Spark Project on Demographic Analysis of Turkey

Xiao Lulu
School of Data Science
Fudan University
17307130304@fudan.edu.cn

Abstract

This is a report on how to use Spark to handle massive amounts of data by operating on specific data. In python or other supported languages, by using some operators, we can tell Spark what we wish to do to compute with data, and as a result, it can construct an efficient query plan for execution. In this project, we not only use low-level RDD API , but also express the same query with high-level DSL operators and the DataFrame API. In addition,In addition, I use packages $MLlib$ designing machine learning pipelines in Spark to solve supervised learning problem such as classification and linear regression. This report contains experimental analysis, discussion of proposed methodology, errors raised in the process and the corresponding solutions.

1 Preprocess the data

1.1 About the data

The dataset contains some Turkish citizenship information which helps to develop an understanding of the age , sex, and geographical distribution of a population. This leak contains the following information for 49,611,709 Turkish citizens. The leaked database only includes adults of 18 or older and do not includes deceased citizens as of 2009.

1.2 Sample and split

Select the years and months from the date of birth to form new columns for extraction of features. As the number of the original data is too large, I sample about 2500 rows randomly from the dataset and split them into training, validation and test set with probability 0.7,0.2 and 0.1. Finally we can get 1789, 492 and 254 samples respectively.

```
df_h1 = df_format_date.sample(False, 0.00005, seed = 2018)
```

1.3 Convert categorical values into numeric values

Most machine learning models in $MLlib$ expect numerical values as input, represented as vectors. To convert categorical values into numeric values, we can one-hot encoding (OHE). Spark internally uses a $SparseVector$ when the majority of the entries are 0, so even though the number of features is large, OHE doesn't massively increase consumption of memory or compute resources. There are a few ways to one-hot encode our data with Spark. A common approach is to use the $StringIndexer$ and $OneHotEncoder$. With this approach, the first step is to apply the $StringIndexer$ estimator to convert categorical values into category indices. Once these category indices have been created, they'll be passed as input to the $OneHotEncoder$. Like this:

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

```
stringIndexer_features = StringIndexer(inputCols=feature_col,outputCols=
    indexOutputCols, handleInvalid="skip")
oheEncoder_features = OneHotEncoder(inputCols=indexOutputCols,outputCols=
    oheOutputCols)
```

1.4 Assemble the input columns into a single vector

For the task of putting all of our features into a single vector, we will use the $VectorAssembler$ transformer. It combines the values of a series of input columns into a single vector called $features$. Like this:

```
vecAssembler = VectorAssembler(inputCols=oheOutputCols, outputCol='features')
```

2 Train A Model

2.1 Select a proper model and correlative features

After setting up our $vectorAssembler$, we have our data prepared and transformed into a format that our linear regression model expects. Take $LogisticRegression$ for example, it is a type of estimator which learns parameters from the data. Our input column for logistic regression $(features)$ is the output from our $vectorAssembler$.

```
lr_h3 = LogisticRegression(featuresCol = 'features',labelCol='first_Index',
maxIter=100, regParam=0.01, elasticNetParam=0)
```

2.2 Fit and transform

$lr.fit()$ returns a $LogisticRegressionModel(lrModel)$, which is a transformer. Once the estimator has learned the parameters, the transformer can apply these parameters to new data points to generate predictions.

```
lrModel_h3 = lr_h3.fit(vecTrainDF_h3)
predictions = lrModel_h3.transform(validData)
```

3 Evaluate a model

3.1 Evaluate the model on the validation set

Now that we have built a model, we need to evaluate how well it performs.

For a logistic regression model, we use the auc to evaluate binary classification model, such as the gender . As for a multi-classification problem, we can inspect the accuracy on the validation set to evaluate it.

```
predictions = lrModel_h2.transform(data)evaluator =
    MulticlassClassificationEvaluator(labelCol='gender_Index',
predictionCol='prediction')
lrAcc = evaluator.evaluate(predictions)
```

For a linear regression problem, we will use $RMSE$ and R^2 to evaluate our models performance. To simplify the evaluation and avoid to define a baseline model to compare against, use R^2 instead to evaluate our model.

```
regresssionEvaluator = RegressionEvaluator(predictionCol='prediction',
labelCol='total', metricName='r2')
r2 = regresssionEvaluator.evaluate(predictions)
```

I first select all men from the dataframe. Because the number of oldest men may not only one, we'd better find the earliest date of birth and than go through each row to find the exact men corresponding to it.

3.2 Hyperparameter tuning

A machine learning model usually contains some hyperparameters which needs to be learned during the training process, such as learning rate and weight decay. We can set them a default value at first and do some fine tuning to it according to the accuracy or error on the validation set. It depends on the model and dataset.

4 Problem

4.1 Top 10 most populous cities

We first have to find the top 10 most populous cities and join it with the area table. Then we can calculate the density.

```
df_area = df_n6.join(df_area,'address_city','left_outer').orderBy('area')
density_df = df_area.withColumn('desity',
    round(df_area['total'] / df_area['area'],2) )
```

Here is the result:

```
+------------+-------+-----+-------+
|address_city|  total| area| desity|
+------------+-------+-----+-------+
|     ANTALYA|1288425| 1417| 909.26|
|     KOCAELI|1017832| 3418| 297.79|
|    ISTANBUL|8821021| 5343|1650.95|
+------------+-------+-----+-------+
only showing top 3 rows.
```

4.2 Proportion of the floating population

To calculate the proportion of the floating population, only need to count the number of citizens whose $id\_registration\_district$ and $address\_district$ are different, as with the cross-urban floating population.

```
df_n7_district = df_format_date.\
    select('id_registration_district','address_district').\
    filter(col('id_registration_district')!=col('address_district'))
propor_district = df_n7_district.count() / total_num
```

Here is the result:

```
Proportion of cross-district floating population:0.523
Proportion of cross-city floating population:0.361
```

4.3 Address-city Prediction Model

After processing the data, we can get two columns called features and labels. Then define a model, set the hyperparameters and apply The accuracy on the validation set is . To get a better prediction, I try to set different hyperparameters to select out properer ones.

By running the command above, we don't need to define a new model. After a few attempts, we can select the one which leads to the best result on the validation set. Table 1 shows the comporison of different hyperparameters.

Table 1: Comparison of regParam

regParam	0.3	0.01	0.001
valid accuracy	42.87%	60.50%	**62.79**%

And when we set regParam=0.001, the validation accuracy is 62.79%. In this problem, the label column have 81 distinct values. The model has a certain predictive ability to some extent. Lets lay our eyes on the top 5 rows. Then we get accuracy of 0.60 on test set. It predicts the result correctly with high probability, such as 0.97 in the fifth row.

```
+------------------+-----------------------------+------+----------+
|national_identifier|                  probability|labels|prediction|
+------------------+-----------------------------+------+----------+
|        49636475318|[0.986710724245762,9.223238...|   0.0|       0.0|
|        29641925480|[0.9747161839881305,0.00211...|   0.0|       0.0|
|        44974414456|[0.9728858078534585,0.00206...|   0.0|       0.0|
|        52075005138|[0.9691073561203233,0.00591...|   0.0|       0.0|
|        51211204220|[0.9688285044137115,0.00479...|   0.0|       0.0|
+------------------+-----------------------------+------+----------+
```

4.4 Gender Prediction Model

This is a binary classification problem, we can also use the logistic regression model. Refer to the last problem, we can set regParam=0.001 accordingly and test it on the validatation set.

```
lr_h2 = LogisticRegression(featuresCol = 'features',labelCol='gender_Index',
    maxIter=100, regParam=0.001, elasticNetParam=0)
lrPipeline_h2 = Pipeline(stages = [vecAssembler,lr_h2])
lrModel_h2 = lrPipeline_h2.fit(trainingData)
```

The test accuracy is 86.22%:

```
+------------------+----------------------+------+-----------+----------+
|national_identifier|           probability|gender|gender_Index|prediction|
+------------------+----------------------+------+-----------+----------+
|        28066584470|[0.1091706539,0.8908...|     E|        1.0|       1.0|
|        10456467890|[0.907792407,0.09220...|     K|        0.0|       0.0|
|        45157719504|[0.219206914,0.78079...|     E|        1.0|       1.0|
|        39589937452|[0.1858360022,0.8141...|     E|        1.0|       1.0|
|        15161744076|[0.195729451,0.80427...|     E|        1.0|       1.0|
+------------------+----------------------+------+-----------+----------+
```

4.5 Name Prediction Model

We can do the same operation as the previous question. Use logistic regression to predict the last name.

```
+------------------+----------------------+-------+----------+----------+
```

```
|national_identifier|           probability|   last|last_Index|prediction|
+-------------------+----------------------+-------+----------+----------+
|        34669954084|[0.075165695,0.01306...|BAHADIR|     107.0|       0.0|
|        51250399116|[0.0173439813,0.0140...|  TANKI|     345.0|      59.0|
|        23258189716|[0.0158237698,0.0148...|   IRAK|     230.0|       0.0|
|        14543163580|[0.01519594009,0.015...|  SENER|     331.0|       1.0|
|        31243805398|[0.01501948932,0.015...| TURGUT|      52.0|       1.0|
+-------------------+----------------------+-------+----------+----------+
```

We find it disapointing that the accuracy is close to 0 which is lower than baseline, that is, randomly predicting. These features do little help to the prediction. And in the top 5 rows, the probability is very low.

4.6 Population Growth Model

Count the number of births per year and I get two columns, 'year' and 'total'. As the there exists some abnormal data in the column $year$ such as '13–', we filter out those years after 1700. Then we can find out the earliest year is 1888 which is set to the benchmark year indexed 0 and get a new column $index$.

Ulike the problems above, we only split the dataset into two parts: training and test set.

```
def to_index(year):
    return year-1888

to_index_udf = udf(to_index, returnType=IntegerType())
min_year = df_population.select(min('year').alias('year')).collect()[0]
print(min_year)
new_df = df_population.withColumn('index',to_index_udf(df_population['year']))
new_df.show()

(trianing, test) = new_df.randomSplit([0.8,0.2],seed = 2020)
```

4.6.1 Ordinary linear regression

I first use the basic linear regression model to fit the growth curve.

```
lr_h4 = LinearRegression(featuresCol='features',labelCol='total')
lrModel_h4 = lr_h4.fit(vecTrainDF)
```

The we get predictions on the test set:

```
+-------+----+-----+--------+------------------+
|  total|year|index|features|        prediction|
+-------+----+-----+--------+------------------+
|1242772|1987|   99|  [99.0]|1199653.5968508604|
|1250520|1983|   95|  [95.0]|1138004.0989916562|
| 976960|1969|   81|  [81.0]| 922230.8564844418|
|1011805|1964|   76|  [76.0]| 845168.9841604368|
| 827839|1956|   68|  [68.0]| 721869.9884420284|
+-------+----+-----+--------+------------------+
```

Use R^2 to evaluate the power of the model:

```
regresssionEvaluator = RegressionEvaluator(predictionCol='prediction',
labelCol='total', metricName='r2')
r2 = regresssionEvaluator.evaluate(predictions)
```

Finally we get $R^2 = 0.932$ which means the model performs well.The formula for the linear regression lines is $num = 15412.37 * index - 326171.48$

4.6.2 LR with Malthus Model

As for a population growth prediction problem, we usually use Malthus and logistic model to solve it. The feature of Malthus model is that the percentage r of population growth per unit of time is constant. In the form of a differential equation, let the number of population at time t = 0 be N_0, then the total population at time t (N_t) satisfies

$$\frac{1}{N_t}\frac{dN_t}{dt} = r \;\Rightarrow\; N_t = N_0 e^{rt}$$

Thus we can deduce the growth equation:

$$N_t = N_0 e^{rt}$$
$$N_{t-1} = N_0 e^{r(t-1)}$$
$$\delta_t = \Delta N_t = N_0(1 - e^{-r})e^{rt}$$
$$log\ \delta_t = log(N_0(1 - e^{-r})) + rt$$

Do a logarithm transformation to the equation and we can get a new equation which is similar in form to a linear equation. So we can use linear regression to do prediction. Add a new column named $logTotal$.

```
+-------+----+-----+---------+
|  total|year|index| logTotal|
+-------+----+-----+---------+
| 785662|1959|   71|13.574282|
|1267087|1990|  102|14.052231|
|     32|1896|    8| 3.465736|
+-------+----+-----+---------+
```

Then assemble the features and build the model:

```
vecAssembler = VectorAssembler(inputCols=['index'],outputCol='features')
lr_h4_log = LinearRegression(featuresCol='features',labelCol='logTotal')
```

Finally we get $R^2 = 0.813$. The formula for the linear regression lines is $log(total) = 0.106 * index + 5.357$.

```
> predictions.show(5)
+-------+----+-----+---------+--------+------------------+
|  total|year|index| logTotal|features|        prediction|
+-------+----+-----+---------+--------+------------------+
|1242772|1987|   99|14.032855|  [99.0]|15.898212123293947|
|1250520|1983|   95| 14.03907|  [95.0]|  15.4722849068492|
| 976960|1969|   81|13.792201|  [81.0]|13.981539649292593|
|1011805|1964|   76|13.827247|  [76.0]|13.449130628736661|
| 827839|1956|   68|13.626574|  [68.0]| 12.59727619584717|
+-------+----+-----+---------+--------+------------------+
```

4.7 Conclusion

In this report, we mainly cover how to solve the machine learning problems using Spark. It includes how to convert a series of feature columns, how to build model, fit and transform data. We can also build pipelines using Spark MLlibin particular, its DataFrame-based API package, spark.ml. We use validation set to evaluate the power of one model and do ajustment to the hyperparameters. Finally we get a good result in most problems.