

FBDP实验4

姓名：盛祺晨 学号：191220093

FBDP实验4

任务及解决方案

任务一

解决方案

Map

Reduce

最终结果

任务二

解决方案

最终结果

任务三

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

解决方案

最终结果

2.统计每个用户最终须缴纳的利息金额：

解决方案

最终结果

3.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。

解决方案

最终结果

任务四

解决方案&结果&思考

采样对抗类别不平衡

遇到的问题

任务及解决方案

任务一

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。 输出格式:

```
<工作领域> <记录数量>
```

解决方案

采用Hadoop Mapreduce框架，采用作业5的类似想法，为了方便，将train_data.csv中的industry列抽取出来，并作为train_industry.txt。其中格式为，每一个<工作领域>都独立为一行。

Map

因为处理过的数据很干净，于是每次设置value的时候只要将行数据（一定是一个industry数据）放入map中就可以。

```
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(Object key, Text value, Context context) throws IOException,
    InterruptedException {
        word.set(value.toString()); //每一行都是一个行业数据，故直接设置
        context.write(word,one);
    }
}
```

Reduce

```
public static class SortReducer extends Reducer<Text, IntWritable,Text,IntWritable> {
    //定义treeMap来保持统计结果,由于treeMap是按key升序排列的,这里要人为指定Comparator以实现倒排
    //这里先使用统计数为key, 被统计的单词为value
    private TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>(new
    Comparator<Integer>() {
        @Override
        public int compare(Integer x, Integer y) {
            return y.compareTo(x);
        }
    });
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
    IOException, InterruptedException {
        //reduce后的结果放入treeMap,而不是向context中记入结果
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        if (treeMap.containsKey(sum)) { //具有相同单词数的单词之间用逗号分隔
            String value = treeMap.get(sum) + "," + key.toString();
            treeMap.put(sum, value);
        } else {
            treeMap.put(sum, key.toString());
        }
    }
    protected void cleanup(Context context) throws IOException, InterruptedException {
        //将treeMap中的结果,按value-key顺序写入context中
        for (Integer key : treeMap.keySet()) {
            if (treeMap.get(key).toString().indexOf(",")!=-1) { // 说明有, 有同样个数的单词
                String[] splitstr=treeMap.get(key).toString().split(",");
                for (int i=0;i<splitstr.length;++i){
                    context.write(new Text(splitstr[i]), new IntWritable(key));
                }
            }
        }
    }
}
```

```

    }
    else{
        String s = treeMap.get(key);
        context.write(new Text(s),new IntWritable(key));
    }
}
}
}
}

```

对于Reduce来说，用TreeMap来让每次的计数结果保持顺序。在reduce主体部分，先计数后，放入treemap，最终在cleanup阶段，把treemap中排序好的结果写好。

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

任务二

编写 Spark 程序，统计网络信用贷产品记录数据中所有用户的贷款金额 total_loan 的分布情况。以 1000 元为区间进行输出。输出格式示例：

```
((2000,3000),1234)
```

解决方案

使用PySpark，用语句“pip3 install pyspark”或者“pip install pyspark”安装PySpark。

使用sc.parallelize(pandas_loan['total_loan'].tolist()).histogram(list(range(0, (int(pandas_loan.max() / 1000) + 1) * 1000, 1000)))将total_loan 的所有数据转化为list，并使用分箱操作，将1000作为间隔，分割，最后输出。

```
Appname = 'practice'
master = "local[4]"

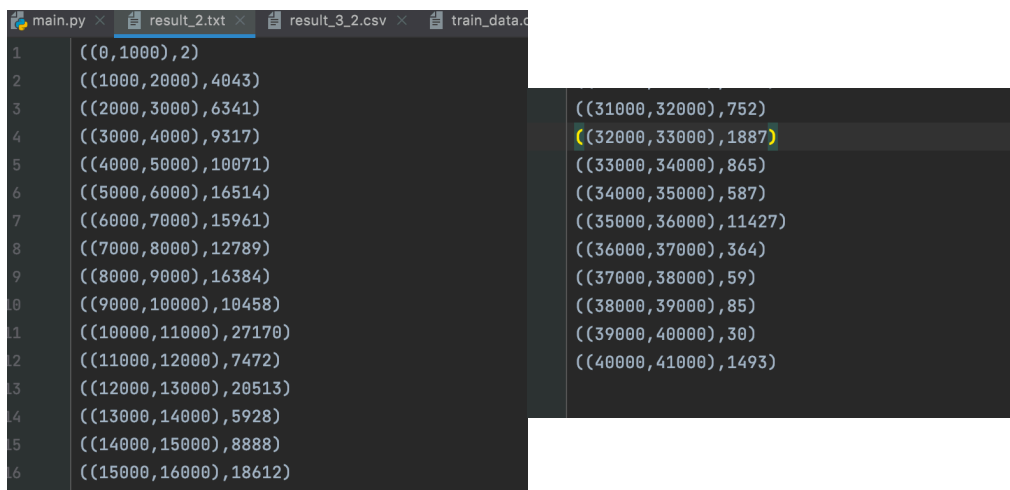
```

```

Config = SparkConf().setAppName(Appname).setMaster(master)
# 题目二, 分区间输出频率
spark = SparkSession.builder.config(conf=Config).getOrCreate() # spark实例化
sc = SparkContext.getOrCreate() # 创建会话
sc.setLogLevel("ERROR")
df_spark = spark.read.option('header', 'true').csv('train_data.csv', inferSchema=True)
df_loan = df_spark.select('total_loan')
pandas_loan = df_loan.toPandas()
# 方法 binning
rdd = sc.parallelize(pandas_loan['total_loan'].tolist()).histogram(
    list(range(0, (int(pandas_loan.max() / 1000) + 2) * 1000, 1000)))
result = []
for key in range(len(rdd[0]) - 1):
    result.append(f'({rdd[0][key]}, {rdd[0][key+1]}), {rdd[1][key]}')
np.savetxt("result_2.txt", result, fmt="%s", delimiter="\n")

```

最终结果



```

1 ((0,1000),2)
2 ((1000,2000),4043)
3 ((2000,3000),6341)
4 ((3000,4000),9317)
5 ((4000,5000),10071)
6 ((5000,6000),16514)
7 ((6000,7000),15961)
8 ((7000,8000),12789)
9 ((8000,9000),16384)
10 ((9000,10000),10458)
11 ((10000,11000),27170)
12 ((11000,12000),7472)
13 ((12000,13000),20513)
14 ((13000,14000),5928)
15 ((14000,15000),8888)
16 ((15000,16000),18612)
17 ((16000,17000),12000)
18 ((17000,18000),10000)
19 ((18000,19000),8000)
20 ((19000,20000),6000)
21 ((20000,21000),4000)
22 ((21000,22000),2000)
23 ((22000,23000),1000)
24 ((23000,24000),500)
25 ((24000,25000),250)
26 ((25000,26000),125)
27 ((26000,27000),62)
28 ((27000,28000),31)
29 ((28000,29000),15)
30 ((29000,30000),8)
31 ((30000,31000),4)
32 ((31000,32000),752)
33 ((32000,33000),1887)
34 ((33000,34000),865)
35 ((34000,35000),587)
36 ((35000,36000),11427)
37 ((36000,37000),364)
38 ((37000,38000),59)
39 ((38000,39000),85)
40 ((39000,40000),30)
41 ((40000,41000),1493)

```

任务三

基于 Hive 或者 Spark SQL 对网络信贷产品记录数据进行如下统计:

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

输出成 CSV 格式的文文件, 输出内容格式为:

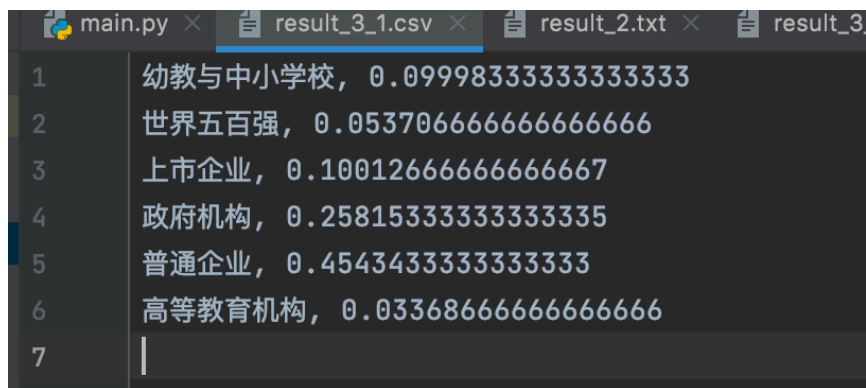
```
<公司类型>,<类型占比>
```

解决方案

```
# 题目三, Spark SQL 对网网络信用用贷产品记录数据进行行行如下统计
df_employer = df_spark.select('employer_type')
pandas_employer = df_employer.toPandas()
all_employer = list(set(pandas_employer['employer_type'].tolist()))
all_employer.sort()
# 方法: 使用 wordcount
rdd = sc.parallelize(pandas_employer['employer_type'].tolist())
result = rdd.map(lambda x:(x,1)).reduceByKey(lambda a, b: a + b)
result = result.collect()
res = [];sum = df_employer.count()
for i in range(len(result)):
    res.append(f"{result[i][0]}, {result[i][1]/sum}")
np.savetxt("result_3_1.csv", res, fmt="%s", delimiter="\n")
```

核心: 创建了rdd, 并用rdd.map(lambda x:(x,1)).reduceByKey(lambda a, b: a + b), 进行了mapreduce计算, 相当于wordcount, 计算出每个类别有几个后, 和总数相除, 最终得到结果。写入文件。

最终结果



1	幼教与中小学校, 0.09998333333333333
2	世界五百强, 0.05370666666666666
3	上市企业, 0.10012666666666667
4	政府机构, 0.25815333333333335
5	普通企业, 0.45434333333333333
6	高等教育机构, 0.03368666666666666
7	

2.统计每个用户最终须缴纳的利息金额:

$$total_money = year_of_loan \times monthly_payment \times 12 - total_loan$$

输出成 CSV 格式的文文件, 输出内容格式为:

```
<user_id>,<total_money>
```

解决方案

```

# 选取所需要的数据列
df_interest =
df_spark.select(['user_id','year_of_loan','monthly_payment','total_loan'])
# 根据需求计算数据
df_interest = df_interest.withColumn("total_money",
    df_interest['year_of_loan'] * 12 * df_interest['monthly_payment']-
df_interest['total_loan'])
# 存储
pandas_interest = df_interest.select(['user_id', 'total_money']).toPandas()
pandas_interest.to_csv('result_3_2.csv',index=0)

```

按照公式计算即可，其中涉及到的基本操作是.select取列，以及.withColumn 创建新列

最终结果

main.py × result_2.txt × result_3_2.csv × train_data.csv × result_3_3.csv ×	
The file size (7 MB) exceeds the configured limit (2.56 MB). Code insight features are not av	
1	user_id,total_money
2	0,3846.0000000000002
3	1,1840.6000000000004
4	2,10465.599999999999
5	3,1758.5200000000004
6	4,1056.8800000000001
7	5,7234.639999999999
8	6,757.9200000000001
9	7,4186.959999999999
10	8,2030.7600000000002
11	9,378.72000000000025
12	10,4066.7599999999984
13	11,1873.5599999999995
14	12,5692.279999999999
15	13,1258.6800000000003
16	14,6833.5999999999985
17	15,9248.2
18	16,6197.1199999999995
19	17,1312.4399999999996
20	18,5125.2000000000001
21	19,1215.8400000000001
22	20,1394.9200000000002
23	21,5771.4000000000015
24	22,3202.4799999999996

3.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。

输出成 CSV 格式的文件，输出内容格式为：

```
<user_id>,<censor_status>,<work_year>
```

解决方案

```
df_wk = df_spark.select(['user_id', 'censor_status', 'work_year'])
# 设定过滤条件
df_wk_f = df_wk.filter(((df_wk['work_year'] > '5 years') | (df_wk['work_year'] == '10+ years'))
& ~(df_wk['work_year'] == '< 1 year'))
# 存储
df_wk_f.toPandas().to_csv('result_3_3.csv',index=0)
```

使用了.filter 进行过滤，其中要注意是根据字符串排序，在选取 > '5 years'条件后，还需要补上备误删的 == '10+ years' 条件，和删掉误增的 == '< 1 year' 条件。

最终结果

我们没有计入5年工作年期的人

```
>>> set(df['work_year'])
{nan, '9 years', '6 years', '10+ years', '5 years', '3 years', '7 years', '< 1 year', '2 years', '4 years', '8 years', '1 year'}
>>> set(t['work_year'])
{'9 years', '6 years', '10+ years', '7 years', '8 years'}
```

发现work_year超过5年的人均被过滤掉。最终文件结果如下，发现成功。

```
main.py × result_2.txt × result_3_2.csv × train_data.csv × result_3_3.csv ×
The file size (2.7 MB) exceeds the configured limit (2.56 MB). Code insight features are not available.
1 user_id,censor_status,work_year
2 1,2,10+ years
3 2,1,10+ years
4 5,2,10+ years
5 6,0,8 years
6 7,2,10+ years
7 9,0,10+ years
8 10,2,10+ years
9 15,1,7 years
10 16,2,10+ years
11 17,0,10+ years
12 18,1,10+ years
13 20,1,7 years
14 21,2,10+ years
15 25,2,10+ years
16 26,0,10+ years
```

任务四

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。

```
计算准确率 acc = 0.8401464545153563
计算召回率 recall = 0.08069264470664596
模型特征重要性:(37,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
模型特征数:37

Process finished with exit code 0
```

解决方案&结果&思考

主要使用了随机森林方法，参考: https://blog.csdn.net/weixin_43790705/article/details/108653416

首先需要将类别变量转化为数值变量。并且对于缺失值，我们采用跳过，由于统计过缺失值不多，认为这样的跳过是没有问题的。


```

from pyspark.ml.feature import StringIndexer
indexer = StringIndexer(inputCols=['class', 'sub_class', 'work_type',
'employer_type', 'industry', 'work_year'],
                        outputCols=['class_index', 'sub_class_index',
'work_type_index', 'employer_type_index',
'industry_index', 'work_year_index'],
                        handleInvalid='skip')
df_r = indexer.fit(df_spark).transform(df_spark)

```

注意转化后的df_r变量将转化后的 **_index 属性贴到df_spark的列的后面。

```

from pyspark.ml.feature import VectorAssembler
feature_to_trans =
['total_loan', 'year_of_loan', 'interest', 'monthly_payment', 'class_index', 'sub_class_index',
'work_type_index', 'employer_type_index', 'industry_index', 'work_year_index', 'house_exists',
'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']
feature = VectorAssembler(inputCols=feature_to_trans, outputCol='Independent Features',
handleInvalid='skip')
df_feature = feature.transform(df_r)
final = df_feature.select(['Independent Features', 'is_default'])
train, test = final.randomSplit([0.8, 0.2])

```

然后除了ID和时间这种标签除外所有标签放入feature_to_trans，用VectorAssembler转化为向量后，和标签装好，并按照0.8:0.2切分测试训练集，形成我们即将喂入机器学习模型的训练测试数据。下图为final变量（训练测试集的样子）的展示。

```
>>> final.show()
+-----+-----+
|Independent Features|is_default|
+-----+-----+
|(37,[0,1,2,3,4,5,...]|0|
|[10700.0,3.0,10.1...]|0|
|[8000.0,3.0,8.24,...]|0|
|[28000.0,3.0,15.5...]|1|
|[6000.0,3.0,7.89,...]|0|
|(37,[0,1,2,3,4,6,...]|0|
|(37,[0,1,2,3,4,5,...]|0|
|[21850.0,3.0,11.4...]|0|
|[10500.0,3.0,10.9...]|0|
|[22000.0,3.0,15.6...]|0|
|[9600.0,3.0,8.18,...]|1|
|(37,[0,1,2,3,4,5,...]|1|
|[28675.0,3.0,13.1...]|1|
|[5925.0,3.0,13.49...]|0|
|[12000.0,5.0,14.9...]|0|
|[9600.0,3.0,7.91,...]|1|
|[8000.0,3.0,10.75...]|0|
|(37,[0,1,2,3,4,5,...]|1|
|(37,[0,1,2,3,4,5,...]|1|
|(37,[0,1,2,3,4,5,...]|0|
+-----+-----+
only showing top 20 rows
```

随后，我们使用随机森林模型计算。并用常用指标来评价预测结果。

随机森林模型参数：maxBins=100（每层最多100个属性来区分），maxDepth=10（每个决策树最多10层），numTrees=20(20个决策树)

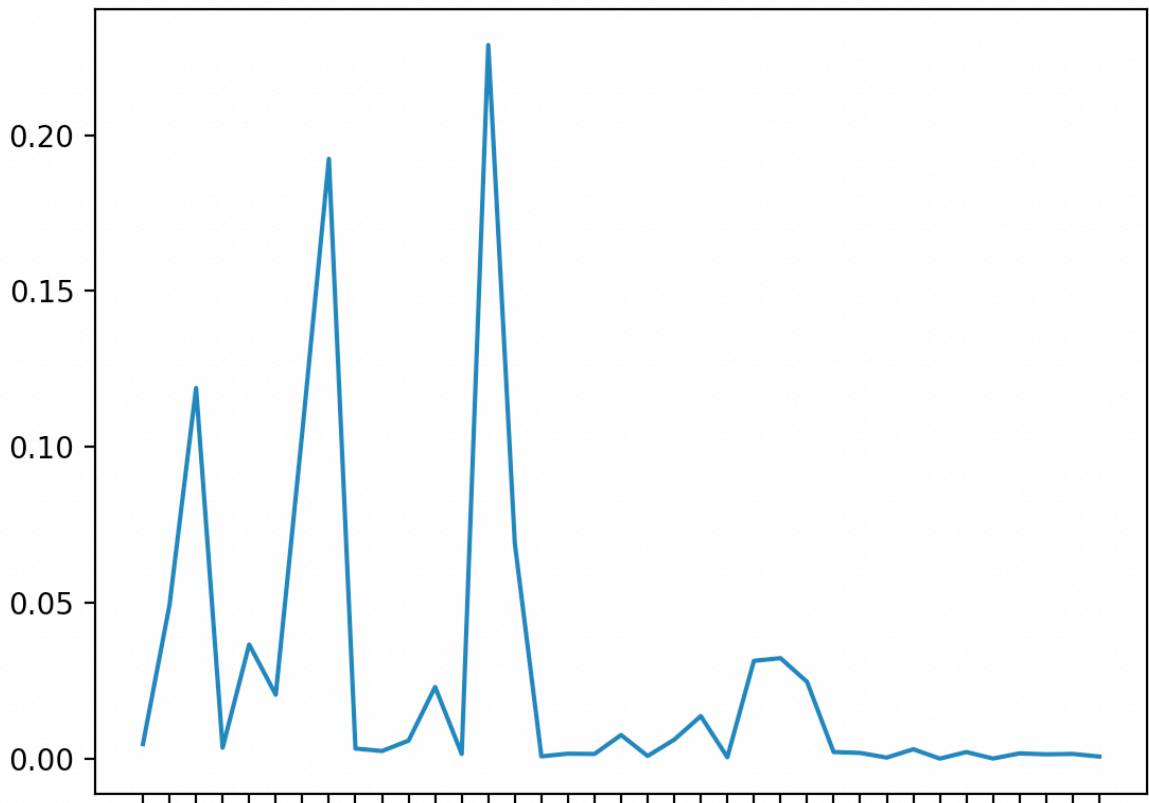
结果：

```
精确率：0.6840439003799071
召回率：0.3216234990572591
准确率：0.8379203703343512
F1分数：0.437529530880864
auc分数：0.8566435485661126
```

同样地，我们测试了线性SVM模型，测试结果: 准确率 acc = 0.8041805040046884 召回率 recall = 0.0

recall非常小，acc也不如RF可观，我们发现由于类别不平衡，变量维度过多，数据量过大，线性SVM器没有很好的表现，并且由于模型自身比较鲁棒，并行计算并不能大幅提高他的计算速度，于是我们在后面主要采取RF（随机森林）模型。

随机森林有一个featureImportances函数可以评估各属性重要性，于是我们绘出每个属性重要性的图，并将这些属性抽取出来，重新跑RF，看能否提高性能，把贡献小的给去掉，另外我们将这些属性抽取后重新跑了一遍SVM，发现准确率并没有改进，于是并不展示。



图：RF给出的全属性重要性图

发现重要属性为'interest', 'work_type_index', 'employer_type_index', 'marriage','offsprings'，抽取重要属性后的性能表现

```
精确率： 0.6124486200822079
召回率： 0.3777958887983338
准确率： 0.8309486144931658
F1分数： 0.46732007840940915
auc分数： 0.8324214479742911
模型特征重要性：(5,[0,1,2,3,4],
[0.24179184903803783,0.14719305156070156,0.24422681307551525,0.36003073223512916,0.0067
575540906161195])
模型特征数:5
```

发现性能只有略微下降，不过大幅降低了模型复杂度。接下来，我们将对数据的类别不平衡进行处理，看看对RF性能是否有改进。

采样对抗类别不平衡

SMOTE过采样

```
精确率: 0.5331780055917987
召回率: 0.5682924406476607
准确率: 0.8188244407862884
F1分数: 0.5501755060826081
auc分数: 0.8435877224118804
```

随机欠采样

```
精确率: 0.7590768364762173
召回率: 0.8117788702718972
准确率: 0.7808894586332709
F1分数: 0.7845437796955298
auc分数: 0.8614903296243576
```

可以看到，无论过采样还是欠采样，采样后都对模型的召回率、F1有提高，对精确率来说，随机欠采样效果更佳，auc小幅提高。对于准确率来说，两种方法的准确率都要比原方法差点。

另外，我们对SVM同样进行了采样后的数据测试，发现提升并没有这么明显，SMOTE采样的正确率比原方法高了1%，而随机欠采样的正确率则低了5%

遇到的问题

1.binning出现问题

```
# 方法1 binning (错误)
rdd = sc.parallelize(pandas_employer['employer_type'].tolist()).histogram(
    all_employer)
result = []
sum = df_employer.count()
for key in range(len(rdd[0]) - 1):
    result.append(f'{rdd[0][key]}, {rdd[1][key]/sum}')
```

发现3.1中，如果用了第二题一样的方法分箱，先将string排序，那么实际产生的箱子是['世界五百强','上市企业')这类的箱子，那么最后一个箱子['普通企业','高等教育机构]，会导致高等教育机构的人数混入普通企业中。

2.在SMOTE采样后，测试性能时候，我将test测试集设定为smote采样后的20%数据，测试结果非常喜人，准确率达到88%，但是SMOTE采样本身就是插值，插值过的数据和原数据不一样会引入噪声，于是应该使用原来的数据来测试，得到了之前写的较低正确率。

```
精确率: 0.8814813018382888
召回率: 0.8786259911042351
准确率: 0.8802528371665801
F1分数: 0.8800513304762597
auc分数: 0.9507530028905377
```