# Test 1 : Only Concatenate the 3 examples Python files : first run

Code : datacamp1.py

C’est juste l’utilisation des 3 exemples fournis (concaténation de l’ensemble dans un seul fichier) : example1.py + 2 +3

Juste prévision sur 1 seul Tag : HTML

Avec une regression linéaire :

from pyspark.ml.classification import LogisticRegression

logistic=LogisticRegression(featuresCol="tf\_idf\_title",labelCol="html",predictionCol='html\_pred',rawPredictionCol="html\_pred\_raw",maxIter=10)

Evaluator :

from pyspark.ml.evaluation import BinaryClassificationEvaluator

evaluator = BinaryClassificationEvaluator(rawPredictionCol="html\_pred\_raw", labelCol='html',metricName="areaUnderPR",)

Exécuté avec 16 Nodes :

spark-submit --master yarn --conf spark.local.dir=/home/christophe.noblanc/tmp --num-executors 16 --driver-memory 2g --conf spark.ui.port=6660 datacamp1.py

---------------------- SUMMARY :

('eval\_train (evaluator) =', 0.7605636522382886)

('eval\_test (evaluator) =', 0.6064325915002078)

('F1 Test Score =', 0.10727987317814343)

('DSSP\_score (valid) =', 0.10743929820378322)

################ : END.

('Total Duration Time (m)= ', 5.0)

('Total Duration Time (s)= ', 49.593294000000014)

# Test 2 : split the “feature & data prep” work with the “ML models“ work

Split the code to have :

1. autonom process to work on data features and pre-process
2. Autonomy process to work on preprocess data to apply ML

1/ Feature extraction and data Prep :

Using Parquet file format , stored in HDFS :

base\_path="/user/christophe.noblanc/datacamp"

fileName\_train=base\_path+"/train\_features\_001.parquet"

fileName\_valid=base\_path+"/valid\_features\_001.parquet"

dataDF.write.format("parquet").mode("overwrite").save("hdfs://"+fileName\_train)

validDF.write.format("parquet").mode("overwrite").save("hdfs://"+fileName\_valid)

About 2 minutes 11 to run this one (using 16 nodes)

---------------------- SUMMARY :

Source Data (Train&Test) Row Count= 1.901.102

Validation Row Count= 335.520

################ : END.

Then, in the ML program :

from pyspark.sql import SQLContext

base\_path="/user/christophe.noblanc/datacamp"

fileName\_train=base\_path+"/train\_features\_001.parquet"

fileName\_valid=base\_path+"/valid\_features\_001.parquet"

sqlContext = SQLContext(sc)

dataDF = sqlContext.read.parquet("hdfs://" + fileName\_train)

validDF = sqlContext.read.parquet("hdfs://" + fileName\_valid)

About 3 minutes 21 to run this one (using 16 nodes)

---------------------- SUMMARY :

eval\_train (evaluator) = 0.760874034183

eval\_test (evaluator) = 0.6080450705

F1 Test Score = 0.107182419153

DSSP\_score (valid) = 0.107713797488

################ : END.

Total Duration Time (m)= 3.0

Total Duration Time (s)= 21.292074

So this way to split the 2 activities is a way to paralyze the work inside the team.

# Test 3 : (ML) Apply exact same logic on All 4 Tags (not only html), individually (4 predict columns)

Source : datacamp\_ML\_002.py

Ça n’a pas été si simple !

Car il a fallu créer un modèle (Logistic Regression) pour chaque TAG (4), les nommer.

Puis créer une fonction qui va faire le « transform() » pour appliquer chacun des modèles sur le dataFrame (pour sortir des prédictions, puis va « merger » les résultats des prédictions en un seul dataframe de sortie : JOIN

def apply\_model(DF):

print "######## Model.Transform() for each TAG"

result\_html=Model\_html.transform(DF).select(col("id").alias("html\_id"),'html\_pred','html\_pred\_raw')

result\_jquery=Model\_jquery.transform(DF).select(col("id").alias("jquery\_id"),'jquery\_pred','jquery\_pred\_raw')

result\_css=Model\_css.transform(DF).select(col("id").alias("css\_id"),'css\_pred','css\_pred\_raw')

result\_javascript=Model\_javascript.transform(DF).select(col("id").alias("javascript\_id"),'javascript\_pred','javascript\_pred\_raw')

print "######## Merge each TAG predictions"

result=DF

result = result.join(result\_html, result.id == result\_html.html\_id,how='left')

result = result.join(result\_jquery, result.id == result\_jquery.jquery\_id,how='left')

result = result.join(result\_css, result.id == result\_css.css\_id,how='left')

result = result.join(result\_javascript, result.id == result\_javascript.javascript\_id,how='left')

return result

J’ai enlevé la partie sur les « evaluators » qu’il faudrait dupliquer aussi (en faire 4)

Ça multiplie les lignes de code. Mais pourquoi faire ?

Mais on commence à avoir quelque chose qui ressemble à un Model prédictif : 0.74

---------------------- SUMMARY :

F1\_Train\_score = 0.800427075874

F1 Test Score = 0.743364304807

DSSP\_score (valid) = 0.744532211549

################ : END.

Total Duration Time (m)= 12

Total Duration Time (s)= 22

En 12 minutes et 22 s avec 16 nodes.

# Test 4 : (ML) Code the 4 TAGS in a specific way (to get a multi-class pb) : prediction on the tuple of tags

Source : datacamp\_ML\_003.py

Au moment de créer les Features :

# EnCoding of Tags in the same order of possible\_tags : ex '0101'=css+html

data['tags\_target']=''

for existing\_tag in possible\_tags.value:

data['tags\_target']=data['tags\_target']+str(data[existing\_tag])

C’est un codage 0/1 sur 4 caractères :

Chaque caractère correspond un Tag dans l’ordre de :

([u'javascript', u'css', u'jquery', u'html'])

Par exemple :

0101 -> css + html

Puis, dans le module de ML :

Création de l’indexage du Target qui devient maintenant le « label » à prédire

label\_stringIndexer = StringIndexer(inputCol = "tags\_target", outputCol = "label").fit(dataDF)

dataDF = label\_stringIndexer.transform(dataDF)

Une fois la prediction faite, il faut utiliser l’inverse de l’indexer :

labelReversePred = IndexToString(inputCol = col\_name+"\_pred", outputCol="encoded\_pred",labels=label\_stringIndexer.labels)

result=labelReversePred.transform(result)

Puis, à partir de la prédiction encodée, il faut fabriquer la liste des Tags :

encoded\_pred=data['encoded\_pred']

predicted=[]

# Decode the encoded prediction

for i, char in enumerate(encoded\_pred):

if char=='1':

# Add the associated TAG in the predicted array

predicted.append(possible\_tags[i])

Il est possible d’utiliser un « evaluator » spécialisé pour le « multi-classes » :

#5. Evaluation of results

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

metricName="f1"

col\_name="label"

evaluator = MulticlassClassificationEvaluator(labelCol=col\_name,predictionCol=col\_name+"\_pred",metricName=metricName,)

print "################ RESULT for : Test"

eval\_test=evaluator.evaluate(result\_test)

print eval\_test

On obtient des résultats “à peine meilleurs » mais on a une infrastructure prête pour appliquer d’autres modèles plus facilement.

eval\_train (MulticlassClassificationEvaluator F1) = 0.598539713389

eval\_test (MulticlassClassificationEvaluator F1) = 0.479588797576

F1\_Train\_score = 0.812744397977

F1 Test Score = 0.74618366308

DSSP\_score (valid) = 0.74664146059

################ : END.

Total Duration Time (m)= 21.0

Total Duration Time (s)= 1.35765

[christophe.noblanc@master-bigdata data\_camp\_final]$