Assignment 8 - Clustering

Here we discuss about an unsupervised ML method Clustering (https://en.wikipedia.org/wiki/Cluster_analysis), which does not require a supervisor to provide the label values for any datapoint. Clustering methods aim to decompose datapoints into a few subsets which we refer to as clusters. They learn a hypothesis for assigning each data point either to one cluster or several clusters with different degrees of belonging.

Broadly speaking, clustering can be divided into two subgroups:

- **Hard Clustering**: In hard clustering, each data point either belongs to a cluster completely or not.
- **Soft Clustering**: In soft clustering, instead of putting each data point into a separate cluster, a probability or likelihood of that data point to be in those clusters is assigned.

Clustering of unlabeled data can be performed with the module `sklearn.cluster` (https://scikit-learn.org/stable/modules/classes.html#module-sklearn.cluster). Each clustering algorithm comes in two variants: a class (python class (https://docs.python.org/3/tutorial/classes.html)), that implements the `fit.()` method to learn the clusters on training data, and a function, that, given training data, returns an array of integer labels corresponding to the different clusters. For the class, the labels over the training data can be found in the `labels_` attribute. Here is the overview of clustering methods (https://scikit-learn.org/stable/modules/clustering.html#overview-of-clustering-methods).

In this assignment, we will use **K-means** to solve a **hard-clustering** task.

Learning goals

- Learn how to **construct** and **train** K-means for **hard-clustering** task using `scikit-learn`.
- Learn the importance of K-means initialization.
- Learn about the properties of K-means clusters.
- Learn how to use inertia and silhouette coefficient to choose the number of clusters k.
- Learn how to construct new features using clustering, and to leverage those for more expressive (supervised) models.

In [102]:

```
# Import basic libraries needed for this assignment
%config Completer.use_jedi = False  # enable code auto-completion
import numpy as np #import numpy to work with arrays
from sklearn.datasets import load_iris                  # function providing iri
s dataset
from sklearn.linear_model import LinearRegression
from sklearn.cluster import KMeans                      # library for K-Means cl
ustering
from sklearn import metrics                             # library providing scor
e functions, performance metrics and pairwise metrics and distance computations
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
from time import time                                   # library providing vari
ous time-related functions
import numpy as np                                      # library for numerical
 computations (vectors, matrices, tensors)
import matplotlib.pyplot as plt                         # library providing tool
s for plotting data
from mpl_toolkits.mplot3d import Axes3D                 # library for 3D axes ob
ject
import pandas as pd
```

Student task A8.1. K-means for hard-clustering problem.

Now, your task is to use **K-means** for a **hard-clustering** problem. You can use [`sklearn.cluster.KMeans`](https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans) to construct the model.

The **KMeans (https://en.wikipedia.org/wiki/K-means_clustering)** algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares (see below). This algorithm requires the number of clusters to be specified. It scales well to large number of samples and has been used across a large range of application areas in many different fields.

The k-means algorithm divides a set of N samples $X = \{x_1, x_2, \ldots, x_n\}$ into K disjoint clusters $C_1, C_2, \ldots, C_k$, each described by the mean μ_j of the samples in the cluster C_j, for $j \in \{1, 2, 3, \ldots, k\}$. The means are commonly called the cluster "centroids"; note that they are not, in general, points from X, although they live in the same space.

The K-means algorithm aims to choose centroids that minimise the **inertia**, or **within-cluster sum-of-squares criterion**:

$$\min_{\mu} \sum_{j=1}^{k} \sum_{x_i \in C_j} \|x_i - \mu_j\|^2$$

Furthermore, since the total variance in the data is a constant, by minimizing the within-cluster (intra) variance we are also implicitly *maximizing* the between-cluster (inter) variance. As a result, K-means tries to find well separated spherical clusters with equal densities. More specifically, K-means can only produce convex clusters, which makes it less useful when the targeted/real clusters are strongly overlapping or entangled. And finally, whether or not there is actually any "clustering/grouping" we as humans might recognize or wish to find in the data, K-means **will** return us k cluster centroids, and those centroids will partition the input space in convex partitions using euclidean distance. It is solely on the analyst to interpret and validate the results using ingenuity and existing tools (goodness measures, visual inspection etc.).

Step 1 - Loading Data

We will use the Iris plants dataset (https://scikit-learn.org/stable/datasets/toy_dataset.html#iris-dataset). It is a classic and very simple multi-class classification dataset. This dataset consists of 3 different types of irises' (Setosa, Versicolour, and Virginica) petal and sepal length, stored in a 150x4 numpy.ndarray. The rows being the samples and the columns being: Sepal Length, Sepal Width, Petal Length and Petal Width. We use these four features for clustering. The true labels, see below `iris_data.target`, will be used to evaluate the performance of Kmeans method.

In [103]:

```
# load dataset
iris_data = load_iris()    # load the iris dataset
X = iris_data.data    # obtain features of datapoints
y = iris_data.target    # obtain labels of datapoints

# shape of samples
print(f'Shape of samples {X.shape}')    # print the shape of features
print(f'Target names {iris_data.target_names}')    # print all label names
```

```
Shape of samples (150, 4)
Target names ['setosa' 'versicolor' 'virginica']
```

Step 2 - Defining and Training Model

We use `KMeans()` and `.fit()` to train our model. Read more parameter settings (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans) in details. Please set the `random_state` as 0, and the `n_clusters` (which is the parameter k in k-means) so that you could hope to capture the different types of irises' - see the targets in the previous code cell; we'll be comparing those against K-means labels.

In [104]:

```
# define and train model

# YOUR CODE HERE
kmeans = KMeans(n_clusters=3, random_state=0)
kmeans.fit(X)
```

Out[104]:

```
KMeans(n_clusters=3, random_state=0)
```

In [105]:

```
# this cell is for A8.1 tests
```

Step 3 - Evaluation

Cluster quality metrics evaluated (see Clustering performance evaluation (https://scikit-learn.org/stable/modules/clustering.html#clustering-evaluation) for definitions and discussions of the metrics):

Shorthand	full name
homo	homogeneity score
compl	completeness score
v-meas	V measure
ARI	adjusted Rand index
AMI	adjusted mutual information
silhouette	silhouette coefficient

NOTE: Homogeneity/Completeness score, V measure, ARI and AMI are all **external validity indices** - so called since they require labels (i.e., external information). Silhouette coefficient on the other hand only requires the data points and cluster assignments, and is an example of an **internal validity index**. Validity indices can be used to evaluate single clustering, choose k, or compare multiple clusterings. However, there are some nuances in the use of these measures and how they should be used with different clustering models, which goes beyond this course. These and other validity indices are dealt with in more depth in the course CS-E4650 (Methods of Data Mining).

In [106]:

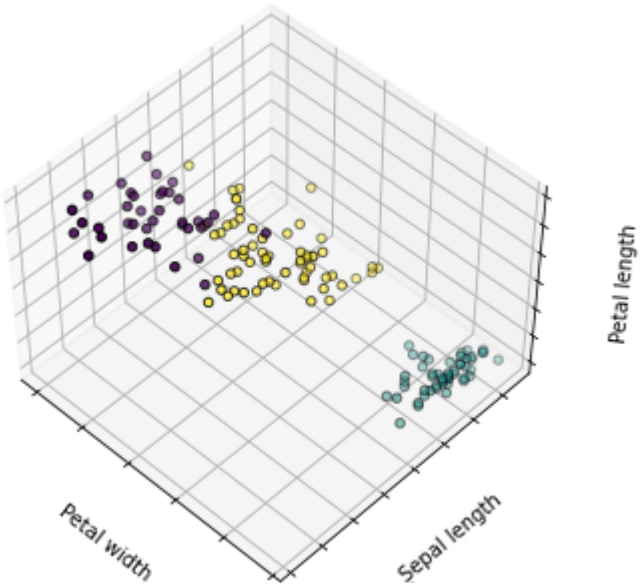
```
# calculate the running time
t0 = time()
estimator = KMeans(n_clusters=3, random_state=0).fit(X)
fit_time = time() - t0
results = [fit_time, estimator.inertia_]

# define the metrics which require only the true labels and estimator labels
clustering_metrics = [
        metrics.homogeneity_score,
        metrics.completeness_score,
        metrics.v_measure_score,
        metrics.adjusted_rand_score,
        metrics.adjusted_mutual_info_score,
]
results += [m(y, estimator.labels_) for m in clustering_metrics]

# the silhouette score requires the full dataset
results += [
        metrics.silhouette_score(
            X,
            estimator.labels_,
            metric="euclidean",
            sample_size=300,
    )
]

# show the results of each metric and print them
formatter_result = (
        "{:.3f}s\t{:.0f}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}"
)
print(66 * "_")
print("time\tinertia\thomo\tcompl\tv-meas\tARI\tAMI\tsilhouette")
print(formatter_result.format(*results))
print(66 * "_")
```

```
__________________________________________________________________
time    inertia homo    compl   v-meas  ARI     AMI     silhouette
0.017s  79      0.751   0.765   0.758   0.730   0.755   0.553
__________________________________________________________________
```

Let's visualize the clustering results.

In [107]:

```
fig = plt.figure()    # create a new figure
ax = Axes3D(fig, elev=48, azim=134)    # use 3D axes
labels = kmeans.labels_    # obtain clustering labels
ax.scatter(X[:, 3], X[:, 0], X[:, 2], c=labels.astype(float), edgecolor='k')
# plot a scatter plot of labels vs. X with varying marker size or color
ax.w_xaxis.set_ticklabels([])    # leave the text values blank of the tick label
s
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])
ax.set_xlabel('Petal width')    # set the label for the x/y/z-axis
ax.set_ylabel('Sepal length')
ax.set_zlabel('Petal length')
plt.show()
```



Demo

The importance of initialization

Here you can visually explore the sensitivity of K-means to initialization, and the effect standardization can have on it.

The problem of finding k globally optimal centroids is strictly speaking too hard for us to solve in finite time. We can still find *good* solutions using an iterative approach (see 'algorithm 12' in ML book chapter 8.1). However, the solutions we find are only locally optimal and there is always the risk that we get stuck in a very suboptimal solution. Most importantly, this means that our solution depends on our initial guess for the centroids. Therefore, it is usually a good practice to run the K-means algorithm multiple times using different initial centroids, and then choose based on some goodness measure or a validation index - or if the data is low dimensional, based on visualization.

First, let's create a toy dataset with 2 distinct clusters:

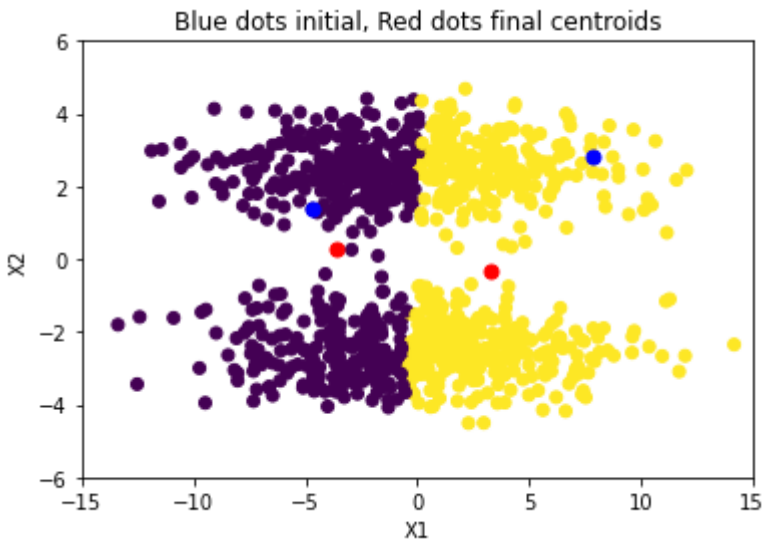
In [108]:

Now, let's choose 2 random points from the dataset as our initial cluster centroids, and then cluster and plot the result as well as the initial centroids. The 'n_init=1' is specifying that we do not want multiple re-starts of the algorithm, as we are using our own initial centroids - sklearn's KMeans makes 10 re-starts by default.

NOTE: You can keep running the cell multiple times (hold shift + press enter)

NOTE: You can also try a more sophisticated initialization scheme (`k-means++` (https://en.wikipedia.org/wiki/K-means%2B%2B)) - it is the sklearn's default K-means initialization when you don't provide the initial centroids.

In [109]:



Only very rarely are the initial centroids chosen such that the algorithm is able to distinguish the 2 distributions (if you keep running the cell you will eventually succeed).

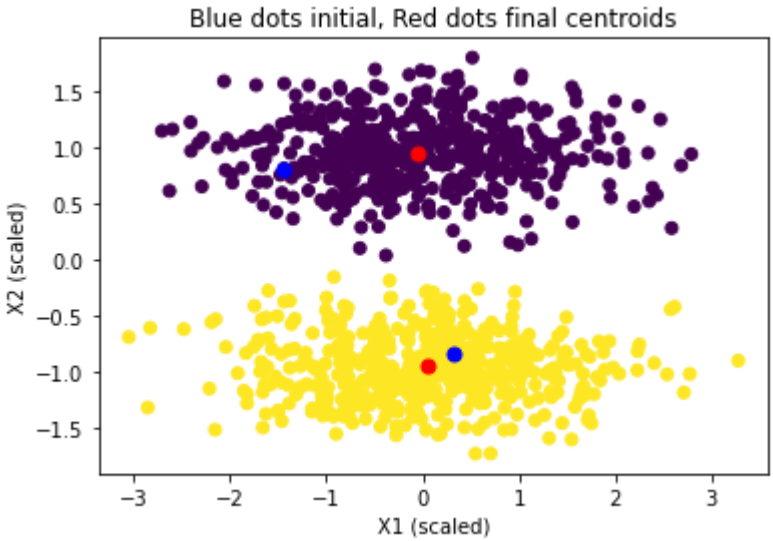
In this particular case, we can try to mitigate this sensitivity to initialization by standardizing our features using Sklearn's 'StandardScaler' (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html):

In [110]:

In [111]:

And again using either randomly chosen data points or k-means++ for initial centroids:

In [112]:



(NOTE the different X1, X2 scales)

Now K-means has no trouble finding the clusters, even with poor initial centroids. As to the effectiveness of scaling, this is a somewhat "cherry-picked" example: In real world task, decision to normalize or standardize your data should be based on your understanding of the data. E.g., if there are features that do not have the same units and are on wildly differing scales, while your prior assumption is that those features are likely to be equally significant for the learning task; it is appropriate to standardize them. However, if your features have the same units (e.g., both are m/s) and you have no information based on which you could conclude about their relative significance, you should not scale the data.

Student Task A8.2: Standardization clearly had an effect on convergence in the previous demo. Based on the scatter plot of the data, could you come up with an even simpler feature transformation that would make the clustering task easier? If you feel unsure about the answer, you can always modify the previous demo and try different approaches, or plot histograms of the features. Answer by setting the 'Answer' variable to the index of the answer that you consider correct. * Answer 1: Shift the data by some constant factor. * Answer 2: Use just one of the features - either one will work. * Answer 3: Use just one of the features - 2nd feature will work.

In [186]:

In [187]:

Demo

The nature of K-means solutions.

Here we can inspect how the clustering evolves through iterations. As explained, in addition to minimizing inertia, the K-means objective is implicitly also maximizing the separation between clusters:

K-means aims to minimize:

$$\sum_{x_i \in C_j} \|x_i - \mu_j\|^2$$

for each cluster. Equivalently we could minimize within-cluster pairwise squared deviations:

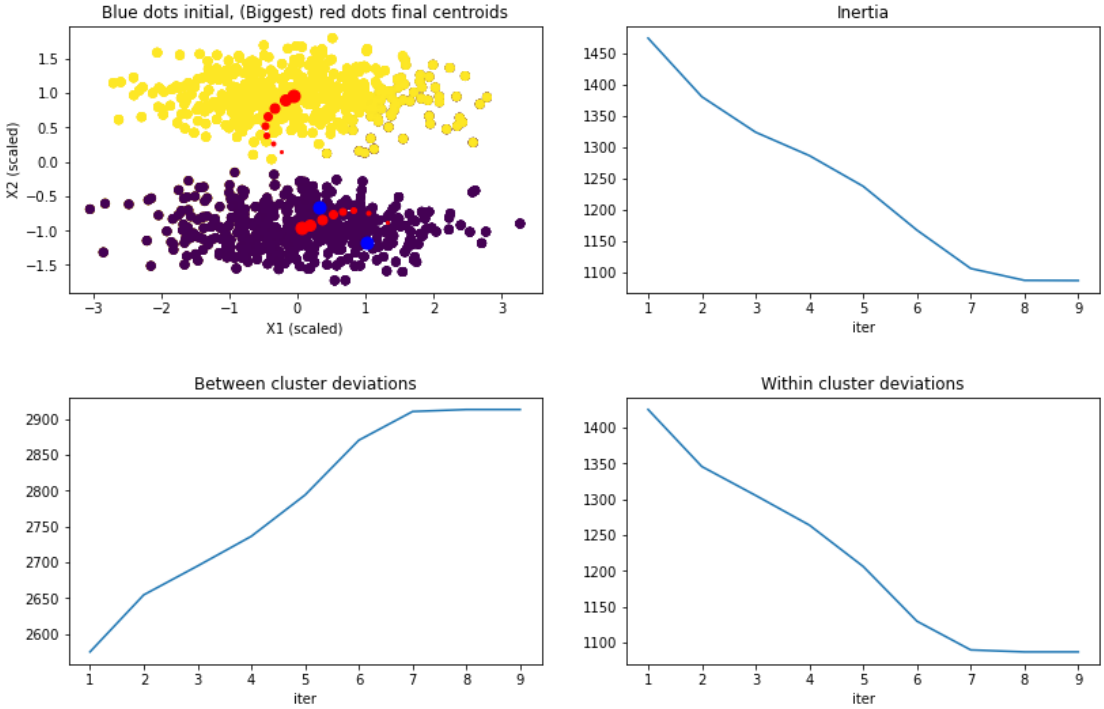
$$\frac{1}{|C_j|} \sum_{x,y \in C_j} \|x - y\|^2$$

and as the total variance in the data remains constant, minimizing the within-cluster deviations will also result in maximizing the pairwise deviations for data points in different clusters (between-cluster deviations).

NOTE: You can change the random seed to test different initializations.

In [115]:

```
Stop Iteration at
9th step
Final centroids:
[[ 0.04 -0.95]
 [-0.04  0.95]]
```



Student task A8.3. Choosing k for K-means. Now, your task is to use inertia and silhuette coefficient to choose a good value for parameter k. You can use [`sklearn.cluster.KMeans`](https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans) to construct the model, it's attribute **.inertia_** for inertia, and [`sklearn.metrics.silhuette_score`](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html) for silhuette coefficient.

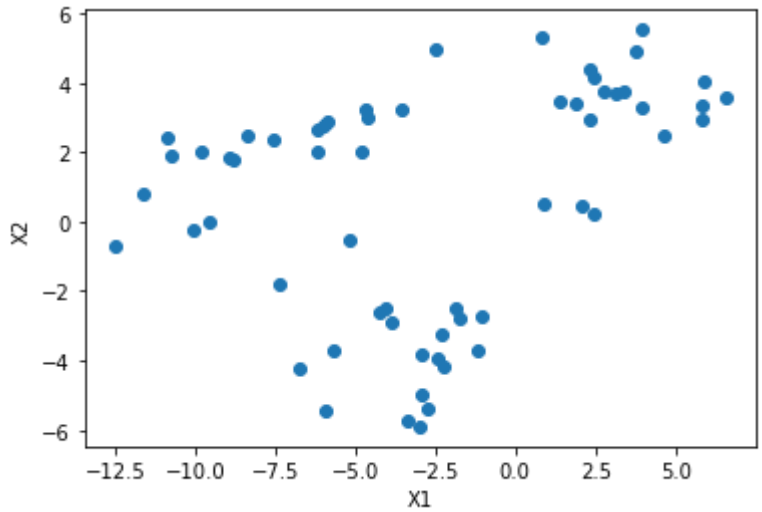
The K-means model requires us to provide the parameter k. With low-dimensional or otherwise trivial data this might be easy to conclude based on visual exploration. However, in many real world scenarios it is hardly a trivial task. In addition to previously mentioned validity indices, the value of the clustering objective (**inertia**) itself can be used to evaluate the clustering at different values of k. And properly interpreted, the inertia can help us to choose the value for k.

The most basic approach to choosing k is called **the elbow method**. In elbow method we plot the value of inertia with different values of k, and choose the smallest k that produces "sufficiently low values of inertia." In practice this is done by looking for the point where the inertia plot is becoming less steep ("elbow" in the curve). This typically occurs when K-means is in some sense starting to overfit, i.e., instead of adding new clusters to capture truly separated sets of data points, it is starting to partition already compact clusters into smaller parts which are not well separated.

As an additional method, we also use Silhuette coefficient/index (**SI**) to infer a good value for k. Higher values are better, and ideally we'd hope to find a peak.

Let's load and visualize the data:

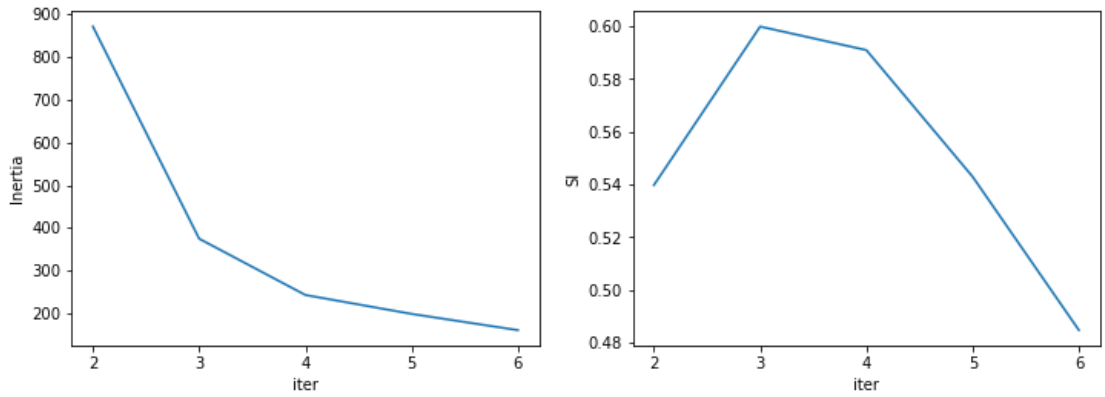
In [116]:



Your task:

Fit K-means model using the whole data X, for each value of k=2,3,...,6, and calculate SI for each instance and collect the inertias and SI's in python lists. Set the K-means random_state=0.

In [190]:

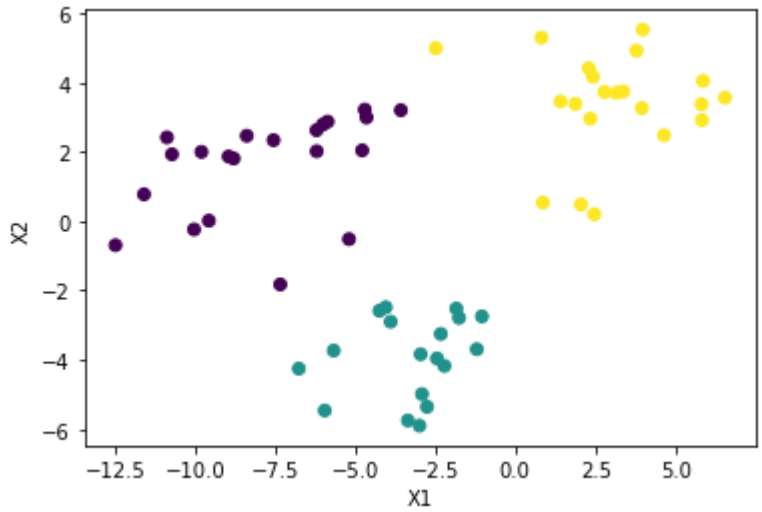


In [118]:

In [119]:

There is some ambiguity based on the plots whether k value should be 3 or 4. Choosing the value is seldom a trivial task, especially with high-dimensional data as we cannot rely on visual inspections for validation. You can fit and plot the kmeans for values k=3,4, to see if you find one superior to other.

In [120]:



Demo

The nature of K-means solutions (continued).

We can also inspect how the algorithm partitions the input space.

In [121]:

We will use the same data as in the previous task. First, let's create a meshgrid over the input space - this helps us to visualize the cluster assignments in the input space more broadly:

In [122]:

Then, we create an iteration counter and choose initial centroids at random from the data.

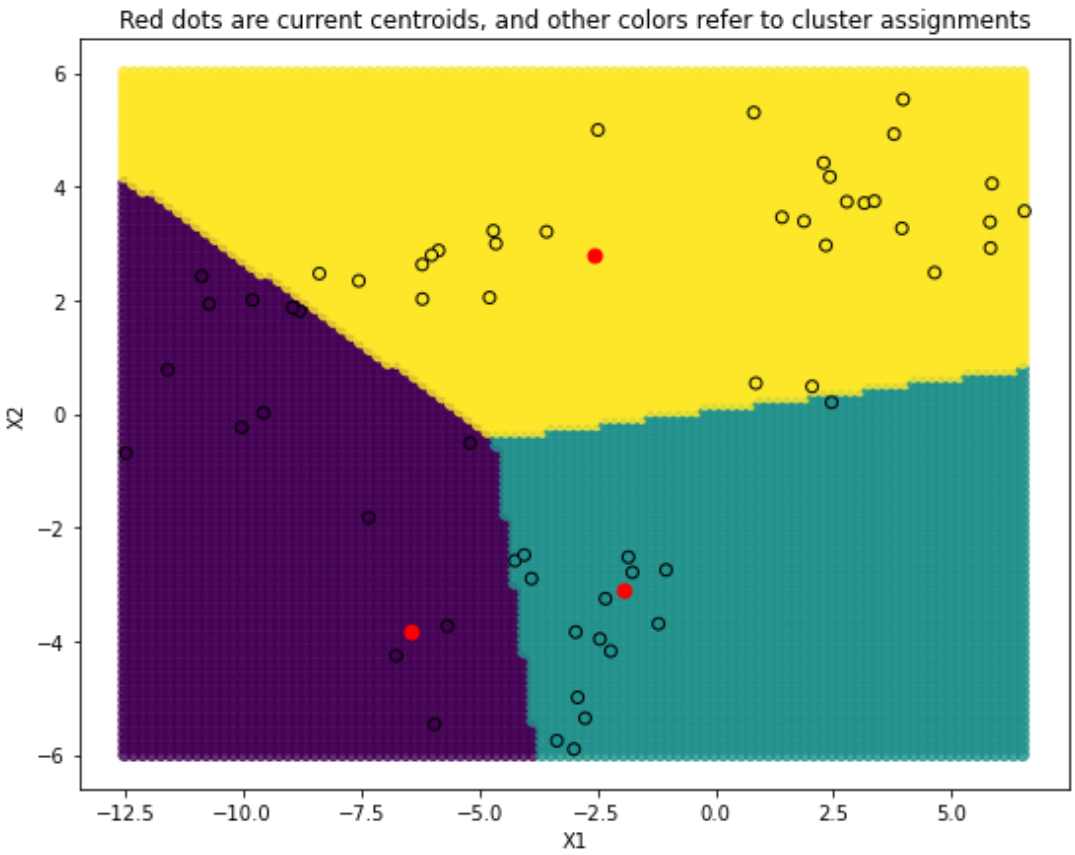
NOTE: You need to run this cell before starting to run the cell after it.

In [123]:

Now, we can run the K-means one iteration at the time, by setting 'max_iter=1' and taking the new updated centroids as the next initial points. We are also using the trained K-means instance to predict for those previously created grid points.

NOTE: You can hold 'Shift' and press 'Enter' to run the cell one time. When the clustering objective is no longer improving, the code will print a message to stop iterating. If you wish to repeat this task, e.g., with different initial points, just run the above code cell again with necessary modifications.

In [124]:



Student Task A8.4: Answer the following quiz questions by setting the corresponding variable to the index of the answer that you consider correct. Question 1: For an arbitrary dataset, can we choose k such that clustering error goes to zero? - Answer 1: Yes - Answer 2: No - Answer 3: It depends on the data. Question 2: Do you agree with the following statement: The lower the clustering error, i.e., the value of the clustering objective/criterion, more faithfully the cluster centroids represent the data (imagine replacing the data points with their assigned cluster centroids). - Answer 1: Yes. - Answer 2: No. Question 3: Do you agree with the following statement: Scikit-learn function kmeans.fit( ) will always determine k cluster means/centroids. - Answer 1: Yes. - Answer 2: No. - Answer 3: It depends on the data.

In [191]:

In [192]:

In [193]:

In [194]:

Student task A8.5. ## Using clustering to construct new features. Now, your task is to leverage K-means clustering to construct new features for a supervised learning task using linear regression.

Assuming our dataset consists of four data points:

$$\mathbf{X} = \begin{pmatrix} x^{(1)} \\ x^{(2)} \\ x^{(3)} \\ x^{(4)} \end{pmatrix}$$

which have been clustered into 2 clusters according to:

$$\mathbf{Y} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

We can construct new features $\mathbf{x_1}, \mathbf{x_2}$ using the cluster assignments. We do it using the following logic: If the i'th data point $x^{(i)}$ is assigned to cluster 0, we set $x_1^{(i)} = x^{(i)}$, and if it is assigned to cluster 1, we set $x_2^{(i)} = x^{(i)}$. Our 4 datapoints, first 2 assigned to cluster 0, and the last 2 to cluster 1, would produce the following features:

$$\mathbf{X_{new}} = \begin{pmatrix} x^{(1)} & 0 \\ x^{(2)} & 0 \\ 0 & x^{(3)} \\ 0 & x^{(4)} \end{pmatrix} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ x_1^{(3)} & x_2^{(3)} \\ x_1^{(4)} & x_2^{(4)} \end{pmatrix}$$

This is effectively the same as partitioning the data using the cluster assignments into 2 (or k more generally) disjoint datasets and training a distinct model with each partition. So, with the previous example we would get 2 models:

$$Model_1 : \alpha_1 x_1 + \beta$$

and

$$Model_2 : \alpha_2 x_2 + \beta$$

However, this approach is limited by the fact that both models have a shared constant term β, which may or may not be appropriate for the task at hand. We can make the "combined" model slightly more expressive by adding a separate constant term $\mathbf{x_c}$ in our features for cluster 1:

$$\mathbf{X_{new}} = \begin{pmatrix} x^{(1)} & 0 & 0 \\ x^{(2)} & 0 & 0 \\ 0 & x^{(3)} & 1 \\ 0 & x^{(4)} & 1 \end{pmatrix}$$

which equals training 2 full models (on their respective partitions of the data):

$$Model_1 : \alpha_1 x_1 + \beta_1$$

and

$$Model_2 : \alpha_2 x_2 + \beta_2$$

or a combined model:

$$Model_{1,2} : \alpha_1 x_1 + \alpha_2 x_2 + \beta_2 x_c + \beta_1$$

(this parameterization assumes the linear model has an intercept term)

Below is a class for constructing the features:

Study how the class is implemented, and what are the parameters. It's fit, transform and fit_transform addhere to the same logic as sklearn's classes you have previously used.

In [164]:

Here is an "alias" for mse function:

In [165]:

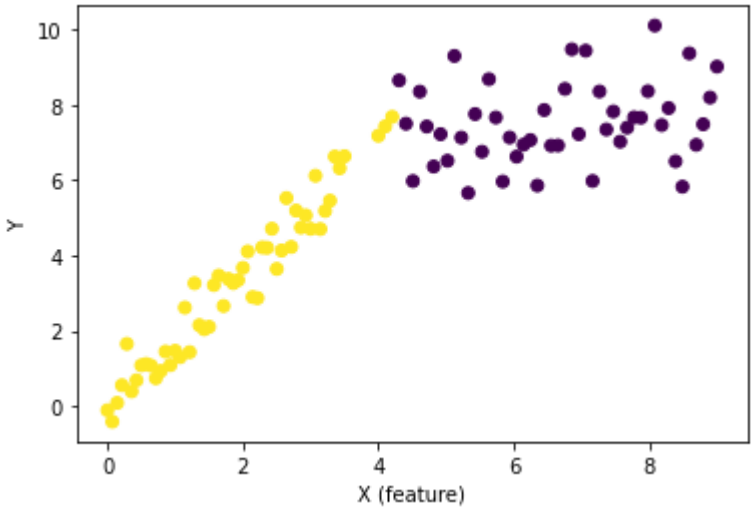
you can use it in place of the mean_squared_error function. This is sometimes handy to make your code more concise and less cluttered.

Let's load some toy data:

In [179]:

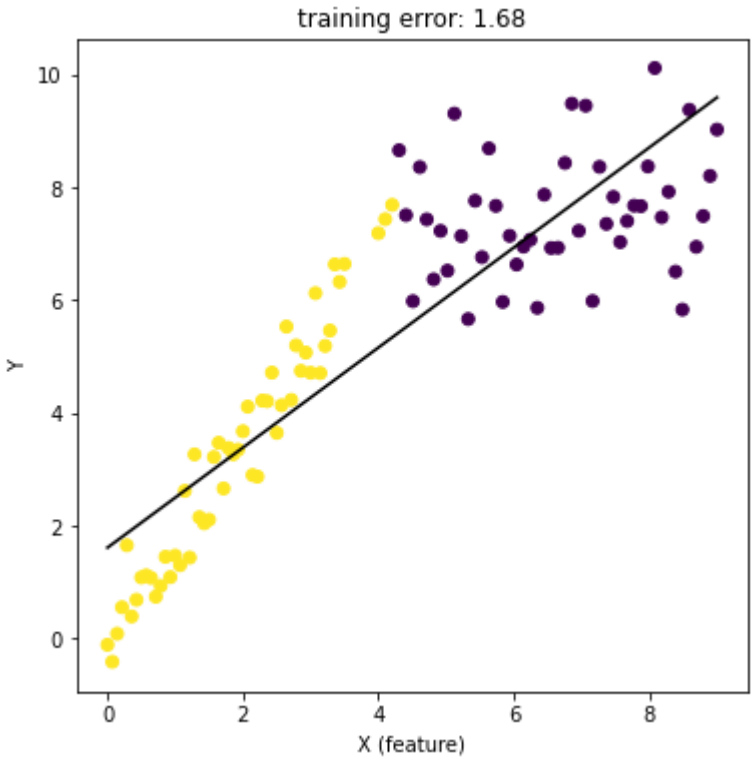
... and see how the data is clustered (based on the feature alone):

In [180]:



Let's train a simple LinearRegression on the training data. We will also define a grid of evaluation points for predicting with the trained model. We use numpy's `linspace` (https://numpy.org/doc/stable/reference/generated/numpy.linspace.html) to get 50 equally spaced data points $x_{eval} \in [0, 9]$. This linear model is our baseline solution:

In [181]:



Clearly, the data seems to have 2 different modalities, and the fit is a compromise between the two.

In [182]:

Now, your task is to try to improve the model with clustered features:

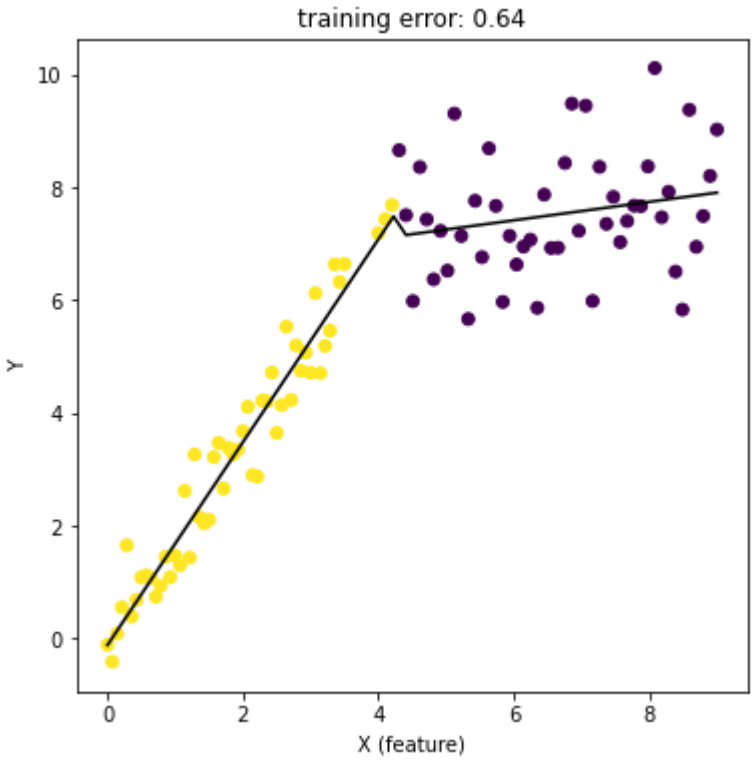
You can use the same grid of evaluation points x_{eval}. In this code cell you only need to construct the features, and in the next one you train the model.

In [195]:

In [196]:

And now you need to fit the model on those features, predict and calculate the training error.

In [202]:



In [203]:

In [204]:

In [205]:

END