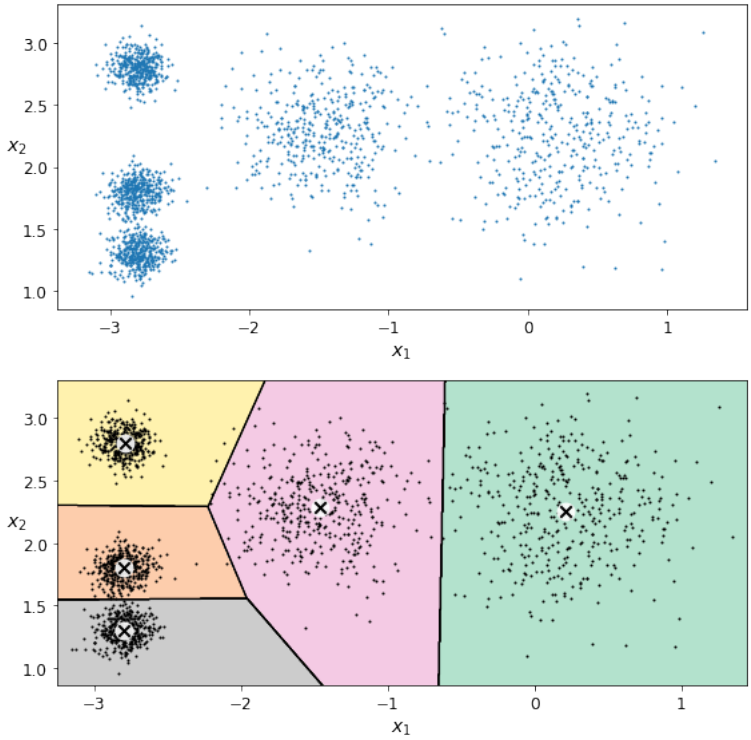
Variance and KMeans
ML_2022: Machine Learning

https://people.sc.fsu.edu/~jburkardt/classes/ml_2022/cluster_lab/cluster_lab.pdf



Using KMeans, we can discover how data has formed cluster patterns.

Variance reduction!

We are interested in how much our data clusters around one or more centers.

- **variance** *measures the tightness of the clustering;*
- *If we suspect a single cluster, we may want to standardize the data first;*
- *For multidimensional data, the covariance matrix reports variance, independence, and correlation of the data;*
- *If there are multiple centers, a better model is available through* `kmeans`*;*
- *To use kmeans, we need to choose the number of clusters;*
- *The behavior of the* **inertia** *suggests the right number of clusters;*

1 Copying the data

Each of the exercises will be carried out on a particular datafile. These datafiles are available on the *datasets* page at the class website:
https://people.sc.fsu.edu/~jburkardt/classes/ml_2022/datasets/datasets.html

You might go ahead now and download them all:

- *hw_data.txt*
- *faithful_data.txt*
- *ruspini_data.txt*
- *blobs_data.txt*
- *blobs_clusters.txt*
- *blobs_centers.txt*

2 Exercise 1:

We want to plot a sample of the height and weight data in `hw_data.txt`, to show how a two-dimensiona set of data can form a single cluster: Write a program `exercise1.py` and:

- use `np.loadtxt()` to read `data` from *hw_data.txt*;
- use `np.shape()` to get and print the number of rows and columns;
- print the first five rows of `data`;
- make a copy of `data` that omits column 0, the data index;
- make a copy of `data` that keeps only the first 350 records;
- compute and print min, max, range, mean, variance, std of `data`;
- create `data2`, a standardized copy of `data`;
- use `plt.scatter ( x values, y values )` to plot height (column 0) versus weight (column 1);
- draw 3 rings around your data:

```
tc = np.linspace ( 0, 2.0 * np.pi, 51 )
xc = np.cos ( tc )
yc = np.sin ( tc )
plt.plot ( xc, yc, 'r-', linewidth = 2 )
plt.plot ( 2.0*xc, 2.0*yc, 'r-', linewidth = 2 )
plt.plot ( 3.0*xc, 3.0*yc, 'r-', linewidth = 2 )
```

- Compute and print the covariance matrix of `data2` using the command

```
cov = np.cov ( data2, rowvar = False )
```

Because the data is standardized, the variance and standard deviation are both 1. The rings therefore indicate data that is no more than 1, 2, and 3 standard deviations from the mean, and should contain almost all of your data.

There is a directional bias in your data. Large heights tend to do with large weights, and small heights with small weights. How does the covariance matrix tell you that this is happening?

3 Prepare for Exercise 2:

In Exercise 2, we will read two-dimensional data that clearly forms two separate clusters. We will use the letter `d` to represent the spatial dimension, so here `d=2`. Each cluster has a center, and points belong to the cluster with the nearest center. The original data has a variance that we already know how to compute. Once we divide the data into two clusters, we can recompute a new version of the variance, known as the **inertia**. It is the sum of the variance of the points in cluster 0 from center 0, plus the variance of points in cluster 1 from center 1:

$$inertia = var_0 + var_1$$

as described in the lecture. If the clustering is done correctly, the two-cluster inertia must be less than the original variance (which we could also call the "one-cluster inertia".

There is a standard algorithm for splitting up our data into `k` clusters, known as the **K Means algorithm**. We will use the implementation available in `scikit-learn`. For more detail, search for `scikit learn kmeans`.

To use the algorithm, we need the statement

```
from sklearn import KMeans  # Capital K, Capital M!
```

If we wish to create `k` clusters, we must define the word `kmeans` as follows:

```
kmeans = KMeans ( n_clusters = k )
```

To cluster the `n` values in `data`, we then write

```
y = kmeans.fit_predict ( data )
```

Here, each of the `n` values `y[i]` satisfies $0 \leq y[i] < k$, and indicates that data item i belongs to cluster `y[i]`.

If we wish to plot all the points belonging to cluster 0, we might use a command like this:

```
plt.scatter ( data[y==0, 0], data[y==0, 1], c = 'red' )
```

with similar commands to plot points from other clusters, perhaps in blue, green, and so on.

Each cluster has a center. The coordinates of these centers are in the $k \times d$ array `kmeans.cluster_centers_` (Notice the trailing underscore!)

It's important to know how the inertia changes as we increase the number of clusters. Once you define `kmeans` and apply it to the data, the corresponding inertia is available as the quantity `kmeans.inertia_`. (Again, notice the trailing underscore!) Therefore, if we want to compare the intertia for the one-cluster and two-cluster cases, then for k = 1 and then k = 2, we have to:

- set `k`;
- define `kmeans`;
- apply `kmeans()` to our data;
- print `kmeans.inertia_`;

This was a lot of preparation, but now we are ready to deal with our tricky two-cluster data from the Old Faithful geyser.

4 Exercise 2:

Write a program `exercise2.py` and:

- use `np.loadtxt()` to read `data` from *faithful_data.txt*;
- use `np.shape()` to get and print the number of rows and columns;
- print the first five rows of `data`;
- compute and print min, max, range, mean, variance, std of `data`;
- create `data2`, a standardized copy of `data`;
- use `plt.scatter ( x values, y values )` to plot eruption time (column 0) versus wait (column 1);
- apply the KMeans algorithm to `data2`, requesting `k=1` clusters, and print the inertia;
- apply the KMeans algorithm to `data2`, requesting `k=2` clusters, and print the inertia;
- in a scatter plot show cluster 0, using `c = 'red'`;
- in the same scatter plot, show cluster 1 using `c = 'cyan'`;
- in the same scatter plot, add the two cluster centers, using `c = 'black'` and `marker = '*'`

You should notice that the inertia decreased a lot when we went from `k=1` to `k=2`. In your scatterplot, you should expect the red and cyan dots to be correctly clustered around a cluster center.

5 Exercise 3:

The Ruspini data naturally breaks into a number of clusters. We will use `KMeans` to cluster the data into $1 \leq k \leq 10$ clusters, compute the inertia each time, and plot the sequence of inertia values, looking for a sort of "elbow" in the plot, which suggests a natural value of `k` to choose.

Write a program `exercise3.py` and:

- use `np.loadtxt()` to read `data` from *ruspini_data.txt*;
- use `np.shape()` to get and print the number of rows and columns;
- print the first five rows of `data`;
- create `data2`, a standardized copy of `data`;
- use `plt.scatter ( x values, y values )` to plot x (column 0) versus y (column 1);
- for $1 \leq k < 11$, set up `kmeans`, cluster the data, set `inertia[k-1]=kmeans.inertia_`;
- print the values `k,inertia[k-1]`;
- plot the values `k,inertia[k-1]`;
- based on the inertia plot, choose a value of `k`;
- using `k`, define `kmeans`, use `kmeans()` to cluster `data2`;
- in a scatter plot, display each set of clustered data;
- in the same scatter plot, add the cluster centers, using `c = 'black'` and `marker = '*'`

If you have chosen `k` well, your plot should show nicely clustered data.

6 Prepare for Exercise 4

For this exercise, we will consider an artificial dataset `X,y` of `n=2020` points in dimension `d=2`, with each point `X[i,:]` already assigned to a cluster `y[i]`. This data will break up naturally into a small number of clusters $k \leq 10$. As we did in the previous exercise, we will try to determine a value of `k` less than 10 that has a relatively small inertia.

Before we start our analysis, we will split the data into 2000 items of `training data`, `X1,y1`, and 20 items of `testing data`, `X2,y2`. We will do an intertia test on the `X1` data, choose a good value of `k`, and then cluster the data and plot it.

Then we will pretend that we have just found 20 new items of data, `X2,y2`. We can use KMeans to assign to each item `X2[i,:]` a cluster index `y2*[i]`. Now we compare the cluster information in `y2` (where the data item came from) to `y2*` (where KMeans thought it should go) and see how well we did in classifying the new data.

This may sound a little complicated, but the only bad part is some tricky indexing we have to do at the end.

7 Exercise 4:

Write a program `exercise4.py` and:

- use `np.loadtxt()` to read `X` from *blobs_data.txt*;
- use `np.loadtxt()` to read `y` from *blobs_clusters.txt*;
- force `y` to integer type by `y = y.astype ( int )`;
- use `np.loadtxt()` to read `C` from *blobs_centers.txt*;
- copy the first 2000 rows of `X` and `y` into `X1` and `y1`;
- copy the remaining 20 rows of `X` and `y` into `X2` and `y2`;
- make a scatterplot of X1;
- for $1 \leq k < 11$, set up `kmeans`, cluster `X1`, set `inertia[k-1]=kmeans.inertia_`;

- plot the values `k,inertia[k-1]`;
- based on the scatter plot and the inertia plot, choose a value of `k`;
- now, using only the value of `k` that you chose...
- * again define `kmeans=KMeans(n_clusters=k)`;
- * again compute `y1_pred=kmeans.fit_predict(X1)`, to cluster the data;
- now, we want to add the `X2` data, without affecting the clustering we've already done.
- compute `y2_pred=kmeans_predict(X2` ... Notice the difference?;
- For each new data item `X2[i,:]`, print `C[y2[i],:]` and `cluster_centers_[y2_pred[i],:]` ;

Data item X2[i,:] was generated by a random deviation from blob center C[y2[i],:]. KMeans doesn't know the values of these blob centers, but the cluster centers are its approximation to those values. Therefore, the best we can hope for is that every new data item X2[i,;] is matched with a cluster center that is the closest to its original blob center.

What we have done is to use the initial data X1 as a training set to estimate where the centers are. After that, we are able to try to **classify** the new data X2, according to which center it should be assigned. As long as we believe that KMeans did a good clustering job on the first set of data, then we have worked out a way to **classify** the new data automatically.