Implementation of the Parallel-k-means clustering using Spark

Aksh Patel (202211037)
Love Barot (202211067)

Introduction

k-means clustering

K-means clustering is an unsupervised algorithm for partitioning n objects with d-dimensional real vectors into k clusters, aiming to minimize a **distance-based cost function**.

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

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

k-means clustering

K-means clustering involves two key stages:

1. **Initialization Stage:**

Randomly select k centroids using methods such as naive or Forgy, or employ advanced techniques like k-means++ or k-means|| for initialization.

2. **Iterative Stage (Lloyd's algorithm):**

Continuously assign each object to the nearest cluster and update centroids until a predefined stopping criterion is satisfied.

Objective

Efficient Implementation in Spark:

- Implement the k-means clustering algorithm in Spark and assess its efficiency using a real-world dataset.

Initialization Algorithm Comparison:

- Compare the performance of three initialization algorithms (naive, k-means++, k-means||) by analyzing both the cost function and execution time.

Spark Performance Study:

- Investigate the impact on Spark's performance by varying configuration parameters, specifically the number of executors and partitions.

Analysis and Validation:

- Conduct a thorough analysis of the k-means|| algorithm to validate its theoretical guarantees.
- Highlight distinctions and advantages compared to existing parallel k-means algorithms.

Integration with Parallel Computing:

- Ensure compatibility with parallel computing models, especially MapReduce.
- Utilize primitive operations for broad applicability and demonstrate significant speedup.

Empirical Evaluation:

- Design experiments to evaluate the performance of k-means|| on real-world datasets.
- Compare the results with traditional k-means++ in both sequential and parallel settings.

Problem
& Solution

Problem with 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**

K-means++ initialization prioritizes evenly spaced centroids by favoring locations distant from the previously chosen ones.

Despite its effectiveness, the sequential process involves multiple passes over the data (k- times). How can we enhance its efficiency?

Image taken from Scalable K-means++ **[2]**

Why k-means|| Initialization?

The k-means|| initialization serves as a parallel counterpart to k-means++, selecting multiple centroids in each iteration while still achieving 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: $\quad \mathcal{C}' \leftarrow$ sample each point $x \in X$ independently with probability $p_x = \frac{\ell \cdot d^2(x, \mathcal{C})}{\phi_X(\mathcal{C})}$
5: $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
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

Image taken from Scalable K-means++ **[2]**

Methodology

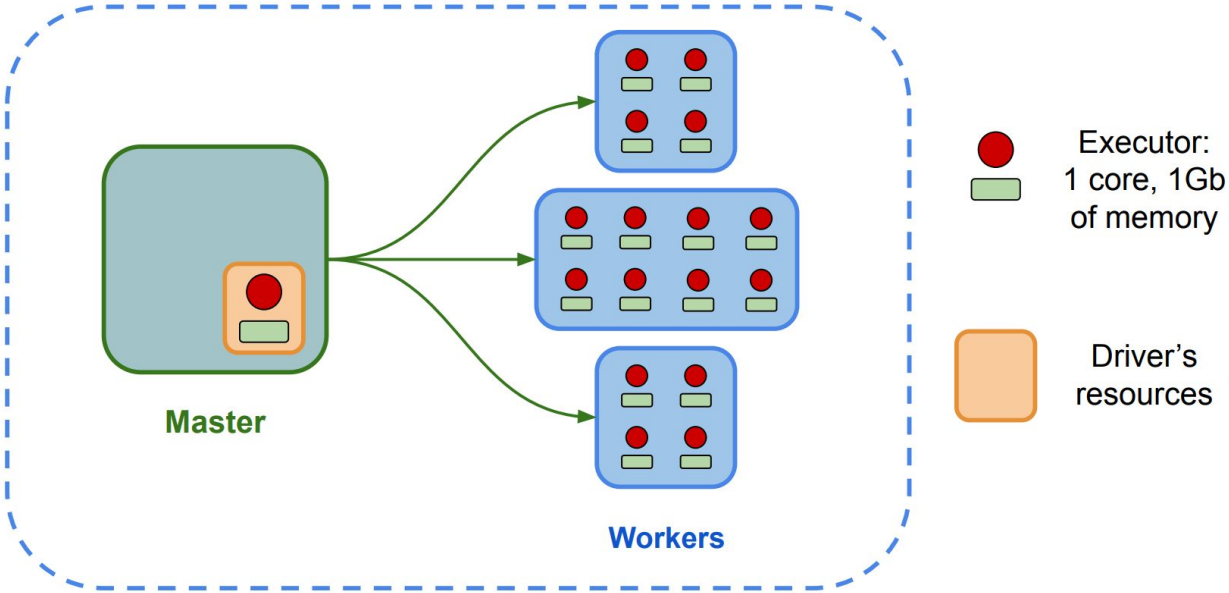
Spark Session

```
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()
```

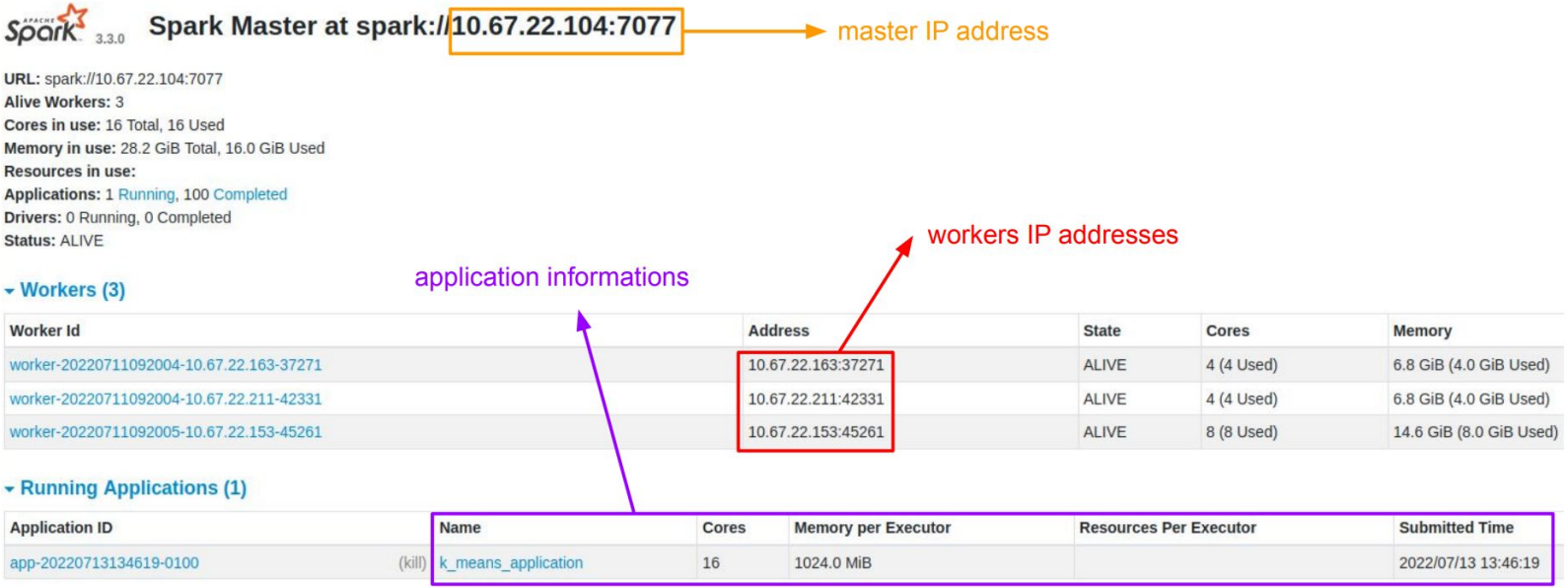
Resources allocated to each executor.

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

Spark Cluster



Spark Master UI



[monitor the status and the resource consumption of the Spark cluster]

Parse files in parallel with Spark

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

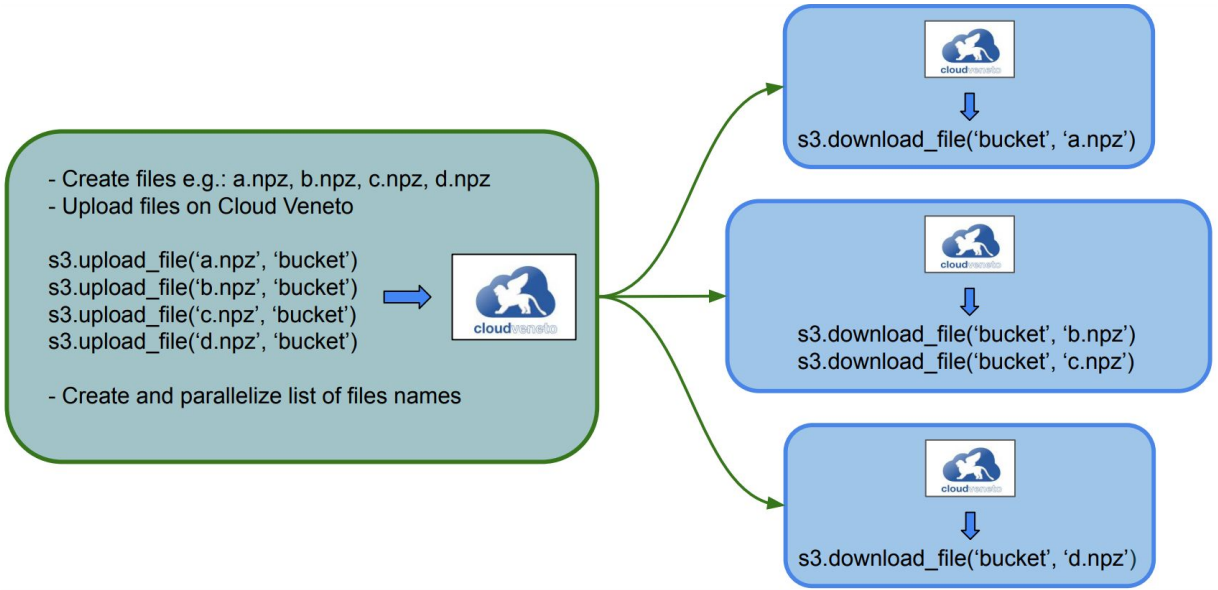
```
file_list = [('data_{}.npz'.format(i), 'target_{}.npz'.format(i))
             for i in range(n_chunk)]
files_rdd = sc.parallelize(file_list)
```

- 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()
```

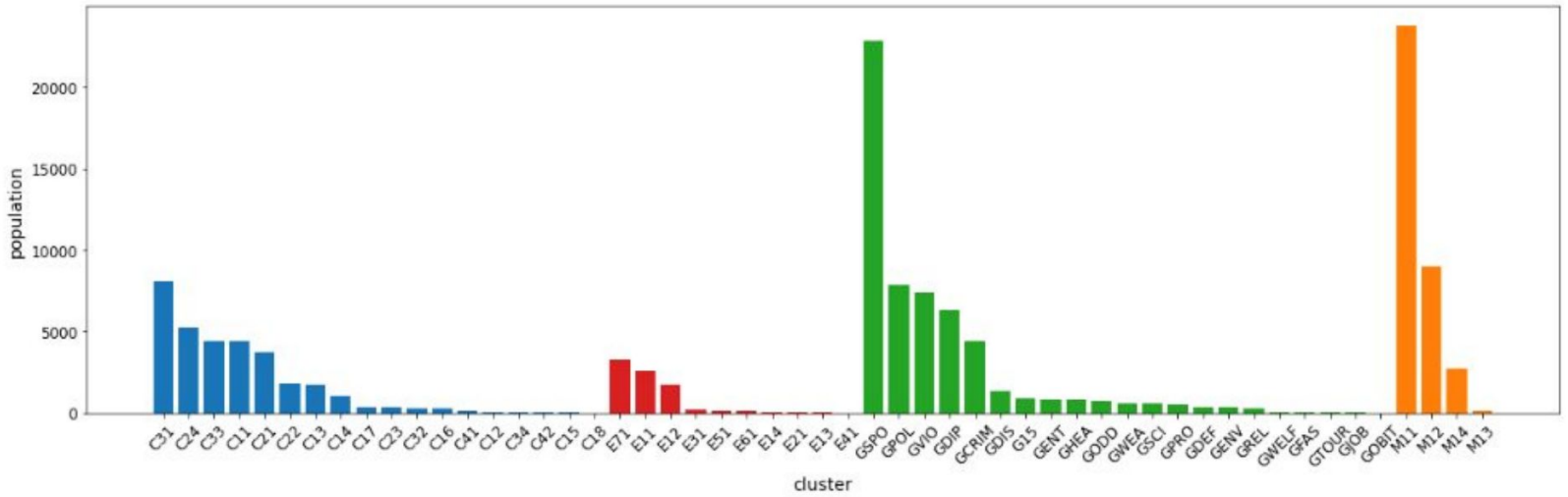
- In this stage, we convert sparse matrices into pairs of labels and sparse dictionaries, focusing on documents associated with individual sub-labels. This choice, encompassing all sub-labels rather than just macro-labels, ensures an ample number of clusters for a robust evaluation of k-means++ and k-means|| initialization methods, specifically in terms of execution times.

Upload and distribution of files using S3



Clusters

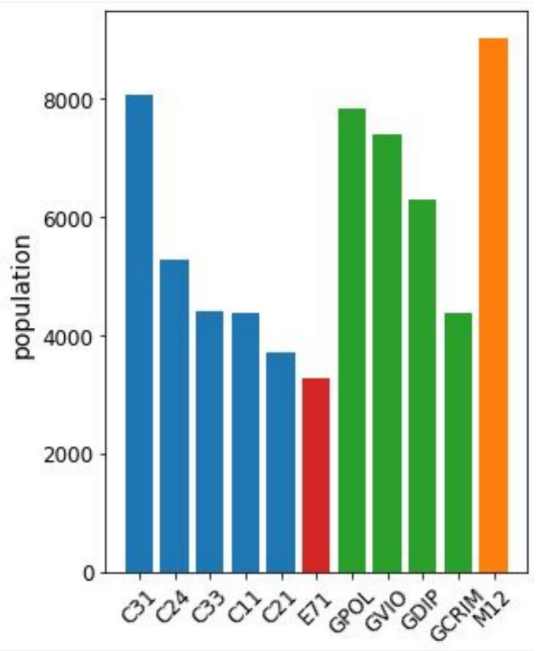
In this graph it is possible to see the population of each cluster grouped by macro-labels and sorted by values.



Cluster filter

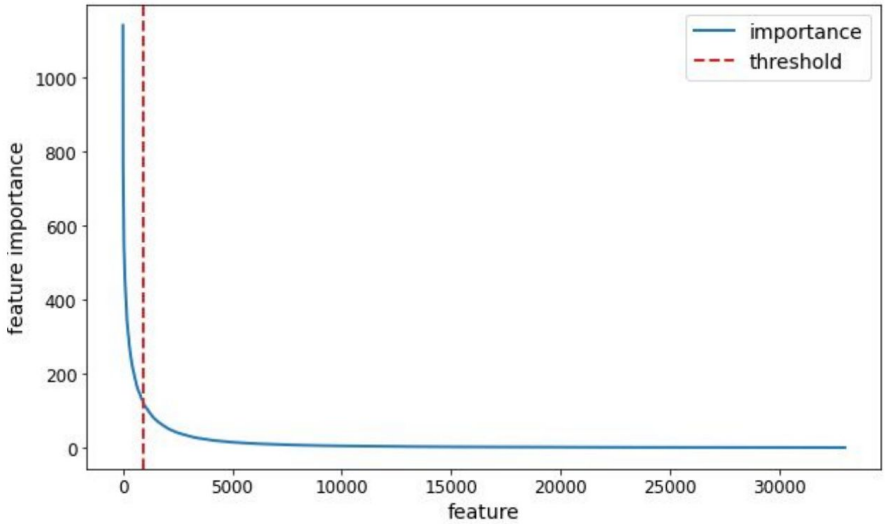
- The k-means clustering requires clusters to be of similar size in order to work properly (otherwise it gives too much weight to big clusters).

- 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

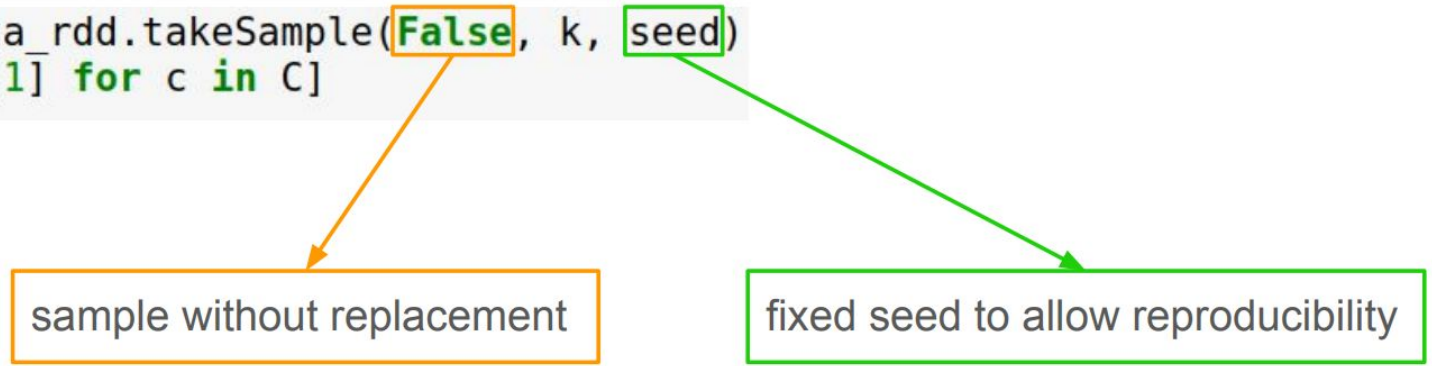
It is possible to reduce the dimensionality of the points by aggregating the features using the sum operation and selecting only the most important ones.



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]
```



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()
```

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()
```

Testing

Dataset

The k-means clustering algorithm is evaluated on the RCV1 **[4]** (Reuters Corpus Volume I) dataset, comprising over 800,000 manually categorized documents. The dataset, accessible through scikit-learn, is presented as a dictionary-like object with the following attributes:

- **Data Representation:** A scipy CSR sparse matrix containing 804,414 samples and 47,236 features. The features are represented by cosine-normalized, log TF-IDF vectors.
- **Target Information:** Another scipy CSR sparse matrix with 804,414 samples, categorizing them into 103 categories. Each category is represented by vectors with a value of 1 in the respective category and 0 in others.
- **Category Names:** An array providing the names of the categories. Notably, each document can belong to more than one category.

Dataset: Attributes

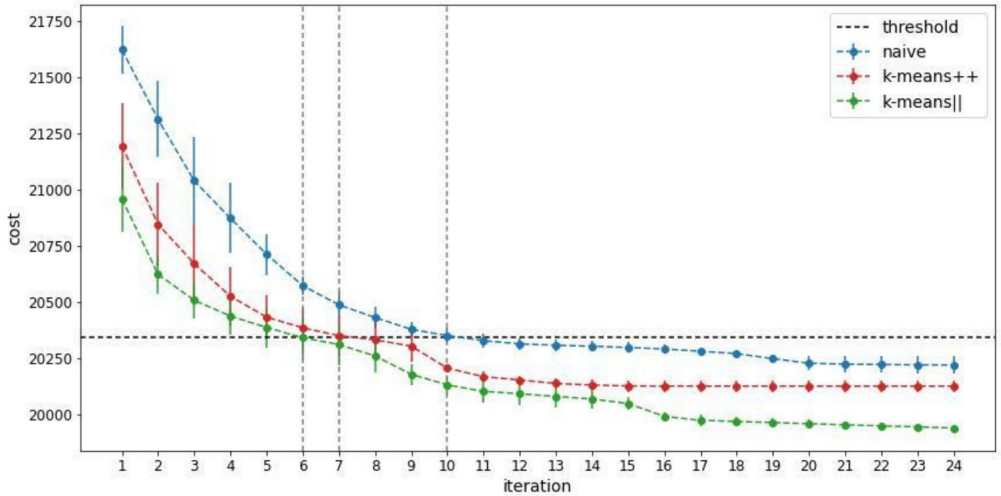
name:	class:	shape:
data	<class 'scipy.sparse._csr.csr_matrix'>	(804414, 47236)
target	<class 'scipy.sparse._csr.csr_matrix'>	(804414, 103)
target_names	<class 'numpy.ndarray'>	(103,)
sample_id	<class 'numpy.ndarray'>	(804414,)

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

Performances with different initializations: cost function

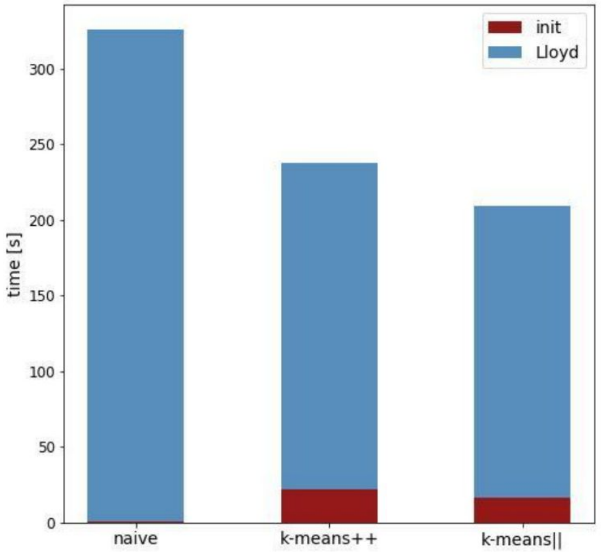
- Dataset: 32 chunks, 16000 documents each
- Number of partitions: 32
- Number of executors: 16, each with 1 core and 1 Gb of memory



Performances with different initializations: execution time

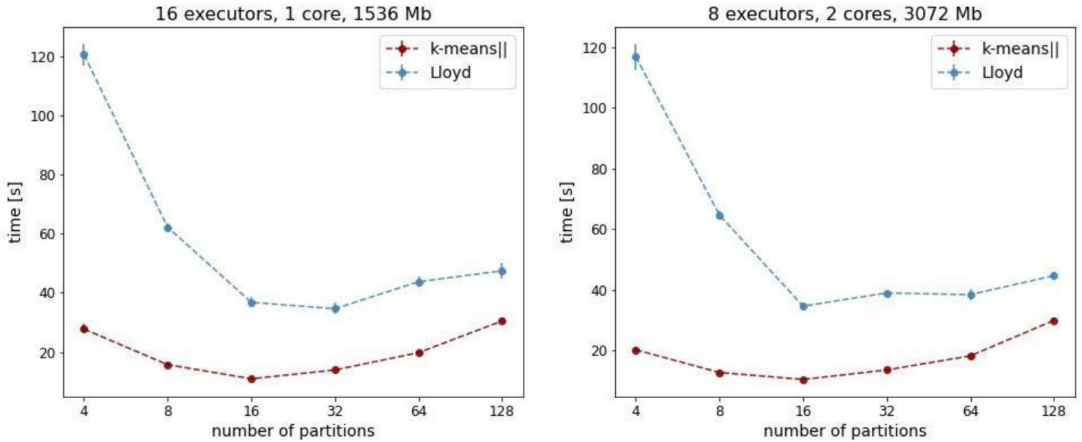
- As can be seen from the graph on the right, k-means|| initialization is on average faster than k-means++ initialization.

- 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



~1.5x

Faster than Naive Approach

References

[1] Bahmani, B., Moseley, B., Vattani, A., Kumar, R., & Vassilvitskii, S. (2012). Scalable K-Means++. ArXiv. /abs/1203.6402

[2] GeeksforGeeks. "K-means Clustering using PySpark in Python." [Online]. Available: https://www.geeksforgeeks.org/k-means-clustering-using-pyspark-python/

[3] Towards Data Science. "K-means Clustering using PySpark on Big Data." [Online]. Available: https://towardsdatascience.com/k-means-clustering-using-pyspark-on-big-data-6214beacdc8b

[4] Scikit-learn. "Real world datasets - Scikit-learn 0.24.2 documentation." [Online]. Available: https://scikit-learn.org/stable/datasets/real_world.html#rcv1-dataset

Thanks!