1 SOLUTIONS: SEMTM0016 Week 17 Worksheet Unsupervised Learning: Guassian Mixture Models and PCA

2 Introduction

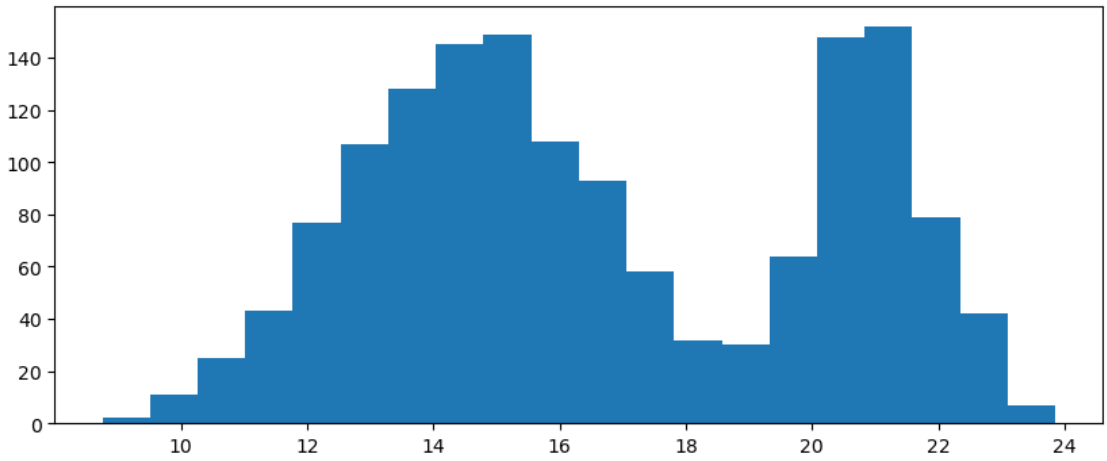
This week we will finish looking at unsupervised clustering algorithms with Gaussian mixture models and will compare the behaviour with that of k-means from last week. We will then move on to look at PCA.

3 Gaussian mixture models

In the lecture we saw that my local pub has visiting times on a Sunday that look like hey can be well modelled by a mixture of Gaussians. Let's look into how we model that.

First we're going to mimic the data we saw - where we had one peak around 2.30pm and another around 9pm. (We'll do this by sampling from some gaussian distributions and then let's see if can learn our original distributions.)

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

from scipy.stats import norm
from numpy.random import default_rng

rng=default_rng(100)

# x1 sampled from a normal distribution with mean 14.5 and std dev 2
x1 = rng.normal(loc=14.5, scale=2, size=1000)

# x2 sampled from a normal distribution with mean 21 and std dev 1
x2 = rng.normal(loc=21, scale=1, size=500)

# Plot the data as a histogram
X=np.hstack([x1, x2])
fig, ax = plt.subplots(figsize=(10, 4))
n, bins, patch = plt.hist(X, bins=20)

# Keep a record of the scaling for later plotting
bin_length=bins[1]-bins[0]
sf1=bin_length*len(x1)
sf2=bin_length*len(x2)
```

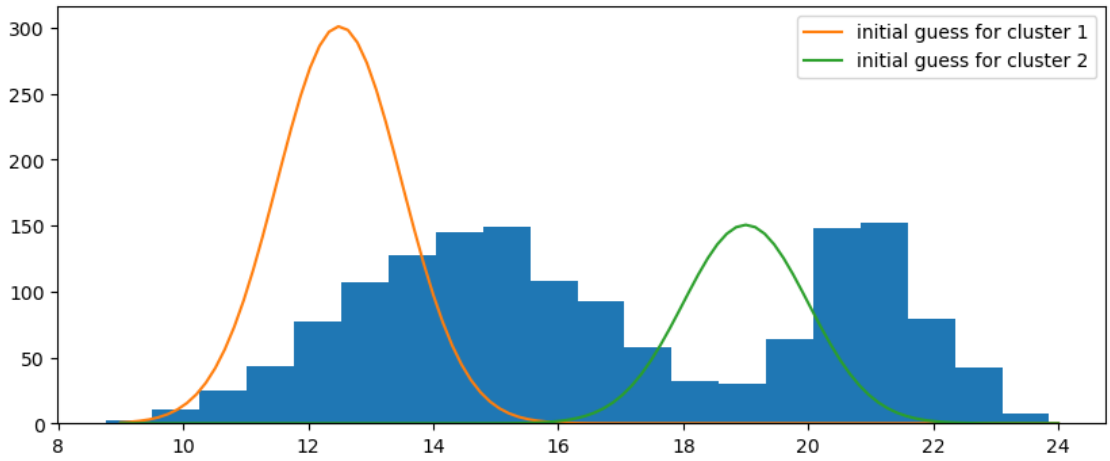


We first make a random guess at the means and the standard deviation. Let's guess that the mean of one cluster is 12.5 (i.e. 12.30pm) and that the mean of the other cluster is 19 (i.e. 7 pm), and that the standard deviation of each is 1. Let's also say that each cluster has weight 0.5.

You can use `norm.pdf` from SciPy to calculate the probability density function.

[2]:
```
# Means
mu1=12.5
mu2=19

# Std devs
sigma1=1
sigma2=1

# Weights
w1=0.5
w2=0.5

xs=np.linspace(9, 24, 100)
ax.plot(xs, norm.pdf(xs, loc=mu1, scale=sigma1)*sf1)
ax.plot(xs, norm.pdf(xs, loc=mu2, scale=sigma2)*sf2)
labels = ['initial guess for cluster 1', 'initial guess for cluster 2']
ax.legend(labels);
fig
```

[2]:



Let's consider the point midday, or 12. We can compute the responsibility of 12 in each cluster as follows:

The responsibility of x in cluster j is

$$r_j(x) = \frac{w_j f_j(x)}{\sum_{i=1}^{k} w_i f_i(x)}$$

where $f_i(x)$ is the pdf of the normal distribution with mean $= \mu_i$ and standard deviation $= \sigma_i$

[3]:
```
def calc_responsibilities(x, means, stds, ws):
    # Calculate the weighted pdf for each of our clusters
    weighted_pdf_1=ws[0]*norm.pdf(x, loc=means[0], scale=stds[0])
    weighted_pdf_2=ws[1]*norm.pdf(x, loc=means[1], scale=stds[1])

    # Calculate the responsibilty for each of our clusters
    responsibilty_1 = weighted_pdf_1/(weighted_pdf_1+weighted_pdf_2)
    responsibilty_2 = weighted_pdf_2/(weighted_pdf_1+weighted_pdf_2)

    return(np.vstack([responsibilty_1, responsibilty_2]))
```

[4]:
```
calc_responsibilities(12, [mu1, mu2], [sigma1, sigma2], [w1, w2])
```

[4]:
```
array([[1.0000000e+00],
       [2.5946095e-11]])
```

Now we can calculate the responsibilities for all our datapoints:

[5]:
```
# Calculate the responsibilities for all the datapoints
rs = calc_responsibilities(X, [mu1, mu2], [sigma1, sigma2], [w1, w2])
```

Based on these responsibilities, we will now calculate the new means. The new means will be a weighted sum of the points in the dataset, where the weighting is based on the responsibilities we have just calculated.

$$\mu_j' = \frac{\sum_x r_j(x)x}{\sum_x r_j(x)}$$

Compare with k-means: in k-means, we considered the responsibility or membership of a cluster as just 1 or 0 (it either is or isn't part of the cluster), and we calculate the new means by taking the average of all the points that have responsibility 1 in a cluster.

[6]:
```
def calc_means(x, responsibilities):
    # Calculate the means
    denoms = np.sum(responsibilities, axis=1)
    numtrs = np.sum(responsibilities*x, axis=1)
    return numtrs/denoms
```

[7]:
```
new_mus= calc_means(X, rs)
print(new_mus)
```

```
[13.63243613 19.56425555]
```

We now calculate the new standard deviations based on the responsibilities and the new means.

$$\sigma_j'^2 = \frac{\sum_x r_j(x)(x-\mu_j')^2}{\sum_x r_j(x)}$$

[8]:
```
def calc_stds(x, means, responsibilities):
    # Calculate the std devs
    denoms = np.sum(responsibilities, axis=1)
    numtrs = np.sum(responsibilities*((x - means.reshape(-1,1))**2), axis=1)
    return np.sqrt(numtrs/denoms)
```

[9]:
```
new_sigmas = calc_stds(X, new_mus, rs)
print(new_sigmas)
```

```
[1.42260955 2.16392885]
```

Finally, we calculate the new weights for each cluster

$$w_j' = \frac{\sum_x r_j(x)}{N}$$

where N is the total number of points (in our case 1500)

[10]:
```
def calc_weights(responsibilities):
    # Calc weights
    return np.sum(responsibilities, axis=1)/responsibilities.shape[1]
```

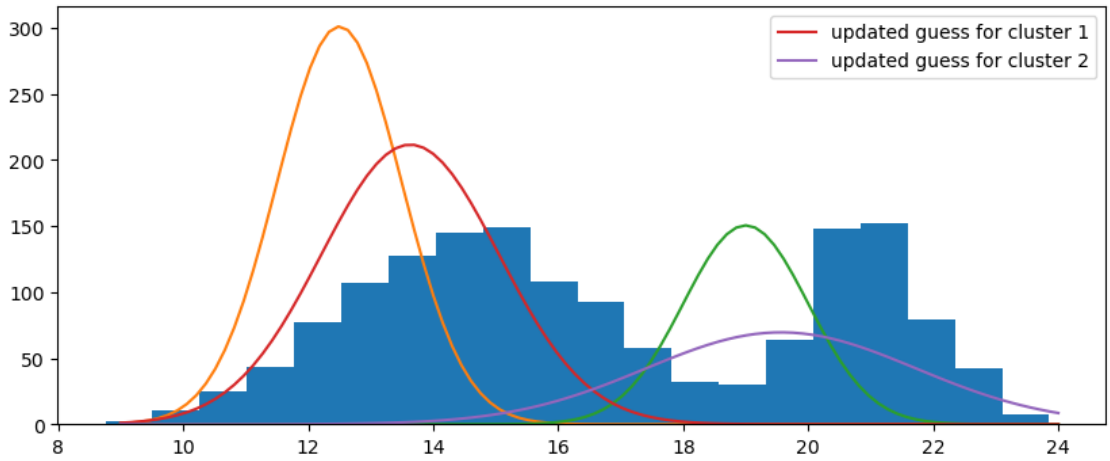
[11]:
```
new_weights = calc_weights(rs)
print(new_weights)
```

```
[0.4777986 0.5222014]
```

Now let's plot our updated curves:

[12]:
```
ax.plot(xs, norm.pdf(xs, loc=new_mus[0], scale=new_sigmas[0])*sf1,␣
 ↪label='updated guess for cluster 1')
ax.plot(xs, norm.pdf(xs, loc=new_mus[1], scale=new_sigmas[1])*sf2,␣
 ↪label='updated guess for cluster 2')
ax.legend()
fig
```

[12]:



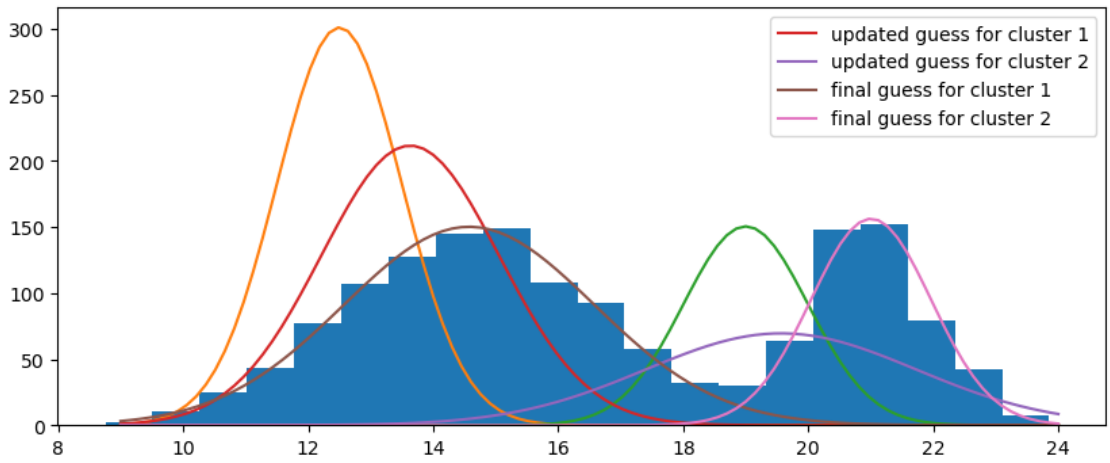
Now, we want to continue running all these steps until we see no more changes

[13]:
```
mus=np.array([mu1, mu2])
sigmas=np.array([sigma1, sigma2])
ws=np.array([w1, w2])
convergence = False

while not convergence:
    rs = calc_responsibilities(X, mus, sigmas, ws)
    new_mus = calc_means(X, rs)
    new_sigmas = calc_stds(X, new_mus, rs)
    new_ws = calc_weights(rs)
    convergence = np.allclose(mus, new_mus) and np.allclose(sigmas, new_sigmas)␣
 ↪and np.allclose(ws, new_ws)
    mus = new_mus
    sigmas = new_sigmas
    ws = new_ws
```

[14]:
```
ax.plot(xs, norm.pdf(xs, loc=mus[0], scale=sigmas[0])*sf1, label='final guess␣
 ↪for cluster 1')
ax.plot(xs, norm.pdf(xs, loc=mus[1], scale=sigmas[1])*sf2, label='final guess␣
 ↪for cluster 2')
```

```
ax.legend()
fig
```

[14]:



And as we can see we're pretty close to our original distributions:

[15]:
```
print("Final means", mus)
print("Final sigmas", sigmas)
print("Final weights", ws)
```

```
Final means [14.57560302 21.00478039]
Final sigmas [2.00430525 0.96305298]
Final weights [0.66489686 0.33510314]
```

3.1 Using Gaussian mixture models from Scikit-learn

3.1.1 Clustering on the iris dataset... again...

In this question we investigate the use of Gaussian clustering on the Iris data set and compare it to k-means.

[2]:
```
# Import the iris dataset, and save the data into a variable X (take a look at␣
 ↪the documentation here:
# https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.
 ↪html)

from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
```

Scikit-learn has GMMs built in. We import it using the command `from sklearn.mixture import GaussianMixture as GMM`. Look at the documentation for Gaussian Mixtures.

Again let's begin by assuming that since there are 3 types of iris, then there may be 3 clusters. Instantiate a GMM classifier with 3 clusters, and fit it to the data. Print out the parameters

(means and covariances) for the learnt Gaussian Distributions. You can visualise the resulting clusters by generating scatter plots projected on 2 dimensions. Try generating scatter plots for various combinations of features.

Extra question Generate one large plot with subplots for each combination of features.

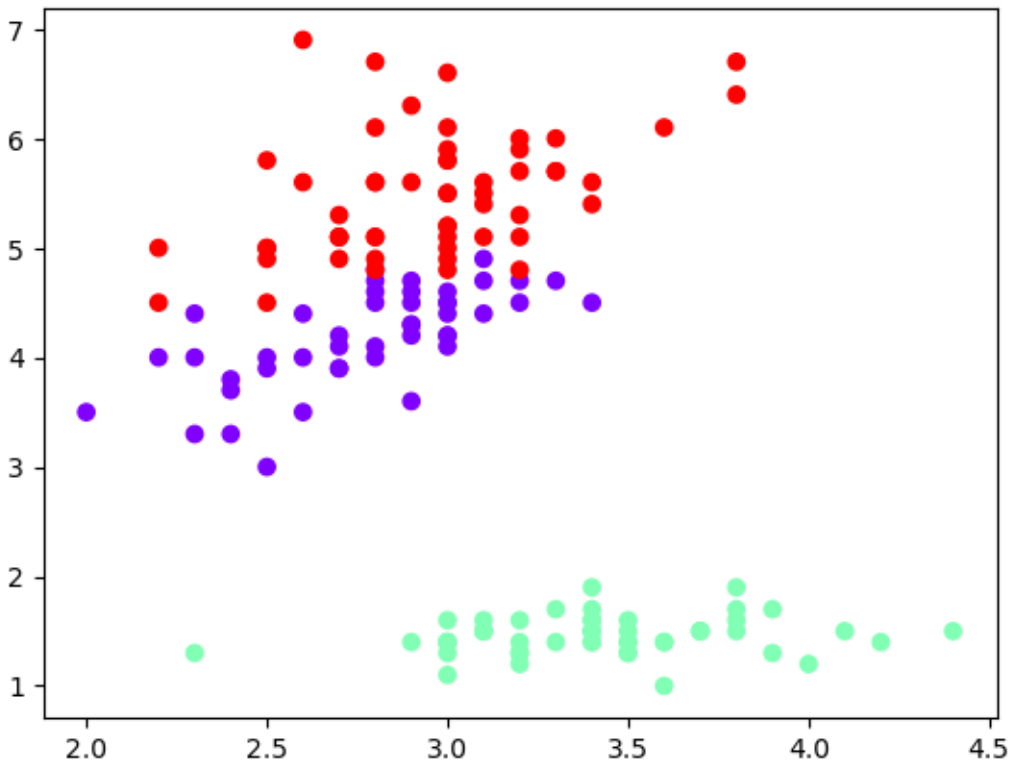
[3]:
```
from sklearn.mixture import GaussianMixture as GMM

# Fit the iris dataset using GMM
gmm = GMM(n_components=3);
gmm.fit(X);
```

A: We can extract the parameters for the learnt Gaussian distributions as follows:

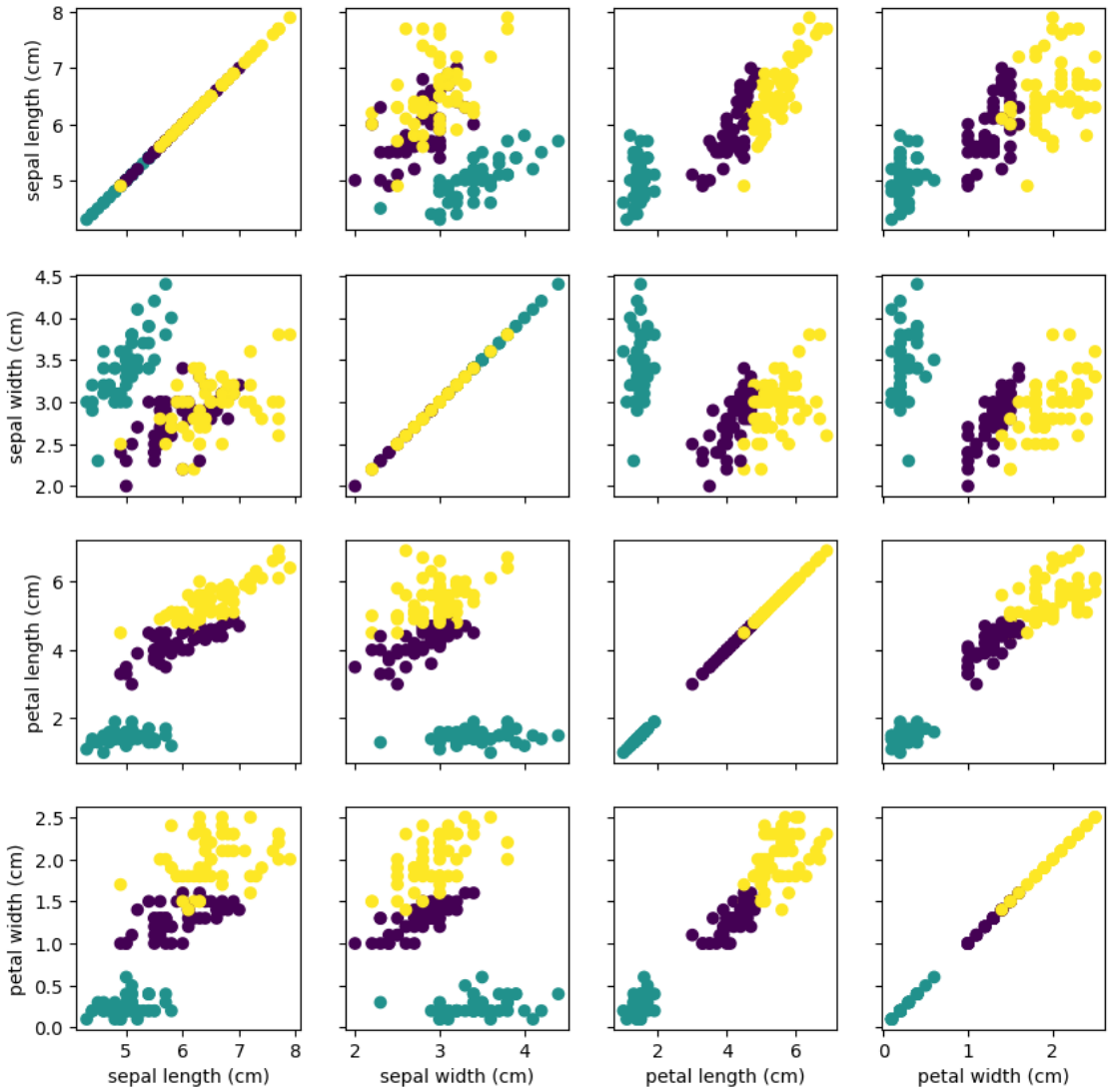
[4]:
```
# print out GMM means and covariances

print(gmm.means_)
print(gmm.covariances_)
```

```
[[5.91697517 2.77803998 4.20523542 1.29841561]
 [5.006      3.428      1.462      0.246     ]
 [6.54632887 2.94943079 5.4834877  1.98716063]]
[[[0.27550587 0.09663458 0.18542939 0.05476915]
  [0.09663458 0.09255531 0.09103836 0.04299877]
  [0.18542939 0.09103836 0.20227635 0.0616792 ]
  [0.05476915 0.04299877 0.0616792  0.03232217]]

 [[0.121765   0.097232   0.016028   0.010124  ]
  [0.097232   0.140817   0.011464   0.009112  ]
  [0.016028   0.011464   0.029557   0.005948  ]
  [0.010124   0.009112   0.005948   0.010885  ]]

 [[0.38741443 0.09223101 0.30244612 0.06089936]
  [0.09223101 0.11040631 0.08386768 0.0557538 ]
  [0.30244612 0.08386768 0.32595958 0.07283247]
  [0.06089936 0.0557538  0.07283247 0.08488025]]]
```

[5]:
```
# Make a scatter plot of the data on the first two axes
# Experiment with looking at different axes

plt.figure();
plt.scatter(X[:,1],X[:,2], c=gmm.predict(X), cmap='rainbow');
```



[6]:
```
# Plot all combinations of data in one figure
fig, axs = plt.subplots(4,4, figsize=(10,10), sharey='row', sharex='col')
for i in range(4):
    for j in range(4):
        axs[j,i].scatter(X[:,i],X[:,j], c=gmm.predict(X))
        if j == 3:
            axs[j,i].set_xlabel(iris.feature_names[i])
        if i == 0:
            axs[j,i].set_ylabel(iris.feature_names[j])
```

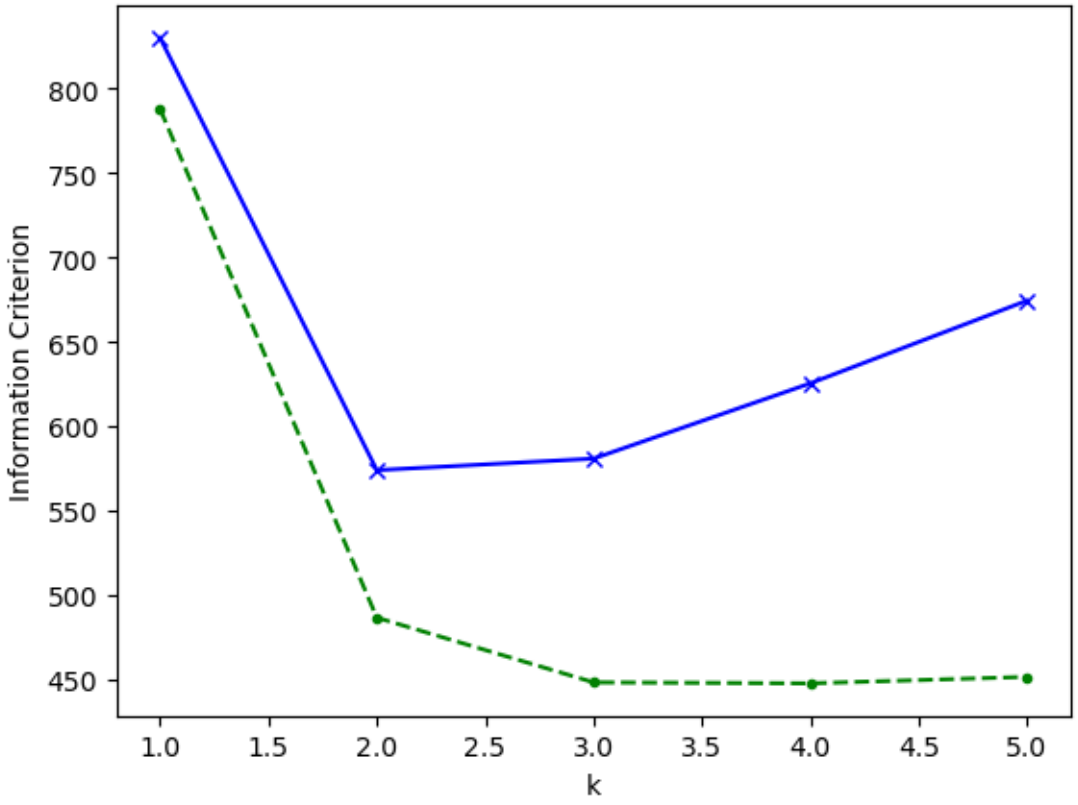


Now how to select the number of clusters?

We can no longer use the inertia or the silhouette score as we did for k-means as they both assume spherical equally sized clusters. For Gaussian mixtures we instead try to find the model that minimises a *theoretical information criterion*. The log-likelihood of the data is what we're trying to maximise in EM, i.e. how plausible are the parameter values of the model given our data. (You can access the per-sample average log-likelihoods from the `score` method)

Information criterion are then likelihood-based measures of model fit that include some penalty for complexity e.g. the number of clusters. Handily, the Scikit-learn's `GaussianMixture` has two built-in information criterion methods: `aic` and `bic`. They can often end up selecting the same model, but if they do differ it tends to be that BIC has selected a simpler model that doesn't fit the data quite as well.

[21]:
```
# Generate a plot showing the aic and bic scores
bics = []
aics = []
K = 5
for k in range(1, K+1):
    gmm = GMM(n_components=k)
    gmm.fit(X)

    bics.append(gmm.bic(X))
    aics.append(gmm.aic(X))

# Plot the scores
plt.figure();
plt.plot(range(1, K+1), bics, 'bx-');
plt.plot(range(1, K+1), aics, 'g.--');
plt.xlabel('k');
plt.ylabel('Information Criterion');
```

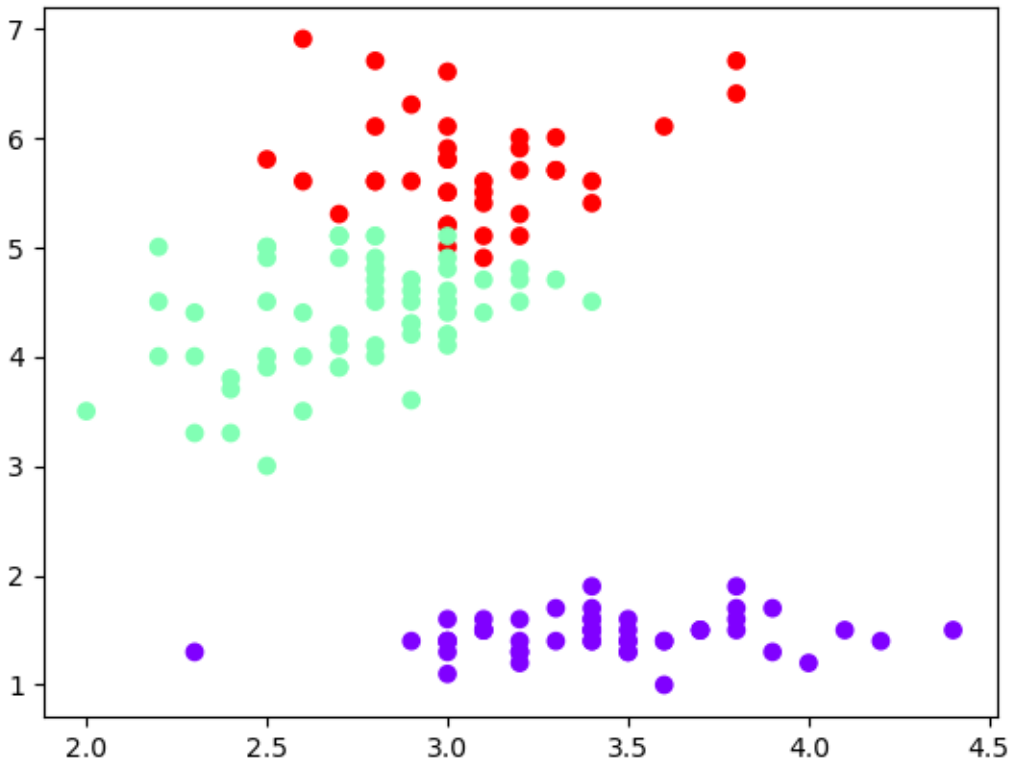


Q: How do the means for the three distributions compare with the centroids from a 3-cluster k-means on this dataset?

A: One of the means is identical: that of the more isolated cluster. The others are close.

[7]:
```
from sklearn.cluster import KMeans

# Fit the iris dataset using k-means
kmm = KMeans(n_clusters = 3)
kmm.fit(X)
print(kmm.cluster_centers_)

# Make a scatter plot of the data on the first two axes
# Experiment with looking at different axes
plt.figure();
plt.scatter(X[:,1],X[:,2], c=kmm.labels_, cmap='rainbow');
```

```
[[5.006      3.428      1.462      0.246     ]
 [5.9016129  2.7483871  4.39354839 1.43387097]
 [6.85       3.07368421 5.74210526 2.07105263]]
```



Use the command `print(gmm.weights_)` to look at the weights for each distribution for our GMM with 3 clusters.

Q: What do these weights tell us about the composition of the three clusters?

A: The weight of the isolated cluster is 1/3, indicating that this cluster accounts for 1/3 of the points. The other clusters each account for just under and just over 1/3 of the points.

[8]:
```
gmm = GMM(n_components=3)
gmm.fit(X)

print(gmm.weights_)
```

```
[0.36539575 0.33333333 0.30127092]
```

3.2 Using PCA from Scikit-learn

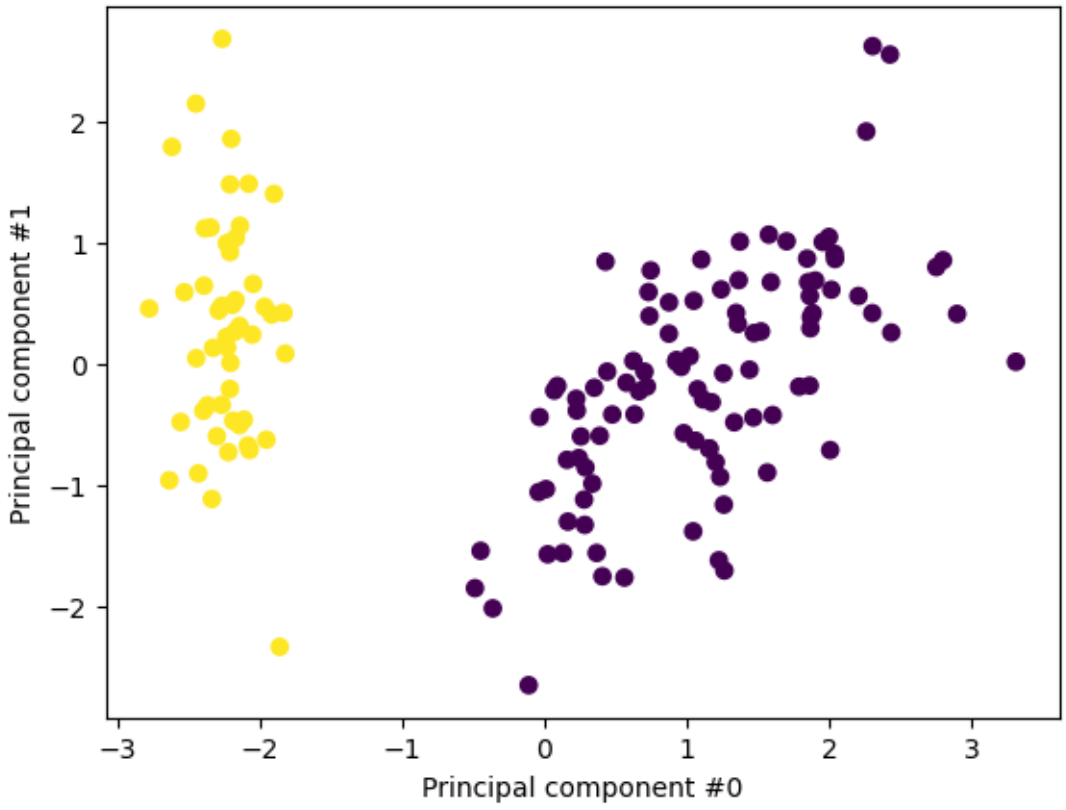
A common approach to visualising data after we have clustered it, is to use a dimensionality reduction method such as PCA. You can find the documentation for Scikit-learn's built in PCA here: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

Apply PCA to the iris dataset, reducing the data to two dimensions. Create a two scatter plots of these two dimensions and use the cluster assignments from your trained GMM to colour the datapoints. Print out the explained variance of the components.

To standardise the data first we can use Scikit-learn's `StandardScaler`.

Q: What does `StandardScaler` do? A: Standardises the dataset by removing the mean and scaling to unit variance.

[24]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Standardise the deatures
X_scaled = StandardScaler().fit_transform(X)

# Find principle components using PCA
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X_scaled)

# Fit the iris dataset using GMM
gmm = GMM(n_components=2)
gmm.fit(X)

fig, ax = plt.subplots()

ax.scatter(X_reduced[:, 0], X_reduced[:, 1], c=gmm.predict(X))
ax.set_ylabel("Principal component #1");
ax.set_xlabel("Principal component #0");
```



[25]:
```
# print explained variance
print(pca.components_)
```

```
[[ 0.52106591 -0.26934744  0.5804131   0.56485654]
 [ 0.37741762  0.92329566  0.02449161  0.06694199]]
```

[26]:
```
# print explained variance
print(pca.explained_variance_)
```

```
[2.93808505 0.9201649 ]
```

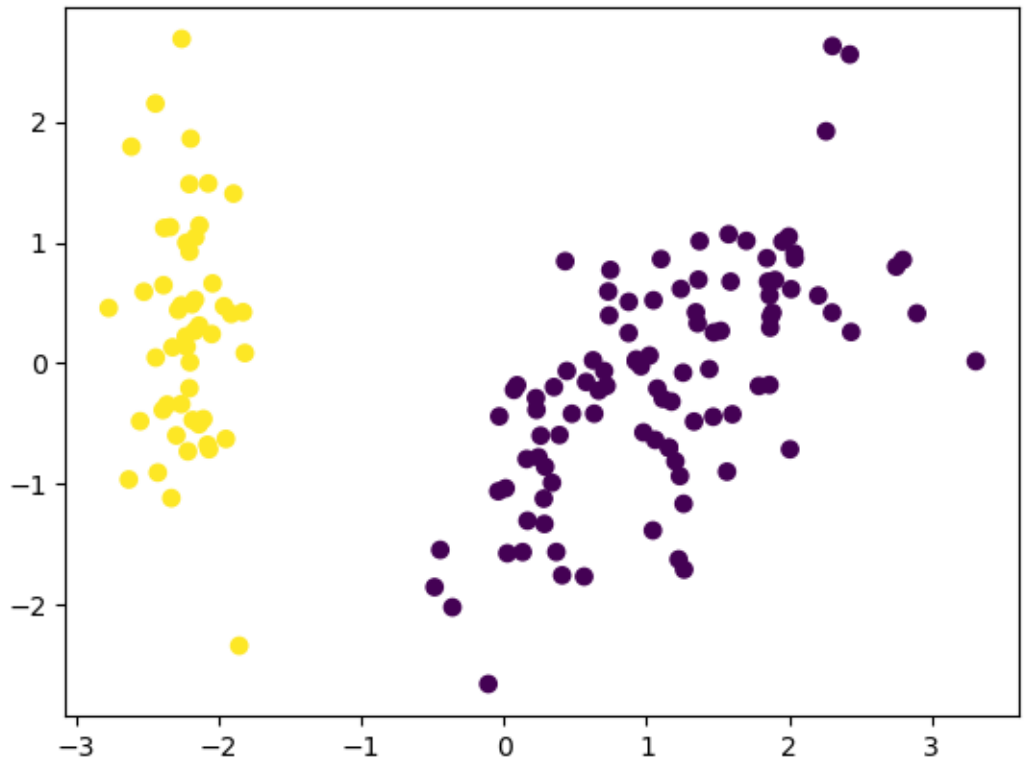
3.2.1 (Optional) Now try implementing PCA on the iris dataset yourself.

Reproduce the same plot as above and check you calculate the same components and explained variance.

Use `np.cov` and `np.linalg.eig` to calculate the covariance matrix and eigenvalue/eigenvector pairs.

[31]:
```
## TODO

# Compute the covariance matrix
COV = np.cov(X_scaled.T)
```

```
# Compute the eigenvalue/eigenvector pairs
evals, evecs = np.linalg.eig(COV)

# Find the index of the two largest eigenvalues
i, j = np.argsort(-evals)[:2]

# Set the eigenvalue, eigenvector pairs with largest eigenvalues
c1, c2 = evecs[:, i], evecs[:, j]
l1, l2 = evals[i], evals[j]

# Project the data
V = np.dot(X_scaled, np.vstack((c1,c2)).T)

fig, ax = plt.subplots()
ax.scatter(V[:, 0], (-1)*V[:, 1], c=gmm.predict(X));
```



[32]:
```
# print components
print(c1)
print(c2)
```

```
[ 0.52106591 -0.26934744  0.5804131   0.56485654]
```

```
[-0.37741762 -0.92329566 -0.02449161 -0.06694199]
```

[33]:
```
# print the explained variance for the first two components
print(l1)
print(l2)
```

```
2.9380850501999967
0.920164904162488
```

Q: What is the total percentage of explained variance with two components?

[34]:
```
(l1+l2) / evals.sum()
```

[34]:
```
0.9581320720000164
```

Finally, PCA is useful but not without its limitations - see this nice blogpost here: https://ekamperi.github.io/mathematics/2021/02/23/pca-limitations.html