What Is Clustering?

Ref: https://realpython.com/k-means-clustering-python/

Clustering is a set of techniques used to **partition data into groups**, or **clusters**.

Clusters are loosely defined as **groups of data objects** that are more similar to other objects in their cluster than they are to data objects in other clusters.

In practice, clustering helps identify two qualities of data:

1. Meaningfulness
2. Usefulness

Meaningful clusters expand domain knowledge. For example, in the medical field, researchers applied clustering to gene expression experiments. The clustering results identified groups of patients who respond differently to medical treatments.

Useful clusters, on the other hand, serve as an intermediate step in a data pipeline. For example, businesses use clustering for customer segmentation. The clustering results segment customers into groups with similar purchase histories, which businesses can then use to create targeted advertising campaigns.

Three popular categories of clustering algorithms:

1. Partitional clustering
2. Hierarchical clustering
3. Density-based clustering

Partitional Clustering

Partitional clustering divides data objects into **nonoverlapping groups**.

In other words, no object can be a member of more than one cluster, and every cluster must have at least one object.

These techniques require the user to specify the number of clusters, indicated by the variable ***k***. Many partitional clustering algorithms work through an iterative process to assign subsets of data points into ***k* clusters**. Two examples of partitional clustering algorithms are **k-means** and **k-medoids**.

These algorithms are both nondeterministic, meaning they could produce different results from two separate runs even if the runs were based on the same input.

Partitional clustering methods have several strengths:

```
- They work well when clusters have a spherical shape.
- They're scalable with respect to algorithm complexity.
```

They also have several weaknesses:

```
- They’re not well suited for clusters with complex shapes
and different sizes.
- They break down when used with clusters of different
densities.
```

Hierarchical Clustering

Hierarchical clustering determines cluster assignments by building a hierarchy. This is implemented by either a **bottom-up** or a **top-down approach**:

```
**Agglomerative clustering** is the **bottom-up approach**.
It merges the two points that are the most similar until all
points have been merged into a single cluster.

**Divisive clustering** is the **top-down approach**. It
starts with all points as one cluster and splits the least
similar clusters at each step until only single data points
remain.
```

These methods produce a **tree-based hierarchy** of points called a **dendrogram**. Similar to partitional clustering, in hierarchical clustering the number of clusters (k) is often predetermined by the user.

Clusters are assigned by **cutting the dendrogram** at a specified depth that results in k groups of smaller dendrograms.

Unlike many partitional clustering techniques, hierarchical clustering is a deterministic process, meaning cluster assignments won’t change when you run an algorithm twice on the same input data.

The strengths of hierarchical clustering methods include the following:

```
- They often reveal the finer details about the
relationships between data objects.
- They provide an interpretable dendrogram.
```

The weaknesses of hierarchical clustering methods include the following:

```
- They’re computationally expensive with respect to
algorithm complexity.
- They’re sensitive to noise and outliers.
```

Density-Based Clustering

Density-based clustering determines cluster assignments based on the **density** of data points in a region. Clusters are assigned where there are **high densities of data points**

separated by low-density regions.

Unlike the other clustering categories, this approach doesn't require the user to specify the number of clusters. Instead, there is a **distance-based parameter** that acts as a **tunable threshold**. This threshold determines how close points must be to be considered a cluster member.

Examples of **density-based clustering** algorithms include **Density-Based Spatial Clustering of Applications with Noise**, or **DBSCAN**, and **Ordering Points To Identify the Clustering Structure**, or **OPTICS**.

The strengths of density-based clustering methods include the following:

```
- They excel at identifying clusters of nonspherical shapes.
- They're resistant to outliers.
```

The weaknesses of density-based clustering methods include the following:

```
- They aren't well suited for clustering in high-dimensional
spaces.
- They have trouble identifying clusters of varying
densities.
```

K-Means Algorithm

Algorithm 1 k-means algorithm

1: Specify the number k of clusters to assign.
2: Randomly initialize k centroids.
3: **repeat**
4: **expectation:** Assign each point to its closest centroid.
5: **maximization:** Compute the new centroid (mean) of each cluster.
6: **until** The centroid positions do not change.

In [1]:
```
import matplotlib.pyplot as plt
from kneed import KneeLocator
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler
```

You can generate the data from the above GIF using **make_blobs()**, a convenience function in scikit-learn used to generate synthetic clusters. **make_blobs()** uses these parameters:

```
**n_samples** is the total number of samples to generate.
**centers** is the number of centers to generate.
**cluster_std** is the standard deviation.
```

make_blobs() returns a tuple of two values:

1. A two-dimensional NumPy array with the **x- and y-values** for each of the samples
2. A one-dimensional NumPy array containing the **cluster labels** for each sample

In [2]:
```
## Generate the synthetic data and labels:

features, true_labels = make_blobs(
   n_samples=200,
   centers=3,
   cluster_std=2.75,
   random_state=42
   )
```

Here's a look at the first five elements for each of the variables returned by **make_blobs()**:

In [4]:
```
features[:5]
```

Out[4]:
```
array([[  9.77075874,   3.27621022],
       [ -9.71349666,  11.27451802],
       [ -6.91330582,  -9.34755911],
       [-10.86185913, -10.75063497],
       [ -8.50038027,  -4.54370383]])
```

In [5]:
```
true_labels[:5]
```

Out[5]:
```
array([1, 0, 2, 2, 2])
```

In [3]:
```
scaler = StandardScaler()
scaled_features = scaler.fit_transform(features)
```

In [7]:
```
scaled_features[:5]
```

Out[7]:
```
array([[ 2.13082109,  0.25604351],
       [-1.52698523,  1.41036744],
       [-1.00130152, -1.56583175],
       [-1.74256891, -1.76832509],
       [-1.29924521, -0.87253446]])
```

In [4]:
```
kmeans = KMeans(
init="random",
n_clusters=3,
n_init=10,
max_iter=300,
random_state=42
)
```

In [6]:
```
kmeans.fit(scaled_features)
```

Out[6]:
```
KMeans(init='random', n_clusters=3, random_state=42)
```

In [7]:
```
# The lowest SSE value
kmeans.inertia_
```

Out[7]:
```
74.57960106819854
```

In [8]:
```
# Final locations of the centroid
kmeans.cluster_centers_
```

Out[8]:
```
array([[-0.25813925,  1.05589975],
       [-0.91941183, -1.18551732],
       [ 1.19539276,  0.13158148]])
```

In [9]:
```
# The number of iterations required to converge
kmeans.n_iter_
```

Out[9]:
```
2
```

In [10]:
```
# the first five predicted labels:
kmeans.labels_[:5]
```

Out[10]:
```
array([2, 0, 1, 1, 1])
```

Choosing the Appropriate Number of Clusters

In this section, you'll look at two methods that are commonly used to evaluate the appropriate number of clusters:

1. The elbow method
2. The silhouette coefficient

These are often used as complementary evaluation techniques rather than one being preferred over the other.

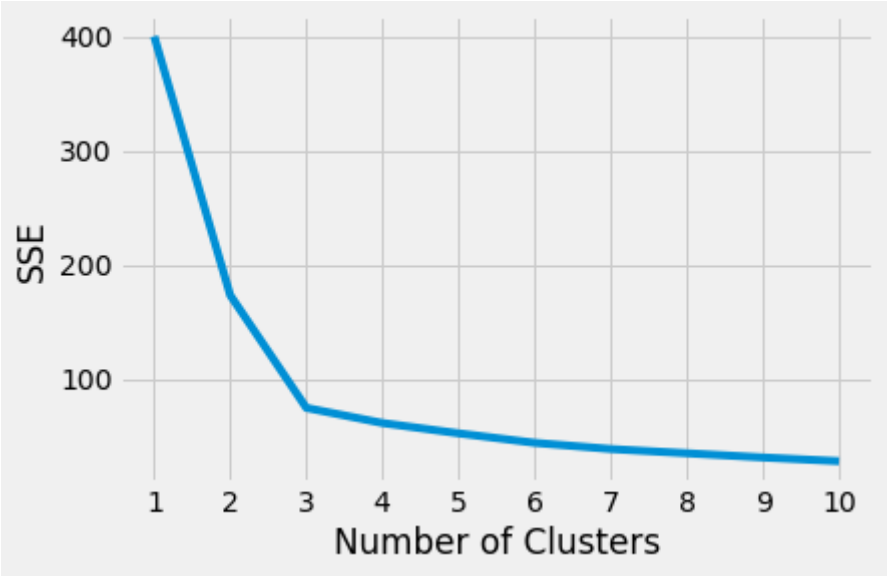
To perform the **elbow method**, run several k-means, increment k with each iteration, and record the SSE:

In [11]:
```
kmeans_kwargs = {
"init": "random",
"n_init": 10,
"max_iter": 300,
"random_state": 42,
}

# A list holds the SSE values for each k
sse = []
for k in range(1, 11):
     kmeans = KMeans(n_clusters=k, **kmeans_kwargs)
     kmeans.fit(scaled_features)
     sse.append(kmeans.inertia_)
```

```
C:\Users\jitfr\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:1036:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, whe
n there are less chunks than available threads. You can avoid it by setting
the environment variable OMP_NUM_THREADS=1.
  warnings.warn(
```

In [12]:
```
plt.style.use("fivethirtyeight")
plt.plot(range(1, 11), sse)
plt.xticks(range(1, 11))
plt.xlabel("Number of Clusters")
plt.ylabel("SSE")
plt.show()
```



In [13]:
```
kl = KneeLocator(
range(1, 11), sse, curve="convex", direction="decreasing"
)

kl.elbow
```

Out[13]:
```
3
```

The **silhouette coefficient** is a measure of **cluster cohesion** and **separation**. It quantifies how well a data point fits into its assigned cluster based on two factors:

```
1. How close the data point is to other points in the
cluster
2. How far away the data point is from points in other
clusters
```

Silhouette coefficient values range between -1 and 1. Larger numbers indicate that samples are closer to their clusters than they are to other clusters.

In [14]:
```
# A list holds the silhouette coefficients for each k
silhouette_coefficients = []

# Notice you start at 2 clusters for silhouette coefficient
for k in range(2, 11):
     kmeans = KMeans(n_clusters=k, **kmeans_kwargs)
     kmeans.fit(scaled_features)
     score = silhouette_score(scaled_features, kmeans.labels_)
     silhouette_coefficients.append(score)
```

Plotting the average silhouette scores for each k shows that the best choice for k is 3 since it has the maximum score:

In [15]:
```
plt.style.use("fivethirtyeight")
plt.plot(range(2, 11), silhouette_coefficients)
plt.xticks(range(2, 11))
plt.xlabel("Number of Clusters")
plt.ylabel("Silhouette Coefficient")
plt.show()
```


Silhouette Coefficient
0.60
0.55
0.50
0.45
0.40
0.35
2
3
4
5
6
7
8
9
10
Number of Clusters

In [ ]: