

# 大数据实验四报告

徐冰清 191098273

实验环境：IDEA & BDkit

## 大数据实验四报告

### Task1

框架

job 1

job 2

### Task2

### Task3

统计所有用户所在公司类型 employer\_type 的数量分布占比情况。

统计每个用户最终须缴纳的利息金额

统计工作年限 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

### Task4

数据读取

数据预处理

评估指标

决策树

逻辑回归

随机森林

遇到的困难

实验感想

## Task1

编写 MapReduce 程序，统计每个工作领域 industry 的网贷记录的数量，并按数量从大到小进行排序。

### 框架

总体上看，使用两个job完成实验。第一个job统计网贷记录的数量，第二个job按数量从大到小进行排序。

```
1 Job job = Job.getInstance(conf, "word count");
2 job.setJarByClass(wordCount.class);
3 job.setMapperClass(TokenizerMapper.class);
4 job.setCombinerClass(IntSumReducer.class);
5 job.setReducerClass(IntSumReducer.class);
6 job.setOutputKeyClass(Text.class);
7 job.setOutputValueClass(IntWritable.class);
8 List<String> otherArgs = new ArrayList<String>();
9 for (int i = 0; i < remainingArgs.length; ++i) {
10     otherArgs.add(remainingArgs[i]);
11 }
12 FileInputFormat.addInputPath(job, new Path(otherArgs.get(0)));
13 Path tempPath = new Path("tmp-count" );
```

```

14 FileOutputFormat.setOutputPath(job, new Path(otherArgs.get(1)));
15 job.setOutputFormatClass(SequenceFileOutputFormat.class);
16 System.out.print("ok1");
17 if(!job.waitForCompletion(true)){
18     Job sortjob = Job.getInstance(conf, "sort count");
19     sortjob.setJarByClass(WordCount.class);
20     FileInputFormat.addInputPath(sortjob, tempPath);
21     sortjob.setInputFormatClass(SequenceFileInputFormat.class);
22     sortjob.setMapperClass(InverseMapper.class);
23     sortjob.setReducerClass(SortReducer.class);
24     FileOutputFormat.setOutputPath(sortjob, new Path(otherArgs.get(1)));
25     sortjob.setOutputKeyClass(IntWritable.class);
26     sortjob.setOutputValueClass(Text.class);
27
28     sortjob.setSortComparatorClass(IntWritableDecreasingComparator.class);
29     FileSystem.get(conf).deleteOnExit(tempPath);
30 }

```

## job 1

**TokenizerMapper**: 主要操作为在map函数中，忽略表格第一行标题，选取industry记录作为key、1为value写入。

```

1 @Override
2     public void map(Object key, Text value, Context context)
3         throws IOException, InterruptedException {
4         if (!key.toString().equals("0")){
5             String[] line = value.toString().split(",");
6             context.write(new Text(line[10]), one);
7         }
8     }

```

**IntSumReducer**: 对同一个key下的value求和，无特殊操作。

```

1 @Override
2     public void reduce(Text key, Iterable<IntWritable> values,
3         Context context) throws IOException,
4         InterruptedException {
5         int sum = 0;
6         for (IntWritable val : values) {
7             sum += val.get();
8         }
9         result.set(sum);
10        context.write(key, result);

```

## job 2

**InverseMapper**: 调换key和value的位置。

**SortReducer**: 对value进行排序，设置输出格式。

```
1 public static class SortReducer extends Reducer<IntWritable, Text, Text,  
NullWritable>{  
2     private Text result = new Text();  
3  
4     @Override  
5     protected void reduce(IntWritable key, Iterable<Text> values, Context  
context) throws IOException, InterruptedException{  
6         for(Text val: values){  
7             result.set(val.toString());  
8             context.write(new Text(result+" "+key), NullWritable.get());  
9         }  
10    }  
11 }  
12  
13 private static class IntWritableDecreasingComparator extends  
IntWritable.Comparator {  
14     public int compare(WritableComparable a, WritableComparable b) {  
15         return -super.compare(a, b);  
16     }  
17  
18     public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {  
19         return -super.compare(b1, s1, l1, b2, s2, l2);  
20     }  
21 }
```

运行结果如下

1	金融业 48216
2	电力、热力生产供应业 36048
3	公共服务、社会组织 30262
4	住宿和餐饮业 26954
5	文化和体育业 24211
6	信息传输、软件和信息技术服务业 24078
7	建筑业 20788
8	房地产业 17990
9	交通运输、仓储和邮政业 15028
10	采矿业 14793
11	农、林、牧、渔业 14758
12	国际组织 9118
13	批发和零售业 8892
14	制造业 8864
15	

## Task2

编写 Spark 程序，统计网络信贷产品记录数据中所有用户的贷款金额 total\_loan 的分布情况。

思路：

数据读取，用“，”对每一行数据进行分隔，选取loan\_id。

```
1 sc = SparkContext(appName="LoanDistribution")
2 lines = sc.textFile("train_data.csv")
3 data = lines.filter(lambda line: line.split(',')[0]!="loan_id")
```

因间隔1000，故通过除以1000的余数确定分组。先map形成key-value对，后reduce统计频数。

```
1 output = data.map(lambda line: (int(int(line.split(',')[2].split('.')[0])/1000),
1)).reduceByKey(lambda a, b: a+b).sortBy(lambda x: x[0]).collect()
```

调整格式，写入文件夹

```
1 with open("LoanDistribution.txt", "w") as f:
2     for (scale, count) in output:
3         f.write("({},{})\n".format(scale*1000, (scale+1)*1000, count))
```

运行结果如下：



```
LoanDistribution.txt
((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)
```

## Task3

统计所有用户所在公司类型 **employer\_type** 的数量分布占比情况。

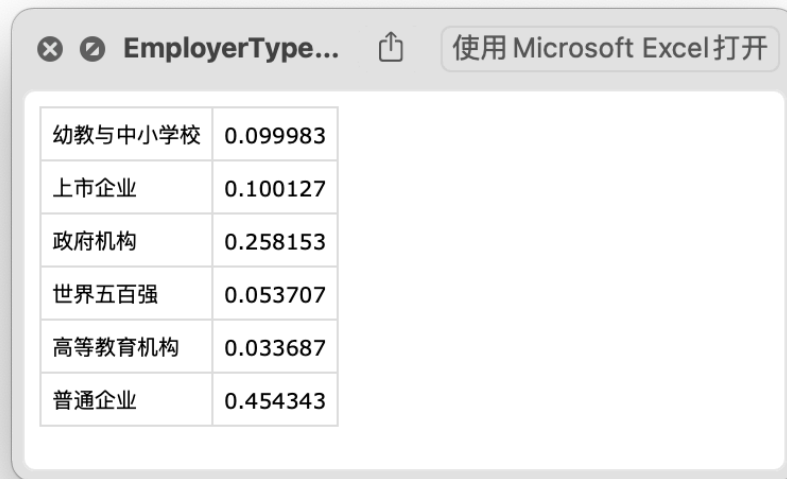
采用SparkSession打开文件：

```
1 spark = SparkSession.builder.appName("task3").getOrCreate()
2 df = spark.read.options(header='True', inferSchema='True',
  delimiter=',').csv(r'train_data.csv')
```

使用groupby对公司类型进行统计；对所有员工数量进行统计；二者相除即公司类型占比。

```
1 num = df.count()
2 df1 = df.groupBy('employer_type').count()
3 df2 = df1.withColumn('count', df1['count']/num)
```

计算结果如下：



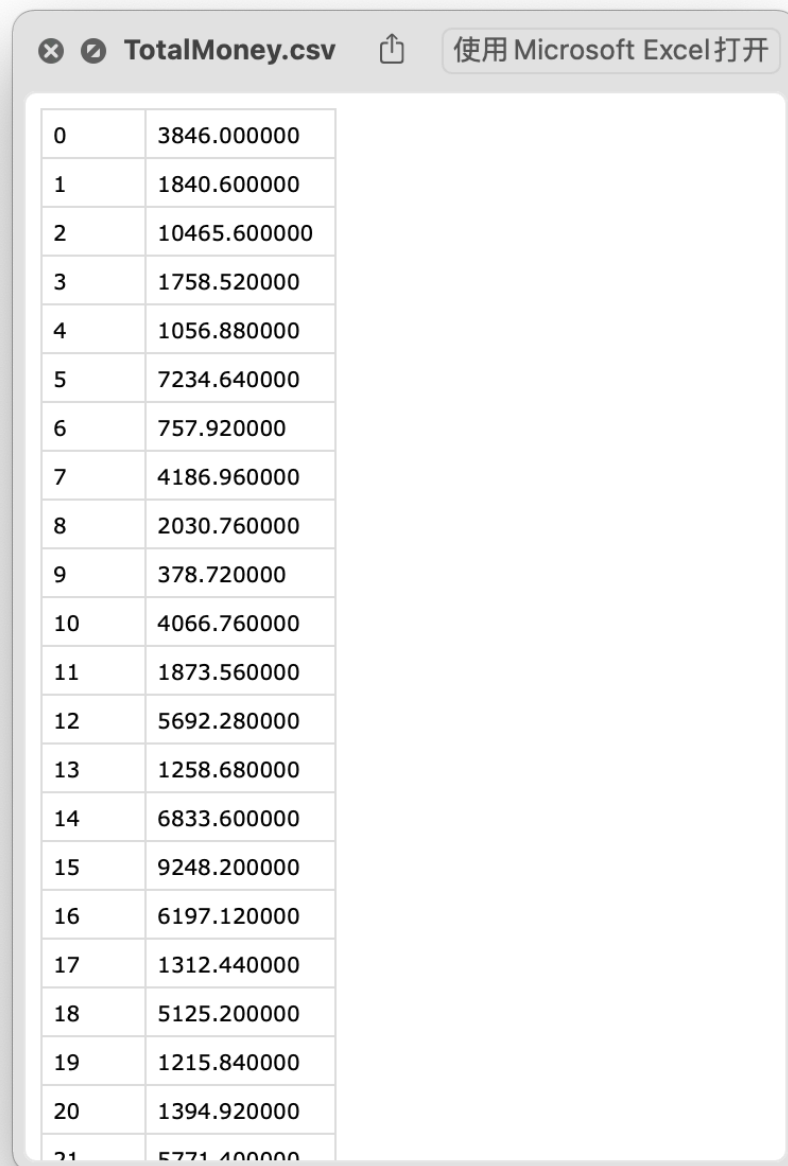
幼教与中小学校	0.099983
上市企业	0.100127
政府机构	0.258153
世界五百强	0.053707
高等教育机构	0.033687
普通企业	0.454343

## 统计每个用户最终须缴纳的利息金额

使用withColumn对各列进行四则运算，得出每个用户最终须缴纳的利息金额，选取计算结果列和id列存入csv。

```
1 df3 = df.select("user_id", "total_loan", "year_of_loan", "monthly_payment")
2 df_expand = df3.withColumn("total_money",
3   df3['year_of_loan']*df3['monthly_payment']*12-df3['total_loan'])
4 df4 = df_expand.select("user_id", "total_money")
5 output2 = df4.collect()
```

计算结果如下：



0	3846.000000
1	1840.600000
2	10465.600000
3	1758.520000
4	1056.880000
5	7234.640000
6	757.920000
7	4186.960000
8	2030.760000
9	378.720000
10	4066.760000
11	1873.560000
12	5692.280000
13	1258.680000
14	6833.600000
15	9248.200000
16	6197.120000
17	1312.440000
18	5125.200000
19	1215.840000
20	1394.920000
21	5771.400000

## 统计工作年限 `work_year` 超过 5 年的用户的房贷情况 `loan_status` 的数量分布占比情况

首先删除工作年限不满五年的数据，接着选取需要的id、房贷情况、工作年限这三列。再对工作年限进行转换，使用withColumn替代，消除特殊符号和years。

```

1 df_filtered = df.where(
2     (col('work_year') != '< 1 year') &
3     (col('work_year') != '1 year') &
4     (col('work_year') != '2 years') &
5     (col('work_year') != '3 years') &
6     (col('work_year') != '4 years') &
7     (col('work_year') != '5 years'))
8 df5 = df_filtered.select("user_id", "censor_status", "work_year")
9 df6 = df5.withColumn('work_year', F.when(df5.work_year == '10+ years',
10 '10').when(df5.work_year == '9 years', '9').when(df5.work_year == '8 years',
11 '8').when(df5.work_year == '7 years', '7').when(df5.work_year == '6 years', '6'))
10 output3 = df6.collect()
11 with open(r"CensorStatus.csv",

```

计算结果如下：

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

## Task4



根据给定的数据集，基于 Spark MLlib 或者 Spark ML 编写程序预测有可能违约的借贷人，并评估实验结果的准确率。

## 数据读取

在读 csv 时，设置 `inferSchema='True'` 模式推理，使 spark 能够推断出数据类型。

```
1 sc = SparkContext(appName="LoanDistribution")
2 spark = SparkSession.builder.appName("task4").getOrCreate()
3 df = spark.read.options(header='True', inferSchema='True',
    delimiter=',').csv(r'train_data.csv')
```

## 数据预处理

### 1. 填充缺失值

对于数值变量，填入数字-1；对于 string，填入字符串-1。

```
1 df = df.na.fill(-1)
2 df = df.na.fill('-1')
```

### 2. 将 string 转为数字。

1. 运用 StringIndexer 对分类数据进行处理。

```
1 # class
2 class_stringIndexer =
    StringIndexer(inputCol="class", outputCol="class_class", stringOrderType="frequencyD
    esc")
3 df = class_stringIndexer.fit(df).transform(df)
4 class_encoder =
    OneHotEncoder(inputCol="class_class", outputCol="class_onehot").setDropLast(False)
5 df = class_encoder.fit(df).transform(df)
6
7 # sub_class
8 sub_class_stringIndexer =
    StringIndexer(inputCol="sub_class", outputCol="sub_class_class", stringOrderType="fr
    equencyDesc")
9 df = sub_class_stringIndexer.fit(df).transform(df)
10 sub_class_encoder =
    OneHotEncoder(inputCol="sub_class_class", outputCol="sub_class_onehot").setDropLast
    (False)
11 df = sub_class_encoder.fit(df).transform(df)
12
13 # work_type
14 work_type_stringIndexer =
    StringIndexer(inputCol="work_type", outputCol="work_type_class", stringOrderType="fr
    equencyDesc")
```

```

15 df = work_type_stringIndexer.fit(df).transform(df)
16 work_type_encoder =
  OneHotEncoder(inputCol="work_type_class",outputCol="work_type_onehot").setDropLast
  (False)
17 df = work_type_encoder.fit(df).transform(df)
18
19 #employer_type
20 employer_type_stringIndexer =
  StringIndexer(inputCol="employer_type",outputCol="employer_type_class",stringOrder
  Type="frequencyDesc")
21 df = employer_type_stringIndexer.fit(df).transform(df)
22 employer_type_encoder =
  OneHotEncoder(inputCol="employer_type_class",outputCol="employer_type_onehot").set
  DropLast(False)
23 df = employer_type_encoder.fit(df).transform(df)
24
25 # industry
26 industry_stringIndexer =
  StringIndexer(inputCol="industry",outputCol="industry_class",stringOrderType="freq
  uencyDesc")
27 df = industry_stringIndexer.fit(df).transform(df)
28 industry_encoder =
  OneHotEncoder(inputCol="industry_class",outputCol="industry_onehot").setDropLast(F
  alse)
29 df = industry_encoder.fit(df).transform(df)

```

2. 将含义不明的数据转化成数字。对work\_year，使用替换去除years和符号，再转化为int类型。

```

1 df = df.withColumn("work_year", F.when(df.work_year == '10+ years',
  '10').when(df.work_year == '9 years', '9').when(df.work_year == '8 years',
  '8').when(df.work_year == '7 years', '7').when(df.work_year == '6 years',
  '6').when(df.work_year == '5 years', '5').when(df.work_year == '4 years',
  '4').when(df.work_year == '3 years', '3').when(df.work_year == '2 years',
  '2').when(df.work_year == '1 year', '1').when(df.work_year == '< 1 year', '0'))
2 df = df.withColumn("work_year", df['work_year'].cast('int'))

```

3. 将issue\_date日期转换为对应的数字，时间越早数字越小。将earlies\_credit\_mon日期转换为对应的数字，时间越晚数字越小。

```

1 #issue_date
2 @F.udf(returnType = IntegerType())
3 def process_issue_data(x):
4     result = 0
5     if x == "-1":
6         result = 0
7     else:
8         result = (int(x[0:4])-2000)*12 + int(x[5:7])

```

```

9     return result
10 df = df.withColumn("issue_date", process_issue_data(F.col("issue_date")))
11
12 # earlies_credit_mon
13 @F.udf(returnType = IntegerType())
14 def process_credit_mon(x):
15     result = 0
16     if x == "-1":
17         result = 0
18     else:
19         month = x[0:3]
20         day = int(x[-4:])
21         if month == 'Jan':
22             result = (2021 - day)*12 + 11
23         elif month == 'Feb':
24             result = (2021 - day)*12 + 10
25         elif month == 'Mar':
26             result = (2021 - day)*12 + 9
27         elif month == 'Apr':
28             result = (2021 - day)*12 + 8
29         elif month == 'May':
30             result = (2021 - day)*12 + 7
31         elif month == 'Jun':
32             result = (2021 - day)*12 + 6
33         elif month == 'Jul':
34             result = (2021 - day)*12 + 5
35         elif month == 'Aug':
36             result = (2021 - day)*12 + 4
37         elif month == 'Sep':
38             result = (2021 - day)*12 + 3
39         elif month == 'Oct':
40             result = (2021 - day)*12 + 2
41         elif month == 'Nov':
42             result = (2021 - day)*12 + 1
43         else:
44             result = (2021 - day)*12
45     return result
46
47 df = df.withColumn("earlies_credit_mon",
48 process_credit_mon(F.col("earlies_credit_mon")))
49 df.select('earlies_credit_mon').show(5)

```

经过这些处理，可以不遗漏地得到所有有效信息。我们删除无效的数据，留下这些列做为特征：['total\_loan', 'year\_of\_loan', 'interest', 'monthly\_payment', 'work\_year', 'house\_exist', 'house\_loan\_status', 'censor\_status', 'marriage', 'offsprings', 'issue\_date', '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', 'earlies\_credit\_mon', 'title',

```
'policy_code', 'f0', 'f1', 'f2', 'f3', 'f4', 'f5', 'class_class', 'sub_class_class',  
'work_type_class', 'employer_type_class', 'industry_class']
```

```
1 df.drop(['loan_id', 'user_id', 'class', 'sub_class', 'work_type', 'employer_type',  
  'industry', 'class_onehot', 'sub_class_onehot', 'work_type_onehot',  
  'employer_type_onehot', 'industry_onehot'])
```

#### 4. 特征集成

```
1 df_feature = df.drop('is_default')  
2 assembler = VectorAssembler(inputCols=df_feature.columns, outputCol="features")  
3 data = assembler.transform(df)
```

#### 5. 划分训练集与测试集

```
1 df_train, df_test = data.randomSplit([0.8, 0.2])
```

## 评估指标

采取精确率、召回率、F1、准确率、auc这5个指标对机器学习效果做出判断。

```
1 def cal_evaluate(predictions):  
2     print('=====')  
3     tp = predictions[(predictions.is_default == 1) & (predictions.prediction ==  
4     1)].count()  
5     tn = predictions[(predictions.is_default == 0) & (predictions.prediction ==  
6     0)].count()  
7     fp = predictions[(predictions.is_default == 0) & (predictions.prediction ==  
8     1)].count()  
9     fn = predictions[(predictions.is_default == 1) & (predictions.prediction ==  
10    0)].count()  
11    print(tp, tn, fp, fn)  
12    try:  
13        precision = float(tp) / float(tp + fp)  
14    except:  
15        precision = 0  
16    try:  
17        recall = float(tp) / float(tp + fn)  
18    except:  
19        recall = 0  
20    try:  
21        f1 = 2 * precision * recall / (precision + recall)  
22    except:  
23        f1 = 0
```

```

20     acc = predictions[predictions.is_default ==
predictions.prediction].count()/predictions.count()
21     auc =
BinaryClassificationEvaluator(labelCol='is_default').evaluate(predictions)
22     print("精确率: ", precision)
23     print("召回率: ", recall)
24     print("F1分数: ", f1)
25     print("准确率: ", acc)
26     print("auc: ", auc)
27     return 1

```

## 决策树

```

1 dt = DecisionTreeClassifier(labelCol="is_default").fit(df_train)
2 predictions_dt = dt.transform(df_test)
3 cal_evaluate(predictions_dt)

```

评价指标为

精确率: 0.6276339070291137  
召回率: 0.32261264985531213  
F1分数: 0.4261686325906509  
准确率: 0.824951273551117  
auc: 0.6896668886716617

## 逻辑回归

```

1 lr = LogisticRegression(regParam=0.01, labelCol='is_default').fit(df_train)
2 predictions_lr = lr.transform(df_test)
3 cal_evaluate(predictions_lr)

```

评价指标为

精确率: 0.6626301253452305  
召回率: 0.2569404399044402  
F1分数: 0.3702956191380743  
准确率: 0.823699518032242  
auc: 0.814343835958197

## 随机森林

```
1 rf = RandomForestClassifier(labelCol='is_default', maxDepth=7, maxBins=700,
2   numTrees=50).fit(df_train)
3 predictions_rf = rf.transform(df_test)
cal_evaluate(predictions_rf)
```

评价指标为

精确率: 0.7373461012311902

召回率: 0.17760935826674357

F1分数: 0.2862643563699131

准确率: 0.8213229183978727

auc: 0.8284648716818422

总结如下:

1. 三个算法具有相似的准确度;
2. 随机森林的精确率最高、召回率最低、auc最高;
3. 召回率都很低, 说明样本中违约的人在模型里大多数未被预测准确。

## 遇到的困难

1. 报错 `py4j.protocol.Py4JError:`

`org.apache.spark.api.python.PythonUtils.isEncryptionEnabled does not exist in the JVM`

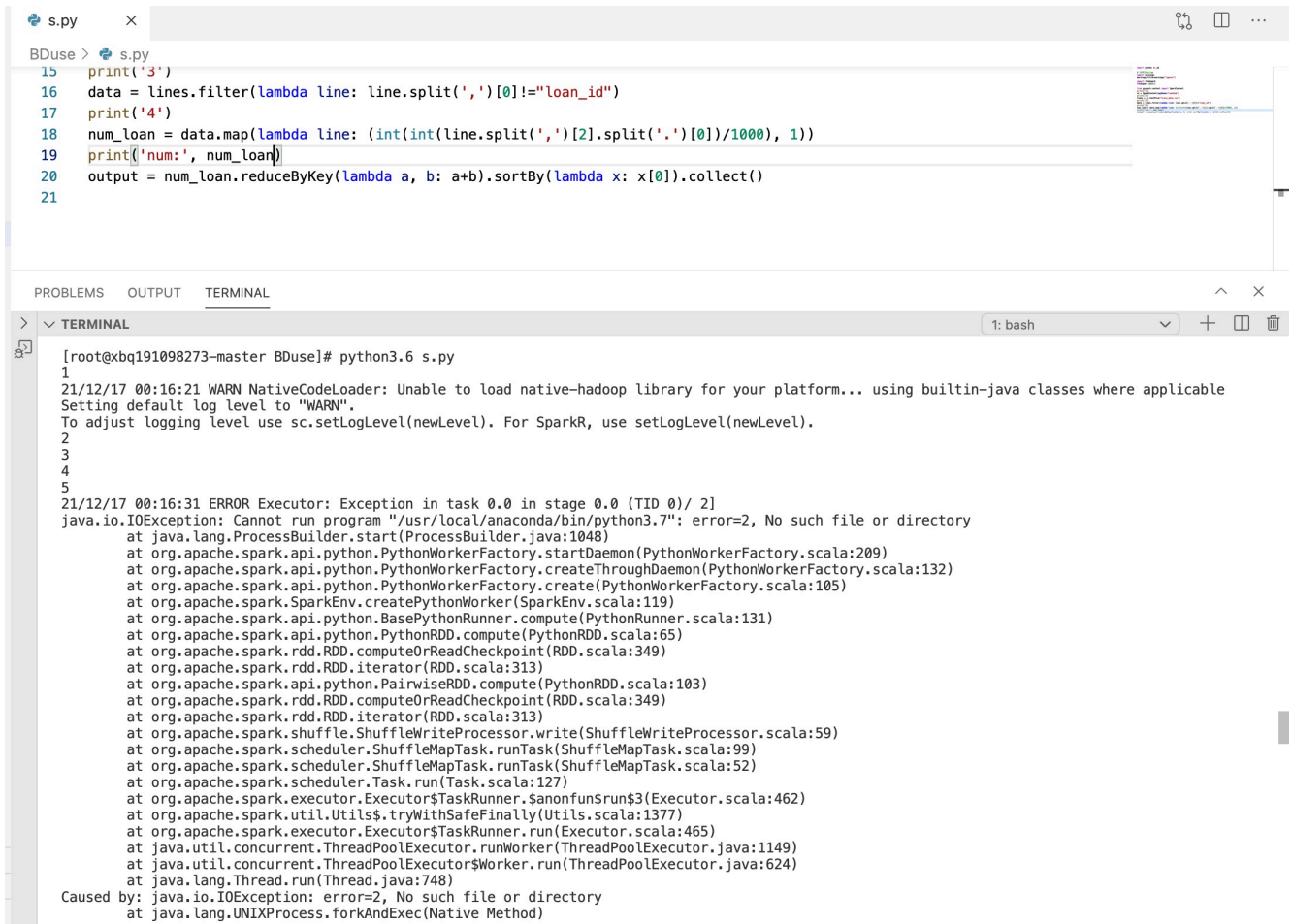
```
[root@xbq191098273-master BDuse]# python3.6 task3.py
21/12/17 02:13:43 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java cl
asses where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Traceback (most recent call last):
  File "task3.py", line 25, in <module>
    .appName("task3") \
  File "/usr/local/lib/python3.6/site-packages/pyspark/sql/session.py", line 228, in getOrCreate
    sc = SparkContext.getOrCreate(sparkConf)
  File "/usr/local/lib/python3.6/site-packages/pyspark/context.py", line 392, in getOrCreate
    SparkContext(conf=conf or SparkConf())
  File "/usr/local/lib/python3.6/site-packages/pyspark/context.py", line 147, in __init__
    conf, jsc, profiler_cls)
  File "/usr/local/lib/python3.6/site-packages/pyspark/context.py", line 226, in _do_init
    str(self._jvm.PythonUtils.getPythonAuthSocketTimeout(self._jsc))
  File "/usr/local/lib/python3.6/site-packages/py4j/java_gateway.py", line 1536, in __getattr__
    "{0}.{1} does not exist in the JVM".format(self._fqcn, name))
py4j.protocol.Py4JError: org.apache.spark.api.python.PythonUtils.getPythonAuthSocketTimeout does not exist in the JVM
```

解决方法:

在代码前加上即可。

```
1 import findspark
2 findspark.init()
```

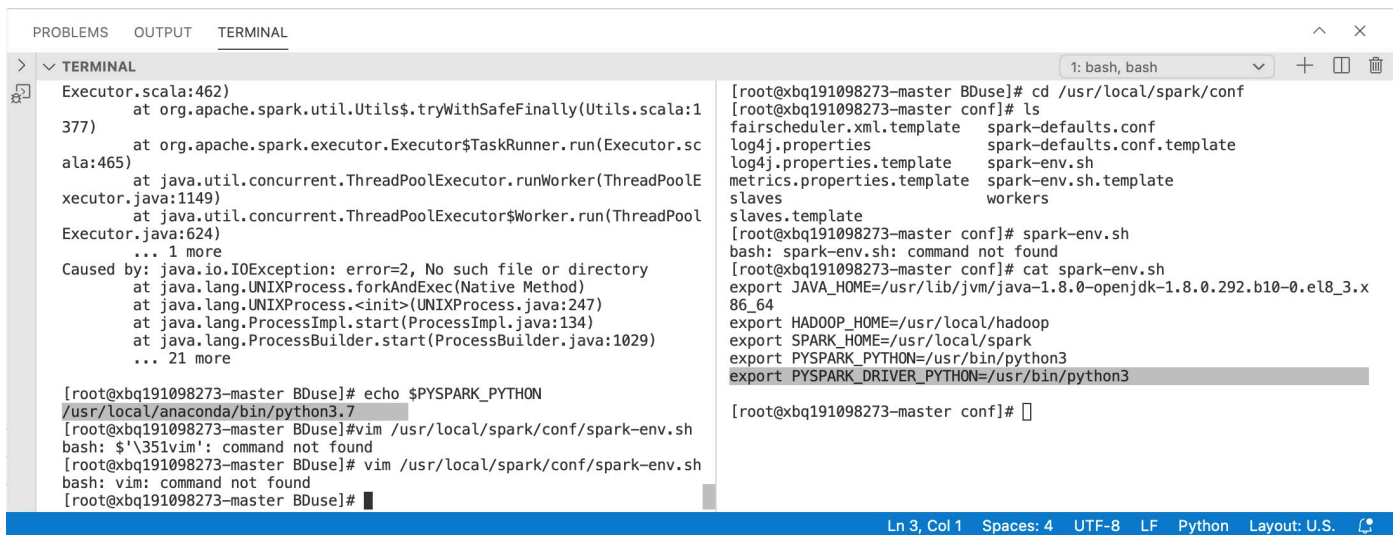
2. 在程序执行的时候，总是找别的python路径。其实本人并没有安装conda。



```
s.py
BDuse > s.py
15 print('3')
16 data = lines.filter(lambda line: line.split(',')[0]!="loan_id")
17 print('4')
18 num_loan = data.map(lambda line: (int(int(line.split(',')[2].split('.')[0])/1000), 1))
19 print('num:', num_loan)
20 output = num_loan.reduceByKey(lambda a, b: a+b).sortBy(lambda x: x[0]).collect()
21

PROBLEMS OUTPUT TERMINAL
> TERMINAL
1: bash
[root@xbq191098273-master BDuse]# python3.6 s.py
1
21/12/17 00:16:21 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
2
3
4
5
21/12/17 00:16:31 ERROR Executor: Exception in task 0.0 in stage 0.0 (TID 0) / 2]
java.io.IOException: Cannot run program "/usr/local/anaconda/bin/python3.7": error=2, No such file or directory
    at java.lang.ProcessBuilder.start(ProcessBuilder.java:1048)
    at org.apache.spark.api.python.PythonWorkerFactory.startDaemon(PythonWorkerFactory.scala:209)
    at org.apache.spark.api.python.PythonWorkerFactory.createThroughDaemon(PythonWorkerFactory.scala:132)
    at org.apache.spark.api.python.PythonWorkerFactory.create(PythonWorkerFactory.scala:105)
    at org.apache.spark.SparkEnv.createPythonWorker(SparkEnv.scala:119)
    at org.apache.spark.api.python.BasePythonRunner.compute(PythonRunner.scala:131)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:65)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:349)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:313)
    at org.apache.spark.api.python.PairwiseRDD.compute(PythonRDD.scala:103)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:349)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:313)
    at org.apache.spark.shuffle.ShuffleWriteProcessor.write(ShuffleWriteProcessor.scala:59)
    at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:99)
    at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:52)
    at org.apache.spark.scheduler.Task.run(Task.scala:127)
    at org.apache.spark.executor.Executor$TaskRunner.$anonfun$run$3(Executor.scala:462)
    at org.apache.spark.util.Utils$.tryWithSafeFinally(Utils.scala:1377)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:465)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
    at java.lang.Thread.run(Thread.java:748)
Caused by: java.io.IOException: error=2, No such file or directory
    at java.lang.UNIXProcess.forkAndExec(Native Method)
```

经检查发现 echo \$PYSPARK\_PYTHON 和 /usr/local/spark/conf/spark-env.sh 里的 PYSPARK\_PYTHON 路径不一样。



```
PROBLEMS OUTPUT TERMINAL
> TERMINAL
1: bash, bash

Executor.scala:462)
    at org.apache.spark.util.Utils$.tryWithSafeFinally(Utils.scala:1
377)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.sc
ala:465)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolE
xecutor.java:1149)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPool
Executor.java:624)
    ... 1 more
Caused by: java.io.IOException: error=2, No such file or directory
    at java.lang.UNIXProcess.forkAndExec(Native Method)
    at java.lang.UNIXProcess.<init>(UNIXProcess.java:247)
    at java.lang.ProcessImpl.start(ProcessImpl.java:134)
    at java.lang.ProcessBuilder.start(ProcessBuilder.java:1029)
    ... 21 more

[root@xbq191098273-master BDuse]# echo $PYSPARK_PYTHON
/usr/local/anaconda/bin/python3.7
[root@xbq191098273-master BDuse]# vim /usr/local/spark/conf/spark-env.sh
bash: $'\351vim': command not found
[root@xbq191098273-master BDuse]# vim /usr/local/spark/conf/spark-env.sh
bash: vim: command not found
[root@xbq191098273-master BDuse]#

[root@xbq191098273-master BDuse]# cd /usr/local/spark/conf
[root@xbq191098273-master conf]# ls
fairscheduler.xml.template  spark-defaults.conf.template
log4j.properties.template  spark-env.sh.template
metrics.properties.template  spark-env.sh.template
slaves                       workers
slaves.template
[root@xbq191098273-master conf]# spark-env.sh
bash: spark-env.sh: command not found
[root@xbq191098273-master conf]# cat spark-env.sh
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.292.b10-0.e18_3.x
86_64
export HADOOP_HOME=/usr/local/hadoop
export SPARK_HOME=/usr/local/spark
export PYSPARK_PYTHON=/usr/bin/python3
export PYSPARK_DRIVER_PYTHON=/usr/bin/python3

[root@xbq191098273-master conf]#
```

解决方案：

运行程序前，在命令行里输 export PYSPARK\_PYTHON=/usr/bin/python3.6。

3. 出现java、spark内部通讯的问题（可能描述不准确）。把bdkit关掉重开就不报错了，神奇！

## 实验感想

---

这次实验四让我对pyspark接口熟悉起来。在写代码的过程中，我发现很多操作和我之前熟悉的pandas、sklearn库很相似，颇有些融会贯通的意味。

总体来看，这门课程的四个实验带着我走进了处理大数据的世界，docker、hadoop、hbase这些高深的工具走到我身边，成为了解决问题的好帮手，痛苦与快乐并存。很感谢这门课带给我的奇妙体验。