Lab - 4

Objectives:

1. Running ML algorithms in Spark.
2. Using scikit-learn to perform computation on driver node.
3. Using scikit-learn and Spark to perform computation in distributed setting.
4. Understanding distinction between *embarrassingly* parallel vs native parallel algorithms - scikit-learn vs Spark MLlib.

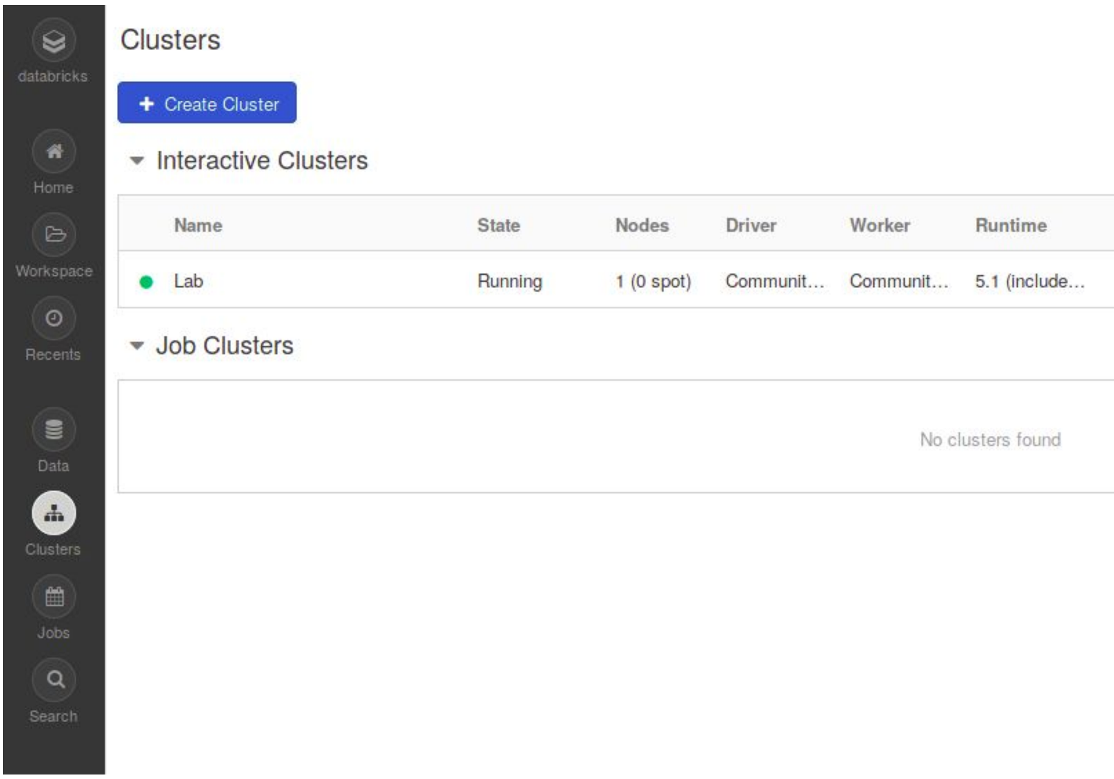
Instructions:

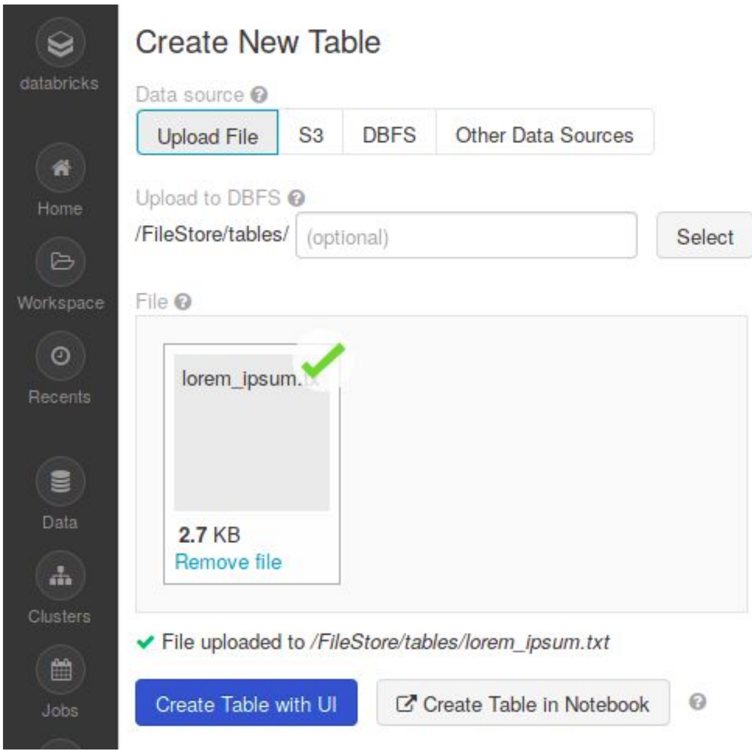
Signing up for Databricks Community Edition:

1. Create a Databricks CE account by following the link: https://databricks.com/try-databricks
2. Log in to your Databricks CE account.

Setting up Databricks CE environment:

1. Click the "**Clusters**" tab and create a cluster using the following config:
    a. Cluster name: Lab 4 (preferably)
    b. Runtime version: 5.2 (preferably)
    c. Python version: 3





2. Click the “**Data**” tab and upload the ***lorem_ipsum.txt*** file used in previous lab. Click “**Create Table in Notebook**”.
3. The notebook should be imported to Databricks CE environment.
4. Open the notebook and get yourself familiar with the setup.
5. Try out the **WordCount** example from the previous lab. Also view the Spark Jobs, DAG structure, executor node status, etc. in the same UI.

While Ambari is a popularly used Hadoop solution that offers a number of services (which we will use later on), this setup is great for running Spark in a cluster setup. You don’t have to worry about credits and can experiment with this setup easily.

Now that we are familiar with the environment, let’s use **iris** dataset to perform a Multiclass Classification using Logistic Regression.

Using scikit-learn on a single node:

```
from sklearn.preprocessing import normalize
from sklearn.cross_validation import train_test_split
from sklearn import linear_model, cross_validation
import numpy
import pandas

pandasData = pandas.read_csv("/dbfs/FileStore/tables/ads.csv",
header=None)
display(pandasData)

# Split data, into data and labels
data = pandasData.iloc[:, 0:4].values
labels = pandasData.iloc[:,4].values

data = normalize(data, axis=0)

trainingLabels, testLabels, trainingFeatures, testFeatures =
train_test_split(labels, data, test_size=0.3)
ntrain, ntest = len(trainingLabels), len(testLabels)
print(ntrain)


origAlpha = 0.5 # "alpha" is the regularization hyperparameter
origClf = linear_model.Ridge(alpha=origAlpha)
origClf.fit(data, labels)
print('Trained model with fixed alpha = %g' % origAlpha)
print('  Model intercept: %g' % origClf.intercept_)

# Score the initial model.  It does not do that well.
origScore = origClf.score(testFeatures, testLabels)
origScore

numFolds = 3 # You may want to use more (10 or so) in practice
# Extract indices for this fold
kf = cross_validation.KFold(ntrain, n_folds=3)
print(ntrain)

# Define function to execute on multiple nodes
def trainOneModel(alpha, fold):
  """
  Given 1 task (1 hyperparameter alpha value + 1 fold index), train the
corresponding model.
  Return: model, score on the fold's test data, task info.
  """
```

```
  trainIndex, valIndex = [], []
  fold_ = 0 # index into folds 'kf'
  for trainIndex_, valIndex_ in kf:
    if fold_ == fold:
      trainIndex, valIndex = trainIndex_, valIndex_
      break
    fold_ += 1
  # Get training data from the broadcast variables
  localTrainingFeatures = trainingFeaturesBroadcast.value
  localTrainingLabels = trainingLabelsBroadcast.value
  X_train, X_val = localTrainingFeatures[trainIndex],
localTrainingFeatures[valIndex]
  Y_train, Y_val = localTrainingLabels[trainIndex],
localTrainingLabels[valIndex]

  # Train the model, and score it
  clf = linear_model.Ridge(alpha=alpha)
  clf.fit(X_train, Y_train)
  score = clf.score(X_val, Y_val)
  return clf, score, alpha, fold

# "alphas" is a list of hyperparameter values to test
alphas = [0.0001, 0.00001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0,
10000.0]
# Create a list of tasks to distribute
tasks = []
for alpha in alphas:
  for fold in range(numFolds):
    tasks = tasks + [(alpha, fold)]

tasksRDD = sc.parallelize(tasks, numSlices = len(tasks))

trainingFeaturesBroadcast = sc.broadcast(trainingFeatures)
trainingLabelsBroadcast = sc.broadcast(trainingLabels)

# LEARN!  We now map our tasks RDD and apply the training function to
each task.
# After we call an action ("count") on the results, the actual training
is executed.
trainedModelAndScores = tasksRDD.map(lambda alpha_fold:
trainOneModel(alpha_fold[0], alpha_fold[1]))
trainedModelAndScores.cache()

# Collect the results.
allScores = trainedModelAndScores.map(lambda x: (x[1], x[2],
x[3])).collect()
# Average scores over folds
```

```
avgScores = dict(map(lambda alpha: (alpha, 0.0), alphas))
for score, alpha, fold in allScores:
  avgScores[alpha] += score
for alpha in alphas:
  avgScores[alpha] /= numFolds
avgScores

trainingFeatures
```

Using Spark MLLib (PySpark):

```
from pyspark.ml.feature import StringIndexer,VectorAssembler,Normalizer
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

from pyspark.ml import Pipeline

file_location =
"/databricks-datasets/Rdatasets/data-001/csv/datasets/iris.csv"

training = spark.read.format("csv").option("header",
True).load(file_location).toDF("index", "sepal_length", "sepal_width",
"petal_length", "petal_width", "species")

training = training.withColumn("sepal_length",
training["sepal_length"].cast("float")).withColumn("sepal_width",
training["sepal_width"].cast("float")).withColumn("petal_length",
training["petal_length"].cast("float")).withColumn("petal_width",
training["petal_width"].cast("float"))

train,test = training.randomSplit([0.8,0.2])

indexer = StringIndexer(inputCol="species", outputCol="label")

assembler = VectorAssembler(
    inputCols=["sepal_length", "sepal_width", "petal_length",
"petal_width"],
    outputCol="temp_features")

normalizer = Normalizer(inputCol="temp_features", outputCol="features",
p=1.0)

lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

pipeline = Pipeline(stages=[indexer, assembler, normalizer, lr])

model = pipeline.fit(train)
```

```
model.stages[-1].coefficientMatrix

paramGrid = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.001, 0.01, 0.1, 0.3, 0.5, 0.7]) \
    .build()
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,

evaluator=MulticlassClassificationEvaluator(),
                          numFolds=3)

# Run cross-validation, and choose the best set of parameters.
cvModel = crossval.fit(train)
prediction = cvModel.transform(test)
selected = prediction.select("label", "probability", "prediction")
print(MulticlassClassificationEvaluator(metricName =
"accuracy").evaluate(prediction))
```

Using Spark MLLib (Scala):

```
import org.apache.spark.ml.feature.{StringIndexer, VectorAssembler,
Normalizer}
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

val irisDf = spark.read.format("csv")
  .option("inferschema", "true")
  .option("header", "true")
 .load("/databricks-datasets/Rdatasets/data-001/csv/datasets/iris.csv")
  .toDF("index", "sl", "sw", "pl", "pw", "species")

val Array(train, test) = irisDf.randomSplit(Array(0.7, 0.3), seed =
2019)

val stringIndexer = new StringIndexer()
  .setInputCol("species")
  .setOutputCol("label")

val featureCols = Array("sl", "sw", "pl", "pw")

val vectorAssembler = new VectorAssembler()
  .setInputCols(featureCols)
  .setOutputCol("feature_temp")

val normalizer = new Normalizer()
  .setInputCol("feature_temp")
```

```
  .setOutputCol("features")

val lr = new LogisticRegression()
  .setMaxIter(10)

val pipeline = new Pipeline()
  .setStages(Array(stringIndexer, vectorAssembler, normalizer, lr))

val paramGrid = new ParamGridBuilder()
  .addGrid(lr.regParam, Array(0.0001, 0.001, 0.1, 0.3, 0.5, 0.7))
  .build()

val cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEvaluator(new MulticlassClassificationEvaluator)
  .setEstimatorParamMaps(paramGrid)
  .setNumFolds(3)
  .setParallelism(2)

val cvModel = cv.fit(train)

val prediction = cvModel.transform(test)
val evaluation = new
MulticlassClassificationEvaluator().setMetricName("accuracy").evaluate(
prediction)
println(evaluation)
```

Report:

1. Download the dataset(3d-road-network-denmark.csv) from moodle and perform regression (using Spark ML) using **Altitude** as the label. Also, note down your observations for the model you generate along with the cross validation you perform. What are the rationales behind the way your model is performing? Do attach the screenshots of your code.

2. Download the dataset from moodle(clusterData.csv) and perform clustering on the same using **UNS**. Also, note down your observations for the model you generate along with the cross validation you perform. What are the rationales behind the way your model is performing? Do attach the screenshots of your code.

Data properties:

SCG (The degree of repetition number of user for goal object material)
STG (The degree of study time for goal object materials)
STR (The degree of study time of user for related objects with goal obj
LPR (The exam performance of user for related objects with goal objc
PEG (The exam performance of user for goal objects)
UNS (The knowledge level of user)