navigation

December 8, 2020

1 LAB BIG DATA : Introduction to Spark

1.0.1 Author : Chouaib Mounaime

1.1 Importing packages and Initializing SparkContext

[3]:
```
import sys
from pyspark import SparkContext
from timeit import default_timer as timer
import time
import math

# start spark with 1 worker thread
sc = SparkContext("local[1]")
sc.setLogLevel("ERROR")
```

1.2 Definition of function used in this lab

[4]:
```
# Finds out the index of "name" in the array firstLine
# returns -1 if it cannot find it
def findCol(firstLine, name):
        if name in firstLine:
                return firstLine.index(name)
        else:
                return -1


# Remove quotes around a string
# And convert it to lowercase
# Used in question 6 and 8 (to have places names with the same format)
def stringFormat(str):
    return str.replace('"','').lower()

# Round a float number with n decimal
# Used in question 9 (to round the temperature value)
def truncate(f, n):
    return math.floor(f * 10 ** n) / 10 ** n
```

1.3 Importing, splitting and caching Dataset

[5]:
```
#### Driver program


# read the input file into an RDD[String]
wholeFile = sc.textFile("./data/CLIWOC15.csv")

# The first line of the file defines the name of each column in the cvs file
# We store it as an array in the driver program
firstLine = wholeFile.filter(lambda x: "RecID" in x).collect()[0].
 ↪replace('"','').split(',')

# filter out the first line from the initial RDD
entries = wholeFile.filter(lambda x: not ("RecID" in x))

# split each line into an array of items
entries = entries.map(lambda x : x.split(','))

# keep the RDD in memory
entries.cache()
```

[5]:
```
PythonRDD[3] at RDD at PythonRDD.scala:53
```

1.4 1. New version : ignoring white-space :

[6]:
```
##### Create an RDD that contains all nationalities observed in the
##### different entries

# Information about the nationality is provided in the column named
# "Nationality"

# First find the index of the column corresponding to the "Nationality"
column_index=findCol(firstLine, "Nationality")
print("{} corresponds to column {}".format("Nationality", column_index))

# Use 'map' to create a RDD with all nationalities and 'distinct' to remove␣
 ↪duplicates
nationalities = entries.map(lambda x: x[column_index])
nationalities = nationalities.map(lambda x: x.replace(" ", "")).distinct()

# Display the 5 first nationalities
print("A few examples of nationalities:")
for elem in nationalities.sortBy(lambda x: x).take(5):
    print(elem)
```

```
Nationality corresponds to column 20
```

```
A few examples of nationalities:
"American"
"British"
"Danish"
"Dutch"
"French"
```

1.5 2. Count the total number of observations included in the dataset :

[7]:
```
#Entries is an RDD which contains all the lines expect the first one
nbObservations = entries.count()

print('the total number of observations :',nbObservations)
```

```
the total number of observations : 280280
```

1.6 3. counting the number of years over which observations have been made

[78]:
```
#Getting the index of the column "Year" in the first line
year_index = findCol(firstLine, "Year")   # ==> 40

#Extract the Year on each entry
years = entries.map(lambda x: x[year_index])

#Filtering observations with "NA" value,
#and keep only distinct observations
years = years.filter(lambda x: x != "NA").distinct()

#Counting number of years
nb_years = years.count()

print('the number of years :',nb_years)
```

```
the number of years : 118
```

1.7 4. Display the oldest and the newest year of observation

[79]:
```
##Using min and max actions
oldest = years.min()
newest = years.max()

print('the oldest year :',oldest)
print('the newest year :',newest)
```

```
the oldest year : 1662
the newest year : 1855
```

2 5. Display the years with the minimum and the maximum number of observations (and the corresponding number of observations)

[80]:
```
#new copy of the years
#and filtering "NA" values
year_observations = entries.map(lambda x: x[year_index])
year_observations = year_observations.filter(lambda x: x != "NA")

#creation a tuples for each observation with value 1
year_observations = year_observations.map(lambda x: (x, 1))

#group tuples by key (the year) then count the size of each group
year_observations = year_observations.groupByKey().mapValues(len)

max_observations = year_observations.sortBy(lambda x: -x[1]).first()
min_observations = year_observations.sortBy(lambda x: x[1]).first()

print('the minimum number of observations was in :',min_observations)
print('the maximum number of observations was in :',max_observations)
```

```
the minimum number of observations was in : ('1747', 4)
the maximum number of observations was in : ('1778', 8509)
```

3 6. Count the distinct departure places (column “VoyageFrom”) using two methods (i.e., using the function distinct() or reduceByKey()) and compare the execution time.

[81]:
```
#getting the index of the column "VoyageFrom" in the first line
departure_index = findCol(firstLine, "VoyageFrom")   # ==> 14

#extract the departure place on each entry
departure_places = entries.map(lambda x: x[departure_index])

#filtering "NA" values
departure_places = departure_places.filter(lambda x: x != 'NA')

#convert to lowercase and remove quotes around (see StringFormat above)
departure_places = departure_places.map(lambda x: stringFormat(x))
```

3.1 6.1 Using the function distinct()

[82]:
```
start = timer()
count1 = departure_places.distinct().count()
end = timer()

print('counted with distinct :', count1)
print('elapsed time :', truncate(end - start,2),'sec.')
```

```
counted with distinct : 974
elapsed time : 4.1 sec.
```

3.2 6.2 Using the function reduceByKey()

[83]:
```
start = timer()
pairs = departure_places.map(lambda x: (x, 1))
count2 = pairs.reduceByKey(lambda a, b: a + b)
end = timer()

print('counted with reduceByKey :', count1)
print('elapsed time :', truncate(end - start,2),'sec.')
```

```
counted with reduceByKey : 974
elapsed time : 0.04 sec.
```

By comparing execution times, we can conclude that the **recudeByKey** method is almost **20 times faster** than the **distinct** method.

3.3 7. Display the 10 most popular departure places

[88]:
```
#creating a tuple for each place observation with value 1
##Using departures RDD created previously
places_tuples = departure_places.map(lambda x: (x, 1))

#group tuples by key (the year) then count the size of each group
places_tuples = places_tuples.reduceByKey(lambda a, b: a + b).sortBy(lambda x:␣
 ↪-x[1])

print('the 10 most popular departure places are :')
for place in places_tuples.take(10):
    print(f'\t- {place[0]} :\t {place[1]}')
```

```
the 10 most popular departure places are :
        - batavia :      25920
        - la coruña :    16120
        - montevideo :   11625
        - rotterdam :    9757
        - nederland :    8697
```

```
        - spithead :      8298
        - la habana :     7906
        - cádiz :         7522
        - nieuwediep :    6713
        - texel :         6445
```

3.4 8. Display the 10 roads (defined by a pair "VoyageFrom" and "VoyageTo") the most often taken.

3.5 Version 1 : version where a pair A-B and a pair B-A correspond to different roads

3.5.1 NB : We assume that pairs where (departure place = destination place) are not considered as a road

[89]:
```
voyageFrom_index = findCol(firstLine, "VoyageFrom")  #  ==> 14
voyageTo_index   = findCol(firstLine, "VoyageTo")    #  ==> 15

# RDD of pair (from,to) of places
# We format places at this step (to lowercase and removed quotes)
# This RDD will be used for the VERSION 2.
roads = entries.map(
    lambda x: (
                stringFormat(x[voyageFrom_index]) ,
                stringFormat(x[voyageTo_index])
               ))

# Filtering "na" values (not "NA" because we converted roads to lowercase)
# Filtering road where departure place is equal to destination place
# This RDD will be used for the VERSION 2.
roads = roads.filter(lambda x: x[0] != 'na' and x[1] != 'na' and x[0] != x[1])

roads1 = roads.map(lambda x: (x, 1))
roads1 = roads1.reduceByKey(lambda a, b : a+b).sortBy(lambda x: -x[1])

print("The 10 road most often taken :")
for road, nb in roads1.take(10):
    print('\t-',road,'\t', nb, 'times')
```

```
The 10 road most often taken :
        - ('la coruña', 'montevideo')    8514 times
        - ('montevideo', 'la coruña')    8459 times
        - ('la coruña', 'la habana')     7525 times
        - ('rotterdam', 'batavia')       7341 times
        - ('la habana', 'la coruña')     6068 times
        - ('batavia', 'rotterdam')       5256 times
        - ('nieuwediep', 'batavia')      5256 times
        - ('batavia', 'nieuwediep')      4564 times
        - ('nederland', 'batavia')       3996 times
```

```
        - ('batavia', 'nederland')       3534 times
```

3.6 Version 2 : version where A-B and B-A are considered as the same road

[90]:
```
# Here we use the RDD roads computed above.
# Storing road pairs according to the alphabetical order
# x ==> ( (minPlace, maxPlace), 1 )
roads2 = roads.map(lambda x: ((min(x[0], x[1]), max(x[0], x[1])), 1) )

# Now we can reduce by key without having redundant roads
roads2 = roads2.reduceByKey(lambda a, b : a+b).sortBy(lambda x: -x[1])

print("The 10 road most often taken :")
for road, nb in roads2.take(10):
    print('\t-',road,'\t', nb, 'times')
```

```
The 10 road most often taken :
        - ('la coruña', 'montevideo')    16973 times
        - ('la coruña', 'la habana')     13593 times
        - ('batavia', 'rotterdam')       12597 times
        - ('batavia', 'nieuwediep')      9820 times
        - ('batavia', 'nederland')       7530 times
        - ('batavia', 'texel')   3998 times
        - ('amsterdam', 'batavia')       3164 times
        - ('batavia', 'hellevoetsluis')          2661 times
        - ('cádiz', 'montevideo')        2231 times
        - ('cádiz', 'la habana')         1668 times
```

3.7 9. Compute the hottest month (defined by column "Month") on average over the years considering all temperatures (column "ProbTair") reported in the dataset

[91]:
```
months_index = findCol(firstLine, 'Month')
probTair_index = findCol(firstLine, 'ProbTair')

# For each entry in the dataset, we create a tuple with the month and probtair
temperatures = entries.map(lambda x: (x[months_index], x[probTair_index]))

# Filtering 'NA' value
temperatures = temperatures.filter(lambda x: x[0] != 'NA' and x[1] != 'NA')

# Cast month and probtair values from String to Int/Float to use arithmetic␣
 ↪operations (addition, division ...)
# x ==> (month, (probtair,1) )
temperatures = temperatures.map(lambda x: (int(x[0]), (float(x[1]), 1)))

# Computing the sum of probtair and the sum of observations of each month
temperatures = temperatures.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
```

```
# Computing the average temperature for each month
# We divide the sum of the temperature for each month by the total number of␣
 ↪probtair observations for the same month
# x[0]    == the month
# x[1][0] == the sum of probtair for this month
# x[1][1] == the sum of probtair observations for this month
temperatures = temperatures.map(lambda x: (x[0], x[1][0] / x[1][1])).
 ↪sortBy(lambda x: -x[1])


month, temp = temperatures.first()
print(f'the hottest month is : month {month} with {truncate(temp,1)} °C\n')

print(f'all months ordered by average temperature :')
for month, temp in temperatures.collect():
    print(f'\t- month : {month}\tavg temp : {truncate(temp,1)} °C')
```

```
the hottest month is : month 2 with 22.9 °C

all months ordered by average temperature :
        - month : 2     avg temp : 22.9 °C
        - month : 1     avg temp : 22.8 °C
        - month : 11    avg temp : 22.6 °C
        - month : 10    avg temp : 22.6 °C
        - month : 3     avg temp : 22.5 °C
        - month : 4     avg temp : 22.4 °C
        - month : 5     avg temp : 22.3 °C
        - month : 9     avg temp : 22.2 °C
        - month : 8     avg temp : 22.2 °C
        - month : 12    avg temp : 22.2 °C
        - month : 6     avg temp : 22.0 °C
        - month : 7     avg temp : 21.8 °C
```