

Management and Analysis of Physics Dataset (mod. B)

Project:

Implementation of the k-means clustering using Spark

Authors:

Fella Eugenio, Pedrazzi Matteo,
Ricucci Gaetano, Sgorlon Gaiatto Carlo

What is k -means clustering

K-means clustering is an unsupervised data processing algorithm that aims to partition n objects, represented by d -dimensional real vectors, into k cluster minimizing a **distance-based cost function**:

$$\phi_X(C) = \sum_{x \in X} \min_{i=1, \dots, k} \|x - c_i\|^2$$

where X is the set of objects and C is the set of k cluster centroids.

How *k*-means clustering works

K-means clustering consists of two stages:

- **Initialization stage**

Select k random centroids (method called naive or Forgy) or use some more advanced initialization algorithm like the k-means++ or k-means||.

- **Iterative stage (Lloyd's algorithm)**

Repeatedly assign each object to the closest cluster and update the centroids until a stopping criteria is met.

Why k-means++ initialization

```
1:  $\mathcal{C} \leftarrow$  sample a point uniformly at random from  $X$ 
2: while  $|\mathcal{C}| < k$  do
3:   Sample  $x \in X$  with probability  $\frac{d^2(x, \mathcal{C})}{\phi_X(\mathcal{C})}$ 
4:    $\mathcal{C} \leftarrow \mathcal{C} \cup \{x\}$ 
5: end while
```

The idea is that for evenly spaced clusters, during the selection of a new centroid, preference should be given to those further away from the previously selected ones.

The main downside is that it is sequential and it needs to pass over the data k times to produce the required k initial centroids. **How can it be improved?**

Why k-means|| initialization

The k-means|| initialization is a parallel version of the k-means++ in the sense that it select more than one centroids at each iteration obtaining nonetheless a nearly optimal solution.

- 1: $\mathcal{C} \leftarrow$ sample a point uniformly at random from X
- 2: $\psi \leftarrow \phi_X(\mathcal{C})$
- 3: **for** $O(\log \psi)$ times **do**
- 4: $\mathcal{C}' \leftarrow$ sample each point $x \in X$ independently with probability $p_x = \frac{\ell \cdot d^2(x, \mathcal{C})}{\phi_X(\mathcal{C})}$
- 5: $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
- 6: **end for**
- 7: For $x \in \mathcal{C}$, set w_x to be the number of points in X closer to x than any other point in \mathcal{C}
- 8: Recluster the weighted points in \mathcal{C} into k clusters

Objectives

- Implement the k-means clustering algorithm efficiently using Spark and test it on a real world dataset.
- Compare the performances of three different initialization algorithms (naive, k-means++ and k-means||) analyzing the cost function and the execution time.
- Study the performances of Spark by varying the configuration parameters (number of executors and number of partitions).

Dataset

The dataset used to test the k-means clustering is the RCV1 (Reuters Corpus Volume I), a collection of over 800,000 manually categorized documents. It can be fetched from [scikit-learn](#) as a dictionary-like object, with the following attributes:

- **data**: a scipy CSR sparse matrix, with 804414 samples and 47236 features represented by cosine-normalized, log TF-IDF vectors.
- **target**: it is a scipy CSR sparse matrix, with 804414 samples and 103 categories represented by vectors with a value of 1 in its categories, and 0 in others.
- **target_names**: array with the names of the categories (each document can belong to more than one category).

Dataset: attributes

name:	class:	shape:
data	<class 'scipy.sparse._csr.csr_matrix'>	(804414, 47236)
target	<class 'scipy.sparse._csr.csr_matrix'>	(804414, 103)
target_names	<class 'numpy.ndarray'>	(103,)
sample_id	<class 'numpy.ndarray'>	(804414,)

name:	class:	shape:	size [Mb]:
data	<class 'numpy.ndarray'>	(60915113,)	487.3209
indices	<class 'numpy.ndarray'>	(60915113,)	243.6605
indptr	<class 'numpy.ndarray'>	(804415,)	3.2177

name:	class:	shape:	size [Mb]:
data	<class 'numpy.ndarray'>	(2606875,)	2.6069
indices	<class 'numpy.ndarray'>	(2606875,)	10.4275
indptr	<class 'numpy.ndarray'>	(804415,)	3.2177

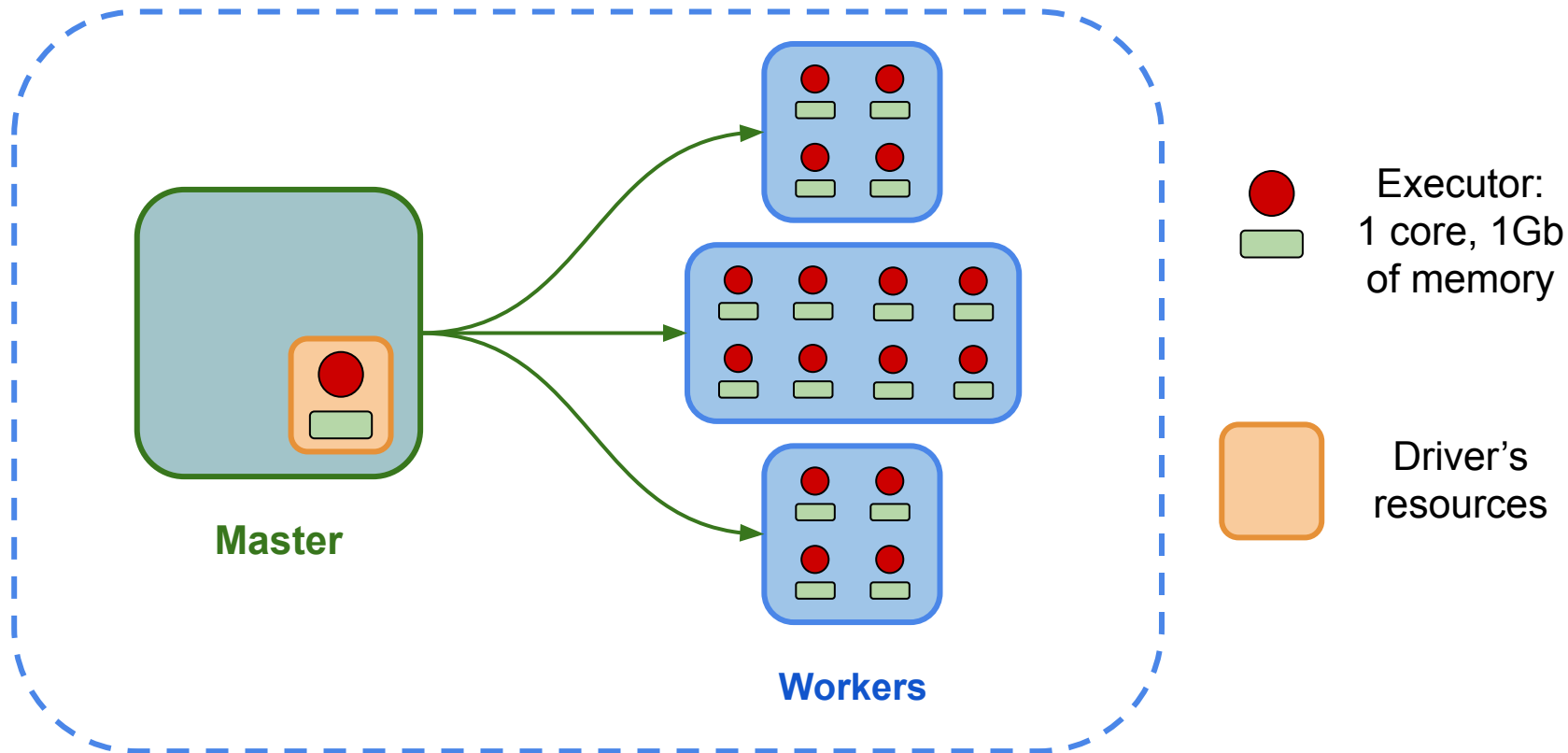
Spark session

```
spark = SparkSession.builder \  
    .master('spark://master:7077')\  
    .appName('k_means_application')\  
    .config('spark.driver.host', '10.67.22.104')\  
    .config('spark.executor.memory', '1g')\  
    .config('spark.executor.cores', '1')\  
    .config('spark.default.parallelism', '32')\  
    .getOrCreate()
```

Resources allocated to each executor.


Indicates the default number of partitions in RDDs returned by transformations.

Spark cluster



Spark Master UI

User interface to monitor the status and the resource consumption of the Spark cluster.

 3.3.0 **Spark Master at spark://10.67.22.104:7077** → master IP address

URL: spark://10.67.22.104:7077
Alive Workers: 3
Cores in use: 16 Total, 16 Used
Memory in use: 28.2 GiB Total, 16.0 GiB Used
Resources in use:
Applications: 1 [Running](#), 100 [Completed](#)
Drivers: 0 Running, 0 Completed
Status: ALIVE

▼ **Workers (3)**

Worker Id	Address	State	Cores	Memory
worker-20220711092004-10.67.22.163-37271	10.67.22.163:37271	ALIVE	4 (4 Used)	6.8 GiB (4.0 GiB Used)
worker-20220711092004-10.67.22.211-42331	10.67.22.211:42331	ALIVE	4 (4 Used)	6.8 GiB (4.0 GiB Used)
worker-20220711092005-10.67.22.153-45261	10.67.22.153:45261	ALIVE	8 (8 Used)	14.6 GiB (8.0 GiB Used)

▼ **Running Applications (1)**

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time
app-20220713134619-0100 (kill)	k_means_application	16	1024.0 MiB		2022/07/13 13:46:19

Annotations:

- application informations (points to the Worker Id column)
- workers IP addresses (points to the Address column)

Parse files in parallel with Spark

Construct a list of file names and parallelize it making an RDD.

```
file_list = [('data_{}.npz'.format(i), 'target_{}.npz'.format(i))  
             for i in range(n_chunk)]  
files_rdd = sc.parallelize(file_list)
```

Apply a flatMap transformation using the 'parse_file' UDF and persist it on memory.

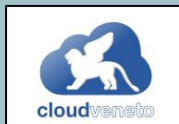
```
data_rdd = files_rdd.flatMap(lambda file: parse_file(file, target_names)).persist()
```

In the above stage we convert the sparse matrices into pairs of labels and sparse dictionaries selecting only documents belonging to 1 sub-label, as k-means clustering assigns each point to a single cluster. We have considered all sub-labels and not only the macro-labels, in order to have a larger number of clusters necessary to properly test the different execution times of the k-means++ and k-means|| initialization methods.

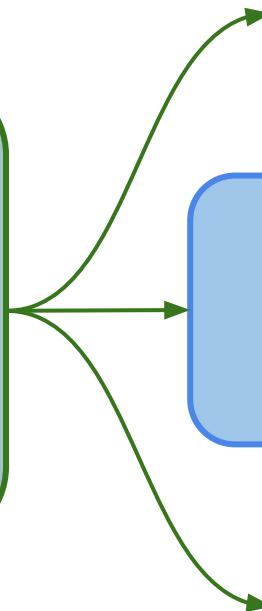
Upload and distribution of files using S3

- Create files e.g.: a.npz, b.npz, c.npz, d.npz
- Upload files on Cloud Veneto

```
s3.upload_file('a.npz', 'bucket')  
s3.upload_file('b.npz', 'bucket')  
s3.upload_file('c.npz', 'bucket')  
s3.upload_file('d.npz', 'bucket')
```



- Create and parallelize list of files names



```
s3.download_file('bucket', 'a.npz')
```



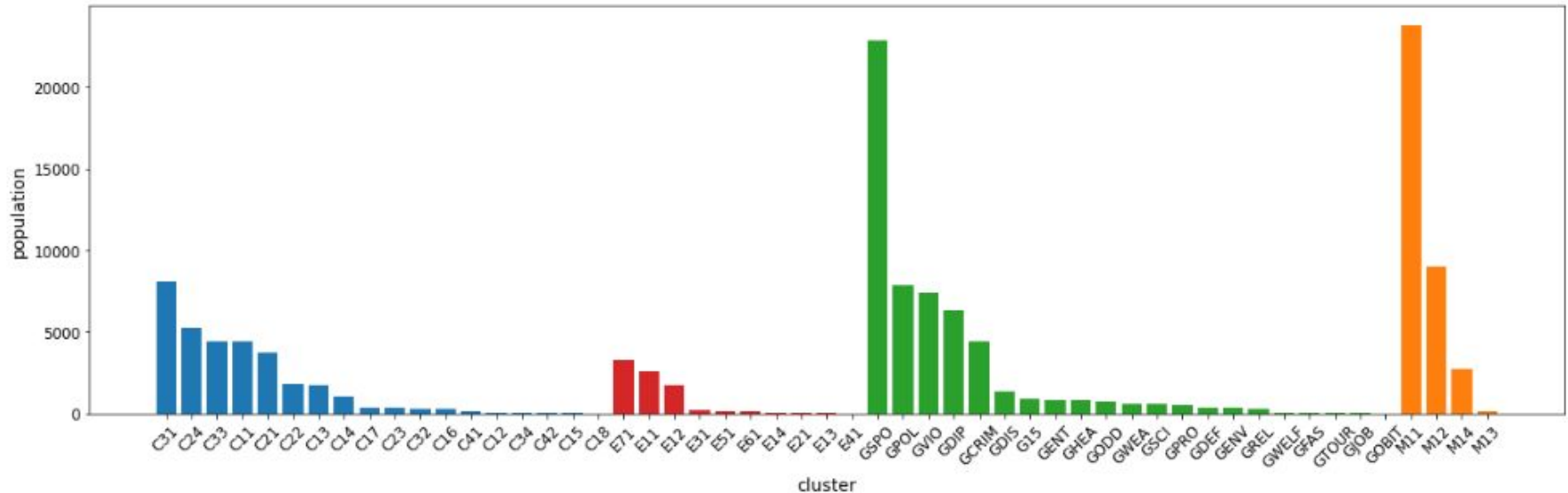
```
s3.download_file('bucket', 'b.npz')  
s3.download_file('bucket', 'c.npz')
```



```
s3.download_file('bucket', 'd.npz')
```

Clusters

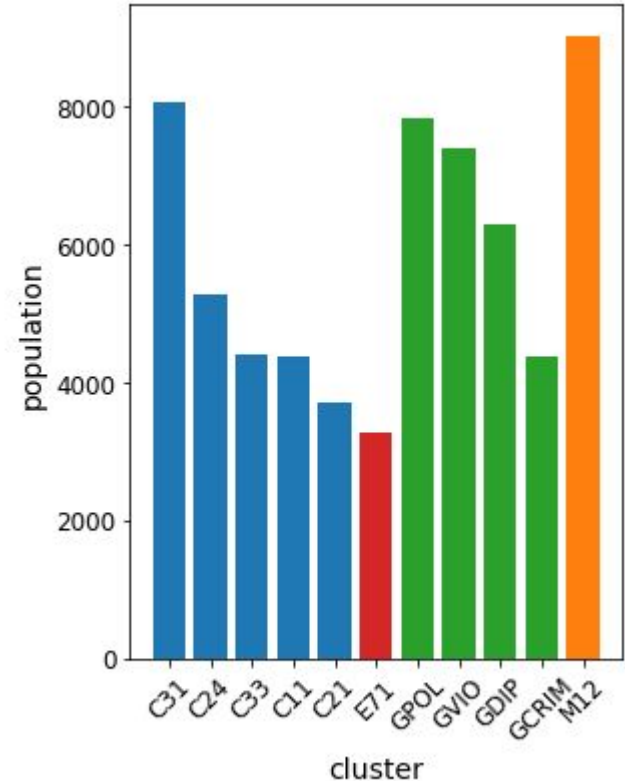
In this graph it is possible to see the population of each cluster grouped by macro-labels and sorted by values.



Cluster filter

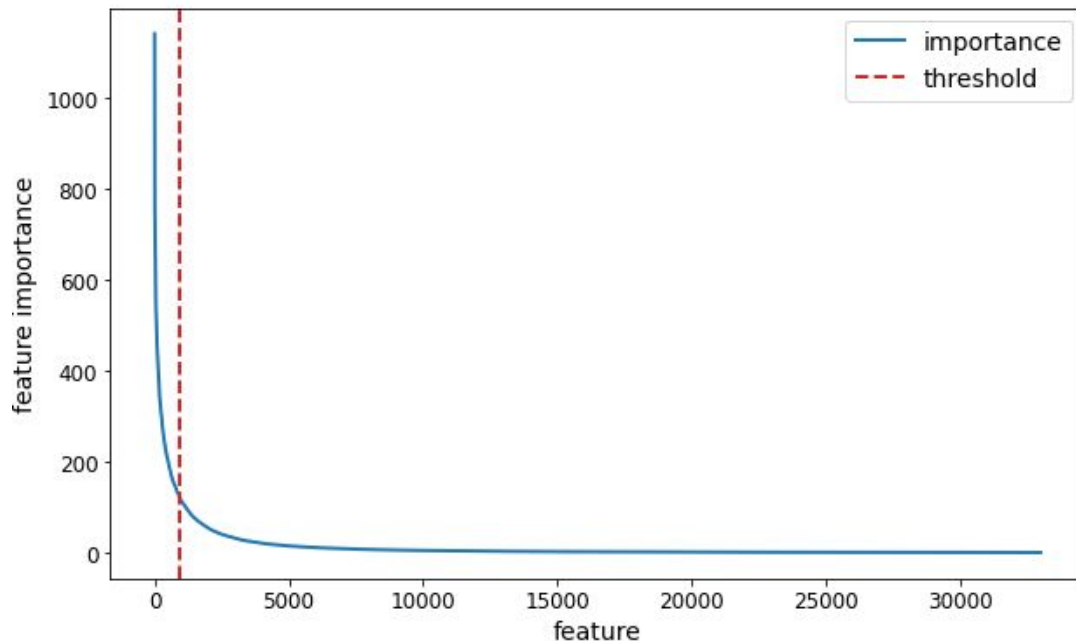
The k-means clustering requires clusters to be of similar size in order to work properly (otherwise it gives too much weight to big clusters).

Therefore we only select clusters with nearly homogeneous populations by setting an upper and a lower threshold. As it can be seen from the graph on the right side, we are left with $k = 11$ clusters.



Features filter

It is possible to reduce the dimensionality of the points by aggregating the features using the sum operation and selecting only the most important ones.



Naive (Forgy) initialization

Select k centroids uniformly at random at once and return a list of k key-value pairs, where the key is the label (that is going to be discarded) and the value is a parse dictionary describing the centroid's features.

```
C = data_rdd.takeSample(False, k, seed)
C = [c[1] for c in C]
```



sample without replacement



fixed seed to allow reproducibility

K-means++ initialization

Select first centroid uniformly at random

```
c = data_rdd.takeSample(False, 1, seed)
C.append(c[0][1])
```

Select the remaining $k - 1$ centroids:

- persist the dataset in memory after computing the squared distance

```
data_rdd_cached = data_rdd \
    .map(lambda el: (compute_d2(el[1], C), el[1])) \
    .persist()
```

- compute cost and trigger the caching

```
cost = data_rdd_cached \
    .reduce(lambda x, y: (x[0] + y[0], ''))[0]
```

K-means++ initialization

- perform a weighted sampling


```
c = data_rdd_cached \
    .map(lambda el: (True, el[1]) if np.random.uniform(size = 1) < el[0]/cost
                  else (False, el[1])) \
    .filter(lambda el: el[0]) \
    .takeSample(False, 1, seed)
```

- free up memory

```
data_rdd_cached.unpersist()
```

Storage

▼ RDDs



ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached
6	PythonRDD	Memory Serialized 1x Replicated	32	100%
25	PythonRDD	Memory Serialized 1x Replicated	32	100%

K-means|| initialization

Select first centroid uniformly at random

```
c = data_rdd.takeSample(False, 1, seed)
C.append(c[0][1])
```

Select the remaining $k - 1$ centroids:

- persist the dataset in memory after computing the squared distance

```
data_rdd_cached = data_rdd \
    .map(lambda el: (compute_d2(el[1], C), el[1])) \
    .persist()
```

- compute cost and trigger the caching

```
cost = data_rdd_cached \
    .reduce(lambda x, y: (x[0] + y[0], ''))[0]
```

K-means|| initialization

- compute normalized probabilities and multiply by the oversampling factor, then perform weighted sampling and get new centroids

```
rows = data_rdd_cached \
    .map(lambda el: (True, el[1]) if np.random.uniform(size = 1) < l*el[0]/cost
                  else (False, el[1])) \
    .filter(lambda el: el[0]) \
    .collect()

c_list = [row[1] for row in rows]
C += c_list
```

- free up memory

```
data_rdd_cached.unpersist()
```

- if the number of selected centroids is higher than k , reduce it using k-means++

Lloyd's algorithm

Compute initialization cost

```
cost = data_rdd \
    .map(lambda el: compute_d2(el[1], C)) \
    .reduce(lambda x, y: x + y)
```

Iterate t times:

- persist the dataset in memory after cluster assignment

```
data_rdd_cached = data_rdd \
    .map(lambda el: assign_cluster(el[1], C)) \
    .persist()
```

- compute clusters population and trigger caching

```
pop_clusters = data_rdd_cached.countByKey()
```

Lloyd's algorithm

- compute new centroids

```
C = data_rdd_cached \  
    .reduceByKey(lambda el1, el2: sum_points(el1, el2)) \  
    .map(lambda el: compute_centroids(el, pop_clusters)) \  
    .values() \  
    .collect()
```

- compute new cost

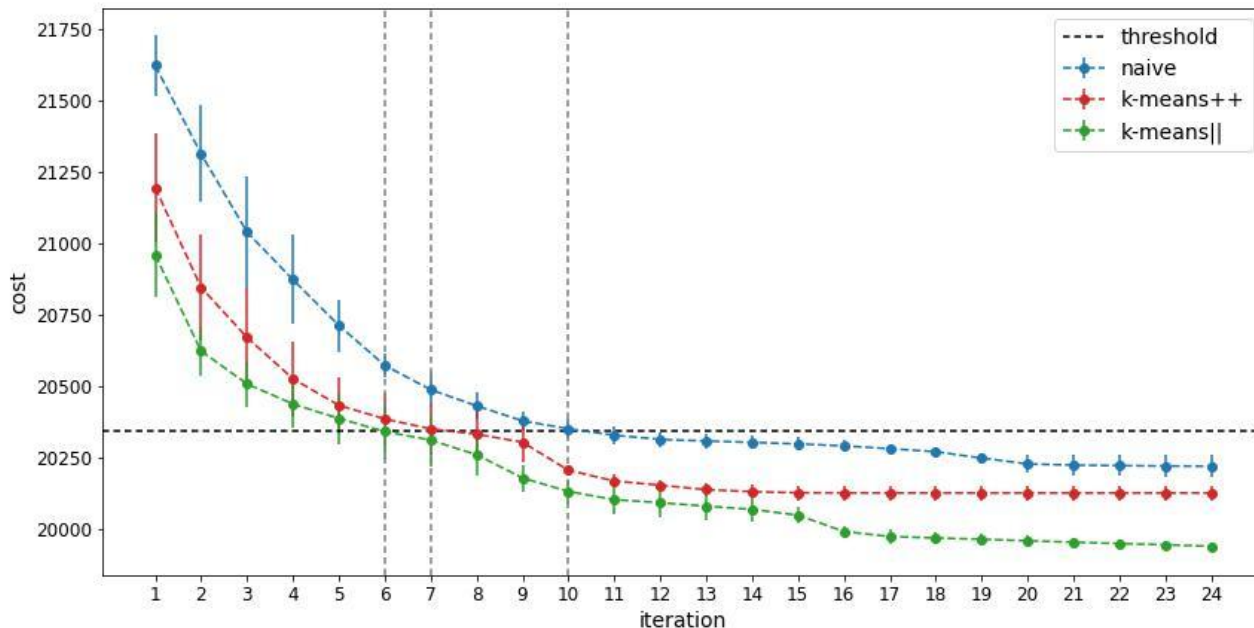
```
cost_best = data_rdd \  
    .map(lambda el: compute_d2(el[1], C)) \  
    .reduce(lambda x, y: x + y)
```

- free up memory

```
data_rdd_cached.unpersist()
```

Performances with different initializations: cost function

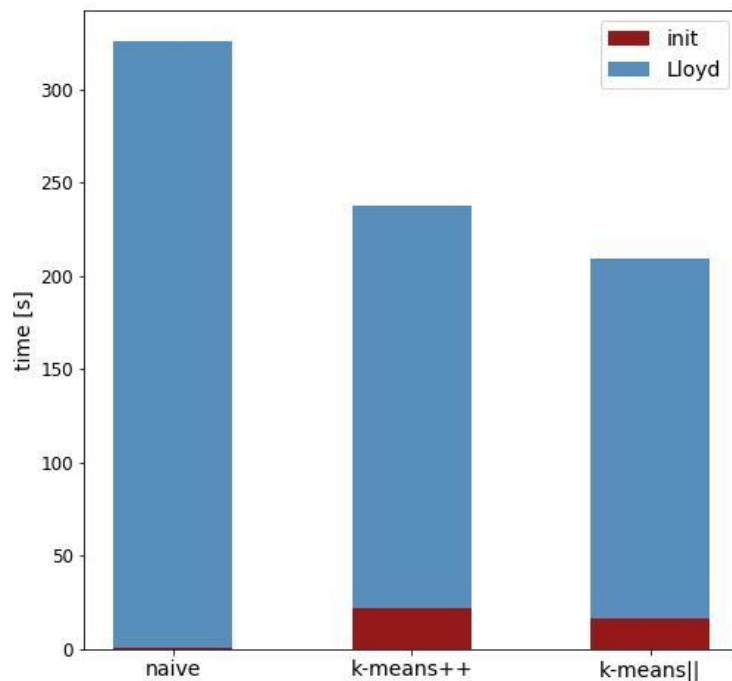
- Dataset: 32 chunks, 16000 documents each
- Number of partitions: 32
- Number of executors: 16, each with 1 core and 1 Gb of memory



Performances with different initializations: execution time

As can be seen from the graph on the right, k-means|| initialization is on average faster than k-means++ initialization.

In addition, both reach a fixed threshold with fewer Lloyd iterations than the naive initialization method.



Performances with different Spark configurations

- Dataset: 128000 documents
- Initialization method: k-means|| with $l = 3$ and $t = 5$
- Number of iterations of Lloyd's algorithm: 5

