تمرین کامپیوتری سوم (بخش اول) درس یادگیری ماشین توزیع شده

پدرام رستمی – ۸۱۰۱۰۰۳۵۳

سوال ۱

الف) برای حل این سوال، ابتدا فایل City.txt در آدرس /rostami/City.txt در hdfs قرار داده می‌شود. سپس به کمک دستور .read.text() خوانده می‌شود. سپس dataframe با دستور .collect() به data_collect تبدیل شده و تا بتوان بر روی هر سطر ورودی حلقه زد. در حلقه‌ی زده شده تعداد تکرار هر شهر به دست آمده و در dataframe ای با ستون‌های city و repetition ذخیره می‌شود. در نهایت به کمک دستورهای .select() ، .col() و .lit() خروجی نهایی مطابق فرمت داده شده، تولید می‌شود و در hdfs در آدرس /rostami/Spark_rdd_1 ذخیره می‌شود. کد این بخش در شکل ۱ قابل مشاهده است.

```python
from pyspark.sql import SparkSession
from collections import defaultdict
from pyspark.sql.functions import concat, col, lit

path = "hdfs://raspberrypi-dml0:9000/rostami/City.txt"

if __name__ == '__main__':
    spark = SparkSession \
        .builder \
        .appName("Pedram app 1") \
        .getOrCreate()
    df = spark.read.text(path)
    data_collect = df.collect()
    cities = defaultdict(lambda: 0)
    for row in data_collect:
        line = row['value']
        cieties_in_line = line.split(',')
        for city in cieties_in_line:
            cities[city] += 1
    out_data = []
    for city in list(cities.keys()):
        out_data.append((city, cities[city]))
    out_df = spark.createDataFrame(out_data, ('city', 'repetition'))
    temp = out_df.select(concat(col("city"), lit(": "), col("repetition")))
    temp.coalesce(1).write.format("text").option("header", "false").mode("append").save("hdfs://raspberrypi-dml0:9000/rostami/Spark_rdd_1")
```

شکل ۱- پیاده سازی تعداد تکرار شهرها

بخشی از خروجی این بخش در شکل ۲ قابل مشاهده است و خروجی کامل آن در فایل Spark_rdd_1.txt آورده شده است.

```
Ahvāz: 1
Qom: 1
Rasht: 6
Bābol: 1
Qāyen: 1
Marāgheh: 5
Sārī: 6
Kermān: 3
Zāhedān: 2
Tehrān: 2
Fīrūzābād: 4
Bam: 1
Kāzerūn: 2
Dāmghān: 3
Yesuj: 4
Āmol: 5
```

شکل ۲- بخشی از خروجی نهایی تعداد تکرار شهرها

ب) کد این بخش در شکل ۳ قابل مشاهده است. بخش‌های ابتدایی مانند خواندن فایل متنی مانند بخش الف انجام شده است. در حلقه‌ای که بر روی داده‌ها زده می‌شود، ابتدا با دستور split() شهرها از همدیگر جدا شده و سپس به کمک دستور sort() مرتب می‌شوند. در نهایت شهرها با , به هم متصل شده و در هر سطر از dataframe نهایی ذخیره شده و سطرهای dataframe نهایی در آدرس /rostami/Spark_rdd_2 ذخیره می‌شوند.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import concat, col, lit

path = "hdfs://raspberrypi-dml0:9000/rostami/City.txt"

if __name__ == '__main__':
    spark = SparkSession \
        .builder \
        .appName("Spark rdd 2") \
        .getOrCreate()
    df = spark.read.text(path)
    data_collect = df.collect()
    all_cities = []
    for row in data_collect:
        line = row['value']
        cities = line.split(',')
        cities.sort()
        all_cities.append(cities)
    out_data = []
    for cities in all_cities:
        out_data.append((','.join(cities), ))
    out_df = spark.createDataFrame(out_data, ('cities', ))
    temp = out_df.select(concat(col("cities")))
    temp.coalesce(1).write.format("text").option("header", "false").mode("append").save("hdfs://raspberrypi-dml0:9000/rostami/Spark_rdd_2")
```

شکل ۳- پیاده سازی مرتب سازی اسامی شهرها در هر خط

همچنین در شکل ۴ خروجی این بخش قابل مشاهده است.

```
Ahvāz,Bābol,Fīrūzābād,Kermān,Marāgheh,Qom,Qāyen,Rasht,Sārī,Tehrān,Zāhedān
Bam,Dāmghān,Kāzerūn,Marāgheh,Qazvīn,Rasht,Shūshtar,Sārī,Yesuj,Āmol
Dezfūl,Eṣfahān,Fīrūzābād,Gorgān,Hamadan,Marāgheh,Yesuj,Zanjān,Āmol
Dāmghān,Eṣfahān,Fīrūzābād,Kermānshāh,Kāshān,Rasht,Sanandaj,Zanjān,Īlām
Behbehān,Dārāb,Khorramābād,Khoy,Kāzerūn,Mahābād,Rasht
Fīrūzābād,Khoy,Mashhad,MasjedSoleymān,Neyshābūr,Sārī,Tabrīz
Kermān,Khorramshahr,Kāshān,Orūmīyeh,Sārī,Yesuj,Ābādān,Āmol
Borūjerd,Khorramshahr,Marāgheh,Qūchān,Rasht,Sanandaj,Semnān,ShahrKord,Sārī,Āmol
Dāmghān,Kermān,Marāgheh,Shīrāz,Tabrīz,Tehrān,Ţāq-eBostān
Ardabīl,BandarAnzalī,Rasht,Sārī,Yazd,Yesuj,Zāhedān,Āmol
```

شکل ۴- مرتب سازی شهرهای هر سطر

سوال ۲

الف) در این بخش هم فایل data.csv در hdfs و در آدرس /rostami/data.csv قرار داده می‌شود تا از آنجا خوانده شود. در ابتدا برای نمایش ۵ سطر ابتدایی داده از دستور show(5). استفاده می‌شود. در شکل ۵ خروجی این دستور نمایش داده شده است.

```
+--------+--------+--------+--------+------+
|feature1|feature2|feature3|feature4| label|
+--------+--------+--------+--------+------+
|    8.34|   40.77| 1010.84|   90.01|480.48|
|   23.64|   58.49|  1011.4|    74.2|445.75|
|   29.74|    56.9| 1007.15|   41.91|438.76|
|   19.07|   49.69| 1007.22|   76.79|453.09|
|    11.8|   40.66| 1017.13|    97.2|464.43|
+--------+--------+--------+--------+------+
only showing top 5 rows
```

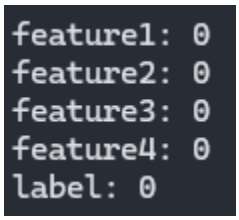
شکل ۵- نمایش ۵ سطر ابتدایی داده

برای نمایش مولفه‌های آماری مینیمم، ماکزیمم، میانگین و واریانس کافی است از دستور describe(). استفاده شود. در شکل ۶ خروجی این دستور نمایش داده شده است.

```
+-------+-----------------+-----------------+-----------------+-----------------+-----------------+
|summary|         feature1|         feature2|         feature3|         feature4|            label|
+-------+-----------------+-----------------+-----------------+-----------------+-----------------+
|  count|             9568|             9568|             9568|             9568|             9568|
|   mean|19.651231187291014|54.305803720735966|1013.2590781772483| 73.30897784280928|454.3650940063547|
| stddev| 7.452473229611075|12.707892998326807| 5.938783705811638|14.600268756728957|17.06699499980342|
|    min|             1.81|            25.36|           992.89|            25.56|           420.26|
|    max|            37.11|            81.56|           1033.3|           100.16|           495.76|
+-------+-----------------+-----------------+-----------------+-----------------+-----------------+
```

شکل ۶- بررسی مولفه‌های آماری داده

در نهایت برای شمردن تعداد Null ها در هر ستون، ابتدا دستور data[col].isNull() را برای هر ستون صدا زده و به کمک data.filter() سطرهایی که آن ستون در آن‌ها Null است را پیدا می‌کنیم. درنهایت به کمک تابع .count() تعداد آن‌ها را می‌شماریم. خروجی این بخش در شکل ۷ قابل مشاهده است. همچنین کدهای مربوط به این بخش در فایل Pre_Model.py قابل مشاهده‌اند.



شکل ۷- تعداد مقادیر Null در هر ستون

ب) برای تقسیم داده‌ها به صورت تصادفی، می‌توان از دستور .randomSplit() استفاده کرد. برای تقسیم به صورت ۸۰ به ۲۰، کافیست ورودی [0.8, 0.2] به آن داده شود. همچنین برای تقسیم تصادفی ولی یکسان بین مدل‌های دیگر، مقدار seed برای همه‌ی مدل‌ها ۱۶۶ داده می‌شود.

ج و د) برای پیاده سازی مدل‌ها، در ابتدا لازم است به کمک VectorAssembler() تمام ستون‌های ویژگی به یک ستون تبدیل شوند. سپس به کمک مدل‌های رگرسیون پیاده سازی شده (مانند LinearRegression)، مدل آموزش می‌بینند. به کمک Pipeline می‌توان این دو مرحله را در کنار همدیگر قرار داده تا همزمان با هم بر روی داده‌ها انجام شوند. سپس کافیست Pipelineایجاد شده را بر روی داده‌های آموزش fit کرده و پیشبینی آن‌ها را برای داده‌های تست به دست آورد. در نهایت هم ۵ سطر ابتدایی از داده‌های تست که شامل پیشبینی مدل هم می‌شود، نشان داده می‌شوند. پیاده سازی این بخش برای مدل LinearRegression در شکل ۸ قابل مشاهده است.

```
stage_assembler = VectorAssembler(inputCols = ['feature1', 'feature2', 'feature3', 'feature4'], outputCol = 'features')
stage_regressor = LinearRegression(featuresCol='features', labelCol='label')
regression_pipeline = Pipeline(stages= [stage_assembler, stage_regressor])
model = regression_pipeline.fit(train_df)
out_df = model.transform(test_df)
out_df.show(5)
```

شکل ۸- پیاده سازی آموزش مدل LinearRegression

همچنین برای ارزیابی عملکرد مدل‌ها، کافیست تا مقادیر پیشبینی شده و واقعی را به صورت متناظر در لیستی از Tuple ها ذخیره کرده و تابع .parallelize() را اجرا کنیم. سپس می‌توان به کمک RegressionMetrics() امتیاز مدل در معیارهای مختلف مانند Root Mean Square Error و R^2 را به دست آورد. پیاده سازی این بخش در شکل ۹ قابل مشاهده است.

```
out_data = out_df.collect()
exp_pred = []
for row in out_data:
    exp_pred.append((row['label'], row['prediction']))
exp_pred = spark.sparkContext.parallelize(exp_pred)
metrics = RegressionMetrics(exp_pred)
print(f'RMSE: {metrics.rootMeanSquaredError}, r2: {metrics.r2}')
```

شکل ۹- پیاده سازی ارزیابی مدل در معیارهای مختلف

در جدول ۱ مقادیر خطای R^2 و Root Mean Square Error برای مدل‌های خواسته شده آورده شده است. معیار R^2 مشخص می‌کند که چقدر مقادیر پیشبینی شده به مقادیر واقعی نزدیک هستند و هر چقدر به ۱.۰ نزدیک‌تر باشد، پیشبینی‌های مدل بهتر بوده است. طبق امتیازهای مدل‌ها در این معیار، مدل Gradient-Boosted Trees بهترین مدل بوده است. همچنین در معیار Root Mean Square Error فاصله‌ی پیشبینی‌ها با مقادیر واقعی محاسبه می‌شود و هر چقدر کمتر باشد، پیشبینی‌های مدل بهتر بوده است. در این معیار هم مدل Gradient-Boosted Trees بهترین مدل بوده است.

	R^2	*Root Mean Square Error*
Linear Regression	0.9235	4.5549
Decision Tree	0.9253	4.4844
Random Forrest	0.9313	4.2079
Gradient-Boosted Trees	0.9388	4.0925

جدول ۱- مقایسه‌ی مدل‌های رگرسیون

در بین الگوریتم‌های رگرسیونی که از درخت تصمیم استفاده می‌کنند، Random Forrest به دلیل درخت‌های مختلفی که می‌سازد، generalization بیشتری داشته و از Decision Tree بهتر است. همچنین Gradient-Boosted Trees به دلیل اینکه درخت‌های بهتری نسبت به Random Forrest می‌سازد، عملکرد بهتری دارد. زیرا Random Forrest درخت‌ها را مستقل از همدیگر تولید می‌کند. در حالی که Gradient-Boosted Trees درخت‌های جدید را بر اساس درخت‌های ساخته‌ی پیشین تولید می‌کند.

سوال ۳

در این سوال از همان کد GBT_Model.py در سوال دوم استفاده شد. تنها برای اختصاص نام صحیح، نام app به صورت متغیر داده می‌شود. در این بخش تنها با تغییر پارامترهای –total-executor-cores و –executor-cores، بهترین مدل (Gradient-Boosted Trees) در حالت‌های مختلف اجرا می‌شود.

برای اجرای بهترین مدل بر روی ۱ ماشین و ۱ هسته از دستور زیر استفاده شده است:

```
$ spark-submit –total-executor-cores 1 –executor-cores 1 Best_Model.py Best_Model_1_Node_1_Core
```

برای اجرای بهترین مدل بر روی ۱ ماشین و ۲ هسته از دستور زیر استفاده شده است:

```
$ spark-submit --total-executor-cores 2 Best_Model.py Best_Model_1_Node_2_Cores
```

برای اجرای مدل بر روی ۲ ماشین و ۱ هسته از هر کدام، از دستور زیر استفاده شده است:

```
$ spark-submit --total-executor-cores 2 --executor-cores 1 Best_Model.py Best_Model_2_Nodes_1_Core
```

همچنین برای اجرای مدل بر روی ۲ ماشین و ۲ هسته از هر کدام، از دستور زیر استفاده شده است:

```
$ spark-submit --total-executor-cores 4 --executor-cores 2 Best_Model.py Best_Model_2_Nodes_2_Cores
```

در جدول ۲ نتایج مقایسه‌ی اجرای مدل در حالت‌های مختلف آمده است. ستون‌های R^2 و RMSE که مربوط به خطا هستند و در طول اجراهای مختلف تغییری نکرده است. ستون Com Time زمان اجرای محاسباتی که spark در هنگام اجرای دستور در terminal می‌نویسد، است و ستون Tot Time زمان اجرای کلی است که در Spark Master قابل مشاهده است. در این جدول با افزایش تعداد هسته‌ها یا ماشین‌ها زمان محاسبات افزایش می‌یابد زیرا communication time زیاد می‌شود. همچنین از آنجایی که communication time بین هسته‌ها کمتر از ماشین‌ها است، افزایش ماشین‌ها از افزایش هسته‌ها در افزایش زمان محاسبات بیشتر است.

	R^2	*RMSE*	*Com Time (s)*	*Tot Time (min)*
1 Node, 1 Core	0.9388	4.0925	1.7178	2.7
1 Node, 2 Cores	0.9388	4.0925	2.9658	2.6
2 Nodes, 1 core	0.9388	4.0925	3.5003	2.5
2 Nodes, 2 Cores	0.9388	4.0925	4.4722	2.6

جدول ۲- مقایسه‌ی مدل‌ها در حالت اجرا روی چند هسته و ماشین

-- در فایل Best_Model.txt مقدار ۵ سطر اول و مقادیر R^2 و RMSE تمام اجراها قرار داده شده است. همچنین در فایل‌های Best_Model_x_Node(s)_x_Core(s).jpg اسکرین شاتی از application های اجرا شده در این بخش که تعداد ماشین‌ها و هسته‌های استفاده شده از آن‌ها مشخص است، قرار داده شده است.