钱勋潮 191250115 实验四

任务一

mapreduce代码设计

框架分析

任务需求：统计`industry`的频率并按照频数降序排列

任务一比较类似与作业5的词频统计，首先使用一个job统计`industry`的频数并且输出到临时文件夹，然后再使用一个job从临时文件中读取数据，将key与value转置，通过限定reduce数目为1来排列转置之后的key，最后输出到目标文件夹。

伪代码

```
//第一个job的map与reduce
class IndustryMapper
method Map(Object key,Text t)
    word = t.split(",")[10]
    Emit(word,1)

class IntSumReducer
method Reduce(Text word,Int val)
    int sum = 0
    sum += val
    Emit(word,sum)

//第二个job的map与reduce
class InverseMapper
method Map(Word t,Int val)
    Emit(val,t)

class SimpleReducer
method Reduce(Int val,Word t)
    Emit(t,val)
```

细节设计

`industry`数据的读取

csv文件按照`,`隔开，于是对读取的一行`String`按照`,`切割得到一个`String`列表，并且读取列表对应位置的值。因为第一行索引也被读取进去了，所以多加一道判断去除industry的索引名。

```
String str = value.toString();
String[] strList = str.split(",");
if(strList[10].equals("industry"))
    return;
word.set(strList[10]);
context.write(word, one);
```

运行结果

industry	count
金融业	48216
电力、热力生产供应业	36048
公共服务、社会组织	30262
住宿和餐饮业	26954
文化和体育业	24211
信息传输、软件和信息技术服务业	24078
建筑业	20788
房地产业	17990
交通运输、仓储和邮政业	15028
采矿业	14793
农、林、牧、渔业	14758
国际组织	9118
批发和零售业	8892
制造业	8864

运行参数

`hdfs://localhost:9000/exam4/input  hdfs://localhost:9000/exam4/output`

运行结果eclipse打开是乱码，在notepad++上用UTF-8格式打开就正确了

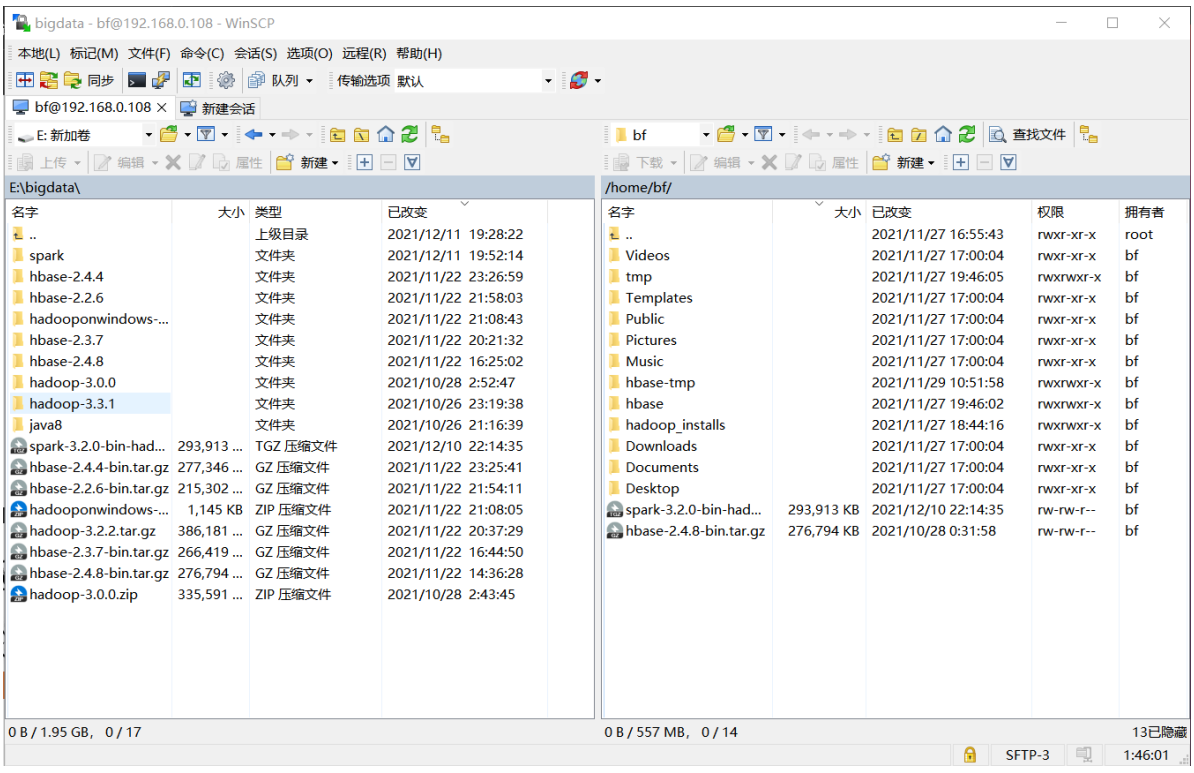
```
金融业  48216
电力、热力生产供应业    36048
公共服务、社会组织  30262
住宿和餐饮业    26954
文化和体育业    24211
信息传输、软件和信息技术服务业  24078
建筑业  20788
房地产业    17990
交通运输、仓储和邮政业  15028
采矿业  14793
农、林、牧、渔业    14758
国际组织    9118
批发和零售业    8892
制造业  8864
```

Spark安装与配置

环境与版本：

软件	版本
Ubuntu	20.04.3 LTS
Java	1.8.0_301
hadoop	3.3.1
spark	3.2.0
python	3.8.10

spark下载与安装



在官网下载spark压缩包，winscp发送到虚拟机，解压到文件夹。

编辑系统变量

```
export SPARK_HOME=/usr/local/spark/spark-3.2.0-bin-hadoop3.2
export PATH=${SPARK_HOME}/bin:$PATH
```

试运行样例代码

```
bf@bf-VirtualBox:/usr/local/spark/spark-3.2.0-bin-hadoop3.2$ ./bin/run-example S
parkPi 10
21/12/11 21:52:45 WARN Utils: Your hostname, bf-VirtualBox resolves to a loopbac
k address: 127.0.1.1; using 192.168.0.108 instead (on interface enp0s3)
21/12/11 21:52:45 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
address
21/12/11 21:52:45 WARN NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
21/12/11 21:52:46 INFO SparkContext: Running Spark version 3.2.0
```

在一堆INFO里找到输出

```
Stage finished
21/12/11 21:52:49 INFO DAGScheduler: Job 0 finished: reduce at SparkPi.scala:38,
 took 1.147011 s
Pi is roughly 3.1423551423551426
21/12/11 21:52:49 INFO SparkUI: Stopped Spark web UI at http://192.168.0.108:404
```

pyspark使用

```
SparkSession available as 'spark'.
>>> from pyspark import SparkContext
>>> from pyspark import SparkConf
>>> data = list(range(1,1000))
>>> rdd = sc.parallelize(data)
>>> rdd.collect()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82,
```

python开发环境安装

安装pip3`sudo apt install python3-pip`

```
bf@bf-VirtualBox:~$ pip3 --version
pip 20.0.2 from /usr/lib/python3/dist-packages/pip (python 3.8)
```

安装jupyter`pip install jupyter`

```
bf@bf-VirtualBox:~$ jupyter --version
Selected Jupyter core packages...
IPython          : 7.30.1
ipykernel        : 6.6.0
ipywidgets       : 7.6.5
jupyter_client   : 7.1.0
jupyter_core     : 4.9.1
jupyter_server   : not installed
jupyterlab       : not installed
nbclient         : 0.5.9
nbconvert        : 6.3.0
nbformat         : 5.1.3
notebook         : 6.4.6
qtconsole        : 5.2.1
traitlets        : 5.1.1
```

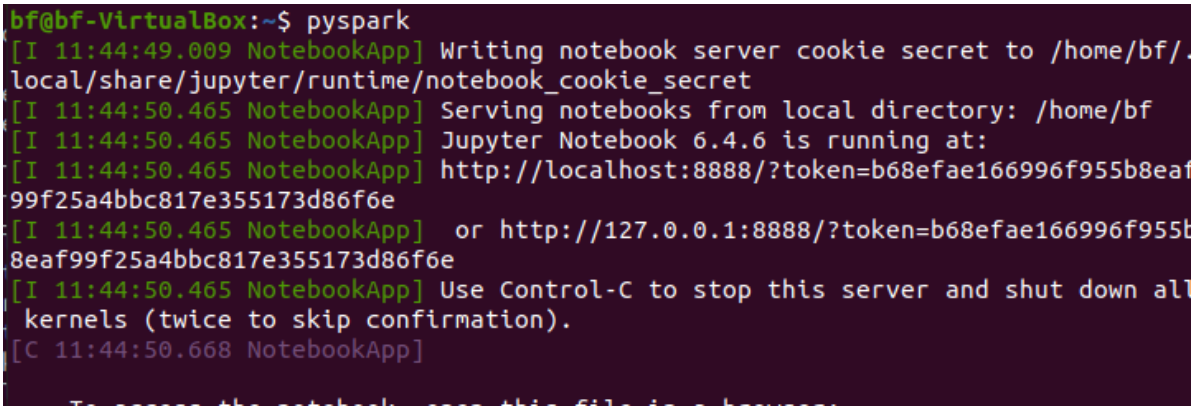
在`~/.bashrc`文件最后添加Pyspark driver的环境变量

```
export PYSPARK_DRIVER_PYTHON=jupyter
export PYSPARK_DRIVER_PYTHON_OPTS='notebook'
```

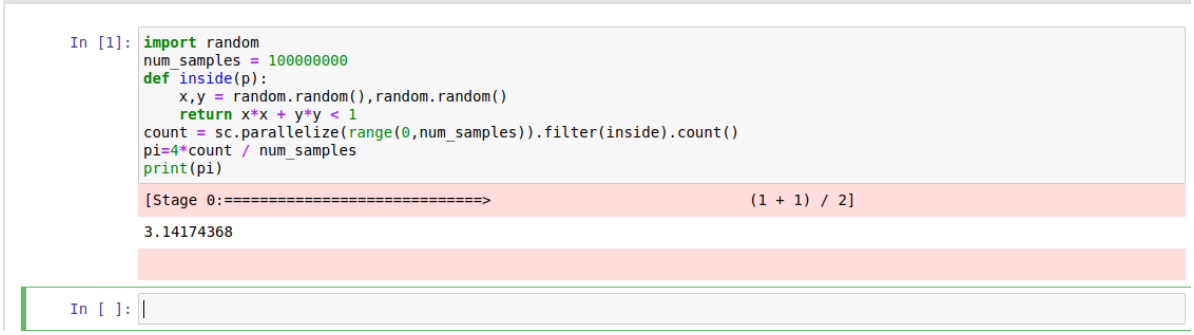
运行一下刚刚修改的初始化文件

```
$ source ~/.bashrc
```

此时打开pyspark就自动打开jupyter notebook



试运行一个程序，配置成功！



任务二

代码设计

- 首先读取数据到RDD，每一行csv是一个元素

```
import math
lines = sc.textFile('train_data.csv')
header = lines.first()#第一行
lines = lines.filter(lambda row:row != header)#删除第一行
```

- 转化操作，对每个元素按照`,`分隔，另元素为列表中的第三个元素也就是`total_loan`。

```
loan_lines = lines.map(lambda x:float(x.split(',')[2]))
```

- 定义一个`pair rdd`的生成函数，键为`(2000,3000)`，值为1

```
def get_pair(x):
    floor = int(x/1000)*1000
    ceil = math.ceil(x/1000)*1000
    if floor == ceil:
        ceil += 1000
    key = '('+str(floor)+','+str(ceil)+')'
    return (key,1)
```

- 转化操作，对RDD每个元素应用上述函数，生成`pair rdd`

```
pair_loan_lines = loan_lines.map(get_pair)
```

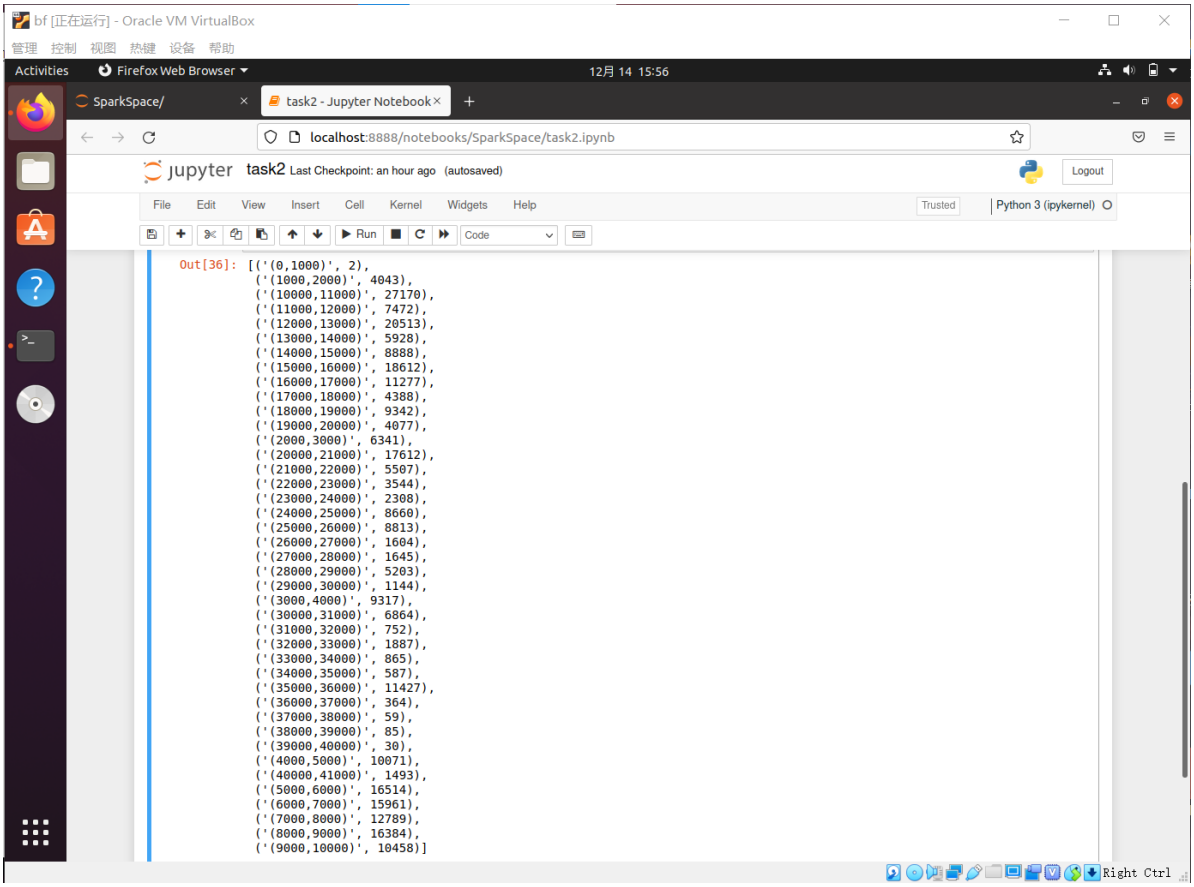
- 行动操作，对`pair rdd`进行按键聚合并输出

```
result = pair_loan_lines.reduceByKey(lambda x, y: x + y)
result.sortByKey().collect()
```

运行结果

范围	样本数
(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
(22000,23000)	3544
(23000,24000)	2308
(24000,25000)	8660
(25000,26000)	8813
(26000,27000)	1604
(27000,28000)	1645
(28000,29000)	5203
(29000,30000)	1144

范围	样本数
(30000,31000)	6864
(31000,32000)	752
(32000,33000)	1887
(33000,34000)	865
(34000,35000)	587
(35000,36000)	11427
(36000,37000)	364
(37000,38000)	59
(38000,39000)	85
(39000,40000)	30
(40000,41000)	1493



任务三

问题1

代码设计

首先读取csv文件到dataframe，然后生成表`loan_table`

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .enableHiveSupport().getOrCreate()
df = spark.read.options(header='True', inferSchema='True', delimiter=',') \
  .csv("train_data.csv")
df.createOrReplaceTempView("loan_table")
```

用sql语句进行查询

```
result = spark.sql("SELECT employer_type,COUNT(*)/(SELECT count(*) FROM
loan_table) AS percentage FROM loan_table GROUP BY (employer_type);")
```

运行结果

```
result.show()
```

```
[Stage 47:============================>
```

```
+--------------+--------------------+
| employer_type|          percentage|
+--------------+--------------------+
|幼教与中小学校| 0.09998333333333333|
|      上市企业| 0.10012666666666667|
|      政府机构| 0.25815333333333335|
|    世界五百强|0.053706666666666666|
|  高等教育机构| 0.03368666666666666|
|      普通企业|  0.4543433333333333|
+--------------+--------------------+
```

```
employer_type,percentage
幼教与中小学校,0.09998333333333333
上市企业,0.10012666666666667
政府机构,0.25815333333333335
世界五百强,0.053706666666666666
高等教育机构,0.03368666666666666
普通企业,0.4543433333333333
```

问题2

代码设计

使用sql语句进行计算

```
result2 = spark.sql("select user_id,year_of_loan*monthly_payment*12-total_loan
as total_money from loan_table")
```

运行结果

```
result2.show()
```

```
+-------+------------------+
|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|
+-------+------------------+
only showing top 20 rows
```

```
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
20,1394.920000000002
21,5771.4000000000015
22,3202.4799999999996
23,4940.5999999999985
24,12231.0
```

问题3

代码设计

发现用来筛选的条件的数据为字符串，查看所有数据种类，其中代表市场大于五年的有如下几种`'10+ years','6 years','7 years','8 years','9 years'`，所以只需要判断工作年限是否为这几个值中的一个即可。

```
result3 = spark.sql("select user_id,censor_status,work_year from loan_table
where work_year in ('10+ years','6 years','7 years','8 years','9 years')")
```

运行结果

```
result3.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
```

```
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
40,1,6 years
45,1,6 years
46,0,8 years
```

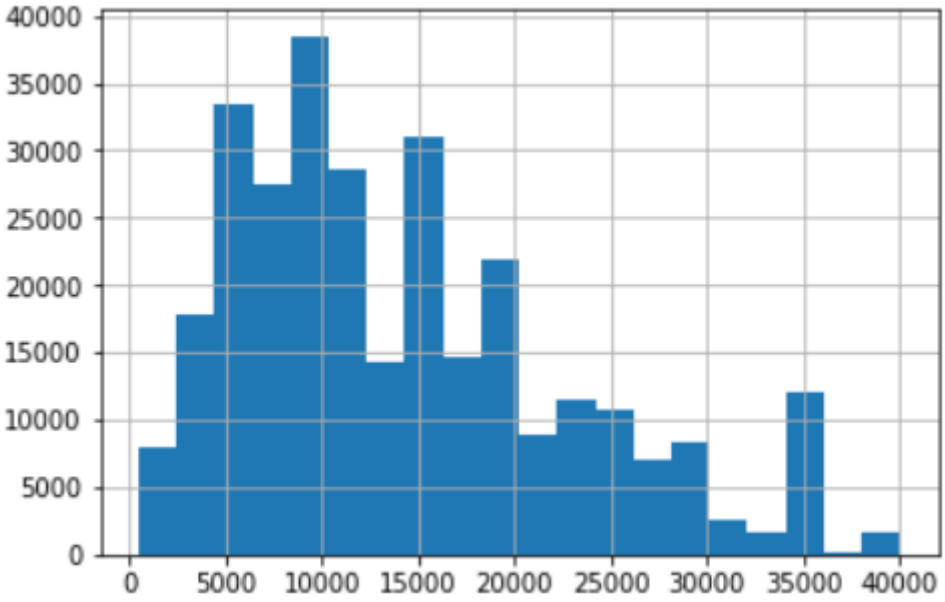
任务四

数据预处理

数据预览与缺失值处理

数据分布情况

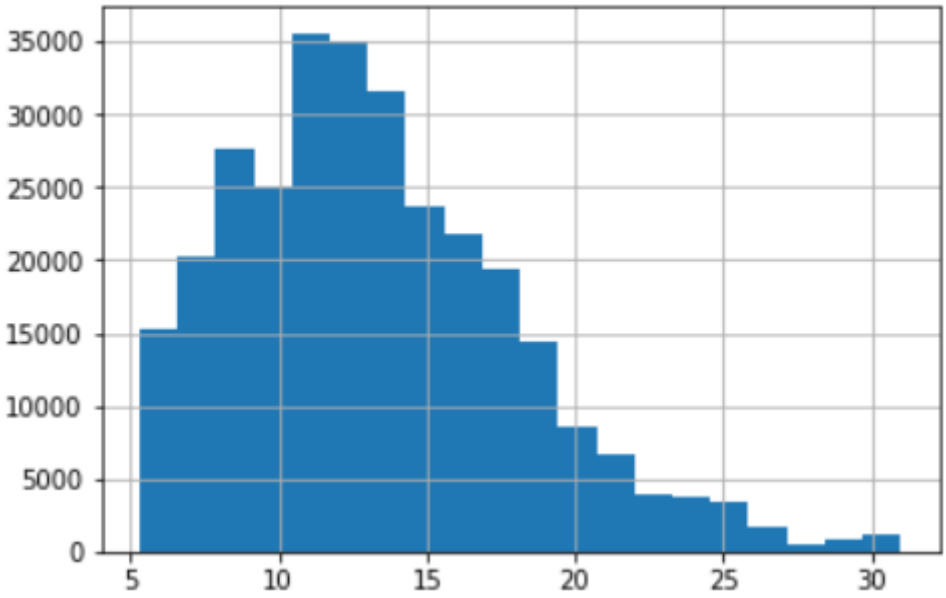
`total_loan`数据分布



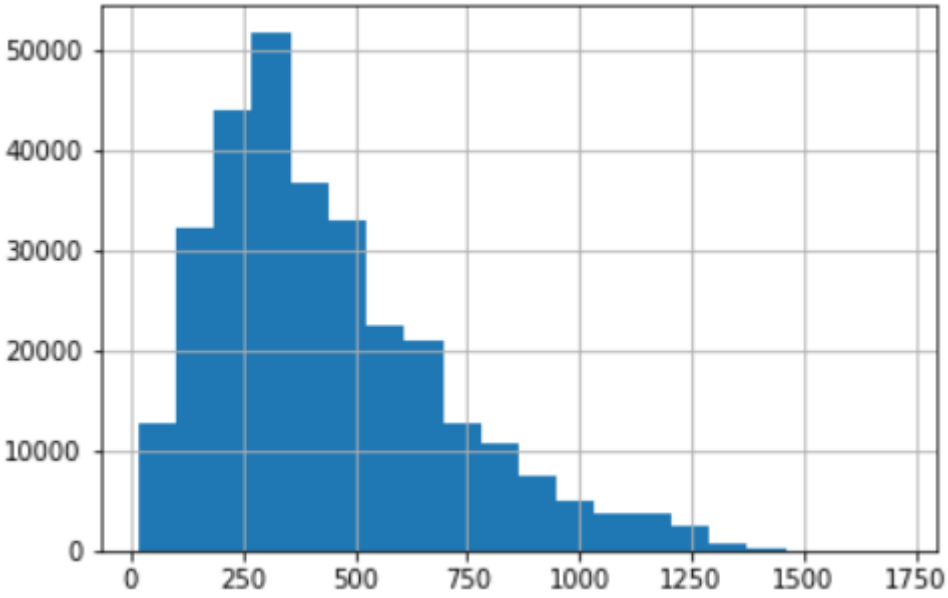
`year_of_loan` 贷款年限为3年或5年，主要为3年

```
3     227445
5      72555
```

`interest` 网络贷款利率，数据分布，利率真高啊



`monthly_payment` 分期付款金额



`class` 网络贷款等级

B	87477
C	85247
A	52417
D	44644
E	20995
F	7188
G	2032

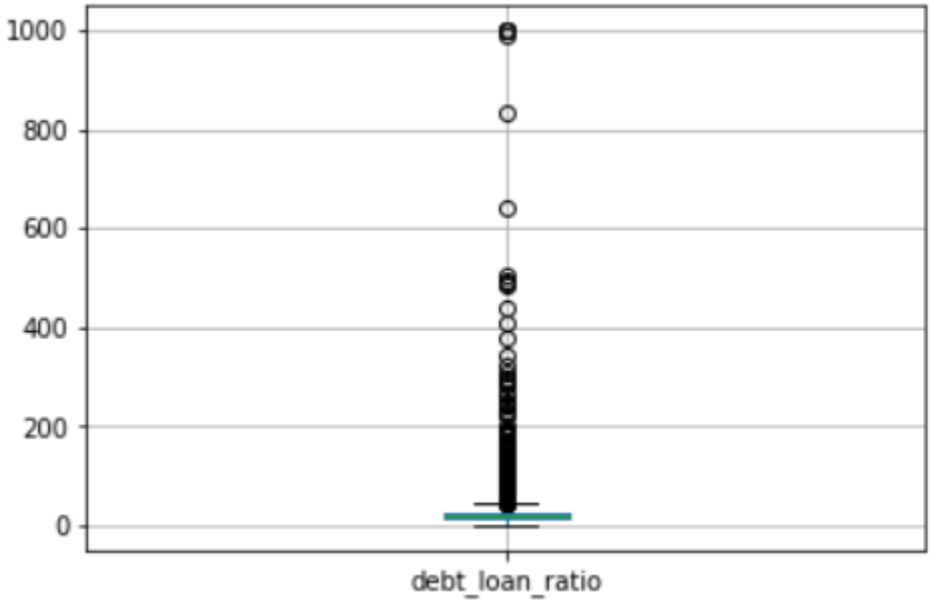
`house_exist` 是否有房

0	148433
1	119169
2	32276
3	87
5	29
4	6

`house_loan_status` 数据分布

0	153750
1	113823
2	32427

`debt_loan_ratio` 数据分布

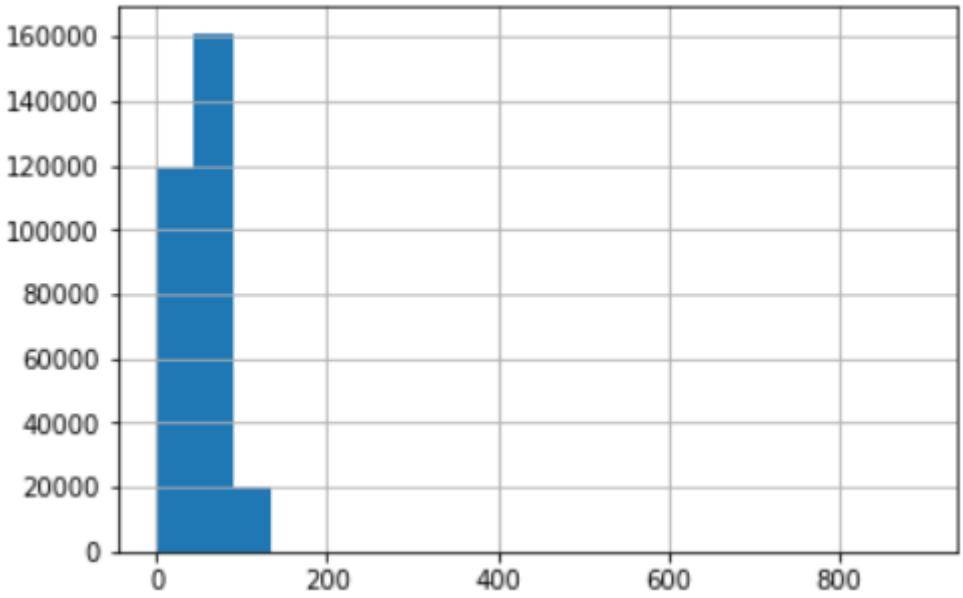


count	299916.000000
mean	18.252060
std	10.321016
min	-1.000000
25%	11.780000
50%	17.590000
75%	24.040000
max	999.000000

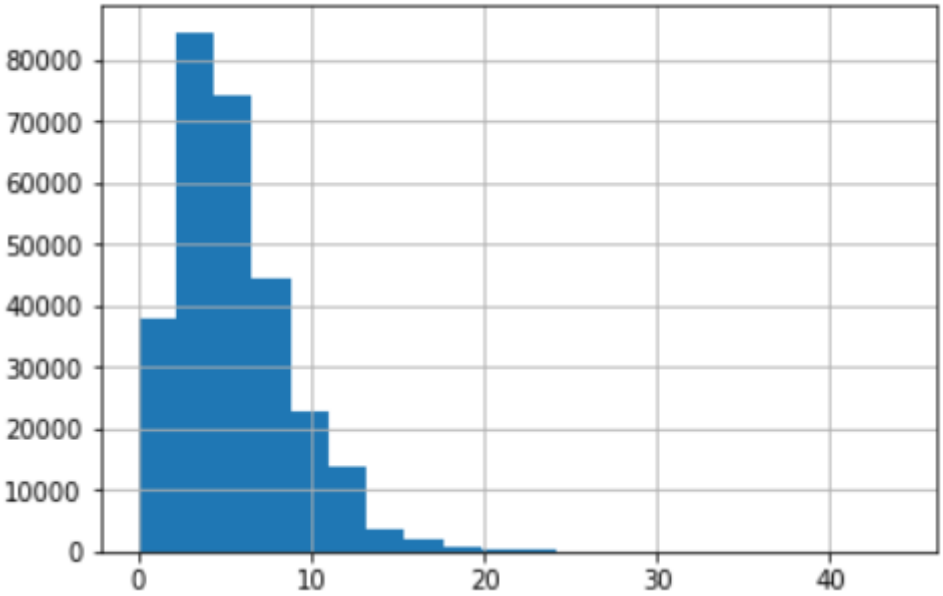
`pub_dero_bankrup` 数据分布

0.0	262454
1.0	35149
2.0	1705
3.0	380
4.0	89
5.0	35
6.0	9
7.0	4
8.0	3
12.0	1
9.0	1

`recircle_u` 数据分布



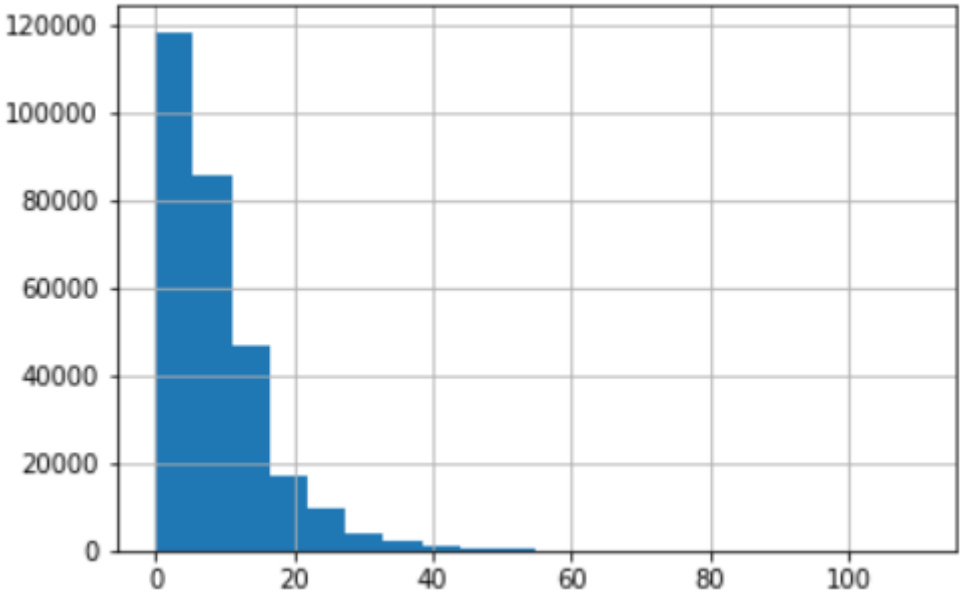
f0



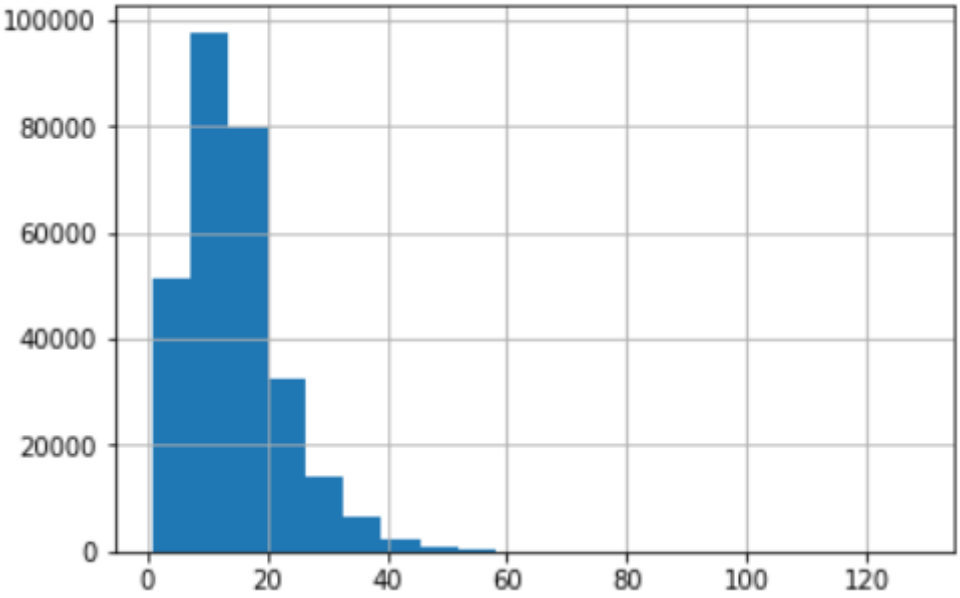
f1

0.0	273670
1.0	206
2.0	10
4.0	1

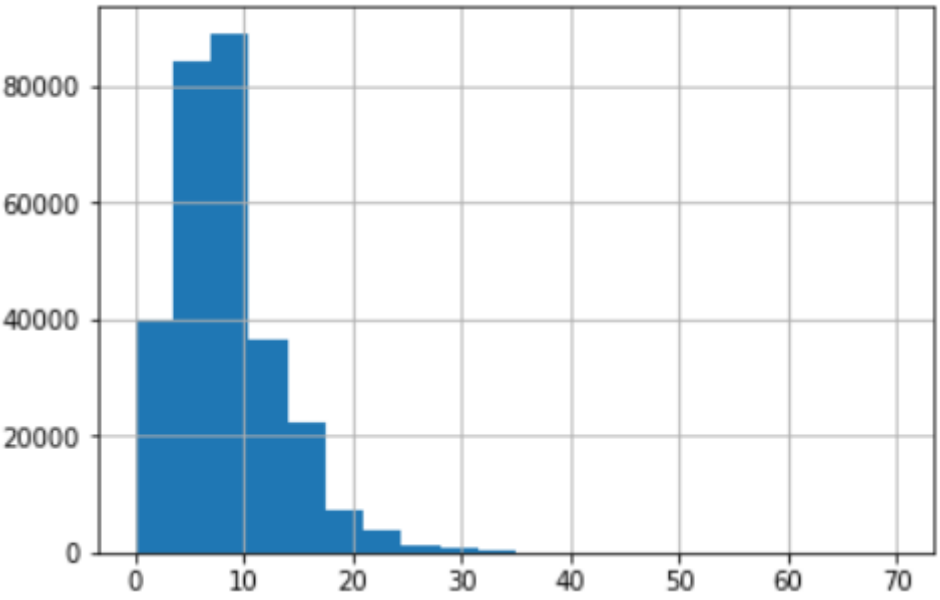
f2


120000
100000
80000
60000
40000
20000
0
0
20
40
60
80
100

`f3`


100000
80000
60000
40000
20000
0
0
20
40
60
80
100
120

`f4`


80000
60000
40000
20000
0
0
10
20
30
40
50
60
70

`f5`

0.0	217311
1.0	35944
2.0	14520
3.0	6534
4.0	3999
5.0	2325
6.0	1520
7.0	959
8.0	595
9.0	360
10.0	245
11.0	170
12.0	104
13.0	68
14.0	56
16.0	27
15.0	25
18.0	12
17.0	12
23.0	11
22.0	10
19.0	8
21.0	8
24.0	5
25.0	3
20.0	3
31.0	2
27.0	2

缺失值处理

其中`work_year` `post_code` `debt_loan_ratio` `pub_dero_bankrup` `recircle_u` `f0` `f1` `f2` `f3` `f4` `f5`这些指标存在缺失值，根据统计结果，缺失值处理方式如下：

`work_year`空值17428个，使用0填充

`post_code`空值1个，用众数填充

`debt_loan_ratio`空值84个，使用中位数填充

`pub_dero_bankrup`空值170个，使用中位数填充

`recircle_u`空值196个，使用中位数填充空值

`f系列` `f1`空值26113个，其他`f`值空值15154个，均用中位数填充

数据特征化

1. 特征向量中去除`loan_id`与`user_id`
2. `class`网络贷款等级A-G转换为1-7
3. `sub_class`转换为1-5
4. `work_type` `employment_type` `industry`直接使用*LabelEncoder*编码处理
5. `work_year`小于1年转为0，10+转为10，并转为整型

添加新变量

将任务三问题3计算出来的`total_money`作为新变量加入

逻辑回归LR

优缺点分析

优点：

- （模型）模型清晰，背后的概率推导经得住推敲。
- （输出）输出值自然地落在0到1之间，并且有概率意义。
- （参数）参数代表每个特征对输出的影响，可解释性强。
- （简单高效）实施简单，非常高效（计算量小、存储占用低），可以在大数据场景中使用。
- （过拟合）解决过拟合的方法很多，如L1、L2正则化。

- （多重共线性）L2正则化就可以解决多重共线性问题。

缺点：

- （特征相关情况）因为它本质上是一个线性的分类器，所以处理不好特征之间相关的情况。
- （特征空间）特征空间很大时，性能不好。
- （精度）容易欠拟合，精度不高。

在本例子中，特征向量较大，性能可能受到影响。可能存在多重共线性，需要正则化调优。

原始版本

```
###第一步，对原始数据集进行向量化，得到一个features向量
df_assembler = VectorAssembler(inputCols=[
'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', 'post_code', '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', 'title','policy_code',
 'f0', 'f1', 'f2', 'f3', 'f4', 'f5', 'total_money'],outputCol='features')
labeled_df = df_assembler.transform(df)


###第二步，划分数据集
labeled_df = labeled_df["is_default","features"]
data_set = labeled_df.select(['features', 'is_default'])
train_df, test_df = data_set.randomSplit([0.8, 0.2])


###第三步，模型训练与预测
log_reg = LogisticRegression(labelCol = 'is_default').fit(train_df)
test_result = log_reg.evaluate(test_df).predictions
evaluator =
BinaryClassificationEvaluator(rawPredictionCol='rawPrediction',labelCol='is_defa
ult')
print('AUC: ', evaluator.evaluate(test_result))
```

```
evaluator = BinaryClassificationEvaluator(rawPredictionCol='rawPrediction',labelCol='is_default')
print('AUC: ', evaluator.evaluate(test_result))
```

```
AUC:  0.8014095445033663
```

在不进行调参优化的情况下，AUC为0.8。

调优

首先尝试进行L1,L2正则化调优，让模型更具有泛化能力。

但是发现并不能对模型有优化左右，参数越大，效果越差。

然后尝试了对 `threshold` 参数进行修改，也就是逻辑回归模型里，根据输出值划分0，1预测值的阈值。默认为0.5，考虑到在本样本中，1的占比很低，需要，可能在划分标准上进行修改可以更加贴近拟合结果。

```
for i in range(0,40):
    threshold = 0.3+0.01*i
    log_reg_2 = LogisticRegression(threshold = threshold,labelCol =
'is_default').fit(train_df)
    test_result_2 = log_reg_2.evaluate(test_df).predictions
    evaluator =
BinaryClassificationEvaluator(rawPredictionCol='rawPrediction',labelCol='is_defa
ult')
    print('AUC: '+str(threshold), evaluator.evaluate(test_result_2))
```

```
AUC: 0.47 0.8014090317401117

AUC: 0.48 0.8014140111247791

AUC: 0.49 0.8014126158506163

AUC: 0.5 0.8014120638452764

AUC: 0.51 0.8014137407904098

AUC: 0.52 0.8014100537784358
```

总体来说，效果非常有限，提升在万分之一以内。在这个特征向量构造的情况下，AUC基本上能达到0.801

神经网络——多层感知器MLP

优缺点分析

多层感知器（Multilayer Perceptron,缩写MLP）是一种前向结构的人工神经网络，映射一组输入向量到一组输出向量。MLP可以被看作是一个有向图，由多个的节点层所组成，每一层都全连接到下一层。除了输入节点，每个节点都是一个带有非线性激活函数的神经元（或称处理单元）。一种被称为反向传播算法方法常被用来训练MLP。多层感知器遵循人类神经系统原理，学习并进行数据预测。它首先学习，然后使用权重存储数据，并使用算法来调整权重并减少训练过程中的偏差，即实际值和预测值之间的误差。主要优势在于其快速解决复杂问题的能力。多层感知的基本结构由三层组成：第一输入层，中间隐藏层和最后输出层，输入元素和权重的乘积被馈给具有神经元偏差的求和结点,主要优势在于其快速解决复杂问题的能力。 MLP是感知器的推广，克服了感知器不能对线性不可分数据进行识别的弱点。

原始版本

输入的变量有38个，输出的变量有2个，中间设置两层隐藏层，第一层节点数60，第二层节点数30。

```
###模型训练
layers = [38,60,30,2]
MLPC_trainer = MultilayerPerceptronClassifier(labelCol='is_default',
featuresCol="features", maxIter=100, layers=layers, blockSize=128, seed=1234)
MLPC_model = MLPC_trainer.fit(train_df)
###模型预测
MLPC_predictions = MLPC_model.transform(test_df)
evaluator =
BinaryClassificationEvaluator(rawPredictionCol='rawPrediction',labelCol='is_defa
ult')
print('AUC: ', evaluator.evaluate(MLPC_predictions))
```

但是节点数过多，导致内存不够😭😭😭，于是设置第一层的隐藏层为40，AUC为0.65左右。

因为虚拟机性能限制，无法充分发挥这一算法。

随机森林RF

首先尝试了单个决策树

```
evaluator = BinaryClassificationEvaluator(rawPredictionCol='rawPrediction',labelCol='is_default')
print('AUC: ', evaluator.evaluate(dt_pred))
```

```
[Stage 148:>                                                        (0 + 2) / 2]
```

```
AUC:  0.5868737697818849
```

但是单个决策树的效果非常不佳，也不想调参了，尝试使用随机森林。

但是在随机森林调参之后，得到启发，可以对决策树调参，设置最深层数为15

dt

```
dt = DecisionTreeClassifier(labelCol="is_default", featuresCol="features",maxDepth=15)
```

```
dt_model = dt.fit(train_df)
```

```
21/12/19 14:07:32 WARN package: Truncated the string representation of a plan since it was too large. This behavior
can be adjusted by setting 'spark.sql.debug.maxToStringFields'.
```

```
dt_pred = dt_model.transform(test_df)
```

```
evaluator = BinaryClassificationEvaluator(rawPredictionCol='rawPrediction',labelCol='is_default')
print('AUC: ', evaluator.evaluate(dt_pred))
```

```
AUC:  0.7556666089317783
```

AUC能够达到0.756，还行吧。

优缺点分析

随机森林是由很多决策树构成的，不同决策树之间没有关联。

当我们进行分类任务时，新的输入样本进入，就让森林中的每一棵决策树分别进行判断和分类，每个决策树会得到一个自己的分类结果，决策树的分类结果中哪一个分类最多，那么随机森林就会把这个结果当做最终的结果。

1. 一个样本容量为N的样本，有放回的抽取N次，每次抽取1个，最终形成了N个样本。这选择好了的N个样本用来训练一个决策树，作为决策树根节点处的样本。
2. 当每个样本有M个属性时，在决策树的每个节点需要分裂时，随机从这M个属性中选取出m个属性，满足条件m << M。然后从这m个属性中采用某种策略（比如说信息增益）来选择1个属性作为该节点的分裂属性。
3. 决策树形成过程中每个节点都要按照步骤2来分裂（很容易理解，如果下一次该节点选出来的那一个属性是刚刚其父节点分裂时用过的属性，则该节点已经达到了叶子节点，无须继续分裂了）。一直到不能够再分裂为止。注意整个决策树形成过程中没有进行剪枝。
4. 按照步骤1~3建立大量的决策树，这样就构成了随机森林了。

优点：

- 对于很多种资料，它可以产生高准确度的分类器。
- 它可以处理大量的输入变量。
- 它可以在决定类别时，评估变量的重要性。
- 在建造森林时，它可以在内部对于一般化后的误差产生不偏差的估计。
- 它包含一个好方法可以估计丢失的资料，并且，如果有很大一部分的资料丢失，仍可以维持准确度。
- 它提供一个实验方法，可以去侦测variable interactions。
- 对于不平衡的分类资料集来说，它可以平衡误差。
- 它计算各例中的亲近度，对于数据挖掘、侦测离群点和将资料可视化非常有用。
- 使用上述。它可被延伸应用在未标记的资料上，这类资料通常是使用非监督式聚类。也可侦测偏离者和观看资料。
- 学习过程是很快速的。

缺点：

- 据观测，如果一些分类/回归问题的训练数据中存在噪音，随机森林中的数据集会出现过拟合的现象。
- 比决策树算法更复杂，计算成本更高。
- 由于其本身的复杂性，它们比其他类似的算法需要更多的时间来训练。

各个平台之间的对比，spark怎么这么拉😭😭😭😭😭😭😭😭😭

而且占用内存巨大，导致之后树个数和深度增加就会OutOfMemory

平台	行数	时间（秒）	RAM（GB）	准确率
scikit-learn	10,000	1.62	0.18	0.933
	100,000	15.33	1.1	0.929
	1,000,000	308.00	9.7	0.936
Spark MLlib	10,000	396.13	16.8	0.924
	100,000	9980.65	134.6	0.926
	1,000,000	NA	OOM	NA
DolphinDB	10,000	0.56	0.39	0.928
	100,000	4.85	3.3	0.928
	1,000,000	82.29	34.2	0.938
XGBoost	10,000	5.44	0.11	0.927
	100,000	70.56	0.48	0.922
	1,000,000	386.26	3.9	0.893

原始版本

```
rf_classifier=RandomForestClassifier(labelCol='is_default',
featuresCol="features",numTrees=numTrees,maxDepth=maxDepth,impurity=impurity,max
Bins=maxBins).fit(train_df)
rf_predictions=rf_classifier.transform(test_df)
evaluator =
BinaryClassificationEvaluator(rawPredictionCol='rawPrediction',labelCol='is_defa
ult')
print('AUC: '+str(numTrees)+str(maxDepth)+impurity+str(maxBins),
evaluator.evaluate(rf_predictions))
```

```
evaluator = BinaryClassificationEvaluator(rawPredictionCol='rawPrediction',labelCol='is_default')
print('AUC: ', evaluator.evaluate(rf_predictions))
```

```
AUC:  0.8131726411882917
```

未调参的随机森林，训练结果比逻辑回归好了一些。

超参数搜索优化

本文选择在随机森林算法中比较重要的几个超参数进行调优，分别是：决策树个数`numTrees`，决策树最大深度`maxDepth`，节点的不纯净度测量`impurity`，构建节点时数据分箱数`maxBins`四种。

超参数	范围
`numTrees`	[20,50,100,150,200,300]
`maxDepth`	[5,10,15,20,30,40,50]
`impurity`	['gini', 'entropy']
`maxBins`	[24,32,40]

```
numTrees_list = [20,50,100,150,200,300]
maxDepth_list = [5,10,15,20,30,40,50]
impurity_list = ['gini', 'entropy']
maxBins_list = [24,32,40]
for numTrees in numTrees_list:
    for maxDepth in maxDepth_list:
        for impurity in impurity_list:
            for maxBins in maxBins_list:
                rf_classifier=RandomForestClassifier(labelCol='is_default',
featuresCol="features",numTrees=numTrees,maxDepth=maxDepth,impurity=impurity,max
Bins=maxBins).fit(train_df)
                rf_predictions=rf_classifier.transform(test_df)
                evaluator =
BinaryClassificationEvaluator(rawPredictionCol='rawPrediction',labelCol='is_defa
ult')
                print('AUC: '+str(numTrees)+str(maxDepth)+impurity+str(maxBins),
evaluator.evaluate(rf_predictions))
```

```
numTrees_list = [20,50,100,150,200,300]
maxDepth_list = [10,15,20,30,40,50]
impurity_list = ['gini', 'entropy']
maxBins_list = [32]
for numTrees in numTrees_list:
    for maxDepth in maxDepth_list:
        for impurity in impurity_list:
            for maxBins in maxBins_list:
                rf_classifier=RandomForestClassifier(labelCol='is_default', featuresCol="features",numTrees=numTrees,
                rf_predictions=rf_classifier.transform(test_df)
                evaluator = BinaryClassificationEvaluator(rawPredictionCol='rawPrediction',labelCol='is_default')
                print('AUC: '+str(numTrees)+str(maxDepth)+impurity+str(maxBins), evaluator.evaluate(rf_predictions))
```

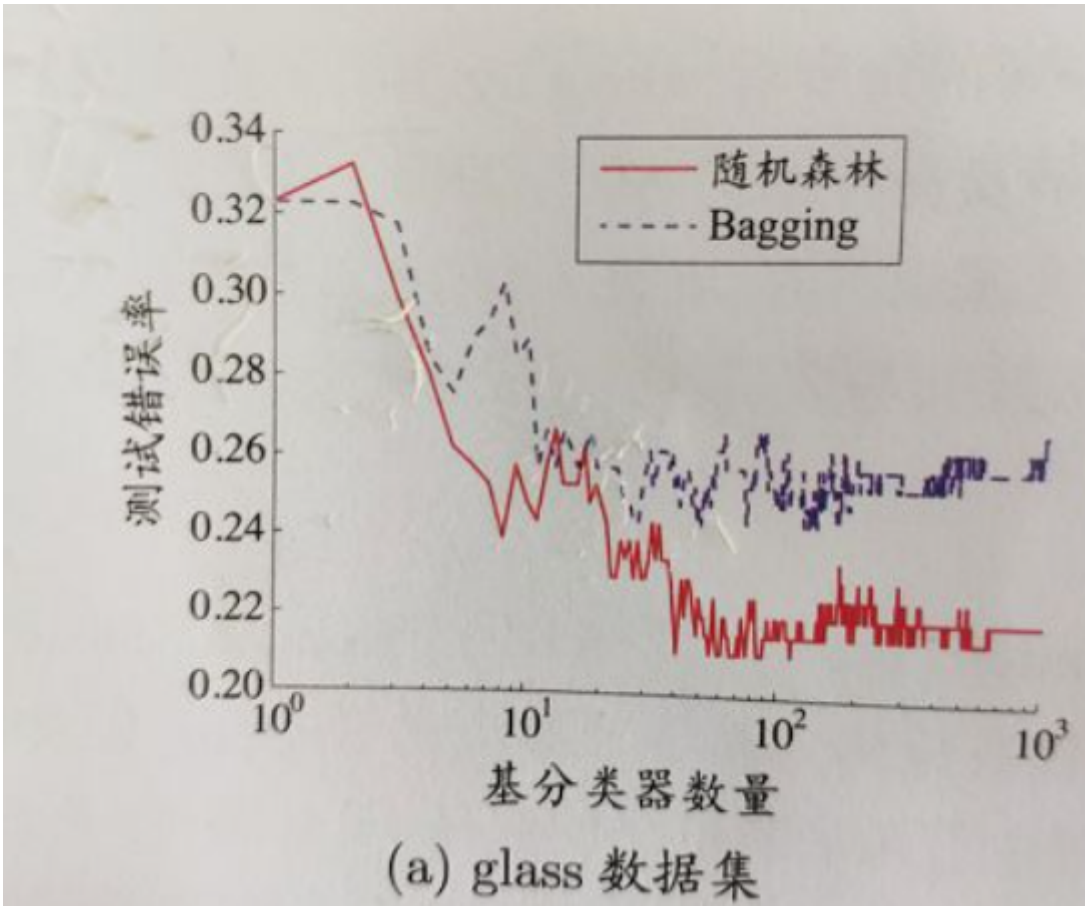
```
21/12/18 21:28:02 WARN DAGScheduler: Broadcasting large task binary with size 1468.7 KiB
21/12/18 21:28:05 WARN DAGScheduler: Broadcasting large task binary with size 2.3 MiB
21/12/18 21:28:07 WARN DAGScheduler: Broadcasting large task binary with size 3.7 MiB
21/12/18 21:28:11 WARN DAGScheduler: Broadcasting large task binary with size 5.8 MiB
21/12/18 21:28:14 WARN DAGScheduler: Broadcasting large task binary with size 1074.2 KiB
21/12/18 21:28:16 WARN DAGScheduler: Broadcasting large task binary with size 8.8 MiB
21/12/18 21:28:19 WARN DAGScheduler: Broadcasting large task binary with size 1474.5 KiB
21/12/18 21:28:22 WARN DAGScheduler: Broadcasting large task binary with size 12.8 MiB
21/12/18 21:28:27 WARN DAGScheduler: Broadcasting large task binary with size 1920.0 KiB
21/12/18 21:28:31 WARN DAGScheduler: Broadcasting large task binary with size 7.6 MiB

AUC: 2015entropy32 0.8626899891879092
```

经过调优，**影响精度的主要的因素**在于`numTrees`和`maxDepth`，在`numTrees=20`和`maxDepth=15` `impurity=entropy` `maxBins=32`时达到0.863

其中对**精度影响最大**的是`maxDepth`，在默认设置为5的情况下AUC为0.82左右，10的时候在0.85左右，15的时候达到最大值，在15附近进行尝试，发现15基本已经达到最佳。

对**精度影响第二大**的是`numTrees`，参与投票的树越多，也越容易得到正确答案。



由图可以看出当基分类器的数目超过一定值时，模型的错误率基本收敛，再增加的基分类器的数目，效果基本不会提升，只会是使的代码变慢。

在树的个数达到20时，已经达到较好效果，增加到30，40时基本在上下震荡，也许已经收敛。

在`impurity`的选择上，发现一个有趣的现象，在树的深度较浅时'gini'的表现优于'entropy'，在树的深度达到10以上时，'entropy'的表现优于'gini'，总体来说差距不大。

在`maxBins`的选择上，同样出现类似现象，当树的深度较浅时，24和40的分箱数优于32，但是在树深度较深的时候，32略微优于其他分箱个数。

总结

总共尝试了三类算法。

第一类：**逻辑回归**

逻辑回归的速度最快，也一如既往的稳定，虚拟机的内存也能够胜任所有的参数组合。优化一尝试正则化提升泛化能力，但是并没有带来提升。优化二尝试修改分类的阈值，提升非常有限，仅有万分之一不到。

最终AUC能达到0.801左右

速度：　☆☆☆☆☆

内存：　☆☆☆☆☆

准确度：☆☆☆

第二类：**多层感知器**

多层感知器对性能要求较高，虚拟机难以胜任基本的要求😭，勉强跑出来的结果仅有0.65左右。

速度：　☆

内存：　☆

准确度：🐻

第三类：**随机森林**

首先尝试了决策树，不调参的效果很差，修改节点最大深度为15时，AUC达到0.756左右。

速度：　☆☆☆

内存：　☆☆☆

准确度：☆☆

随机森林的速度相对较快，并且占用内存也在能接受的程度之内。在优化了决策树个数`numTrees`，决策树最大深度`maxDepth`，节点的不纯净度测量`impurity`，构建节点时数据分箱数`maxBins`四种超参数之后，最终得到了虚拟机能力力所能及的最佳超参数组合。

AUC达到0.863左右。因为虚拟机性能限制，最大深度只能探索到18层，树的数目在深度15层时只能探索到40颗树。假如性能能够优化，AUC提到0.87有可能。

速度：　☆☆

内存：　☆☆

准确度：☆☆☆☆☆