Spark

Part 4:

Advanced Programming, ML use case: k-means

Dario Colazzo

Credits: Amir H. Payberah

Accumulators and Broadcast variables

Not totally shared-nothing…

- When Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task.
- Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program.
- General read-write shared variables across tasks is inefficient.
- Two types of shared variables: accumulators and broadcast variables.

Accumulators

Aggregating values from worker nodes back to the driver program.

Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Accumulators

- Example: counting events that occur during job execution.
- Worker code can add to the accumulator with its += method.
- The driver program can access the value by calling the value property on the accumulator

```
>>> accum = sc.accumulator(0)
>>> accum
Accumulator<id=0, value=0>

>>> >>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
>>> accum
Accumulator<id=0, value=10>
>>> accum.value
10
```

attention: foreach() is an **action**

Example

- How many lines from a file were blank?

```
>>> t = sc.textFile(“file:/home/dario.colazzo/data/JSON/people.txt",4)
>>> t.collect()
[u'Michael, 29', u'', u'', u'Andy, 30', u'', u'Justin, 19']
>>> blankLines = sc.accumulator(0)
>>> blankLines
Accumulator<id=4, value=0>
>>> def f(x):
...     if x=="" :
...             blankLines.add(1)
...             return []
...     return [y.encode('utf-8') for y in x.split(" ")]
...
>>> words = t.flatMap(f)
>>> blankLines.value
0
>>> words.collect()
['Michael,', '29', 'Andy,', '30', 'Justin,', '19']
>>> blankLines.value
3
```

Broadcast Variables

- Recall that each time Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task.
- The broadcast values are sent to each node only once, rather than shipping a copy of it whenever a task is activated.
- Broadcast variables are cached in main memory, and are read-only
- The task using broadcast variables can access its value with the value property.

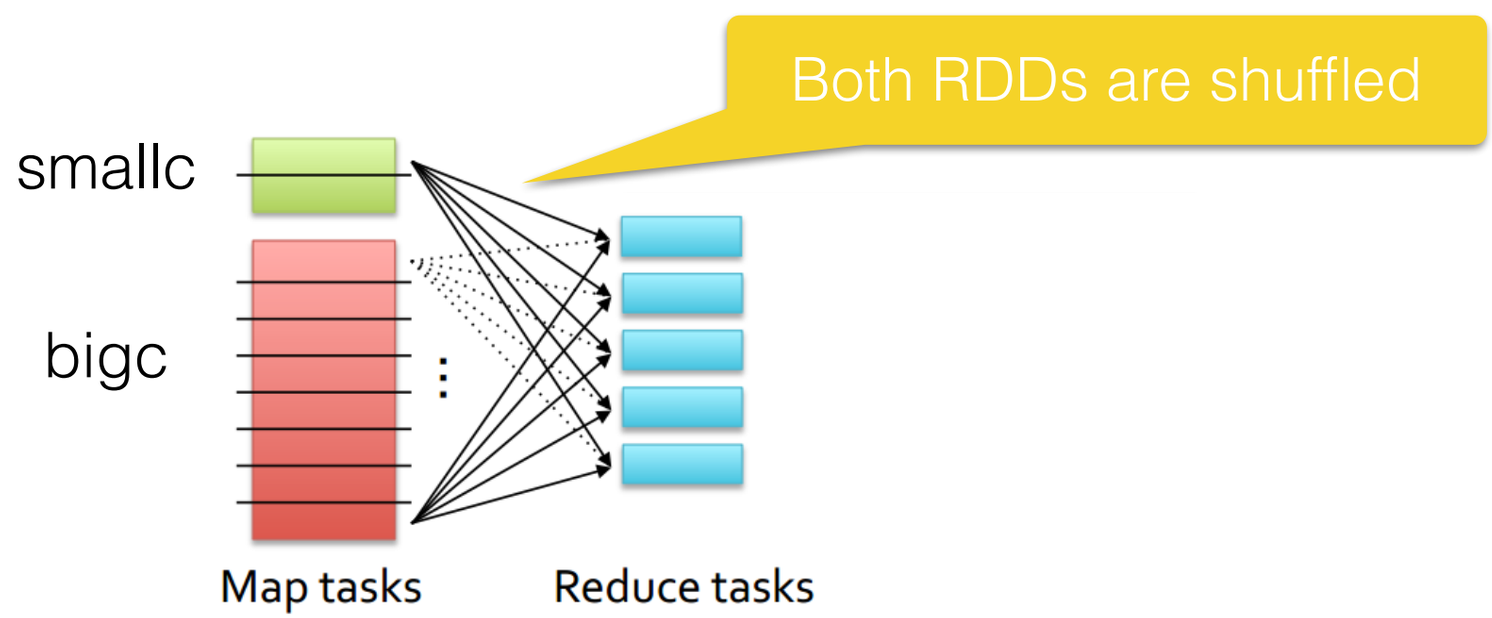
```
>>> rdd = sc.parallelize(range(100))
>>> broadcastVar = sc.broadcast([1, 2, 3])
>>> broadcastVar.value
[1, 2, 3]
>>> rdd.flatMap(lambda y : broadcastVar.value).count()
300
```

Example: *Map joins*

- Let's first see standard join

```
>>>>>> bigCollection =
  [(random.randint(1,3) ,random.randint(1,100) ) for x in range(1000)]

>>> smallCollection=[(1, 'a'),(2, 'b'), (3,'c')]
>>> bigc = sc.parallelize(bigCollection)
>>> smallc = sc.parallelize(smallCollection)
>>>
>>> smallc.join(bigc).take(5)
[(1, ('a', 38)), (1, ('a', 11)), (1, ('a', 67)), (1, ('a', 89)), (1,
('a', 71))]
```



Example: *Map joins*

- Map join : *smallc* can be broadcasted

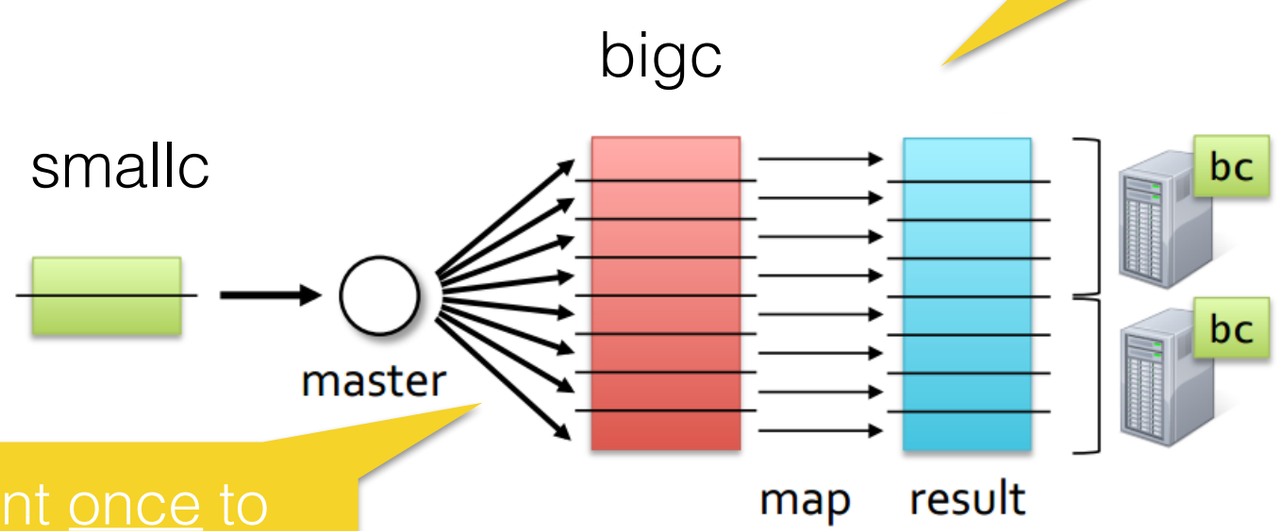
```
>>> smallDict=dict( (x[0], x[1]) for x in smallc.collect() )
>>> smallDict[1]
'a'
>>> bc=sc.broadcast(smallDict)
>>> mapJoined = bigc.map(lambda x : (x[0], (bc.value[x[0]], x[1])))
>>> mapJoined.take(5)
[(3, ('c', 60)), (2, ('b', 26)), (3, ('c', 96)), (3, ('c', 29)), (2,
('b', 63))]
>>>bigc.take(5)
[(3, 60), (2, 26), (3, 96), (3, 29), (2, 63)]
>>>
```

Example: *Map joins*

- Map join : smallc can be broadcasted

```
>>> smallDict=dict( (x[0], x[1]) for x in smallc.collect() )
>>> smallDict[1]
'a'
>>> bc=sc.broadcast(smallDict)
>>> mapJoined = bigc.map(lambda x : (x[0], (bc.value[x[0]], x[1])))
>>> mapJoined.take(5)
[(3, ('c', 60)), (2, ('b', 26)), (3, ('c', 96)), (3, ('c', 29)), (2,
('b', 63))]
>>>bigc.take(5)
[(3, 60), (2, 26), (3, 96), (3, 29), (2, 63)]
>>>
```

No shuffle!



smallc is sent <u>once</u> to Workers

Conclusion on Spark

- Spark enables dataflow programming
- RDD as a new type added to existing Programming Languages
- RDD distribution in main memory
- Two types of operations : transformations and actions
- Execution engine based on stages, task and lineage
- Spark SQL: Dataframes, also used in ML
- Counters and broadcast variables

K-means in spark

Thanks to Paul-Henri PERRIN, PhD student at Data Science team @ Dauphine

Basic notions

- In French! Borrowed from nice notes of Francis Back nice notes at ENS (https://www.di.ens.fr/~fbach/courses/fall2010/cours3.pdf)

K-means est un algorithme de quantification vectorielle (clustering en anglais). K-means est un algorithme de minimisation alternée qui, étant donné un entier K, va chercher à séparer un ensemble de points en K clusters (Figure 3.1).

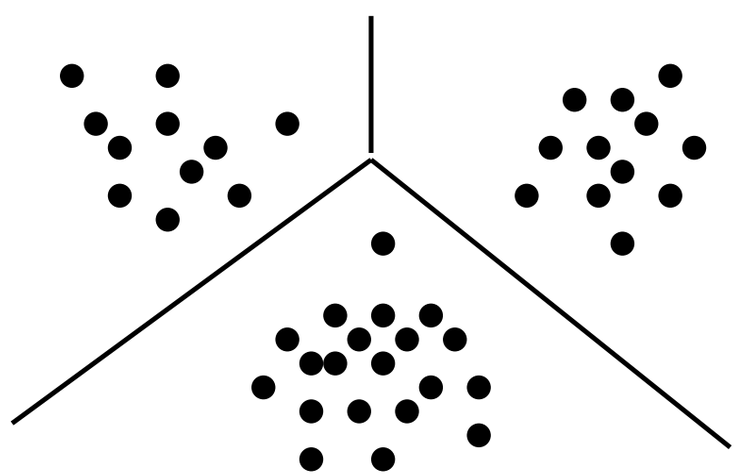


Fig. 3.1. Clustering sur un ensemble de points 2D, 3 clusters.

Notations

On utilise les notations suivantes :

- Les $x_i \in \mathbb{R}^p$, $i \in \{1, ..., n\}$ sont les points à séparer.
- Les z_i^k sont des variables indicatrices associées aux x_i telles que $z_i^k = 1$ si x_i appartient au cluster k, $z_i^k = 0$ sinon. z est la matrice des z_i^k.
- μ est le vecteur des $\mu_k \in \mathbb{R}^p$, où μ_k est le centre du cluster k.

On définit de plus *la mesure de distorsion* $J(\mu, z)$ par :

$$J(\mu, z) = \sum_{i=1}^{n} \sum_{k=1}^{n} z_i^k \|x_i - \mu_k\|^2$$

L'algorithme

Le but de l'algorithme est de minimiser $J(\mu, z)$, il se présente sous la forme d'un algorithme de minimisation alternée :

- Etape 0 : "choisir le vecteur μ"
- Etape 1 : on minimise J par rapport à z : $z_i^k = 1$ pour $k \in \arg\min \|x_i - \mu_k\|$, *ie* on associe à x_i le centre μ_k le plus proche.
- Etape 2 : on minimise J par rapport à μ : $\mu_k = \frac{\sum_i z_i^k x_i}{\sum_i z_i^k}$.
- Etape 3 : retour à l'étape 1 jusqu'à convergence.

Data

- We will use a simple (classical) data set describing features of flowers (available at https://www.dropbox.com/s/9kits2euwawcsj0/iris.data.txt)

Le jeu de données [modifier | modifier le code]

Fisher's *Iris* Data

longueur des sépales (en cm) *(Sepal length)*	largeur des sépales (en cm) *(Sepal width)*	longueur des pétales (en cm) *(Petal length)*	largeur des pétales (en cm) *(Petal width)*	Espèce *(Species)*
5.1	3.5	1.4	0.2	*I. setosa*
4.9	3.0	1.4	0.2	*I. setosa*
4.7	3.2	1.3	0.2	*I. setosa*
4.6	3.1	1.5	0.2	*I. setosa*
5.0	3.6	1.4	0.2	*I. setosa*
5.4	3.9	1.7	0.4	*I. setosa*
4.6	3.4	1.4	0.3	*I. setosa*
5.0	3.4	1.5	0.2	*I. setosa*

source : https://fr.wikipedia.org/wiki/Iris_(jeu_de_données)

How do we proceed

- I will give you the code soon
- To launch a Python program from the master node of the cluster.

```
> spark-submit    --executor-memory 7g \
                  --num-executors 10  \
                  --executor-cores 4 \
                  kmeans-dario.py
```

- Step-by-step presentation of the code
- Then you can launch the job and eventually modify it.

Nice Cloudera documentation on how to tune your submit command:
http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

K-means clustering

Initialising variables and RDDs

```
data = lines.map(lambda x: x.split(','))\
             .map(lambda x: [float(i) for i in x[:4]] + [x[4]])\
             .zipWithIndex()\
             .map(lambda x: (x[1], x[0]))

# zipWithIndex allows us to give a specific index to each point
# (0, [5.1, 3.5, 1.4, 0.2, 'Iris-setosa'])

clusteringDone = False

number_of_steps = 0
```

K-means clustering

Initialising the centroids

In the same manner, zipWithIndex gives an id to each cluster

```
centroides =
sc.parallelize(data.takeSample('withoutReplacment',nb_clusters
))\
.zipWithIndex()\
.map(lambda x: (x[1],x[0][1][:-1]))

# (0, [4.4, 3.0, 1.3, 0.2])
```

K-means clustering

Points repartition

```
joined = data.cartesian(centroides)
# ((0, [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']), (0, [4.4, 3.0, 1.3,
0.2]))

# We compute the distance between the points and each cluster

dist = joined.map(lambda x: (x[0][0],(x[1][0], computeDistance(x[0][1][:-1], x[1]
[1]))))

def computeDistance(x,y):
    return sqrt(sum([(a - b)**2 for a,b in zip(x,y)]))

# (0, (0, 0.866025403784438))


dist_list = dist.groupByKey().mapValues(list)

# (0, [(0, 0.866025403784438), (1, 3.7), (2, 0.5385164807134504)])
```

K-means clustering

```
def closestCluster(dist_list):
    cluster = dist_list[0][0]
    min_dist = dist_list[0][1]
    for elem in dist_list:
        if elem[1] < min_dist:
            cluster = elem[0]
            min_dist = elem[1]
    return (cluster,min_dist)
```

Points repartition

```
# We keep only the closest cluster to each point.
min_dist = dist_list.mapValues(closestCluster)

# (0, [(0, 0.866), (1, 3.7), (2, 0.538)])
# (0, (2, 0.538))
```

```
# assignment will be our return value : It contains the datapoint,
# the id of the closest cluster and the distance of the point to the centroid

assignment = min_dist.join(data)

# (0, ((2, 0.538), [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']))
```

K-means clustering

New centroids

```
# (0, ((2, 0.538), [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']))

clusters = assignment.map(lambda x: (x[1][0][0], x[1][1]
[:-1]))

# (2, [5.1, 3.5, 1.4, 0.2])

count = clusters.map(lambda x: (x[0],1)).
\reduceByKey(lambda x,y: x+y)

somme = clusters.reduceByKey(sumList)


newCentroides = somme.join(count).map(lambda x :
(x[0],moyenneList(x[1][0],x[1][1])))

def moyenneList(x,n):
    return [x[i]/n for i in range(len(x))]
```

K-means clustering

End Condition

Based on counting the number of point moves

```
if number_of_steps > 0:
    switch = prev_assignment.join(min_dist)\
                            .filter(lambda x: x[1][0][0] != x[1][1][0])\
                            .count()
    else:
         switch = 150
```

K-means clustering

End Condition

```
if switch == 0 or number_of_steps == 100:
    clusteringDone = True
    error = sqrt(min_dist.map(lambda x: x[1][1]).\
                 reduce(lambda x,y: x + y))/nb_elem.value

else:
    centroides = centroidesCluster
    prev_assignment = min_dist
    number_of_steps += 1
```

The whole program

Available at https://www.dropbox.com/s/k5cn7otu5q9ck4c/kmeans-dario.py

ATTENTION : you need to

- load the iris data on your HDFS home

  https://www.dropbox.com/s/jj8h4hypq03r89x/iris.data.txt

- change the program by indicating where to load the data and to store the result - this is left as an exercice

- run the program with a command of the kind (of course you need to change the path/filename for the program - `last line`)

```
> spark-submit    --executor-memory 7g \
                  --num-executors 10  \
                  --executor-cores 4 \
                  kmeans-dario.py
```

Work on the program

- Study the whole code
- Identify weaknesses of the way the algorithm is implemented and suggest possible improvements
- Time permitting implement and test your modifications.