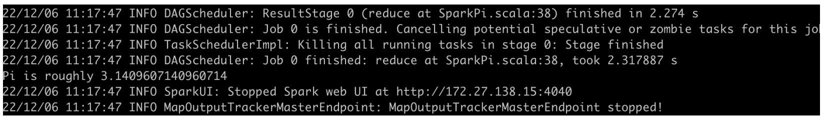
实验四实验报告

1.spark安装

在https://spark.apache.org/downloads.html下载spark安装包，将安装包解压后移动到`/usr/local/Cellar`路径下。运行样例脚本如下，正常输出结果。

```
bin/spark-submit --class org.apache.spark.examples.SparkPi --master
'local[2]' ./examples/jars/spark-examples_2.12-3.3.1.jar 100
```



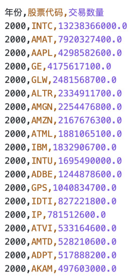
```
22/12/06 11:17:47 INFO DAGScheduler: ResultStage 0 (reduce at SparkPi.scala:38) finished in 2.274 s
22/12/06 11:17:47 INFO DAGScheduler: Job 0 is finished. Cancelling potential speculative or zombie tasks for this jo
22/12/06 11:17:47 INFO TaskSchedulerImpl: Killing all running tasks in stage 0: Stage finished
22/12/06 11:17:47 INFO DAGScheduler: Job 0 finished: reduce at SparkPi.scala:38, took 2.317887 s
Pi is roughly 3.1409607140960714
22/12/06 11:17:47 INFO SparkUI: Stopped Spark web UI at http://172.27.138.15:4040
22/12/06 11:17:47 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
```

2.任务一

2.1 编写Spark程序，统计stocks_small.csv表中每支股票每年的交易数量，并按年份，将股票交易数量从大到小进行排序。

利用spark.read读取stock_small.csv，将其存储为dataframe。按年份，先筛选出对应年份的交易，再根据stock_symbol对交易量进行求和，最后从大到小进行排序，将每年的统计结果单独存在一张表中，具体运行结果保存在result/task1_1文件夹中，每年的结果保存在单独的表格中。对应的代码以及运行结果截图如下。

```
tmp = df.filter(func.year(df['date']) == y)
tmp = tmp.groupBy('stock_symbol').agg({'stock_volume': 'sum'})
ans = tmp.sort(tmp['sum(stock_volume)'].desc())
```



```
年份,股票代码,交易数量
2000,INTC,13238366000.0
2000,AMAT,7920327400.0
2000,AAPL,4298582600.0
2000,GE,4175617100.0
2000,GLW,2481568700.0
2000,ALTR,2334911700.0
2000,AMGN,2254476800.0
2000,AMZN,2167676300.0
2000,ATML,1881065100.0
2000,IBM,1832906700.0
2000,INTU,1695490000.0
2000,ADBE,1244878600.0
2000,GPS,1040834700.0
2000,IDTI,827221800.0
2000,IP,781512600.0
2000,ATVI,533164600.0
2000,AMTD,528210600.0
2000,ADPT,517888200.0
2000,AKAM,497603000.0
```

2.2 编写Spark程序，统计stocks_small.csv表中收盘价（price_close）比开盘价（price_open）差价最高的前十条记录。

利用spark.read读取stock_small.csv，将其存储为dataframe。计算每条记录收盘价与开盘价差的绝对值，然后从大到小进行排序，选取前十条保存。对应的代码以及运行结果截图如下。

```
df = df.select(df['exchange'], df['stock_symbol'], df['date'],
df['stock_price_close'], df['stock_price_open'],
(func.abs(df['stock_price_close'] - df['stock_price_open'])).alias('tmp'))
ans = df.sort(df['tmp'].desc()).limit(10)
```

```
交易所,股票代码,交易日期,收盘价,开盘价,差价
NASDAQ,IGLD,2000-04-17,9.5,33821.0,33811.5
NASDAQ,ATCO,2000-03-16,9.38,31007.0,30997.62
NASDAQ,AWRE,2000-06-06,51.25,29595.0,29543.75
NASDAQ,ISSI,2000-04-11,25.5,3028.0,3002.5
NASDAQ,INFY,2000-02-11,670.06,534.5,135.55999999999995
NASDAQ,ARWR,2000-07-13,155.95,249.92,93.97
NASDAQ,ARWR,2000-07-18,155.95,249.92,93.97
NASDAQ,INFY,2000-02-10,528.5,441.5,87.0
NASDAQ,INFY,2000-02-14,543.0,621.0,78.0
NASDAQ,INCY,2000-03-14,143.5,200.5,57.0
```

3.任务二

3.1 统计IBM公司（stock_symbol = IBM）从2000年起所有支付股息的交易日（dividends表中有对应记录）的收盘价（stock_price_close）。

利用spark.read读取stock_small.csv和dividens_small.csv文件，并创建视图，利用spark_sql以及以下sql语句先统计出从2000年取所有支付股息的交易日，再统计出对应交易日的收盘价。

```
select date, stock_symbol, stock_price_close from stock_small where
stock_symbol = 'IBM' and date in (select date from dividends_small where
symbol = 'IBM')
```

将结果保存在csv文件中，结果截图如下所示。

```
交易日期,股票代码,收盘价
2010-02-08,IBM,121.88
2009-11-06,IBM,123.49
2009-08-06,IBM,117.38
2009-05-06,IBM,104.62
2009-02-06,IBM,96.14
2008-11-06,IBM,85.15
2008-08-06,IBM,129.16
2008-05-07,IBM,124.14
2008-02-06,IBM,103.59
2007-11-07,IBM,111.08
2007-08-08,IBM,112.98
2007-05-08,IBM,103.29
2007-02-07,IBM,99.54
2006-11-08,IBM,92.59
2006-08-08,IBM,75.33
2006-05-08,IBM,82.89
2006-02-08,IBM,80.8
2005-11-08,IBM,83.15
2005-08-08,IBM,83.36
```

3.2 统计苹果公司 (stock_symbol = AAPL) 年平均调整后收盘价(stock_price_adj_close) 大于50美元的年份以及当年的年平均调整后收盘价。

同样地，利用spark.read读取stock_small.csv文件，并创建视图，利用spark_sql以及以下sql语句先求出苹果公司每年的年平均调整后收盘价，再筛选出其中大于50美元的年份。

```
select year(date) as year, stock_price_adj_close as price from stock_small
where stock_symbol = 'AAPL'
select year, avg from (select year, AVG(price) as avg from AAPL group by
year) where avg > 50
```

将结果保存在csv文件中，结果截图如下所示。

```
年份,年平均调整后收盘价
2006,70.81063745019918
2007,128.2739043824701
2008,141.97901185770743
2009,146.81412698412706
2010,204.7216
```

4.任务三：根据表stock_data.csv 中的数据，基于Spark MLlib 或者Spark ML 编写程序在收盘之前预测当日股票的涨跌，并评估实验结果的准确率。

首先注意到stock_data.csv中的数据保存的类型是string，因此需要进行数据类型的转换。

```
for x in ['stock_price_open', 'stock_price_high', 'stock_price_low',
'stock_volume', 'label']:
    df = df.withColumn(x, df[x].astype('float'))
```

接着需要划分特征和想要预测的标签。

```
vectorAssembler = VectorAssembler(inputCols=['stock_price_open',
'stock_price_high', 'stock_price_low', 'stock_volume'], outputCol =
'features')
new_df = vectorAssembler.transform(df)
new_df = new_df.select(['features', 'label'])
```

同时，需要去除重复数据和缺失值。

```
new_df = new_df.dropDuplicates()
new_df = new_df.na.drop()
```

按8:2的比例划分数据集，得到训练集和测试集。

```
train, test = new_df.randomSplit([0.8, 0.2], seed = 10)
```

接下来利用不同的模型进行训练，以对率回归为例，首先训练模型。

```
lr = LogisticRegression(featuresCol='features', labelCol='label')
lr_model = lr.fit(train)
```

最后根据训练完成的模型对测试集进行预测，然后计算出具体的tp，tn，fp，fn，进而计算出准确率，查准率，查全率和f1值。

```
predictions = lr_model.transform(test)
lr_evaluator = BinaryClassificationEvaluator().setLabelCol('label')
accuracy = lr_evaluator.evaluate(predictions)
tp = predictions[(predictions.label == 1) & (predictions.prediction ==
1)].count()
tn = predictions[(predictions.label == 0) & (predictions.prediction ==
0)].count()
```

```
    fp = predictions[(predictions.label == 0) & (predictions.prediction ==
    1)].count()
    fn = predictions[(predictions.label == 1) & (predictions.prediction ==
    0)].count()
    recall = float(tp) / (tp + fn)
    precision = float(tp) / (tp + fp)
    f1 = 2 * recall * precision / (recall + precision)
    result.append(['Logistic Regression', accuracy, precision, recall, f1])
```

实验中一共使用了四种不同的模型：对率回归，决策树，随机森林，朴素贝叶斯。模型评估结果保存在result文件夹中的task3.csv中。观察结果可以发现，对率回归的准确率较高，达到80%，但四个模型的f1值都偏低，可能的原因是模型普遍会判断当日股票下跌，使得f1偏低。因此需要改进数据以及数据输入的特征，实现更可靠的预测。

```
Model,Accuracy,precision,recall,f1
Logistic Regression,0.8138762115285253,0.7030181258549931,0.2040018855753486,0.316237837006269
Decision Tree,0.6283628917692095,0.6180221678422341,0.26630030268446386,0.37221600908562863
Random Forest,0.7332240782206872,0.6016784091182341,0.3104004366595544,0.40952871954762315
Naive Bayes,0.6941525485041792,0.3886248053969901,0.23224830050116607,0.2907436309565406
```