**Spark configurations**

1. To download the spark image and launch a jupyter pyspark notebook in docker, use:

* **docker run --security-opt seccomp=unconfined --name sparklab -it -p 8888:8888 -p 4040:4040 -p 4041:4041 jupyter/pyspark-notebook**
* If there is a few space, download the version that is only 1GB instead of 4GB

**docker run -it apache/spark-py /opt/spark/bin/pyspark**

1. To start the jupyterLab when the container is stopped:

* **docker start sparklab**
* **docker exec -it sparklab jupyter server list**
* Copy the token that is generated and paste it into:

**http://127.0.0.1:8888/lab?token=…**

* Then open it in browser

1. Launch the shell on the spark interpreter:

* **docker exec -it sparklab bash**

1. Put the text file in the container:

* **docker cp D:\Pavlo\Universite\_Paris\_Saclay\T4\LSDDP\docs-tp-spark-en\lai-eliduc.txt sparklab:/**

**If the file does not appear in the main directory of the container, try to add it using GUI.**

1. Check where pyspark executable is and identify the Spark executables that are available in the same directory while being inside the container:

* **printenv | grep spark**

You should receive the following response:

* **SPARK\_HOME=/usr/local/spark**
* **PATH=/opt/conda/bin:/opt/conda/condabin:/opt/conda/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/spark/bin**

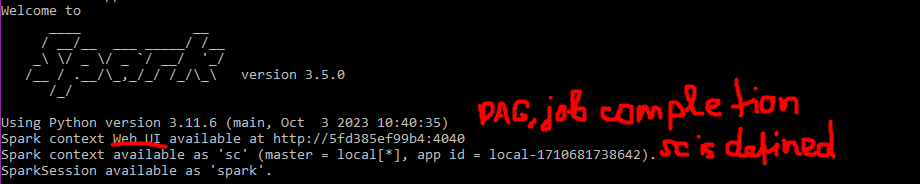
1. Instruction to use IPython (which is used by Jupyter notebooks as a kernel) interpreter for the pyspark instead of plain python use command:

* **export PYSPARK\_DRIVER\_PYTHON=/opt/conda/bin/ipython**

1. To run pyspark use command:

* **pyspark**

You should receive:



Use <http://localhost:4040/> to access GUI

**Spark Application (Interpreter)**

1. To create rdd as a Python list:

* **first\_rdd = sc.parallelize([1,1,2,3,5,8,13,21])**

1. Show first two elements:

* **first\_rdd.take(2)**

**Out:**

* **[1,1]**

1. Load the text file and name it “lignes”:

* **lignes = sc.textFile(“lei-eliduc.txt”)**

1. Count the number of lines in this RDD:

* **lignes.count()**

**Out:**

* **1296**

1. Count the number of partitions in the RDD:

* **lignes.getNumPartitions()**

**Out:**

* **2**

**Spark Application (GUI)**

1. from pyspark.sql import SparkSession

# Spark session & context

spark = SparkSession.builder.master('local').getOrCreate()

sc = spark.sparkContext

1. Here we run our application from a notebook in local mode. But in general we would have executed the Python application with spark-submit, which allows to specify cluster options.
2. Write a wordcount program in Spark:

**lines\_split = lines.flatMap(lambda line: re.split(r"\W+", line))**

**words\_in\_line = lines\_split.map(lambda word: (word, 1))**

**words\_counter = words\_in\_line.reduceByKey(lambda a,b: a + b)**

**count = words\_counter.sortBy(lambda x: x[1], ascending=False).collect()**

**result = sc.parallelize(count).saveAsTextFile("nbmots")** *# converts list into scala collection and saves as a text file*

1. To read the content of the result file, open bash, move to the nbmots directory and use command:

* **cat \***

1. Add a transformationto the above “program” . We wish to drop items that do appear less than 8 times. Do not write the result into a file but display the first 10 RDD elements in the REPL.

* **remove\_small\_occ = words\_counter.filter(lambda x: x[1] >= 8)**

**remove\_small\_occ.take(10)**

**Spark Dataframes**

1. Retrieve the datafile:

* **wget -O food.csv** [**https://data.cityofchicago.org/api/views/4ijn-s7e5/rows.csv**](https://data.cityofchicago.org/api/views/4ijn-s7e5/rows.csv)

1. Load all the libraries needed for this lab:

* **from pyspark.ml import Pipeline**

**from pyspark.ml.classification import LogisticRegression**

**from pyspark.ml.feature import HashingTF, Tokenizer**

**from pyspark.sql import Row**

**from pyspark.sql.functions import udf, desc, col**

**from pyspark.sql.types import \***

**from pyspark.sql import SparkSession**

1. Load a Dataframe df1 from file food.csv. Then, display the Dataframe schema using .show() and .printSchema()

* # Spark session & context

**spark = SparkSession.builder.master('local').getOrCreate()**

**sc = spark.sparkContext**

* **df1 = spark.read.csv("food.csv")**

**df1.show()**

* **df1.printSchema()**

1. Create a dataframe inspections by projecting on the columns that we shall use:

'Inspection ID', 'DBA Name','results', 'Violations' and then removing all lines containing a “NULL”.

* **inspections = df1.select("Inspection ID", "DBA Name", "Results", "Violations")**

**inspections.show()**

* **inspections = inspections.dropna()**

**inspections.show()**

Would you save or waste time if you loaded the inspections dataframe from file food.csv instead of df1? Explain.

*It would cost more if I loaded again the data from csv file into newly created dataFrame, since it requires the read from the disk that might take a great amount of time if the file is huge.*

1. Compute from inspections the list nbre of possible inspection results, and the number of inspections yielding such results, ordered by decreasing number of inspections.

* **nbre= inspections.groupBy(“Results”).count().withColumnRenamed(“count”, “nb”).orderBy(col(“nb”).desc())**

**nbre.printSchema()**

1. Display the first 4 lines of the result: what difference between .take(4) and .show(4)?

* **nbre.take(4)**

**Out:**

**[Row(Results='Pass', nb=105212),**

**Row(Results='Fail', nb=49172),**

**Row(Results='Pass w/ Conditions', nb=40204),**

**Row(Results='No Entry', nb=658)]**

* **nbre.show(4)**

**Out:**

**+------------------+------+**

**| Results| nb|**

**+------------------+------+**

**| Pass|105212|**

**| Fail| 49172|**

**|Pass w/ Conditions| 40204|**

**| No Entry| 658|**

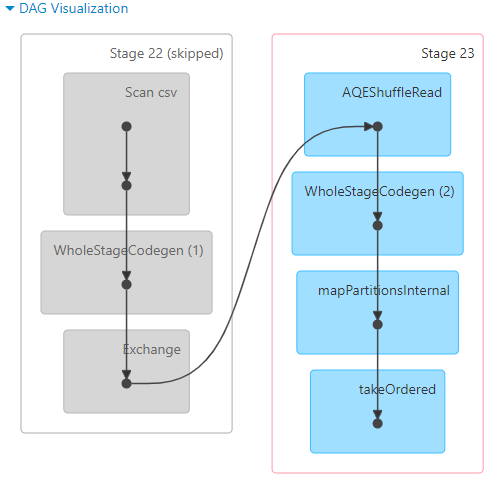
**+------------------+------+**

*Take(4) shows the results in a form of list where each element is an instance of Row of the dataFrame.*

*Show(4) displays the results in a shape of DataFrame.*

1. Display the execution plan nbre with explain(true).

* **nbre.explain(True)**

1. Check in the web interface the DAG of tasks.
2. Display stats about nbre using .describe()

* **nbre.describe()**

Out:

* **DataFrame[summary: string, Results: string, nb: string]**

1. Visualize the number of inspections for each possible result in a pie chart

* **plot\_pd = nbre.toPandas()**

**plot\_pd.set\_index("Results", inplace=True)**

**plt = plot\_pd.plot.pie(y="nb", figsize=(10,10))**

**plt.figure.savefig('pie.pdf')**

**Predictions with spark**

**Labelling data: defining categories**

1. Labels for a logistic regression:

* **def labelForResults(s):**

**if s == 'Fail':**

**return 0.0**

**elif s == 'Pass w/ Conditions' or s == 'Pass':**

**return 1.0**

**else:**

**return -1.0**

1. Register the above function as a udf, and use it to transform the data: we want to obtain a 2-column DataFrame labeledData : the columns are label and violations, where category label takes values 0.0 or 1.0.. Values less than 0 are filtered out.

* **udf\_logistic = udf(labelForResults, DoubleType())**

**labeledData = inspections.select(udf\_logistic("Results"), "Violations").withColumnRenamed("labelForResults(Results)", "label").withColumnRenamed("Violations", "violations")**

**labeledData = labeledData.filter(labeledData["label"] >= 0)**

**labeledData.show(20)**

1. We split our data in 2: (i) a Dataframe training containing 75% of labeledData records will beused to train the model, whereas (ii) Dataframe validationDf containing the remaining records will be used for validation. Partition labeledData records through random sampling. Use the Spark function randomSplit() , taking 105 as a ( seed) to initialize the random generator.

* trainingDf, validationDf = labeledData.randomSplit([0.75, 0.25], 105)

**Defining the model: specifying predictive variables and tuning parameters**

1. We have to extract from the text field violations the variables from which our model will buid predictions. For this, we convert that field to a vector of numbers. The logistic regression will then be applied to those vectors.

We define a 3-steps pipeline:

• the first step splits violations into a sequence of words (Tokenizer)

• the second step converts the sequences to a frequency vector (each word gets assigned an index, then we map each list to its corresponding frequency vector simply by counting the occurrences of each word)

• the last step applies the linear regression (use 10 iterations, with regularisation parameter equal 0.01)

* **tokenizer = Tokenizer(inputCol="violations", outputCol="words")**

**hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")**

**lr = LogisticRegression(maxIter=10, regParam=0.01)**

**pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])**

1. Train your pipeline on the training data, then validate the pipeline on test data.

* **model = pipeline.fit(trainingDf)**
* **prediction = model.transform(validationDf)**
* **selected = prediction.select("violations", "probability", "prediction")**

**counter = 0**

**for row in selected.collect():**

**if counter == 20:**

**break**

**violation, prob, prediction = row**

**print(f"{violation} {str(prob)} {prediction}")**

**counter+=1**

**Executing a spark application**

1. Use the wordcount code (wordcount.py) to write an application that takes as parameters the name of input and output files.

* **/usr/local/spark/bin/spark-submit --master local[1] wordcount.py**

1. Read and understand some of the examples. Identify where cache() is used and why:

* **/usr/local/spark/examples/src/main/python/ml/kmeans\_example.py**

from pyspark.ml.clustering import KMeans

from pyspark.ml.evaluation import ClusteringEvaluator

# $example off$

from pyspark.sql import SparkSession

if \_\_name\_\_ == "\_\_main\_\_":

spark = SparkSession\

.builder\

.appName("KMeansExample")\

.getOrCreate()

# $example on$

# Loads data.

dataset = spark.read.format("libsvm").load("data/mllib/sample\_kmeans\_data.txt")

# Trains a k-means model.

kmeans = KMeans().setK(2).setSeed(1)

model = kmeans.fit(dataset)

# Make predictions

predictions = model.transform(dataset)

# Evaluate clustering by computing Silhouette score

evaluator = ClusteringEvaluator()

silhouette = evaluator.evaluate(predictions)

print("Silhouette with squared euclidean distance = " + str(silhouette))

# Shows the result.

centers = model.clusterCenters()

print("Cluster Centers: ")

for center in centers:

print(center)

# $example off$

spark.stop()

*In the provided KMeans example script,* ***cache()*** *is not explicitly used. Typically, for ML algorithms like KMeans, caching can be beneficial when you have a large dataset that fits into memory and you're performing multiple iterations over the dataset, as it happens during the training phase. However, in this simplified example, the dataset is likely small.*

* **/usr/local/spark/examples/src/main/python/transitive\_closure.py**

import sys

from random import Random

from typing import Set, Tuple

from pyspark.sql import SparkSession

numEdges = 200

numVertices = 100

rand = Random(42)

def generateGraph() -> Set[Tuple[int, int]]:

edges: Set[Tuple[int, int]] = set()

while len(edges) < numEdges:

src = rand.randrange(0, numVertices)

dst = rand.randrange(0, numVertices)

if src != dst:

edges.add((src, dst))

return edges

if \_\_name\_\_ == "\_\_main\_\_":

"""

Usage: transitive\_closure [partitions]

"""

spark = SparkSession\

.builder\

.appName("PythonTransitiveClosure")\

.getOrCreate()

partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2

tc = spark.sparkContext.parallelize(generateGraph(), partitions).cache()

# Linear transitive closure: each round grows paths by one edge,

# by joining the graph's edges with the already-discovered paths.

# e.g. join the path (y, z) from the TC with the edge (x, y) from

# the graph to obtain the path (x, z).

# Because join() joins on keys, the edges are stored in reversed order.

edges = tc.map(lambda x\_y: (x\_y[1], x\_y[0]))

oldCount = 0

nextCount = tc.count()

while True:

oldCount = nextCount

# Perform the join, obtaining an RDD of (y, (z, x)) pairs,

# then project the result to obtain the new (x, z) paths.

new\_edges = tc.join(edges).map(lambda \_\_a\_b: (\_\_a\_b[1][1], \_\_a\_b[1][0]))

tc = tc.union(new\_edges).distinct().cache()

nextCount = tc.count()

if nextCount == oldCount:

break

print("TC has %i edges" % tc.count())

spark.stop()

*In the Transitive Closure example, cache() is used right after the initial RDD (tc) is created:*

**tc = spark.sparkContext.parallelize(generateGraph(), partitions).cache()**

*and again after each iteration when the transitive closure (tc) is expanded:*

**tc = tc.union(new\_edges).distinct().cache()**

*Here, caching is crucial because the algorithm is iterative. Each iteration expands the transitive closure by joining with the original edges, which requires recomputing the RDDs in each iteration. Caching the RDD after each iteration avoids recomputation of the entire lineage of transformations leading up to the current state of the RDD, significantly improving performance.*

* **/usr/local/spark/examples/src/main/python/pagerank.py**

import re

import sys

from operator import add

from typing import Iterable, Tuple

from pyspark.resultiterable import ResultIterable

from pyspark.sql import SparkSession

def computeContribs(urls: ResultIterable[str], rank: float) -> Iterable[Tuple[str, float]]:

"""Calculates URL contributions to the rank of other URLs."""

num\_urls = len(urls)

for url in urls:

yield (url, rank / num\_urls)

def parseNeighbors(urls: str) -> Tuple[str, str]:

"""Parses a urls pair string into urls pair."""

parts = re.split(r'\s+', urls)

return parts[0], parts[1]

if \_\_name\_\_ == "\_\_main\_\_":

if len(sys.argv) != 3:

print("Usage: pagerank <file> <iterations>", file=sys.stderr)

sys.exit(-1)

print("WARN: This is a naive implementation of PageRank and is given as an example!\n" +

"Please refer to PageRank implementation provided by graphx",

file=sys.stderr)

# Initialize the spark context.

spark = SparkSession\

.builder\

.appName("PythonPageRank")\

.getOrCreate()

# Loads in input file. It should be in format of:

# URL neighbor URL

# URL neighbor URL

# URL neighbor URL

# ...

lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])

# Loads all URLs from input file and initialize their neighbors.

links = lines.map(lambda urls: parseNeighbors(urls)).distinct().groupByKey().cache()

# Loads all URLs with other URL(s) link to from input file and initialize ranks of them to one.

ranks = links.map(lambda url\_neighbors: (url\_neighbors[0], 1.0))

# Calculates and updates URL ranks continuously using PageRank algorithm.

for iteration in range(int(sys.argv[2])):

# Calculates URL contributions to the rank of other URLs.

contribs = links.join(ranks).flatMap(lambda url\_urls\_rank: computeContribs(

url\_urls\_rank[1][0], url\_urls\_rank[1][1] # type: ignore[arg-type]

))

# Re-calculates URL ranks based on neighbor contributions.

ranks = contribs.reduceByKey(add).mapValues(lambda rank: rank \* 0.85 + 0.15)

# Collects all URL ranks and dump them to console.

for (link, rank) in ranks.collect():

print("%s has rank: %s." % (link, rank))

spark.stop()

*In the PageRank example,* ***cache()*** *is used when initializing the links from the input file:*

**links = lines.map(lambda urls: parseNeighbors(urls)).distinct().groupByKey().cache()**

*The PageRank algorithm is inherently iterative, updating the ranks of pages over several iterations based on the links between pages. Since the links RDD represents the structure of the web graph and does not change between iterations, caching it is essential. This prevents Spark from re-reading the input and recalculating the graph structure in each iteration of the algorithm, thereby reducing the overall computation time.*