Algorithm Paper:

Hilawi Belachew, Christian Junge, Akhil Ketkar, Juaquin Sanchez, Daniel Yue

The Scikit-Learn implementation of K-means contains several optimizations that are designed to reduce computation. This implementation includes several options that allow the user to navigate the tradeoffs between desired accuracy and computation. These fall under the following categories:

1. Initialization
2. Thresholds
3. Parallelization of multiple jobs
4. C under the hood
5. Precompute Distances
6. Initialization

The k-means clustering algorithm finds cluster centers iteratively by starting with a set of cluster center assignments, and then assigning each point to the closest cluster center. Once all points are assigned, cluster centers are recalculated to be the center of all of the points in that cluster, and each point is reassigned to the closest cluster center. This terminates when no updates are made to the cluster assignments or cluster centers.

This process is guaranteed to converge, but not necessarily on the global optimum. The worst case complexity is given by O(n^(k+2/p)) with n = n\_samples, p = n\_features. (D. Arthur and S. Vassilvitskii, ‘How slow is the k-means method?’ SoCG2006), but in practice the algorithm converges very quickly.

Given a specific set of initial clusters, the k-means algorithm is deterministic, so the results are sensitive to initialization. The most basic initialization is to randomly select k of the data points to be the initial cluster centers. The weakness of this method is that if these initial points are close together, the algorithm may fail to find the true cluster centers. The k-means++ initialization attempts to take care of this weakness by first randomly assigning one point to be the first initial cluster center, then choosing the next initial cluster center based on a probability weighted by its distance to the first cluster center, and so on until k initial cluster centers have been assigned.

The advantage of k-means++ is that it will, on average, space the initial cluster centers out more evenly across the space than a completely random initialization. Because the probabilities of being selected are merely weighted by the distance, however, this initialization does not guarantee that the centers will be more widely spaced, it simply makes this more likely. Furthermore, there is a computational overhead for the k-means++ initialization, as it requires distances between every point and every cluster center so far created to be calculated as each cluster center is added. The tradeoff is that this initialization tends to produce better cluster centers, and can potentially reduce the number of iterations for convergence.

The scikit-learn implementation also allows the user to specify the initial cluster centers rather than using random assignments. This could be useful if the user has some a priori idea of where the cluster centers should be, potentially based on some domain knowledge, or when adding more data after earlier clustering runs. This option would be unlikely to be used for clustering that is performed as part of data exploration, however, and in practice most users will be relying on some level of randomization for their initial cluster assignments.

Because the cluster assignments are deterministic, they are very sensitive to the initialization, and the algorithm is not guaranteed to converge on the global optimum. One way to avoid being stuck in an unfavorable local optimum is to reinitialize and run the clustering again. The scikit-learn implementation includes a ‘n\_init’ keyword argument to allow the user to specify the number of re-initializations that are performed, and the final version that is returned is the one that minimizes the sum of the distances from the cluster centers.

1. Thresholds

The scikit-learn k-means implementation has two features that allow the user to limit computation explicitly: a threshold for convergence, and a maximum number of iterations.

The convergence threshold is passed as the ‘tol’ argument, and the basic idea behind this option is to stop early and declare that the algorithm has converged when changes to the cluster centers become very small, rather than waiting for them to stop changing completely. In practice, this works by first calculating the product of tol with the mean of the variances of each feature in the whole dataset, producing a quantity to be used to determine convergence that takes into account the scale of the variation of the individual features. Then when the change in the squared norm falls below this tolerance value, the algorithm terminates, cutting off what could potentially be many more iterations that are making very minute changes to what is already a nearly complete clustering run.

The maximum number of iterations is passed as the ‘max\_iter’ argument, and the algorithm simply runs within a for loop for this number of iterations. Within each iteration, the convergence check described above is performed, and allows the algorithm to break out of the for loop early when the updates become small.

1. Parallelization of multiple jobs

While the k-means algorithm is serial, the user has the option (described above) to perform multiple runs with different initializations and take the result with the lowest total distance to the cluster centers. The scikit-learn implementation parallelizes these runs using scikit-learn’s own Parallel function within its externals/joblib module. By default, this uses Python multiprocessing, and the user can specify the number of jobs to run with the ‘n\_jobs’ argument. The user can allow this assignment to be made automatically by passing ‘n\_jobs=-1’, in which case all CPU’s are used. The user can also opt to run everything with a single job, in which case multiprocessing is not used at all.

1. C under the hood

The computation of the cluster means and the computation of the cluster assignments within each iteration are embarrassingly parallel. All of this computation is performed in Cython in the .\_k\_means.pyx file. Both the cluster means and cluster assignments calculations have two versions each, one for dense arrays and one for sparse arrays. The code first uses scipy.sparse.issparse() to determine identify whether the array to compute on is a regular numpy matrix or a scipy Compressed Sparse Row matrix before selecting which function to pass the array to. All of these Cython functions enjoy increased efficiency gains from defining variables with static C type declarations and performing matrix multiplication with cblas\_ddot.

1. Precompute Distances

In all cases, label assignments are updated in-place, reducing memory usage. Distances to cluster centers can also be updated in place, but speed gains can be made by precomputing the distances via numpy matrix operations within the euclidian\_distances function from sklearn.metrics.pairwise, rather than one-by-one within the for loop that updates label assignments. These speed gains come with the tradeoff of increasing memory usage by requiring two copies of distances to be stored, however, so this option is given to the user. There is an ‘auto’ option, which precomputes the distances if it will not result in memory usage above a hard-coded threshold. This threshold corresponds to 100MB of overhead per job, and is calculated as number of samples \* number of clusters < 12e6.

The option to precompute distances is ignored for scipy sparse matrices. Instead, the distance of each point from its cluster center is updated in a for loop as in the precompute\_distances=False option for a regular numpy array. Examination of the euclidean\_distances function revealed that it can operate on scipy sparse matrices, so it is unclear why this option is not applied for this data type.