实验4 Spark w/ 个贷违约预测

191840265 吴偲羿

0. Spark环境配置

万能的homebrew倒了！🍺

brew install spark装了个不知道是什么的东西 7.2kb

```
spark                          sparkle
[quinton_541@MacBook-Air-de-541 ~ % brew install spark                          ]
Warning: Treating spark as a formula. For the cask, use homebrew/cask/spark
==> Downloading https://mirrors.tuna.tsinghua.edu.cn/homebrew-bottles/bottles/sp
Already downloaded: /Users/quinton_541/Library/Caches/Homebrew/downloads/b25799e
d97a7386441038dad5c90194888fc3366bd5b52e2f576222fb6f8e5dd--spark-1.0.1.all.bottl
e.tar.gz
==> Pouring spark-1.0.1.all.bottle.tar.gz
🍺  /opt/homebrew/Cellar/spark/1.0.1: 6 files, 7.2KB
```

不管了，这次就老老实实到阿帕奇官网下了个

然后 解压安装

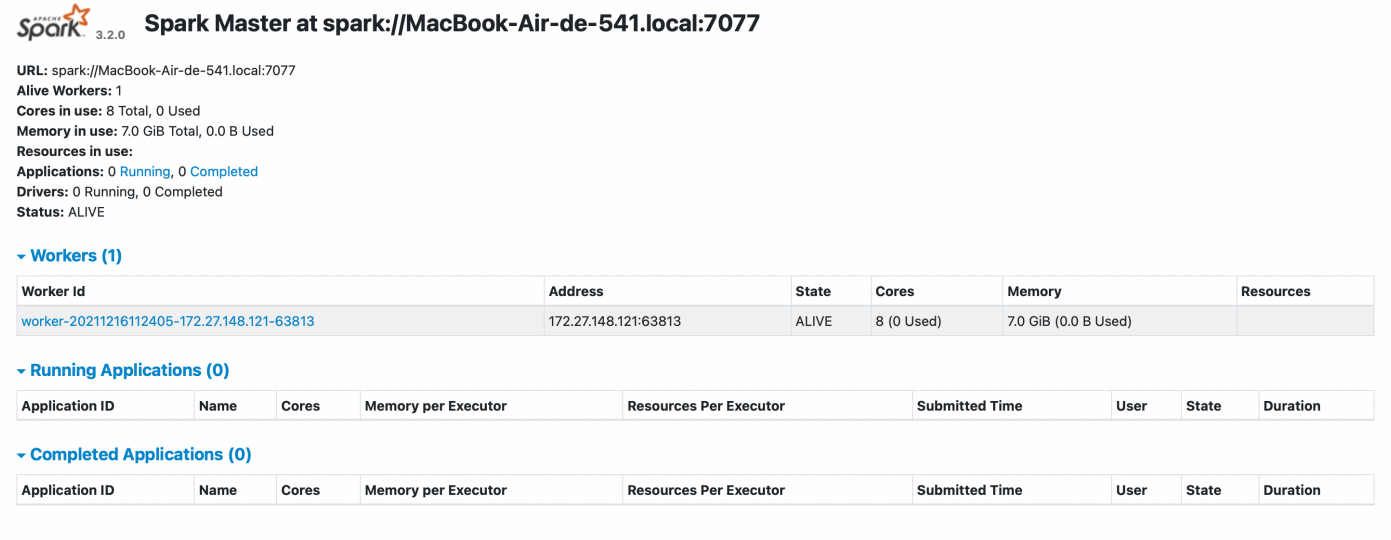
```
quinton_541@MacBook-Air-de-541 bin % spark-shell
21/12/16 11:12:22 WARN Utils: Your hostname, MacBook-Air-de-541.local resolves to a loopback addres
s: 127.0.0.1; using 172.27.148.121 instead (on interface en0)
21/12/16 11:12:22 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
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).
21/12/16 11:12:31 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform...
using builtin-java classes where applicable
Spark context Web UI available at http://172.27.148.121:4040
Spark context available as 'sc' (master = local[*], app id = local-1639624353053).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.2.0
      /_/

Using Scala version 2.12.15 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_301)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

我要赞美spark！赞美spark的话在我的心里，在我的唇上！

啥配置都不要改就能开



我要收回赞美spark的话！因为我白装了！

```
quinton_541@MacBook-Air-de-541 sbin % pyspark
Python 3.8.9 (default, Aug  3 2021, 19:21:54)
[Clang 13.0.0 (clang-1300.0.29.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
21/12/16 11:35:37 WARN Utils: Your hostname, MacBook-Air-de-541.local resolves to a loopback addres
s: 127.0.0.1; using 172.27.148.121 instead (on interface en0)
21/12/16 11:35:37 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
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).
21/12/16 11:35:38 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform...
using builtin-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.2.0
      /_/

Using Python version 3.8.9 (default, Aug  3 2021 19:21:54)
Spark context Web UI available at http://172.27.148.121:4040
Spark context available as 'sc' (master = local[*], app id = local-1639625739420).
SparkSession available as 'spark'.
>>>
```

pip install了一下pyspark 还在纳闷为什么pyspark为什么这么大（）

我还是要赞美spark！因为spark支持python！

至此环境就 很方便的，花了十分钟的，配置完了（在这里继续谴责hbase与永远找不到的regionserver）

1.任务一

我要谴责并呵斥java，一个任务耗的时间比我后面两个任务加起来都多

按照java的劣性加上我对java一丁点也不熟，如果我要把csv中industry那一列提出来，估计得折我一天性命

所以干脆直接python把数据预处理好了喂给java：

```
import pandas as pd
orig_df=pd.read_csv("train_data.csv")
new_df=orig_df['industry']
new_df.to_csv("CondemnJava.csv",index=False,header=False)
```

下面就是一个简单的WordCount了，按照job1:wordcount，job2:排序进行，如下：

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;


import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.map.InverseMapper;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;


public class Loan_Count {
    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>
{
        //实现map()函数
        public void map(Object key, Text value, Context context)throws
IOException,InterruptedException {
            String word = value.toString();
            context.write(new Text(word), new IntWritable(1));
        }
    }
    public static class IntSumReducer extends Reducer <Text, IntWritable, Text,
IntWritable> {
        private IntWritable result = new IntWritable();//实现reduce()函数
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
            int sum = 0;//遍历迭代values，得到同一key的所有value
            for (IntWritable val : values) { sum += val.get(); }
            result.set(sum);//产生输出对<key, value>
            context.write(key, result);
        }
    }

    private static class IntWritableDecreasingComparator extends IntWritable.Comparator
{
        public int compare(WritableComparable a, WritableComparable b) {
            //System.out.println("ss");
            return -super.compare(a, b);
```

```
        }
        public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
            //System.out.println("ss1");
            return -super.compare(b1, s1, l1, b2, s2, l2);
        }
    }

    public static void main(String[] args) throws Exception{
        Configuration config = new Configuration();
        Job job = Job.getInstance(config);
        job.setJobName("Loan_Count");
        job.setJarByClass(Loan_Count.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class);
        Path outPath = new Path("/user/quinton_541/tempDir");
        FileInputFormat.addInputPath(job, new Path("/user/quinton_541/input/"));
        FileOutputFormat.setOutputPath(job, outPath);
        job.waitForCompletion(true);
        Job sortjob = Job.getInstance(config);
        FileInputFormat.addInputPath(sortjob, outPath);
        sortjob.setOutputKeyClass(IntWritable.class);
        sortjob.setOutputValueClass(Text.class);
        sortjob.setInputFormatClass(SequenceFileInputFormat.class);
        sortjob.setMapperClass(InverseMapper.class);
        sortjob.setNumReduceTasks(1);
        sortjob.setSortComparatorClass(IntWritableDecreasingComparator.class);
        outPath= new Path("/user/quinton_541/Mission1_Output");
        FileOutputFormat.setOutputPath(sortjob,outPath);
        sortjob.waitForCompletion(true);
    }
}
```

（后来想了下其实直接用java处理源csv也没啥难度，但是还是要先在外面把第一行header去掉，mapper类改成这样

```
public void map(Object key, Text value, Context context)throws
IOException,InterruptedException {
            String line = value.toString();
            String[] words = line.split(",");
            context.write(new Text(words[10]), new IntWritable(1));
        }
```

执行结果如下：

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

由于value和key已经在job2中的InverseMapper中交换过了，所以输出好像不太符合要求。

理论上再来一个job3再用一次InverseMapper就可以恢复顺序了，可这太烦了，一点也不优雅！

那就再次请出python处理一下吧（py贴贴：

```
f=open("Misson1_Output/part-r-00000",'r')
lines=f.readlines()
f_to_write=open("Mission1_Real_Output",'w')
for line in lines:
    key=line.split("\t")[1].strip()
    value=line.split("\t")[0]
    f_to_write.write("%s %s\n"%(key,value))
```

最终结果

```
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
```

2. 任务二

我要赞美Spark，赞美Spark的有福了！

因为Spark提供了python接口，所以很方便的，在了解了rdd相关操作后很快就写完了任务二（私python本当上手）

基本思路：

1.读取csv文件为rdd

2.剔除首行的headers

3.按逗号分割每一个对象，提取出每一行中“Total_loan”

4.将total_loan的值规约为其所在区间的下限，即1541->1000，54541->54000等

5.执行统计，按照键值排序会，将规约后的区间下限恢复为区间并输出。

完整代码如下：

```
from pyspark import SparkContext
if __name__ == '__main__':
    sc=SparkContext.getOrCreate()
    # 读取文件为rdd对象
    lines=sc.textFile('train_data.csv')
    header = lines.first()  # 第一行 print(header)
    # 剔除csv的首行
    lines = lines.filter(lambda row: row != header)
    # 按逗号分割后提取第三个元素：total_loan
    lines=lines.map(lambda line:line.split(',')[2])
    # int((float(x)//1000)*1000) 提取元素所在区间下限，即1541->1000，54541->54000
    counts = lines.map(lambda x: (int((float(x)//1000)*1000), 1)).reduceByKey(lambda
a,b:a+b)
    #排序
    counts=counts.sortByKey()
    output=counts.collect()
    #统计好后，将区间下限还原为区间：1000->(1000,2000)
    for (item, count) in output: print("(%i,%i),%i"%(item,item+1000,count))
    counts.saveAsTextFile("Misson2_Output")
```

运行结果：

```
(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
(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
```

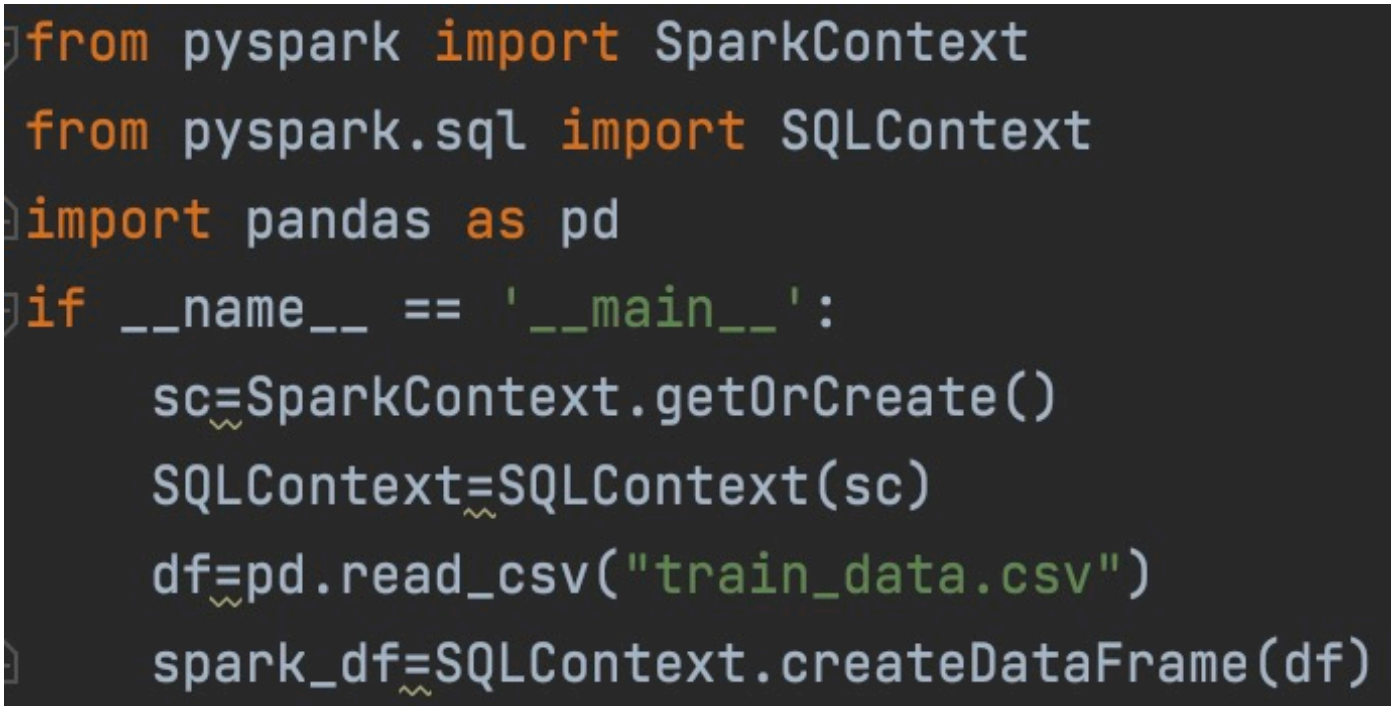
3.任务3

任务三是利用Spark SQL来执行的，因此第一步是需要将数据读入spark的dataframe：

（srds，最后还是转化成pdd啊呸rdd）

```
sc=SparkContext.getOrCreate()
spark=SQLContext(sc)
    #读取数据至spark的dataframe
spark_df=spark.read.format("csv").option('header','true').option('inferScheme','true').
load("train_data.csv")
```

由于有空值，和格式上的一些问题，如果先读入pandas的df再读入spark的df会报错：




```
    raise TypeError(new_msg("Can not merge type %s and %s" % (type(a), type(b))))
TypeError: field work_year: Can not merge type <class 'pyspark.sql.types.DoubleType'> and <class 'pyspark.sql.types.StringType'>
```

1.统计所有用户所在公司类型 employer_type 的数量分布占比情况。

说是数量分布占比，本质上还是一个wordcount，因此首先读取spark_df中的employer_type列转换为rdd，并执行一个简单的wordCount。

注意，像我一样先提取employer_type列作为子数据框时，在rdd.map的时候还是要将employer_type的值提取出来。代码如下：

```
    #执行任务1:统计所有用户所在公司类型 employer_type 的数量分布占比情况。
    #读取employer_type为单独数据框
    employer_type=spark_df.select("employer_type")
    #将employer_type转换为rdd，并提取每一行具体对象的值后进行词频统计
    counts=employer_type.rdd.map(lambda x:"{}".format(x.employer_type))\
                            .map(lambda x:(x,1)).reduceByKey(lambda a,b:a+b)
    output=counts.collect()
    sum=0
    for (item,count) in output: sum+=count
    #写入csv
    with open("Misson3.1_Output.csv", "w") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(["公司类型","类型占比"])
        for (item,count) in output:
            writer.writerow([item,count/sum])
```

结果如下：

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

2.统计每个用户最终须缴纳的利息金额

本来想着直接在dataframe里操作的，后来觉得这是个mapreduce的活，就又转成rdd来做了0v0

思路：把需要的数据"user_id","year_of_loan","monthly_payment","total_loan"从df提取出来转换成rdd，对于rdd中每个对象，计算后整理成（user_id,total_money），提取后输出。

完整代码如下：

```
#执行任务2：统计每个用户最终须缴纳的利息金额
    #在spark数据框中选取所需数据至子数据框

total_money_para=spark_df.select("user_id","year_of_loan","monthly_payment","total_loan
")
    #转化为rdd后按公式计算
    res=total_money_para.rdd.map(lambda x:
(x.user_id,float(x.year_of_loan)*float(x.monthly_payment)*12-float(x.total_loan)))
    output=res.collect()
    with open("Misson3.2_Output.csv", "w") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(["user_id","total_money"])
        for (user_id,total_money) in output:
            writer.writerow([user_id,total_money])
```

部分结果如下：

```
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
```

3.统计工工作年限 work_year 超过 5 年的用户的房贷情况 censor_status 的数量分布占比情况

这玩意比较麻烦，因为work_year的格式不是简单的一个数字，而是“< 1 year”,"x years"和“10+ years”，还有好多空值，为了直观看出这东西的结构，我先对work_year的键值做了一个word_count统计，结果如下：

```
    work_year=spark_df.select("work_year")
    counts=work_year.rdd.map(lambda x:"{}".format(x.work_year))\
                        .map(lambda x:(x,1)).reduceByKey(lambda a,b:a+b)
    output=counts.collect()
    for (a,b) in output: print(a,",",b)
```

```
4 years , 18044
10+ years , 98287
8 years , 13480
7 years , 13281
None , 17428
3 years , 23977
5 years , 19016
1 year , 19799
< 1 year , 24080
9 years , 11330
2 years , 27328
6 years , 13950
```

因此，为了提取出work_year所实际代表的工作年数，我们做如下处理：

1.将空值补充为”0 year“；2.Dataframe转换为rdd; 3.将work_year的字符串按空格分割后提取倒数第二项

这样，work_year就变成了“x”与“10+”，满足条件的即为"10+"与大于5的个位数

将work_year看作字符串，那么字符串长度>1与字符串长度为1并且转化为int后大于5的即为满足条件者

用rdd.filter函数提取即可。

完整代码如下：

```
    #执行任务3：统计工工作年限 work_year 超过 5 年的用户的房贷情况 censor_status 的数量分布占比情况
    #在spark数据框中读取所需数据
    parameter=spark_df.select("user_id","censor_status","work_year")
```

```
    #work_year 空值填补为"0 year"
    parameter=parameter.fillna("0 year")
    #work year形式为"<1 year","x years"与"10+ year"，按空格分割后提取倒数第二位
    #满足条件的为"10+"与大于5的个位数
    res=parameter.rdd.map(lambda x:(x.user_id,x.censor_status,x.work_year.split(" ")
[-2]))\
                                .filter(lambda x:((len(x[2])>1) or (len(x[2])==1 and
int(x[2])>5)))
    output=res.collect()
    with open("Misson3.3_Output.csv", "w") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(["user_id", "censor_status","work_year"])
        for (user_id, censor_status, work_year) in output:
            writer.writerow([user_id, censor_status, work_year])
```

部分结果如下：

```
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+
```

4.任务4

任务4是一个利用pyspark ml库实现machine learning的任务。从提供的数据类别来看，我们的数据有int与string类型（由于太多了，就不在此罗列

我们首先剔除与结果无关的因子：loan_id与user_id

```
spark_df = spark_df.drop('loan_id').drop('user_id')
```

并将所有的str类别的因子，通过StringIndexer转变为Int型变量

```
    StrLabel=
["class","sub_class","work_type","employer_type","issue_date","industry","earlies_credi
t_mon","work_year"]
    for item in StrLabel:
        indexer=StringIndexer(inputCol=item,outputCol="%sIndex"%item)
        spark_df=indexer.fit(spark_df).transform(spark_df)
        spark_df=spark_df.drop(item)
```

到这里还有一个问题，尽管之前所有的类别，长得像Int/Double型，但是通过print(spark_df)来看，这些不过是披着狼皮的羊——还是String类型：

```
DataFrame[total_loan: string, year_of_loan: string, interest: string, monthly_payment:
string, house_exist: string, house_loan_status: string, censor_status: string,
marriage: string, offsprings: string, issue_date: string, use: string, post_code:
string, region: string, debt_loan_ratio: string, del_in_18month: string, scoring_low:
string, scoring_high: string, pub_dero_bankrup: string, early_return: string,
early_return_amount: string, early_return_amount_3mon: string, recircle_b: string,
recircle_u: string, initial_list_status: string, title: string, policy_code: string,
f0: string, f1: string, f2: string, f3: string, f4: string, f5: string, is_default:
string, classIndex: double, sub_classIndex: double, work_typeIndex: double,
employer_typeIndex: double, industryIndex: double, earlies_credit_monIndex: double,
work_yearIndex: double]
```

因此首先通过.cast(DoubleType)全转换为Double：

```
#将长得像Double但实际上还是String的变量转换为Double
    for item in spark_df.columns:
        spark_df=spark_df.withColumn(item,spark_df[item].cast(typ.DoubleType()))
```

至此，数据处理完毕，现在数据框展示如下（部分）：

total_loan	year_of_loan	interest	monthly_payment	house_exist	house_loan_status	censor_status	marriage	offsprings	use	post_code	region	debt_loan_ratio	del_in_18month	s
12000.0	5.0	11.53	264.1	0.0	0.0	2.0	0.0	0.0	0.0	814.0	4.0	5.07	1.0	
8000.0	3.0	13.98	273.35	0.0	1.0	2.0	1.0	3.0	2.0	240.0	21.0	15.04	0.0	
20000.0	5.0	17.99	507.76	0.0	0.0	1.0	0.0	0.0	0.0	164.0	20.0	17.38	1.0	
10700.0	3.0	10.16	346.07	2.0	0.0	2.0	0.0	0.0	4.0	48.0	10.0	27.87	0.0	
8000.0	3.0	8.24	251.58	1.0	2.0	0.0	0.0	0.0	4.0	122.0	9.0	3.47	0.0	
28000.0	3.0	15.59	978.74	2.0	0.0	2.0	0.0	0.0	0.0	149.0	22.0	24.33	0.0	
6000.0	3.0	7.89	187.72	0.0	1.0	0.0	0.0	0.0	0.0	634.0	32.0	8.43	0.0	
20000.0	3.0	12.79	671.86	0.0	0.0	2.0	0.0	0.0	0.0	197.0	4.0	19.48	0.0	
9450.0	3.0	13.11	318.91	0.0	0.0	0.0	1.0	0.0	4.0	19.0	14.0	18.64	0.0	
4500.0	3.0	5.32	135.52	0.0	0.0	0.0	1.0	0.0	9.0	468.0	0.0	7.4	0.0	
21850.0	3.0	11.44	719.91	0.0	0.0	2.0	1.0	3.0	0.0	466.0	36.0	27.67	0.0	
10500.0	3.0	10.99	343.71	1.0	1.0	1.0	1.0	0.0	0.0	33.0	22.0	17.49	0.0	
22000.0	3.0	15.61	769.23	0.0	0.0	1.0	1.0	0.0	0.0	765.0	39.0	27.49	0.0	
9600.0	3.0	8.18	301.63	1.0	1.0	1.0	0.0	0.0	4.0	157.0	8.0	18.73	0.0	

f0	f1	f2	f3	f4	f5	is_default	classIndex	sub_classIndex	work_typeIndex	issue_dateIndex	employer_typeIndex	industryIndex	earlies_credit_monIndex	work_yearIndex
1.0	0.0	8.0	17.0	8.0	1.0	1.0	0.0	2.0	1.0	13.0	0.0	9.0	335.0	7.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	6.0	0.0	98.0	0.0	11.0	248.0	0.0
6.0	0.0	10.0	8.0	3.0	0.0	0.0	3.0	13.0	2.0	18.0	2.0	5.0	90.0	0.0
3.0	0.0	4.0	11.0	6.0	0.0	0.0	0.0	8.0	1.0	52.0	0.0	1.0	38.0	1.0
3.0	0.0	8.0	6.0	4.0	1.0	0.0	0.0	8.0	0.0	45.0	1.0	0.0	36.0	5.0
3.0	0.0	6.0	10.0	3.0	1.0	1.0	1.0	9.0	1.0	18.0	3.0	2.0	44.0	0.0
3.0	0.0	13.0	5.0	5.0	1.0	0.0	2.0	10.0	1.0	2.0	1.0	5.0	5.0	9.0
8.0	0.0	3.0	12.0	8.0	0.0	0.0	1.0	0.0	4.0	19.0	2.0	0.0	59.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	2.0	69.0	1.0	5.0	45.0	1.0
2.0	0.0	9.0	8.0	2.0	0.0	0.0	2.0	14.0	0.0	44.0	5.0	4.0	189.0	0.0
3.0	0.0	14.0	6.0	2.0	0.0	0.0	0.0	1.0	1.0	24.0	4.0	6.0	326.0	0.0
9.0	0.0	3.0	15.0	8.0	0.0	0.0	0.0	1.0	1.0	7.0	1.0	5.0	10.0	3.0
3.0	0.0	5.0	39.0	19.0	0.0	0.0	3.0	12.0	0.0	17.0	1.0	4.0	145.0	4.0
8.0	0.0	4.0	8.0	3.0	0.0	1.0	0.0	8.0	1.0	9.0	0.0	0.0	57.0	2.0
7.0	0.0	5.0	17.0	8.0	0.0	0.0	1.0	7.0	3.0	45.0	3.0	3.0	157.0	6.0
6.0	0.0	7.0	18.0	8.0	0.0	1.0	4.0	21.0	0.0	9.0	0.0	6.0	331.0	10.0
4.0	0.0	12.0	8.0	4.0	0.0	1.0	1.0	6.0	0.0	6.0	0.0	1.0	182.0	0.0
6.0	0.0	5.0	8.0	6.0	2.0	0.0	1.0	4.0	0.0	23.0	4.0	3.0	146.0	0.0
8.0	0.0	5.0	17.0	10.0	4.0	0.0	1.0	7.0	3.0	24.0	0.0	11.0	109.0	0.0

我们提取除了is_default外的所有参数形成features列向量，并和is_default一同提取出来，作为我们的测试模型：

```
#将除了is_default的features列转成向量,并加入df中
    cols=spark_df.columns
    cols.remove("is_default")
    ass = VectorAssembler(inputCols=cols, outputCol="features")
    spark_df=ass.transform(spark_df)
    #选取features与label形成清晰的feature-label模型
    model_df = spark_df.select(["features","is_default"])
```

按照8:2的比例拆分训练集和测试集，随机数种子取541：

```
#按8：2的比例，随机数种子541 分割训练集与测试集
    train_df,test_df=model_df.randomSplit([0.8,0.2],seed=541)
```

至此，模型的数据选择结束，下面开始建立模型

训练模型选取：

由于本题本质上还是一个二元类别识别的问题，模型的选取与训练模型的建立都比较方便，加之pyspark.ml.classification提供了太多的分类器供选择，因此随便找了四个：随机森林（RandomForest）逻辑回归（Logistic Regression）决策树（DecisionTree Classifier）和支持向量机（SVM）。

赞美Spark ML！代码简洁明了：

```
    #随机森林
    rf = cl.RandomForestClassifier(labelCol="is_default", numTrees=128,
maxDepth=9).fit(train_df)
    res = rf.transform(test_df)
    rf_auc = ev.BinaryClassificationEvaluator(labelCol="is_default").evaluate(res)
    print("随机森林预测准确率为：%f"%rf_auc)
    #逻辑回归
    log_reg = cl.LogisticRegression(labelCol='is_default').fit(train_df)
    res = log_reg.transform(test_df)
    log_reg_auc=ev.BinaryClassificationEvaluator(labelCol="is_default").evaluate(res)
    print("逻辑回归预测准确率为：%f" % log_reg_auc)
```

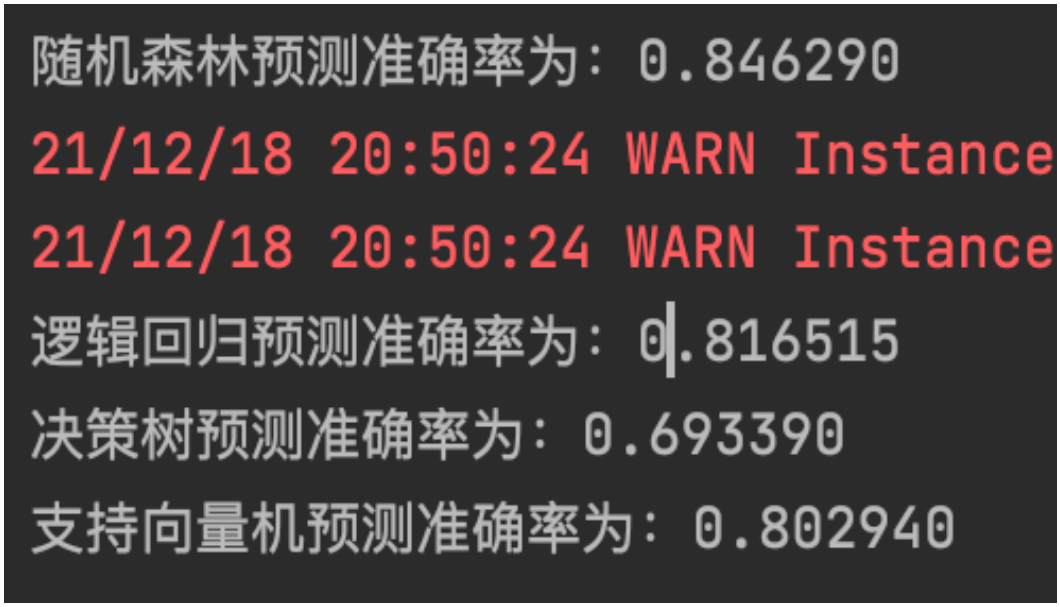
```
#决策树
DTC=cl.DecisionTreeClassifier(labelCol='is_default').fit(train_df)
res=DTC.transform(test_df)
DTC_auc=ev.BinaryClassificationEvaluator(labelCol="is_default").evaluate(res)
print("决策树预测准确率为：%f" % DTC_auc)
#支持向量机
SVM=cl.LinearSVC(labelCol='is_default').fit(train_df)
res=SVM.transform(test_df)
SVM_auc=ev.BinaryClassificationEvaluator(labelCol="is_default").evaluate(res)
print("支持向量机预测准确率为：%f" % SVM_auc)
```

结果如下：

更改训练测试集为7:3与6:4，结果如下：

7:3:

6:4



我们发现，

1.由于数据量比较大（30万条），训练集不管是占比为0.7或0.9，训练量都很大，模型已经成熟，因此训练/测试集合的大小在准确率上的反应不大，甚至出现了训练集越小，准确率反而变大的情况

2.整体来说在本次模型中，模型预测准确率按下列排序：

$$RF > Log\ Reg > SVM > Decision\ Tree$$

至于为什么不用9:1...是因为提示下列错误：

```
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

> 今天出现了一个很奇怪的异常：java.lang.OutOfMemoryError: GC overhead limit exceeded ，超出了GC开销限制。科普了一下，这个是JDK6新添的错误类型。是发生在GC占用大量时间为释放很小空间的时候发生的，是一种保护机制。一般是因为堆太小，导致异常的原因：没有足够的内存。
>
> Sun 官方对此的定义：超过98%的时间用来做GC并且回收了不到2%的堆内存时会抛出此异常。
>
> 为什么会出现这个问题呢？经过反复测试发现，下载数据时文件大小超过某一峰值是会报这个错误。原因是在页面点击下载时，在数据库查询了很庞大的数据量，导致内存使用增加，才会出现这个问题。

hmm...看起来当时买8G的内存确实是个错误

解决方案：换电脑（）

当然，作为一个机器学习任务，这里的优化空间太大了。

对于随机森林来说，这里的参数设置可以进一步探索，优化：

```
rf = cl.RandomForestClassifier(labelCol="is_default", numTrees=128, maxDepth=9).fit(train_df)
```

同时，提供的30w条数据中的噪音数据也可以进一步处理等等等等

但是总体来说，85%的正确率已经是非常可以接受的一个结果了。

完整代码如下：

```
import pyspark.sql.types as typ
import pyspark.ml.classification as cl
import pyspark.ml.evaluation as ev
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark import SparkContext
from pyspark.sql import SQLContext
if __name__ == '__main__':
    sc = SparkContext.getOrCreate()
    spark = SQLContext(sc)
    spark_df = spark.read.format("csv").option('header', 'true').option('inferScheme',
'true').load("train_data.csv")
    #删除无关数据
    spark_df = spark_df.drop('loan_id').drop('user_id')
    spark_df = spark_df.fillna("0")
    #将长得像string的类型变量通过映射转换为Double
    StrLabel=
["class","sub_class","work_type","issue_date","employer_type","industry","earlies_credi
t_mon","work_year"]
    for item in StrLabel:
        indexer=StringIndexer(inputCol=item,outputCol="%sIndex"%item)
        spark_df=indexer.fit(spark_df).transform(spark_df)
        spark_df=spark_df.drop(item)
    #将长得像Double但实际上还是String的变量转换为Double
    for item in spark_df.columns:
        spark_df=spark_df.withColumn(item,spark_df[item].cast(typ.DoubleType()))
    #将除了is_default的features列转成向量,并加入df中
    cols=spark_df.columns
    cols.remove("is_default")
    ass = VectorAssembler(inputCols=cols, outputCol="features")
    spark_df=ass.transform(spark_df)
    #选取features与label形成清晰的feature-label模型
    model_df = spark_df.select(["features","is_default"])
    #按8：2的比例，随机数种子541 分割训练集与测试集
    train_df,test_df=model_df.randomSplit([0.8,0.2],seed=541)
    #随机森林
    rf = cl.RandomForestClassifier(labelCol="is_default", numTrees=128,
maxDepth=9).fit(train_df)
    res = rf.transform(test_df)
    rf_auc = ev.BinaryClassificationEvaluator(labelCol="is_default").evaluate(res)
    print("随机森林预测准确率为：%f"%rf_auc)
    #逻辑回归
    log_reg = cl.LogisticRegression(labelCol='is_default').fit(train_df)
    res = log_reg.transform(test_df)
    log_reg_auc=ev.BinaryClassificationEvaluator(labelCol="is_default").evaluate(res)
```

```
print("逻辑回归预测准确率为：%f" % log_reg_auc)
#决策树
DTC=cl.DecisionTreeClassifier(labelCol='is_default').fit(train_df)
res=DTC.transform(test_df)
DTC_auc=ev.BinaryClassificationEvaluator(labelCol="is_default").evaluate(res)
print("决策树预测准确率为：%f" % DTC_auc)
#支持向量机
SVM=cl.LinearSVC(labelCol='is_default').fit(train_df)
res=SVM.transform(test_df)
SVM_auc=ev.BinaryClassificationEvaluator(labelCol="is_default").evaluate(res)
print("支持向量机预测准确率为：%f" % SVM_auc)
```

5.总结

Spark使用起来还是非常非常方便的，至少最耗时间的环境配置部分不用浪费太多的时间（在这里继续谴责hbase与永远找不到的regionserver），由于支持python，在写代码的过程中也相对是比较顺利的。

可以考虑一下在金工量化上的运用，对于GB、TB级别的高频交易数据，Spark可以便捷，有效的提高程序执行效率（真正的金融大数据）