ex30

August 12, 2022

[1]:
```
inputPath = "data/Ex30/data"
outputPath = "out30/"
```

[ ]:
```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("ex30")
sc = SparkContext(conf=conf)
```

[3]:
```
inputRDD = sc.textFile(inputPath).filter(lambda line:line.find("google")>0)
```

[4]:
```
inputRDD.saveAsTextFile(outputPath)
```

ex31

August 12, 2022

[ ]:
```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName("ex31")
sc = SparkContext(conf=conf)
```

[2]:
```
inputPath = "data/Ex31/data"
outputPath = "out31/"
```

[10]:
```
def filterAndExtractIp(line):
    IPS = list()

    if line.lower().find("google")>=0:
        IP = line.split("-")[0]
        IPS.append(IP)
    return IPS
```

[11]:
```
ipsRDD = sc.textFile(inputPath).flatMap(filterAndExtractIp).distinct()
```

[12]:
```
ipsRDD.saveAsTextFile(outputPath)
```

ex32

August 12, 2022

[ ]:
```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName("ex32")
sc = SparkContext(conf=conf)
```

[4]:
```
inputPath = "data/Ex32/data"
```

[3]:
```
pm10RDD = sc.textFile(inputPath).map(lambda line:line.split(",")[2])
maxpm10 = pm10RDD.max()
print(maxpm10)
```

```
60.2
```

ex33

August 12, 2022

[ ]:
```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("ex33")
sc = SparkContext(conf=conf)
```

[2]:
```
inputPath = "data/Ex33/data"
```

[ ]:
```
pm10RDD = sc.textFile(inputPath).map(lambda line:line.split(",")[2]).top(3)
print(pm10RDD)
```

ex34

August 12, 2022

[ ]:
```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("ex34")
sc = SparkContext(conf=conf)
```

[2]:
```
inputPath = "data/Ex34/data"
outputPath = "out34/"
```

[3]:
```
inputRDD = sc.textFile(inputPath)
maxPM10 = inputRDD.map(lambda line:float(line.split(",")[2])).max()
maxPM10ReadingsRDD = inputRDD.filter(lambda line:float(line.
 ↪split(",")[2])==maxPM10)
```

[4]:
```
maxPM10ReadingsRDD.saveAsTextFile(outputPath)
```

ex35

August 12, 2022

[ ]:
```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("ex35")
sc = SparkContext(conf=conf)
```

[6]:
```
inputPath = "data/Ex35/data/"
outputPath = "out35/"
```

[7]:
```
inputRDD = sc.textFile(inputPath)
maxpm10Value = inputRDD.map(lambda line:float(line.split(",")[2])).max()
```

[8]:
```
pm10DatesRDD = inputRDD.filter(lambda line:float(line.
↪split(",")[2])==maxpm10Value).map(lambda line:line.split(",")[1])
```

[9]:
```
pm10DatesRDD.saveAsTextFile(outputPath)
```

ex36

August 12, 2022

[ ]:
```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("ex36")
sc = SparkContext(conf=conf)
```

[2]:
```
inputPath = "data/Ex36/data"
```

[3]:
```
inputRDD = sc.textFile(inputPath)
pm10ReadingsRDD = inputRDD.map(lambda line : float(line.split(",")[2]))
readings = pm10ReadingsRDD.count()
totalpm10 = pm10ReadingsRDD.reduce(lambda r1,r2: r1+r2)
print(totalpm10/readings)
```

```
39.86666666666667
```

ex37

August 12, 2022

[ ]:
```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("ex37")
sc = SparkContext(conf=conf)
```

[7]:
```
inputPath = "data/Ex37/data/"
outputPath = "out37/"
```

[8]:
```
inputRDD = sc.textFile(inputPath)
sensorPM10RDD = inputRDD.map(lambda line : (line.split(",")[0],line.
 ↪split(",")[2]))
```

[9]:
```
groupedPairRDD = sensorPM10RDD.reduceByKey(lambda pm1, pm2 : max(pm1, pm2))
```

[10]:
```
groupedPairRDD.saveAsTextFile(outputPath)
```

[ ]:
```

```

ex38

August 12, 2022

[ ]:
```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName("ex38")
sc = SparkContext(conf=conf)
```

[8]:
```
inputPath = "data/Ex38/data/"
outputPath = "out38/"
```

[9]:
```
inputRdd = sc.textFile(inputPath).filter(lambda line : float(line.
 ↪split(",")[2])>50.0)
```

[10]:
```
readingsRdd = inputRdd.map(lambda line : (line.split(",")[0],1))
```

[11]:
```
counterRdd = readingsRdd.reduceByKey(lambda count1, count2 : count1 + count2)
```

[12]:
```
finalRdd = counterRdd.filter(lambda line : line[1]>=2)
```

[13]:
```
finalRdd.saveAsTextFile(outputPath)
```

ex392

August 12, 2022

[ ]:
```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("ex392")
sc = SparkContext(conf=conf)
```

[2]:
```
inputPath = "data/Ex39bis/data/"
outputPath = "out392/"
```

[3]:
```
inputRDD = sc.textFile(inputPath)
```

[4]:
```
datesRDD = inputRDD\
    .filter(lambda line : float(line.split(",")[2])>50.0)\
        .map(lambda line : (line.split(",")[0],line.split(",")[1]))\
            .groupByKey()\
                .mapValues(lambda dates: list(dates))
##stesso procedimento della versione precedente
```

[5]:
```
#colleziono tutti gli ID
sensorsRDD = inputRDD\
    .map(lambda line : line.split(",")[0])
#rimuovo gli ID buoni e rimango solo con quelli che non hanno superato la␣
 ↪treshold, e li mappo con una lista vuota in un pair RDD
badSensorsRDD = sensorsRDD.subtract(datesRDD.keys())\
    .map(lambda sensorid : (sensorid, list()))
```

[6]:
```
finalRDD = badSensorsRDD.union(datesRDD) #infine faccio l'unione dei due RDD
```

[7]:
```
finalRDD.saveAsTextFile(outputPath)
```

ex39

August 12, 2022

[ ]:
```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("ex39")
sc = SparkContext(conf=conf)
```

[8]:
```
inputPath = "data/Ex39/data/"
outputPath = "out39/"
```

[9]:
```
inputRDD = sc.textFile(inputPath).filter(lambda line : float(line.
 ↪split(",")[2])>50.0)
```

[10]:
```
readingsRDD = inputRDD\
.map(lambda line : (line.split(",")[0],line.split(",")[1]))\
.groupByKey()\
.mapValues(lambda dates: list(dates))
```

[11]:
```
readingsRDD.saveAsTextFile(outputPath)
```

ex40

August 12, 2022

[ ]:
```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("ex40")
sc = SparkContext(conf=conf)
```

[5]:
```
inputPath = "data/Ex40/data/"
outputPath = "out40/"
```

[6]:
```
inputRDD = sc.textFile(inputPath)
```

[7]:
```
datesRDD = inputRDD.filter(lambda line : float(line.split(",")[2])>50.0)\
    .map(lambda line : (line.split(",")[0], line.split(",")[1]))\
        .groupByKey()\
            .mapValues(lambda dates : len(list(dates)))\
                .map(lambda record : (record[1], record[0]))\
                    .saveAsTextFile(outputPath)
```

ex41

August 12, 2022

[ ]:
```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName("ex41")
sc = SparkContext(conf=conf)
```

[10]:
```
inputPath = "data/Ex41/data/"
outputPath = "out41/"
k = 1
```

[11]:
```
inputRDD = sc.textFile(inputPath)
```

[12]:
```
pm10ReadingsRDD = inputRDD.filter(lambda line : float(line.split(",")[2])>50.0)\
    .map(lambda line : (line.split(",")[0], 1))\
        .reduceByKey(lambda value1, value2: value1+value2)\
            .top(k, lambda reading : reading[1])
```

[13]:
```
finalRDD = sc.parallelize(pm10ReadingsRDD).saveAsTextFile(outputPath)
```

ex42

August 12, 2022

[ ]:
```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName("ex42")
sc = SparkContext(conf=conf)
```

[23]:
```
inputPathQuestions = "data/Ex42/data/questions.txt"
inputPathAnswers = "data/Ex42/data/answers.txt"
outputPath = "out42/"
```

[24]:
```
questionsRDD = sc.textFile(inputPathQuestions)
answersRDD = sc.textFile(inputPathAnswers)
```

[25]:
```
questionPairRDD = questionsRDD.map(lambda line : (line.split(",")[0], line.
 ↪split(",")[2]))
answerPairRDD = answersRDD.map(lambda line : (line.split(",")[1], line.
 ↪split(",")[3]))
```

[26]:
```
questionAnswersRDD = questionPairRDD.cogroup(answerPairRDD)
```

[27]:
```
finalRDD = questionAnswersRDD.mapValues(lambda value : (list(value[0]),␣
 ↪list(value[1])))
```

[28]:
```
finalRDD.saveAsTextFile(outputPath)
```

ex43

August 12, 2022

[14]:
```
# Solution Ex. 43
```

[1]:
```
#inputPathReadings = "/data/students/bigdata-01QYD/ex_data/Ex43/data/readings.
 ↪txt"
#inputPathNeighbors = "/data/students/bigdata-01QYD/ex_data/Ex43/data/neighbors.
 ↪txt"
#outputPath = "res_out_Ex43/"
#outputPath2 = "res_out_Ex43_2/"
#outputPath3 = "res_out_Ex43_3/"
#thresholdFreeSlots = 3
#thresholdCriticalPercentage = 0.8

inputPathReadings = "data/Ex43/data/readings.txt"
inputPathNeighbors = "data/Ex43/data/neighbors.txt"
outputPath = "res_out_Ex43/"
outputPath2 = "res_out_Ex43_2/"
outputPath3 = "res_out_Ex43_3/"
thresholdFreeSlots = 3
thresholdCriticalPercentage = 0.8
```

[2]:
```
# Solution Ex. 43 - part I
# Selection of the stations with a percentage of critical situations
# greater than 80%
```

[3]:
```
# Read the content of the readings file
readingsRDD = sc.textFile(inputPathReadings).cache()
```

[4]:
```
def criticalSituation(line):
    fields = line.split(",")
    # fields[0] is the station id
    # fields[5] is the number of free slots
    stationId = fields[0]
    numFreeSlots = int(fields[5])

    if  numFreeSlots < thresholdFreeSlots:
        return (stationId, (1, 1))
    else:
```

```
        return (stationId, (1, 0))
```

[5]:
```
# Count the number of total and critical readings for each station
# Create an RDD of pairs with
# key: stationId
# value: (numReadings, numCriticalReadings)
# ------- numReadings: 1 for each input line
# --------numCriticalReadings: 0 if the situation is not critical. 1 if it is␣
 ↪critical
stationCountPairRDD = readingsRDD.map(criticalSituation)
```

[7]:
```
#stationCountPairRDD.collect()
```

[8]:
```
# Compute the number of total and critical readings for each station
stationTotalCountPairRDD = stationCountPairRDD\
.reduceByKey(lambda c1, c2: (c1[0]+c2[0], c1[1]+c2[1]) )
```

[9]:
```
#stationTotalCountPairRDD.collect()
```

[10]:
```
# Compute the percentage of critical situations for each station
stationPercentagePairRDD = stationTotalCountPairRDD\
.mapValues(lambda counters: counters[1]/counters[0])
```

[11]:
```
#stationPercentagePairRDD.collect()
```

[12]:
```
# Select stations with percentage > 80%
selectedStationsPairRDD = stationPercentagePairRDD\
.filter(lambda sensorPerc: sensorPerc[1]>thresholdCriticalPercentage)
```

[13]:
```
#selectedStationsPairRDD.collect()
```

[14]:
```
# Sort the stored stations by decreasing percentage of critical situations
selectedStationsSortedPairRDD = selectedStationsPairRDD\
.sortBy(lambda sensorPerc: sensorPerc[1], ascending=False)
```

[15]:
```
#selectedStationsSortedPairRDD.collect()
```

[17]:
```
selectedStationsSortedPairRDD.saveAsTextFile(outputPath)
```

[18]:
```
# Solution Ex. 43 - part II
# Selection of the pairs (timeslot, station) with a percentage of
# critical situations greater than 80%
```

[19]:
```
def criticalSituationTimeslots(line):

    fields = line.split(",")
```

```
    # fields[0] is the station id
    # fields[2] is the hour
    # fields[5] is the number of free slots

    stationId = fields[0]
    numFreeSlots = int(fields[5])

    minTimeslotHour = 4 * ( int(fields[2]) // int(4))
    maxTimeslotHour = minTimeslotHour + 3

    timestamp = "ts[" + str(minTimeslotHour) + "-" + str(maxTimeslotHour) + "]"

    key = (timestamp, stationId)

    if  numFreeSlots < thresholdFreeSlots:
        return (key, (1, 1))
    else:
        return (key, (1, 0))
```

[20]:
```
# The input data are already in readingsRDD

# Count the number of total and critical readings for each (timeslot,stationId)
# Create an RDD of pairs with
# key: (timeslot,stationId)
# value: (numReadings, numCriticalReadings)
# ------- numReadings: 1 for each input line
# --------numCriticalReadings: 0 if the situation is not critical. 1 if it is␣
 ↪critical

timestampStationCountPairRDD = readingsRDD.map(criticalSituationTimeslots)
```

[21]:
```
#timestampStationCountPairRDD.collect()
```

[22]:
```
# Compute the number of total and critical readings for each (timeslot,station)
timestampStationTotalCountPairRDD = timestampStationCountPairRDD \
.reduceByKey(lambda c1, c2: (c1[0]+c2[0], c1[1]+c2[1]) )
```

[23]:
```
#timestampStationTotalCountPairRDD.collect()
```

[24]:
```
# Compute the percentage of critical situations for each (timeslot,station)
timestampStationPercentagePairRDD = timestampStationTotalCountPairRDD\
.mapValues(lambda counters: counters[1]/counters[0])
```

[25]:
```
#timestampStationPercentagePairRDD.collect()
```

[26]:
```
# Select (timeslot,station) pairs with percentage > 80%
selectedTimestampStationsPairRDD = timestampStationPercentagePairRDD\
.filter(lambda sensorPerc: sensorPerc[1]>thresholdCriticalPercentage)
```

[27]:
```
#selectedTimestampStationsPairRDD.collect()
```

[28]:
```
# Sort the stored pairs by decreasing percentage of critical situations
percentageTimestampStationsSortedPairRDD = selectedTimestampStationsPairRDD\
.sortBy(lambda sensorPerc: sensorPerc[1], ascending=False)
```

[30]:
```
#percentageTimestampStationsSortedPairRDD.collect()
```

[31]:
```
percentageTimestampStationsSortedPairRDD.saveAsTextFile(outputPath2)
```

[32]:
```
# Solution Ex. 43 - part III
# Select a reading (i.e., a line) of the first input file if and only if the␣
 ↪following constraints are true
# - The line is associated with a full station situation
# - All the neighbor stations of the station Si are full in the time stamp␣
 ↪associated with the current line
```

[33]:
```
# Read the file containing the list of neighbors for each station
neighborsRDD = sc.textFile(inputPathNeighbors)
```

[34]:
```
# Map each line of the input file to a pair stationid, list of neighbor stations
nPairRDD = neighborsRDD.map(lambda line: (line.split(",")[0], line.
 ↪split(",")[1].split(" ")) )
```

[35]:
```
# nPairRDD.collect()
```

[36]:
```
# Create a local dictionary in the main memory of the driver that will be used␣
 ↪to store the mapping
# stationid -> list of neighbors
# There are only 100 stations. Hence, you can suppose that data about neighbors␣
 ↪can be stored in the main memory
neighbors=nPairRDD.collectAsMap()
```

[37]:
```
# The input data are already in readingsRDD
```

[38]:
```
# Select the lines/readings associated with a full status (number of free slots␣
 ↪equal to 0)
fullStatusLines = readingsRDD.filter(lambda line: int(line.split(",")[5])==0)
```

[39]:
```
def extractTimestamp(reading):
    fields = reading.split(",")
    timestamp = fields[1] + fields[2] + fields[3]
```

```
    return timestamp
```

[40]:
```
# Create an RDD of pairs with key = timestamp and value=reading associated with␣
 ↪that timestamp
# The concatenation of fields[1], fields[2], fields[3] is the timestamp of the␣
 ↪reading
fullLinesPRDD = fullStatusLines.map(lambda reading: (extractTimestamp(reading),␣
 ↪reading))
```

[42]:
```
#fullLinesPRDD.collect()
```

[43]:
```
#  Collapse all the values with the same key in one single pair (timestamp,␣
 ↪reading associated with that timestamp)
fullReadingsPerTimestamp = fullLinesPRDD.groupByKey()
```

[45]:
```
#fullReadingsPerTimestamp.mapValues(lambda v: list(v)).collect()
```

[46]:
```
def selectReadingssFunc(pairTimeStampListReadings):
    # Extract the list of stations that appear in the readings
    # associated with the current key
    # (i.e., the list of stations that are full in this timestamp)
    # The list of readings is in the value part of the inpput key-value pair
    stations = []
    for reading in pairTimeStampListReadings[1]:
        # Extract the stationid from each reading
        fields = reading.split(",")
        stationId = fields[0]
        stations.append(stationId)


    # Iterate again over the list of readings to select the readings satistying␣
 ↪the constraint on the
    # full status situation of all neighboors
    selectedReading = []

    for reading in pairTimeStampListReadings[1]:
        # This reading must be selected if all the neighbors of
        # the station of this reading are also in the value of
        # the current key-value pair (i.e., if they are in list stations)
        # Extract the stationid of this reading
        fields = reading.split(",")
        stationId = fields[0]

        # Select the list of neighbors of the current station
        nCurrentStation = neighbors[stationId]
```

```
        # Check if all the neighbors of the current station are in value
        # (i.e., the local list stations) of the current key-value pair
        allNeighborsFull = True

        for neighborStation in nCurrentStation:
            if neighborStation not in stations:
                # There is at least one neighbor of th current station
                # that is not in the full status in this timestamp
                allNeighborsFull = False

        if allNeighborsFull == True:
            selectedReading.append(reading)

    return selectedReading
```

[47]:
```
# Each pair contains a timestamp and the list of readings (with number of free␣
 ↪slots equal to 0)
# associated with that timestamp.
# Check, for each reading in the list, if all the neighbors of the station of␣
 ↪that reading are
# also present in this list of readings
# Emit one "string" for each reading associated with a completely full status
selectedReadingsRDD = fullReadingsPerTimestamp.flatMap(selectReadingssFunc)
```

[49]:
```
#selectedReadingsRDD.collect()
```

[130]:
```
# Store the result in HDFS
selectedReadingsRDD.saveAsTextFile(outputPath3)
```

[ ]:
```

```

ex44

August 12, 2022

[1]:
```
# Solution Ex. 44
```

[1]:
```
#inputPathWatched = "/data/students/bigdata-01QYD/ex_data/Ex44/data/
 ↪watchedmovies.txt"
#inputPathPreferences = "/data/students/bigdata-01QYD/ex_data/Ex44/data/
 ↪preferences.txt"
#inputPathMovies = "/data/students/bigdata-01QYD/ex_data/Ex44/data/movies.txt"
#outputPath = "res_out_Ex44/"
#threshold = 0.5

inputPathWatched = "data/Ex44/data/watchedmovies.txt"
inputPathPreferences = "data/Ex44/data/preferences.txt"
inputPathMovies = "data/Ex44/data/movies.txt"
outputPath = "res_out_Ex44/"
threshold = 0.5
```

[2]:
```
# Read the content of the watched movies file
watchedRDD = sc.textFile(inputPathWatched)
```

[3]:
```
# Select only userid and movieid
# Define an RDD or pairs with movieid as key and userid as value
movieUserPairRDD = watchedRDD.map(lambda line:  (line.split(",")[1], line.
 ↪split(",")[0]))
```

[4]:
```
# Read the content of the movies file
moviesRDD = sc.textFile(inputPathMovies)
```

[5]:
```
# Select only movieid and genre
# Define an RDD of pairs with movieid as key and genre as value
movieGenrePairRDD = moviesRDD.map(lambda line:  (line.split(",")[0], line.
 ↪split(",")[2]))
```

[6]:
```
# Join watched movies with movies
joinWatchedGenreRDD = movieUserPairRDD.join(movieGenrePairRDD)
```

[7]:
```
# Select only userid (as key) and genre (as value)
```

```
usersWatchedGenresRDD = joinWatchedGenreRDD.map(lambda pair: (pair[1][0],␣
 ↪pair[1][1]))
```

[8]:
```
# Read the content of preferences.txt
preferencesRDD = sc.textFile(inputPathPreferences)
```

[9]:
```
# Define an RDD of pairs with userid as key and genre as value
userLikedGenresRDD = preferencesRDD.map(lambda line: (line.split(",")[0], line.
 ↪split(",")[1]))
```

[10]:
```
# Cogroup the lists of watched and liked genres for each user
# There is one pair for each userid
# the value contains the list of genres (with repetitions) of the
# watched movies and the list of liked genres
userWatchedLikedGenres = usersWatchedGenresRDD.cogroup(userLikedGenresRDD)
```

[13]:
```
#userWatchedLikedGenres.mapValues(lambda v: (list(v[0]), list(v[1]))).collect()
```

[14]:
```
def misleadingProfileFunc(userWatchedLikedGenresLists):
    # Store in a local list the "small" set of liked genres
    # associated with the current user
    likedGenres = list(userWatchedLikedGenresLists[1][1])

    # Iterate over the watched movies (the genres of the watched movies)and␣
 ↪count
    # - The number of watched movies for this user
    # - How many of watched movies are associated with a not liked genre
    numWatchedMovies = 0
    notLiked = 0

    for watchedGenre in userWatchedLikedGenresLists[1][0]:
        numWatchedMovies = numWatchedMovies+1
        if watchedGenre not in likedGenres:
            notLiked = notLiked+1

    # Check if the number of watched movies associated with a non-liked genre
    # is greater that threshold%
    if float(notLiked) > threshold * float(numWatchedMovies):
        return True
    else:
        return False
```

[15]:
```
# Filter the users with a misleading profile
misleadingUsersListsRDD = userWatchedLikedGenres.filter(misleadingProfileFunc)
```

[16]:
```
# Select only the userid of the users with a misleading profile
misleadingUsersRDD = misleadingUsersListsRDD.keys()
```

[18]:
```
#misleadingUsersRDD.collect()
```

[113]:
```
misleadingUsersRDD.saveAsTextFile(outputPath)
```

ex45

August 12, 2022

[25]:
```
# Solution Ex. 45
```

[3]:
```
#inputPathWatched = "/data/students/bigdata-01QYD/ex_data/Ex45/data/
 ↪watchedmovies.txt"
#inputPathPreferences = "/data/students/bigdata-01QYD/ex_data/Ex45/data/
 ↪preferences.txt"
#inputPathMovies = "/data/students/bigdata-01QYD/ex_data/Ex45/data/movies.txt"
#outputPath = "res_out_Ex45/"
#threshold = 0.5

inputPathWatched = "data/Ex45/data/watchedmovies.txt"
inputPathPreferences = "data/Ex45/data/preferences.txt"
inputPathMovies = "data/Ex45/data/movies.txt"
outputPath = "res_out_Ex45/"
threshold = 0.5
```

[4]:
```
# Read the content of the watched movies file
watchedRDD = sc.textFile(inputPathWatched)
```

[5]:
```
# Select only userid and movieid
# Define an RDD or pairs with movieid as key and userid as value
movieUserPairRDD = watchedRDD.map(lambda line:  (line.split(",")[1], line.
 ↪split(",")[0]))
```

[6]:
```
# Read the content of the movies file
moviesRDD = sc.textFile(inputPathMovies)
```

[7]:
```
# Select only movieid and genre
# Define an RDD of pairs with movieid as key and genre as value
movieGenrePairRDD = moviesRDD.map(lambda line:  (line.split(",")[0], line.
 ↪split(",")[2]))
```

[8]:
```
# Join watched movies with movies
joinWatchedGenreRDD = movieUserPairRDD.join(movieGenrePairRDD)
```

[9]:
```
# Select only userid (as key) and genre (as value)
```

```
usersWatchedGenresRDD = joinWatchedGenreRDD.map(lambda pair: (pair[1][0],␣
 ↪pair[1][1]))
```

[10]:
```
# Read the content of preferences.txt
preferencesRDD = sc.textFile(inputPathPreferences)
```

[11]:
```
# Define an RDD of pairs with userid as key and genre as value
userLikedGenresRDD = preferencesRDD.map(lambda line: (line.split(",")[0], line.
 ↪split(",")[1]))
```

[12]:
```
# Cogroup the lists of watched and liked genres for each user
# There is one pair for each userid
# the value contains the list of genres (with repetitions) of the
# watched movies and the list of liked genres
userWatchedLikedGenres = usersWatchedGenresRDD.cogroup(userLikedGenresRDD)
```

[13]:
```
# This function is used in the next transformation to select users with a␣
 ↪misleading profile
def misleadingProfileFunc(userWatchedLikedGenresLists):
    # Store in a local list the "small" set of liked genres
    # associated with the current user
    likedGenres = list(userWatchedLikedGenresLists[1][1])

    # Iterate over the watched movies (the genres of the watched movies)and␣
 ↪count
    # - The number of watched movies for this user
    # - How many of watched movies are associated with a not liked genre
    numWatchedMovies = 0
    notLiked = 0

    for watchedGenre in userWatchedLikedGenresLists[1][0]:
        numWatchedMovies = numWatchedMovies+1
        if watchedGenre not in likedGenres:
            notLiked = notLiked+1

    # Check if the number of watched movies associated with a non-liked genre
    # is greater that threshold%
    if float(notLiked) > threshold * float(numWatchedMovies):
        return True
    else:
        return False
```

[14]:
```
# Filter the users with a misleading profile
misleadingUsersListsRDD = userWatchedLikedGenres.filter(misleadingProfileFunc)
```

[15]:
```
# This function is used in the next transformation to select the pairs␣
 ↪(userid,misleading genre)
def misleadingGenresFunc(userWatchedLikedGenresLists):
    # Store in a local list the "small" set of liked genres
    # associated with the current user

    userId = userWatchedLikedGenresLists[0]
    likedGenres = list(userWatchedLikedGenresLists[1][1])

    # In this solution I suppose that the number of distinct genres for each␣
 ↪user
    # is small and can be stored in a local variable.
    # The local variable is a dictionary that stores for each non-liked genre
    # also its number of occurrences in the list of watched movies of the␣
 ↪current user
    numGenres = {}


    # Iterate over the watched movies (the genres of the watched movies).
    # Select the watched genres that are not in the liked genres and
    # count their number of occurrences. Store them in the numGenres dictionary
    for watchedGenre in userWatchedLikedGenresLists[1][0]:
        # Check if the genre is not in the liked ones
        if watchedGenre not in likedGenres:
            # Update the number of times this genre appears
            # in the list of movies watched by the current user
            if watchedGenre in numGenres:
                numGenres[watchedGenre] = numGenres[watchedGenre] + 1
            else:
                numGenres[watchedGenre] = 1

    # Select the genres, which are not in the liked ones,
    # which occur at least 5 times
    selectedGenres = []

    for genre, occurrences in numGenres.items():
        if occurrences>=5:
            selectedGenres.append( (userId, genre) )

    return selectedGenres
```

[16]:
```
# Select the pairs (userid,misleading genre)
misleadingUserGenrePairRDD = misleadingUsersListsRDD.
 ↪flatMap(misleadingGenresFunc)
```

[18]:
```
#misleadingUserGenrePairRDD.collect()
```

[ ]:
```
misleadingUserGenrePairRDD.saveAsTextFile(outputPath)
```

ex46

August 12, 2022

[1]:
```
# Solution Ex. 46
```

[2]:
```
import sys
```

[3]:
```
inputPath = "data/Ex46/data/readings.txt" # "/data/students/bigdata-01QYD/
 ↪ex_data/Ex46/data/readings.txt"
outputPath = "res_out_Ex46v2/"
```

[4]:
```
# Read the content of the readings
readingsRDD = sc.textFile(inputPath)
```

[9]:
```
readingsRDD.collect()
```

[9]:
```
['1451606400,12.1',
 '1451606460,12.2',
 '1451606520,13.5',
 '1451606580,14.0',
 '1451606640,14.0',
 '1451606700,15.5',
 '1451606760,15.0']
```

[5]:
```
# Generate the elements of each window.
# Each reading with start time t belongs to 3 windows with a window size equal␣
 ↪to 3:
# - The one starting at time t-120s
# - The one starting at time t-60s
# - The one starting at time t

def windowElementsFunc(reading):
    fields = reading.split(",")

    # Time stamp of this reading
    t = int(fields[0])
    # Temperature
    temperature = float(fields[1])

    # The current reading, associated with time stamp t,
```

```
    # is part of the windows starting at time t, t-60s, t-120s

    # pairs is a list containing three pairs (window start timestamp, current reading) associated with
    # the three windows containing this reading
    pairs = []

    # Window starting at time t
    # This reading is the first element of the window starting at time t
    pairs.append((t, reading))

    # Window starting at time t-60
    # This reading is the second element of that window starting at time t-60
    pairs.append((t-60, reading))

    # Window starting at time t-120
    # This reading is the third element of that window starting at time t-120
    pairs.append((t-120, reading))

    return pairs
```

[6]:
```
windowsElementsRDD = readingsRDD.flatMap(windowElementsFunc)
```

[7]:
```
# Use groupByKey to generate one sequence for each time stamp
timestampsWindowsRDD = windowsElementsRDD.groupByKey()
```

[8]:
```
timestampsWindowsRDD.mapValues(lambda v: list(v)).collect()
```

[8]:
```
[(1451606400, ['1451606400,12.1', '1451606460,12.2', '1451606520,13.5']),
 (1451606340, ['1451606400,12.1', '1451606460,12.2']),
 (1451606280, ['1451606400,12.1']),
 (1451606460, ['1451606460,12.2', '1451606520,13.5', '1451606580,14.0']),
 (1451606520, ['1451606520,13.5', '1451606580,14.0', '1451606640,14.0']),
 (1451606580, ['1451606580,14.0', '1451606640,14.0', '1451606700,15.5']),
 (1451606640, ['1451606640,14.0', '1451606700,15.5', '1451606760,15.0']),
 (1451606700, ['1451606700,15.5', '1451606760,15.0']),
 (1451606760, ['1451606760,15.0'])]
```

[10]:
```
# This function is used in the next transformation to select the windows with an incrasing temperature trend
def increasingTrendFunc(pairInitialTimestampWindow):

    # The key of the input pair is the intial timestamp of the current window
    minTimestamp = pairInitialTimestampWindow[0]

    # Store the (at most) 3 elements of the window in a dictionary
    # containing enties time stamp -> temperature
```

```
    timestampTemp = {}

    # pairInitialTimestampWindow[1] contains the elements of the current window
    window = pairInitialTimestampWindow[1]


    for timestampTemperature in window:
        fields = timestampTemperature.split(",")
        t = int(fields[0])
        temperature = float(fields[1])

        timestampTemp[t] = temperature


    # Check if the list contains three elements.
    # If the number of elements is not equal to 3 the window is incomplete and must be discarded
    if len(timestampTemp) != 3:
        increasing = False
    else:
        # Check is the increasing trend is satisfied
        if timestampTemp[minTimestamp]<timestampTemp[minTimestamp+60] and timestampTemp[minTimestamp+60]<timestampTemp[minTimestamp+120]:
            increasing = True
        else:
            increasing = False

    return increasing
```

[11]:
```
seletedWindowsRDD = timestampsWindowsRDD.filter(increasingTrendFunc)
```

[31]:
```
# The result is in the value part of the returned pairs
```

[12]:
```
seletedWindowsRDD.values().map(lambda window: list(window)).collect()
```

[12]:
```
[['1451606400,12.1', '1451606460,12.2', '1451606520,13.5'],
 ['1451606460,12.2', '1451606520,13.5', '1451606580,14.0']]
```

[19]:
```
# Store the result. Map the iterable associated with each window to a list
```

[20]:
```
seletedWindowsRDD.values().map(lambda window: list(window)).
 ↪saveAsTextFile(outputPath)
```

[ ]:
```

```