Analysis on Optimization of K-means Clustering Algorithm in Scikit-learn

A Project Report presented

by

Jeewon Hwang, Yuhan Tang, Chenchen Zhang, Qing Zhao

in partial fulfillment of the requirements

of CS 207: Systems Development for Computational Science

Analysis on Optimization of K-means Clustering Algorithm in Scikit-learn

Abstract

K-means clustering is a very popular clustering technique which is used in numerous applications. Given a set of n data points in R^d and an integer k, the problem is to determine a set of k points R^d, called centers, so as to minimize the mean squared distance from each data point to its nearest center. A popular heuristic for k-means clustering is Lloyd's algorithm. In this paper, we first analyzed implemented this algorithm in Python and then took a close look at its implementation in Scikit-learn library. The analysis is mainly focused on Scikit-learn's optimization of the algorithm by comparing the performance of the two implementations over data set with various sizes.

Índice general

Índice de figuras

1

Introduction to K-means Clustering

K-means algorithm(Lloyd, 1982) is a non-probabilistic technique of finding clusters in a set of data points. We also treat K-means as an unsupervised learning technique. K-means algorithm is intuitively blatant to perceive. First of all, we find K cluster centers, then assign all data point to one and only one cluster. Thinking deeper we find that this algorithm is actually not that easy to implement due to the existence of two-sided dependencies between assigned classes and cluster center. In another word, when we change cluster centers, the change would affect data points' cluster assignments. Nevertheless, after changes made to cluster assignments, the cluster center's values must be updated accordingly. I will propose a unique algorithm of Expectation

Maximization algorithm, called Max-max algorithm to solve this problem. Note this algorithm is also called Floyd's algorithm.

1.1. Lloyd's Algorithm

We begin by considering the problem of identifying clusters of data points in a multidimensional space. Suppose we have a data set consisting of N data points $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$, and each data points has D-dimensions.

1.1.1. Goal

We want to partition the data set into some number K of clusters, and K is determined by the user. In our perspective, K is given, and $K \in \mathbb{Z} \forall K$. We want to find a cluster as comprising a group of data points whose inter-point distances are small compared with the distances to points outside the cluster.

1.1.2. Procedures

Lets define variables first.

- $\boldsymbol{\mu}_\mathbf{k}$ center values of k^{th} cluster, where $k \in \{1, \ldots, K\}$
- $\mathbf{x}_\mathbf{n}$ is the n^{th} data point
- r_{nk} is the indicator variable. $r_{nk} = 1$ if and only if the n^{th} point has been assigned to k^{th} class.
    - $\mathbf{r}_n$ is K-dimension vector, consisting of all 0's expect one 1's. The only 1's occurs when at position k, where k is the cluster that data point $\mathbf{x}_n$ has been assigned to.
    - $\sum_K r_{nk} = 1$

- define an objective function, which is the total Euclidean distance of all data points to their own cluster centers.

$$J = \sum_{N} \sum_{K} r_{nk} ||\mathbf{x}_n - \boldsymbol{\mu}_k||^2$$

  - note that the inner summation over K has only one contributor when the n^{th} data point has been assigned to a certain k, and the rest of them are all 0's

Our goal is to find values for $\{r_{nk}\}$ and $\{\mu_k\}$ that minimize J. Here we are facing the problem introduced above about the two directional dependencies between $\{r_{nk}\}$ and $\{\mu_k\}$. The following context in this section is to introduce Expectation Maximization algorithm to solve the problem.

1.1.3. Expectation Maximization

1. Randomly initialize $\boldsymbol{\mu}_k$
2. E-step.
   - Minimize J with respect to r_{nk}, while treading $\boldsymbol{\mu}_\mathbf{k}$ fixed, as the following:
   - $$r_{nk} = \begin{cases} 1 & \text{if} \quad k = \text{argmin}_j ||\mathbf{x}_n - \boldsymbol{\mu}_k||^2 \\ 0 & \text{otherwise} \end{cases}$$
   - In another word, it means that we are assigning the n^{th} to a cluster k centered at a $\boldsymbol{\mu}_k$ that has the smallest Euclidean distances across all clusters k∈K.
3. M-step
   - Minimize J with respect to $\boldsymbol{\mu}_k$, keeping r_{nk} at fixed values calculated above

- $$\frac{\partial J}{\partial \boldsymbol{\mu}_k} = 2 \sum_N r_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k)$$

- setting $\frac{\partial J}{\partial \boldsymbol{\mu}_k} = 0$ gives us

$$\boldsymbol{\mu}_k = \frac{\sum_N r_{nk}\mathbf{x}_n}{\sum_N r_{nk}}$$

- note that the denominator in this expression is equal to the number of points assigned to cluster k. This means that : take the average of all points assigned to class k.

4. Redo part 1-3 until no further change in r_{nk} and $\boldsymbol{\mu}_k$. This step will be achieved because we are having hard assignment to r_{nk} because $r_{nk} \in \{0, 1\}$, and this is exactly the reason why K-means is a non-probabilistic clustering method.

1.1.4. Conclusion

In this chapter we examined K-means algorithm in detail, and proposed how to solve this NP-hard problem using Expectation Maximization.

2

Implementation of K-means Clusterting in Python

In this chapter we will discuss about one K-mean implementation in detail. This implementation is build upon Lloyd's algorithm that we introduced in Chapter 1, together with an naive optimization called K-mean++. Like all Machine Learning methods provided by Scikit-learn, this implementation provides a fit method, together with a corresponding predict method. The data set used is MNIST handwritten digit data set, and we used this algorithm to classified each picture into its corresponding digit cluster

2.1. Initialization

```
    def initMus(self,X):
        if not self.useKMeansPP:
            return X[np.random.randint(low = 0, high = self.
    num_pic, size = self.K)]
        elif self.useKMeansPP:
            return self.kpp(X)
        else:
            raise ValueError
```

The code above shows that we initialized all the cluster centers randomly. Since there are K clusters, so we initialize K centers. Each center has the same dimensionality as the input data.

2.2. FIT

```
    def fit(self, X):
        self.num_pic = X.shape[0]
        mus = self.initMus(X)
        # testing dimension of centers
        m = 1000
        centeriods = mus
        j = []
        while True:
            R = np.zeros((self.num_pic,self.K))
            assert np.sum(R) < 1
            dist = np.zeros_like(R)
            for ks in xrange(self.K):
                dist[:,ks] = np.linalg.norm(X - mus[ks],axis
    =(1,2))
            for i in xrange(R.shape[0]):
                R[i,np.argmin(dist[i,:])] =1
            for i in xrange(self.K):
                sel = np.array(R[:,i],dtype=np.bool)
                mus[i]= (np.mean(X[R[:,i]==1],axis = (0)))
            # keep track of error
```

```
            j.append(np.sum(dist[R.astype(np.bool)]))
            if len(j)>1:
                if j[-1] == j[-2]:
                    print 'Converged in {} iterations, find {} means\n---------- processing rep next ----------\n'.format(len(j),self.K)
                    break
            elif len(j)>1000:
                raise ValueError('maximum iteration reached')
        self.__dist__ = dist
        self.j = j
        self.mus = mus
```

This function used a matrix to record all cluster assignment. This assignment matrix is denoted as $\mathbf{R} \in \mathbb{R}^{K \times N}$, where N is the size of the entire data set. Note that this matrix is large and sparse, since for each $n \in \mathbb{N}$, there is only one non-zero element, namely $\sum_n r_{nk} = 1$. Inside the loop, we used a max-max algorithm, which belongs to the family of Expectation-Maximization algorithm. After initialization we assign each data point to one cluster by calculating the two norms between this data point and cluster centers, denoted as μ_k and assigning the data point to whichever cluster that has the smallest distance. After that we update cluster centers μ_k by taking the average of all datum that had been assigned to this cluster. This procedure is continued until convergence- whenever there is no more reassignments. In practical, this loop breaks as soon as R is not changing anymore.

3

Implementation of K-mean Clustering in Scikit-learn

In this chapter, we'll dive into **Scikit-learn K_Means** implementation and explore what kind of strategies they utilized to optimize the algorithm to achieve higher computational performance. We'll summarize those optimization into 4 categories, namely Sparse matrix, Cythonization, Parallelization, and Mini-Batch.

1. Sparse Matrix

In scikit-learn k-means implementation, they always transform input data matrix (in the size of features and observations) into scipy sparse CSR matrix representation, which leads to substantial memory reductions by storing only the non-zero entries. Therefore it results significant expedition in the computational time when doing matrix operations (multiplication, division and power, etc.).

Let's see some examples that scikit-learn k-means utilized sparse matrix.

Example : Euclidean distances
In KMeans Algorithm, every iteration, distances between every point and different cluster's mean have to be computed to update the membership of points. Therefore, distance computation is the most frequently executed part. Thus scikit-learn k-means used csr matrix multiplication when computing the distance.

For efficiency reasons, the euclidean distance between a pair of row vector x and y is computed as

$$dist(x, y) = sqrt(dot(x, x) - 2 * dot(x, y) + dot(y, y))$$

Note that in this formula, there's heavy matrix multiplication so using sparse can greatly enhance the efficiency. The code below explains how the sparse matrix dot product is implemented.

```
1 def euclidean_distances(X, Y=None, Y_norm_squared=None,
    squared=False,X_norm_squared=None):
2     ...
3     distances = safe_sparse_dot(X, Y.T, dense_output=True)
4     YY = row_norms(Y, squared=True)[np.newaxis, :]
5     ...
```

Listing 3.1: Sparse matrix Multiplication computing Euclidean distance

2. Cythonization

(a) Implementation for the sparse CSR representation is completely written in cython, which we believe it helps great deal of speedup. Below, we list major 4 categories of cdef variables.

- C arrays

```
1 cdef:
2     np.ndarray[DOUBLE, ndim=1] X_data = X.data
3     np.ndarray[int, ndim=1] X_indices = X.indices
```

```
    np.ndarray[int, ndim=1] X_indptr = X.indptr

```

- Counter variables

```
cdef:
    unsigned int n_samples = X.shape[0]
    unsigned int n_clusters = centers.shape[0]
    unsigned int n_features = centers.shape[1]
    int old_count, new_count

```

- indices

```
cdef:
    unsigned int sample_idx, center_idx, feature_idx
    unsigned int k

```

- Loop variables

```
cdef int i, j, c
    int old_count, new_count

```

(b) Sample code for Cython

Here, we chose mini batch update function to demonstrate how scikit-learn k-means optimization is achieved using cython.

```
@cython.boundscheck(False)
@cython.wraparound(False)
@cython.cdivision(True)
def _mini_batch_update_csr(...):
    """Incremental update of the centers for sparse
    MiniBatchKMeans."""
    ...
    for center_idx in range(n_clusters):
        for sample_idx in range(n_samples):
```

```
            if nearest_center[sample_idx] == center_idx:
                new_count += 1
    ...
```

Note that in this mini batch update function, it reassigns the membership of each data point. To prevent the global count value being modified by multiple threads at the same time, scikit-learn kmean keeps the GIL (Global Interpreter Lock), and instead of parallelization, it uses cythonized function to optimize naive loop iteration.

3. Parallelization

Addition to Cythonized part, scikit-learn does utilize python Parallel module for further speedup. In fact the main function k-means is also implemented in parallel. For instance, k-means iterate several times until the means of clusters converges. So they defined **_kmeans_single** function and run under multi-threads setting.

Before we take a look at how parallel approach is written in k-means, let's first understand **n_jobs** parameter.

> n_jobs : int The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.
>
> - If -1 all CPUs are used.
> - If 1 is given, no parallel computing code is used at all, which is useful for debugging.
> - For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

```
def k_means(X, n_clusters, init='k-means++', n_jobs=1, ...):
# parallelisation of k-means runs
    results = Parallel(n_jobs=n_jobs, verbose=0)(
        delayed(_kmeans_single)(X, n_clusters,
        max_iter=max_iter,
        init=init, verbose=verbose, tol=tol,
```

```
        precompute_distances=precompute_distances,
        x_squared_norms=x_squared_norms,
        # Change seed to ensure variety
        random_state=seed)
        for seed in seeds)
```

4

Compare Python Implementation and Scikit-learn Implementation

In order to quantify and calibrate the performance improvement of Scikit-learn implementation, we timed the execution time of both algorithms by running them against the MNIST dataset. You can learn more about it at http://yann.lecun.com/exdb/mnist/. The MNIST task is widely used in supervised learning, and modern algorithms with neural networks do very well on this task. We can also use MNIST for interesting unsupervised tasks. We have representations of 6000 MNIST images, each of which are 28×28 handwritten digits.

We first compared the execution time of the two algorithm against data set with different sizes. As we can see in Figure 4.0.1, the Scikit-learn

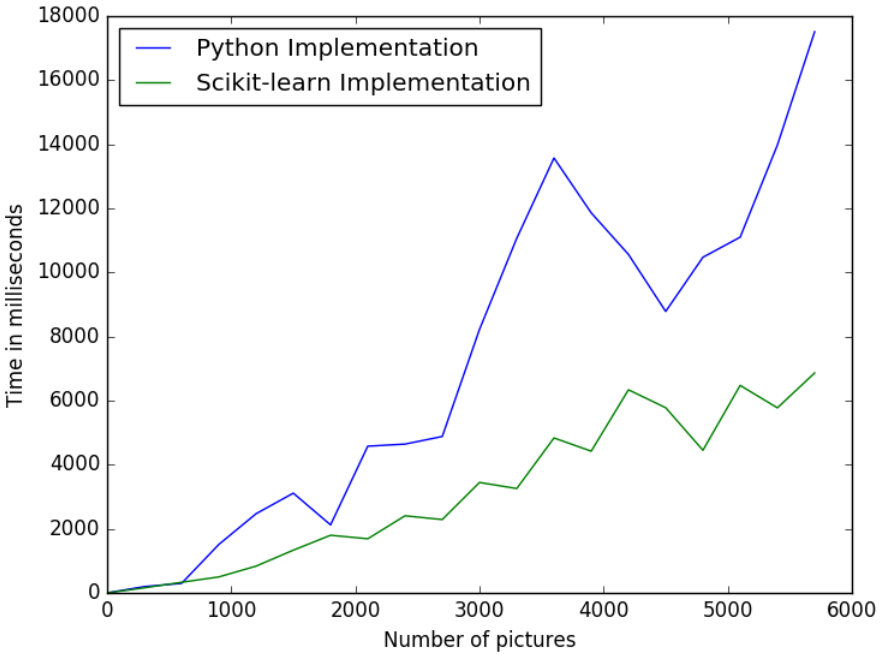

Figura 4.0.1: Execution Time v.s. Number of Pictures

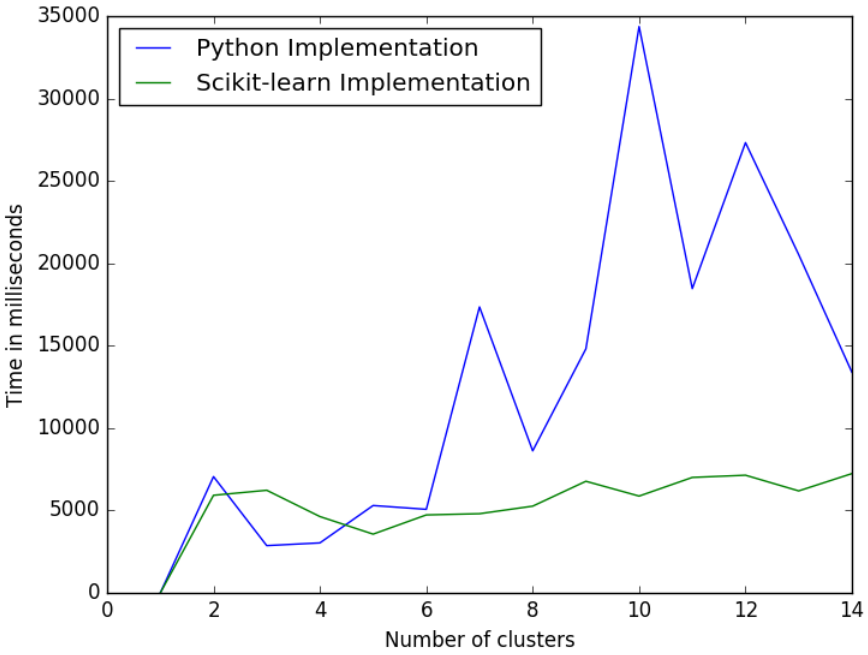

Figura 4.0.2: Execution Time v.s. Number of Clusters

implementation is generally faster than our Python implementation, and the difference becomes more obvious as we increase the data size.

The second set of comparison of execution time is performed by fixing the data size while varying the number of clusters. As we can see our implementation outperforms the Scikit-learn implementation for small number of clusters (Figure 4.0.2), which is not surprising, as the true number of clusters should be 10. (The data set contains pictures of digits form 0 to 9, hence 10 clusters.) Predictably, the most significant different is observed when the number of clusters is 10.

On average the Scikit-learn algorithm is twice as fast as our Python implementation, due to the optimizations mentioned in last chapter.

Bibliografía