Chapter 11 - Gaussian Mixture Models

Solution Workbook for Student Practical Classes

(c) Dr Rob Collins 2023

2023-05-08

Table of contents

List of Figures

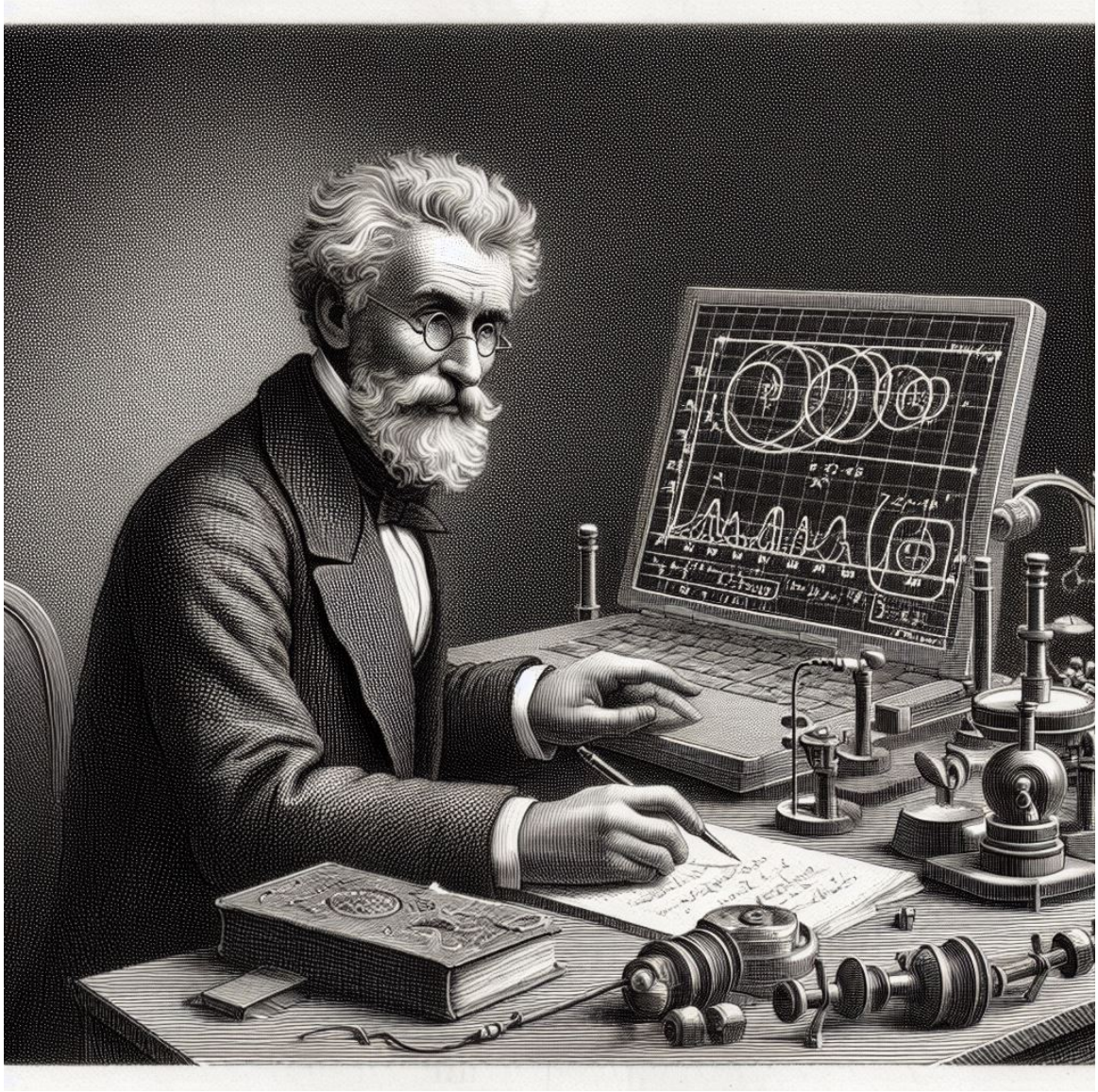

Figure 1: Johann Carl Friedrich Gauss - the so called ‘Prince of Mathematicians’ engrossed in the third iteration of the now famous ‘mixture model’

11 Gaussian Mixture Models

11.1 Introduction

In this workshop we will be experimenting with fitting multiple, over-lapping Gaussian Models to data-sets.

Gaussian Mixture Models have been used to model scientific data in diverse domains including medicine, biology and astro-physics. They are useful when dealing with data that may include multiple populations - but we don't know the characteristics of those populations.

Thus, we may have a data-set of what we believe is a single population - for example a collection of biological specimens that appear superficially to be the same. However, when we look more closely at the data it may become apparent that a better model is achieved by assuming that we actually have multiple populations, each with their own Gaussian Distribution. That is, each population has a set of variables that are characterised by a mean (average) a standard deviation and follow a characteristically 'normal' (bell-shaped) population.

The task is to recreate the most likely set of individual, overlapping Gaussian distributions from a mixed population.

Note : Although there are many cells in this workshop activity you will see that many of them repeat. These include all of the cells that display graphs of the data we use. Thus you should find it easy to copy-paste section of earlier code into later sections.

11.2 Instructions for Students

In this workbook there are regular 'callout' blocks indicating where you should add your own code. They look like this:

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided 'clues' towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class

8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

11.3 Importing required libraries

As always we start by importing any required libraries. In this case, we will be using Numpy to generate some data-sets. Those data-sets will contain a mixture of multiple populations that each have a Gaussian Distribution. Initially we will be building our Gaussian Mixture Models (GMM) 'by hand', but later we will be using a algorithm that is part of the sklearn library to complete the task quickly and efficiently. We also use a KDE plot to display the data.

For this practical session we will need the following libraries:

- 'pandas' : Conventionally given the reference name 'pd'
- 'preprocessing' from sklearn
- 'metrics' from sklearn
- 'numpy' : Conventionally given the reference name 'np'
- 'norm' : from scipy.stats
- 'GaussianMixture : 'from sklearn.mixture

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm
from sklearn.mixture import GaussianMixture
```

11.4 Manual fitting of a Gaussian Mixed Model (GMM)

In this section we are going to attempt to fit a mixed Gaussian model 'by hand'. I have created a data-set and it will be your task to create a simple GMM for the data.

11.4.1 Loading the source data

Add a cell to load the data from the csv file "Mixed population Data.csv" in a dataframe called 'mixed_population'. Display the first 10 rows of the dataframe and a description of the data.

```
mixed_population = pd.read_csv("Mixed population Data.csv")
print(mixed_population.describe())
mixed_population[0:10]
```

```
          The_Data
count   982.000000
mean     80.265438
std      34.307295
min      -5.150000
25%      49.980000
50%      84.825000
75%     110.805000
max     147.660000
```

	The_Data
0	113.56
1	61.05
2	37.02
3	108.99
4	110.36
5	20.46
6	50.60
7	110.84
8	119.30
9	126.40

As you can see, this is a single column of data consisting of 981 data points. Such data might be, for example, a set measurements of the size of Turtle shells measured by a biologist. The question is .. how best to model this data?

Let's first take a look at the data and see if that can provide some clues.

Add a cell to plot a kde plot for the data.

```
plt.figure(figsize=(10, 6))
sns.kdeplot(mixed_population['The_Data'],
            label='mixed population',
            shade=True,
            color="orange")
plt.title(f"KDE Plot for 'mixed_population' data-set")
plt.legend()
plt.show()
```

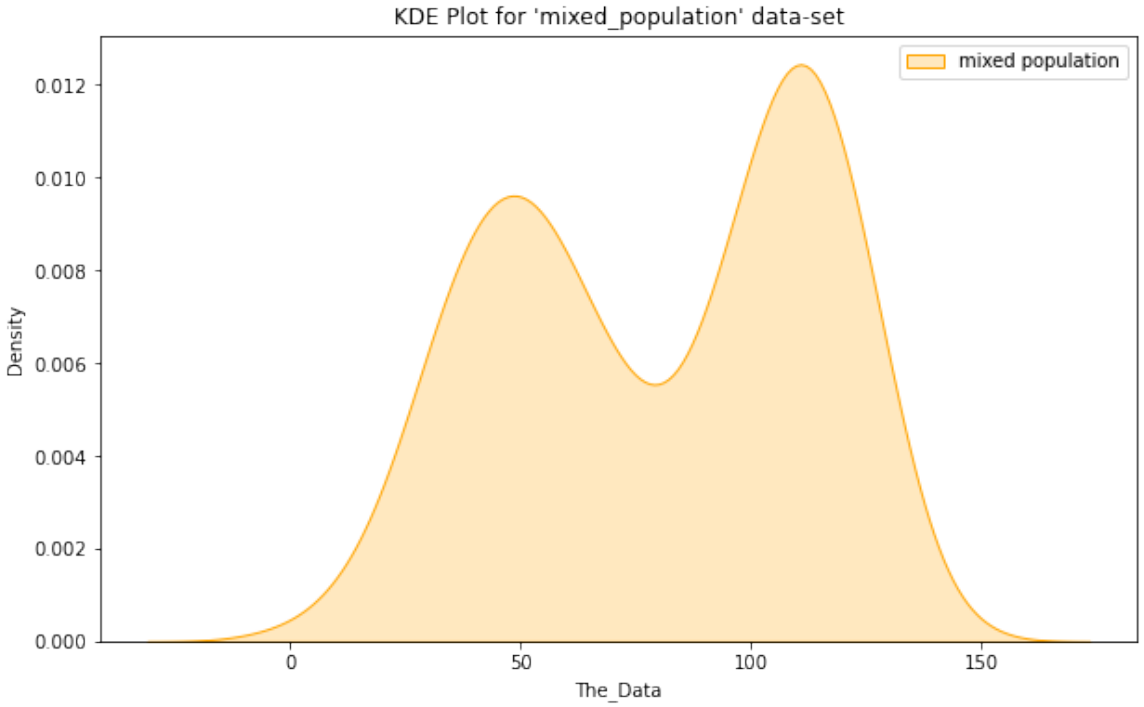

Figure 2: kde plot of the mixed population data

We could have equally well plotted a more familiar histogram at this point. However, has been stated in previous practicals an issue is that they can provide a misleading visualisation. In the following cell I have illustrated this with a rather extreme example - plotting the data with only 5 buckets. But 'too few buckets' is only one of the artifacts that histograms are prone to; issues may also occur when there are too many buckets and sometimes when the bucket-boundaries coincide with specific peaks in data etc.

On the other hand, histograms are more familiar to a broader audience. If you employ kde plots in your analysis for a less expert audience you may have to explain and justify your choice of chart. You should make a critical judgment regarding your use of data display in each case.

```
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=[10, 6])
ax.hist(mixed_population, bins=5, density=True, alpha=0.5, color="#0070FF")

plt.title(f"Histogram for 'mixed_population' data-set")
ax.set_ylabel("Number of Turtles")
ax.set_xlabel("Turtle shell width")

# Draw legend
plt.show()
```

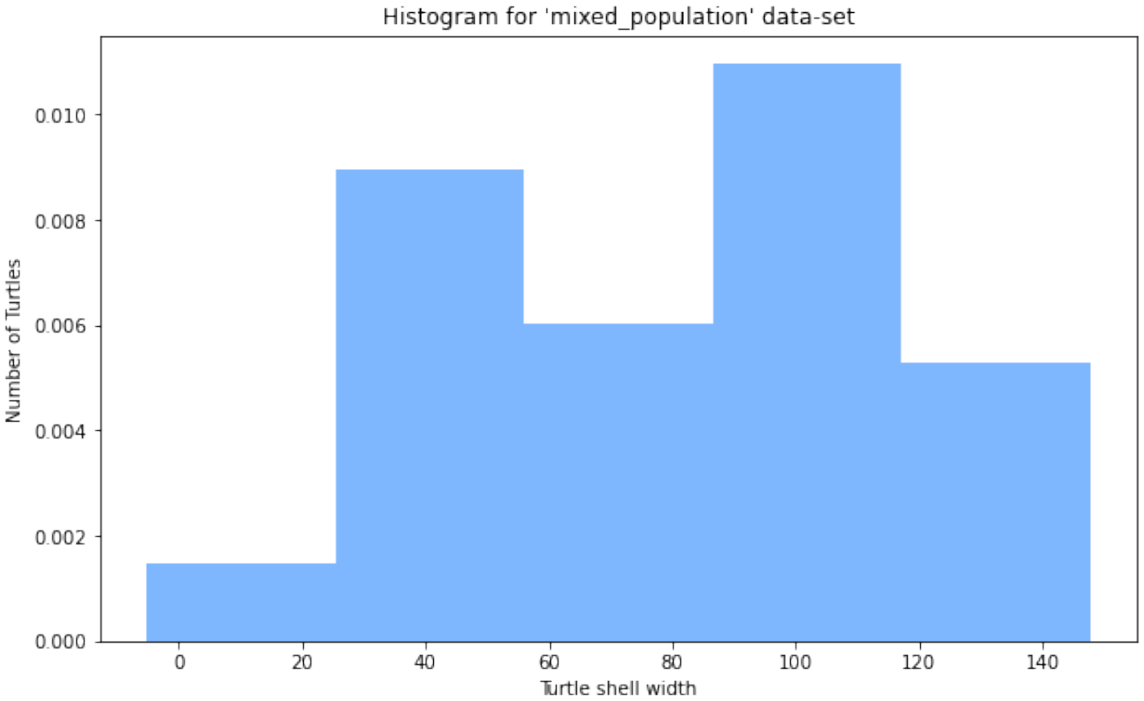

Figure 3: Histogram of the mixed population data illustrating the impact of a poor choice of bucket count

The overall shape of the combined data does not appear to be Gaussian. This data appears to be 'bi-modal' : That is, there are two peaks. Each of these peaks may be Gaussian - but the data-set as a whole is not. This may suggest to us that we are looking at two (or more) separate populations. We may have collected data for two different species of turtle!

So the idea here is to generate multiple Gaussian Distributions (in this case 2) - and fit those to the peaks and troughs in the above distribution.

11.4.2 Generating data to experiment with fitting

In this section we are going to generate our own data-set using Python. Then later we will experiment with this data set to try to make it match the data in the 'Mixed population Data.csv' file

11.4.2.1 Data for the first Gaussian ('Normal') Distribution

Add a cell to define the number of samples to be generated .. in this case 2000

```
number_of_samples = 2000
print(f"number_of_samples = {number_of_samples}")
```

```
number_of_samples = 2000
```

Add a cell to define parameters for the first of the Gaussian Distributions. Start wth a mean of -0.5 and a standard deviation of 0.7

```
sample_mean = -5.0
sample_sd = 0.7

print(f"sample_mean = {sample_mean}")
print(f"sample_sd = {sample_sd}")
```

```
sample_mean = -5.0
sample_sd = 0.7
```

The synthetic data-set can be generated using numpy.random.normal:

```
samples = np.random.normal(sample_mean, sample_sd, number_of_samples)
```

Note that Numpy uses the term 'normal' rather than the term 'Gaussian' - but they are the same thing.

11.4.2.2 Visualising the first population

Add a cell to visualise this data using a kde plot

```
fig, ax = plt.subplots(figsize = (10,6))

sns.kdeplot(samples, label='samples', shade=True, color="orange", ax=ax)

ax.set_xlabel("Turtle shell width")
ax.set_ylabel("Density")
ax.set_title("KDE Plot for 'samples' data-set")

plt.legend()
plt.show()
```

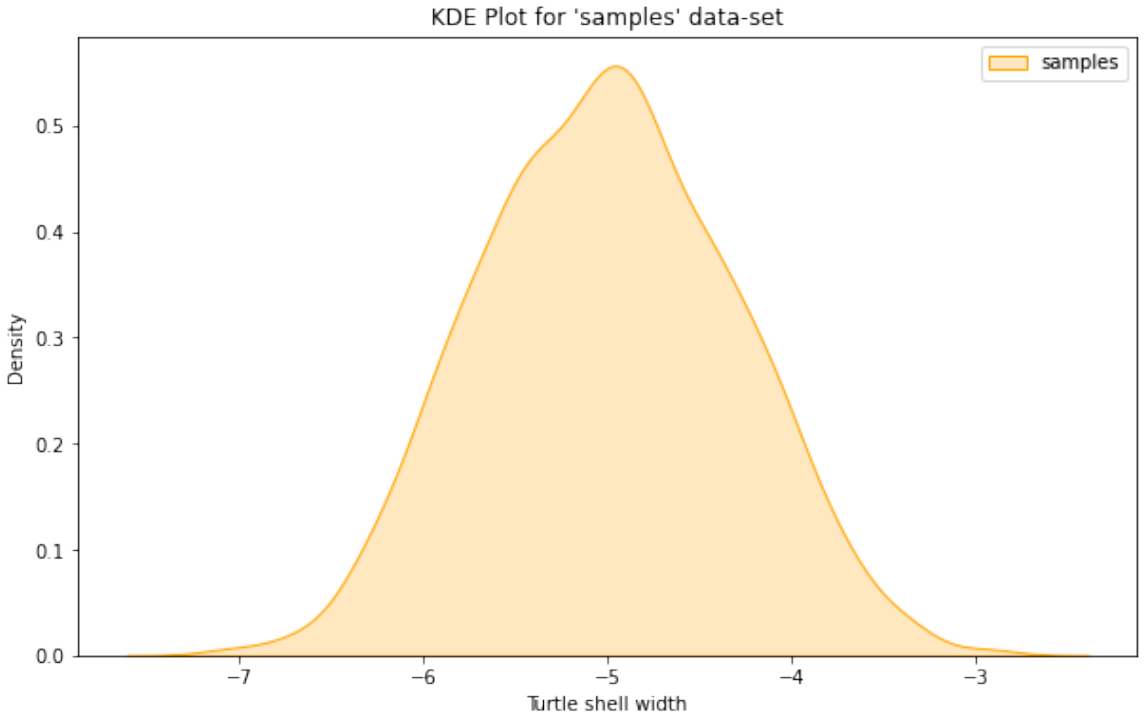

Figure 4: kde plot 1st synthetic population

11.4.2.3 Adding a second population

Add a cell to create a second synthetic distribution called 'samples2'. Set the parameters as follows:

- Mean = 1
- Standard deviation = 0.3
- Number of samples = number_of_samples

```
samples2 = np.random.normal(1, 0.3, number_of_samples)
print(f"samples2.shape = {samples2.shape}")
```

```
samples2.shape = (2000,)
```

Then add a cell to plot both data-sets on a combined graph:

```
fig, ax = plt.subplots(figsize=(10, 6))
sns.kdeplot(samples, label='samples', shade=True, color="orange", ax=ax)
sns.kdeplot(samples2, label='samples2', shade=True, color="blue", ax=ax)

# Set labels and title
```

```
ax.set_xlabel("Turtle shell width")
ax.set_ylabel("Density")
ax.set_title("kde plot for both synthetic populations")

ax.legend()

plt.show()
```

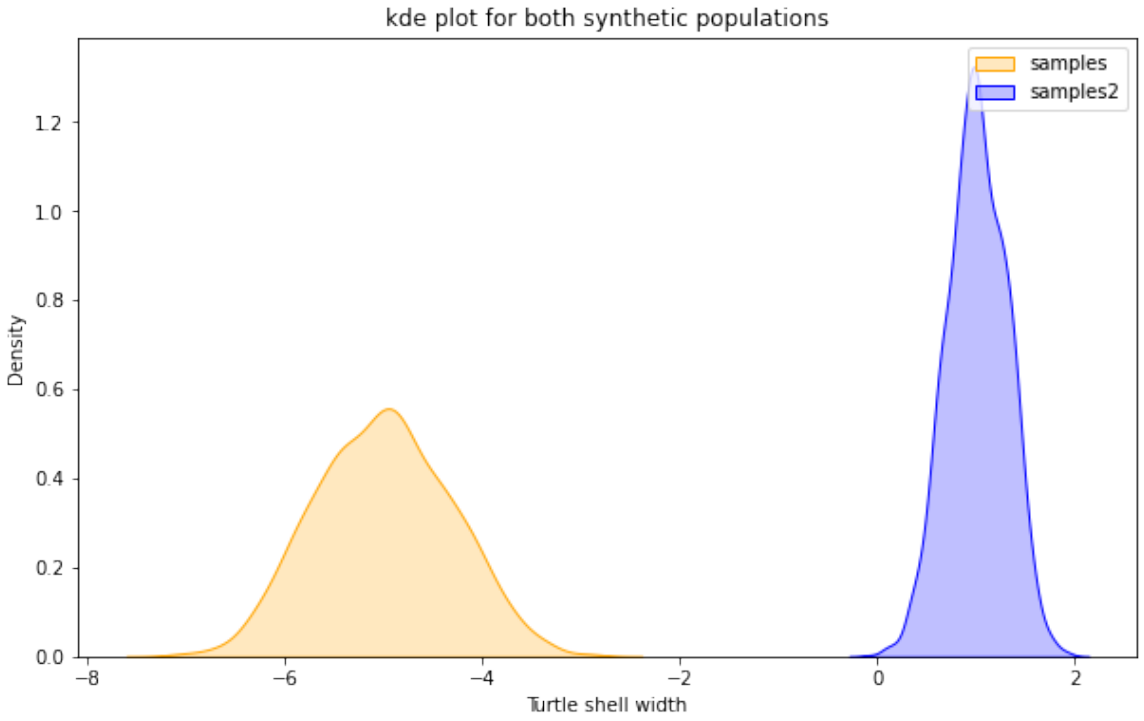

Figure 5: kde plot for each synthetic population plotted separately

At this point we have two, separate Gaussian Distributions. But in the original data the two populations are combined. We can mimic that by combining our two distributions.

Add a cell to that creates a new numpy array called ‘combined_samples’ by appending ‘samples2’ to ’sampels)

```
combined_samples = np.append(samples, samples2)
```

Then add a cell that creates a kde plot for the ‘combined_samples’ synthetic data-set. This is the first attempt at replicating the bi-model data from the original data-set.

```
fig, ax = plt.subplots(figsize = (10,6))

sns.kdeplot(combined_samples,
            label='combined_samples',
            shade=True,
            color="orange",
            ax=ax)

ax.set_xlabel("Turtle shell width")
ax.set_ylabel("Density")
ax.set_title("KDE Plot for 'samples' data-set")

plt.legend()
plt.show()
```

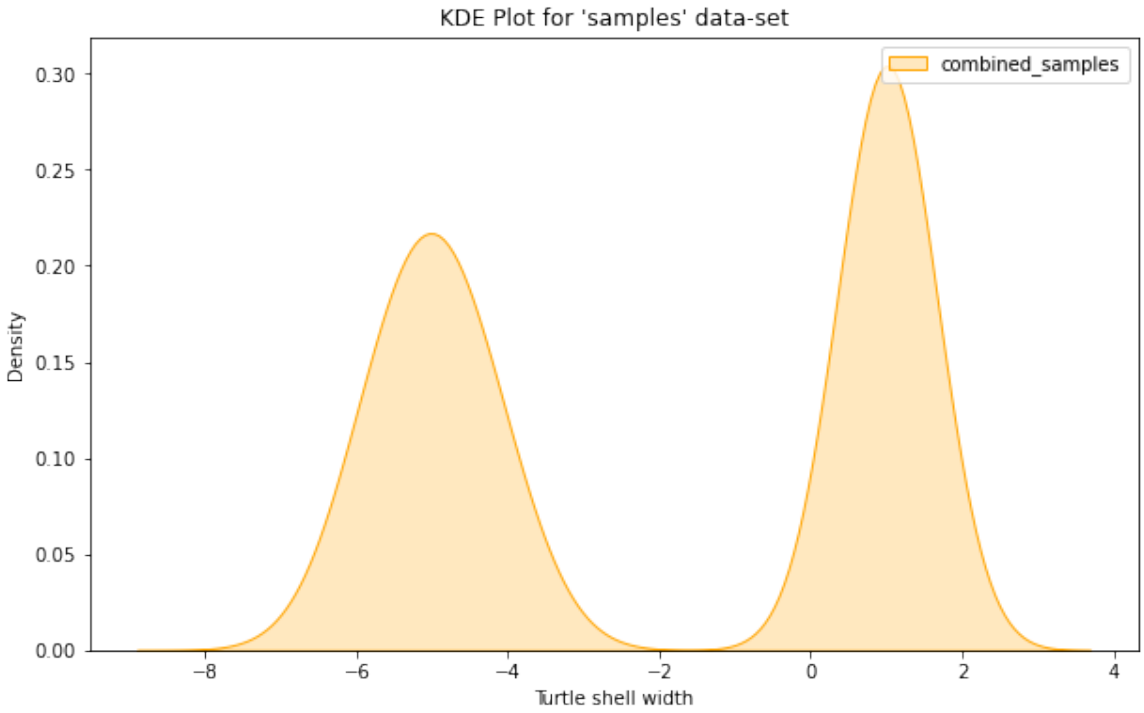

Figure 6: kde plot of the synthetic combined data

11.4.3 Experimenting to match synthetic data to given data-set

So we have created a bi-modal distribution which is the combination of two Gaussian distributions. It has broadly the same shape as our original data-set (two peaks), but the size and shape of those peaks does not match our original data.

We need to do some experiments to try to make the data-match.

Experiment by modifying the parameters (numbers) in the following cell to try to make your generated data match that of the original collected data. The cell below provides a combined plot of your generated data and the original data I supplied. Note that the match will never be perfect. After all, the data is generated randomly. However, you should be able to create a reasonably good match.

The graphic below is the result I obtained after some experimentation. You should try to produce something like this.

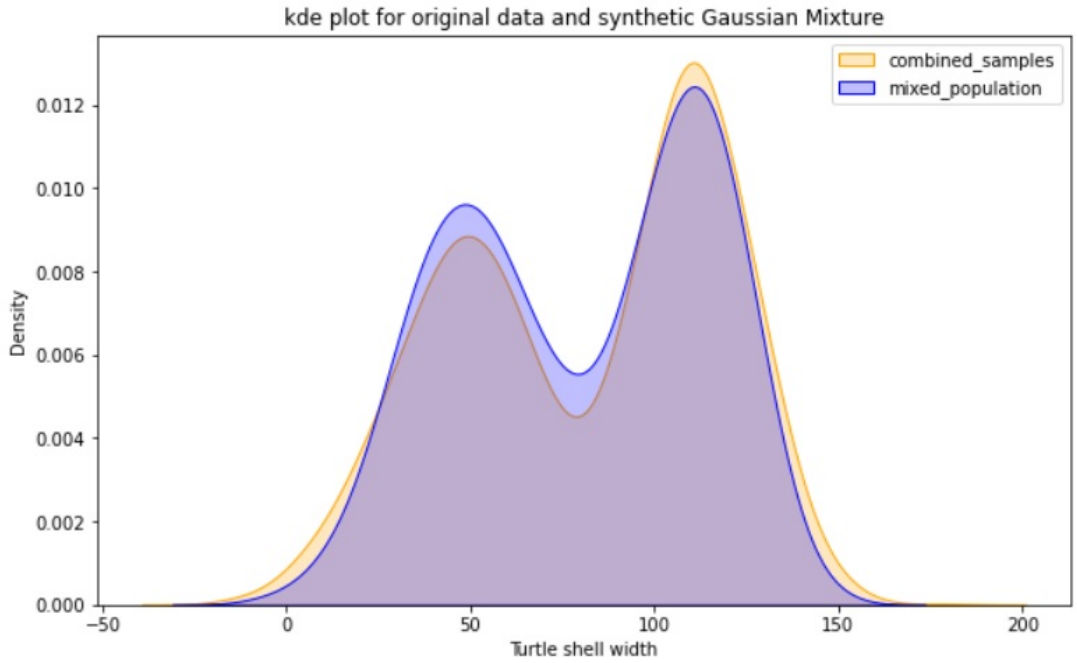

Figure 7: Example of a Gaussian Mixture Model generated manually

Create a single cell that replicates some of the above code into a single cell:

- Create a numpy array called 'samples' containing data for a Gaussian distribution, with a given mean and standard deviation and size
- Create a second numpy array called 'samples2' containing data for another Gaussian distribution with a different mean, standard deviation and size
- Create a combined data-set called 'combined_samples' by appending 'samples2' to 'samples'

```
samples = np.random.normal(48, 21, 1000)
samples2 = np.random.normal(112, 15, 1100)
combined_samples = np.append(samples, samples2)
```

Create a numpy array called 'mixed_population_np' from the 'The_Data' column of the original 'mixed_population' dataframe (we need this because it makes plotting the kde plot easier in the following cells).

```
mixed_population_np = mixed_population['The_Data'].to_numpy()
```

Now plot a kde chart of both the ‘mixed_population’ data (‘mixed_population_np’) and the synthetic, bi-modal data just generated (‘combined_samples’).

```
fig, ax = plt.subplots(figsize=(10, 6))
sns.kdeplot(combined_samples,
                label='combined_samples',
                shade=True,
                color="orange",
                ax=ax)
sns.kdeplot(mixed_population_np,
                label='mixed_population',
                shade=True,
                color="blue",
                ax=ax)

# Set labels and title
ax.set_xlabel("Turtle shell width")
ax.set_ylabel("Density")
ax.set_title("kde plot for original data and synthetic Gaussian Mixture")

ax.legend()

plt.show()
```

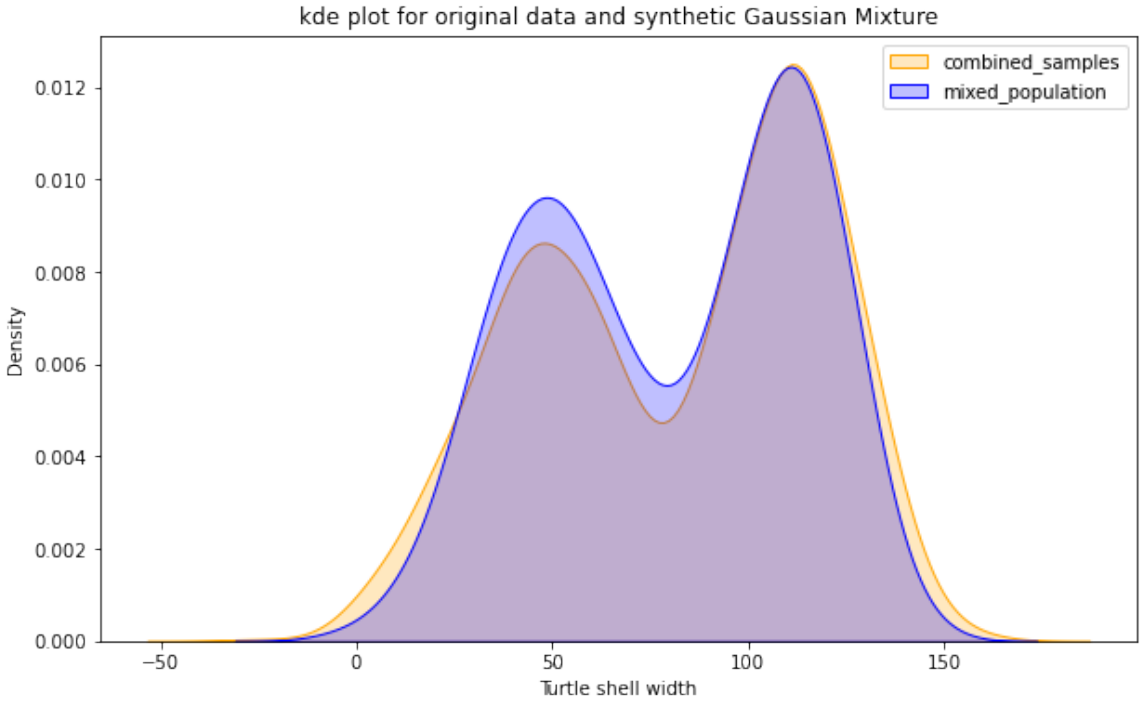

Figure 8: kde plots for original 'mixed_population' data set and synthetic data-set composed of two Gaussians

11.5 GMM the easy way - using sklearn

Having worked created a Gaussian Mixture Model manually, you will be pleased to know that sklearn includes a library function that implements a fast version of the complete algorithm. It is almost trivially easy to fit the required model.

```
gmm = GaussianMixture(n_components=2, covariance_type="full", tol=0.001)
gmm = gmm.fit(X=mixed_population)
```

Now add cells to print the modelled mean values:

```
print (f"gmm.means_ = \n{gmm.means_}")
```

```
gmm.means_ =
[[ 48.85430785]
 [109.5768361 ]]
```

Print the Standard Deviations:

```
print(f"np.sqrt(gmm.covariances_) = \n{np.sqrt(gmm.covariances_)}")
```

```
np.sqrt(gmm.covariances_) =
[[[17.7847952 ]]

 [[14.07012376]]]
```

Print the weights (that is, the proportion of items in each population). In the case of our data, their are an equal numer of items of each sub-population in the total population. Of course, that will often not be the case, and we need to recover the proportion in each.

```
print(f"gmm.weights_ = \n{gmm.weights_}")
```

```
gmm.weights_ =
[0.48271044 0.51728956]
```

As above, let's visualise this result.

Note that the '.item()' function is used in Numpy to extract a single value from an array. This is required since without it the result would be returned as a single item array.

```
mean_1 = gmm.means_[0].item()
mean_2 = gmm.means_[1].item()

sd_1 = np.sqrt(gmm.covariances_[0]).item()
sd_2 = np.sqrt(gmm.covariances_[1]).item()

totalPopulation = 4000

# Numpy requires an integer for the 3rd
# parameter of 'np.random.normal'
weight_1 = int(totalPopulation * gmm.weights_[0].item())
weight_2 = int(totalPopulation * gmm.weights_[1].item())
```

As previously, generate two synthetic data-sets based on these values, combine the results and plot them:

```
samples = np.random.normal(mean_1, sd_1, weight_1)
samples2 = np.random.normal(mean_2, sd_2, weight_1)
combined_samples = np.append(samples, samples2)
```

Then plot the result:

```
fig, ax = plt.subplots(figsize=(10, 6))
sns.kdeplot(combined_samples,
                        label='combined_samples',
                        shade=True,
                        color="orange",
                        ax=ax)
sns.kdeplot(mixed_population_np,
                        label='mixed_population',
                        shade=True,
                        color="blue",
                        ax=ax)

# Set labels and title
ax.set_xlabel("Turtle shell width")
ax.set_ylabel("Density")
ax.set_title("kde plot for original data and synthetic Gaussian Mixture")

ax.legend()

plt.show()
```

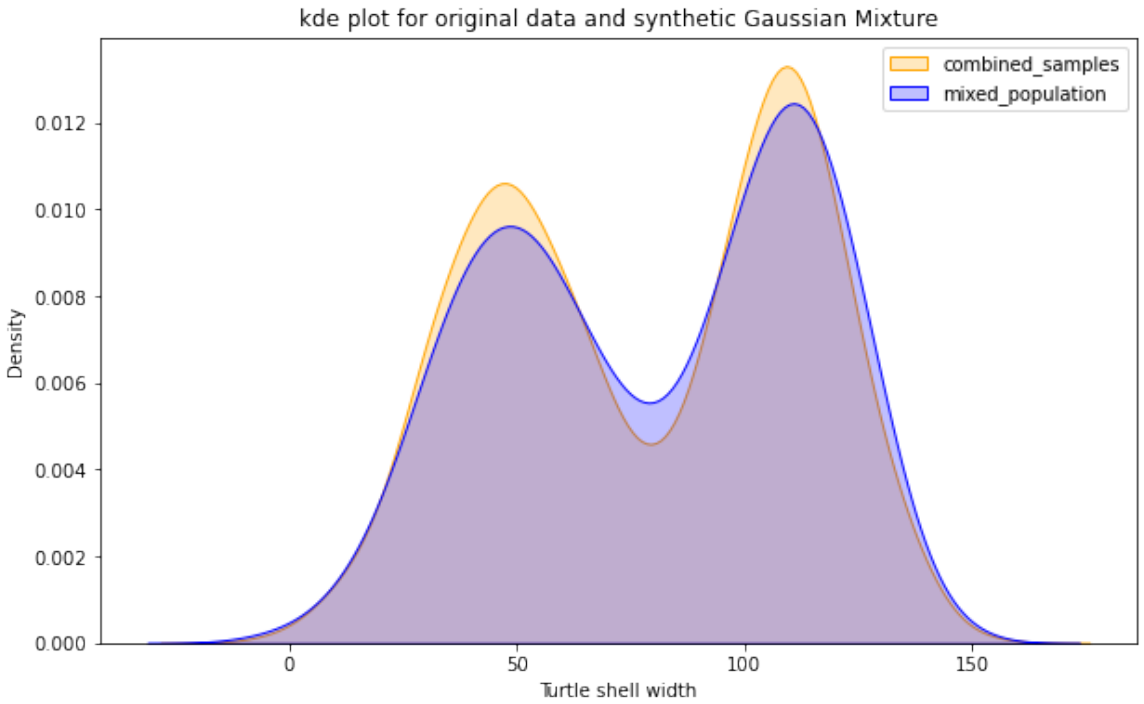

Figure 9: kde plots for GMM model for 'mixed_population' generated by sklearn

11.6 Optional (but fun!) - Astrophysics example

In this final section I am going to demonstrate how we might apply GMM to a problem in Astrophysics. We are going to use the model to locate the centre of galaxies from photographs.

(If you are less interested in the stars - then you might prefer to imagine this example as finding the centre of cells in medical images or possibly the centre of populations from archeological data. GMM is a very flexible tool!"

First lets create some 'image data'. You have to imagine that this was really obtained from astro-photography. In this case we are creating the data since it enables you to experiment with different images and to decide how well you think it works.

First we decide the number of 'dots' in our images. These will be points of light that have been detected by our imaging system. To start the experiment I am going to use 3000 points, but you may experiment with this value.

Add a cell to define a variable 'n_samples' with a value of 3000, then print it.

```
n_samples = 3000
print(f"n_samples = {n_samples}")
```

```
n_samples = 3000
```

Our first galaxy is purely spherical. It just appears as a circle of points in our final image.

To start, I have located this galaxy at coordinates 5,5. You should experiment with these values to see how well the GMM detects the centre of each galaxy.

```
# Student task step 4
galaxy_1 = np.random.randn(n_samples, 2) + np.array([5, 5])
```

Our second galaxy is not spherical. It is a disk galaxy and we are veiwing it from a slight angle.

```
C = np.array([[0., -0.7], [3.5, .7]])
galaxy_2 = np.dot(np.random.randn(n_samples, 2), C)
```

We combine all of this data into a single data-set to form our 'photograph':

```
astro_photograph = np.vstack([galaxy_1, galaxy_2])
```

Now we can visualise this data-set. I have set the colours to make it look like a photograph of the night sky through a powerful telescope.

```
ax = plt.axes()
ax.set_facecolor("black")
plt.scatter(astro_photograph[:, 0],
                astro_photograph[:, 1],
                c="white",
                s=1)
plt.show()
```

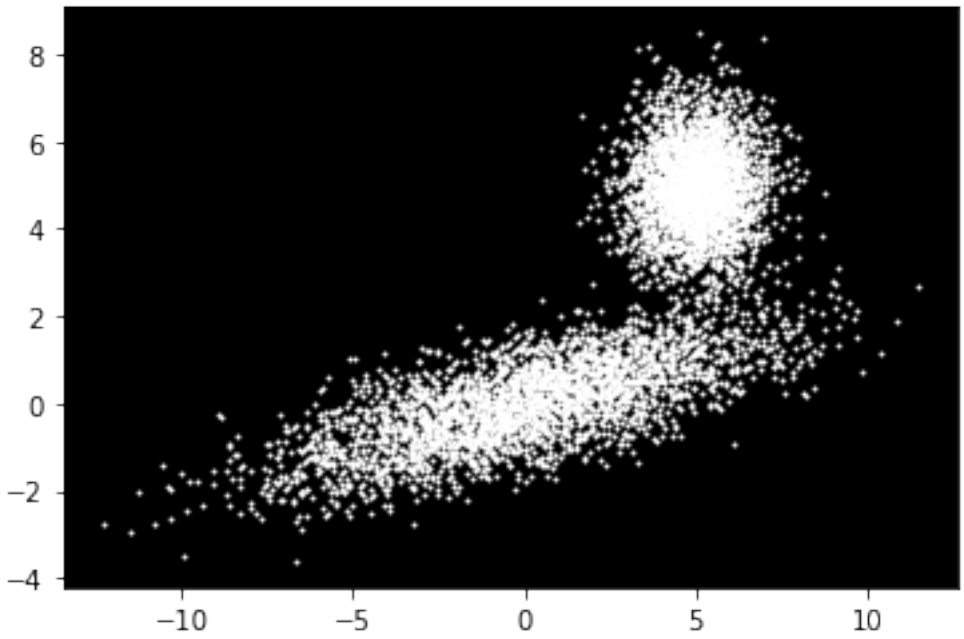

Figure 10: Synthetic data suggesting an image of two star clusters

Of course, in reality, the photograph is all we would have .. we would not know the true centres of these two galaxies. We could use a GMM to model the data and to decide the best estimate of where the centres are:

```
gmm = GaussianMixture(n_components=2, covariance_type="full", tol=0.001)
gmm = gmm.fit(X=astro_photograph)
```

What is the centre of each galaxy based on the results from the model?

```
galaxy_1_modelled_center = gmm.means_[0]
print("Centre of galaxy 1 modelled as ", galaxy_1_modelled_center)

galaxy_2_modelled_center = gmm.means_[1]
print("Centre of galaxy 2 modelled as ", galaxy_2_modelled_center)
```

```
Centre of galaxy 1 modelled as  [5.01626246 4.98477159]
Centre of galaxy 2 modelled as  [0.04811604 0.03387559]
```

Student task: Experiment by changing the location of the two galaxies my modifying the values in the cell labelled 'Student task step 4'. To what extent is the GMM able to recover the true centre of the two galaxies?

11.7 Fitting a Gaussian Mixture Model Algorithmically

This section is unfinished and TBD

11.7.1 Determing the value of a Gaussian for a specific x value

We will first need a function that computes a continuous Gaussian random variable. This will be used to compute the probability of getting a particular value within a distribution with a specified mean and standard deviation. Such a function is available within 'scipy.stats'.

The function below answers this question: If we have a Gaussian Distribution with mean = 7 and a standard deviation of 2, what would be the probability of selecting the value '4' at random.

```
x_value = 10
mean = 7
standard_dev = 2
print(f"norm.pdf(x, loc=mean, scale=standard_dev) = ")
print(f"    {norm.pdf(x_value, loc=mean, scale=standard_dev):.3f}")
```

This can be illustrated as follows:

```
x = np.linspace(mean - 3*standard_dev, mean + 3*standard_dev, 1000)
pdf_values = norm.pdf(x, loc=mean, scale=standard_dev)

plt.plot(x, pdf_values, label='Gaussian Distribution')
plt.scatter([x_value], [norm.pdf(x_value, loc=mean, scale=standard_dev)], color='red', lab
plt.xlabel('X-axis')
plt.ylabel('Probability Density')
plt.title('Gaussian Distribution')
plt.legend()
plt.show()
```

11.7.2 The core GMM algorithm

Define 'k' to be the number of populations that we want to model .. in this case 2

```
k = 2
print(f"k = {k}")
```

The algorithm allows us to model mixed populations of different sizes. The proportion of the total population in each sub-population is defined in the 'weights' variable. This is initialized with the assumption that there is an equal number of members of each population

```
weights = np.ones((k)) / k
print(f"weights = {weights}")
```

We also need to initalise the mean value and the variance (square of the standard-deviation) for each population. These are arbitrary values - since the algorithm will update these values step-by-step. Normally these values would be set as random numbers. However, in this case I am going to first set the values to be known, fixed values. This will ensure that we get similar results when demonstrating the algorithm in class.

You can experiment with setting the initial values to some random values.

```
means = np.array([10, 30])
variances = np.array([1, 1])

print(f"means = {means}")
print(f"variances = {variances}")
```

'eps' below is just a small number intended to prevent a 'divide by zero' error in the following algorithm

```
eps=1e-8
```

Just for reference we will maintain a count of the number of iterations that we execute before completion:

```
iterationCount = 0
```

The following cell implements the core of the algorithm.

Note that this cell only implements a single iteration of the algorithm. To obtain a good model, the following cell must be executed multiple times. I had to execute the following cell around 80 times to obtain a good model. I created the code in this way so that you could watch the model evolving towards a good solution.

Execute the following cell ('Student task step 1' multiple times and look at the printed results as the converge on a good model. You may also execute the following two cells ('Student task step 2' and 'Student task step 3' to visualise the evolving model

Optional student task : After a while you may get bored of repeatedly executing the following cell by hand. If you wish you may modify the following code to add an outer loop that executed the code multiple times.

```
# Student task step 1

iterationCount = iterationCount + 1
print("Iteration: ", iterationCount)

# Initialise the vector of likelihood values
likelihood = []

# Expectation step
for j in range(k):
    # print("Means[j]", means[j], "Standard Deviation", np.sqrt(variances[j]))
    # likelihood.append(norm_dist(X, means[j], np.sqrt(variances[j])))
    likelihood.append(norm.pdf(x,
                               loc=means[j],
                               scale=np.sqrt(variances[j])))

likelihood = np.array(likelihood)

b = []

# Maximization step
for j in range(k):
    print("Population: ", j+1)
    # use the current values for the means and standard deviations to compute
    # the likelhood that each point would be within that population
    b.append((likelihood[j] * weights[j]) /
             (np.sum([likelihood[i] * weights[i] for i in range(k)], axis=0)+eps))

    # update mean and variance
    means[j] = np.sum(b[j] * x) / (np.sum(b[j]+eps))
    print("    Mean : ", means[j])
    variances[j] = np.sum(b[j] * np.square(x - means[j])) / (np.sum(b[j]+eps))
    print("    Standard deviation :", np.sqrt(variances[j]))

# update the weights
for j in range(k):
    weights[j] = np.mean(b[j])

```

```
print("weights[i] :", [weights[i] for i in range(k)])
```

We can visualise the generated model by generating a 'synthetic' distribution as in section 1:

```
# Student task step 2
samples = np.random.normal(means[0], np.sqrt(variances[0]), 2000)
samples2 = np.random.normal(means[1], np.sqrt(variances[1]), 2000)
combined_samples = np.append(samples, samples2)
```

```
# Student task step 3

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=[8, 5])
ax.hist(mixed_population, bins=50, density=True, alpha=0.5, color="#0070FF")
ax.hist(combined_samples, bins=50, density=True, alpha=0.5, color="#ff7000")


# Annotate diagram
ax.set_ylabel("Probability density")
ax.set_xlabel("Arbitrary units")

# Draw legend
plt.show()
```