K-means inertia & silhouette score

Import statements

Before we begin with the exercises and analyzing the data, we need to import all libraries and extensions required for this programming exercise. Throughout the course, we will be using numpy and pandas for operations, and seaborn for data visualization.

Of particular note here, are the Kmeans, silhouette_score, and StandardScaler statements. These are the elements directly related to the K-means modeling process.

In [3]:
```
# IMPORT STATEMENTS

# Standard operational package imports
import numpy as np
import pandas as pd

##################################################
###  Important imports for modeling and evaluation
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler
##################################################


# (To create synthetic data)
from sklearn.datasets import make_blobs

# Visualization package import
import seaborn as sns
```

Create the data

In practice, you'd have a dataset of real data, and you'd read in this data and perform EDA, data cleaning, and other manipulations to prepare it for modeling. For simplicity and to help us focus on the modeling itself, we're going to use synthetic data for this demonstration.

We'll start by creating a random number generator. This is to help with the process of creating reproducible synthetic data. We'll use it to create clustered data without us knowing how many clusters there are.

In [6]:
```
# Create random number generator
rng = np.random.default_rng(seed=44)
```

By calling the random number generator and assigning the result to a variable, we can avoid seeing the true number of clusters our data has. This keeps the "answer" a secret, and will let us use inertia and silhouette coefficients to determine it.

In [7]:
```
# Create synthetic data w/ unknown number of clusters
centers = rng.integers(low=3, high=7)
X, y = make_blobs(n_samples=1000, n_features=6, centers=centers, random_st
```

The above steps that generated our synthetic data return two things: X and y. X is an array of the values for the synthetic data itself and y is an array that contains the cluster assignment for each sample in X (represented as an integer).

Right now we're concerned with X, because it is our mystery data. It's currently an array, but it's usually helpful to view your data as a pandas dataframe. This is often how your data will be organized when modeling real-world data, so we'll convert our data to a pandas df.

In [8]:
```
# Create Pandas dataframe from the data
X = pd.DataFrame(X)
X.head()
```

Out[8]:

	0	1	2	3	4	5
0	-1.534288	5.467808	-6.945988	1.403934	1.553836	-7.618236
1	-6.681020	6.717808	2.764396	4.460744	-8.286569	10.959708
2	-8.678310	7.825306	3.139699	5.609951	-9.948079	8.072149
3	-6.667385	7.147637	2.145505	4.712937	-9.544708	11.093248
4	-2.753835	-4.209968	0.620345	-7.439505	-4.405723	-2.046149

We see that the data has 6 features (columns). This is too many for us to visualize in 2-D or 3-D space. We can't see how many clusters there are, so we'll need to use our detective skills to determine this.

Scale the data

Since K-means uses distance between observations and centroids as its measure of similarity, it's important to scale your data before modeling, if it's not already scaled. It's important to scale because K-means doesn't know what your unit label is for each variable. Suppose you had data for penguins, and height were measured in meters and weight were measured in grams. Without scaling, significant differences in height would be represented as small numbers, while minor differences in weight would be represented as much larger numbers.

To perform scaling, we'll use scikit-learn's StandardScaler. StandardScaler scales each point x_i by subtracting the mean value for that feature and dividing by the standard deviation:

x-scaled = (x_i – mean(x)) / σ

This ensures that, after scaling, each feature variable has a mean of 0 and variance/standard deviation of 1. There are a number of scaling techniques available, including StandardScaler, MinMaxScaler, Normalizer, and others, each scaling the data in a particular way. There's no hard rule for determining which method will work best, but with K-means models, using any scaler will almost always result in better results than not scaling at all.

You can instantiate StandardScaler and transform your data in a single step by using the .fit_transform() method and passing to it your data as an argument.

Tip: If your computing environment has sufficient resources, it's helpful to keep an unscaled

In [9]:
```
# Scale the data
X_scaled = StandardScaler().fit_transform(X)
X_scaled[:2,:]
```

Out[9]:
```
array([[-0.03173691,  0.4864719 , -1.32178135,  0.59808997,  1.5703227 ,
        -0.88951855],
       [-1.05006137,  0.68381835,  0.74465777,  1.2564266 , -0.97057774,
         1.92995522]])
```

Instantiate the model

Now that the data is scaled, we can start modeling. Since we don't know how many clusters exist in the data, we'll begin by examining the inertia values for different values of k.

One thing to note is that, by default, scikit-learn implements an optimized version of the K-means algorithm, called K-means++. This helps to ensure optimal model convergence by initializing centroids far away from each other. Because we're using K-means++, we will not rerun the model multiple times.

We'll begin by instantiating the model. If we want to build a model that clusters the data into three clusters, we'd set the n_clusters parameter to 3. We'll also set the random_state to an arbitrary number. This is only so others can reproduce your results. If you left this value blank, it's possible others could replicate your code exactly and still get different results due to the random initial placement of centroids.

In [10]:
```
# Instantiate model
kmeans3 = KMeans(n_clusters=3, random_state=42)
```

Fit the data

Now that we've instantiated the model, the next step is to fit it to the data. We do this by using the .fit() method and passing to it our scaled data.

In [11]:
```
# Fit model to data
kmeans3.fit(X_scaled)
```

Out[11]:
```
KMeans(n_clusters=3, random_state=42)
```

This returns a model object that has "learned" your data. You can now call its different attributes to see inertia, location of centroids, and class labels, among others. See the K-means documentation for a full list of available attributes.

We can get the cluster assignments by using the .labels_ attribute. Similarly, we can get the inertia by using the .inertia_ attribute.

Let's see what happens when we check the cluster assignments and inertia for this model.

In [12]:
```
print('Clusters: ', kmeans3.labels_)
print('Inertia: ', kmeans3.inertia_)
```

```
Clusters:  [1 2 2 2 0 0 0 2 2 1 0 2 2 1 2 2 2 2 0 0 0 2 2 2 1 1 0 2 2 0 0 2 0
0 1 1 0
 2 2 2 0 2 0 2 2 0 0 1 2 1 0 0 0 1 1 1 0 2 0 1 0 0 0 0 0 0 1 0 0 2 0 0 0
 2 0 2 1 1 2 1 2 0 1 0 2 1 1 2 1 2 0 1 2 0 1 0 2 2 0 0 2 2 2 2 2 2 0 0 2 1
 0 2 0 2 2 0 0 1 0 0 0 2 0 0 2 2 0 1 2 1 1 1 0 2 1 1 0 2 0 0 0 2 1 1 1 0 0
 2 0 0 2 0 0 2 2 1 2 1 2 2 0 2 2 2 2 0 0 2 0 2 2 0 2 2 1 2 2 1 1 0 2 0 2 0
 2 1 2 1 2 0 0 0 2 2 2 1 1 2 0 2 2 0 2 0 0 0 0 0 1 0 1 2 2 0 1 1 2 0 0 1 0
 2 2 2 2 0 0 0 2 1 0 1 2 1 2 1 0 0 2 2 2 1 1 0 0 1 1 1 2 2 2 0 0 0 2 2 2 0
 2 0 1 0 2 2 1 1 0 0 2 0 1 2 0 2 2 2 2 0 2 2 0 2 2 0 0 0 0 0 2 1 0 2 0 1 1
 0 2 2 0 2 2 2 2 2 2 1 2 1 0 2 2 1 0 0 2 0 2 2 1 2 0 2 2 2 0 0 0 1 1 1 0 1
 1 1 2 2 1 2 1 0 2 0 1 2 1 0 2 0 2 2 2 2 0 0 2 0 2 0 0 0 1 2 2 2 2 2 0 0 1
 0 2 2 0 2 0 0 2 0 2 1 2 2 0 2 0 0 1 2 0 0 0 1 0 0 2 0 2 0 2 1 2 0 2 2 0 0
 1 2 0 0 1 0 0 2 2 2 2 0 2 0 2 2 1 1 0 1 0 0 0 2 2 1 0 2 2 2 0 2 0 2 1 2 0
 1 0 0 0 2 0 1 2 0 1 2 2 0 0 0 0 0 2 2 0 2 2 0 2 2 0 0 1 0 1 2 0 0 0 0 1 2
 2 0 2 2 1 1 2 0 2 1 1 2 0 2 2 0 0 0 1 2 0 0 2 1 1 0 1 2 0 0 2 2 0 2 0 2 0
 0 0 2 2 0 0 2 2 1 2 1 0 0 0 1 2 0 0 0 2 1 1 1 1 1 2 0 1 2 0 2 2 0 1 0 2 1
 2 2 2 0 0 2 0 0 0 1 1 2 2 0 0 2 2 1 2 0 2 2 1 2 0 0 0 1 2 1 0 0 1 2 2 0 0
 0 0 0 2 2 1 2 1 0 1 0 1 1 2 0 0 2 1 2 1 0 1 2 0 2 0 0 2 2 0 2 2 0 2 2 0 0
 2 2 2 2 0 1 0 2 2 0 2 0 0 2 2 0 0 0 1 1 1 0 0 1 2 1 1 0 1 2 2 2 2 2 2 0 1
 2 2 0 0 0 2 2 0 2 0 2 0 0 2 0 2 0 2 0 2 0 2 2 0 2 0 2 2 1 0 0 0 2 1 2 0 1
 1 2 0 2 2 0 2 2 2 1 2 0 2 2 0 1 2 1 2 0 0 0 2 0 2 1 2 0 2 0 2 2 1 2 2 0 0
 0 2 0 0 2 2 0 1 1 2 0 0 0 2 0 0 1 0 1 0 1 0 0 2 2 0 0 0 1 1 0 0 0 1 0 2 1
 0 0 0 0 2 1 2 0 2 0 2 1 2 0 0 2 0 1 0 0 2 1 0 2 0 0 1 0 0 2 0 1 2 1 1 2 1
 2 0 0 0 0 0 1 1 2 2 0 2 0 2 2 0 2 1 0 1 2 0 0 0 2 2 2 2 2 2 0 1 2 2 2 2 2
 0 1 1 1 2 0 0 1 2 2 2 0 0 0 1 0 2 0 0 2 0 0 0 0 0 0 1 2 0 2 0 2 2 0 0 2
 1 2 2 2 1 0 2 0 0 1 0 1 0 1 2 1 2 1 1 2 2 2 2 0 0 0 2 0 0 1 2 2 2 0 0 2 0
 1 2 0 2 0 2 2 2 2 2 2 1 2 1 2 2 1 2 2 0 0 2 2 0 0 0 0 2 2 1 2 2 2 2 2 0 0
 0 2 0 1 2 2 2 0 2 1 0 2 1 2 0 0 1 0 0 0 0 2 2 0 0 0 0 0 1 0 2 0 1 1 1 2 0
 0]
Inertia:  1748.1488703079513
```

The .labels_ attribute returns a list of values that is the same length as the training data. Each value corresponds to the number of the cluster to which that point is assigned. Since our K-means model clustered the data into three clusters, the value assigned to each observation will be 0, 1, or 2. (Note that the cluster number itself is arbitrary, and serves only as a label.)

The .inertia_ attribute returns the sum of the squared distances of samples from their closest

Evaluate inertia

This inertia value isn't helpful by itself. We need to compare the inertias of multiple k values. To do this, we'll create a function that fits a K-means model for multiple values of k, calculates the inertia for each k value, and appends it to a list.

In [13]:
```python
# Create a list from 2-10.
num_clusters = [i for i in range(2, 11)]

def kmeans_inertia(num_clusters, x_vals):
    '''
    Fits a KMeans model for different values of k.
    Calculates an inertia score for each k value.

    Args:
        num_clusters: (list of ints)  - The different k values to try
        x_vals:       (array)         - The training data

    Returns:
        inertia:      (list)          - A list of inertia scores, one for
                                        value of k
    '''

    inertia = []
    for num in num_clusters:
        kms = KMeans(n_clusters=num, random_state=42)
        kms.fit(x_vals)
        inertia.append(kms.inertia_)

    return inertia
```

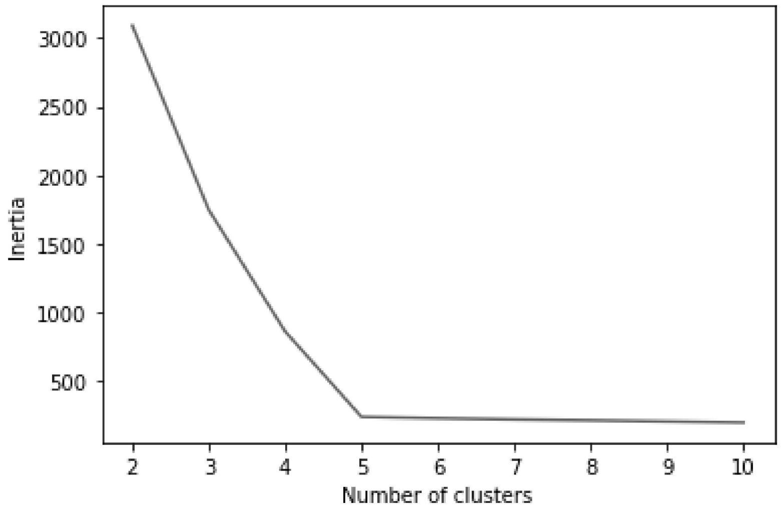
In [14]:
```python
# Calculate inertia for k=2-10
inertia = kmeans_inertia(num_clusters, X_scaled)
inertia
```

Out[14]:
```
[3090.3260348468543,
 1748.1488703079513,
 863.1663243212959,
 239.65434758718436,
 229.97193447062892,
 221.55016192539154,
 214.37627403442247,
 206.67023398404046,
 199.2336322973373]
```

Elbow plot

Now we can plot these values in a simple line graph, with the k values along the x-axis and inertia on the y-axis. This type of plot is called an elbow plot. The "elbow" is usually the part of the curve with the sharpest angle, where the reduction in inertia that occurs when a new cluster is added begins to level off.

In [15]:
```
# Create an elbow plot
plot = sns.lineplot(x=num_clusters, y=inertia)
plot.set_xlabel("Number of clusters");
plot.set_ylabel("Inertia");
```



This plot contains an unambiguous elbow at five clusters. Models with more than five clusters don't seem to reduce inertia much at all. Right now, it seems like a 5-cluster model might be optimal.

Let's now check silhouette scores. Hopefully the results will corroborate our findings from the assessment of inertia.

Evaluate silhouette score

Unlike inertia, silhouette score doesn't have its own attribute that can be called on the model object. To get a silhouette score, we have to use the silhouette_score() function that we imported from sklearn.metrics. You must pass to it two required parameters: your training data and their assigned cluster labels. Let's see what this looks like for the kmeans3 model we created earlier.

In [17]:
```
# Get silhouette score for kmeans3 model
kmeans3_sil_score = silhouette_score(X_scaled, kmeans3.labels_)
kmeans3_sil_score
```

Out[17]:
```
0.5815196371994132
```

It worked! However, this value isn't very useful if we have nothing to compare it to. Just as we did for inertia, we'll write a function that compares the silhouette score of each value of k, from 2 through 10.

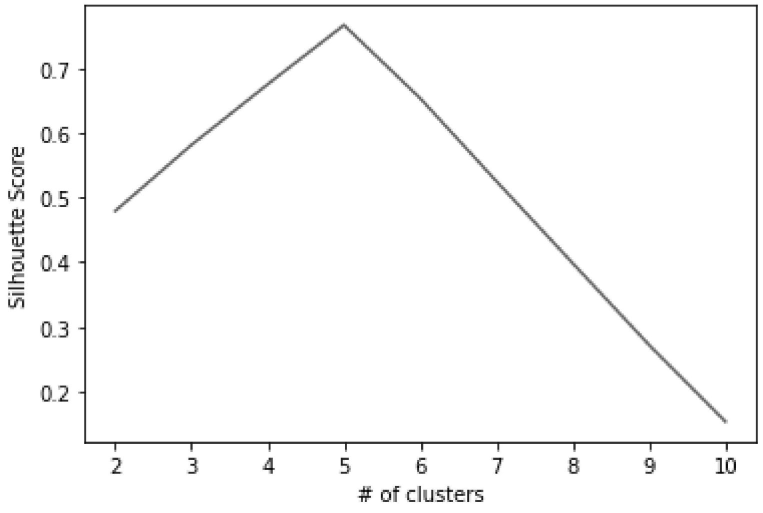
In [18]:
```
def kmeans_sil(num_clusters, x_vals):
    '''
    Fits a KMeans model for different values of k.
    Calculates a silhouette score for each k value

    Args:
        num_clusters: (list of ints)  - The different k values to try
        x_vals:       (array)         - The training data

    Returns:
        sil_score:    (list)          - A list of silhouette scores, one f
                                        value of k
    '''

    sil_score = []
    for num in num_clusters:
        kms = KMeans(n_clusters=num, random_state=42)
        kms.fit(x_vals)
        sil_score.append(silhouette_score(x_vals, kms.labels_))

    return sil_score
```

In [19]:
```
# Calculate silhouette scores for k=2-10
sil_score = kmeans_sil(num_clusters, X_scaled)
sil_score
```

Out[19]:
```
[0.4792051309087745,
 0.5815196371994132,
 0.6754359269330664,
 0.7670656870960783,
 0.6536170057112888,
 0.5251340670197663,
 0.39825007246113,
 0.2718615021565435,
 0.1533823508053291]
```

We can plot the silhouette score for each value of k, just as we did for inertia. However, remember that for silhouette score, greater numbers (closest to 1) are better, so we hope to see at least one clear "peak" that is close to 1.

In [20]:
```python
# Create a line plot of silhouette scores
plot = sns.lineplot(x=num_clusters, y=sil_score)
plot.set_xlabel("# of clusters");
plot.set_ylabel("Silhouette Score");
```



This plot indicates that the silhouette score is closest to 1 when our data is partitioned into five clusters. It confirms what we saw in the inertia analysis, where we noticed an elbow where k=5.

Conclusion

At this point, between our inertia and silhouette score analyses, we can say with a reasonable degree of confidence that it makes the most sense to group our data into five clusters.

Since we used synthetic data for this activity, we can cheat and check to see how many clusters actually existed in our data. We can do this by calling the centers variable, which we created with the random number generator at the beginning of this notebook.

In [22]:
```python
# Verify our findings (only possible when "correct" number of clusters exi
centers
```

Out[22]:
```
5
```

We were right! Just as we predicted, there are indeed five distinct clusters in our data. We were able to deduce this by using inertia and silhouette score!

Further analysis¶

Although we know that five clusters is the best grouping for the data, the work is far from done. At this point, we'll instantiate a new K-means model with n_clusters=5 and fit it to our data. (Note that if we had saved all the models that we fit above for different values of k, we wouldn't need to refit a model now, we could just call that model from earlier. But since it wasn't saved, we must fit another model.)

In [23]:
```
# Fit a 5-cluster model to the data
kmeans5 = KMeans(n_clusters=5, random_state=42)
kmeans5.fit(X_scaled)
```

Out[23]:
```
KMeans(n_clusters=5, random_state=42)
```

In [24]:
```
print(kmeans5.labels_[:5])
print('Unique labels:', np.unique(kmeans5.labels_))
```

```
[3 2 2 2 1]
Unique labels: [0 1 2 3 4]
```

Now that we have our labels, it's important to understand what they mean and decide whether this clustering makes sense for our use case. Here's where it helps to keep our unscaled data from the beginning. We can assign a new column to the original unscaled dataframe with the cluster assignment from the final K-means model.

In [25]:
```
# Create new column that indicates cluster assignment in original datafram
X['cluster'] = kmeans5.labels_
X.head()
```

Out[25]:

	0	1	2	3	4	5	cluster
0	-1.534288	5.467808	-6.945988	1.403934	1.553836	-7.618236	3
1	-6.681020	6.717808	2.764396	4.460744	-8.286569	10.959708	2
2	-8.678310	7.825306	3.139699	5.609951	-9.948079	8.072149	2
3	-6.667385	7.147637	2.145505	4.712937	-9.544708	11.093248	2
4	-2.753835	-4.209968	0.620345	-7.439505	-4.405723	-2.046149	1

In [26]:
```
# Create a new observation (for demonstration)
new_observation = rng.uniform(low=-10, high=10, size=6).reshape(1, -1)
new_observation
```

Out[26]:
```
array([[-4.8377385 , -1.88458544,  9.38367896, -6.7536575 ,  7.14587347,
        -6.73909458]])
```

Just as before, we must scale this new data the same way we did earlier. This means that we need to subtract the mean of the training data and divide by the standard deviation of the training data. If you forgot to scale here you'd get invalid results, because your model would be trained on scaled data while the new data going into it would be unscaled.

Here, we must reinstantiate a scaler and fit it to the original data, because we didn't save the scaler object itself when we performed scaling above. We fit and transformed the data in a single line of code, without saving the fit scaler object.

Above, if instead of: X_scaled = StandardScaler().fit_transform(X)

we had written: scaler = StandardScaler()fit.(X) X_scaled = scaler.transform(X)

then we could have reused scaler in this next step without having to assign it.

In [27]:
```python
# Instantiate the scaler and fit it to the original X data
scaler = StandardScaler().fit(X.iloc[:,:-1])

# Apply the scaler to the new observation
new_observation_scaled = scaler.transform(new_observation)
new_observation_scaled
```

Out[27]:
```
array([[-0.68535259, -0.67430308,  2.1532887 , -1.15878741,  3.01424824,
        -0.75609599]])
```

We can use the .predict() method of our kmeans5 model to predict a cluster assignment by passing to it the new observation. In this case, we only have a single observation, but it's also possible to pass an array of new data as an argument, and it would return an array of cluster predictions.

In [28]:
```python
# Predict cluster assignment of new_observation
new_prediction = kmeans5.predict(new_observation_scaled)
new_prediction
```

Out[28]:
```
array([1])
```

The model has assigned this new observation to cluster 1.

Calculating the distance to each centroid

The KMeans model also lets us access the distances of observations from each centroid. For new data, we can do this using the .transform_ method of the fit model object.

In [29]:
```python
# Calculate distances between new data and each centroid
distances = kmeans5.transform(new_observation_scaled)
distances
```

Out[29]:
```
array([[4.55233304, 3.46792667, 5.95732403, 4.14567617, 5.11675395]])
```

Notice that the .transform_ method returns an array. In this case, we gave the model a single new data point, and it returned an array of with five numbers (because our model has five clusters). Each value in the array represents the distance between new_observation_scaled and the centroid of the cluster at that index.

So, the distance between new_observation_scaled and the centroids of:

Cluster 0 = 4.55 Cluster 1 = 3.47 Cluster 2 = 5.96 Cluster 3 = 4.15 Cluster 4 = 5.12

The shortest distance is 3.47—between new_observation_scaled and cluster 1's centroid. This is why the point was assigned to cluster 1 when we used the .predict() method above.