MapReduce Algorithms for k-means Clustering

Max Bodoia

1 Introduction

The problem of partitioning a dataset of unlabeled points into clusters appears in a wide variety of applications. One of the most well-known and widely used clustering algorithms is Lloyd's algorithm, commonly referred to simply as k-means [1]. The popularity of k-means is due in part to its simplicity - the only parameter which needs to be chosen is k, the desired number of clusters - and also its speed.

A number of efforts have been made to improve the quality of the results produced by k-means. One such attempt is the k-means++ algorithm, which uses the same iterative method but a different initialization [2]. This algorithm tends to produce better clusters in practice, though it sometimes runs more slowly due to the extra initialization step.

Given the ubiquity of k-means clustering and its variants, it is natural to ask how this algorithm might be adapted to a distributed setting. In this paper we show how to implement k-means and k-means++ using the MapReduce framework for distributed computing. Furthermore, we describe two other MapReduce algorithms, k-means** and k-means+*, which may also be effective on large datasets. Finally, we implement all four distributed algorithms in Spark, test them on large real-world datasets, and report the results.

2 Serial k-means algorithms

We begin by providing a formal description of the k-means problem and two serial algorithms for finding an approximate solution. Let $X = \{x_1, \ldots, x_n\}$ be a set of n data points, each with dimension d. The k-means problem seeks to find a set of k means $M = \{\mu_1, \ldots, \mu_k\}$ which minimizes the function

$$f(M) = \sum_{x \in X} \min_{\mu \in M} ||x - \mu||_2^2$$

In other words, we wish to choose k means so as to minimize the sum of the squared distances between each point in the dataset and the mean closest to that point.

Finding an exact solution to this problem is NP-hard. However, there are a number of heuristic algorithms which yield good approximate solutions. We will describe two: standard k-means, and k-means++.

2.1 Standard k-means

The standard algorithm for solving k-means uses an iterative process which guarantees a decrease in total error (value of the objective function $f(M)$) on each step [1]. The algorithm is as follows:

Standard k-means

---

1. Choose k initial means $\mu_1, \ldots, \mu_k$ uniformly at random from the set X.
2. For each point $x \in X$, find the closest mean μ_i and add x to a set S_i.
3. For $i = 1, \ldots, k$, set μ_i to be the centroid of the points in S_i.
4. Repeat steps 2 and 3 until the means have converged.

The convergence criterion for step 4 is typically when the total error stops changing between steps, in which case a local optimum of the objective function has been reached. However, some implementations terminate the search when the change in error between iterations drops below a certain threshold.

Each iteration of this standard algorithm takes time $O(nkd)$. In principle, the number of iterations required for the algorithm to fully converge can be very large, but on real datasets the algorithm typically converges in at most a few dozen iterations.

2.2 k-means++

The second algorithm we present is a version of the standard algorithm which chooses the initial means differently [2]. The algorithm is as follows:

---

k-means++

---

1. Choose the first mean μ_1 uniformly at random from the set X and add it to the set M.
2. For each point $x \in X$, compute the squared distance $D(x)$ between x and the nearest mean μ in M.
3. Choose the next mean μ randomly from the set X, where the probability of a point $x \in X$ being chosen is proportional to $D(x)$, and add μ to M.
4. Repeat steps 2 and 3 a total of $k - 1$ times to produce k initial means.
5. Apply the standard k-means algorithm, initialized with these means.

This algorithm is designed to choose a set of initial means which are well-separated from each other. In the paper where they first describe the k-means++ algorithm, the authors Arthur and Vassilvitskii test it on a few real-world datasets and demonstrate that it leads to improvements in final error

over the standard k-means algorithm [2]. Since each iteration of this initialization takes $O(|M|nd)$ time and the size of M increases by 1 each iteration until it reaches k, the total complexity of k-means++ is $O(k^2nd)$, plus $O(nkd)$ per iteration once the standard k-means method begins.

3 Distributed k-means algorithms

We now consider how to reformulate these two algorithms for solving the k-means problem so that they can be applied in a distributed setting. Specifically, we will formulate distributed versions of these algorithms using the MapReduce framework. We begin by presenting a MapReduce version of the two algorithms described in section 2, and then consider two new algorithms which have the potential to be effective on large-scale datasets.

3.1 Standard k-means via MapReduce

Recall that each iteration of standard k-means can be divided into two phases, the first of which computes the sets S_i of points closest to mean μ_i, and the second of which computes new means as the centroids of these sets. These two phases correspond to the Map and Reduce phases of our MapReduce algorithm.

The Map phase operates on each point x in the dataset. For a given x, we compute the squared distance between x and each mean and find the mean μ_i which minimizes this distance. We then emit a key-value pair with this mean's index i as key and the value $(x, 1)$. So our function is

`k-meansMap`(x):
    `emit` $(\text{argmin}_i ||x - \mu_i||_2^2, (x, 1))$

The Reduce phase is just a straightforward pairwise summation over the values for each key. That is, given two value pairs for a particular key, we combine them by adding each corresponding element in the pairs. So our function is:

`k-meansReduce`$(i, [(x, s), (y, t)])$:
    `return` $(i, (x + y, s + t))$

The MapReduce characterized by these two functions produces a set of k values of the form

$$\Big(i, \Big(\sum_{x \in S_i} x,\ |S_i|\Big)\Big)$$

where S_i denotes the set of points closest to mean μ_i. We can then compute the new means (the centroids of the sets S_i) as

$$\mu_i \leftarrow \frac{1}{|S_i|} \sum_{x \in S_i} x$$

Note that in order for the Map function to compute the distance between a point x and each of the means, each machine in our distributed cluster must

have the current set of means. We must therefore broadcast the new means across the cluster at the end of each iteration.

We can write the entire algorithm (henceforth referred to as `K-MEANS`) as follows:

`K-MEANS`

---

1. Choose k initial means $\mu_1, \ldots, \mu_k$ uniformly at random from the set X.
2. Apply the MapReduce given by `k-meansMap` and `k-meansReduce` to X.
3. Compute the new means $\mu_1, \ldots, \mu_k$ from the results of the MapReduce.
4. Broadcast the new means to each machine on the cluster.
5. Repeat steps 2 through 4 until the means have converged.

In the Map phase of this algorithm, we must do $O(knd)$ total work. The total communication cost is $O(nd)$, and largest number of elements associated with a key in the Reduce phase is $O(n)$. However, since our Reduce function is commutative and associative, we can use combiners and bring down the communication cost to $O(kd)$ from each machine.

Also, after the Map phase is completed, we must perform a one-to-all communication to broadcast the new set of means with size $O(kd)$ out to all the machines in the cluster. So in total, each iteration of `K-MEANS` does $O(knd)$ work (the same as the serial algorithm) and has communication cost $O(kd)$ when combiners are used, which can be broadcast in all-to-one and one-to-all communication patterns. As noted in the introduction, the number of iterations required for convergence can theoretically be quite large, but in practice it is typically a few dozen even for large datasets.

3.2 k-means++ via MapReduce

Next, we consider how to implement the k-means++ algorithm using MapReduce. Recall that each iteration in the initialization of k-means++ has two phases, the first of which computes the squared distance $D(x)$ between each point x and the mean nearest to x, and the second of which samples a member of X with probability proportional to $D(x)$. These two phases correspond to the Map and Reduce phases of our MapReduce algorithm for k-means++.

The Map phase operates on each point x in the dataset. For a given x, we compute the squared distance between x and each mean in M and find the minimum such squared distance $D(x)$. We then emit a single value $(x, D(x))$, with no key. So our function is

`k-means++Map`(x):
    `emit` $(x, \min_{\mu \in M} ||x - \mu||_2^2)$

The Reduce phase aggregates over all emissions from the Map phase, since these emissions do not have a key. We reduce the first element of two value pairs

by choosing one of these elements with probability proportional to the second element in each pair, and reduce the second element of the pairs by summation. So our function is

```
k-means++Reduce([(x, p), (y, q)]):
    with probability p/(p + q):
        return (x, p + q)
    else:
        return (y, p + q)
```

The MapReduce characterized by these two functions produces a single value of the form $(x, 1)$ where x is a member of the set X, and the probability of any particular element $x \in X$ being returned is proportional to the squared distance $D(x)$ between x and the nearest mean in M. This value x is then added to M as the next initial mean. As before, we must broadcast the new set of means M to the entire cluster between each iteration.

We can write the entire algorithm (henceforth referred to as `K-MEANS++`) as follows:

`K-MEANS++`

---

1. Initialize M as $\{\mu_1\}$, where μ_1 is chosen uniformly at random from X.
2. Apply the MapReduce `k-means++Map` and `k-means++Reduce` to X.
3. Add the resulting point x to M.
4. Broadcast the new set M to each machine on the cluster.
5. Repeat steps 2 through 4 a total of $k-1$ times to produce k initial means.
6. Apply the standard k-means MapReduce algorithm, initialized with these means.

In the Map phase of this algorithm, we must do $O(|M|nd)$ total work, where M is the current set of means. Since the algorithm runs for exactly $k-1$ iterations and each iteration increases the size of M by 1, the total amount of work done in the Map phase will be $O(k^2nd)$. The communication cost is $O(nd)$ per iteration for a total of $O(knd)$, and since we do not use a key, the largest number of elements associated with a key in the Reduce phase is $O(n)$. However, since our Reduce function is commutative and associative, we can use combiners and bring down the communication cost to $O(d)$ from each machine per iteration.

Also, after the Map phase is completed, we must perform a one-to-all communication to broadcast the newest mean with size $O(d)$ out to all the machines in the cluster. So over the course of k iterations, `K-MEANS++` does $O(k^2nd)$ work (the same as the serial algorithm) and has communication cost $O(kd)$ when combiners are used, which can be broadcast in all-to-one and one-to-all communication patterns. These costs are incurred in addition to the costs of `K-MEANS`, which runs after the initialization has completed.

3.3 Sampled k-means via MapReduce

The next MapReduce algorithm we present is identical to standard k-means, except that during the Map phase, we only process and emit a value for each point x with some probability α. This means that each iteration is only applied to a sampled subset of the data. We refer to this algorithm as k-means**. Our Map function is as follows:

```
k-means**Map(x):
    with probability min(α, 1):
        emit (argmin_i ||x − μ_i||_2^2, (x, 1))
```

The Reduce phase is identical to standard k-means.

For the algorithm as a whole, we initialize α to be some value less than one, and then on each iteration, we increase α by a factor of β, for some β greater than 1. When α eventually grows greater than 1, the algorithm will behave exactly like standard k-means. The idea here is that for the first several iterations of the algorithm, we are only running k-means on a sampled subset of the data, which gradually scales up to the full dataset as our algorithm converges.

We can write the entire algorithm (henceforth referred to as `K-MEANS**`) as follows:

`K-MEANS**`

---

1. Initialize $\alpha < 1$ and $\beta > 1$.
2. Choose k initial means $\mu_1, \ldots, \mu_k$ uniformly at random from the set X.
3. Apply the MapReduce given by `k-means**Map` and `k-meansReduce` to X.
4. Compute the new means $\mu_1, \ldots, \mu_k$ from the results of the MapReduce.
5. Set $\alpha = \beta \cdot \alpha$.
6. Broadcast the new means and new α to each machine on the cluster.
7. Repeat steps 3 through 6 until the means have converged.

In the Map phase of this algorithm, we must do $O(kd)$ work for each element with probability α, and none otherwise. This means that the total expected work is $O(\alpha nkd)$. The total expected communication cost is $O(\alpha nd)$, and the expected largest number of elements associated with a key in the Reduce phase is $O(\alpha n)$. However, since our Reduce function is commutative and associative, we can use combiners and bring down the communication cost to $O(kd)$ from each machine (assuming that $k < \alpha n$).

Also, after the Map phase is completed, we must perform a one-to-all communication to broadcast the new set of means with size $O(kd)$ and new α out to all the machines in the cluster. So in total, each iteration of `K-MEANS**` does

$O(\alpha nkd)$ work in expectation and has communication cost $O(kd)$ when combiners are used, which can be broadcast in all-to-one and one-to-all communication patterns.

As with `K-MEANS`, the number of iterations is theoretically large, and for `K-MEANS**` it will tend to be larger than `K-MEANS` since we expect to make less progress per iteration while $\alpha < 1$. However, we find that in practice the number of iterations is comparable to `K-MEANS` (see section 4).

3.4 Sampled k-means++ via MapReduce

The final MapReduce algorithm we present is one that we refer to as k-means+*, and can be thought of as a hybrid between k-means++ and k-means**, combining the initialization from the former and the sampling from the latter. Specifically, we modify the k-means++ Map function so that it also only emits a value for each x with some probability α. So our Map function can be written as follows:

```
k-means+*Map(x):
    with probability α:
        emit (x, min_{μ∈M} ||x − μ||_2^2)
```

The Reduce function is identical to the k-means++ Reduce function. The resulting MapReduce will thus generate a random sample from a subset of X with expected size $\alpha|X|$ rather than the full set X, while retaining the property that each point is chosen with probability proportional to its squared distance from the nearest mean.

We can write the entire algorithm (henceforth referred to as `K-MEANS+*`) as follows:

`K-MEANS+*`

---

1. Initialize M as $\{\mu_1\}$, where μ_1 is chosen uniformly at random from X.
2. Apply the MapReduce `k-means+*Map` and `k-means++Reduce` to X.
3. Add the resulting point x to M.
4. Broadcast the new set M to each machine on the cluster.
5. Repeat steps 2 through 4 a total of $k-1$ times to produce k initial means.
6. Apply the k-means** MapReduce algorithm, initialized with these means.

Note that in step 6 we run the k-means** algorithm after selecting the initial means. This means that the k-means+* algorithm samples during both the mean initialization and the subsequent iterative optimization.

In the Map phase of this algorithm, we do $O(|M|d)$ work on each element with probability α, and none otherwise. This means that our Map phase will do $O(\alpha n|M|d)$ work in expectation on each iteration. Since the algorithm runs for

exactly $k-1$ iterations and each iteration increases the size of M by 1, the total amount of work done in the Map phase will be $O(\alpha nk^2d)$ in expectation. The expected communication cost is $O(\alpha nd)$ per iteration for a total of $O(\alpha nkd)$, and since we do not use a key, the largest number of elements associated with a key in the Reduce phase is $O(\alpha n)$ in expectation. However, since our Reduce function is commutative and associative, we can use combiners and bring down the communication cost to $O(d)$ from each machine per iteration (assuming that $d < \alpha n$).

Also, after the Map phase is completed, we must perform a one-to-all communication to broadcast the newest mean with size $O(d)$ out to all the machines in the cluster. So over the course of k iterations, `K-MEANS++` does $O(\alpha nk^2d)$ work in expectation and has communication cost $O(kd)$ when combiners are used, which can be broadcast in all-to-one and one-to-all communication patterns. These costs are incurred in addition to the costs of `K-MEANS**`, which runs after the initialization has completed.

4 Empirical tests

To compare the effectiveness of these algorithms, we implemented each one in Spark and tested them on real-world datasets.

4.1 Datasets

We use two large real-world datasets from the UC Irvine Machine Learning Repository [3]. The first dataset (*Power*) consists of power consumption readings from various households and contains 512,320 real-valued data points, each with dimension 7 [4]. The second dataset (*Census*) consists of data from the 1990 US Census and contains 614,571 integer-valued data points, each with dimension 68 [5].

4.2 Results

We ran the four algorithms `K-MEANS`, `K-MEANS++`, `K-MEANS**`, and `K-MEANS+*` on each of the datasets and used two different values for k, $k = 5$ and $k = 10$. As our convergence criterion, we tracked the average error and terminated the algorithm when the change in average error after an iteration was less than 0.1%. For the algorithms `K-MEANS**` and `K-MEANS+*`, we used the parameter value $\alpha = 0.1$ in all cases. We set $\beta = 1.25$ when $k = 5$, and $\beta = 1.5$ when $k = 10$. These values were chosen so that the number of iterations before α grew above 1 (and the algorithm reverted to regular, non-sampling behavior) was roughly equal to the number of iterations required for the `K-MEANS` and `K-MEANS++` algorithms.

For each combination of dataset, algorithm, and k-value, we performed 10 full runs and recorded the average error, minimum error, and average runtime. The results are presented in the four tables below.

Table 1: *Power* dataset, $k = 5$.

Algorithm	Average Error	Minimum Error	Average Runtime
K-MEANS	32.78	27.02	308
K-MEANS++	48.50	27.61	270
K-MEANS**	35.33	27.01	204
K-MEANS+*	28.06	27.01	217

Table 2: *Power* dataset, $k = 10$.

Algorithm	Average Error	Minimum Error	Average Runtime
K-MEANS	21.66	19.46	459
K-MEANS++	18.69	17.32	844
K-MEANS**	26.10	19.08	366
K-MEANS+*	18.35	16.59	465

Table 3: *Census* dataset, $k = 5$.

Algorithm	Average Error	Minimum Error	Average Runtime
K-MEANS	397.73	136.33	402
K-MEANS++	148.44	136.33	527
K-MEANS**	491.62	141.98	342
K-MEANS+*	153.60	136.33	396

Table 4: *Census* dataset, $k = 10$.

Algorithm	Average Error	Minimum Error	Average Runtime
K-MEANS	154.17	100.62	616
K-MEANS++	104.05	99.64	1217
K-MEANS**	356.20	100.36	618
K-MEANS+*	105.68	99.51	915

4.3 Discussion

In this section, we consider the performance of each algorithm in turn across the four test cases.

For the K-MEANS algorithm, we see mediocre performance in both error and runtime. This algorithm ranks third in three out of the four cases by average error, and either third or fourth in three out of the four cases by minimum error. Its runtime is better, ranking second or third in three of the four cases, but not spectacular enough to offset its poor average and minimum error.

For the K-MEANS++ algorithm, we see much better performance in terms of error: first or second in average error and minimum error in three of the four test cases (though it surprisingly comes last in both these categories for the *Power* dataset with $k = 5$). However, its main weakness is runtime - it comes dead last in three of the four cases, usually by a significant margin. This is to be expected given the extra initialization step used by the algorithm.

The K-MEANS** algorithm is the opposite: impressive in terms of runtime but less so in terms of error. It is the fastest algorithm in all but the last case,

and even then effectively ties for first place. However, it ranks fourth on three of the four cases by average error and either third or fourth in all cases by minimum error. This behavior is also to be expected, given that the sampling method used by the algorithm trades off accuracy for speed.

Finally, the `K-MEANS+*` algorithm seems to give us the best of both worlds. It ranks first or second in every case by average error, and first in every case by minimum error. Yet it also ranks second or third in every case by runtime, and it is significantly faster in all cases than `K-MEANS++`, its nearest competitor in terms of average and minimum error.

5 Conclusion

For this project, we considered how the popular serial clustering algorithms k-means and k-means++ might be implemented in a distributed computing environment. First, we described MapReduce formulations of these two algorithms (`K-MEANS` and `K-MEANS++`) and analyzed their complexity and communication costs. Then, we modified these algorithms to produce two new distributed clustering algorithms. The first one, `K-MEANS**`, uses an iteratively scaling sampling scheme to speed up the first several iterations of standard `K-MEANS`. The second one, `K-MEANS+*`, applies the same sampling principle to the initialization of the `K-MEANS++` algorithm. We also provided an analysis of complexity and communication cost for these two novel algorithms. Finally, we implemented all four algorithms in Spark and tested them on two large real-world datasets.

We found that the first two distributed algorithms `K-MEANS` and `K-MEANS++` performed much like their serial counterparts: `K-MEANS++` produced clusters with lower average and minimum error than `K-MEANS`, but unfortunately sacrificed speed in the process. The `K-MEANS**` algorithm, our first attempt at a sampling-based k-means algorithm, succeeded in running faster than its non-sampling counterparts, but failed to match the lower error rate of `K-MEANS++`. However, the `K-MEANS+*` algorithm managed to leverage sampling to beat the slow runtime of `K-MEANS++` (and often `K-MEANS` as well) while maintaining the lowest error rate of all four algorithms. This provides evidence that when working with large datasets in a distributed environment, sampling can be used to speed up the operation of the k-means clustering algorithm and its variants without sacrificing cluster quality.

References

[1] Stuart P Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[2] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[3] M. Lichman. *UCI Machine Learning Repository.* University of California, Irvine, School of Information and Computer Sciences, http://archive.ics.uci.edu/ml, 2013.

[4] Alice Berard and Georges Hebrail. Searching time series with hadoop in an electric power company. In *Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 15–22. ACM, 2013.

[5] Bo Thiesson, Christopher Meek, and David Heckerman. Accelerating em for large databases. *Machine Learning*, 45(3):279–299, 2001.