金融大数据-实验4

施宇 191250119

任务一

将整个任务分为两个部分统计特征`industry`每个取值的数量，并按照数量进行排序。

统计数量

与WordCount任务类似，在Map阶段，对于读入的每一行数据，先按照逗号分开，得到每个单独的数据，然后取出industry的取值，这里有一点需要注意，Map第一次读入的数据为列名，需要去除。Map的输出为`(industry特征的值， 1)`。

```
public static class ICMap extends Mapper<LongWritable, Text, Text, LongWritable>
{
    private boolean first = true;
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
Text, LongWritable>.Context context){
        if(first){
            first = false;
            return;
        }
        String train_data = value.toString();
        String[] features = train_data.split(",");
        String industry = features[10];
        //            System.out.println(industry);
        try {
            context.write(new Text(industry), new LongWritable(1));
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

在Reduce阶段，合并每个industry取值的计数，输出最终的计数结果：

```
public static class ICReducer
    extends Reducer<Text, LongWritable, Text, LongWritable> {

    @Override
    protected void reduce(Text key, Iterable<LongWritable> values,
                          Context context
                         ){
        long count = 0;
        for(LongWritable value: values){
            count += value.get();
        }
        try {
            context.write(key, new LongWritable(count));
        }catch (Exception e){
            e.printStackTrace();
```

```
        }
    }
}
```

计数结果：

```
交通运输、仓储和邮政业    15028
住宿和餐饮业 26954
信息传输、软件和信息技术服务业    24078
公共服务、社会组织    30262
农、林、牧、渔业 14758
制造业    8864
国际组织 9118
建筑业    20788
房地产业 17990
批发和零售业 8892
文化和体育业 24211
电力、热力生产供应业 36048
采矿业    14793
金融业    48215

```

排序

由于是按照值进行排序的，所以通过构造一个新的数据类，将industry和值都作为该类的变量，该类之间进行排序比较的时候，先按照值的大小比较，再按照industry的大小比较。

在Map中，读入上一步计数的结果后，输出`((industry, count), Null)`：

```
public static class ISMapper extends Mapper<LongWritable, Text ,
TextIntWritable, NullWritable>{

    @Override
    public void map(LongWritable key, Text value, Context context){
        try{
            TextIntWritable k = new TextIntWritable();
            String []string = value.toString().split("\t");
            k.set(new Text(string[0]), new
IntWritable(Integer.valueOf(string[1])));
            context.write(k,  NullWritable.get());
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

在Reduce中，由于读入的键值对已经是有序的，所以直接输出即可：

```
public static class ISReducer extends Reducer<TextIntWritable, NullWritable,
TextIntWritable, NullWritable>{
    public void reduce(TextIntWritable key, Iterable<NullWritable> value,
Context context) throws IOException, InterruptedException{
        for(NullWritable v : value)
            context.write(key, v);
    }
}
```

排序结果：

```
金融业 48215
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
```

通过Spark复现结果

先读入数据：

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as f

session = SparkSession.builder.master("local[*]").appName("test").getOrCreate()
df = session.read.csv("train_data.csv", encoding="utf-8", header=True)
```

选出industry列后按照取值分组计数，然后按照计数结果降序排列：

```
df.select("industry").groupBy("industry").count().orderBy(f.desc("count")).show(
)
```

结果：

In [4]:
```
df.select("industry").groupBy("industry").count().orderBy(f.desc("count")).show()
```

```
+------------------------------+-----+
|                      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|
+------------------------------+-----+
```

任务二

使用Spark的Python API，PySpark完成后续的spark程序的编写。

由于需要按区间对total_loan进行计数，所以新建一列`total_loan_class`，该列的值为该行`total_loan`的值所在的区间。再新建一列total_loan_sort，为区间的左端，用于后续的排序。生成该列的函数如下，将该函数注册为udf函数：

```
def total_loan_classify(str_value):
    value = float(str_value)
    c = int(value // 1000)
    res = "(" + str(c*1000) + "," + str((c+1)*1000) + ")"
    return res
def total_loan_sort(str_value):
    value = int(str_value.split(",")[0][1:])
    return value
udf_total_loan_classify = f.udf(total_loan_classify, StringType())
udf_total_loan_sort = f.udf(total_loan_sort, StringType())
```

选出total_loan和total_loan_class列后，按照total_loan_class列分组计数，然后按照total_loan_sort进行升序排列：

```
df = df.withColumn("total_loan_class", udf_total_loan_classify("total_loan"))
res = df.select(["total_loan",
"total_loan_class"]).groupBy("total_loan_class").count()
res = res.withColumn("total_loan_sort", udf_total_loan_sort("total_loan_class"))
res = res.withColumn("total_loan_sort",
res["total_loan_sort"].cast(IntegerType()))
res = res.orderBy("total_loan_sort")
res = res.select(["total_loan_class", "count"])
```

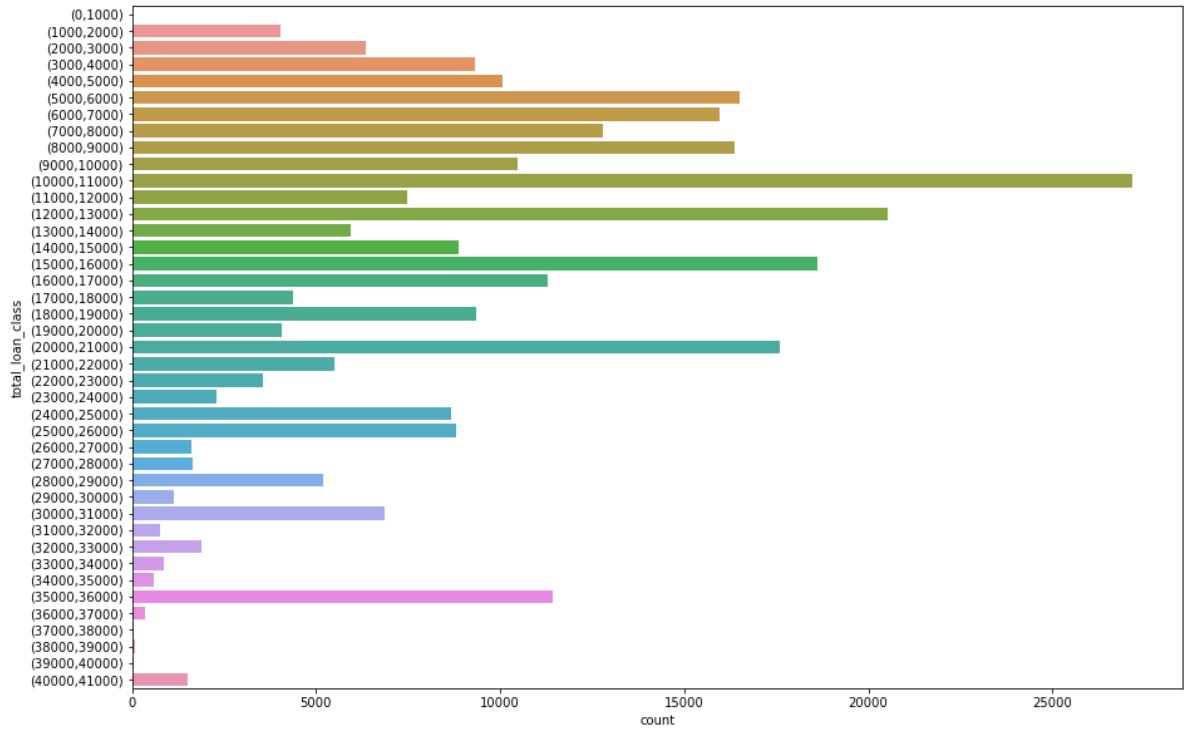
结果如下：

```
res.show(50)
```

```
+----------------+-----+
|total_loan_class|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|
|    (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|
+----------------+-----+
```

根据统计结果绘制频度分布直方图：

```
respd = res.toPandas()
plt.figure(figsize=(15, 10))
sns.barplot(respd['count'], respd['total_loan_class'], orient='h')
```



可以看出贷款金额分布大致呈左偏分布。

任务三

第一问

先计算数据总数：

```
In [8]: total_employer_type = df.select("employer_type").count()
        total_employer_type

Out[8]: 300000
```

所以每个数据的占比为1/300000，将该值存储为列`employer_type_percent`：

```
df = df.withColumn("employer_type_percent", f.lit(1/total_employer_type))
```

从数据中选出employer_type和employer_type_percent，然后按照employer_type分组对employer_type_percent求和，再按照求和的大小正序排列并保留4位小数，导出成csv格式：

```
res = df.select(["employer_type",
"employer_type_percent"]).groupBy("employer_type").agg(f.sum("employer_type_perc
ent")).orderBy(f.sum("employer_type_percent"))
res =
res.withColumnRenamed('sum(employer_type_percent)','employer_type_percent')
res = res.select(['employer_type', f.bround("employer_type_percent",
scale=4).alias('employer_type_percent')])
res.repartition(1).write.csv("./work3-1", encoding="gbk", header=True)
```

结果：

employer_type	employer_type_percent
高等教育机构	0.0337
世界五百强	0.0537
幼教与中小学校	0.1
上市企业	0.1001
政府机构	0.2582
普通企业	0.4543

第二问

由于数据读入的时候，所有的数据都默认设置为了String类型，因此先将需要参与计算的三个变量`year_of_loan`、`monthly_payment`、`total_loan`转为合适的类型：

```
df = df.withColumn("year_of_loan", df["year_of_loan"].cast(IntegerType()))
df = df.withColumn("monthly_payment", df["monthly_payment"].cast(FloatType()))
df = df.withColumn("total_loan", df["total_loan"].cast(FloatType()))
```

构建新列`total_money`，按照题目提供的公式计算，然后导出csv格式：

```
df = df.withColumn("total_money", df["year_of_loan"]*df["monthly_payment"]*12-
df["total_loan"])
res = df.select(["user_id", "total_money"])
res.repartition(1).write.csv("./work3-2", encoding="gbk", header=True)
```

结果（部分）：

user_id	total_money
0	3846
1	1840.6006
2	10465.602
3	1758.5195
4	1056.8799
5	7234.6406
6	757.9204
7	4186.959
8	2030.7598
9	378.71973
10	4066.7598
11	1873.5605
12	5692.2793

第三问

由于work_year特征的类别为String，且总体可分为三种类型：< 1 year，x years，10+ years，为了筛选出所有大于5 years的值，需要分情况进行处理。新增列work_year_num，对于work_year取值为：Null，设置其值为-1；< 1 year，设置其值为0；x years，设置其值为x；10+ years，设置其值为11。再筛选出所有work_year_num大于5的行即可：

```
def work_year_process(str_value):
    if str_value is None:
        return -1
    elif '10+' in str_value:
        return 11
    elif "<" in str_value:
        return 0
```

```
    else:
        str_num = str_value.split(' ')[0]
        return int(str_num)
udf_work_year_process = f.udf(work_year_process, StringType())
df = df.withColumn("work_year_num", udf_work_year_process('work_year'))
res = df.select(['user_id', 'censor_status', 'work_year',
'work_year_num']).filter(df['work_year_num']>5)
res = res.select(['user_id', 'censor_status', 'work_year'])
res.repartition(1).write.csv("./work3-3", encoding="gbk", header=True)
```

结果（部分）：

user_id	censor_sta	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

任务四

先根据数据中每个特征的含义和内容，确定其合理的数据类型，然后按照该类型读取数据：

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as f
from pyspark.ml.feature import VectorAssembler
import pyspark.sql.types as typ
import pandas as pd
import  pyspark.ml.feature as ft
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer
from pyspark.ml.classification import RandomForestClassifier
import matplotlib.pyplot as plt
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

session = SparkSession.builder.master("local[*]").appName("test").getOrCreate()
labels = [('loan_id', typ.IntegerType()),
          ('user_id', typ.IntegerType()),
          ('total_loan', typ.DoubleType()),
          ('year_of_loan', typ.IntegerType()),
          ('interest', typ.DoubleType()),
```

```
            ('monthly_payment', typ.DoubleType()),
            ('class', typ.StringType()),
            ('sub_class', typ.StringType()),
            ('work_type', typ.StringType()),
            ('employment_type', typ.StringType()),
            ('industry', typ.StringType()),
            ('work_year', typ.StringType()),
            ('house_exist', typ.IntegerType()),
            ('house_loan_status', typ.IntegerType()),
            ('censor_status', typ.IntegerType()),
            ('marriage', typ.IntegerType()),
            ('offsprings', typ.IntegerType()),
            ('issue_date', typ.StringType()),
            ('use', typ.IntegerType()),
            ('post_code', typ.DoubleType()),
            ('region', typ.IntegerType()),
            ('debt_loan_ratio', typ.DoubleType()),
            ('del_in_18month', typ.DoubleType()),
            ('scoring_low', typ.DoubleType()),
            ('scoring_high', typ.DoubleType()),
            ('pub_dero_bankrup', typ.DoubleType()),
            ('early_return', typ.IntegerType()),
            ('early_return_amount', typ.IntegerType()),
            ('early_return_amount_3mon', typ.DoubleType()),
            ('recircle_b', typ.DoubleType()),
            ('recircle_u', typ.DoubleType()),
            ('initial_list_status', typ.IntegerType()),
            ('earlies_credit_mon', typ.StringType()),
            ('title', typ.DoubleType()),
            ('policy_code', typ.DoubleType()),
            ('f0', typ.DoubleType()),
            ('f1', typ.DoubleType()),
            ('f2', typ.DoubleType()),
            ('f3', typ.DoubleType()),
            ('f4', typ.DoubleType()),
            ('f5', typ.DoubleType()),
            ('is_default', typ.IntegerType()),
            ]
schema = typ.StructType([
    typ.StructField(e[0], e[1], True) for e in labels
])
df = session.read.csv("train_data.csv", encoding="utf-8", header=True,
schema=schema)
```

填充缺失值，对于数值变量，缺失值填充为-1，对于字符串变量，缺失值填充为'-1'：

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

对所有String类型的特征进行编码，转为数值类型特征：

```
strings = ['class', 'sub_class', 'work_type', 'employment_type', 'industry',
'work_year', 'issue_date', 'earlies_credit_mon']
indexes = [StringIndexer(inputCol =s, outputCol=s+"_ind") for s in strings]
pipeline = Pipeline(stages=indexes)
tmodel = pipeline.fit(df)
dfres = tmodel.transform(df)
dfres = dfres.drop(*strings)
```

考虑到日期特征取值较多，进行类型编码后类型数量过多，因此对两个日期特征进行分桶，设置桶的数量为20：

```
discretizers = [QuantileDiscretizer(numBuckets=20, inputCol=s, outputCol=s+"b")
for s in ["issue_date_ind", "earlies_credit_mon_ind"]]
pipeline = Pipeline(stages=discretizers)
bmodel = pipeline.fit(dfres)
dfres = bmodel.transform(dfres)
dfres = dfres.drop("issue_date_ind", "earlies_credit_mon_ind")
```

```
dfres.select("issue_date_ind", "earlies_credit_mon_ind", "issue_date_indb", "earlies_credit_mon_indb").show(10)
```

```
+--------------+---------------------+--------------+----------------------+
|issue_date_ind|earlies_credit_mon_ind|issue_date_indb|earlies_credit_mon_indb|
+--------------+---------------------+--------------+----------------------+
|          13.0|                335.0|           7.0|                  18.0|
|          98.0|                248.0|          19.0|                  17.0|
|          18.0|                 90.0|           8.0|                   9.0|
|          52.0|                 38.0|          16.0|                   4.0|
|          45.0|                 36.0|          15.0|                   4.0|
|          18.0|                 44.0|           8.0|                   5.0|
|           2.0|                  5.0|           1.0|                   0.0|
|          19.0|                 59.0|           9.0|                   6.0|
|          69.0|                 45.0|          18.0|                   5.0|
|          44.0|                189.0|          15.0|                  15.0|
+--------------+---------------------+--------------+----------------------+
only showing top 10 rows
```

进行特征集成，将所有特征合并到一个数组feature中：

```
df_assembler = VectorAssembler(inputCols=data.columns, outputCol="features")
data = df_assembler.transform(dfres)
```

集成结果：

```
data.select(['features','is_default']).show(3)
```

```
+--------------------+----------+
|            features|is_default|
+--------------------+----------+
|[119262.0,0.0,120...|         1|
|[369815.0,1.0,800...|         0|
|[787833.0,2.0,200...|         0|
+--------------------+----------+
only showing top 3 rows
```

按照8：2的比例划分训练集和测试集：

```
train_data, test_data = data.randomSplit([0.8,0.2])
```

Logistic回归

使用Logistic回归进行分类：

```
logistic = cl.LogisticRegression(maxIter=10, regParam=0.01,
labelCol='is_default').fit(train_data)
# 预测
predictions = logistic.transform(test_data)
```

计算预测准确性：

```
auc = BinaryClassificationEvaluator(labelCol='is_default').evaluate(predictions)
print('BinaryClassificationEvaluator 准确性： {0:.0%}'.format(auc))
```

```
auc = BinaryClassificationEvaluator(labelCol='is_default').evaluate(predictions)
print('BinaryClassificationEvaluator 准确性： {0:.0%}'.format(auc))
```

```
BinaryClassificationEvaluator 准确性： 79%
```

随机森林

使用随机森林模型进行分类：

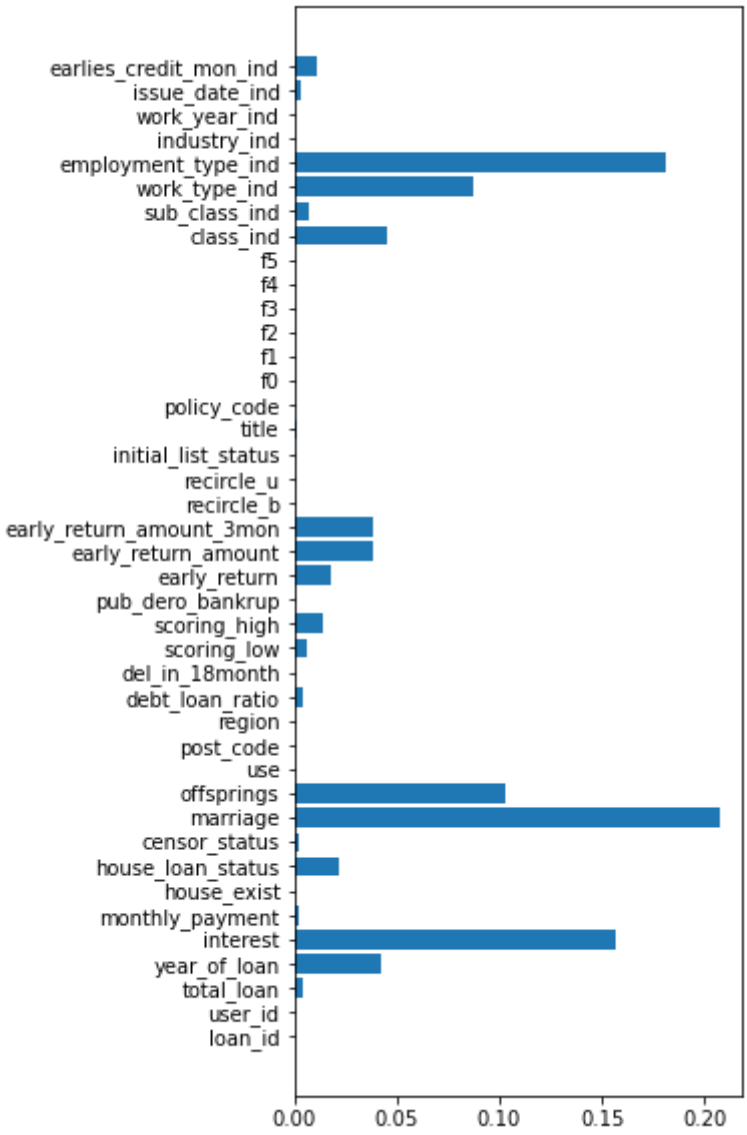
```
cla = RandomForestClassifier(labelCol='is_default', maxDepth=7, maxBins=700,
numTrees=30).fit(train_data)
```

训练完成后，对测试集中的数据进行预测：

```
predictions = cla.transform(test_data)
```

绘制特征重要性柱状图：

```
fig = plt.figure(figsize=(4,10))
plt.barh(feas, cla.featureImportances)
```



可以发现，employment_type_ind、work_type_ind、offsprings、marriage和interest的重要性较高。

查看测试集的预测情况：

```
predictions.select(['probability','is_default','prediction']).show(10,False)
```

```
+----------------------------------------+----------+----------+
|probability                             |is_default|prediction|
+----------------------------------------+----------+----------+
|[0.8254504460579561,0.1745495539420439] |0         |0.0       |
|[0.8918362346769779,0.1081637653230221] |0         |0.0       |
|[0.7710806562194464,0.22891934378055362]|0         |0.0       |
|[0.819250046766342,0.1807499532336579]  |0         |0.0       |
|[0.8024661647271126,0.1975338352728874] |0         |0.0       |
|[0.4141285200987836,0.5858714799012164] |0         |1.0       |
|[0.9190357149089542,0.08096428509104586]|0         |0.0       |
|[0.8623894050549803,0.13761059494501976]|1         |0.0       |
|[0.5209574912763438,0.4790425087236561] |0         |0.0       |
|[0.9043538663055933,0.09564613369440665]|0         |0.0       |
+----------------------------------------+----------+----------+
only showing top 10 rows
```

可以看到大部分预测是准确的，有小部分的预测是错误的。计算模型在测试集上的准确率：

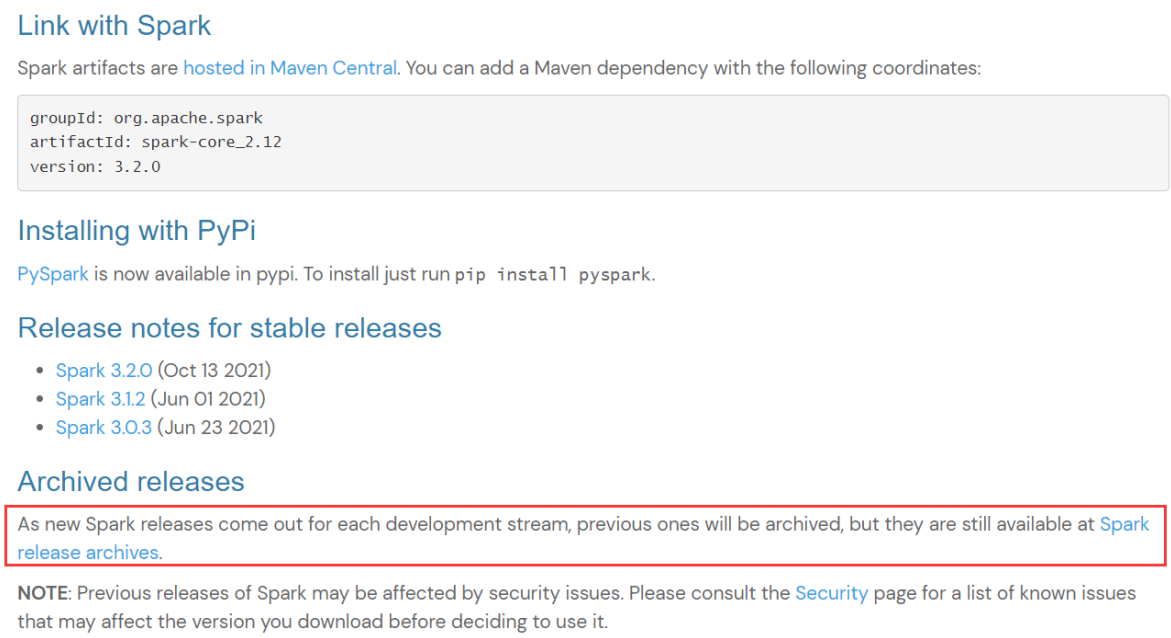
```
auc = BinaryClassificationEvaluator(labelCol='is_default').evaluate(predictions)
print('BinaryClassificationEvaluator 准确性:  {0:.0%}'.format(auc))
```

```
BinaryClassificationEvaluator 准确性:  85%
```

准确率为85%，较Logistic模型79%的准确率有一定程度的提高。

附录-安装Spark(Python版本)

到官网下载Spark，这里选择2.4.5版本，需要去历史release里下载：



Link with Spark

Spark artifacts are hosted in Maven Central. You can add a Maven dependency with the following coordinates:

```
groupId: org.apache.spark
artifactId: spark-core_2.12
version: 3.2.0
```

Installing with PyPi

PySpark is now available in pypi. To install just run `pip install pyspark`.

Release notes for stable releases

- Spark 3.2.0 (Oct 13 2021)
- Spark 3.1.2 (Jun 01 2021)
- Spark 3.0.3 (Jun 23 2021)

Archived releases

As new Spark releases come out for each development stream, previous ones will be archived, but they are still available at Spark release archives.

NOTE: Previous releases of Spark may be affected by security issues. Please consult the Security page for a list of known issues that may affect the version you download before deciding to use it.

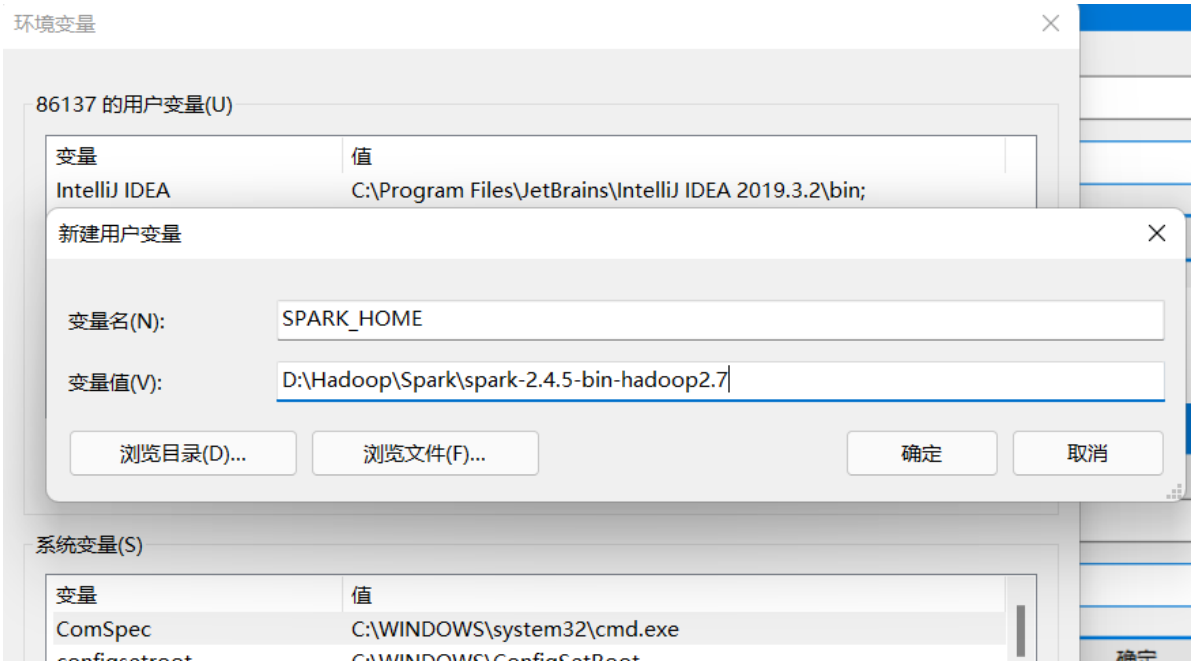
由于目前已经安装的Hadoop版本为2.9.1，所以这里选择对应Hadoop版本为2.7的spark：

Index of /dist/spark/spark-2.4.5

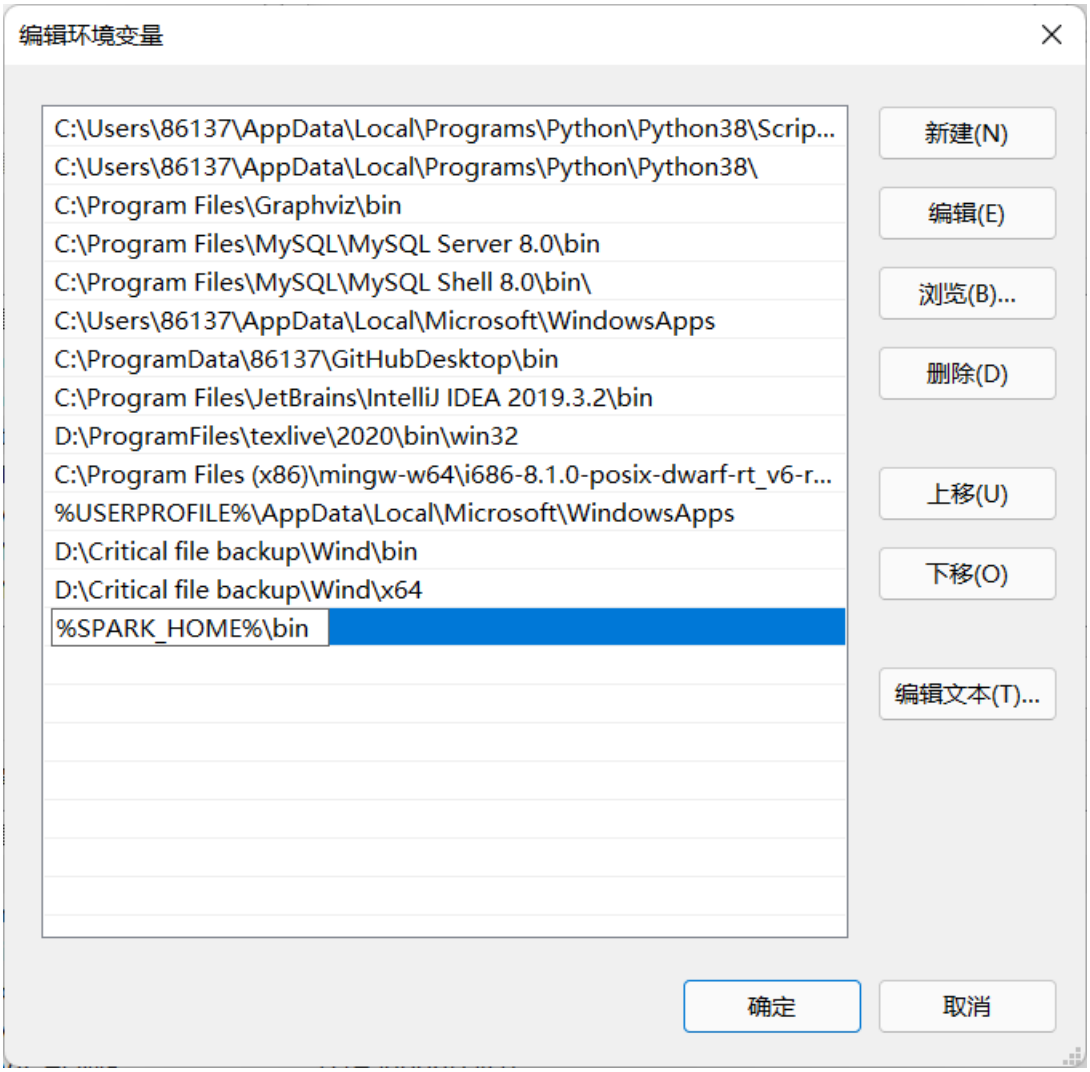
Name	Last modified	Size	Description
Parent Directory		-	
SparkR_2.4.5.tar.gz	2020-02-02 20:27	310K	
SparkR_2.4.5.tar.gz.asc	2020-02-02 20:27	819	
SparkR_2.4.5.tar.gz.sha512	2020-02-02 20:27	207	
pyspark-2.4.5.tar.gz	2020-02-02 20:27	208M	
pyspark-2.4.5.tar.gz.asc	2020-02-02 20:27	819	
pyspark-2.4.5.tar.gz.sha512	2020-02-02 20:27	210	
spark-2.4.5-bin-hadoop2.6.tgz	2020-02-02 20:27	220M	
spark-2.4.5-bin-hadoop2.6.tgz.asc	2020-02-02 20:27	819	
spark-2.4.5-bin-hadoop2.6.tgz.sha512	2020-02-02 20:27	268	
spark-2.4.5-bin-hadoop2.7.tgz	2020-02-02 20:27	222M	
spark-2.4.5-bin-hadoop2.7.tgz.asc	2020-02-02 20:27	819	
spark-2.4.5-bin-hadoop2.7.tgz.sha512	2020-02-02 20:27	268	
spark-2.4.5-bin-without-hadoop-scala-2.12.tgz	2020-02-02 20:27	139M	
spark-2.4.5-bin-without-hadoop-scala-2.12.tgz.asc	2020-02-02 20:27	819	
spark-2.4.5-bin-without-hadoop-scala-2.12.tgz.sha512	2020-02-02 20:27	193	
spark-2.4.5-bin-without-hadoop.tgz	2020-02-02 20:27	160M	
spark-2.4.5-bin-without-hadoop.tgz.asc	2020-02-02 20:27	819	
spark-2.4.5-bin-without-hadoop.tgz.sha512	2020-02-02 20:27	288	
spark-2.4.5.tgz	2020-02-02 20:27	15M	
spark-2.4.5.tgz.asc	2020-02-02 20:27	819	
spark-2.4.5.tgz.sha512	2020-02-02 20:27	195	

下载后解压。

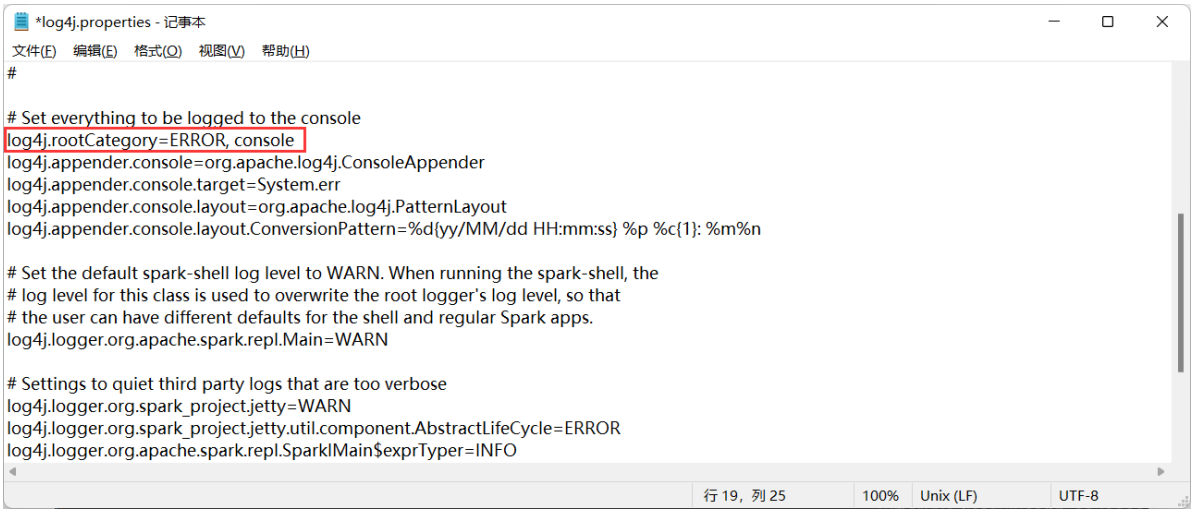
配置环境变量：新建变量SPARK_HOME



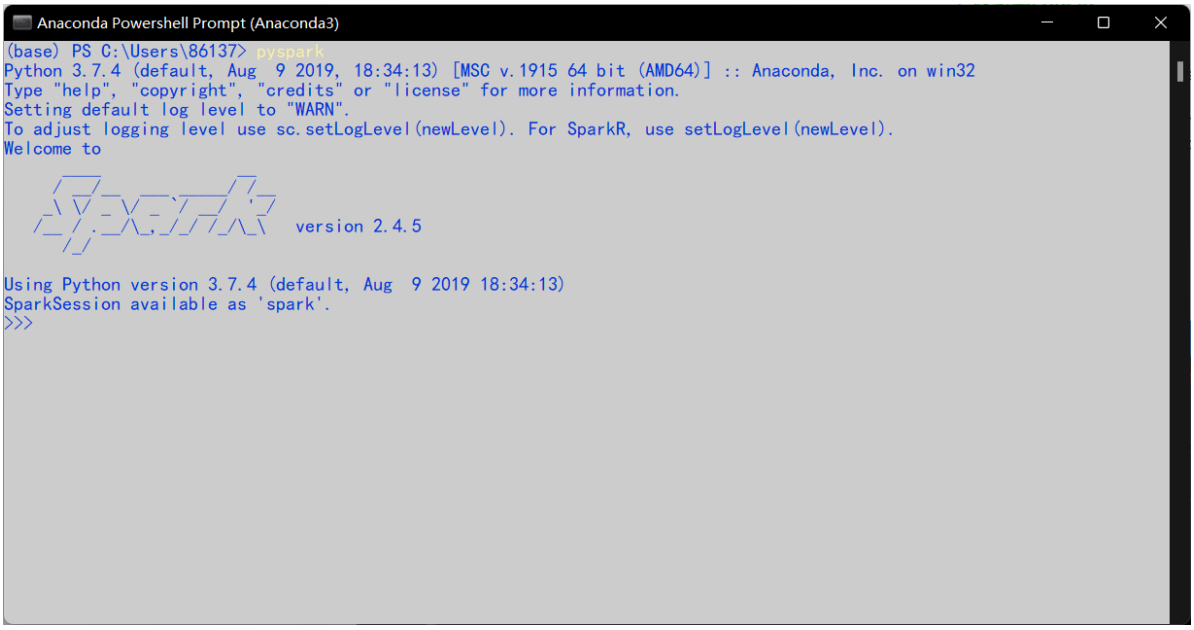
在Path变量中添加：



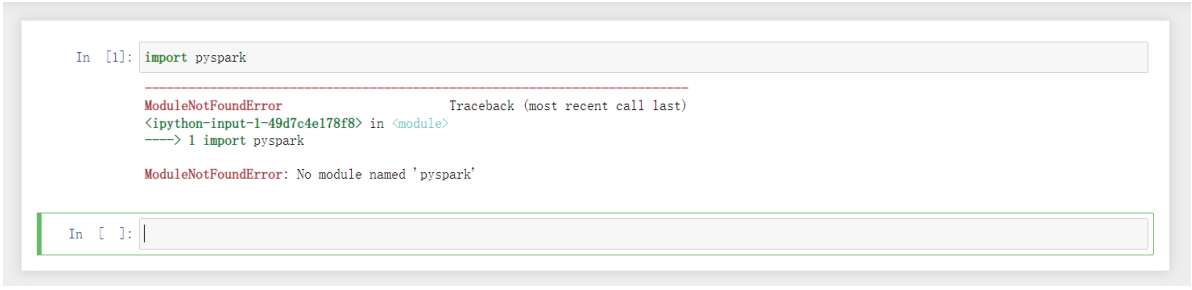
为了防止Spark每次运行时给出许多信息，影响结果的直观性，因此需要修改一下INFO设置。找到spark中的conf文件夹并打开，找到log4j.properties.template文件，复制一份修改文件名为log4j.properties，并写字板打开修改INFO为ERROR（或WARN）



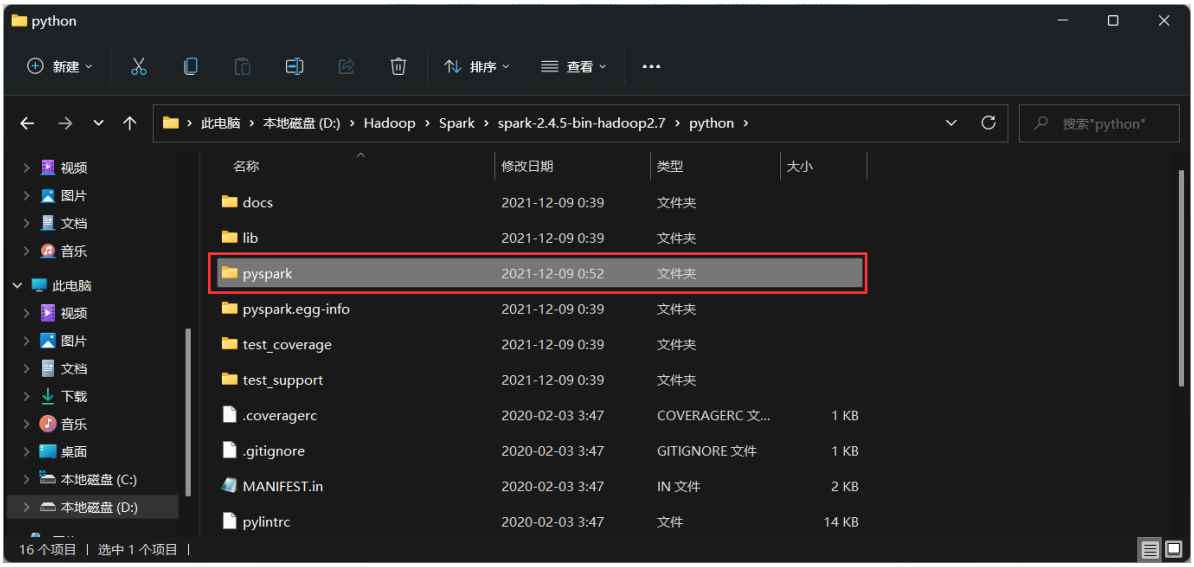
打开Anaconda命令行，输入pyspark验证是否安装成功：

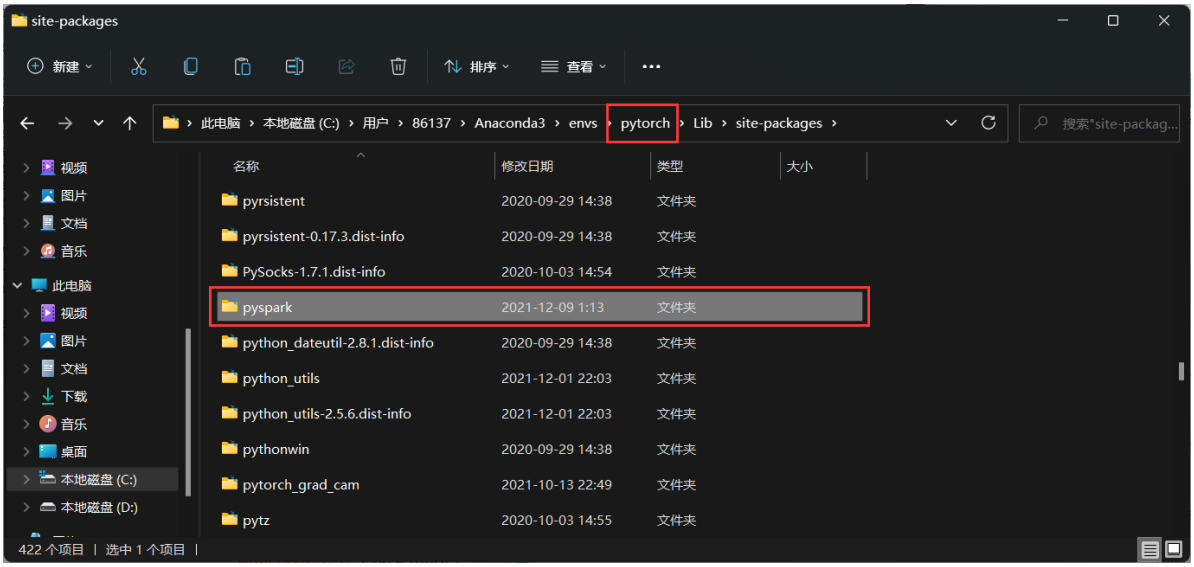


启动Jupter Notebook，尝试导入spark，发现失败：



这是因为当前的Anaconda的环境中还没有pyspark包，因此这里手动安装，把Spark路径下的pyspark文件夹复制粘贴到安装的Anaconda3下envs中对应的环境的Lib下的site-packages下面：





再次测试，发现提示没有py4j包：

```
In [1]: import pyspark

---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-49d7c4e178f8> in <module>
----> 1 import pyspark

ModuleNotFoundError: No module named 'pyspark'
```

```
In [4]: import pyspark

---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-4-49d7c4e178f8> in <module>
----> 1 import pyspark

~\Anaconda3\envs\pytorch\lib\site-packages\pyspark\__init__.py in <module>
     49
     50 from pyspark.conf import SparkConf
---> 51 from pyspark.context import SparkContext
     52 from pyspark.rdd import RDD, RDDBarrier
     53 from pyspark.files import SparkFiles

~\Anaconda3\envs\pytorch\lib\site-packages\pyspark\context.py in <module>
     27 from tempfile import NamedTemporaryFile
     28
---> 29 from py4j.protocol import Py4JError
     30
     31 from pyspark import accumulators

ModuleNotFoundError: No module named 'py4j'
```

使用Anaconda在当前环境中安装该包：

```
Anaconda Powershell Prompt (Anaconda3)
(base) PS C:\Users\86137> conda activate pytorch
(pytorch) PS C:\Users\86137> pip install py4j
Collecting py4j
  Using cached py4j-0.10.9.3-py2.py3-none-any.whl (198 kB)
Installing collected packages: py4j
Successfully installed py4j-0.10.9.3
(pytorch) PS C:\Users\86137>
```

再次测试，发现导入成功：

```
In [5]: import pyspark

In [6]: pyspark?

In [ ]:
```

```
Type:        module
String form: <module 'pyspark' from 'C:\\Users\\86137\\Anaconda3\\envs\\pytorch\\lib\\site-packages\\pyspark\\__init__.py'>
File:        c:\users\86137\anaconda3\envs\pytorch\lib\site-packages\pyspark\__init__.py
Docstring:
PySpark is the Python API for Spark.

Public classes:

  - :class:`SparkContext`:
      Main entry point for Spark functionality.
  - :class:`RDD`:
      A Resilient Distributed Dataset (RDD), the basic abstraction in Spark.
  - :class:`Broadcast`:
      A broadcast variable that gets reused across tasks.
  - :class:`Accumulator`:
      An "add-only" shared variable that tasks can only add values to.
  - :class:`SparkConf`:
      For configuring Spark.
  - :class:`SparkFiles`:
      Access files shipped with jobs.
```