(16000, 17000), 11277
```

任务3

1. 公司类型

- 先统计有哪些公司，转化为一个list

In [10]:
```
# 获取公司类型有哪些
firms = df.select(df.employer_type).distinct()
firms.show()
```

```
+--------------+
| employer_type|
+--------------+
|幼教与中小学校|
|      上市企业|
|      政府机构|
|    世界五百强|
|  高等教育机构|
|      普通企业|
+--------------+
```

In [11]:
```
# 将其转化为Pandas，由于一共就几个，这一段可以用Pandas来做
firms = firms.toPandas()
firms.iloc[0, 0]
```

Out[11]: '幼教与中小学校'

In [12]:
```
# 将Pandas存入list，每一项为string
firms_list = []
for i in range(len(firms)):
    firms_list.append(firms.iloc[i, 0])
firms_list
```

Out[12]: ['幼教与中小学校', '上市企业', '政府机构', '世界五百强', '高等教育机构', '普通企业']

- 使用filter进行统计

In [18]:
```
# 统计数量
num = []
for item in firms_list:
    num_temp = df.filter(df.employer_type == item).count()
    num.append(num_temp)
num
```

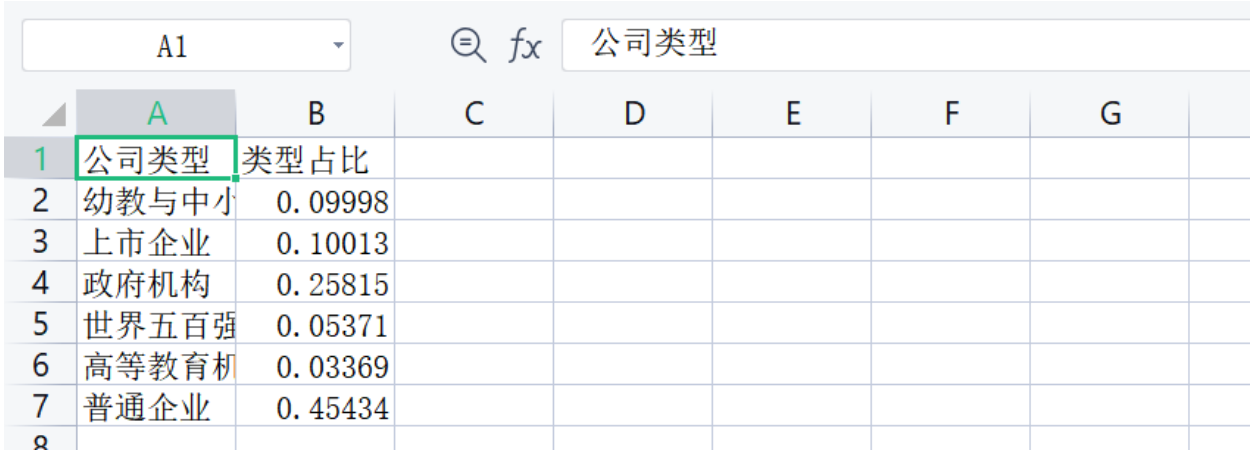
Out[18]: [29995, 30038, 77446, 16112, 10106, 136303]

- 计算比例（保留5位小数），打印，并写入文件

In [28]:
```
# 计算结果，保留5位小数，打印结果
total_num = 0
for i in range(len(num)):
    total_num += num[i]
for i in range(len(num)):
    num[i] /= total_num
    num[i] = round(num[i], 5)
for i in range(len(num)):
    print(firms_list[i] + ':    ' + str(num[i]))
```

```
幼教与中小学校:    0.09998
上市企业:    0.10013
政府机构:    0.25815
世界五百强:    0.05371
高等教育机构:    0.03369
普通企业:    0.45434
```

In [31]:
```
# 转为Pandas的DataFrame，写入文件
import pandas as pd
results = pd.DataFrame({'公司类型': firms_list, '类型占比': num})
results.to_csv('公司类型占比.csv', index = None)
```



	A	B	C	D	E	F	G
1	公司类型	类型占比					
2	幼教与中小	0.09998					
3	上市企业	0.10013					
4	政府机构	0.25815					
5	世界五百强	0.05371					
6	高等教育机	0.03369					
7	普通企业	0.45434					

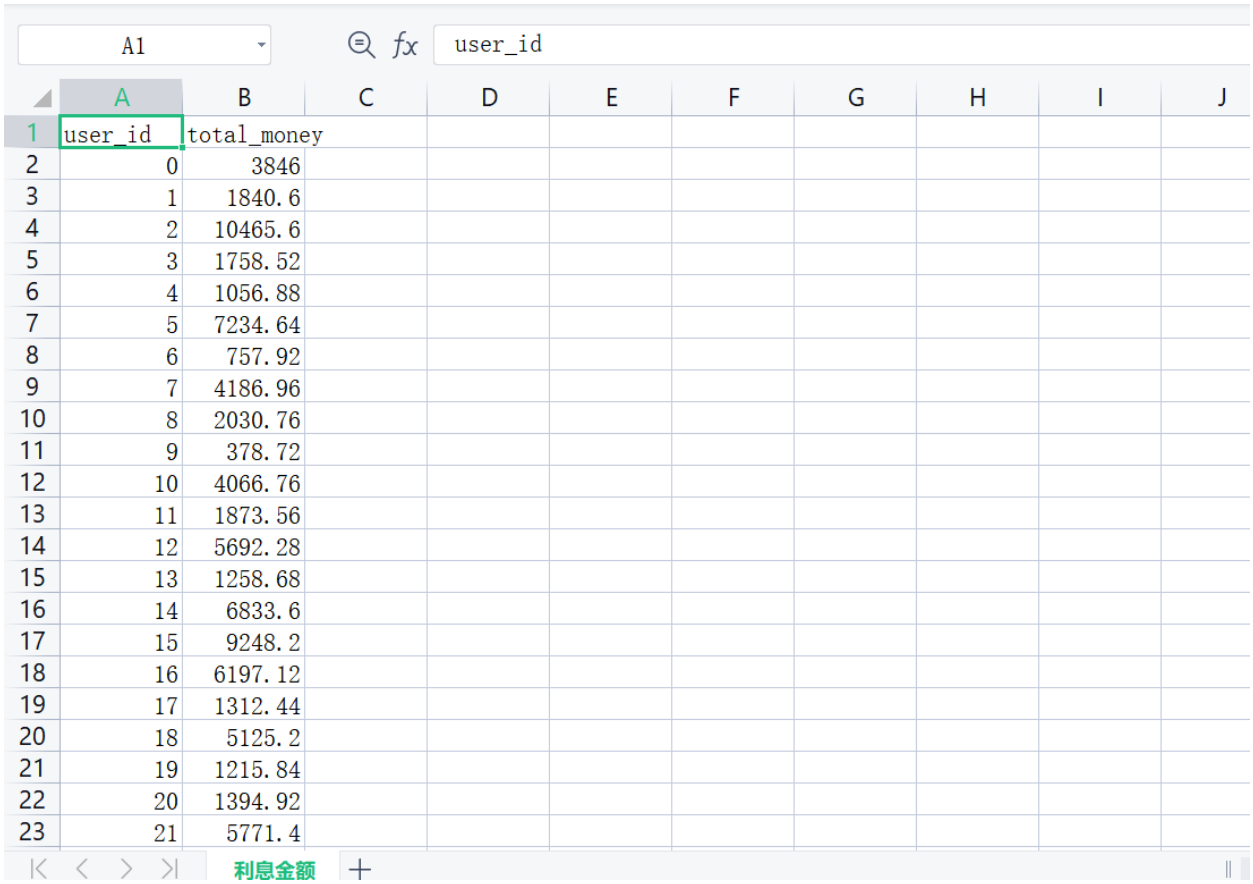
2. 利息金额

- 计算得到并增加新的列，使用withColumn函数

```
In [36]: # 增加利息金额列，展示，存入文件
interest = df.withColumn('total_money', df.year_of_loan * df.monthly_payment * 12 - df.total_loan).select('user_id','total_money')
interest.show()
interest = interest.toPandas()
interest.to_csv('利息金额.csv', index = None)
```

```
+-------+------------------+
|user_id|       total_money|
+-------+------------------+
|      0|            3846.0|
|      1|1840.6000000000004|
|      2|10465.600000000002|
|      3|1758.5200000000004|
|      4| 1056.880000000001|
|      5| 7234.639999999999|
|      6| 757.9200000000001|
|      7| 4186.959999999999|
|      8|2030.7600000000002|
|      9|378.72000000000116|
|     10| 4066.760000000002|
|     11|1873.5599999999977|
|     12| 5692.279999999999|
|     13|1258.6800000000003|
|     14|6833.5999999999985|
|     15| 9248.200000000004|
|     16| 6197.119999999995|
|     17|1312.4400000000005|
|     18| 5125.200000000001|
|     19|1215.8400000000001|
```

- 文件结果



	A	B	C	D	E	F	G	H	I	J
1	user_id	total_money								
2	0	3846								
3	1	1840.6								
4	2	10465.6								
5	3	1758.52								
6	4	1056.88								
7	5	7234.64								
8	6	757.92								
9	7	4186.96								
10	8	2030.76								
11	9	378.72								
12	10	4066.76								
13	11	1873.56								
14	12	5692.28								
15	13	1258.68								
16	14	6833.6								
17	15	9248.2								
18	16	6197.12								
19	17	1312.44								
20	18	5125.2								
21	19	1215.84								
22	20	1394.92								
23	21	5771.4								

利息金额

3. 工作年限超过5年的用户的房贷情况分布

- 观察工作年限的数据类型

```
In [37]: # 观察工作年限
         df.select(df.work_year).distinct().show()
```

```
+---------+
|work_year|
+---------+
|  5 years|
|  9 years|
|     null|
|   1 year|
|  2 years|
|  7 years|
|  8 years|
|  4 years|
|  6 years|
|  3 years|
| 10+ years|
|  < 1 year|
+---------+
```

- 将符合的工作年限整入一个list

```
In [39]: # 将符合的工作年限整入一个list
         work_years = df.select(df.work_year).distinct().toPandas()
         work_list = []
         num_list = [1, 5, 6, 8, 10]
         for i in num_list:
             work_list.append(work_years.iloc[i, 0])
         work_list
```

```
Out[39]: ['9 years', '7 years', '8 years', '6 years', '10+ years']
```

- 使用filter提取符合的用户，再选取所需的三列

```
In [43]: # 使用filter提取
         df_new = df.filter(df.work_year.isin(work_list) == True).select('user_id', 'censor_status', 'work_year')
         df_new.show()
```

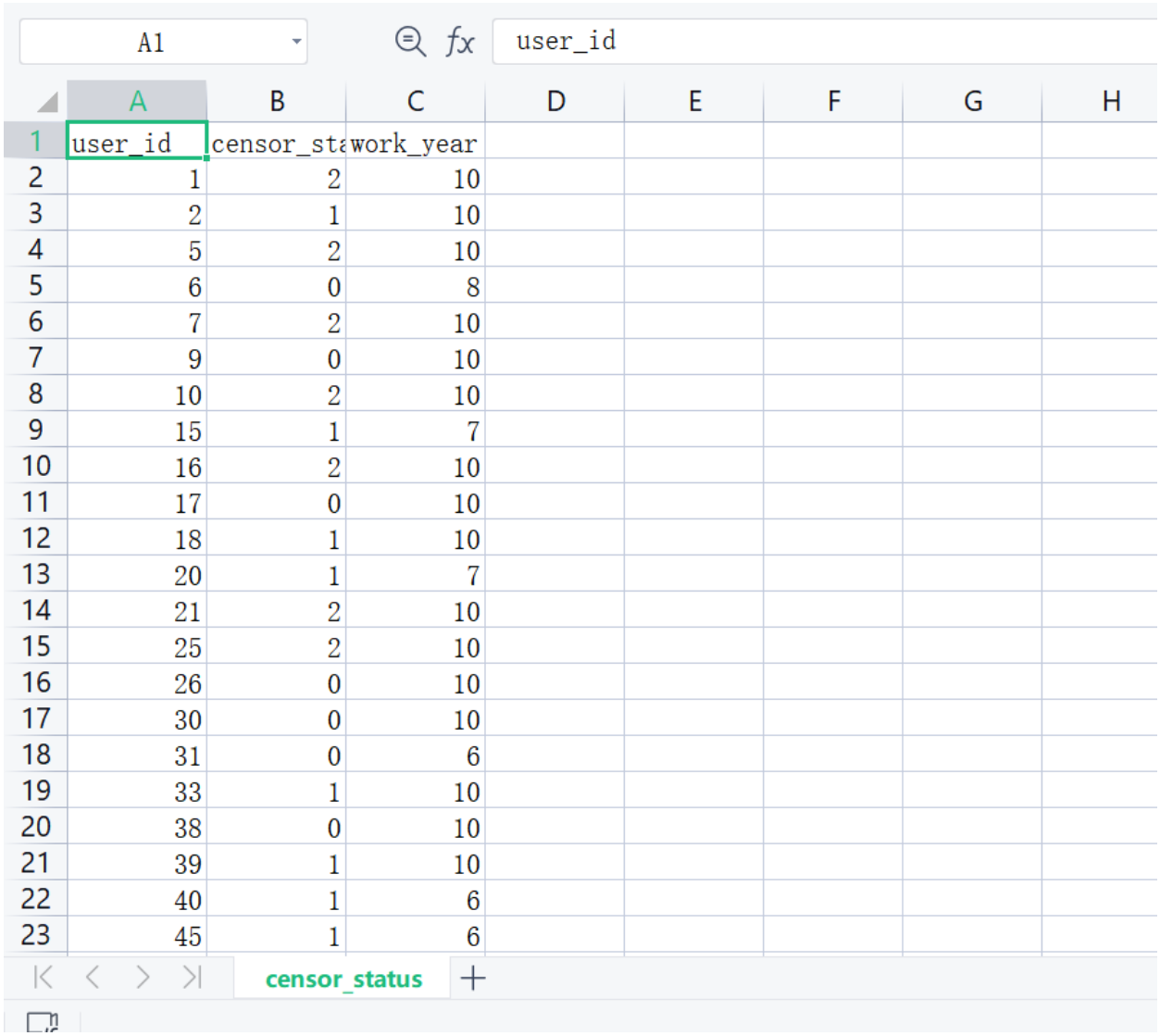
```
+-------+-------------+---------+
|user_id|censor_status|work_year|
+-------+-------------+---------+
|      1|            2|10+ years|
|      2|            1|10+ years|
|      5|            2|10+ years|
|      6|            0|  8 years|
|      7|            2|10+ years|
|      9|            0|10+ years|
|     10|            2|10+ years|
|     15|            1|  7 years|
|     16|            2|10+ years|
|     17|            0|10+ years|
|     18|            1|10+ years|
|     20|            1|  7 years|
|     21|            2|10+ years|
|     25|            2|10+ years|
|     26|            0|10+ years|
|     30|            0|10+ years|
|     31|            0|  6 years|
|     33|            1|10+ years|
|     38|            0|10+ years|
|     39|            1|10+ years|
+-------+-------------+---------+
only showing top 20 rows
```

- 使用withColumn构造新列，将work_year的字符串中只保留数字部分

```
In [72]: # 使用withColumn构造新列，将work_year的字符串中只保留数字部分
         df_new = df_new.withColumn('work_year', df.work_year[0:2])
         df_new.show()
```

```
+-------+-------------+---------+
|user_id|censor_status|work_year|
+-------+-------------+---------+
|      1|            2|       10|
|      2|            1|       10|
|      5|            2|       10|
|      6|            0|        8|
|      7|            2|       10|
|      9|            0|       10|
|     10|            2|       10|
|     15|            1|        7|
|     16|            2|       10|
|     17|            0|       10|
|     18|            1|       10|
|     20|            1|        7|
|     21|            2|       10|
|     25|            2|       10|
|     26|            0|       10|
|     30|            0|       10|
|     31|            0|        6|
|     33|            1|       10|
|     38|            0|       10|
|     39|            1|       10|
+-------+-------------+---------+
only showing top 20 rows
```

- 转换格式，存储



user_id	censor_sta	work_year
1	2	10
2	1	10
5	2	10
6	0	8
7	2	10
9	0	10
10	2	10
15	1	7
16	2	10
17	0	10
18	1	10
20	1	7
21	2	10
25	2	10
26	0	10
30	0	10
31	0	6
33	1	10
38	0	10
39	1	10
40	1	6
45	1	6

任务4

1. 思路
   - 先选取可能对is_default产生影响的用户信息列作为特征，这里排除loan_id、user_id、issue_date、post_code（个人认为邮政反应地区可能是有用的，但其种类太多了，加上有地区编码，就不将它作为特征了）、earlies_credit_line、title、policy_code
   - 然后进行特征工程，得到输入机器学习模型的最终特征数据
   - 划分数据集，进行训练
   - 比较不同的模型训练的结果，包括accuracy和f1 score，但是由于每次都随机划分训练集，可能存在偶然性，所以最后对每个模型随即划分5次、取平均数，来比较各个模型（在给定的参数下）的对于这个数据集的预测能力
2. 提取特征

In [1]:
```
import pyspark
import pandas as pd
from pyspark.sql import functions as F
from pyspark.sql import SQLContext
from pyspark import SparkContext
sc = SparkContext()
sqlContext = SQLContext(sc)
df = sqlContext.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load("train_data .csv")
```

```
C:\Users\DELL\anaconda3\envs\tensorflow\lib\site-packages\pyspark\sql\context.py:77: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
  warnings.warn(
```

In [2]:
```
# 选择特征
df = df.select('total_loan', 'year_of_loan','interest', 'monthly_payment',
               'class', 'sub_class', 'work_type', 'employer_type',
               'industry', 'work_year', 'house_exist', 'house_loan_status',
               'censor_status', 'marriage', 'offsprings', 'use', 'region',
               'debt_loan_ratio', 'del_in_18month', 'scoring_low', 'scoring_high',
               'pub_dero_bankrup', 'early_return', 'early_return_amount',
               'early_return_amount_3mon', 'recircle_b', 'recircle_u',
               'initial_list_status', 'policy_code', 'f0', 'f1', 'f2', 'f3', 'f4', 'f5',
               'is_default')
```

3. 特征工程，得到输入机器学习模型的最终特征数据

In [4]:
```
from pyspark.ml.feature import RFormula
supervised = RFormula(formula="is_default ~ total_loan + year_of_loan + interest + monthly_payment + sub_class + work_type + employer_type + in
fittedRF = supervised.fit(df)
preparedDF = fittedRF.transform(df)
train, test = preparedDF.randomSplit([0.8, 0.2])
train.show(5)
```

```
+----------+------------+--------+---------------+-----+---------+---------+-------------+--------------------+---------+-----------+----------+
--------+------------+--------+----------+---+------+-------------+-------------+----------+----------+---------------+------------+
-------------------+-------------------------+----------+----------+-------------------+-----------+----+----+----+----+----+----------+
--------------------+-----+
|total_loan|year_of_loan|interest|monthly_payment|class|sub_class|work_type|employer_type|            industry|work_year|house_exist|house_loa
n_status|censor_status|marriage|offsprings|use|region|debt_loan_ratio|del_in_18month|scoring_low|scoring_high|pub_dero_bankrup|early_return|
early_return_amount|early_return_amount_3mon|recircle_b|recircle_u|initial_list_status|policy_code|  f0|  f1|  f2|  f3|  f4|  f5|is_default|
features|label|
+----------+------------+--------+---------------+-----+---------+---------+-------------+--------------------+---------+-----------+----------+
--------+------------+--------+----------+---+------+-------------+-------------+----------+----------+---------------+------------+
-------------------+-------------------------+----------+----------+-------------------+-----------+----+----+----+----+----+----------+
--------------------+-----+
|     500.0|           3|    8.07|          15.69|    A|       A4|     职员|     普通企业|公共服务、社会组织|  < 1 year|          1|
0|           0|       0|          0|  5|   13|         4.27|          0.0|     720.0|     724.0|           null|           0|
0|                     0.0|       0.0|       0.0|                  1|        1.0|null|null|null|null|null|         0|(71,[0,1,2,3,15,
3...|  0.0|
|     900.0|           3|   10.28|          29.16|    C|       C1|     工人|     政府机构|公共服务、社会组织|  3 years|          1|
0|           0|       0|          0| 13|   21|         16.7|          0.0|     685.0|     689.0|           null|           0|
0|                     0.0|    3084.0|      65.6|                  1|        1.0|null|null|null|null|null|null|         1|(71,[0,1,2,3,4,4
0...|  1.0|
```

4. 使用逻辑回归

In [5]:
```
# 使用逻辑回归预测并展示结果
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(labelCol="label",featuresCol="features")
fittedLR = lr.fit(train)
fittedLR.transform(test).select("label", "prediction").show(30)
```

```
+-----+----------+
|label|prediction|
+-----+----------+
|  0.0|       0.0|
|  0.0|       0.0|
|  1.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  1.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  1.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  1.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
```

5. 计算准确率和f1 score

- 先统计出各类预测情况的数量

In [10]:
```
# 保存结果，并计算各类预测情况的数量
result = fittedLR.transform(test).select("label", "prediction")
TP = result.where("label = 1 and prediction = 1").count()
TN = result.where("label = 0 and prediction = 0").count()
FP = result.where("label = 0 and prediction = 1").count()
FN = result.where("label = 1 and prediction = 0").count()
print(TP, TN, FP, FN)
```

```
4775 45168 2616 7195
```

- 计算准确率和f1 score

In [12]:
```
# 计算准确率、精准度、召回率和f1 score
accuracy = (TP + TN) / (TP + TN + FP + FN)
precision = TP / (TP + FP)
recall = TP / (TP + FN)
f1 = 2 * precision * recall / (precision + recall)
print('准确率为：', accuracy)
print('精准度为：', precision)
print('召回率为：', recall)
print('f1 score = ', f1)
```

```
准确率为： 0.835810154968705
精准度为： 0.6460560140711676
召回率为： 0.3989139515455305
f1 score = 0.4932596456794586
```

可见模型精确度尚可，但是f1 score很低，其原因在于召回率过低（即正的样本中被判断为正的概率过低）

6. 使用GBT模型训练，效果和逻辑回归接近

In [18]:
```
from pyspark.ml.classification import GBTClassifier
gbt = GBTClassifier(labelCol="label", featuresCol="features", maxIter=10)
gbtModel = gbt.fit(train)
result = gbtModel.transform(test).select("label", "prediction")
TP = result.where("label = 1 and prediction = 1").count()
TN = result.where("label = 0 and prediction = 0").count()
FP = result.where("label = 0 and prediction = 1").count()
FN = result.where("label = 1 and prediction = 0").count()
accuracy = (TP + TN) / (TP + TN + FP + FN)
precision = TP / (TP + FP)
recall = TP / (TP + FN)
f1 = 2 * precision * recall / (precision + recall)
print('TP：', TP, '  TN：', TN, '  FP：', FP, '  FN：', FN)
print('准确率为：', accuracy)
print('精准度为：', precision)
print('召回率为：', recall)
print('f1 score = ', f1)
```

```
TP： 4429  TN： 45376  FP： 2408  FN： 7541
准确率为： 0.8335006861465342
精准度为： 0.6477987421383647
召回率为： 0.3700083542188805
f1 score = 0.470994842345935
```

7. 为了减少随机划分训练集带来的影响，将随机划分训练集重复十次，每次记下准确率和f1 score，然后取平均，比较上述两种方法的情况

- 先将逻辑回归、GBT分别抽象成函数，输入为train和test，返回为accuracy和f1 score的一个list

In [22]:
```
def Logistic(train, test):
    lr = LogisticRegression(labelCol="label",featuresCol="features")
    fittedLR = lr.fit(train)
    result = fittedLR.transform(test).select("label", "prediction")
    TP = result.where("label = 1 and prediction = 1").count()
    TN = result.where("label = 0 and prediction = 0").count()
    FP = result.where("label = 0 and prediction = 1").count()
    FN = result.where("label = 1 and prediction = 0").count()
    accuracy = (TP + TN) / (TP + TN + FP + FN)
    precision = TP / (TP + FP)
    recall = TP / (TP + FN)
    f1 = 2 * precision * recall / (precision + recall)
    lis = [accuracy, f1]
    return lis
def GBT(train, test):
    gbt = GBTClassifier(labelCol="label", featuresCol="features", maxIter=10)
    gbtModel = gbt.fit(train)
    result = gbtModel.transform(test).select("label", "prediction")
    TP = result.where("label = 1 and prediction = 1").count()
    TN = result.where("label = 0 and prediction = 0").count()
    FP = result.where("label = 0 and prediction = 1").count()
    FN = result.where("label = 1 and prediction = 0").count()
    accuracy = (TP + TN) / (TP + TN + FP + FN)
    precision = TP / (TP + FP)
    recall = TP / (TP + FN)
    f1 = 2 * precision * recall / (precision + recall)
    lis = [accuracy, f1]
    return lis
```

- 循环十遍，每次随机划分，算两种方法的平均值，输出

In [25]:
```
# 循环十遍，每次重新随机划分，取十次的平均准确率和f1 score
LOG_acc = 0
LOG_f1 = 0
GBT_acc = 0
GBT_f1 = 0
for i in range(10):
    train, test = preparedDF.randomSplit([0.8, 0.2])
    LOG_lis = Logistic(train, test)
    GBT_lis = GBT(train, test)
    LOG_acc += LOG_lis[0]
    LOG_f1 += LOG_lis[1]
    GBT_acc += GBT_lis[0]
    GBT_f1 += GBT_lis[1]
# 取平均数
LOG_acc /= 10
LOG_f1 /= 10
GBT_acc /= 10
GBT_f1 /= 10
print('逻辑回归的平均准确率为：', LOG_acc, '平均f1 score为：', LOG_f1)
print('GBT的平均准确率为：', GBT_acc, '平均f1 score为：', GBT_f1)
```

```
逻辑回归的平均准确率为： 0.836561214701522 平均f1 score为： 0.49984261019117165
GBT的平均准确率为： 0.833929930088191 平均f1 score为： 0.48826365228720814
```

两者的准确率接近，逻辑回归的f1 score要高出一些