python的Spark编程

2022年11月16日 16:49

首先配置相关环境

cd ~
vim .bashrc
export PATH=$PATH:$SPARK_HOME/bin
export PYSPARK_DRIVER_PYTHON=jupyter
export PYSPARK_DRIVER_PYTHON_OPTS='notebook --packages julioasotodv:spark-tree-plotting:0.2'
source .bashrc
注意确保前面已经配置$SPARK_HOME
第一个导入bin目录下有pyspark运行文件，能够直接运行pyspark命令
第二个导入配置是运行pyspark时的IDE
第三个导入配置是运行IDE时的参数设置
启动spark：
$HADOOP_HOME/sbin/start-all.sh

运行pyspark

首先将jupyter程序文件和数据集传入Ubuntu中
cd ~
mkdir algorithm_application
cd ~/algorithm_application
mv dataset.csv .
mv app.ipynb .
将数据集传到分布式文件系统中
hdfs dfs -put dataset.csv input
以jupyter notebook启动pyspark
pyspark

代码介绍

```
import numpy as np
from pyspark.ml.feature import StringIndexer, VectorIndexer, StandardScaler,
VectorAssembler
import pandas as pd
import matplotlib.pyplot as plt
import emoji
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, IDF, Tokenizer
from pyspark.ml.feature import StringIndexer
from pyspark.ml import Pipeline
from pyspark import Row
from pyspark.ml.feature import PCA
from pyspark.ml.linalg import Vectors,DenseVector,VectorUDT
from pyspark.sql.functions import col
```

利用spark在分布式文件系统读取数据集

```
dataset_df = spark.read.format("csv").option("header", "true") .option("inferSchema",
"true").load('input/dataset.csv')
```

spark为pyspark自带对象

' . '就相当于linux中的空格，表示参数，' ("") '内设置参数值

read	读取
formart	读取格式
load	读取文件位置（分布式文件系统中）
option	选项，header列标题，inferSchema字段名称设置

数据预处理

将数据转为dataframe类型

```
dataset_p = dataset_df.toPandas()
```

设置数据框字段名称

```
dataset_p.columns = ['target', 'text']
```

数据去缺失值、去重、去表情符号

```
import emoji

dataset_z = dataset_p.dropna()
dataset_z = dataset_z.drop_duplicates(keep='first',inplace='True',subset='text')
dataset_z['text'] = dataset_z['text'].apply(lambda x: emoji.replace_emoji(x, ""))
```

将数据变为spark数据库类型的数据框，用于后续处理

```
dataset_df = sqlContext.createDataFrame(dataset_z)
```

特征提取

首先定义操作：

```
#文本分词，默认是基于符号分词（只适合英文）
tokenizer = Tokenizer(inputCol="text", outputCol="words")
#词频矩阵，hashtf将词集transform到固定维度，是CountVectorizer+
hashtf = HashingTF(numFeatures=256, inputCol="words", outputCol='tf')
#TF-IDF类似计算，降低多文档高频词权重
idf = IDF(inputCol='tf', outputCol="features", minDocFreq=5)
#将上述操作放到pipeline中
pipeline = Pipeline(stages=[tokenizer, hashtf, idf])
```

数据集通过pipeline处理：

```
# “训练”操作模型
pipelineFit = pipeline.fit(dataset_df)
#处理数据集
dataset_df = pipelineFit.transform(dataset_df)
```

查看数据处理的结果，输出5个

```
dataset_df.show(5)
```

```
+------+--------------------+--------------------+-----------------+--------------------+
|target|                text|               words|               tf|            features|
+------+--------------------+--------------------+-----------------+--------------------+
|     1|专门选在工作日过来体验一下融汇温泉...|[专门选在工作日过来体验一下融汇温...| (256,[47],[1.0])|(256,[47],[5.2923...|
|     1|这次是夜场换票的人很多，说实话同程...|[这次是夜场换票的人很多，说实话同...|(256,[235],[1.0])|(256,[235],[5.134...|
|     1|评价有点晚了，完全是值啊值啊，一元...|[评价有点晚了，完全是值啊值啊，一...| (256,[28],[1.0])|(256,[28],[5.5606...|
|     1|一块钱能玩到已经很不错了，但是在同...|[一块钱能玩到已经很不错了，但是在...|(256,[210],[1.0])|(256,[210],[4.998...|
|     1|距主城温泉水质最棒哒哒的哦；员工服...|[距主城温泉水质最棒哒哒的哦；员工...|(256,[218],[1.0])|(256,[218],[5.353...|
+------+--------------------+--------------------+-----------------+--------------------+
```

每个outputCol都存放在里面

！！可以看到words里面有问题，不能使用spark的tokenizer分词正文，正确结果应该为：

```
+-----------------------------------+-----------------------------------------+------+
|sentence                           |words                                    |tokens|
+-----------------------------------+-----------------------------------------+------+
|Hi I heard about Spark             |[hi, i, heard, about, spark]             |5     |
|I wish Java could use case classes |[i, wish, java, could, use, case, classes]|7     |
|Logistic,regression,models,are,neat|[logistic,regression,models,are,neat]    |1     |
+-----------------------------------+-----------------------------------------+------+
```

建模前数据准备

配置标签索引

```
labelIndexer = StringIndexer(inputCol="target", outputCol="label")
labelIndexer = labelIndexer.fit(dataset_df)
dataset_df = labelIndexer.transform(dataset_df)
```

标准化处理

```
normalizer = StandardScaler(inputCol="features", outputCol="std_features")
normalizer_model = normalizer.fit(dataset_df)
dataset_labeled_df = normalizer_model.transform(dataset_df)
```

去掉无用的过程列，保留模型需要的标签和特征（就是数据库操作select）

```
dataset_labeled_df = dataset_labeled_df.select("std_features","label","features")
```

数据集分割

```
(training_data, test_data) = dataset_labeled_df.randomSplit([0.7, 0.3])
```

建模并训练、预测

配置模型参数

```
dtree = lr = LogisticRegression(featuresCol='std_features',labelCol='label',threshold=0.55,maxIter=256)
```

模型训练

```
model = lr.fit(training_data)
```

模型预测

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

predictions是结果数据框，包含真实标签label和预测标签prediction等

```
predictions.show(1)
```

```
+--------------------+-----+--------------------+--------------------+--------------------+----------+
|        std_features|label|            features|       rawPrediction|         probability|prediction|
+--------------------+-----+--------------------+--------------------+--------------------+----------+
|(256,[0],[14.8003...|  0.0|(256,[0],[5.37833...|[1.82418505924062...|[0.86106754358676...|       0.0|
+--------------------+-----+--------------------+--------------------+--------------------+----------+
only showing top 1 row
```

模型评估

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

准确率评估

```
acc_evaluator_dt = MulticlassClassificationEvaluator(
  labelCol="label", predictionCol="prediction", metricName="accuracy")
accuracy_dt = acc_evaluator_dt.evaluate(predictions)
print("Acc: %g" % (accuracy_dt))
```

F1值评估

```
f1_evaluator_dt = MulticlassClassificationEvaluator(
  labelCol="label", predictionCol="prediction", metricName="f1")
f1_score_dt= f1_evaluator_dt.evaluate(predictions)
print("F1: %g" % (f1_score_dt))
```

结果可视化

将SparkVector转为Array形式

```
from pyspark.sql.functions import UserDefinedFunction

def convert_to_dense(col):
        return DenseVector(col.toArray())

convert_to_dense_udf = UserDefinedFunction(lambda x: convert_to_dense(x))
```

```
test_dataset_labeled_df = test_data.withColumn("dense_features",
convert_to_dense_udf(col("std_features")))
```

利用PCA降维用于绘图

```
PCA_m = PCA(k=1, inputCol="dense_features", outputCol="pcaFeatures")
PCA_model_cluster = PCA_m.fit(test_dataset_labeled_df)
test_data_points =
PCA_model_cluster.transform(test_dataset_labeled_df).select("pcaFeatures","lab
el")
```

python绘图（这里给的案例太差，无意义）

设立x轴数据和y轴数据

```
test_x = list(map(lambda x: x[0][0], test_data_points_l))
test_y = list(map(lambda x: x[1], test_data_points_l))
```

```
print(test_x[0])
print(test_y[0])
```

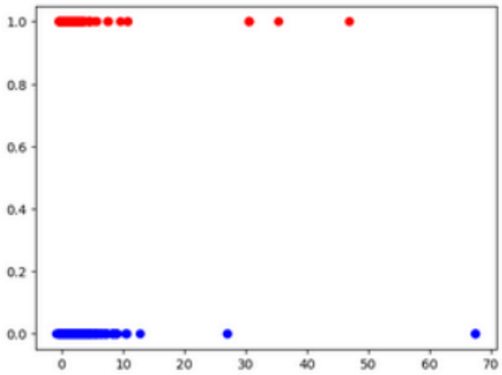
```
0.34789795441153376
0.0
```

设置点的颜色

```
colors_1 = ['red' if label == 1 else 'blue' for label in test_y]
```

绘图

```
plt.scatter(test_x, test_y, c = colors_1)
```



补充：

①spark数据库dataframe操作

select、where等基本数据库操作

pandas与spark dataframe转换

```
dataframe = sparkDataframe.select("colname").toPandas()
sparkDataframe = sqlContext.createDataFrame(dataframe)
```

对spark dataframe中某一列数据进行操作生成新的列，spark vector转为array类型

```
sparkDataframe.withColumn("newColName",ProcessFun(col("colName")))
```

newColName为生成字段的名称

ProcessFun为处理函数

colName为要被处理的字段

其中ProcessFun为用户自定义函数，内容为lambda表达式

```
ProcessFunc = UserDefinedFunction(lambda x: MyFunc(x),VectorUDT())
```

MyFunc为实际的处理函数

VectorUDT()是为了数据在转为array后spark仍能进行处理

处理流程是列数据一个个传入执行（经过print实验得出）

```
def MyFunc(col):
    return DenseVector(col.toArray())
```

这里的功能为将sparkvector转为vector

将spark dataframe某一列取出来为列表类型

```
dataList = SparkDataframe.rdd.map(lambda x: x[0]).collect()
```

这里取第一列数据，不难看出利用lambda表达式可进行其他操作

lambda表达式中的x表示的是每一行的数据

将spark dataframe某一列取出来的同时将spark vector转为array类型

```
dataArrayList = SparkDataframe.select("probability").rdd.map(lambda x:x[0].toArray()).collect()
```

②SparkVector可以像np.array进行运算，虽然看起来不同