Clustering Optimization using k-means

Course: Optimization, 88784-01
Lecturer: Prof. Gil Ariel
submitters:
Shoham Yamin, 319151213
Haim Lavi, 038712105

Github repository: https://github.com/HaimL76/clustering.git

Preface

In this paper, we shall demonstrate two approaches for optimizing the k-means algorithm for clustering.

The problem of clustering is a good example of an optimization problem. The problem is, first, how to divide a set of n elements into a given number of k groups ($k << n$). We have many ways to do this, and we need to find an optimal grouping, assigning to each group its best matching elements.

Moreover, in real-life problems, k itself is unknown and needs to be found by the machine itself. We can theoretically declare the whole set of elements as one cluster, or consider each element as its own cluster, but that would be pointless. Thus, we need to find a way to optimize the choice of the number k, assuming the first problem itself (finding the optimal grouping, for a given k) has an optimal solution.

There are two kinds of clustering that we would like to consider, which may affect the algorithm that we run,

1. Clustering of scattered details

In this kind of clustering, we try to identify clusters from details that are scattered along the image. This can have various applications, such as **Military Intelligence** (clusters of troops, aircraft, armored vehicles), **Nature Science** (clustered structures of birds, insects, fish), and more.

2. Identification of objects as clusters

In this kind of clustering, we try to identify large objects, from samples found in the image. A major example of this kind of clustering would be face recognition, where we have many features in some image, and we try to isolate and identify the face of a person, or of several people.

Definition of the problem

First, we shall observe the obvious fact that there is a finite number of choosing k clusters out of n elements.
In fact, this number is known as a **Stirling number of the second kind** [3], it is marked as $\left\{ {n \atop k} \right\}$ and is given by the formula
$S(n,k) = \left\{ {n \atop k} \right\} := \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n$
Moreover, if we are not given the value of k, then the total number of options, to choose any $1 \leq k \leq n$ from n, is known as the n^{th} **Bell number**, and is given by $B_n := \sum_{k=0}^{n} \left\{ {n \atop k} \right\}$

Obviously, we need to find the optimal partition of the set of n elements, and, if we cannot find the absolute optimum, we would have to find a local optimum partition, which will give us a reasonable clustering.

So, given a set, S, of n elements, and looking from the view of topology, we always have the trivial topology $\{\varnothing, S\}$, as well as the discrete topology, where each element is an open set. The same goes for clustering, obviously, we always have trivial partitions, meaning either the whole set of elements is considered as one single cluster, or every single element is considered a cluster on its own, both, clearly, being useless.
So, we need some quantitative indicator, for each k clustering, to measure the quality of our clustering algorithms.

In addition, we want to set this score for each possible k, so we can find the optimal k, for a given set of n elements. The basic concept is to run from some starting number, $k_0 \leq k \leq n$, until we find an optimal k (if we have reached $k = n$, then clearly the algorithm is wrong).

Within-Cluster Sum of Square

One of the well-known clustering quality indicators is the **Within-Cluster Sum of Square (WCSS)** [4]

which means, we calculate the squared distances of all the elements from their associated clusters,

That is (using $w(k)$ instead of WCSS(k)),

$w(k) := \sum_{j=1}^{k} \sum_{i=1}^{m_j} \|c_j - p_{j_i}\|^2 = \sum_{j=1}^{k} \sum_{i=1}^{m_j} ((x_j - x_{j_i})^2 + (y_j - y_{j_i})^2)$

Where

m_j - the number of elements associated to cluster j

$c_j = (x_j, y_j)$ - the center point of cluster j

$p_{j_i} = (x_{j_i}, y_{j_i})$ - the point i of cluster j

So, for a given n, our goal would be, allegedly, to find arg min(w(k)), which is a classical optimization problem.

However, this is not so accurate, since, at some point, we will start drifting away from the optimal k.

Indeed, as we compute clusters for higher and higher values of k, $w(k)$ will decrease more and more, without giving us any benefit, but actually ruining the clustering.

It is a trivial observation since when we reach $k = n$, that is, declare each element as a separate cluster, we have then $w(k = n) := \sum_{j=1}^{k} \sum_{i=1}^{m_j} 0 = \sum_{j=1}^{k} 0 = 0$, which is clearly the minimal $w(k)$, but obviously a useless clustering result.

We will see this computation in the classic approach that we are showing next. The other approach we are presenting will make use of a different calculation of the clustering score.

The basic k-means algorithm

One of the very well-known algorithms for clustering is the k-means algorithm.

This algorithm is based on a very simple concept of acquiring initial data, then adjusting this data until the algorithm stables. This algorithm is called k-means because we are trying to find well-scattered clusters, whose center points are the means of their associated elements, calculated, for each $c_j = (x_j, y_j) = (\{x_{j_i}\}_{i=1}^{m_j}, \{y_{j_i}\}_{i=1}^{m_j})$ as $\mu(c_j) = \mu(x_j, y_j) := (\frac{\sum_{i=1}^{m_j} x_{j_i}}{m_j}, \frac{\sum_{i=1}^{m_j} y_{j_i}}{m_j})$
We call each k-mean, a **centroid**.

The basic description of this algorithm, for a given k, is,

1. Initialization Initialize a set of k centroids within the pixels of the image.

2. Association For each element, calculate its distance from the center of each cluster, find the centroid with the minimal distance from this element, and associate the element to this centroid.

3. Recalculation Using the association of the elements, calculate the center point of each centroid again, by taking the mean point of all its associated elements.

4. Iteration Iterate steps 2 and 3 until the algorithm turns stable, that is, there are no more moves of associated elements between centroids.

Algorithm 1 Calculate k-means

Require: S: the input list of samples
k: the number of clusters to find
Ensure: C: the output list of centroids
$C \leftarrow$ InitializeCentroids(S) //**using some seeding method**
$r \leftarrow true$
$n \leftarrow size(S)$

while r is $true$ **do**
$a \leftarrow 0$ //**counter of moves of samples between centroids**
$i \leftarrow 0$
while $i < n$ **do**
$s \leftarrow S[i]$

$f \leftarrow null$ //**nearest centroid found for the current sample**
$m \leftarrow null$ //**distance from sample center of nearest centroid**

$j \leftarrow 0$
while $j < k$ **do**
$c \leftarrow C[j]$

$dx \leftarrow s.x - c.x$
$dy \leftarrow s.y - c.y$
$d2 \leftarrow dx^2 + dy^2$

if m is null **or** $d2 < m$ **then**
$m \leftarrow d2$
$f \leftarrow c$
end if

if $s.c \neq f$ **then**
$a \leftarrow a + 1$
end if

$j \leftarrow j + 1$
end while

$i \leftarrow i + 1$
end while

if $a < 1$ **then**
$r \leftarrow false$
end if

end while

This algorithm, as described, is promised to converge, that is, to achieve a stable state, where the stopping condition (no more moves between centroids) is satisfied.

The proof of convergence is given by the basic observation that the number of grouping options, for a given number of k clusters, out of n elements, is obviously finite, and that on each step, we get a better score on the clustering. A full proof can be found at [1]

Proposition The k-means algorithm does not necessarily give an optimal solution, for a given k

Explanation The given proof only proves that if we start from some initial setting of the system, we are sure to converge, at some point. However, this convergence is to a local minimum only, because, if we start from a different setting, we may converge to another local minimum, possibly to the best existing solution, which we shall refer to as the global minimum.

Seeding methods

As proved, the k-means algorithm is promised to converge to some local minimum, but it's not guaranteed to converge to the global minimum,
meaning, we do not necessarily obtain the best clustering for the input image.
One way to manipulate the algorithm to a better convergence is by using a more sophisticated way for the algorithm initialization, which means, the seeding of the initial centroids.
The first dilemma, regarding centroid seeding, is whether we should take as centroids only points from the list of elements, on which we perform the clustering or points from the whole range of the image pixels.
We shall review three different seeding methods, and relate them to these two general categories.
A highly detailed summary of different seeding methods can be found in [5].

1. Unified Scattering

In this method, we simply seed our initial centroids at equal distances, along the image pixels. This method does not consider the list of elements for the scattering, therefore, it is most efficient for images that have the elements scattered all over the image pixels, and not just densely concentrated in several specific locations.

We have successfully tried this method, in our algorithm, taking the initial centroids from the top left corner, in equal distances, to the bottom right corner.

2. Random Scattering

This method simply scatters the initial centroids randomly across the image. This could be randomly choosing k points from the list of elements, or from the image pixels. If we use the first, then we can simply take the k centroids in equal spaces, from the list of elements. So, if the list size is n, we take $\{i \cdot \frac{n}{k}\}_{i=1}^{k}$ elements from the list, which can be considered random enough.

3. k-means++

This method is actually doing pre-processing on the list of elements, in order to achieve an optimal initial scattering, which will result in obtaining an optimal clustering.

The algorithm description is,

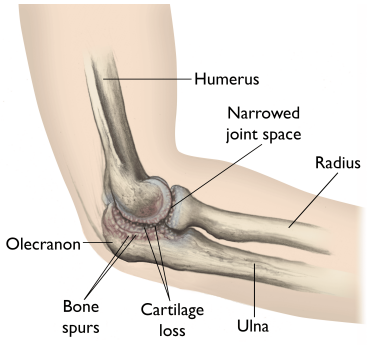
1. Choose a random point from the list of elements, and add it to the list of centroids.
2. Go over the list of elements, choose the element whose distance from the first point is the maximal, and add it to the list of centroids.
3. Go over the list of elements, choose the element whose distance from all the points in the list of centroids is the maximal, and add it to the list of centroids.
4. Repeat step 3, until the list of centroids has k centroids.

The Elbow method

Recall from above, one major problem that we have, with the k-means algorithm, is that the native algorithm requires an input number of k, for running. We also recall that the total number of ways to group n points to $1 \leq k \leq n$ clusters is given by the n^{th} Bell number, which is a sum of the k^{th} Stirling numbers, for each k, thus significantly large.

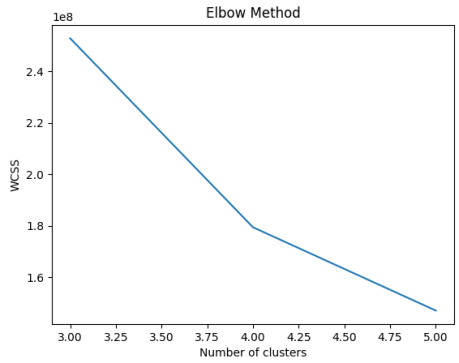
To automatically choose the optimal number of k, we need a way to compare the scores of different values of k, running under the same conditions. This brings us to an optimization method, called the **Elbow Method**. The basic concept of this method is that we can compute some score on each k, and then present this score as a function of k.
So, our natural choice, for this score function, would be the WCSS, described above.
However, as mentioned before, we cannot look for a minimal value of the WCSS, rather we look for an optimal value. Taking a range of k values, that is, $\{k_1, k_2, k_3, \ldots, k_m\}$, we shall observe that for the lower values of k, the function is decreasing rapidly, while after a certain value of k, the function is taking a significant turn, from a high (negative) slope, into a nearly asymptotic graph. This means that for less than the optimum k value, our clusters are too large, and for more than the optimum, the more k clusters we calculate, in the k-means algorithm running, we do not add any improvement for the clustering, but exactly the opposite, meaning the output clusters will split the real clusters in the image, and not give us any beneficial clustering information.

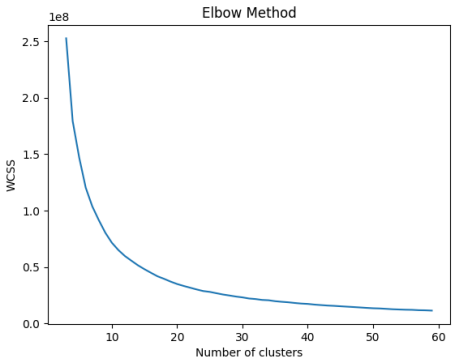
In other words, the optimal number of k is the turning point of the graph, from the high slope to the asymptote. This is why it is called "elbow" because it resembles a folded arm and the elbow that is the outmost point in the arm.



So, if we can compute different ranges of numbers, for different maximal k values, $\{km_1, km_2, km_3, \ldots, km_l\}$, and we get the same elbow (that is, a specific value of k), for each m_i, then we have the optimal number of k, for this image. We can even assume that the optimum will move up and down, but will maintain some boundaries, from which we can take the average k, with or without some weight or probability considerations.

We would like to have a clear elbow point, for each image, so we can have a simple algorithm to calculate it,
but in reality, this is hardly the case. For example, we compare two computations of the elbow method, on the same image,
one is computing a range of $\{3, 4, 5, 6\}$ clusters, while the other is computing a range of $\{3, 4, ..., 60\}$.





Both graphs do not have a minimal point, in the classic sense, but while the 6 clusters computation has a clear elbow point (at $k = 4$),
which can be computed by calculating, for instance,
$\arg\min \angle k_{i-1}k_ik_{i+1}, 0 \leq i \leq m$ (the angle between every two segments in the graph), the 60 clusters graph (schematically looking like exponential decay), is approximating a smooth function, and does not have a clear elbow point.

So, we need to find a way to compute the optimal elbow point, when the k is not given, obviously, and thus can theoretically get to $k = n$
Our approach, to this problem, is based on the following observation,
If we draw a straight line, A, between the first k and last k points $(k_0, w_0), (k_m, w_m)$, where $w_i = w(k_i)$
We can see that the elbow is the most distant point from this line (when the distance from a point (k_i, w_i) to A is given by $\min\{\|(k_i, w_i) - a\| : a \in A\}$

So, a simple way for us to calculate the most distant point, would be to rotate the set of points,
so the first and last k have the same value of y, and calculate the lowest y of all the k clusters that we have computed.
This can be achieved by rotating all the $(k_i, w(k_i))$ around $(k_0, w(k_0))$
This can be done simply by using a rotation matrix, of the form,

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

where $\theta = tan^{-1}(\frac{w_m - w_0}{k_m - k_0})$
So, for $1 \leq i \leq m$, mark (x'_i, y'_i) as (x_i, y_i) rotated by θ,
To be more accurate, since we want to rotate the whole graph around the first point
(so, after the rotation, both the first and the last point will have the same y' coordinate),

we need to bring $(k_0, w(k_0))$, to the origin, and the whole graph along with it.

So, the calculation will be,

$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x_i - x_0 \\ y_i - y_0 \end{bmatrix}$$

The enveloping algorithm

Now, we need to construct an algorithm that will automatically determine the optimal k values, by finding the elbow. So, basically, our algorithm outline will look like this,

1. set k, an initial number of desired clusters

2. calculate k centroids, using the k-means algorithm from above

3. calculate $w(k)$, if it is the optimal value, return k

4. if $k = n$, return k. otherwise, increment k, and repeat step 3.

So, this algorithm can run, theoretically, until $k = n$.

However, we can observe that this is not exactly the desired algorithm, because,

how do we know that we have achieved the elbow, that is, the optimal value?

So, we edit this algorithm in a slightly different manner,

1. set s, the number of successful optimal k calculations

2. set t, the tolerance for optimal k values.

3. set q, a number representing a quant of k values to calculate

4. set k, an initial number of desired clusters

5. calculate k centroids, using the k-means algorithm from above

6. calculate $w(k)$, and store it in an array

7. if k=0 mod q, calculate the optimal k using all the stored WCSS values and store it in an array

8. if the array of optimal k values has s identical values of k, or if their difference is in the range of t, we take their value (in case they are identical), or some average between them, and return this value as the final value of k (the elbow).

9. if $k = n$, return k. otherwise, increment k, and repeat step 5.

Algorithm 2 Calculate elbow for k-means

Require:
q: quant of k for optimum calculations
s: number of successful elbow calculations
t: tolerance of k value
k_0: the k to start from
S: the list of samples
Ensure: k_1 optimal elbow value
$C \leftarrow null$
$k \leftarrow k_0$

$wcss \leftarrow$ float[]
$opts \leftarrow$ integer[]
$r \leftarrow true$

while r is $true$ and $k < n$ **do**
$C \leftarrow$ k-means(S, k)

$wcss[k] \leftarrow$ CalcWcss(C)

if $k \bmod q = 0$ **then**
$m \leftarrow null$
$i \leftarrow 0$
while $i < k$ **do**
$w \leftarrow wcss[i]$

if $m = null$ or $m > w$ **then**
$m \leftarrow w$
end if

$i \leftarrow i + 1$
end while
$opts \leftarrow$ append$(opts, i)$
end if
//**From the array of the elbows we have found so far,**
//**find all the identical values, up to** t**, the tolerance**
$o \leftarrow$ GetIdentical$(opts, t)$
if size$(o) = s$ **then**
$r \leftarrow false$
else
$k \leftarrow k + 1$
end if

end while
return k_1=GetAverage$(opts, t)$

A statistical k-means hybrid algorithm

Our approach is a hybrid method, which is using the basic k-means algorithm, together with statistical considerations.

The basic concept of our method is making practical use of the **Central Limit Theorem (CLT)**.

CLT is a well-known fundamental theorem in probability and statistics, which says that given a sequence of n random variables, $\{X_i\}_{i=1}^n$, with mean μ and variance σ^2, we mark the average $\bar{X_n} := \frac{\sum_{i=1}^n X_i}{n}$, then,
the sequence $\frac{\bar{X_n}-\mu}{\sigma/\sqrt{n}}$ converges in distribution to the normal (Gaussian) distribution, with μ and σ^2 as parameters.

This theorem, although appearing abstract, since we are referring to the convergence of the sequence as n tends to infinity, is actually very useful in applications, since we practically do not need a significantly large n, but it works also on small values of n (see, for example, [6]).

This allows us to utilize CLT for our purposes. We can run the basic k-means algorithm on the elements, but then we can automatically split clusters by a very basic statistical observation.

The normal distribution, in statistics, is used for calculating, for each random variable a score, which is called a **Z-score** or a **standard score** [7]. This means that we have a linear scale, for measuring the distance of each random variable from the mean, in units of standard deviation, and there exists a formula, which gives us the probability for each Z-score. For each sample, X_i, its Z-score is $Z_i := \frac{X_i-\mu}{\sigma}$. The probability, $\mathbb{P}(Z_i < z)$, can be found in a standard table of already calculated values.
Also, we will be interested in calculating the absolute value of the Z-score, since, for our specific problem, there is no meaning to the question if the sample is located "before" or "after" the center point of the centroid, so there is no difference between $Z_i = \frac{X_i-\mu}{\sigma}$ and -$Z_i = \frac{\mu-X_i}{\sigma}$
Indeed, since we are obviously using the standard Euclidean distance, in our

calculations, then the distance of any sample from the mean must be positive.
And so, if we say that a specific Z-score is higher or lower, we shall refer to its absolute value.

Thus, the algorithm, after combining this statistical calculation, is,

1. set an initial value of k, which can be quite small.

2. associate all the elements to their closest centroid.

3. for each centroid, calculate its mean, std, and the Z-score of each associated element.

4. for each centroid, count the number of elements that have a Z-score higher than z_0. If the count is higher than n_0, create a new centroid, and associate all these elements to this centroid. If the count is lower than n_0, then check the "ignore" flag. If it is true, then remove the association of these elements to the current centroid. If it is false, then keep these elements associated with the current centroid.

5. for each centroid, calculate its center point again, as the mean of all the associated elements.

6. while the number of clusters is increasing, or associated elements are still moving from one cluster to another, repeat step 2.

7. (after the algorithm turned stable) returns the current list of centroids.

Proposition The algorithm above is converging to a local minimum.

Explanation This is trivial since each iteration is computing the k-means algorithm, for a given k, and this algorithm is promised to converge to a local minimum, as stated above.

The splitting of the clusters, using the statistical criteria, must obviously stop, at some point, because, theoretically, we can split the clusters until each element is declared a separate cluster (in this case, we will get a convergence, but the whole clustering will be, of course, useless). However, using the minimal size of a cluster we are setting for the algorithm, the splitting is expected to stop when the resulting clustering cannot be split anymore, thus, yielding a local minimum (optimum), coming from the constraint on the minimal cluster size, and from the convergence of the k-means algorithm.

Algorithm 3 Calculate k-means with Z-score

Require:
s_0: the minimal count of samples in a cluster (the default is 30)
z_0: the maximal allowed Z-score for a cluster (the default is 3)
k_0: the k to start from
S: the list of samples
Ensure: C: the list of clusters
$r \leftarrow true$

$C \leftarrow$ InitializeCentroids(S) //**using some seeding method**
while r is true **do**
$C \leftarrow$ k-means(S, k)
$l \leftarrow size(C)$
$j \leftarrow 0$
while $j < k$ **do**
$c \leftarrow C[j]$
$i \leftarrow 0$
$c_0 \leftarrow sample[]$ //**array for the distant samples of centroid**
while $i < size(c.samples)$ **do**
$s \leftarrow c.samples[i]$
$z \leftarrow$ Z-score(c, s)
if $z < z_0$ **then**
remove($c.samples, s$)
append(c_0, s)
end if
$i \leftarrow i + 1$
end while
if size(c_0) $> s_0$ **then**
append($C, centroid(c_0)$)
end if
$j \leftarrow j + 1$
end while
//**if no new centroid was added, then stop the algorithm**
if $size(C) = l$ **then**
$r \leftarrow false$
end if
end while
return C

Our work summary

So far, we have described two approaches for the clustering problem, using the k-means algorithm, in its pure form, or in a wider, more hybrid solution. We shall now summarize exactly what we did in the code, and then show the results of running both algorithms, and compare them.

native k-means and the elbow method

In this function, we used the well-known implementation of the k-means algorithm, in the sklearn library, and added a wrapper code, in order to run the k-means in a loop (over a range of k values), and calculate the optimal elbow.

As we will display in the results, our calculation of the optimal k (elbow) shows that for larger and larger ranges of k values, the elbow keeps increasing. This means, allegedly, that we cannot use this computation for a real-life automatic clustering solution, because the code cannot determine the elbow just by computing it.

However, the elbow is increasing relatively slowly, in relation to the size of the range of k, thus, we have considered trying to use some threshold on the rate of elbow increase, in order to declare the optimal elbow value, once the increase rate goes lower than this given threshold.

Another way we have considered is using a library, for computing the elbow, the same way we use the sklean k-means implementation.

As a demonstration (not appearing in the final code), we have used the yellowbrick library elbow visualizer, in order to display their computed elbow for different ranges of k. We actually got similar results to our computation (which is described above), meaning this library implementation also computes elbow values that are increasingly higher and higher.

At this point, we have designed and implemented our own solution, as described above.

hybrid k-means with standard deviation

In this function, we have implemented everything ourselves, including the k-means algorithm itself.

The reason, obviously, is that we wanted to try different ways to combine the standard deviation computation in the k-means algorithm itself, for example, by splitting the clusters, using the Z-score, after each centroid calculation, and not only after the stabilization of the k-means algorithm, for each value of k

The main function

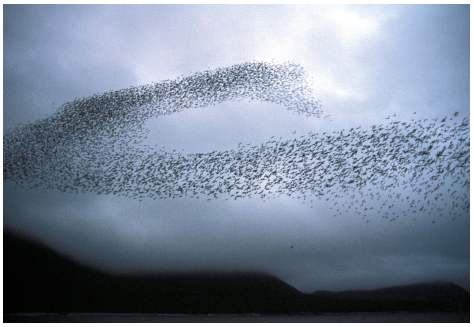
The program itself is running the two functions above, on the same set of images.

For all the operations on the images, such as reading them, displaying them, and calculating the image elements, we are using the well-known OpenCv library, where the heart of the image processing, for our program, is the **Harris corners detection** function.

Although the image processing and computer vision aspects of this program are not directly related to the scope of this work, it would be very suitable to mention here that the benefit of a good clustering algorithm can result also in the ability to cluster image elements also for noisy images (where not all the corners are very accurate), and still yield reasonable clustering.

Results

We present the results of running two clustering algorithms on the same set of images. First, we show some of the images we used for testing:



(a) Birds 1

(b) Birds 2

(c) Beans 1

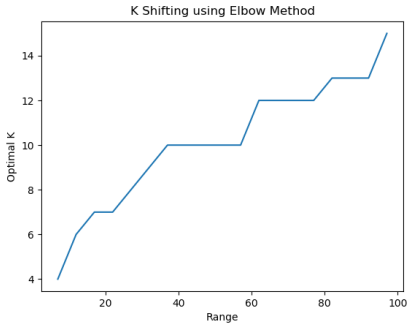
(d) Beans 2

We applied our implementation of k-means clustering and the implementation provided by the scikit-learn library to 25 images of bowls of beans. Our implementation produced 32 clusters, with each cluster isolating a single bowl from the others. The scikit-learn implementation, with k selected using the elbow method when running from k=3 to k=100, produced 16 clusters, with some clusters containing multiple different bowls.

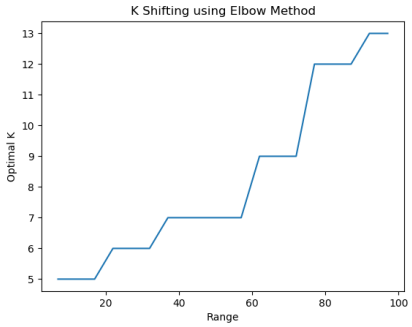
(a) 25 bowls Our Implementation 32 clusters

(b) 25 bowls k-means by Sklearn 16 clusters

We also examined the use of the elbow method for selecting k in the scikit-learn implementation. The Figures show that the value of k changes as we vary the range of k, indicating that the elbow method may not provide a clear choice of k in some cases.

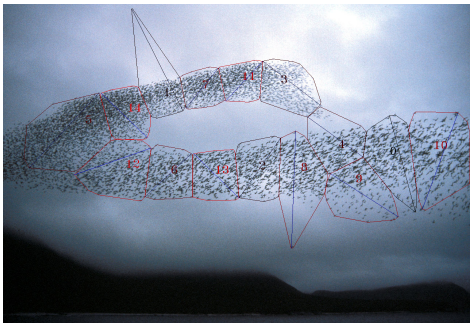


(a) K Shifting - birds Image

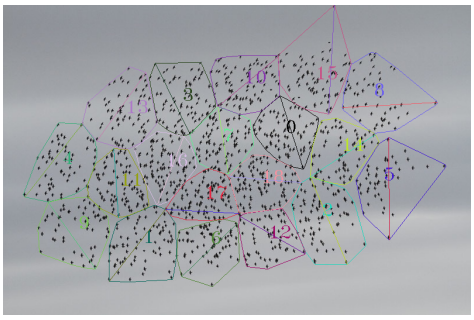


(b) K Shifting - beans Image

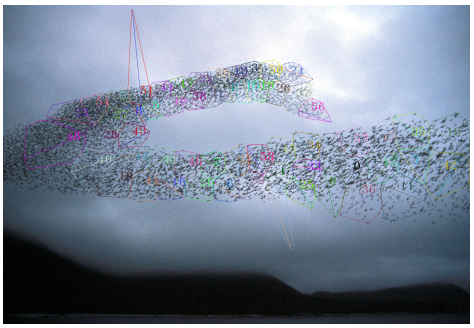
Here some of the birds clustering images:



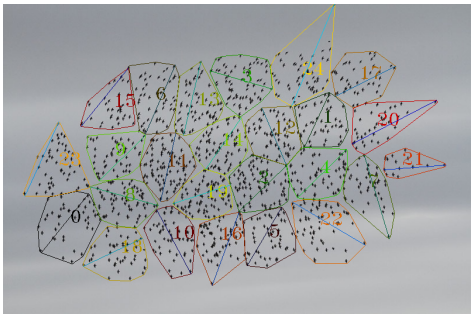
(a) Birds 1 by sklearn k-means 15 clusters



(b) Birds 2 by sklearn k-means 19 clusters



(c) Birds 1 by Our implementation 58 clusters



(d) Birds 2 by Our implementation 25 clusters

Bibliography

[**1**] https://khoury.northeastern.edu/home/hand/teaching/cs6140-fall-2021/Day-18-K-Means-Clustering.pdf

[**2**] https://towardsdatascience.com/elbow-method-is-not-sufficient-to-find-best-k-in-k-means-clustering

[**3**] wikipedia, Stirling numbers of the second kind

[**4**] https://towardsdatascience.com/machine-learning-algorithms-part-9-k-means-example-in-python-f2ad05ed5203

[**5**] ESWA2013.pdf

[**6**] investopedia, limit theorem.asp

[**7**] https://www.investopedia.com/terms/z/zscore.asp