CS207 Algorithm: K-Means Clustering

Amy Lee Vinay Subbiah Victor Lei Isadora Nun Fanny Heneine

May 11, 2016

Introduction

K-means is an unsupervised learning algorithms that solve the clustering problem by partitioning and classifying n observations of a given data set into a fixed number k of clusters[1]. It defines k centroids, one for each cluster and each observation belongs to the cluster with the nearest mean μ_k. The clusters tend to be of comparable spatial extent and this process results in a partitioning of the data space into Voronoi cells [2]. The objective function to be minimized for the algorithm is a squared error function:

$$\sum_{k=1}^{K} \sum_{x_i \in c_k} \left|\left| x_i - \mu_k \right|\right|^2 \tag{1}$$

$$\mu_k = \frac{1}{C_k} \sum_{x_i \in c_k} x_i \tag{2}$$

where $||x_i - \mu_k||$ is the Euclidean distance between x_i and μ_k (the mean of the k^{th} cluster), c_k is the number of data points in the k^{th} cluster and K is the number of clusters.

After the first classification has been done, the μ_k new centroids are recalculated as the mean of all points belonging to each cluster, and the clustering of observations is re-conducted and updated. The process is iterative until the k centroids do not change location anymore: The algorithm has converged, but the solution might be a local minimum. This is known as Lloyds algorithm[3]

$$C_k = \{x_i : ||x_i - \mu_k|| \leq \text{all } ||x_i - \mu_l||\} \tag{3}$$

For the starting set of centroids, several methods can be employed, for instance random assignation.

Naive Implementation in Python

As a first step, we implemented a K-means clustering algorithm using Numpy. The snippets of the code are shown below in figure 1.

[1] Data Clustering algorithm at https://sites.google.com/site/dataclusteringalgorithms/k-means-clustering-algorithm

[2] A Voronoi diagram is a partitioning of a plane into regions based on distance to points in a specific subset of the plane

[3] https://datasciencelab.wordpress.com/2013/12/12/clustering-with-k-means-in-python/

```python
def cluster(X, mu):
    clusters  = {}
    for x in X:
        best_mu = min([(i[0], np.linalg.norm(x-mu[i[0]])) \
                    for i in enumerate(mu)], key=lambda t:t[1])[0]
        try:
            clusters[best_mu].append(x)
        except KeyError:
            clusters[best_mu] = [x]
    return clusters

def update(mu, clusters):
    newmu = []
    keys = sorted(clusters.keys())
    for k in keys:
        newmu.append(np.mean(clusters[k], axis = 0))
    return newmu
```

```python
def converged(mu, oldmu):
    # The algorith converge when the centroids stop moving
    return (set([tuple(a) for a in mu]) == set([tuple(a) for a in oldmu]))

def Kmeans_imp(X, K):
    iterations= 0
    loss=[]
    # Initialize to K random centers
    oldmu = random.sample(X, K)
    mu = random.sample(X, K)
    while not converged(mu, oldmu):
        oldmu = mu
        # Assign all points in X to clusters
        clusters= cluster(X, mu)
        # Reevaluate centers
        mu = update(oldmu, clusters)
        iterations+=1
    return(mu, clusters, iterations,loss)
```

Figure 1: Python and Numpy naive implementation

The process starts by randomly sampling the cluster centers. Next, the points are assigned to each cluster using the minimum Euclidean distance and the centers are then re-evaluated. A while loop is used to ensure the process is iterative until the cluster barycentre's do not update anymore and remain the same.

To ensure validity of the algorithm, random data (of size 5000) is generated and the K-means naive implementation results are compared to the Scikit-Learn results[4], which are plotted in figure 2. We must note that the sklearn algorithm includes two implementation: The Kmeans and MiniBatch Kmeans, which differences will be discussed later on.

Comparison

We compare the performance of our naively implemented algorithm and the Scikit-Learn modules (both KMeans and MiniBatch KMeans) on several randomly generated datasets of increasing size. To ensure consistency in our comparison, we initialize all input similarly: The standard deviation is

[4]Example taken from:http://scikit-learn.org/stable/auto-examples/cluster/plot-mini-batch-kmeans.html

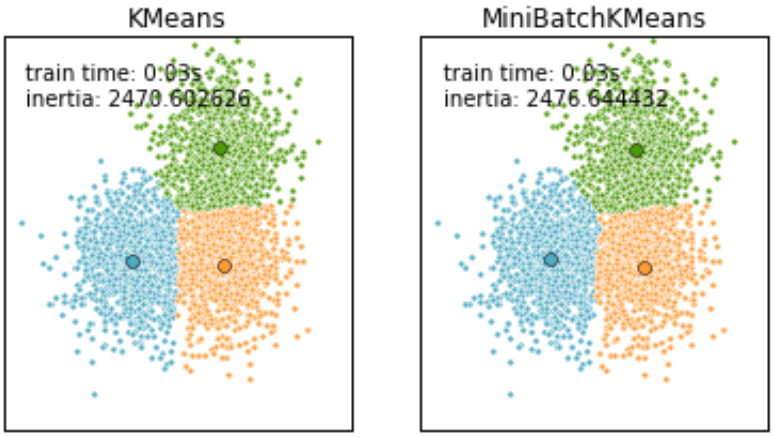
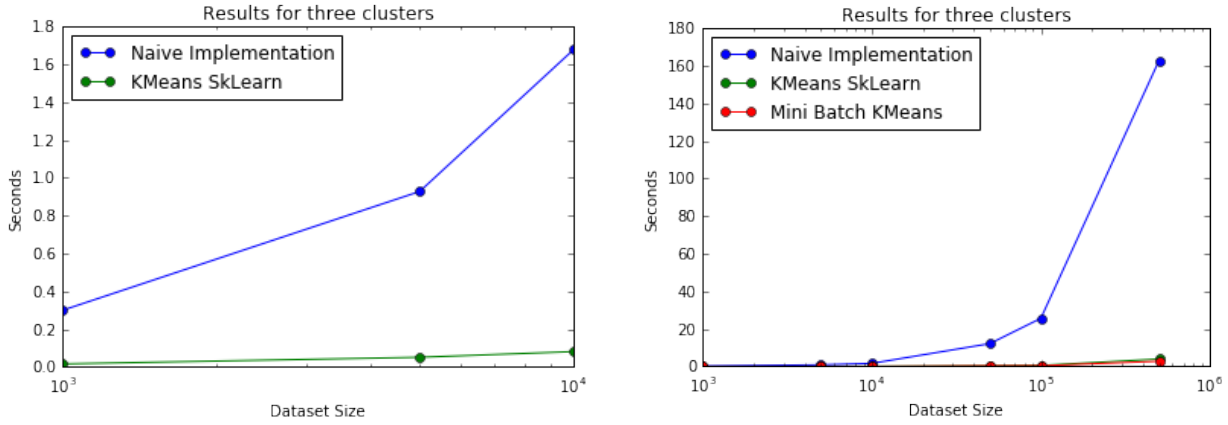


Figure 2: Sklearn Implementation Results

equal across each clustering, three clusters are generated and the batch size is always 45 for the miniBatch algorithm. A discussion of the best input parameter is beyond the score of the paper. Our naive implementation is inherently slower, and the difference grows exponentially with the size of the dataset, as illustrated in figure 3 below. Two figures are shown as the full figure on the left hand side does not clearly illustrate the difference in time for smaller datasets due to its scale.



(a) Small datasets

(b) Bigger Datasets

Figure 3: Comparison between naive implementation and Scikit-Learn K-Means Project

Optimization

When comparing both Scikit-Learn Project implementations, which primary module is `k_means_.py`, with our naive Python implementation, an obvious gap in performance is observed. The performance of the SkLearn project algorithms has been drastically more efficient, with a much shorter convergence time. The naive implementation performance is quite slow, with 160 seconds required to group into three clusters half a million data points. Therefore, the need for optimization becomes

obvious in order to handle bigger datasets, but also to decrease the number of iterations required.

In this section, we examine how the algorithm is optimized to improve convergence time. The Scikit-Learn KMeans includes several optimizations components, both within the pure Python implementation and by making use of Cython. In addition, a variant of KMeans (Mini Batch KMeans) is detailed to analyze how its specificities enhances efficiency.

Pure Python Optimization tools

The first optimization consists of careful seeding. The algorithm selects initial cluster centers for k-mean clustering in a smart way to speed up convergence, hence decreasing the number of iterations needed to reach a solution.

Indeed, we have seen that the algorithm converges but maybe to a local minimum. This is a directly impacted by the initialization of the centroids in the first step, which leads to several computation of the algorithm (in order to randomly generate different initial centroids). The Scikit Learn algorithm provides a solution that reduces computation times and speeds up the algorithm by providing a method that solves this issues: The k-means++ initialization generates centroids that are distant from each other, which yields results that are better than when using randomly chosen centroids. This first optimization tool reduces computation time by 25%: We run the algorithm on the same dataset with and without the input parameter "init='k-means++'", and we see the run time reduce from 0.057 seconds to 0.044 seconds.

Another feature implemented in K-means is pre-computing distance with a preliminary check for memory issues: When pre-computing the distances, the algorithm creates a m by n matrix with m being the number of clusters and n the number of samples. Therefore this feature is activated only under a certain threshold of 12 million entries, to ensure that memory is not eaten up.

Finally, K-means also provides the possibility of parallel computing to optimize running time. By using the n_{jobs} input to K-means (which has a default value of 1), and assigning it a positive value, the algorithm will use the number of assigned processors. This parallelization of k-means runs increases speeds but also increases the memory cost of as several copies of centroids need to be stored.

Cython

Cython is an optimising static compiler that creates an interface between Python and C, allowing to combine higher-level and lower-level code. It allows users to easily write Python code that calls back and forth from C: Cython transforms Python code into a format that compiles into C code[5]. The main advantage of this extension resides in the fact that the speed of C, which results from the ability to compile, can now be incorporated into Python. Cython therefore allows developers to improve the performance of their code without having to deal with the intricacy of lower-level coding.

[5]Cython C-Extension for Python: http://cython.org/

In the Scikit-Learn `k_means_.py` module, several components are written in Cython: `_k_means.pyx` and `_k_means\_elkan.pyx` are imported and used within the module. `_k_means.pyx` is a Cython code for efficient computation of distances and labels within dense matrices or Compressed Sparse Row Format (or sparse CSR) matrix representation[6]. Numpy operations are implemented in C to compute label assignment and inertia[7] and to calculate cluster means, for either a sparse or dense matrix, as functions exist for both scenarios. It is also used in the Mini batch variant, which we discuss below, to conduct incremental update of the centers and compute squared difference and inertia of the sampled obeservations. This association between C and Numpy allows faster operations.

On the other hand, `_k_means\_elkan.pyx` is a variant of the algorithm beyond the scope of the paper. It is only called either when the data is dense or if the user calls algorithm="elkan" and uses the triangle inequality to speed up k-means for dense data: It calculates an upper and lower bound distance for each sample.

Mini Batch Kmeans

The MiniBatchKMeans is a different version of the KMeans algorithm which uses the same function to be minimized but samples subsets of the input data to reduce the computation time. These subsets are called mini-batches, and are randomly sampled in each iteration. The MiniBatchKmeans algorithm steps are as follows: Samples are extracted from the dataset (this is a mini-batch) and are allocated to the closest centroid. The centroids are then updated but each sample. As explained in the Scikit Learn documentation[8]: "For each sample in the mini-batch, the assigned centroid is updated by taking the streaming average of the sample and all previous samples assigned to that centroid". This leads to a important decrease in the rate of change for a centroid over time.

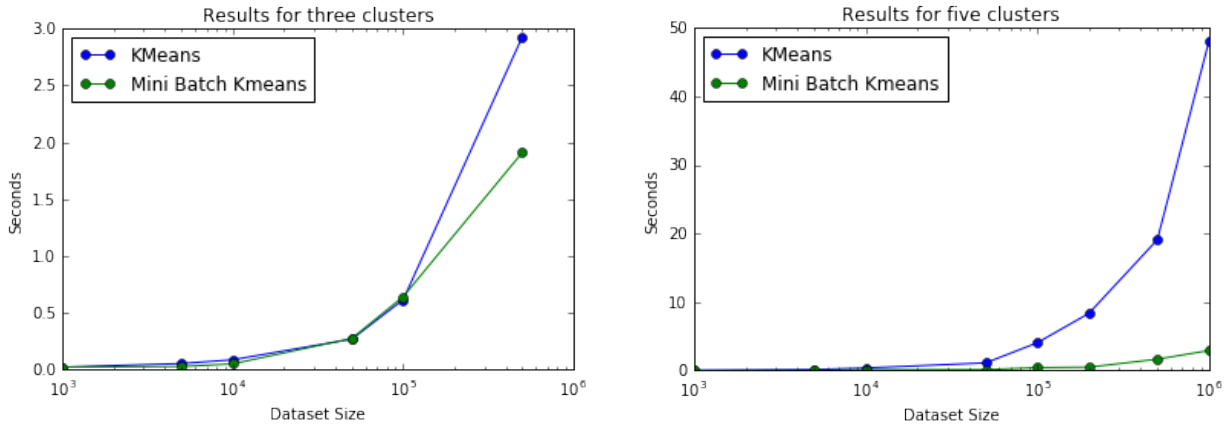
This algorithm therefore converges faster than the standard KMeans as the amount of computation that leads to a local minimum converges is diminished. The difference in convergence time is negligible for small datasets ($\leq 10^5$), as seen in figure 4a but becomes drastic when the size increases, as seen in figure 4b.

We must however note that convergence time is reduced at a certain cost: The mini batch KMeans results are slightly worse than the ones generated by KMeans.

Conclusion

As a conclusion, different tools are combined to enhance the efficiency of the algorithm, from the implementation of pure Python improvements to the use of Cython to enhance performance. The combination of these tools as well as the option to use an even better-performing algorithm (Mini Batch KMeans within the same module) makes this project a very strong clustering algorithm.

[6]Matrix in which most of the elements are zero and which compresses the row indices
[7]Sum of squared distances to the centroids
[8]SKLearn Clustering Documentation: http://scikit-learn.org/stable/modules/clustering.html



(a) Small datasets

(b) Bigger Datasets

Figure 4: Mini Batch KMeans Agorithm Convergence time