Explaining the initialization and iterations of k-means clustering algorithm:

Let us understand the mechanics of k-means on a 1-dimensional example:\footnote{1}\footnote{Unlike higher-dimensional cases, data points in 1D can be sorted, which makes the clustering problem easier. k-means is not applied to 1D data in real-life. In fact, clustering in 1D is usually called by a different name, \textit{segmentation}. An example of 1D clustering algorithm is \textit{Jenks natural breaks optimization}.}

This is the random initialization of 2 clusters (k=2):

This is how the points are assigned to the clusters:
Remove the old cluster centers:

Calculate the new cluster centers!

Show how the points are assigned to the new clusters!

Delete the old centers:

- Calculate the cluster centers in the next iteration!

- Iterate until the centers are stable!
Now a similar problem in 2D:

Show how the points are assigned to the clusters!

□

□ Calculate the coordinates of the new cluster centers!

□ Iterate until the centers are stable!
Each cluster lies within its (linear) boundaries, which can later be used for classification:

How to do our own clustering with Scikit-learn:

```python
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# generate synthetic two-dimensional data
X, y = make_blobs(random_state=1)

# build the clustering model
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
print("Cluster memberships:\n\{\}").format(kmeans.labels_)
```

How to use the trained clustering model to classify new points (the model is not changed by the new points!)

```python
print(kmeans.predict(X))
```

▶ Use Numpy vectorized operations to prove that the two arrays above are identical.

▶ Use the model trained above to classify these three data points: (-1, 42) (1, -42), (42, 42). Make only one call to `predict()`!

The attribute `cluster_centers_` stores the coordinates of the centers:

```python
print kmeans.cluster_centers_
```

We can print the data points and the centers on the same plot using `mnglearn`:
mglearn.discrete_scatter(X[:, 0], X[:, 1],
    kmeans.labels_, markers='o')

mglearn.discrete_scatter(
    kmeans.cluster_centers_[:, 0],
    kmeans.cluster_centers_[:, 1], [0, 1, 2],
    markers='^', markeredgewidth=2)
Solutions:
Failure cases of k-means (pp.175-8)

Shortcomings for k-means algorithm:

A. The number of clusters is a hyper-parameter. There is no universal algorithm to find the “best” number. Many algorithms exist, like the elbow method, Akaike information criterion (AIC), Bayesian information criterion (BIC), or the Deviance information criterion (DIC).

B. Clusters are defined by diameter only, there is no way to account for density. Here is the text example:

C. Clusters are circular, there is no way to account for direction. Here is the text example:

In general, there is no way to account for cluster shape:
D. Finds clusters even in **non-clustered** data\(^2\):

E. Is very sensitive to **scale**\(^3\):

\(^2\)Example from [https://stats.stackexchange.com/questions/133656/how-to-understand-the-drawbacks-of-k-means](https://stats.stackexchange.com/questions/133656/how-to-understand-the-drawbacks-of-k-means)

\(^3\)Example from [https://stats.stackexchange.com/questions/89809/is-it-important-to-scale-data-before-clustering](https://stats.stackexchange.com/questions/89809/is-it-important-to-scale-data-before-clustering)
F. Even on “perfect data”, k-means can get stuck in a **local minimum**⁴: In this example, although the number of centers is “correct”, too many were used in some areas (e.g. the LR corner), and too few in others (e.g. the green cluster towards the UR corner).


⁴Example from [https://stats.stackexchange.com/questions/133656/how-to-understand-the-drawbacks-of-k-means](https://stats.stackexchange.com/questions/133656/how-to-understand-the-drawbacks-of-k-means)
Relationship between k-means and vector quantization (pp.178-83)

PCA and other dimensionality reduction algorithms express the original data points (vectors) as a sum of directions (parts, components) whose number is lower than the original dimensionality. K-means can also be viewed as a dimensionality reduction algorithm - each point is represented by just one component: the center of a cluster.

Vector quantization is a technique from signal processing, originally used for data compression. It works by dividing a large set of points (vectors) into groups having approximately the same number of points closest to them. Each group is represented by its centroid point, as in k-means and some other clustering algorithms5.

The text code on p.179 shows a comparison between the first few PCA components and the first few clusters obtained with k-means on the dataset *Labeled Faces in the Wild*. PCA was done with `n_components`=100, and k-means with `n_clusters` = 100. (Ignore the NMF decomposition, since we didn’t cover it):

And here are a few reconstructions. For k-means, the reconstruction is simply the closest center:

5https://en.wikipedia.org/wiki/Vector_quantization
An advantage of k-means over PCA: We are not restricted to the number of dimensions of the original dataset. Example: The *Two Moons* dataset has only 2 dimensions, so PCA is no help, but k-means can use, say, 10 clusters, which means 10 features:

```python
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
kmeans = KMeans(n_clusters=10, random_state=0)
kmeans.fit(X)
y_pred = kmeans.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=60, cmap='Paired')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=60,
marker='^', c='range(kmeans.n_clusters), linewidth=2, cmap='Paired')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
print("Cluster memberships: \n{}".format(y_pred))
```

Cluster memberships:

```
[9 2 5 4 2 7 9 6 9 6 1 0 2 6 1 9 3 0 3 1 7 6 8 6 8 5 2 7 5 8 9 8 6 5 3 7 0
 9 4 5 0 1 3 5 2 8 9 1 5 6 1 0 7 4 6 3 3 6 3 8 0 4 2 9 6 4 8 2 8 4 0 4 0 5
 6 4 5 9 3 0 7 8 0 7 5 8 9 8 0 7 3 9 7 1 7 2 2 2 0 4 5 6 7 8 9 4 5 4 1 2 3 1
 8 8 4 9 2 3 7 0 9 9 1 5 8 5 1 9 5 6 7 9 1 4 0 2 9 5 2 5 4 7 9 5 5 3 8 1 9 5 6
 3 5 0 2 9 3 0 8 6 0 3 3 5 6 3 2 0 2 3 2 0 2 6 3 4 4 1 5 6 7 1 1 3 2 4 7 2 7
 3 8 6 4 1 4 3 9 5 1 7 5 8 2]
```
Would the “moon” shapes be preserved with fewer than 10 clusters? Experiment to find out!

Aside from the cluster centers themselves, the **transform()** method of `kmeans` allows easy access to the distance between each point and its center:

```python
distance_features = kmeans.transform(X)
print("Distance feature shape: {}".format(distance_features.shape))
print("Distance features:
{}").format(distance_features))
```

```
Distance feature shape: (200, 10)
Distance features:
[[ 0.922  1.466  1.14  ...,  1.166  1.039  0.233]
 [ 1.142  2.517  0.12  ...,  0.707  2.204  0.983]
 [ 0.788  0.774  1.749  ...,  1.971  0.716  0.944]
 ..., 
 [ 0.446  1.106  1.49  ...,  1.791  1.032  0.812]
 [ 1.39  0.798  1.981  ...,  1.978  0.239  1.058]
 [ 1.149  2.454  0.045  ...,  0.572  2.113  0.882]]
```

Complexity: **O(\(kn^i\))**, where \(i\) is the number of iterations, and \(d\) is the number of features (dimension of data point, e.g. number of pixels in the case of gray-level images).

Do you think that KMeans is sensitive to scaling? Explain, either way!

Next, we cover a clustering algorithm that offers support in choosing the “right” number of clusters.
**Agglomerative Clustering (AClust) (pp.184-9)**

Pseudocode:

- Start by declaring each point its own cluster.
- Repeat:
  - Merge the two “closest” clusters
  - Until a certain objective is reached.

```python
mlearn.plots.plot_agglomerative_algorithm()
```

### The Big-Oh complexity of AClust (Not in text)

The nr. of data points is \( n \) and the number of features is \( d \).

Remark: The complexity depends on the exact way in which the “distance” between clusters is measured. Since clusters are merged or linked, the distance is called the “linkage”. The discussion in our text assumes the so-called "single-linkage", a.k.a. “nearest-neighbor” distance; it is the minimum distance between all pairs of points in those clusters.

A relatively straightforward algorithm for single-linkage uses an \( nxn \) “proximity matrix” to keep track of the distances between clusters at each step. It can be shown to have time complexity \( O(n^2d + n^3) = O(n^3) \). Faster, but more intricate algorithms exist for single-linkage\(^6\), that are only \( O(n^2) \).

Interestingly, the Scikit-learn implementation of AClust does not use single-linkage; see below for details.

---

Using AClust in Scikit-learn

```python
from sklearn.cluster import AgglomerativeClustering
X, y = make_blobs(random_state=1)
agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)
mlearn.discrete_scatter(X[:, 0], X[:, 1], assignment)
plt.legend(["Cluster 0", "Cluster 1", "Cluster 2"], loc="best")
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Unlike KMeans, AClust does not have a `predict()` method, because the clusters do not naturally divide the space among themselves. If a new point is given, the “correct” method is to reapply the algorithm to the larger dataset, because the new point may cause the clustering to take a different path, ending up with very different clusters!

We can either:

- apply `predict()` and then use the information in the attribute `labels_`, as with KMeans, or
- apply `fit_predict()` in one step, which returns the cluster labels (as shown in the example above)

Nothing prevents us, of course, from using the labels to create our own prediction algorithm. As an example, we can find the centroid of each cluster, and then use the distance of a new point to those centroids, k-means-style.

The two most important hyper-parameters are:

- `n_clusters` → The number of clusters where the bottom-up process stops. Its default value is 2.
- `linkage` → How the distance between pairs of clusters is calculated, in order to decide which two will be merged next. Here is the official documentation:

  ```
  linkage : ("ward", "complete", "average"), optional, default: "ward"
  
  Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion.
  * ward minimizes the variance of the clusters being merged.
  * average uses the average of the distances of each observation of the two sets.
  * complete or maximum linkage uses the maximum distances between all observations of the two sets.
  ```

It is also possible to override the “blind” linkage if we have extra information about the structure of the data. In the parameter `connectivity`, we can specify a matrix of neighboring data points, known apriori.

The dendrogram

Here is the text visualization of a simple dataset with 12 points:

```
import the dendrogram function and the ward clustering function from Scipy
from scipy.cluster.hierarchy import dendrogram, ward

X, y = make_blobs(random_state=0, n_samples=12)
# Apply the ward clustering to the data array X
# The SciPy ward function returns an array that specifies the distances
# bridged when performing agglomerative clustering
linkage_array = ward(X)
# Now we plot the dendrogram for the linkage_array containing the distances
# between clusters
dendrogram(linkage_array)
```

We can then use the usual matplotlib.pyplot tools to show cuts at various inter-cluster distances:

```
# mark the cuts in the tree that signify two or three clusters
ax = plt.gca()
bounds = ax.get_xbound()
ax.plot(bounds, [7.25, 7.25], '--', c='k')
ax.plot(bounds, [4, 4], '--', c='k')
ax.text(bounds[1], 7.25, 'two clusters', va='center', fontdict={'size': 15})
ax.text(bounds[1], 4, 'three clusters', va='center', fontdict={'size': 15})
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")
```

Note: `.gca` → get current (figure's) axes (or create a new figure)
The vertical segments in the dendrogram show how far apart the clusters are. Intuitively, we want to stop ("cut" the dendrogram) at the largest branches, as shown above.

Conclusions on AClust:

- A little slower than k-means in practice, but still scales well.
- If we manually create a **connectivity** matrix based on the structure of the dataset, it can capture complex shapes of clusters ...
  - But it does not do it "out of the box".
- It still has no concept of the density of the data points.

(But the next clustering algorithm we present below does capture density!)

☐ Do you think that AClust is sensitive to scaling? Explain, either way!
DBSCAN

The acronym stands for *Density-Based Spatial Clustering of Applications with Noise*.

Idea: Clusters are regions of (relatively) high density of points, separated by regions of (relatively) low density.

More detailed idea: Identify *core samples*, then extend the cluster by finding other *core samples*.

```
from sklearn.cluster import DBSCAN
X, y = make_blobs(random_state=0, n_samples=12)
dbscan = DBSCAN()
clusters = dbscan.fit_predict(X)
print("Cluster memberships: \
{}".format(clusters))
```

What happened? The special label -1 means “no cluster”, i.e. all points are classified as noise! Normal labels start at 0, as usual.

We have to tweak two important hyper-parameters:

- **min_samples** (default 5) ⇒ The number of samples (or total weight) in a neighborhood for a point to be considered as a core point (including the point itself).
- **eps** (epsilon, default 0.5): ⇒ The maximum distance between two samples for them to be considered as being in the same neighborhood.

Performance on the *two-moon* dataset:
Big-Oh complexity: \( O(n^2) \), but can be brought down to \( O(n \log n) \)

Conclusions on DBSCAN:

Pros:
- Does not need the nr. of clusters as hyper-parameter, it finds it automatically!
- Models density!
- Can find arbitrarily-shaped clusters!
- Robust to outliers - will leave a data point outside of any cluster if it is too far!
- In practice, it is a little slower than k-means and AC, but not by much.

Cons:
- Does not do well if the dataset has regions with large differences in density.
- Choosing a good threshold requires good understanding of the scaling of the data.
- Not entirely deterministic: border points that are reachable from more than one cluster can be part of either cluster, depending on the order the data are processed\(^8\).

\(^8\)https://en.wikipedia.org/wiki/DBSCAN#Complexity

\(\Box\) Do you think that DBSCAN is sensitive to scaling? Explain, either way!
Read FYI: *Comparing and Evaluating Clustering Algorithms*
Solutions:

☐ Do you think that KMeans is sensitive to scaling? Explain, either way!
A:

☐ Do you think that AClust is sensitive to scaling? Explain, either way!
A:

☐ Do you think that DBSCAN is sensitive to scaling? Explain, either way!
A: Yes, because epsilon changes if one or more dimensions change scales.