Machine Learning to measure Internet traffic

February 22, 2024

1 Machine Learning to measure Internet traffic

[ ]:
```
'''
In this analysis, I am going to use different aspects of Machine Learning␣
 ↪applied to a Big Data framework.
I will apply classification and clustering techniques to measure the Internet␣
 ↪traffic.
A network log trace file summarizing the traffic generated by thousands of␣
 ↪users while browsing the web is used.

A Tstat (TCP STatistic and Analysis Tool) log file will be used.
Each line represents a TCP connection. Besides the connection identifiers␣
 ↪(client and server
IP addresses and ports), Tstat reports dozens of features, such as the number␣
 ↪of packets,
bytes uploaded and downloaded, etc.
'''
```

[ ]:
```
# Input Data from the big data cluster
```

[10]:
```
Tstat = "/data/students/bigdata_internet/lab4/log_tcp_complete_classes.txt"
```

[ ]:
```
'''In this analysis, PySpark was utilized for its robust distributed computing␣
 ↪capabilities,
ideal for handling large datasets efficiently.
If you're using the PySpark shell, no additional setup is necessary.
However, for those working in a Python environment, setting up PySpark involves␣
 ↪the following steps:
1. Install PySpark: Begin by installing PySpark using pip:
pip install pyspark
2. Configure PySpark.sql: In your Python script or interactive session, include␣
 ↪the following configuration
to initialize PySpark.sql:
```python
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```
```

```
Ensure to execute this configuration before performing any PySpark operations.
For comprehensive installation and configuration instructions, refer to the␣
↪official PySpark documentation:
PySpark Installation Guide
'''
```

[ ]:
```
# Reading data
```

[11]:
```
spark = SparkSession.builder.getOrCreate()
df = spark.read.load(Tstat, format="csv", header=True, inferSchema=True, sep='␣
↪')
```

[ ]:
```
# See the columns of our dataframe
```

[1]:
```
df.columns
```

[ ]:
```
# Count the number of columns. There are 207 columns
```

[14]:
```
len(df.columns)
```

[14]: 207

[ ]:
```
# Count the number of rows. There are 100,000 rows which means 100,000 TCP␣
↪connections
```

[15]:
```
df.count()
```

[15]: 100000

[ ]:
```
# The 207th column "class:207" there are the label of classes
```

[17]:
```
class_207 = df.select("class:207")
```

[ ]:
```
# There are 10 classes in this TCP connection dataframe
```

[18]:
```
class_207.distinct().count()
```

[18]: 10

[ ]:
```
# The list of classes can be seen here (such as google, amazon, etc.)
```

[19]:
```
class_207.distinct().show()
```

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

[ ]:
```
# I am going to group the dataframe by classes and count the number of TCP␣
↪connections to see how much connection
# we have for each class
```

[20]:
```
web_services = df.groupBy("class:207").count()
```

[21]:
```
web_services.show()
```

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

[ ]:
```
# Classify TCP connections
```

[ ]:
```
# Split dataframe into train-set and test-set (75, 25)
```

[22]:
```
trainValidation, test = df.randomSplit([0.75, 0.25], 42)
```

[ ]:
```
# Pre-processing the dataset
```

[4]:
```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StringIndexer
```

[24]:
```
# Select features
feat_cols = ['c_bytes_uniq:7', 's_bytes_uniq:21', 'c_pkts_data:8', 's_pkts_data:
↪22']
```

[25]:
```
# Preprocess TrainValidation set
# Vector Assembler
va = VectorAssembler(inputCols=feat_cols, outputCol='features')
vDF = va.transform(trainValidation)
# Convert string to index for target column
indexer = StringIndexer(inputCol="class:207", outputCol="label")
indexerModel = indexer.fit(vDF)
indexedDF = indexerModel.transform(vDF)
```

[26]:
```
# Preprocess Test set
testVDF = va.transform(test)
testIndexedDF = indexerModel.transform(testVDF)
```

[ ]:
```
# For classification, I am going to use Decision Tree model and Random Forest␣
↪model
# In order to compare how much time do these models take to train the I will␣
↪import time to compare them
```

[27]:
```
import time
```

[28]:
```
# 1. DECISION TREE CLASSIFIER
from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier(labelCol="label", featuresCol="features",␣
↪maxDepth=20)
start_time = time.time()
dtModel = dt.fit(indexedDF)
stop_time = time.time()
print(f'It takes {stop_time - start_time} seconds')
finalDFdt = dtModel.transform(indexedDF)
```

```
It takes 42.52218413352966 seconds
```

[29]:
```
testDFdt = dtModel.transform(testIndexedDF)
```

[30]:
```
# 2. RANDOM FOREST CLASSIFIER
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(labelCol="label", featuresCol="features",␣
 ↪numTrees=20, maxDepth=20)
start_time = time.time()
rfModel = rf.fit(indexedDF)
stop_time = time.time()
print(f'It takes {stop_time - start_time} seconds')
finalDFrf = rfModel.transform(indexedDF)
```

```
[Stage 124:>                                                        (0 + 1) / 1]

It takes 61.193100452423096 seconds
```

[31]:
```
testDFrf = rfModel.transform(testIndexedDF)
```

[ ]:
```
# Evaluate the performance of the models
```

[ ]:
```
'''The result shows that Random Forest classifies data a little bit easier,␣
 ↪because the precision of
its classes are more diverse (0.50, 0.84) and this range is smaller for␣
 ↪Decision Tree (0.50, 0.79).
For train set, Decision Tree classifier's accuracy is a little bit more than␣
 ↪Random Forest Classifier
but in the test set, the accuracy of Random Forest is a little more than␣
 ↪Decision Tree.'''
```

[ ]:
```
# Evaluate the Decision Tree model performance
```

[5]:
```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

[33]:
```
accuracy = MulticlassClassificationEvaluator(labelCol="label",␣
 ↪predictionCol="prediction", metricName="accuracy")
```

[34]:
```
# Global accuracy on the TrainValidation set
```

[35]:
```
print(f'Decition Tree Train Model:\t accuracy = {accuracy.
 ↪evaluate(finalDFdt)}\nRandom Forest Train Model:\t accuracy = {accuracy.
 ↪evaluate(finalDFrf)}')
```

```
[Stage 131:============================>                            (1 + 1) / 2]
```

```
Decition Tree Train Model:       accuracy = 0.7894673736377927
Random Forest Train Model:       accuracy = 0.7818869734352935
```

[36]:
```
# Global accuracy on the Test set
```

[38]:
```
print(f'Decition Tree Train Model:\t accuracy = {accuracy.
 ↪evaluate(testDFdt)}\nRandom Forest Train Model:\t accuracy = {accuracy.
 ↪evaluate(testDFrf)}')
```

```
[Stage 147:============================>                              (1 + 1) / 2]

Decition Tree Train Model:       accuracy = 0.6938006255513673
Random Forest Train Model:       accuracy = 0.7055898628598926
```

[39]:
```
dtTestRDD = testDFdt.select("prediction", "label").rdd.map(lambda x:␣
 ↪(float(x[0]), float(x[1])))
```

[40]:
```
rfTestRDD = testDFrf.select("prediction", "label").rdd.map(lambda x:␣
 ↪(float(x[0]), float(x[1])))
```

[6]:
```
from pyspark.mllib.evaluation import MulticlassMetrics
```

[50]:
```
def metrics_calculator(x):
    metrics = MulticlassMetrics(x)
    precision = metrics.precision()
    recall = metrics.recall()
    f1Score = metrics.fMeasure
    labels = x.map(lambda a: a[1]).distinct().collect()
    print("Class \t Precision \t\t Recall \t\t F1Score")
    for label in sorted(labels):
        print(f'{label},\t {metrics.precision(label)},\t {metrics.
 ↪recall(label)},\t {metrics.fMeasure(label, beta=0.1)}')
```

[49]:
```
dtTestMetrics = metrics_calculator(dtTestRDD)
```

```
Class    Precision                Recall                  F1Score

[Stage 163:============================>                              (1 + 1) / 2]

0.0,     0.6794921123509042,      0.7376775271512114,     0.680023180095618
1.0,     0.6761115954664342,      0.6372226787181594,     0.6757033049509563
2.0,     0.7424892703862661,      0.6316430020283975,     0.7412014234204511
3.0,     0.670468948035488,       0.6404358353510896,     0.6701577904322303
4.0,     0.8008057296329454,      0.7150279776179057,     0.7998556896353286
5.0,     0.7561613144137416,      0.8074162679425837,     0.7566368734924604
```

```
6.0,     0.718683197947841,      0.6702551834130781,     0.7181694358904944
7.0,     0.8450230995380092,     0.7980959936533122,     0.844531441561626
8.0,     0.5014416146083613,     0.8155529503712388,     0.5033611217908153
9.0,     0.697817571348629,      0.48540288049824837,    0.6948071650420092
```

[52]:
```
rfTestMetrics = metrics_calculator(rfTestRDD)
```

```
Class    Precision               Recall                  F1Score
[Stage 171:============================>                              (1 + 1) / 2]
0.0,     0.7344537815126051,     0.7301587301587301,     0.7344110085942245
1.0,     0.7387127761767531,     0.6318816762530813,     0.7374782798598517
2.0,     0.785254824344384,      0.6438133874239351,     0.7835504607337521
3.0,     0.6723577235772358,     0.6674737691686844,     0.6723090172973061
4.0,     0.8050847457627118,     0.7214228617106315,     0.8041614101331264
5.0,     0.7290552584670231,     0.8153907496012759,     0.7298203584350973
6.0,     0.7265917602996255,     0.6961722488038278,     0.72627755263418
7.0,     0.862910381543922,      0.7715192383974613,     0.8618995178153834
8.0,     0.5024043966109457,     0.8573661586557249,     0.5044723044946148
9.0,     0.6933471933471933,     0.5192681977423121,     0.6910534495227446
```

[ ]:
```
# Hyper-parameters tuning
```

[ ]:
```
# I am going to use CrossValidation and ParamGridBuilder for tuning
```

[7]:
```
from pyspark.ml.tuning import CrossValidator
from pyspark.ml.tuning import ParamGridBuilder
```

[54]:
```
# Decision Tree parameter Tuning
dtParamGrid = ParamGridBuilder().addGrid(dt.maxDepth, [15, 20, 25]).addGrid(dt.
 ↪impurity, ["Gini", "Entropy"]).build()
# CrossValidation
dtCv = CrossValidator(estimator=dt, evaluator=accuracy,␣
 ↪estimatorParamMaps=dtParamGrid, numFolds=3)
dtCvModel = dtCv.fit(indexedDF)
dtFinalDF = dtCvModel.transform(indexedDF)
```

[55]:
```
import numpy as np
```

[56]:
```python
# Analyze the best parameter of Decision Tree Classifier
dtCvModel.getEstimatorParamMaps()[np.argmax(dtCvModel.avgMetrics)]
```

[56]:
```
{Param(parent='DecisionTreeClassifier_83c6b6ced5c4', name='maxDepth',
doc='Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf node; depth 1
means 1 internal node + 2 leaf nodes.'): 20,
 Param(parent='DecisionTreeClassifier_83c6b6ced5c4', name='impurity',
doc='Criterion used for information gain calculation (case-insensitive).
Supported options: entropy, gini'): 'Gini'}
```

[57]:
```python
# Random Forest parameter Tuning
rfParamGrid = ParamGridBuilder().addGrid(rf.maxDepth, [15, 20, 25]).addGrid(rf.
 ↪impurity, ["Gini", "Entropy"]).addGrid(rf.numTrees, [15, 20, 25]).build()
# CrossValidation
rfCv = CrossValidator(estimator=rf, evaluator=accuracy,␣
 ↪estimatorParamMaps=rfParamGrid, numFolds=3)
rfCvModel = rfCv.fit(indexedDF)
rfFinalDF = rfCvModel.transform(indexedDF)
```

[58]:
```python
# Analyze the best parameter of Decision Tree Classifier
rfCvModel.getEstimatorParamMaps()[np.argmax(rfCvModel.avgMetrics)]
```

[58]:
```
{Param(parent='RandomForestClassifier_e5cd46576df4', name='maxDepth',
doc='Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf node; depth 1
means 1 internal node + 2 leaf nodes.'): 15,
 Param(parent='RandomForestClassifier_e5cd46576df4', name='impurity',
doc='Criterion used for information gain calculation (case-insensitive).
Supported options: entropy, gini'): 'Entropy',
 Param(parent='RandomForestClassifier_e5cd46576df4', name='numTrees',
doc='Number of trees to train (>= 1).'): 25}
```

[59]:
```python
# Calculate best Decision Tree model accuracy
bestDT = DecisionTreeClassifier(labelCol="label", featuresCol="features",␣
 ↪maxDepth=20, impurity="Gini")
bestDTModel = bestDT.fit(indexedDF)
bestDTFinal = bestDTModel.transform(indexedDF)
accuracy.evaluate(bestDTFinal)
```

[59]: 0.7894673736377927

[60]:
```python
# Calculate best Random Forest model accuracy
bestRF = RandomForestClassifier(labelCol="label", featuresCol="features",␣
 ↪maxDepth=20, impurity="Gini", numTrees=20)
```

```
bestRFModel = bestRF.fit(indexedDF)
bestRFFinal = bestRFModel.transform(indexedDF)
accuracy.evaluate(bestRFFinal)
```

[60]: 0.7818869734352935

[66]:
```
# Calculate best Decision Tree model accuracy test set
bestDTFinaltest = bestDTModel.transform(testIndexedDF)
accuracy.evaluate(bestDTFinaltest)
```

[66]: 0.6938006255513673

[67]:
```
# Calculate best Random Forest model accuracy test set
bestRFFinaltest = bestRFModel.transform(testIndexedDF)
accuracy.evaluate(bestRFFinaltest)
```

[67]: 0.7055898628598926

[ ]:
```
# It shows that the best possible model is Random Forest and its accuracy is␣
↪about 70%
```

[ ]:
```
# Clustering users
```

[ ]:
```
# I am going to use k-means model and Gaussian mixture model
```

[ ]:
```
# Calculate how many distinct IPs (users) we have
```

[68]:
```
clients = df.select("#31#c_ip:1").distinct().count()
```

[69]:
```
print(clients)
```

3844

[ ]:
```
# Find the top-5 most active users
```

[13]:
```
connections = df.groupBy("#31#c_ip:1").agg({"#31#c_ip:1": "count", "c_bytes_all:
↪9":"sum", "s_bytes_all:23":"sum", "s_bytes_retx:25":"sum", "s_rtt_avg:52":
↪"avg", "s_first:33":"avg"})
```

[71]:
```
connectionOrdered = connections.sort("count(#31#c_ip:1)", ascending=False).
↪show(5)
```

```
[Stage 3871:============================================>       (171 + 2) / 200]

+--------------+------------------+-----------------+------------------+--------
-----------+-------------------+------------------+
|    #31#c_ip:1| avg(s_rtt_avg:52)|count(#31#c_ip:1)|
avg(s_first:33)|sum(s_bytes_all:23)|sum(s_bytes_retx:25)|sum(c_bytes_all:9)|
+--------------+------------------+-----------------+------------------+--------
-----------+-------------------+------------------+
| 246.25.63.193|126.04335648340408|             1175|57.096006808510644|
3273754|             170919|           2596746|
|246.25.221.106| 42.48682521290321|              620|176.24669516129038|
8000015|              31258|           8934440|
|  180.102.5.86|30.208663604166667|              528|  95.7006685606061|
776618|                560|          1946611|
|  246.25.63.82|103.03908910739855|              419| 292.7211861575179|
62219033|            2091264|          10775700|
|  180.102.5.42| 64.18118633250616|              403|180.22686352357326|
13021759|              90956|           4376418|
+--------------+------------------+-----------------+------------------+--------
-----------+-------------------+------------------+
only showing top 5 rows
```

[ ]:
```
# Calculate the average number of connections
```

[14]:
```
connectionsAvg = connections.agg({"count(#31#c_ip:1)":"avg"})
```

[73]:
```
connectionsAvg.show()
```

```
[Stage 3873:==========================================>         (162 + 3) / 200]

+----------------------+
|avg(count(#31#c_ip:1))|
+----------------------+
|    26.014568158168576|
+----------------------+
```

[2]:
```
# Features selection
feat_cols_cluster = ['count(#31#c_ip:1)', 'sum(s_bytes_all:23)',␣
↪'sum(s_bytes_retx:25)', 'sum(c_bytes_all:9)', 'avg(s_rtt_avg:52)',␣
↪'avg(s_first:33)']
```

[15]:
```python
# Preprocessing
va_cluster = VectorAssembler(inputCols=feat_cols_cluster, outputCol="features")
assembledDF = va_cluster.transform(connections)
# Scaler
from pyspark.ml.feature import StandardScaler
scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures",␣
  ↪withStd=True, withMean=True)
scalerModel = scaler.fit(assembledDF)
scaledDF = scalerModel.transform(assembledDF)
```

[22]:
```python
# Train the K-Means model
from pyspark.ml.clustering import KMeans
kmeans = KMeans(k=10, featuresCol="scaledFeatures")
kmeansModel = kmeans.fit(scaledDF)
kmeansPredictionsDF = kmeansModel.transform(scaledDF)
```

[21]:
```python
# Train the GMM model
from pyspark.ml.clustering import GaussianMixture
gmm = GaussianMixture(k=10, featuresCol="scaledFeatures")
gmmModel = gmm.fit(scaledDF)
gmmPredictionsDF = gmmModel.transform(scaledDF)
```

[18]:
```python
from pyspark.ml.evaluation import ClusteringEvaluator
evaluator = ClusteringEvaluator()
```

[87]:
```python
# Evaluate K-Means performance
silhouetteKMeans = evaluator.evaluate(kmeansPredictionsDF)
print("Silhouette with squared euclidean distance = " + str(silhouette))
print("SSE: ",kmeansModel.computeCost(kmeansPredictionsDF))
```

```
Silhouette with squared euclidean distance = 0.12522028393706328
[Stage 4951:====================================================>(198 + 2) / 200]
SSE:  6406.865780093279
```

[88]:
```python
# Evaluate GMM performance
silhouetteGMM = evaluator.evaluate(gmmPredictionsDF)
print("Silhouette with squared euclidean distance = " + str(silhouette))
```

```
[Stage 4956:==========================================>          (158 + 4) / 200]
Silhouette with squared euclidean distance = 0.12522028393706328
```

[19]:
```python
# Tune K-means Parameters
kmeansBest = KMeans(k=3, featuresCol="scaledFeatures", initSteps=10, maxIter=25)
kmeansModelBest = kmeansBest.fit(scaledDF)
kmeansPredictionsBest = kmeansModelBest.transform(scaledDF)
silhouetteBKM = evaluator.evaluate(kmeansPredictionsBest)
print("Silhouette with squared euclidean distance = " + str(silhouetteBKM))
```

```
[Stage 497:==============================================>        (172 + 4) / 200]
```

```
Silhouette with squared euclidean distance = 0.9916145915641448
```

[20]:
```python
# Tune the GMM parameters
gmmBest = GaussianMixture(k=3, featuresCol="scaledFeatures", maxIter=5)
gmmModelBest = gmmBest.fit(scaledDF)
gmmPredictionsBest = gmmModelBest.transform(scaledDF)
silhouetteBGMM = evaluator.evaluate(gmmPredictionsBest)
print("Silhouette with squared euclidean distance = " + str(silhouetteBGMM))
```

```
[Stage 526:================================>                        (4 + 3) / 7]
```

```
Silhouette with squared euclidean distance = 0.7156195042662563
```