s299165_lab4

January 22, 2024

Laboratory 4 - Machine Learning with SparkMLlib

1 Input Data

[1]:
```
#Address for input & output
inputPath_data = "/data/students/bigdata_internet/lab4/log_tcp_complete_classes.
 ↪txt"
outputaddress ="res_out_Lab4/"
```

[2]:
```
from pyspark.sql import SparkSession
# Read the Register File
spark = SparkSession.builder.getOrCreate()
dfProfiles_data = spark.read.load(inputPath_data, format="csv" ,sep = " ",␣
 ↪header = True, inferSchema=True)
```

- Question 1.1. How many columns/features does the file have?
- Question 2.1. How many TCP connections are there in the log?

Answer: The code below shows the number of columns and TCP connection. For TCP connections, the column 31 [‘durat:31’] is considered as established and completed TCP connection. This number is also equal to all rows of available data, which is equal to 100000; and there are 7 diiferent types of TCP connections

[3]:
```
num_columns = len(dfProfiles_data.columns)
print("Number of columns:", num_columns)
```

```
Number of columns: 207
```

2 Classify TCP connections

[4]:
```
tcp_connections = dfProfiles_data[dfProfiles_data["durat:31"] != 0].distinct()
print("Number of TCP connections:", tcp_connections.count())
```

```
24/01/22 17:01:18 WARN util.Utils: Truncated the string representation of a plan
since it was too large. This behavior can be adjusted by setting
```

```
'spark.debug.maxToStringFields' in SparkEnv.conf.
[Stage 3:=================================================>  (190 + 2) / 200]

Number of TCP connections: 100000
```

[5]:
```
different_tcp_connections = dfProfiles_data.select(dfProfiles_data["con_t:42"]).
  ↪distinct()
print("Number of different TCP type connections:", different_tcp_connections.
  ↪count())
```

```
[Stage 6:=========================================================>(199 + 1) / 200]

Number of different TCP type connections: 7
```

- Question 2.0.1 How many classes are there in the file?
- Question 2.0.2 Can you list all of them?
- Question 2.0.3 How many connections per web service are present in the DataFrame?

Answer:

[6]:
```
# Count the number of unique classes
num_classes = dfProfiles_data.select("class:207").distinct().count()
print("number of unique classes is: ", num_classes)
```

```
[Stage 9:=========================================================>(199 + 1) / 200]

number of unique classes is:  10
```

[7]:
```
# List all the unique classes
unique_classes = [row['class:207'] for row in dfProfiles_data.select("class:
  ↪207").distinct().collect()]
print(unique_classes)
```

```
[Stage 11:=======================================>                 (7 + 3) / 10]

['class:google', 'class:amazon', 'class:instagram', 'class:facebook',
'class:netflix', 'class:ebay', 'class:spotify', 'class:youtube',
'class:linkedin', 'class:bing']
```

[8]:
```
# Count the number of connections per web service
connections_per_service = dfProfiles_data.groupBy("class:207").count().
  ↪orderBy("count", ascending=False)
connections_per_service.show()
```

```
[Stage 13:=================================================>     (9 + 1) / 10]
```

```
+---------------+-----+
|      class:207|count|
+---------------+-----+
|class:instagram|10000|
|  class:netflix|10000|
| class:facebook|10000|
|   class:amazon|10000|
|   class:google|10000|
|  class:spotify|10000|
|     class:ebay|10000|
| class:linkedin|10000|
|  class:youtube|10000|
|     class:bing|10000|
+---------------+-----+
```

2.1 Select features

- Question 2.1.1. Does it make sense to use the IP addresses + ports (`#31#c_ip:1`, `c_port:2`,`s_ip:15`, `s_port:16`) as features?
- Question 2.1.2. Would it be fair to use the Fully Qualified Domain Name (FQDN, `fqdn:127`, forinstance `www.google.com`) for the classification?

Answer 2.1.1) NO, using IP addresses and ports (`#31#c_ip:1`, `c_port:2`, `s_ip:15`, `s_port:16`) as features might not be the most suitable approach asn they may not provide meaningful information directly related to the classification task; moreover, it does not mean anything on the client side. Answer 2.1.2) NO, it wouldn't be fair, as web services deliver their content through various servers with different IP addresses, all of which are identified by their respective domains.

2.2 Read and split the data

[9]:
```
# Select data, which can make sense for the purpose of this exercise
selected_df = dfProfiles_data.select("c_pkts_all:3","c_bytes_uniq:7",␣
 ↪"c_pkts_data:8",  "s_pkts_all:17","s_bytes_uniq:21", "s_pkts_data:22",\
                        "con_t:42", "p2p_t:43", "http_t:44", "p2p_st:59",␣
 ↪"http_req_cnt:111","http_res_cnt:112","fqdn:127", "class:207")
# The data is splitted in the Train and Test Part!!
```

2.3 Pre-process the dataset

[10]:
```
import pyspark.ml
from pyspark.ml.feature import VectorAssembler, StandardScaler, StringIndexer,␣
 ↪OneHotEncoderEstimator
```

[11]:
```
# Map and convert categorical string columns into numerical indices
```

```
WebService = StringIndexer(inputCol="class:207",outputCol="WebService").
 ↪fit(selected_df)
fqdn = StringIndexer(inputCol="fqdn:127",outputCol="fqdn").fit(selected_df)
ConnectionType = StringIndexer(inputCol="con_t:42",outputCol="connection_type").
 ↪fit(selected_df)
p2pType= StringIndexer(inputCol="p2p_t:43",outputCol="p2p_type").
 ↪fit(selected_df)
HttpType = StringIndexer(inputCol="http_t:44",outputCol="http_type").
 ↪fit(selected_df)
p2p_subType = StringIndexer(inputCol="p2p_st:59",outputCol="p2p_subtype").
 ↪fit(selected_df)
```

[12]:
```
#Transform and creae new columns with numerical representations for categorical␣
 ↪features, using transform() function.
transform_selected_df = WebService.transform(selected_df)
transform_selected_df = fqdn.transform(transform_selected_df)
transform_selected_df = ConnectionType.transform(transform_selected_df)
transform_selected_df = p2pType.transform(transform_selected_df)
transform_selected_df = HttpType.transform(transform_selected_df)
transform_selected_df = p2p_subType.transform(transform_selected_df)
```

[13]:
```
# Select the required columns for next steps
new_selected_df = transform_selected_df.select("c_pkts_all:3","c_bytes_uniq:7",␣
 ↪"c_pkts_data:8",  "s_pkts_all:17",\
"s_bytes_uniq:21", "s_pkts_data:22", "http_req_cnt:111","http_res_cnt:112",␣
 ↪"connection_type", "p2p_type", "http_type", "p2p_subtype", "WebService")
```

[14]:
```
#Create the list of features to use in the next step
using_features = ["c_pkts_all:3","c_bytes_uniq:7", "c_pkts_data:8", ␣
 ↪"s_pkts_all:17","s_bytes_uniq:21", "s_pkts_data:22", "http_req_cnt:111",\
                          "http_res_cnt:112", "connection_type", "p2p_type",␣
 ↪"http_type", "p2p_subtype", "WebService"]
```

[15]:
```
# Set the columns into 'features' using vector assembler
vec_assembler = VectorAssembler(inputCols=using_features, outputCol="features")
vec_df = vec_assembler.transform(new_selected_df)
```

2.4 Train at least two different models

- Question2.4.1 How much does it take to train the model (time in seconds), for the different algorithm and parameters?

Answer: The codes below are Regression and Random Forestf train model. The duration of operations are calculated via 'time()'; results are printed or written as comments.

[16]:
```
from pyspark.ml.classification import LogisticRegression,␣
↪DecisionTreeClassifier, RandomForestClassifier
import time
```

[17]:
```
# Split data 70%-30% ; (keep seed = 10 times)
train_df,test_df = vec_df.randomSplit([0.7,0.3],10)
```

[18]:
```
# Regression Model, using train and test data
start = time.time()
LRegression = LogisticRegression(featuresCol = 'features' , labelCol=␣
↪'WebService')
Train_LRegression = LRegression.fit(train_df)
Test_Regression = Train_LRegression.transform(test_df)
stop = time.time()
duration_Reg = stop - start
print(f'Time for Regression model: {duration_Reg} seconds')
```

```
24/01/22 17:02:02 WARN netlib.BLAS: Failed to load implementation from:
com.github.fommil.netlib.NativeSystemBLAS
24/01/22 17:02:02 WARN netlib.BLAS: Failed to load implementation from:
com.github.fommil.netlib.NativeRefBLAS
```

```
Time for Regression model: 27.923652172088623 seconds
```

[19]:
```
# Random Forest Model, using train and test data
start = time.time()
RForest = RandomForestClassifier(labelCol='WebService', featuresCol='features',␣
↪numTrees=10)
Train_RForest = RForest.fit(train_df)
Test_RForest = Train_RForest.transform(test_df)
stop = time.time()
duration_RF = stop - start
print(f'Time for Random Forest model {duration_RF} seconds')
```

```
Time for Random Forest model 11.555283069610596 seconds
```

2.5 Evaluate the performance of the models

- Question 2.5.1 Comment your results: which classes are easier to classify? Which get confused themost?
- Question 2.5.2 Which classifier performs better? Why do you think it is the case?

Answer 2.5.1) Results show the Regression model is very accurate and consistent in predicting the 'WebService'; for the confusion, the confusion matrix is provided after the codes below. Answer 2.5.2) The evaluation metrics below show that the regression model performs better than the

random forest model in terms of Precision, Recall, and F1 Score, with values closer to 1, making it the recommended choice

[20]:
```
# Evaluating the accuracy of the models
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(labelCol='WebService',␣
 ↪metricName='accuracy')
# Models accuracy
print ("The accuracy of Reg is: ", evaluator.evaluate(Test_Regression))
print ("The accuracy of RF is: ", evaluator.evaluate(Test_RForest))
```

```
The accuracy of Reg is:  0.998255033557047
[Stage 292:=============================================>           (8 + 2) / 10]
The accuracy of RF is:  0.6979530201342282
```

[21]:
```
# Evaluating Precision, recall & F1-score for models
evaluator = MulticlassClassificationEvaluator(labelCol='WebService',␣
 ↪predictionCol='prediction', metricName='weightedPrecision')
# Models' precision
precision_Regression = evaluator.evaluate(Test_Regression)
precision_RandomForest = evaluator.evaluate(Test_RForest)
# Models' recall
evaluator.setMetricName('weightedRecall')
recall_Regression = evaluator.evaluate(Test_Regression)
recall_RandomForest = evaluator.evaluate(Test_RForest)
# Models' F1 score
evaluator.setMetricName('f1')
f1_score_Regression = evaluator.evaluate(Test_Regression)
f1_score_RandomForest = evaluator.evaluate(Test_RForest)
```

[22]:
```
print('For Regression Model: \n Precision is: {}\n Recall is: {}\n F1 Score is:\
{}'.format(precision_Regression, recall_Regression, f1_score_Regression))
```

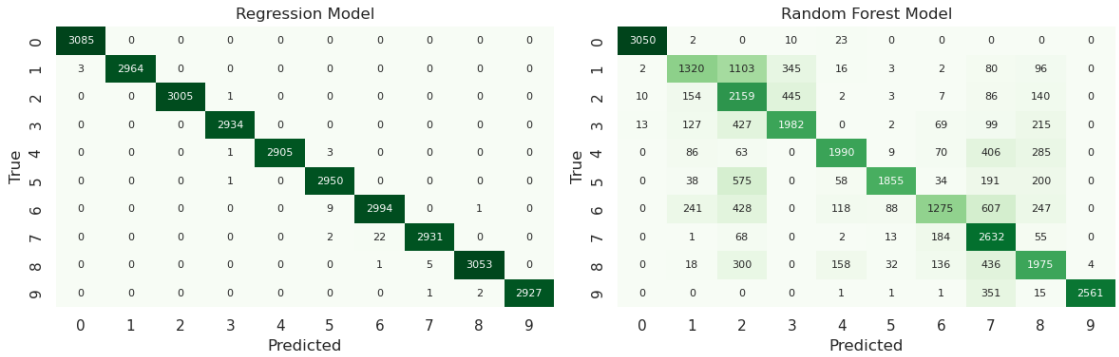
```
For Regression Model:
 Precision is: 0.9982592875904852
 Recall is: 0.998255033557047
 F1 Score is:0.9982551203204156
```

[23]:
```
print('For Random Forest Model: \n Precision is: {}\n Recall is: {}\n F1 Score \
is: {}'.format(precision_RandomForest, recall_RandomForest,␣
 ↪f1_score_RandomForest))
```

```
For Random Forest Model:
 Precision is: 0.7417423268371506
 Recall is: 0.6979530201342282
 F1 Score is: 0.7016237802281289
```

[24]:
```
# Create confusion matrix to check which class get more confuesd in the models
from pyspark.mllib.evaluation import MulticlassMetrics
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
# Convert DataFrame predictions to RDD to create the confusion matrix
predict_label_Reg = Test_Regression.select("prediction", "WebService").rdd.
 ↪map(lambda row: (row.prediction, float(row.WebService)))
predict_label_RF = Test_RForest.select("prediction", "WebService").rdd.
 ↪map(lambda row: (row.prediction, float(row.WebService)))
# Instantiate the MulticlassMetrics class
metrics_Reg = MulticlassMetrics(predict_label_Reg)
metrics_RF = MulticlassMetrics(predict_label_RF)
# Models confusion Matrix
conf_matrix_Reg = metrics_Reg.confusionMatrix()
conf_matrix_RF = metrics_RF.confusionMatrix()
```

[25]:
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
def plot_confusion_matrix(conf_matrix, title, ax):
    sns.heatmap(conf_matrix.toArray(), annot=True, fmt='g', cmap='Greens',␣
 ↪cbar=False,
                annot_kws={"size": 8}, ax=ax)  # Adjust the font size here
    ax.set_title(title)
    ax.set_xlabel('Predicted')
    ax.set_ylabel('True')
# For less space create a figure with two subplots (one row, two columns)
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
plot_confusion_matrix(conf_matrix_Reg, 'Regression Model', axes[0])
plot_confusion_matrix(conf_matrix_RF, 'Random Forest Model', axes[1])
plt.tight_layout()  # Plot the matrix with layout
plt.show()
```



According to matrix, classes (0, 1, 2, 3, 4, 8, 9 for Reg; 0, 9 for RF) are easily classified (high diagonal values), while classes 5, 6, and 7 show misclassifications, particularly class 6 challenging for both models, and classes 1, 3, 4 and 5 posing difficulties for the RF.

2.6 Tune the hyper-parameters of the models

- Question 2.6: Report the accuracy results for all the parameters you tried. What can youconclude?

Answer:

[26]:
```
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
import numpy as np
```

[27]:
```
# Tuning hyper-parameters of the Regression model
NewReg = LogisticRegression(labelCol='WebService', featuresCol='features')
paramGrid = ParamGridBuilder().addGrid(NewReg.maxIter, [10, 20])\
.addGrid(NewReg.regParam, [0.01, 0.1]).addGrid(NewReg.elasticNetParam, [0.0, 0.
↪5, 1.0]).build()
```

[28]:
```
# Create a cross-validator
NewRegEvaluator = MulticlassClassificationEvaluator(labelCol='WebService',␣
↪predictionCol='prediction', metricName='accuracy')
cv = CrossValidator(estimator=NewReg, evaluator=NewRegEvaluator,␣
↪estimatorParamMaps=paramGrid, numFolds=3)
# Fit the cross-validator to the training data
cvModelReg = cv.fit(train_df)
```

[29]:
```
# Transform the test data using the best model
cvRegResult = cvModelReg.transform(test_df)
```

[40]:
```
print('Accuracy of Reg. changed to :',evaluator.evaluate(cvRegResult))
```

```
[Stage 2635:============================>                    (1 + 1) / 2]
```

```
Accuracy of Reg. changed to : 0.5896113385870007
```

[31]:
```
# Best set of hyperparameters based on the average evaluation metric
best_params_reg =cvModelReg.getEstimatorParamMaps()[np.argmax(cvModelReg.
 ↪avgMetrics)]
# Print the best hyperparameters to show the values
#print("Best hyperparameters for Reg: {}".format(best_params_reg))
```

[32]:
```
# Tuning hyper-parameters of the Random Forest Model
newRF = RandomForestClassifier(labelCol='WebService', featuresCol='features',␣
 ↪numTrees=20)
paramGrid = ParamGridBuilder().addGrid(newRF.numTrees, [10,20]).addGrid\
(newRF.maxDepth, [5,10]).addGrid(newRF.impurity, ["Gini","Entropy"]).build()
```

[33]:
```
# folds: number of subsets using crossvalidation
NewRFEvaluator =␣
 ↪MulticlassClassificationEvaluator(labelCol='WebService',predictionCol='prediction',metricNa
cv = CrossValidator(estimator=newRF, evaluator=NewRFEvaluator,␣
 ↪estimatorParamMaps=paramGrid, numFolds=3)
cvModelRf=cv.fit(train_df)
```

[34]:
```
cvRfResult = cvModelRf.transform(test_df)
```

[41]:
```
print('Accuracy of RF changed to:',evaluator.evaluate(cvRfResult))
```

```
[Stage 2641:============================>                            (1 + 1) / 2]
Accuracy of RF changed to: 0.9990254426169358
```

[36]:
```
# Best set of hyperparameters based on the average evaluation metric
best_params_rf = cvModelRf.getEstimatorParamMaps()[np.argmax(cvModelRf.
 ↪avgMetrics)]
# Print the best hyperparameters to show the values
#print("Best hyperparameters for RF: {}".format(best_params_rf))
```

Conclusion : Overall, changing hyperparameters might not improve the outcome results! according to results in this exercise,, it can be concluded that the RF model seems to be more sensitive to hyperparameter tuning and outperforms the Reg model in this specific exercise. The decrease in Regression Model accuracy might indicate that the hyperparameter tuning did not improve the model's performance, or the selected hyperparameters may not be suitable for the given dataset. It should be considered, new RF model is near overfit!!

2.7 Return the best possible model and estimate its performance on new data

- Question 2.7.1: Report the expected results performance and comment on the results obtained.

Answer: The best possible model is shown in the previous part, which is new Random Forest model. The best set of hyperparameters based on the average evaluation metric are also extracted above. The codes below are provided to answer this section.

[37]:
```
# Evaluate performance with best hyperparameters
train_RF_result = cvModelRf.transform(train_df)
test_RF_result = cvModelRf.transform(test_df)
```

[38]:
```
# Print the performance
best_evaluator = MulticlassClassificationEvaluator(labelCol='WebService',␣
 ↪predictionCol='prediction', metricName='accuracy')
# Best Model accuracy
training_acc = evaluator.evaluate(train_RF_result)
test_acc = evaluator.evaluate(test_RF_result)
# Best Model precision, recall, and F1-score)
evaluator.setMetricName('weightedPrecision')
precision_RF = evaluator.evaluate(test_RF_result)
evaluator.setMetricName('weightedRecall')
recall_RF = evaluator.evaluate(test_RF_result)
evaluator.setMetricName('f1')
f1_RF = evaluator.evaluate(test_RF_result)
```

[39]:
```
print('For Best Model:  \n Training Accuracy is: {}\n Test Accuracy is:\
{}\n Precision is: {}\n Recall is: {}\n F1 Score \
is: {}'.format(training_acc, test_acc, precision_RF, recall_RF, f1_RF))
```

```
For Best Model:
 Training Accuracy is: 0.9991168708948222
 Test Accuracy is:0.9986914434492662
 Precision is: 0.9986956836984415
 Recall is: 0.9986912751677852
 F1 Score is: 0.9986914434492662
```

[ ]: