High Performance K-Means Clustering

February 20, 2022

1 The Basic Idea Behind K-Means

In this quick note, we are going to show an implementation of the K-Means algorithm in Python with high performance. Why? Because different implementations of KMeans will make a huge difference to the computing cost and naive codes without efficiency will ruin it all. So before we dive in the coding part, let's first review K-Means as beginners. If you have already known this the algorithm behind K-Means, you may want to skip this introduction.

1.1 Introduction of KMeans

K-Means is a unsupervised clustering algorithm that is generally used for classification. It select clusters based on the Euclidean distance between different points. The centroid, which is defined as the center of a cluster, is actually calculated by averaging the mean value of the points in a certain cluster. Because I have assumed that you may heard about KMeans somewhere, so it may be better for you if you have the following pseudocode.

Algorithm: *kmeans(X,k)*

Select k unique points from X as initial centroids $m_{1..k}^{(t=0)}$ for clusters $C_{1..k}^{(t=0)}$
repeat
  foreach $x \in X$ **do**
    $j^* = \arg\min_j distance(x, m_j^{(t)})$ (*find closest centroid to* x)
    Add x to cluster $C_{j^*}^{(t+1)}$ (*assign* x *to cluster*)
  end
  for $j = 1..k$ **do**
    $m_j^{(t+1)} = \frac{1}{|C_j^{(t+1)}|} \sum_{x \in C_j^{(t+1)}} x$ (*recompute centroids*)
  end
  $t = t + 1$
until $C_{1..k}^{(t)} = C_{1..k}^{(t-1)}$ (*until clusters don't change*)

1.2 Centroids Initialization

Before we train our model with iterations, we need to choose our initial centroids. In the present note, I would like to introduce three different apporaches for choosing the initial centroids.

1.2.1 Randomly Choosing

The random approach is to randomly start with *k* different points from our given dataset and then use them as a set of starting centroids. This approach is quick for coding but it may take time for a training set to converge.

1.2.2 Naive Selection

A better but naive approach is that we can add a rule for selecting the next centroid so that it can be easier for us to get a converged result. Simply speaking, this naive selection approach is to get the select the next initial centroid based on the distance summerization to all the existing centroids. If we select the point which has a longer distance to all the pre-existing clusters, this points is more likely to be belonged to a new cluster.

1.2.3 Kmeans++

The Kmeans++ is a better idea compared with naive selection approach because the average will ignore some information we need. In the naive selection, it may be possible that we select a point as centroid which is close to some clusters but very far away to some other clusters. However, Kmeans++ solve this problem by seleting the next centroid maxize the minimizing the minimum distance to all the existing clusters so that we will not fall into the "average" trap.

1.3 A Traditional Approach Vs. Numpy

In the pseudocode above, we have seen that the traditional way for KMeans is to traverse all the points and update in one iteration. It will be of course very expensive if we implement this algorithm and the time complexity can grow up to $O(n^3)$, which is definitly the worst case we would like to avoid.

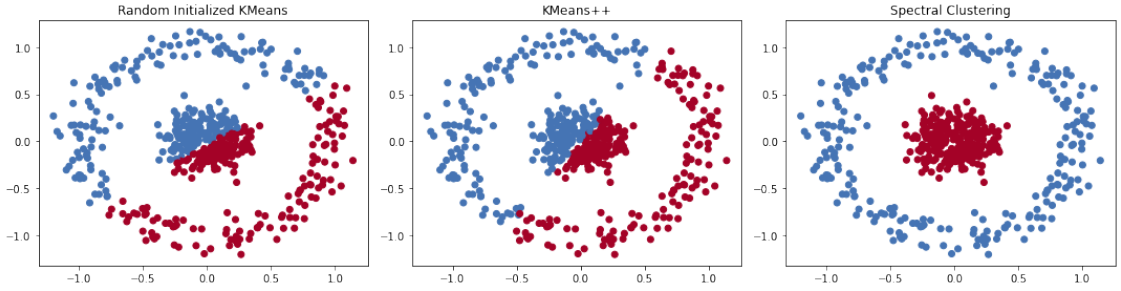
But with the help of Numpy, we can solve this problem with matrics, or with vectorization. We will talk later about how to implement the code in the next section. Here, we just give a brief introduction telling you that Numpy can have much much better performance than the traditional looping.

1.4 Limitations of KMeans

Although it seems pretty good for KMeans because we don't have to label anything before classification, it is unsupervised and it can generate some unwanted classification that can be easily diagnosed. The most famous case here is the nested distribution case where the data points are not classified as what we have expected. For example, in the following figures, both randomly started Kmeans and Kmeans++ yields to a wrong classification butspectral clustering actually do a better job. We will cover more about this part in the last section.

[61]:
```
from kmeans import *

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

X, _ = make_circles(n_samples=500, noise=0.1, factor=.2)

centroids, labels = kmeans(X, 2)
colors=np.array(['#4574B4','#A40227'])
axes[0].scatter(X[:,0], X[:,1], c=colors[labels])
axes[0].set_title("Random Initialized KMeans")

centroids, labels = kmeans(X, 2, centroids="kmeans++")
colors=np.array(['#4574B4','#A40227'])
axes[1].scatter(X[:,0], X[:,1], c=colors[labels])
axes[1].set_title("KMeans++")

cluster = SpectralClustering(n_clusters=2, affinity="nearest_neighbors")
labels = cluster.fit_predict(X)
colors=np.array(['#4574B4','#A40227'])
axes[2].scatter(X[:,0], X[:,1], c=colors[labels])
```

```
axes[2].set_title("Spectral Clustering")

plt.tight_layout()
plt.show()
```



2 Time Complexity for Different K-Means Implementations

2.1 Brute-Force Naive K-Means Selection

We have discussed that the naive selection of initial centroids is to select the point which has the longest summerized distance. However, if we use a solution of looping, and suppose we have k clusters with n records, we will have a time complexity of $O(n^3)$ in the worst case. This is not preferred and it should be improved.

```
for i in range(1, k):
    distances = [0] * len(X)
    for index, x in enumerate(X):
        for centroid in newCentroids:
            distances[index] += distance(x.astype(int), centroid.astype(int))
    newCentroids.append(X[np.argmax(distances)])
    X = np.concatenate((X[:np.argmax(distances)], X[np.argmax(distances) + 1:]), axis=0)
```

2.2 Naive K-Means Selection Vectorization

Instead of looping through the whole set, we can use a vectorized approach with the help from `Numpy`.

```
for i in range(1, k):
    distances = np.zeros(len(X))
    for centroid in newCentroids:
        distances += distance(centroid, X, byrow=True)
    index = np.argmax(distances)
    newCentroids.append(X[index])
    X = np.concatenate((X[:index], X[index + 1:]), axis=0)
```

In this case, function `distance` is called to calculate the current centroid with all the X in the set. Because `distance` is implemented using vectors, it takes $O(1)$ for computing this result and this gives us a better performance.

```
distance = np.sqrt(((a - b) ** 2).sum(1))
```

2.3 Brute-Force Clustering

For clustering and labeling in each iteration, the worst case is to use a brute-force approach, which means to loop through all the points in `X` and for each `x`, we need to loop through all the centroids for calculating the correct label. This process is very expensive with $O(n^2)$ complexity, and we can make it faster by using vectorizations.

```
for x in X:
    distances = []
    for centroid in centroids:
        distances.append(distance(x, centroid))
    index = int(np.argmin(distances))
    labels.append(index)
    clusters[index].append(x)
```

2.4 A Slightly Better Clustering Approach

Here is a result if we leverage some linear algebra to our code but this implementation still takes $O(n)$ complexity. This means it is good but not good enough, and we can still find a better way to improve this.

```
for x in X:
    distances = distance(x, np.array(centroids), byrow=True)
    index = int(np.argmin(distances))
    labels.append(index)
    clusters[index].append(x)
```

2.5 Vectorized Clustering

The best solution for this clusting phase only takes $O(1)$ complexity and it saves us all the dirty work for loops. The only trick we need here is some matrix transformations and then we will get the distance we want.

```
dist_matrix = distance(X, np.array(centroids), byall=True)
labels = np.argmin(dist_matrix, axis=1)
```

Note that the `distance` function is called again here but using a different argument `byall`. This will call the following linear algebra magics for calculating the distance matrix in a clean and beautiful way.

```
dist = (a[:, np.newaxis] - b) ** 2
dist = np.sqrt(dist.reshape(len(a) * len(b), a.shape[1]).sum(1))
dist = dist.reshape(len(a), len(b))
```

2.6 Brute-Force Recalculating

Now that we have all the clusters and labels, and what we have to do is to recalculate the centroids. This approach takes a time complexity of $O(n^2)$ and it is getting even slower on a big input set like graph processing. So it may not be a good idea to do so.

```
for i in range(k):
    cluster = clusters[i]
    if not cluster:
        continue
    nNodes = len(cluster)
    nodeSum = int(cluster[0].copy()[0])
    for node in cluster[1:]:
        nodeSum += int(node.copy()[0])
```

```
    newCentroid = nodeSum / nNodes
    newCentroids[i] = newCentroid
```

2.7 Vectorized Recalculating

Using a vectorized recalculating for generating new centroids is pretty elegent and we can directly use `np.mean` for getting the new centroids.

```
for i, cluster in enumerate(clusters):
    if cluster:
        newCentroids[i] = np.mean(cluster, axis=0).copy()
    else:
        continue
```

2.8 K-Means++ with Vectorizations

I didn't implement the brute-force attempt for Kmeans++ because we already have the convinent function `distance` with argument `byall`, so why bothering? The other problem is that it will be overly complex if we implement kmeans++ using loops and this will be super expensive.

```
for i in range(1, k):
    dist_matrix = distance(X, np.array(newCentroids), byall=True)
    index = np.argmax(dist_matrix.min(axis=1))
    newCentroids.append(X[index])
    X = np.concatenate((X[:index], X[index + 1:]), axis=0)
```

3 Use Cases for Testing

3.1 1D Classification: Grades

After seeing all these programs, let's now apply the model to some applications. The first example here is to classify a 1D array of grads and it takes less than 1s to complete.

[55]:
```
grades = [92.65, 93.87, 74.06, 86.94, 92.26, 94.46, 92.94, 80.65, 92.86,
          85.94, 91.79, 95.23, 85.37, 87.85, 87.71, 93.03]
k = 3

grades = np.array(grades).reshape(-1,1)

start = time.time()
kmeans(grades, k)
end = time.time()
print(f"The time for training is {end - start}")
```

```
The time for training is 0.0013420581817626953
```

3.2 2D Classification: Compressing Grayscales

Kmeans can also be used for compressing images. Here we can apply it on a grayscale, which can be treated as a 2D matrix, and the result will be a compressed image. It takes less than 3 seconds to finish this task on my computer.

[56]:
```
fig, ax = plt.subplots(1,2,figsize=(15,5))

image = Image.open("north-africa-1940s-grey.png")
```

```
X = np.array(image)

ax[0].imshow(X, cmap='gray')
ax[0].axis('off')
ax[0].set_title("Original Picture")

h, w = X.shape
X = X.flatten()
X = X.reshape(-1,1)

k=4

start = time.time()
centroids, labels = kmeans(X, k=k, centroids='kmeans++', tolerance=.01, verbose=False,␣
↪max_iter=30)
end = time.time()
print(f"The time for training is {end - start}")

centroids = centroids.astype(np.uint8)
X = centroids[labels]

ax[1].imshow(X.reshape(h,w), cmap='gray')
ax[1].axis('off')
ax[1].set_title("Compressed Picture")

plt.tight_layout()
plt.show()
```

```
The time for training is 2.5313918590545654
```

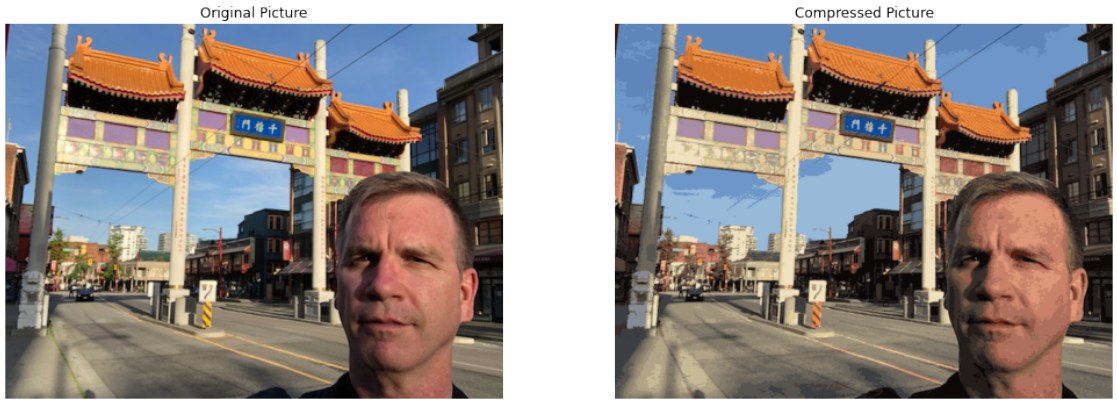
3.3 3D Classification: Compressing RGB Pictures

We can also test our model on some more complicated examples like a RGB picture. For training this 3D array, the result takes about 25 seconds to get to the maximum iteration number.

[57]:
```
fig, ax = plt.subplots(1,2,figsize=(15,5))

image = Image.open("parrt-vancouver.jpg")
```

```
X = np.array(image)

ax[0].imshow(X)
ax[0].axis('off')
ax[0].set_title("Original Picture")

h, w, n = X.shape
X = X.reshape(h*w, n)

k=32

start = time.time()
centroids, labels = kmeans(X, k=k, centroids='kmeans++', tolerance=.01, verbose=False,␣
↪max_iter=30)
end = time.time()
print(f"The time for training is {end - start}")

centroids = centroids.astype(np.uint8)
X = centroids[labels]
X = X.reshape(h, w, n)

ax[1].imshow(X)
ax[1].axis('off')
ax[1].set_title("Compressed Picture")

plt.tight_layout()
plt.show()
```

```
The time for training is 25.74117612838745
```

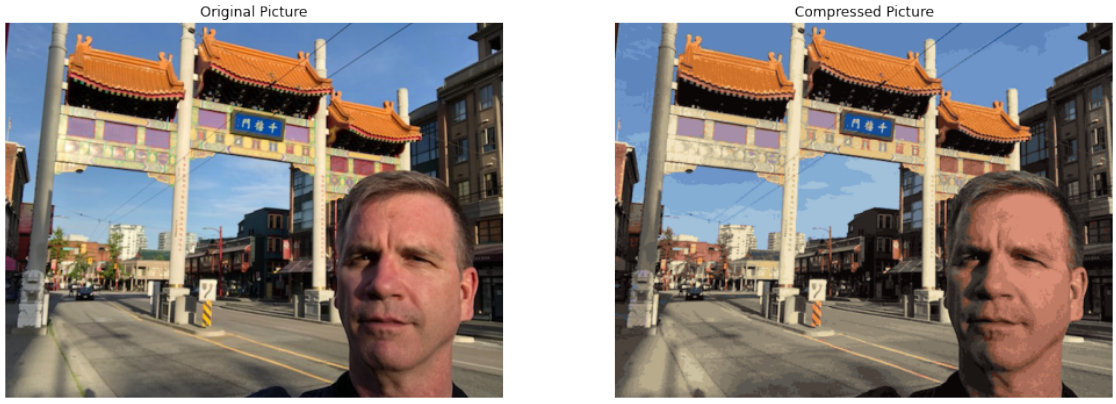


3.4 Benchmark: How Fast is Sklearn?

We can also compare our implementation with Sklearn. It turns out that Sklearn do have a better performance than us with only 1/3 of the time cost. But becasue there's not a huge difference, our model can still be considered as a high performance one.

[58]:
```
fig, ax = plt.subplots(1,2,figsize=(15,5))

image = Image.open("parrt-vancouver.jpg")
X = np.array(image)

ax[0].imshow(X)
ax[0].axis('off')
ax[0].set_title("Original Picture")

h, w, n = X.shape
X = X.reshape(h*w, n)
start = time.time()
kmeans = KMeans(n_clusters=32, init='k-means++', verbose=0, max_iter=30, tol=.01).
 ↪fit(X)
end = time.time()
print(f"The time for training is {end - start}")

centroids = kmeans.cluster_centers_.astype(np.uint8)
X = centroids[kmeans.predict(X)]
X = X.reshape(h, w, 3)

ax[1].imshow(X)
ax[1].axis('off')
ax[1].set_title("Compressed Picture")

plt.tight_layout()
plt.show()
```

```
The time for training is 7.008099794387817
```



4 Limitions of K-Means

4.1 Why K-Means is NOT Good Enough?

Because Kmeans is a clustering without any domain knowledge, it does not guarantee the local optimium after training, and it is provably going to find a local minimum. So the initial values does matter if we want to find a global minimum and that's why we leverage kmeans++ for initialization.

Another concern as we have explained in the beginning, Kmeans does not work for nested distributions and we may probably want to use spectral clustering alternatively.

Also, when there is categorical data or the number of dimensions gets very high, Educlidean distances are not proper for clustering. Instead, we may use mechanism of nearest neighbors or some similarity matrix trained from random forest. We will talk about this in the next part.

4.2 Breiman's Trick for Unsupervised Random Forest

So now here's our problem. For now we have a set of data which can be used to contruct similarity matrix. However, it is not easy to use random forest for constructing the similarities because we don't have labels. A trick developed by Breiman is to duplicate and bootstrap columns of the inputs X to get X' and then create labels y to distinguish X from X'. After that, a random forest model can be applyed on the stacked $[X, X']$ for fitting the target labels y. Finally, we can walk through all the leaves and check the pairs in the leaf for getting the proximities.

To get X' from X, we have,

```
def df_scramble(X: pd.DataFrame) -> pd.DataFrame:
    X_rand = X.copy()
    for colname in X:
        X_rand[colname] = np.random.choice(X[colname], len(X), replace=True)
    return X_rand
```

Then to generate y stacked $[X, X']$, we have,

```
def conjure_twoclass(X: np.ndarray) -> (np.ndarray, np.ndarray):
    X = pd.DataFrame(X)
    X_rand = df_scramble(X)
    X_synth = pd.concat([X, X_rand], axis=0)
    y_synth = np.concatenate([np.zeros(len(X)),
    np.ones(len(X_rand))], axis=0)
    return np.array(X_synth), np.array(pd.Series(y_synth))
```

In order to traverse all the leaves, we can use the provided function called `leaf_samples`,

```
def leaf_samples(rf, X: np.ndarray):
    n_trees = len(rf.estimators_)
    leaf_samples = []
    leaf_ids = rf.apply(X)  # which leaf does each X_i go to for sole tree?
    for t in range(n_trees):
        # Group by id and return sample indexes
        uniq_ids = np.unique(leaf_ids[:,t])
        sample_idxs_in_leaves = [np.where(leaf_ids[:, t] == id)[0] for id in uniq_ids]
        leaf_samples.extend(sample_idxs_in_leaves)
    return leaf_samples
```

In the end, the similarity matrix can be generated by,

```
def similarity_matrix(X):

    proximity_matrix = np.zeros((len(X), len(X)))

    X_scramble, y_scramble = conjure_twoclass(X)
    rf = RandomForestClassifier(max_depth=2)
    rf.fit(X_scramble, y_scramble)
    leaves = leaf_samples(rf, X)
    for leaf in leaves:
```

```
            combines = combinations(leaf, 2)
            for i, j in combines:
                proximity_matrix[i][j] += 1
                proximity_matrix[j][i] += 1
    sim_matrix = proximity_matrix / len(leaves)

    return sim_matrix
```

Note that the distance matrix would be 1 - similarity matrix.

4.3 Use Case: Cancer Classification

With the cancer data given, we can use Kmeans++ for classification. And the confusion matrix seems as follows.

[62]:
```
cancer = load_breast_cancer()
X = cancer.data
y = cancer.target

sc = StandardScaler()
X = sc.fit_transform(X)

centroids, labels = kmeans(X, k=2, centroids="kmeans++", tolerance=0.01)
likely_confusion_matrix(labels, y)
```

```
         pred F  pred T
Truth
F        175     37
T        13      344
clustering accur 0.9121265377855887
```

Using the similarity matrix from X, we can call `SpectralClustering` function for clustering and this approach here will give a similar result.

[64]:
```
S = similarity_matrix(X) # breiman's trick
cluster = SpectralClustering(n_clusters=2, affinity='precomputed')
labels = cluster.fit_predict(S) # pass similarity matrix not X

likely_confusion_matrix(labels, y)
```

```
         pred F  pred T
Truth
F        200     12
T        44      313
clustering accur 0.9015817223198594
```