# **AdaBoost**

A quick reminder on how AdaBoost works: <https://youtu.be/LsK-xG1cLYA> BAM! Note that this video uses Gini index instead of Entropy and resamples the dataset instead of summing the error weights, but the general approach is the same as the one presented in the lecture.

Key ideas:

* training: train multiple weak learners and assign them a weighted vote
* classification: predict the class that gets the most weighted votes

Advantages:

* simple (few parameters)
* robust to noisy training
* performs well in practice

Disadvantages:

* needs a termination condition (requires meta-optimisation)
* sensitive to outliers

## **Classification**

Given a dataset like the following:

In [1]:

**import** pandas **as** pd

**import** matplotlib.pyplot **as** plt

d **=** pd.DataFrame({

'X1': [**-**1, 0, 0, 1, 1, 1, 2, 2],

'X2': [1, 1, 2, **-**1, 0, 2, 2, 3],

'Y': [0, 1, 0, 0, 1, 1, 0, 1]

})

X, Y **=** d[['X1', 'X2']], d['Y']

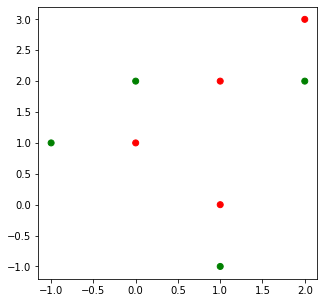
​

c**=** ['green' **if** l **==** 0 **else** 'red' **for** l **in** Y]

fig, ax **=** plt.subplots(figsize**=**(5, 5))

plt.scatter(X['X1'], X['X2'], color**=**c)

plt.show()



we can use AdaBoostClassifier from sklearn.ensemble to classify the point at (0, 0) using 5 weak learners with:

In [2]:

**from** sklearn.ensemble **import** AdaBoostClassifier

ab **=** AdaBoostClassifier(n\_estimators**=**5).fit(X, Y)

ab.predict([[0, 0]])

Out[2]:

array([1])

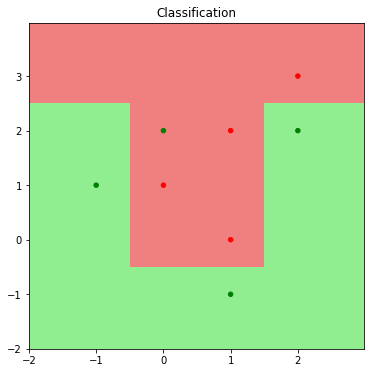
We can visualise the decision surface using matplotlib:

In [3]:

**from** tools.plots **import** plot\_decision\_surface

knn **=** AdaBoostClassifier(n\_estimators**=**5).fit(X, Y)

plot\_decision\_surface(knn, X, Y)

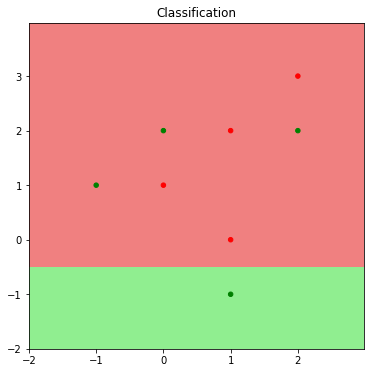


In the particular case where the number of estimators is 1, the decision surface corresponds of a simple decision tree with a single split (usually called a decision stump):

In [4]:

knn **=** AdaBoostClassifier(n\_estimators**=**1).fit(X, Y)

plot\_decision\_surface(knn, X, Y)



## **Accuracy**

Training and cross-validation can be performed just like for any algorithm, using AdaBoostClassifier.score() and sklearn.model\_selection.LeaveOneOut.

# **Agglomerative hierarchical clustering**

A quick reminder on how agglomerative hierarchical clustering works: <https://youtu.be/OcoE7JlbXvY>

Goal:

* similar objects in same group
* dissimilar objects in different groups

Approach:

* form all possible singleton clusters (each containing a single object);
* greedily combine clusters with "maximum similarity" (or "minimum distance") together into a new cluster;
* continue until all objects are contained in a single cluster.

Different types of similarity:

* *single link* - similarity of two clusters considered for merging is determined by the two most similar members of the two clusters;
* *complete link* - similarity of two clusters is determined by the two least similar members of the two clusters;
* *group average* - similarity is determined by the average similarity between all members of the clusters considered.

Considering the following uni-dimensional dataset:

In [5]:

d **=** pd.DataFrame({

'X': [**-**2.2, **-**2.0, **-**0.3, 0.1, 0.2, 0.4, 1.6, 1.7, 1.9, 2.0],

})

Plotting it with pyplot shows roughly three "natural" clusters forming:

In [6]:

**import** numpy **as** np

plt.scatter(d, np.zeros\_like(d))

plt.show()



Using the sklearn.cluster module, we can use the agglomerative cluster algorithm with any of the three similarity (or "linkage") functions. For instance, if we want to use single linkage to find 3 clusters in the above dataset, we could run:

In [7]:

**from** sklearn.cluster **import** AgglomerativeClustering

​

ac **=** AgglomerativeClustering(linkage**=**'single', n\_clusters**=**3)

clusters **=** ac.fit\_predict(d)

​

print(clusters)

[1 1 0 0 0 0 2 2 2 2]

We can use colors to represent them in the original scatter plot:

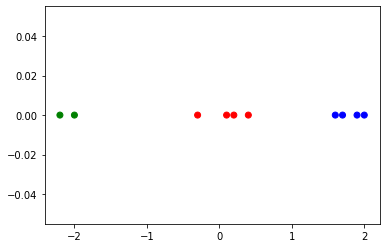
In [8]:

**from** matplotlib.colors **import** ListedColormap

c **=** ListedColormap(['red', 'green', 'blue'])

plt.scatter(d, np.zeros\_like(d), c**=**clusters, cmap**=**c)

plt.show()



The result reflects the three "natural" clusters.

However, we do not always know how many clusters to expect. In this case we can simply generate a dendrogram to help us find the optimum number of clusters. For this, we can set the number of clusters to None and the distance threshold to 0, which will cause the algorithm to start with singleton clusters (effectively assigning each point to its own cluster) and then aggregating those into larger and larger ones:

In [9]:

**from** tools.plots **import** plot\_dendrogram

​

**for** linkage **in** ['single', 'complete', 'average']:

*# setting distance\_threshold=0 ensures we compute the full tree.*

ac **=** AgglomerativeClustering(linkage**=**linkage, distance\_threshold**=**0, n\_clusters**=None**)

ac **=** ac.fit(d)

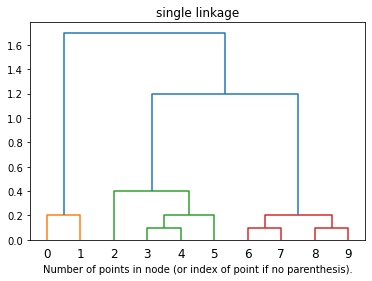
​

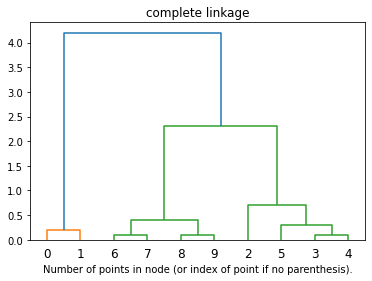
plot\_dendrogram(ac, truncate\_mode**=**'level', p**=**4)

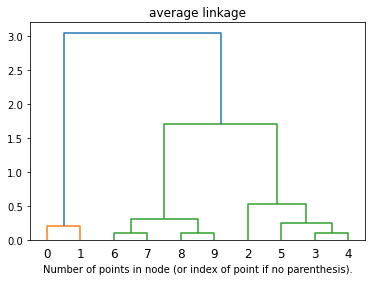
plt.title(f"{linkage} linkage")

plt.xlabel("Number of points in node (or index of point if no parenthesis).")

plt.show()







From the scipy documentation for [hierarchy.dendrogram](https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.dendrogram.html#scipy.cluster.hierarchy.dendrogram):

The dendrogram illustrates how each cluster is composed by drawing a U-shaped link between a non-singleton cluster and its children. The top of the U-link indicates a cluster merge. The two legs of the U-link indicate which clusters were merged. The length of the two legs of the U-link represents the distance between the child clusters. It is also the cophenetic distance between original observations in the two children clusters.

In our examples above, we can then see that the same thee clusters tend to emerge.

We can access the data behind the dendrogram through the children\_ and distances\_ attributes:

In [12]:

ac **=** AgglomerativeClustering(linkage**=**'single', distance\_threshold**=**0, n\_clusters**=None**)

ac **=** ac.fit(d)

ac.children\_

Out[12]:

array([[ 6, 7],

[ 3, 4],

[ 8, 9],

[10, 12],

[11, 5],

[ 0, 1],

[ 2, 14],

[16, 13],

[15, 17]])

children\_ lists the clusters in the order in which they are merged. For instance in the example above, the nodes 6 and 7 were merged first, forming a new cluster 10 (cluster indices start from 0). The second cluster merges 3 and 4 and creates cluster 11 and so on.

In [11]:

ac.distances\_

Out[11]:

array([0.1, 0.1, 0.1, 0.2, 0.2, 0.2, 0.4, 1.2, 1.7])

The distances\_ attribute shows the distance for each cluster join from children\_. For instance, the distance between the first two clusters merged (6 and 7) is 0.1, while the distance between the last two clusters merged (15 and 17) is 1.7.

The dendrogram can help us find the natural clusters by finding biggest distance between two consecutive merges. If we look at the distances\_ attribute, we can see that the largest "jump" is from 0.4 to 1.2 (

1.2−0.4=0.8

1.2−0.4=0.8 is the largest difference between two consecutive numbers). This means that the last two merges bring together rather distant clusters, so if we skip the last two merges, we end up with 3 clusters, as expected.

# **k-Means**

A quick reminder on how k-Means works: <https://youtu.be/4b5d3muPQmA> BAM?

Key ideas:

* a cluster is determined by the nearest centroid;
* the centroids are re-positions within the new clusters.

Advantages:

* simple;
* efficient on large datasets;
* able to incorporate new data without a complete retrain.

Disadvantages:

* needs to know the number of clusters in advance;
* each run can lead to different results;
* sensitive to outliers (can "trap" a centroid);
* sensitive to the number and scale of dimensions - curse of dimensionality (same as for k-NN).

Considering the following uni-dimensional dataset:

In [1]:

**import** pandas **as** pd

d **=** pd.DataFrame({

'X': [**-**2.2, **-**2.0, **-**0.3, 0.1, 0.2, 0.4, 1.6, 1.7, 1.9, 2.0],

})

​

*# Plot the dataset*

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

plt.scatter(d, np.zeros\_like(d))

plt.show()



we can run the k-means alogrithm for 3 clusters using sklearn with:

In [2]:

**from** sklearn.cluster **import** KMeans

​

km **=** KMeans(n\_clusters**=**3, random\_state**=**1)

clusters **=** km.fit\_predict(d)

​

print(clusters)

[1 1 0 0 0 0 2 2 2 2]

And, as usual, we can represent them in the original scatter plot:

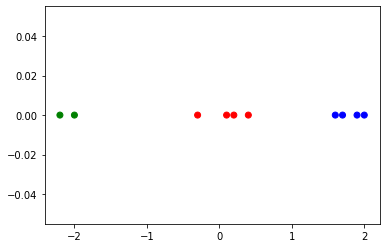
In [3]:

**from** matplotlib.colors **import** ListedColormap

c **=** ListedColormap(['red', 'green', 'blue'])

plt.scatter(d, np.zeros\_like(d), c**=**clusters, cmap**=**c)

plt.show()



We can also visualise the current centroids, which are accessible through the cluster\_centers\_ attribute:

In [4]:

x\_centroids **=** km.cluster\_centers\_[:,0]

print('Centroids: ', x\_centroids)

*# Scatter plot including centroids*

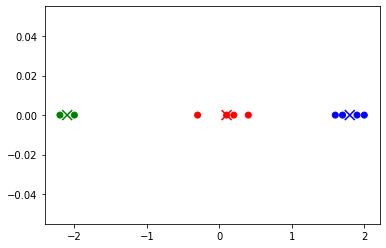
plt.scatter(d, np.zeros\_like(d), c**=**clusters, cmap**=**c)

plt.scatter(x\_centroids, np.zeros\_like(x\_centroids), s**=**100, marker**=**'x',

c**=**range(len(x\_centroids)), cmap**=**c)

plt.show()

Centroids: [ 0.1 -2.1 1.8]



## **The random nature of the algorithm**

In its simplest form, the algorithm will initialise the centroids by choosing random starting points. We can visualise this by setting the init parameter to random and running a single initialization with a single step, using various random seeds:

In [5]:

**for** seed **in** range(3):

km **=** KMeans(init**=**'random', n\_init**=**1, max\_iter**=**1, n\_clusters**=**3, random\_state**=**seed)

clusters **=** km.fit\_predict(d)

x\_centroids **=** km.cluster\_centers\_[:,0]

*# Scatter plot including centroids*

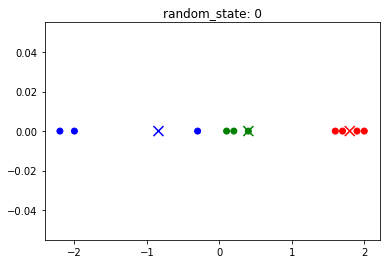
plt.scatter(d, np.zeros\_like(d), c**=**clusters, cmap**=**c)

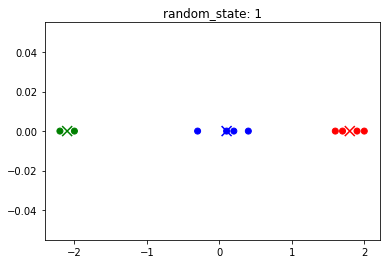
plt.scatter(x\_centroids, np.zeros\_like(x\_centroids), s**=**100, marker**=**'x',

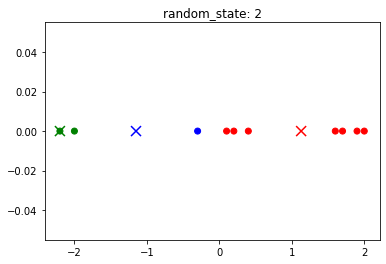
c**=**range(len(x\_centroids)), cmap**=**c)

plt.title(f'random\_state: {seed}')

plt.show()







## **Convergence criteria**

The algorithm will keep updating the centroids until their position no longer changes. We can visualise this by using different values for the max\_iter parameter (controlling the maximum number of iterations):

In [6]:

**for** max\_iter **in** range(1,6):

km **=** KMeans(init**=**'random', n\_init**=**1, max\_iter**=**max\_iter, n\_clusters**=**3, random\_state**=**0)

clusters **=** km.fit\_predict(d)

x\_centroids **=** km.cluster\_centers\_[:,0]

*# Scatter plot including centroids*

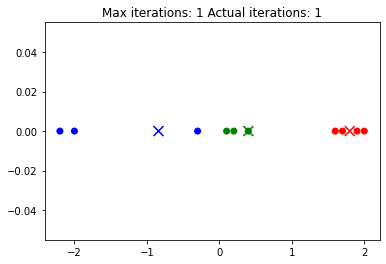
plt.scatter(d, np.zeros\_like(d), c**=**clusters, cmap**=**c)

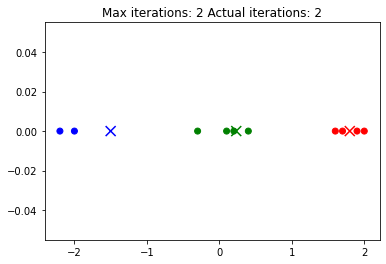
plt.scatter(x\_centroids, np.zeros\_like(x\_centroids), s**=**100, marker**=**'x',

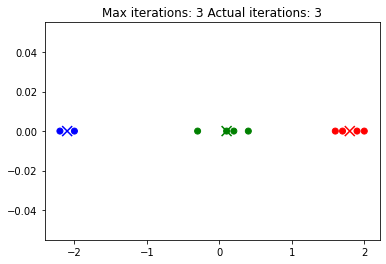
c**=**range(len(x\_centroids)), cmap**=**c)

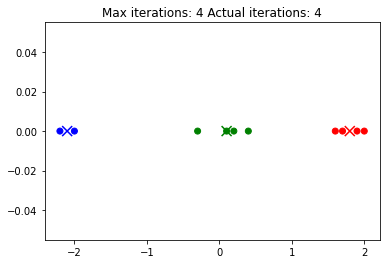
plt.title(f'Max iterations: {max\_iter} Actual iterations: {km.n\_iter\_}')

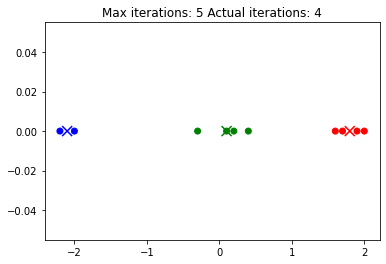
plt.show()











Since the centroids (and implicitly the clusters) have not changed between the third and fourth iteration, the algorithm is said to have *converged* (has reached a stable state) and stops after four iterations.

## **Specifying initial centroids**

By default, the initial centroids are chosen either randomly or using a heuristic. However, starting points can be specified using the init parameter. In this case, the random\_state parameter is no longer needed and n\_init must be 1 to avoid a warning. For instance, if we want to start the algorithm above with the centroids at 1.7, 1.9 and 2 (the right-most points), we could do:

In [7]:

init\_centroids **=** np.array([[1.7], [1.9], [2]])

km **=** KMeans(init**=**init\_centroids, n\_init**=**1, max\_iter**=**1, n\_clusters**=**3, verbose**=**1)

Note that we also used max\_iter=1 to stop the algorithm after 1 iteration to better see the effect of this initialisation:

In [8]:

clusters **=** km.fit\_predict(d)

x\_centroids **=** km.cluster\_centers\_[:,0]

*# Scatter plot including centroids*

plt.scatter(d, np.zeros\_like(d), c**=**clusters, cmap**=**c)

plt.scatter(x\_centroids, np.zeros\_like(x\_centroids), s**=**100, marker**=**'x',

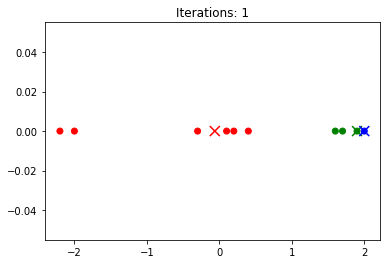
c**=**range(len(x\_centroids)), cmap**=**c)

plt.title(f'Iterations: {km.n\_iter\_}')

plt.show()

Initialization complete

Iteration 0, inertia 39.41



As expected, the centroids are still visibly positioned to the right.

## **Empty centroids exception**

There is one particularity of the sklearn implementation worth mentioning, which is not part of the basic algorithm: when a cluster is empty, the centroid will be repositioned immediately without waiting for the regular steps of the algorithm to change them. For instance, if we start with all 3 centroids at the right of the dataset, after the first iteration all points should be in a single cluster, while the remaining two centroids should be empty, so the situation should be even more unbalanced than above. However, if we try this and start with 3, 4, 5 as centroids, we instead get:

In [9]:

init\_centroids **=** np.array([[3], [4], [5]])

km **=** KMeans(init**=**init\_centroids, n\_init**=**1, max\_iter**=**1, n\_clusters**=**3, verbose**=**1)

clusters **=** km.fit\_predict(d)

x\_centroids **=** km.cluster\_centers\_[:,0]

*# Scatter plot including centroids*

plt.scatter(d, np.zeros\_like(d), c**=**clusters, cmap**=**c)

plt.scatter(x\_centroids, np.zeros\_like(x\_centroids), s**=**100, marker**=**'x',

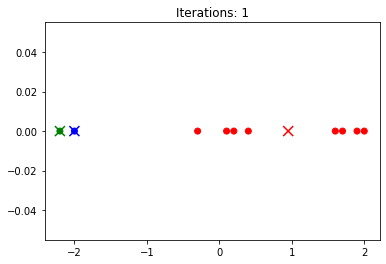
c**=**range(len(x\_centroids)), cmap**=**c)

plt.title(f'Iterations: {km.n\_iter\_}')

plt.show()

Initialization complete

Iteration 0, inertia 91.8



We find 2 of the three centroids on the extreme left of the dataset instead of the right. This prevents the algorithm from getting 'stuck' with centroids completely outside of the dataset.

# **k-Means**

## **Intra-cluster cohesion**

An important property of the k-Means algorithm is that at every steps

𝑡

t it greedily increases the *intra-cluster cohesion* by minimising the least squared distance between each point and its centroid (also called *inertia*, or the

𝐽

J criterion) defined as:

𝐽(

𝐶

𝑡

,

𝜇

𝑡

)=

∑

𝑖=1

𝑛

‖

𝑥

𝑖

−

𝜇

𝑡

𝐶

𝑡

(

𝑥

𝑖

)

‖

2

=

∑

𝑖=1

𝑛

(

𝑥

𝑖

−

𝜇

𝑡

𝐶

𝑡

(

𝑥

𝑖

)

)⋅(

𝑥

𝑖

−

𝜇

𝑡

𝐶

𝑡

(

𝑥

𝑖

)

)

J(Ct,μt)=∑i=1n‖xi−μCt(xi)t‖2=∑i=1n(xi−μCt(xi)t)⋅(xi−μCt(xi)t)

where:

* 𝐶
* 𝑡
* =(
* 𝐶
* 𝑡
* 1
* ,
* 𝐶
* 𝑡
* 2
* ,...,
* 𝐶
* 𝑡
* 𝐾
* )
* Ct=(C1t,C2t,...,CKt) is the set of all clusters (also called the K-partition) at step
* 𝑡
* t;
* 𝜇
* 𝑡
* =(
* 𝜇
* 𝑡
* 1
* ,
* 𝜇
* 𝑡
* 2
* ,...,
* 𝜇
* 𝑡
* 𝐾
* )
* μt=(μ1t,μ2t,...,μKt) is the set of centroids at step
* 𝑡
* t;
* 𝐶
* 𝑡
* (
* 𝑥
* 𝑖
* )
* Ct(xi) is the cluster corresponding to element
* 𝑥
* 𝑖
* xi at step
* 𝑡
* t;
* the "
* ⋅
* ⋅" operator is the scalar multiplication for vectors in
* ℝ
* 𝑑
* Rd;
* "
* ‖‖
* ‖‖" denotes the Euclidean norm.

The sklearn implementation of the algorithm provides this metric through the inertia\_ parameter. We can confirm its value by running the algorithm on a simple dataset:

In [1]:

**from** sklearn.cluster **import** KMeans

**from** matplotlib.colors **import** ListedColormap

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** pandas **as** pd

​

*# Create a simple dataset*

d **=** pd.DataFrame({

'X': [1, 2, 3, 10, 13, 16],

})

​

c **=** ListedColormap(['red', 'green', 'blue'])

​

*# Find clusters*

km **=** KMeans(n\_clusters**=**2, random\_state**=**1)

clusters **=** km.fit\_predict(d)

​

*# Take the centroids*

x\_centroids **=** km.cluster\_centers\_[:,0]

​

*# Scatter plot including centroids*

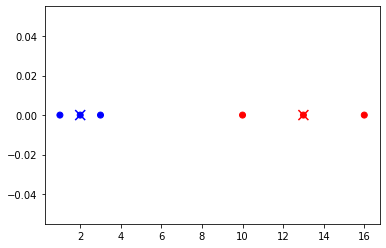
plt.scatter(d, np.zeros\_like(d), c**=**clusters, cmap**=**c)

plt.scatter(x\_centroids, np.zeros\_like(x\_centroids), s**=**100, marker**=**'x',

c**=**range(len(x\_centroids)), cmap**=**c)

plt.show()

print("Intra-cluster cohesion (i.e. inertia):", km.inertia\_)



Intra-cluster cohesion (i.e. inertia): 20.0

Since this is a simple dataset, we can easily calculate the metric independently. For the blue cluster, the sum of the squared distances between each point and the centroid is

1

2

+

0

2

+

1

2

=2

12+02+12=2. For the red cluster, the sum is

3

2

+

0

2

+

3

2

=18

32+02+32=18. Adding up the two clusters, we have

2+18=20

2+18=20, as expected.

## **Improving the starting centroids: k-means++**

In the simplest version, the k-means algorithm will choose the initial centroids at random from the existing points in the dataset. This means that there is a chance that some of the natural clusters will start without a centroid, risking to not be correctly identified by the algorithm, while some of the centroids start very close to each other.

To alleviate this, Arthur and Vassilvitskii proposed in 2007 a method for choosing the starting centroids that aims to spread them as much as possible:

1. choose the first centroid at random from the data points;
2. until all centroids are chosen:
   1. calculate the distance between each data point and the nearest centroid;
   2. choose the next centroid from the data points with a probability proportional to that distance.

The sklearn implementation allows us to switch between the *random* initialisation and the *k-means++* initialisation using the init parameter (the same parameter that allows us to directly specify the starting centroids). Even if we cannot look into the first iteration directly (because we cannot set max\_iter=0), we can see how the centroids can end up relatively close together after the first iteration:

In [2]:

*# Find clusters*

km **=** KMeans(init**=**'random', n\_clusters**=**2, random\_state**=**4, max\_iter**=**1, n\_init**=**1, algorithm**=**'full')

clusters **=** km.fit\_predict(d)

​

*# Take the centroids*

x\_centroids **=** km.cluster\_centers\_[:,0]

​

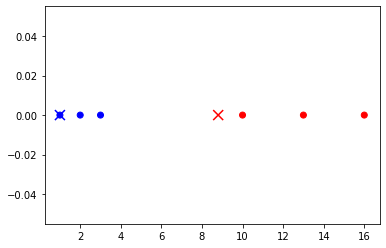
*# Scatter plot including centroids*

plt.scatter(d, np.zeros\_like(d), c**=**clusters, cmap**=**c)

plt.scatter(x\_centroids, np.zeros\_like(x\_centroids), s**=**100, marker**=**'x',

c**=**range(len(x\_centroids)), cmap**=**c)

plt.show()



The above configuration is very improbable with *k-means++*, where most of the configurations after one iteration look like this:

In [3]:

*# Find clusters*

km **=** KMeans(init**=**'k-means++', n\_clusters**=**2, random\_state**=**1, max\_iter**=**1, n\_init**=**1, algorithm**=**'full')

clusters **=** km.fit\_predict(d)

​

*# Take the centroids*

x\_centroids **=** km.cluster\_centers\_[:,0]

​

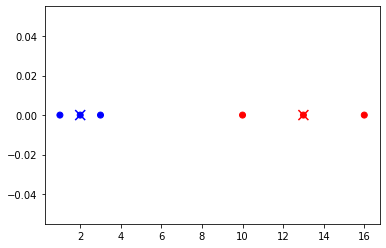
*# Scatter plot including centroids*

plt.scatter(d, np.zeros\_like(d), c**=**clusters, cmap**=**c)

plt.scatter(x\_centroids, np.zeros\_like(x\_centroids), s**=**100, marker**=**'x',

c**=**range(len(x\_centroids)), cmap**=**c)

plt.show()



## **Finding the number of clusters with the elbow method**

An important limitation of k-means is that the number of clusters must be known beforehand. Unlike with hierarchical clustering, where we could analyse the dendrogram to simply choose the best number of clusters after a single run, k-means requires multiple runs to find the best

𝑘

k.

One way to do this is to plot the least squared distance (also known as the

𝐽

J criterion, provided in the inertia\_ attribute) as a line chart for each viable value of

𝑘

k and try to find the "elbow" of the plot, that is to find the value of

𝑘

k where the cohesion stops improving significantly and reaches a plateau. Note that if the number of clusters reaches the number of instances, the least sequared distance will be 0, since each point will be identical to its centroid, this is why the minimal value of

𝐽

J is not relevant.

The elbow can be sometimes difficult to identify, if the clusters are not well defined.

For instance, if this was the plot for the value of

𝐽

J:

In [4]:

​

J **=** [55037, 29241, 10398, 6015, 2991, 2783,

2560, 2333, 2206, 2013, 1827, 1666, 1518, 1532]

k\_range**=**range(1, 1**+**len(J))

fig, ax **=** plt.subplots(figsize**=**(8, 5))

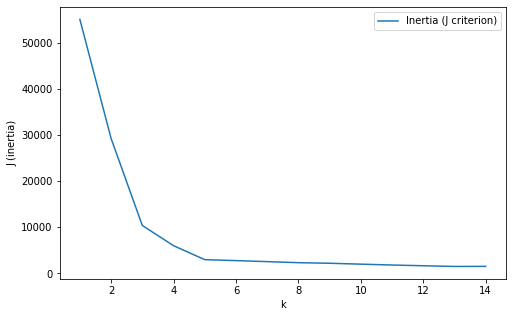
plt.plot(k\_range, J, label**=**'Inertia (J criterion)')

plt.legend()

plt.xlabel("k")

plt.ylabel("J (inertia)")

plt.show()



we can see that after

𝑘=5

k=5,

𝐽

J reaches a plateau, so in this the dataset appears to have 5 fairly distinct clusters. However, another good candidate here would be 3, where another "break" is visible in the graph, so it appears that the dataset consists of 3 larger clusters which can be roughly broken into 5 smaller ones. Indeed, if we plot the data behind this graph, we can confirm this is the case:

In [5]:

**from** sklearn.datasets **import** make\_blobs

n\_samples **=** 1500

random\_state **=** 160

X, y **=** make\_blobs(n\_samples**=**n\_samples, centers**=**5, random\_state**=**random\_state)

d **=** pd.DataFrame(X, columns**=**['X1', 'X2'])

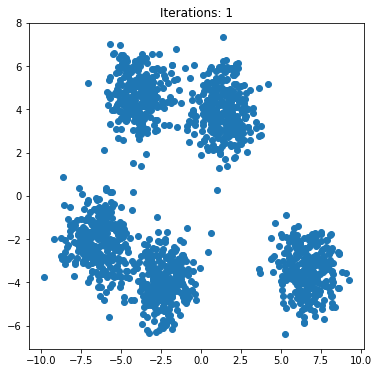
*# Plot clusters*

fig, ax **=** plt.subplots(figsize**=**(6, 6))

plt.scatter(d['X1'], d['X2'])

plt.title(f'Iterations: {km.n\_iter\_}')

plt.show()



# **Estimating the parameters of a distribution**

Consider a dataset consisting of observations from *n* independent random variables *X*1,...,*Xn*, all following the same Bernoulli distribution of an unknown parameter *p*.

We often need to estimate *p* from the observed values of *X*1,...,*Xn*. Formally, we need to find the estimation ˆ*p* of the parameter *p* which maximises the probability of the data *P*(*X*1,...,*Xn*|ˆ*p*). This can be expressed as a function of ˆ*p* called *likelihood* and defined simply as *L*(ˆ*p*|*X*1,...,*Xn*)=*P*(*X*1,...,*Xn*|ˆ*p*). The estimator that maximizes the likelihood function is called *maximum likelihood estimator* (MLE) and is denoted as ˆ*p*MLE.

Online resources:

* StatQuest on Maximul Likelihood in general - <https://youtu.be/XepXtl9YKwc>
* StatQuest on Maximum Likelihood For the Normal Distribution, step-by-step! - <https://youtu.be/Dn6b9fCIUpM>
* Derivative formulas - <https://www.eeweb.com/wp-content/uploads/calculus-derivatives-limits.pdf>
* Logarithm properites - <https://cdn.eeweb.com/tools/assets/math_help/algebra.pdf>
* Help with solving equations and plots - [https://www.wolframalpha.com](https://www.wolframalpha.com/)

## **Example**

An online shop has just updated its website and we want to know the conversion rate after the update (i.e. what is the probability that a visitor will buy before leaving) to see if the new website is performing better or worse than the old version.

The true conversion rate can be considered the unknown parameter *p* of a Bernoulli distribution, and each visitor a random variable *Xi* following that distribution.

Although *p* cannot be directly observed, we can see which of the *n* visitors generated a sale; in other words, we can observe the values of each *X*1,...,*Xn*.

This can be simulated using the scipy package for *n*=10 visitors and a true conversion rate *p*=0.7:

In [1]:

**from** scipy.stats **import** bernoulli

X **=** bernoulli.rvs(p**=**0.7, size**=**10, random\_state**=**42)

X

Out[1]:

array([1, 0, 0, 1, 1, 1, 1, 0, 1, 0])

Note that although the true conversion rate was set to 0.7, only 6 of the 10 visitors happened to buy, so our observed conversion rate is 0.6.

## **Experimental approach**

In an initial attempt to estimate the value of the original *p* from the simulated sample, we can use the fact that 0≤*p*≤1 and perform a simple search to find the candidate ˆ*p* which yields the highest likelihood *L*(ˆ*p*) given *X*=(*X*1,...,*Xn*).

The following code tests 100 values of ˆ*p* between 0 and 1 and stores in a dictionary the likelihood for each of them. It then returns the key with the maximum value from the dictionary:

In [2]:

**import** numpy **as** np

p\_hat\_range **=** np.linspace(0, 1, num**=**100)

L **=** {p\_hat: np.prod([bernoulli.pmf(x, p\_hat) **for** x **in** X])

**for** p\_hat **in** p\_hat\_range}

p\_hat\_max **=** max(L, key**=**L.get)

p\_hat\_max

Out[2]:

0.595959595959596

It seems that of the 100 the values tested, the maximum likelihood was observed for 0.595959595959596, which is very close to 0.6, our observed conversion rate.

And if we want to plot the likelihood corresponding to each ˆ*p*:

In [3]:

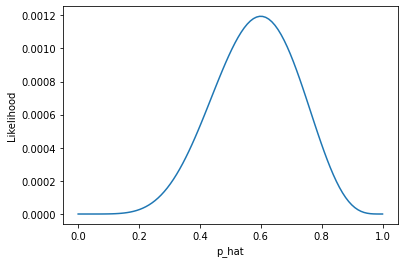
**import** matplotlib.pyplot **as** plt

fig, ax **=** plt.subplots(1, 1)

ax.plot(p\_hat\_range, list(L.values()))

ax.set(xlabel**=**'p\_hat', ylabel**=**'Likelihood')

plt.show()



Again, we can see that the graph peaks approximately at 0.6.

## **Analytical approach**

The previous approach gives a fairly good result, but has two obvious problems: it is inefficient and it is inexact.

### **Step 1 - find the likelihood function**

Instead, we can analytically find the point of maximum for the likelihood *L*(ˆ*p*). Considering the general case where the website has a true probability of conversion *p*, and we observe *n* visitors, out of which *m* have become customers:

*L*(ˆ*p*|*X*)=*P*(*X*|ˆ*p*)=*P*(*X*1,...,*Xn*|ˆ*p*)=*P*(*X*1|ˆ*p*)⋅...⋅*P*(*Xn*|ˆ*p*)=ˆ*pm*⋅(1−ˆ*p*)*n*−*m*

Finding the maximum for \hat{p}^m \cdot (1-\hat{p})^{n-m} \label{mle1} involves taking the derivative.

### **Step 2 - apply the logarithm**

However, the derivative of a multiplication is difficult to calculate when there are many factors. A common approach is to use the logarithm of this function, since it reaches maximum in the same point.

In [4]:

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**from** scipy.stats **import** bernoulli

**from** matplotlib **import** rcParams

rcParams.update({'figure.autolayout': **True**})

X **=** bernoulli.rvs(p**=**0.7, size**=**10, random\_state**=**42)

p\_hat\_range **=** np.linspace(0, 1, num**=**100)

L **=** {p\_hat: np.prod([bernoulli.pmf(x, p\_hat) **for** x **in** X])

**for** p\_hat **in** p\_hat\_range}

l **=** np.log(list(L.values())[1:**-**1])

l **=** [np.nan] **+** list(l) **+** [np.nan]

p\_hat\_max **=** max(L, key**=**L.get)

*# Plot*

fig, (ax1, ax2) **=** plt.subplots(2, sharex**=True**)

ax1.plot(p\_hat\_range, list(L.values()))

ax1.set(ylabel**=**'Likelihood')

ax1.axvline(x**=**0.6, ls**=**'--', color**=**'black')

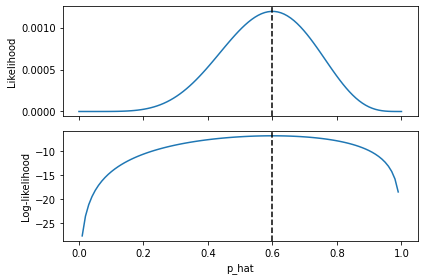
ax2.plot(p\_hat\_range, l)

ax2.set(ylabel**=**'Log-likelihood')

ax2.axvline(x**=**0.6, ls**=**'--', color**=**'black')

plt.xlabel('p\_hat')

plt.show()



This is due to the fact that the logarithm is an increasing function. The logarithm of the function is much easier to differentiate except, of course, in ˆ*p*=0 and ˆ*p*=1, where the likelihood is 0 and the logarithm is not defined. These edge cases can be treated separately as needed. For simplicity, we can call this new function *l*(ˆ*p*):

*l*(ˆ*p*)=ln(*L*(ˆ*p*|*X*))=ln(ˆ*pm*⋅(1−ˆ*p*)*n*−*m*)=*m*ln(ˆ*p*)+(*n*−*m*)ln(1−ˆ*p*)

### **Step 3 - find the maximum of the log-likelihood**

#### **Step 3a - first derivative**

The maximum of this function is reached when the first partial derivative with respect to ˆ*p* is zero. Although in this case we only have a single variable, we're using partial derivatives to illustrate the general scenario where the distribution can have multiple parameters. So we take the first derivative:

∂∂ˆ*pl*(ˆ*p*)=∂∂ˆ*p*(*m*lnˆ*p*)+∂∂ˆ*p*((*n*−*m*)ln(1−ˆ*p*))=*m*∂∂ˆ*p*(ln(ˆ*p*))+(*n*−*m*)∂∂ˆ*p*(ln(1−ˆ*p*))=*m*1ˆ*p*+(*n*−*m*)11−ˆ*p*(−1)=*m*ˆ*p*+*m*−*n*1−ˆ*p*

#### **Step 3b - second derivative**

Formally, at this point we have to check that this is indeed a point of maximum and not minimum. This is done by checking the sign of the second derivative:

∂2∂ˆ*p*2*l*(ˆ*p*)=−*m*ˆ*p*2−*n*−*m*(1−ˆ*p*)2

Since both fractions are positive, the overall result must be negative for the entire interval ˆ*p*∈(0,1), meaning that the function is concave so its extreme point is indeed a maximum, as expected.

#### **Step 3c - find ˆ*p* where first derivative is 0**

Going back to the equation for ∂∂ˆ*pl*(ˆ*p*), setting it equal to zero and solving for ˆ*p* gives:

*m*ˆ*p*+*m*−*n*1−ˆ*p*=0⇒*m*(1−ˆ*p*)=(*n*−*m*)ˆ*p*⇒ˆ*p*=*mn*

In our example with *m*=6 customers and *n*=10 visitors, the estimation is exactly *mn*=0.6, which is the same as the observed conversion. This confirms our previous rough estimations, and is also very intuitive: in the absence of any other details, the best guess for the true conversion is the conversion seen so far.

## **Estimating parameters of common distributions with scipy**

The process above is already implemented in scipy for fitting the parameters of usual distributions. The scipy.stats.bernoulli distribution is so simple that it doesn't really need it, but distributions like the normal or gamma provide the fit function which return the parameters of the distribution using maximum likelihood as detailed above.

For instance, if we want to fit a normal distribution on a dataset representing weights of mice in a lab:

In [5]:

X **=** np.array([30, 35, 40, 32, 36, 33.5, 34]) *# The weights of the mice*

we can use scipy.stats.norm:

In [6]:

**from** scipy.stats **import** norm

mean, stdev **=** norm.fit(X)

var **=** stdev**\*\***2

print(f"mean: {mean}\nvariance: {var}")

mean: 34.357142857142854

variance: 8.622448979591839

If we were to solve this using the analytic version, we would get:

*μMLE*=1*nn*∑*i*=1*xiσ*2*MLE*=1*nn*∑*i*=1(*xi*−*μMLE*)2

and if we calculate these for our dataset, we have:

In [7]:

mean1**=** X.mean()

var1 **=** X.var()

print(f"mean: {mean1}\nvariance: {var1}")

mean: 34.357142857142854

variance: 8.622448979591839

which is identical to the results of calling fit().

To visualise this, we can use matplotlib and the rvs function:

In [8]:

**from** scipy.stats **import** norm

**import** numpy **as** np

x **=** np.linspace(norm.ppf(0.01, mean, var), norm.ppf(0.99, mean, var), 100)

pdf\_X **=** norm.pdf(x, mean, var)

​

fig, ax **=** plt.subplots(1, 1)

ax.plot(x, pdf\_X)

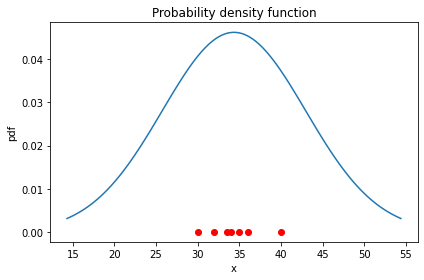
plt.ylabel("pdf")

plt.xlabel("x")

plt.title("Probability density function")

plt.scatter(X, np.zeros\_like(X), color**=**'r')

plt.show()



# **Estimating the parameters of a distribution**

Consider a dataset consisting of observations from *n* independent random variables *X*1,...,*Xn*, all following the same Bernoulli distribution of an unknown parameter *p*.

We often need to estimate *p* from the observed values of *X*1,...,*Xn*. Formally, we need to find the estimation ˆ*p* of the parameter *p* which maximises the probability of the data *P*(*X*1,...,*Xn*|ˆ*p*). This can be expressed as a function of ˆ*p* called *likelihood* and defined simply as *L*(ˆ*p*|*X*1,...,*Xn*)=*P*(*X*1,...,*Xn*|ˆ*p*). The estimator that maximizes the likelihood function is called *maximum likelihood estimator* (MLE) and is denoted as ˆ*p*MLE.

Online resources:

* StatQuest on Maximul Likelihood in general - <https://youtu.be/XepXtl9YKwc>
* StatQuest on Maximum Likelihood For the Normal Distribution, step-by-step! - <https://youtu.be/Dn6b9fCIUpM>
* Derivative formulas - <https://www.eeweb.com/wp-content/uploads/calculus-derivatives-limits.pdf>
* Logarithm properites - <https://cdn.eeweb.com/tools/assets/math_help/algebra.pdf>
* Help with solving equations and plots - [https://www.wolframalpha.com](https://www.wolframalpha.com/)

## **Example**

An online shop has just updated its website and we want to know the conversion rate after the update (i.e. what is the probability that a visitor will buy before leaving) to see if the new website is performing better or worse than the old version.

The true conversion rate can be considered the unknown parameter *p* of a Bernoulli distribution, and each visitor a random variable *Xi* following that distribution.

Although *p* cannot be directly observed, we can see which of the *n* visitors generated a sale; in other words, we can observe the values of each *X*1,...,*Xn*.

This can be simulated using the scipy package for *n*=10 visitors and a true conversion rate *p*=0.7:

In [1]:

**from** scipy.stats **import** bernoulli

X **=** bernoulli.rvs(p**=**0.7, size**=**10, random\_state**=**42)

X

Out[1]:

array([1, 0, 0, 1, 1, 1, 1, 0, 1, 0])

Note that although the true conversion rate was set to 0.7, only 6 of the 10 visitors happened to buy, so our observed conversion rate is 0.6.

## **Experimental approach**

In an initial attempt to estimate the value of the original *p* from the simulated sample, we can use the fact that 0≤*p*≤1 and perform a simple search to find the candidate ˆ*p* which yields the highest likelihood *L*(ˆ*p*) given *X*=(*X*1,...,*Xn*).

The following code tests 100 values of ˆ*p* between 0 and 1 and stores in a dictionary the likelihood for each of them. It then returns the key with the maximum value from the dictionary:

In [2]:

**import** numpy **as** np

p\_hat\_range **=** np.linspace(0, 1, num**=**100)

L **=** {p\_hat: np.prod([bernoulli.pmf(x, p\_hat) **for** x **in** X])

**for** p\_hat **in** p\_hat\_range}

p\_hat\_max **=** max(L, key**=**L.get)

p\_hat\_max

Out[2]:

0.595959595959596

It seems that of the 100 the values tested, the maximum likelihood was observed for 0.595959595959596, which is very close to 0.6, our observed conversion rate.

And if we want to plot the likelihood corresponding to each ˆ*p*:

In [3]:

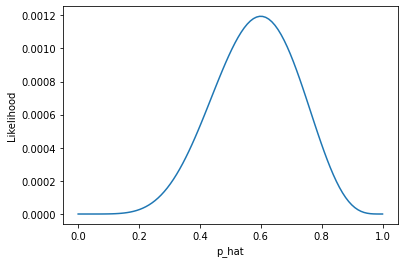
**import** matplotlib.pyplot **as** plt

fig, ax **=** plt.subplots(1, 1)

ax.plot(p\_hat\_range, list(L.values()))

ax.set(xlabel**=**'p\_hat', ylabel**=**'Likelihood')

plt.show()



Again, we can see that the graph peaks approximately at 0.6.

## **Analytical approach**

The previous approach gives a fairly good result, but has two obvious problems: it is inefficient and it is inexact.

### **Step 1 - find the likelihood function**

Instead, we can analytically find the point of maximum for the likelihood *L*(ˆ*p*). Considering the general case where the website has a true probability of conversion *p*, and we observe *n* visitors, out of which *m* have become customers:

*L*(ˆ*p*|*X*)=*P*(*X*|ˆ*p*)=*P*(*X*1,...,*Xn*|ˆ*p*)=*P*(*X*1|ˆ*p*)⋅...⋅*P*(*Xn*|ˆ*p*)=ˆ*pm*⋅(1−ˆ*p*)*n*−*m*

Finding the maximum for \hat{p}^m \cdot (1-\hat{p})^{n-m} \label{mle1} involves taking the derivative.

### **Step 2 - apply the logarithm**

However, the derivative of a multiplication is difficult to calculate when there are many factors. A common approach is to use the logarithm of this function, since it reaches maximum in the same point.

In [4]:

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**from** scipy.stats **import** bernoulli

**from** matplotlib **import** rcParams

rcParams.update({'figure.autolayout': **True**})

X **=** bernoulli.rvs(p**=**0.7, size**=**10, random\_state**=**42)

p\_hat\_range **=** np.linspace(0, 1, num**=**100)

L **=** {p\_hat: np.prod([bernoulli.pmf(x, p\_hat) **for** x **in** X])

**for** p\_hat **in** p\_hat\_range}

l **=** np.log(list(L.values())[1:**-**1])

l **=** [np.nan] **+** list(l) **+** [np.nan]

p\_hat\_max **=** max(L, key**=**L.get)

*# Plot*

fig, (ax1, ax2) **=** plt.subplots(2, sharex**=True**)

ax1.plot(p\_hat\_range, list(L.values()))

ax1.set(ylabel**=**'Likelihood')

ax1.axvline(x**=**0.6, ls**=**'--', color**=**'black')

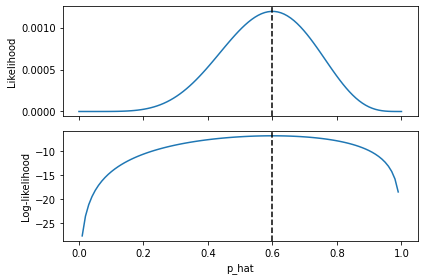
ax2.plot(p\_hat\_range, l)

ax2.set(ylabel**=**'Log-likelihood')

ax2.axvline(x**=**0.6, ls**=**'--', color**=**'black')

plt.xlabel('p\_hat')

plt.show()



This is due to the fact that the logarithm is an increasing function. The logarithm of the function is much easier to differentiate except, of course, in ˆ*p*=0 and ˆ*p*=1, where the likelihood is 0 and the logarithm is not defined. These edge cases can be treated separately as needed. For simplicity, we can call this new function *l*(ˆ*p*):

*l*(ˆ*p*)=ln(*L*(ˆ*p*|*X*))=ln(ˆ*pm*⋅(1−ˆ*p*)*n*−*m*)=*m*ln(ˆ*p*)+(*n*−*m*)ln(1−ˆ*p*)

### **Step 3 - find the maximum of the log-likelihood**

#### **Step 3a - first derivative**

The maximum of this function is reached when the first partial derivative with respect to ˆ*p* is zero. Although in this case we only have a single variable, we're using partial derivatives to illustrate the general scenario where the distribution can have multiple parameters. So we take the first derivative:

∂∂ˆ*pl*(ˆ*p*)=∂∂ˆ*p*(*m*lnˆ*p*)+∂∂ˆ*p*((*n*−*m*)ln(1−ˆ*p*))=*m*∂∂ˆ*p*(ln(ˆ*p*))+(*n*−*m*)∂∂ˆ*p*(ln(1−ˆ*p*))=*m*1ˆ*p*+(*n*−*m*)11−ˆ*p*(−1)=*m*ˆ*p*+*m*−*n*1−ˆ*p*

#### **Step 3b - second derivative**

Formally, at this point we have to check that this is indeed a point of maximum and not minimum. This is done by checking the sign of the second derivative:

∂2∂ˆ*p*2*l*(ˆ*p*)=−*m*ˆ*p*2−*n*−*m*(1−ˆ*p*)2

Since both fractions are positive, the overall result must be negative for the entire interval ˆ*p*∈(0,1), meaning that the function is concave so its extreme point is indeed a maximum, as expected.

#### **Step 3c - find ˆ*p* where first derivative is 0**

Going back to the equation for ∂∂ˆ*pl*(ˆ*p*), setting it equal to zero and solving for ˆ*p* gives:

*m*ˆ*p*+*m*−*n*1−ˆ*p*=0⇒*m*(1−ˆ*p*)=(*n*−*m*)ˆ*p*⇒ˆ*p*=*mn*

In our example with *m*=6 customers and *n*=10 visitors, the estimation is exactly *mn*=0.6, which is the same as the observed conversion. This confirms our previous rough estimations, and is also very intuitive: in the absence of any other details, the best guess for the true conversion is the conversion seen so far.

## **Estimating parameters of common distributions with scipy**

The process above is already implemented in scipy for fitting the parameters of usual distributions. The scipy.stats.bernoulli distribution is so simple that it doesn't really need it, but distributions like the normal or gamma provide the fit function which return the parameters of the distribution using maximum likelihood as detailed above.

For instance, if we want to fit a normal distribution on a dataset representing weights of mice in a lab:

In [5]:

X **=** np.array([30, 35, 40, 32, 36, 33.5, 34]) *# The weights of the mice*

we can use scipy.stats.norm:

In [6]:

**from** scipy.stats **import** norm

mean, stdev **=** norm.fit(X)

var **=** stdev**\*\***2

print(f"mean: {mean}\nvariance: {var}")

mean: 34.357142857142854

variance: 8.622448979591839

If we were to solve this using the analytic version, we would get:

*μMLE*=1*nn*∑*i*=1*xiσ*2*MLE*=1*nn*∑*i*=1(*xi*−*μMLE*)2

and if we calculate these for our dataset, we have:

In [7]:

mean1**=** X.mean()

var1 **=** X.var()

print(f"mean: {mean1}\nvariance: {var1}")

mean: 34.357142857142854

variance: 8.622448979591839

which is identical to the results of calling fit().

To visualise this, we can use matplotlib and the rvs function:

In [8]:

**from** scipy.stats **import** norm

**import** numpy **as** np

x **=** np.linspace(norm.ppf(0.01, mean, var), norm.ppf(0.99, mean, var), 100)

pdf\_X **=** norm.pdf(x, mean, var)

​

fig, ax **=** plt.subplots(1, 1)

ax.plot(x, pdf\_X)

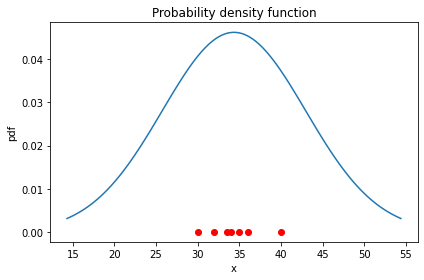
plt.ylabel("pdf")

plt.xlabel("x")

plt.title("Probability density function")

plt.scatter(X, np.zeros\_like(X), color**=**'r')

plt.show()



# **Expectation-maximisation (EM)**

Key ideas:

* assumes data is generated from
* 𝑘
* k unseen distributions
* uses MLE to estimate the parameters of those distributions

Advantages:

* fast
* can learn more complex cluster structures than k-means

Disadvantages:

* does not work well with small clusters
* sensitive to the number of dimensions - curse of dimensionality (same as for k-NN).

Online resources:

* Step-by-step description of EM (Victor Lavrenko): <https://youtu.be/-mYQ-nbykCo>

## **Motivation**

k-means is a robust algorithm, however it has trouble finding "alongated" or overlapping clusters.

In [1]:

**from** sklearn.datasets **import** make\_blobs

**import** numpy **as** np

​

n\_samples **=** 1500

random\_state **=** 170

X, y **=** make\_blobs(n\_samples**=**n\_samples, random\_state**=**random\_state)

*# Anisotropically distributed data*

transformation **=** [[0.60834549, **-**0.63667341], [**-**0.40887718, 0.85253229]]

anis **=** np.dot(X, transformation)

*# Compare these datasets*

d1, d2 **=** X, anis

In [2]:

**from** sklearn.cluster **import** KMeans

**import** matplotlib.pyplot **as** plt

**from** matplotlib.colors **import** ListedColormap

**for** d **in** [d1, d2]:

km **=** KMeans(n\_clusters**=**3)

clusters **=** km.fit\_predict(d)

centroids **=** km.cluster\_centers\_

*# Plot clusters*

fig, ax **=** plt.subplots(figsize**=**(6, 6))

c **=** ListedColormap(['red', 'green', 'blue'])

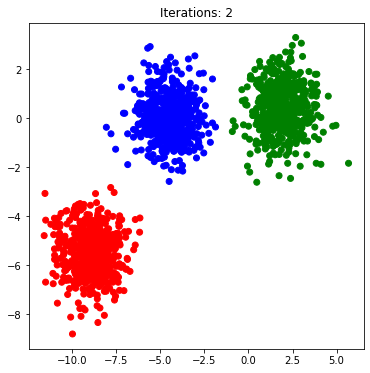
plt.scatter(centroids[:,0], centroids[:,1], s**=**100, marker**=**'x',

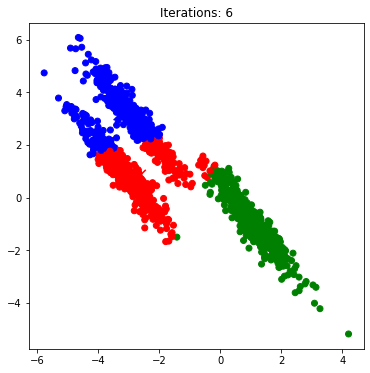
c**=**range(len(centroids)), cmap**=**c)

plt.scatter(d[:,0], d[:,1], c**=**clusters, cmap**=**c)

plt.title(f'Iterations: {km.n\_iter\_}')

plt.show()





EM for Gaussian mixture models refines this idea, by assuming that the clusters are generated from

𝑘

k normal distributions. It then repeatedly applies MLE to find the parameters of those distributions in a two-step process very similar to k-means.

It is then capabable of learning both types of shapes:

In [3]:

*# fit a Gaussian Mixture Model with two components*

**from** sklearn.mixture **import** GaussianMixture

**from** matplotlib.colors **import** LogNorm

**from** tools.plots **import** add\_ellipses

​

**for** d **in** [d1, d2]:

​

clf **=** GaussianMixture(n\_components**=**3, covariance\_type**=**'full')

clf.fit(d)

​

​

fig, ax **=** plt.subplots(figsize**=**(8, 6))

clusters **=** clf.predict(d)

​

c1 **=** ListedColormap(['red', 'green', 'blue'])

c2 **=** ['lightcoral', 'lightgreen', 'lightblue']

ax.scatter(d[:, 0], d[:, 1], c**=**clusters, cmap**=**c1)

​

add\_ellipses(clf, ax, c2)

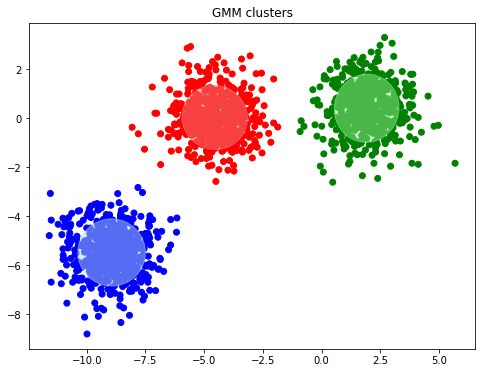
​

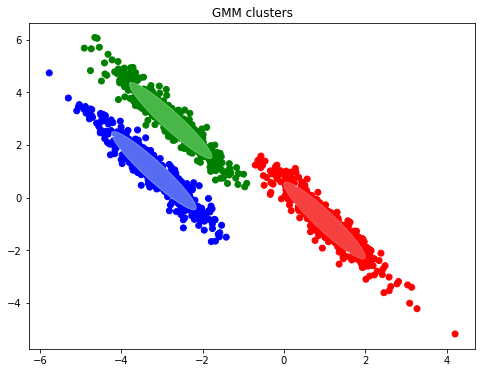
plt.title('GMM clusters')

plt.axis('tight')

plt.show()

​





## **Step-by-step example for estimating the means**

Let's take a simple dataset with six points:

In [4]:

**import** pandas **as** pd

d **=** pd.DataFrame({

'X': [10, 11, 12, 19, 20, 22]

})

fig, ax **=** plt.subplots(figsize**=**(6, 4))

plt.scatter(d['X'], np.zeros\_like(d))

plt.show()



We will assume the data points are generated by 2 distributions of variance 1.5 and unknown means

𝜇

1

μ1 and

𝜇

2

μ2. Note that the variance is typically estimated at the same time as the mean, but we're fixing it here for clarity.

Since the distributions themselves have the same probabilities

𝜋

1

=

𝜋

2

=

1

2

π1=π2=12 of being selected, we can say that that pdf of any point is:

𝑝𝑑𝑓(𝑥)=

1

2

𝑁(𝑥|

𝜇

0

,1.5)+

1

2

𝑁(𝑥|

𝜇

1

,1.5)

pdf(x)=12N(x|μ0,1.5)+12N(x|μ1,1.5)

### **Step 1 - Initial step**

Pick at random a hypothesis

ℎ

(0)

=<

𝜇

(0)

1

,

𝜇

(0)

2

>

h(0)=<μ1(0),μ2(0)>. Let's say that the two means picked are

𝜇

0

=10

μ0=10 and

𝜇

1

=11

μ1=11. We can know plot the two normal distributions:

In [5]:

**from** matplotlib.colors **import** ListedColormap

**from** scipy.stats **import** norm

​

c **=** ListedColormap(['red', 'green'])

​

means **=** [10, 11]

variances **=** [1.5, 1.5]

​

**def** plot\_pdfs(means, variances, ax, colours):

""" Plot the pdfs for given means and variances """

**for** i **in** range(0, 2):

mean, var **=** means[i], variances[i]

x **=** np.linspace(norm.ppf(0.01, mean, np.sqrt(var)), norm.ppf(0.99, mean, var), 100)

pdf **=** norm.pdf(x, mean, var)

ax.plot(x, pdf, color**=**colours(i), label**=**f"$\mu\_{i}$")

ax.legend()

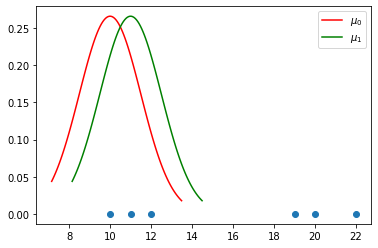
​

fig, ax **=** plt.subplots(figsize**=**(6, 4))

plt.scatter(d, np.zeros\_like(d))

plot\_pdfs(means, variances, ax, c)

plt.show()



### **Step 2 - Expectation**

Let

𝑧

𝑖𝑗

zij be a variable denoting if the point

𝑥

𝑖

xi is generated by the distribution

𝑗

j. Of course, we don't know which point was generated from which distribution, therefore these are called "hidden variables" and they need to be estimated.

Assuming that the current hypothesis

ℎ

(0)

h(0) holds, for each hidden variable

𝑧

𝑖𝑗

zij calculate the expected value

𝐸[

𝑧

𝑖𝑗

]

E[zij] which therefore represents the probability that the point

𝑥

𝑖

xi is generated by the distribution

𝑗

j (as opposed to any other distribution). Expanding this formula gives (*see the lecture for how this was calculated*):

𝐸[

𝑧

𝑖𝑗

]=𝑃(

𝑧

𝑖𝑗

=1|

𝑥

𝑖

)=

𝑝(𝑥=

𝑥

𝑖

|𝜇=

𝜇

𝑗

)

∑

2

𝑛=1

𝑝(𝑥=

𝑥

𝑖

|𝜇=

𝜇

𝑛

)

E[zij]=P(zij=1|xi)=p(x=xi|μ=μj)∑n=12p(x=xi|μ=μn)

where

𝑝(𝑥=

𝑥

𝑖

|𝜇=

𝜇

𝑗

)

p(x=xi|μ=μj) is simply the *pdf* function of the normal distribution of mean

𝜇

𝑗

μj. We can the apply the calculations:

In [17]:

p\_1 **=** norm.pdf(d, means[0], np.sqrt(variances[0]))

p\_2 **=** norm.pdf(d, means[1], np.sqrt(variances[1]))

​

E0 **=** p\_1 **/** (p\_1 **+** p\_2) *# E for j=0 (red distribution)*

E1 **=** p\_2 **/** (p\_1 **+** p\_2) *# E for j=1 (green distribution)*

print("E for j=0:\n", E0)

print("E for j=1:\n", E1)

E for j=0:

[[1.00000000e+00]

[1.00000000e+00]

[1.00000000e+00]

[9.82676909e-10]

[1.95044620e-12]

[7.68387737e-18]]

E for j=1:

[[4.86517278e-16]

[2.45117909e-13]

[1.23495695e-10]

[9.99999999e-01]

[1.00000000e+00]

[1.00000000e+00]]

And if we color each point according to the expected values, they become:

In [7]:

**from** matplotlib.colors **import** LinearSegmentedColormap

​

c\_grad **=** LinearSegmentedColormap.from\_list('mygrad', [c(0), c(1)])

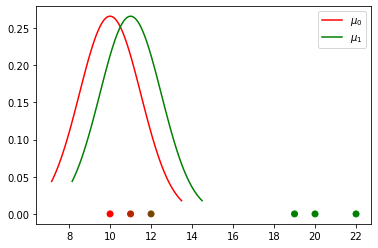
​

fig, ax **=** plt.subplots(figsize**=**(6, 4))

plt.scatter(d, np.zeros\_like(d), c**=**E1, cmap**=**c\_grad)

plot\_pdfs(means, variances, ax, c)

plt.show()



### **Step 3 - Maximisation**

Using the expected values of

𝑧

𝑖𝑗

zij we can go back and refine the two distributions, this time using MLE to find their means instead of choosing them at random. Formally, the algorithm will choose a new maximum likelihood hypothesis

ℎ

(1)

=<

𝜇

(1)

1

,

𝜇

(1)

2

>

h(1)=<μ1(1),μ2(1)>.

Using the formulas for the Gaussian models in our case, we have (*see the lecture for how this was calculated*):

𝜇

(1)

𝑗

←

∑

𝑚

𝑖=1

𝐸[

𝑧

𝑖𝑗

]

𝑥

𝑖

∑

𝑚

𝑖=1

𝐸[

𝑧

𝑖𝑗

]

μj(1)←∑i=1mE[zij]xi∑i=1mE[zij]

In [8]:

x **=** d.to\_numpy()

means[0] **=** (E0 **\*** x).sum() **/** E0.sum()

means[1] **=** (E1 **\*** x).sum() **/** E1.sum()

print("Updated means are: ", means)

Updated means are: [10.792139597966287, 16.981530285942945]

If we plot the distributions corresponding to the new means, we have:

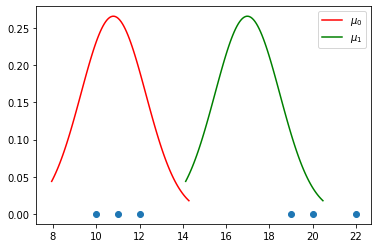
In [9]:

fig, ax **=** plt.subplots(figsize**=**(6, 4))

plt.scatter(d, np.zeros\_like(d))

plot\_pdfs(means, variances, ax, c)

plt.show()



### **Convergence**

The algorithm will keep repeating steps 2 and 3 until the means no longer change significantly.

Let's see the second iteration:

In [10]:

p\_1 **=** norm.pdf(d, means[0], np.sqrt(variances[0]))

p\_2 **=** norm.pdf(d, means[1], np.sqrt(variances[1]))

​

E0 **=** p\_1 **/** (p\_1 **+** p\_2) *# E for j=0*

E1 **=** p\_2 **/** (p\_1 **+** p\_2) *# E for j=1*

​

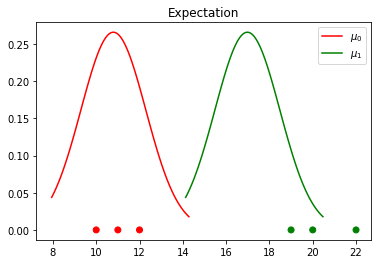
fig, ax **=** plt.subplots(figsize**=**(6, 4))

plt.scatter(d, np.zeros\_like(d), c**=**E1, cmap**=**c\_grad)

plot\_pdfs(means, variances, ax, c)

plt.title('Expectation')

plt.show()



In [11]:

means[0] **=** (E0 **\*** x).sum() **/** E0.sum()

means[1] **=** (E1 **\*** x).sum() **/** E1.sum()

​

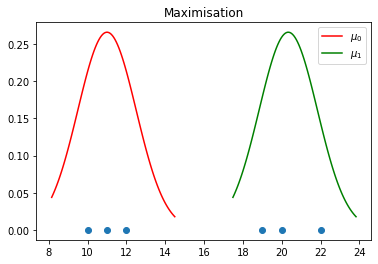
fig, ax **=** plt.subplots(figsize**=**(6, 4))

plt.scatter(d, np.zeros\_like(d))

plot\_pdfs(means, variances, ax, c)

plt.title("Maximisation")

plt.show()



And the third:

In [12]:

p\_1 **=** norm.pdf(d, means[0], np.sqrt(variances[0]))

p\_2 **=** norm.pdf(d, means[1], np.sqrt(variances[1]))

​

E0 **=** p\_1 **/** (p\_1 **+** p\_2) *# E for j=0*

E1 **=** p\_2 **/** (p\_1 **+** p\_2) *# E for j=1*

​

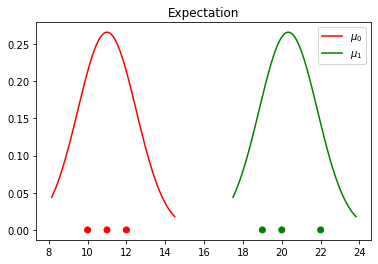
fig, ax **=** plt.subplots(figsize**=**(6, 4))

plt.scatter(d, np.zeros\_like(d), c**=**E1, cmap**=**c\_grad)

plot\_pdfs(means, variances, ax, c)

plt.title('Expectation')

plt.show()



In [13]:

means[0] **=** (E0 **\*** x).sum() **/** E0.sum()

means[1] **=** (E1 **\*** x).sum() **/** E1.sum()

​

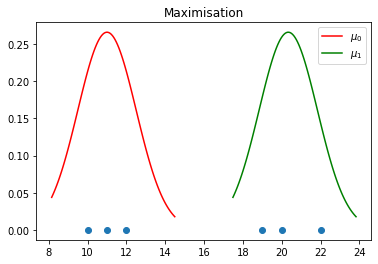
fig, ax **=** plt.subplots(figsize**=**(6, 4))

plt.scatter(d, np.zeros\_like(d))

plot\_pdfs(means, variances, ax, c)

plt.title("Maximisation")

plt.show()



We can already see that the last maximisation step did not move the means by much, so the algorithm has converged.

## **scikit-learn implementation**

The entire algorithm is already available in Python in the GaussianMixture class. It can optimise both the mean and the variance of the Gaussian distributions. Here is how the iterations look like:

In [14]:

clf **=** GaussianMixture(n\_components**=**2, covariance\_type**=**'full', max\_iter**=**12, init\_params**=**'random', random\_state**=**3, means\_init**=**[[11], [12]])

clf.fit(d)

print(clf.predict\_proba(d)[:,0])

print(clf.means\_)

print(clf.weights\_)

print(clf.covariances\_)

[1.00000000e+00 1.00000000e+00 1.00000000e+00 3.85524922e-21

6.54860073e-27 1.44406177e-39]

[[11. ]

[20.33333333]]

[0.5 0.5]

[[[0.66666767]]

[[1.55555656]]]

In [15]:

**from** sklearn.mixture **import** GaussianMixture

**import** warnings

​

**def** print\_GM\_steps():

**for** step **in** range(1, 10):

clf **=** GaussianMixture(n\_components**=**2, covariance\_type**=**'full', \

max\_iter**=**step, init\_params**=**'random', \

random\_state**=**3, means\_init**=**[[11], [12]])

clusters **=** clf.fit(d)

cl\_proba **=** clf.predict\_proba(d)[:,1]

fig, ax **=** plt.subplots(figsize**=**(6, 4))

plt.scatter(d, np.zeros\_like(d), c**=**cl\_proba, cmap**=**c\_grad)

means **=** clf.means\_[:,0]

variances **=** clf.covariances\_[:,0][:,0]

plot\_pdfs(means, variances, ax, c)

plt.title(f"Iteration {step}")

plt.show()

**with** warnings.catch\_warnings():

warnings.simplefilter("ignore")

print\_GM\_steps()

