Report for Project 2 of Using Spark

Zhan Wang
Department of Data Science
Fudan University
Shanghai, China 200433
17307110421@fudan.edu.cn

Abstract

This project completes 6 analysis tasks (N6-N7, H1-H4) based on the part (H1-H4) or whole (N6-N7) leaked mernis database. For each task, we introduce the detailed solutions and implement the corresponding codes. The analysis results are presented by directly printing or listing. The whole project is done on Spark platform in RDD and DataFrame format.

1 Environment & Code

This project is done on Ubuntu 16.04, JDK 18.0, python 3.6.2 and Spark 3.1.2. The detailed codes can be found in the file `spark2.py`.

2 Load & Clean Data

The original dataset contains the information for about 48 million Turkish citizens with other introductory text. Therefore, after loading the dataset as text format, we cleaned the data through filtering necessary information and storing the complete instances (core codes shown below). The final cleaned dataset contains 49,611,216 records.

```
# load and clean the data
data = sc.textFile(data_path)
data = data.map(lambda x: x.split('\t'))
data = data.filter(lambda x: len(x) == 17)
data = data.filter(lambda x: int(2009 -\
        int((x[8].split('/'))[2])) <= 150)
print("The Number of Data: ", data.count())
```

3 Tasks N

3.1 N6: Population Density of Top 10 Largest Cities

First, we search for the top 10 largest cities and store them with their population (which has been implemented in the before project). Then, for each city, compute the population density (number of people per square kilometer) using the corresponding area of the city by google search.

```
# top 10 largest cities
city = data.map(lambda x: (x[9], 1)).reduceByKey(add).collect()
city_top10 = [(v[0], v[1]) for v in sorted(city, \
                key=lambda x:x[1], reverse=True)[:10]]
```

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

```
# area of cities
city_10_area = {'ISTANBUL':5343, 'KONYA':38873, 'IZMIR':11891, \
                'ANKARA':24521, 'BURSA':1036,'SIVAS':2768, \
                'SAMSUN':1055, 'AYDIN':1582, 'ADANA':1945, \
                'SANLIURFA':18584}

# population density
city_top10_pop_km2 = []
for i in city_top10:
    city_top10_pop_km2.append((i[0], i[1]/city_10_area[i[0]]))
```

The population density of the top 10 largest cities in Turkey is:

City	pop./km2	City	pop./km2
ISTANBUL	355.7196	SIVAS	437.0408
KONYA	40.0748	SAMSUN	1132.9156
IZMIR	123.8798	AYDIN	750.2630
ANKARA	52.4989	ADANA	569.8468
BURSA	1207.5811	SANLIURFA	57.3770

3.2 N7: Migrant Population

In this task, we need to compare the inter-city migrant population and the inter-district migrant population of Turkey with the Turkish total population. proportion. We firstly select the information about the city and district of ID registration and address in the dataset. Then filter the data and find how many people choose to live in the place different with the registered address at city and district level.

```
regist_addr = data.map(lambda x: (x[9], x[10], x[11], x[12]))

all_pop = data.count()
migrant_city = regist_addr.filter(lambda x: x[0]!=x[2]).count()
migrant_district = regist_addr.filter(lambda x: x[1]!=x[3]).count()

print("inter-administrative migrant population / total population: \
        %.4f"%(migrant_city/all_pop))
print("inter-ctiy migrant population / total population: \
        %.4f"%(migrant_district/all_pop))
```

The output of above codes is

```
inter-administrative migrant population / total population: 0.3614
inter-ctiy migrant population / total population: 0.5239
```

The most common letter in the Turkey citizens' names is 'A' with frequency 82319942. To compare visibly, we also make a plot about all the appeared letters and their frequency in names as below.

4 Tasks H

4.1 Prerequisites

The clean dataset contains about nearly 50 million records. It is too large to run all of the data since it takes uncountable time even using all threads. Therefore, we randomly select 0.1% records (about 49,351) to implement H1-H4 tasks.

```
data_1, data_99 = data.randomSplit([0.001,0.999], 2021)
```

Also, to make our codes reproducible, we set all the random seeds used in the exercise as 2021.

4.2 H1: City Prediction Model

In this task, we need to implement a multi-label classification model. Considering the data characteristics and the running time, we trained a Naive Bayes model using the address district to predict the address city. Before train the model, we need to convert the data type to string then encoder the used features as one hot. We compute the accuracy of the whole valid set and top1-top5 accuracy of test set.

```
# use data_1 as rawdata
data = rawdata.map(lambda x: Row(address_city=x[11],\
        address_district=x[12]))
data_DF = sqlContext.createDataFrame(data)

indexer = StringIndexer(inputCol="address_city", outputCol="label")
indexed = indexer.fit(data_DF).transform(data_DF)
indexer = StringIndexer(inputCol="address_district", \
        outputCol="address_district_index")
indexed = indexer.fit(indexed).transform(indexed)
encoder = OneHotEncoder(inputCol = "address_district_index",\
        outputCol="address_district_one")
encoded = encoder.transform(indexed)
assembler = VectorAssembler(inputCols=["address_district_one"],\
        outputCol="features")
data_h1 = assembler.transform(encoded)

train, valid, test = data_h1.randomSplit([0.7, 0.1, 0.2], 2021)
nb = NaiveBayes(smoothing=1.0, modelType="multinomial")
model = nb.fit(train)

valid_predictions = model.transform(valid)
evaluator = MulticlassClassificationEvaluator(labelCol="label", \
                predictionCol="prediction", metricName="accuracy")
valid_accuracy = evaluator.evaluate(valid_predictions)
print("Valid set accuracy = %.4f" % (valid_accuracy))

test_predictions = model.transform(test)
data_probs = test_predictions.select("probability","label").rdd
for i in range(1,6):
    print("Top %s accuracy = %.4f" % (i, topK_acc(data_probs, i)))
```

To compute the top K accuracy, we use function `topK_acc`.

```
def topK(probs,k):
    count = 0
    record = {}
    for i in probs:
        record[count] = i
        count = count + 1
    record = sorted(record.items(),key=lambda x:x[1],reverse=True)
    result = [record[i][0] for i in range(k)]
    return result

def topK_acc(data, k):
    topk_data = data.map(lambda x:(topK(x[0],k),x[1]))
    res = topk_data.filter(lambda x:int(x[1]) in x[0]).count() \
                / float(topk_data.count())
    return res
```

The output of above codes can be listed in the below table. The prediction results show our trained model performs very well.

Set	Top	Accuracy
Valid	1	0.9780
Test	1	0.9785
	2	0.9872
	3	0.9881
	4	0.9905
	5	0.9914

4.3 H2: Gender Prediction Model

Since the first name of a person if strongly correlated to its gender, thus we use the first name to build a Naive Bayes classification to predict gender.

```
# use data_1 as rawdata
data = rawdata.map(lambda x: Row(first_name=x[2],gender=x[6]))
data_DF = sqlContext.createDataFrame(data)

indexer = StringIndexer(inputCol="gender", outputCol="label")
indexed = indexer.fit(data_DF).transform(data_DF)
indexer = StringIndexer(inputCol="first_name", \
        outputCol="first_name_index")
indexed = indexer.fit(indexed).transform(indexed)
encoder = OneHotEncoder(inputCol = "first_name_index",\
        outputCol="first_name_one")
encoded = encoder.transform(indexed)
assembler = VectorAssembler(inputCols=["first_name_one"],\
        outputCol="features")
data_h2 = assembler.transform(encoded)

train, valid, test = data_h2.randomSplit([0.7, 0.1, 0.2], 2021)
nb = NaiveBayes(smoothing=1.0, modelType="multinomial")
model = nb.fit(train)

valid_predictions = model.transform(valid)
evaluator = MulticlassClassificationEvaluator(labelCol="label", \
        predictionCol="prediction", metricName="accuracy")
valid_accuracy = evaluator.evaluate(valid_predictions)
print("Valid set accuracy = %.4f" % (valid_accuracy))

test_predictions = model.transform(test)
test_accuracy = evaluator.evaluate(test_predictions)
print("Test set accuracy = %.4f" % (test_accuracy))
```

The accuracy of the valid set and test set is:

	Valid Set	Test Set
Accurary	0.9466	0.9464

From above results, valid accuracy is about 0.9466 and test accuracy is nearly is 0.9464 which proves our assumption is appropriate.

4.4 H3: Last Name Prediction Model

In the before project, we have learned that last name and city are correlated, so we tried to use the information of registration city to train a Naive Bayes classification model to predict the last name. Since codes is similar to those in task H1, we only present something different.

```
# use data_1 as rawdata
```

```
data = rawdata.map(lambda x: Row(last_name=x[3], birth_city=x[7], city=x[9]))
data_DF = sqlContext.createDataFrame(data)

indexer = StringIndexer(inputCol="last_name", outputCol="label")
indexed = indexer.fit(data_DF).transform(data_DF)
indexer = StringIndexer(inputCol="city", outputCol="city_index")
indexed = indexer.fit(indexed).transform(indexed)
encoder = OneHotEncoder(inputCol = "city_index",outputCol="city_one")
encoded = encoder.transform(indexed)
assembler = VectorAssembler(inputCols=["city_one"],outputCol="features")
data_h3 = assembler.transform(encoded)
```

The accuracy of the whole valid set and top1-top5 accuracy of test set are shown in the below table.

Set	Top	Accuracy
Valid	1	0.0171
	1	0.0138
	2	0.0257
Test	3	0.0341
	4	0.0428
	5	0.0518

From the results, we can find the accuracy of the prediction is not high on several data sets and the trained model performs not very well. Although the city has influence on last name, we should take some other dependent variables into accounts, such as parents' last name, religious belief and language used. Besides, there are so many categories that we need to try a more adequate model.

4.5 H4: Population Prediction Model

In this task, we first calculate the newly born population for every year. Then spilt data into train set, valid set and test set by 7:3:1. Use the train set to build a linear regression model and test the model on the test set.

```
# use data_1 as rawdata
birth_pop = rawdata.map(lambda x: (int((x[8].split("/"))[2]), 1))\
                .reduceByKey(add).collect()

birth_pop_rdd = sc.parallelize([Row(features=Vectors.dense(i[0]), \
                label=i[1]) for i in birth_pop])
birth_pop_df = sqlContext.createDataFrame(birth_pop_rdd)

train, valid, test = birth_pop_df.randomSplit([0.7, 0.1, 0.2], 2021)
lr = LinearRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
model = lr.fit(train)
predictions = model.transform(test)

print(predictions.show())
```

The output of above codes is

Year	Label	Prediction	Year	Label	Prediction
1910	4	-114	1952	560	625
1919	29	44	1953	626	643
1922	39	97	1960	960	766
1932	265	273	1971	1056	960
1937	303	361	1980	1358	1119
1939	316	396	1984	1259	1189
1940	362	414	1988	1050	1260

From the results, we can find the linear model's prediction is closely to the truly newly born population except for very few deviations.

References

[1] https://spark.apache.org/docs/3.1.2/