# Python For Applied Machine Learning - Practical 5

Michael Halstead

April 29, 2024

## 1 Introduction

What will we be covering today?

1. Multi-variate Gaussian; and
2. Multi-class Gaussian mixture models.

## 2 Multi-Variate Gaussian

For this exercise we will use the data we created in Practical 4, *data/mvg_data.pkl*. If you need to re-create this data you can do it by re-running the create data from that practical. Load the dictionary into a variable, then extract the *X* and *LX* keys from that variable into new variables. Using these two pieces of information you will split *X* into two sets *X0* where *LX* is 0 and *X1* where *LX* is 1.

Next, we will code up the multi-variate Gaussian class in a new library file called *Gaussian.py*. This part is a little bit math heavy, I will show you some of the maths but the important parts will be highlighted. First of all, the MVG calculates the distribution of it's samples through the mean and covariance matrix. In our case we will have 2 dimensions, but this scales to N dimensions. The overall equation for a Gaussian distribution is:

Probability Density Function (PDF)

$$f(x) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} exp^{-\frac{(x-\mu)^T \Sigma^{-1} (x-\mu)}{2}}, \quad (1)$$

where $\mu$ is the mean $\Sigma$ is the covariance, $|.|$ represents the determinant and $\Sigma^{-1}$ is the inverse matrix. So the first thing we need to compute is the mean and covariance of the data (X), where x is of shape (m,n) where m is the number of samples and n is the dimensionality. You should be up to speed with how to calculate the mean of the input data (mean of each dimension). The covariance is a little bit more tricky, you could just use *np.cov* but I want you to think

about how to implement it mathematically, you can check your working with *np.cov*. Essentially the covariance formula is:

$$Cov[i,j] = \frac{\sum_{n=0}^{N-1}(X_{i,n} - \mu_i)(X_{j,n} - \mu_j)}{N}. \tag{2}$$

In this case we use $N$ as the divisor as we don't care so much about bias, so when you use *np.cov* you'll have to make sure you set the bias flag to *True*. Your first thought here might be to use multiple for loops depending on the size of $N$ but there is a simpler way to do this, dot product. You can use *np.dot(..)* here or you can simply use the matrix multiplication operator @. I will let you work out how the matrix dimensions need to be given to the function/operator, keeping in mind that we need an $n * n$ matrix for covariance not an $m * m$ matrix...

The next step is a little bit more complex and I will briefly step you through the process, there are a lot of good resources online for working out the log likelihood function and if you are struggling with the maths I recommend you look there, here is a start:
`https://blogs.sas.com/content/iml/2020/07/15/multivariate-normal-log-likelihood.html`

We need to get from the above general function for a Gaussian to something that predicts how close we are to that distribution (a metric).

$$loglike = log(\frac{1}{\sqrt{(2\pi)^n|\Sigma|}} exp^{-\frac{(x-\mu)^T\Sigma^{-1}(x-\mu)}{2}}) \tag{3}$$

Multiplications in a log can be separated as additions:

$$loglike = log(\frac{1}{\sqrt{(2\pi)^n|\Sigma|}}) + log(exp^{-\frac{(x-\mu)^T\Sigma^{-1}(x-\mu)}{2}}) \tag{4}$$

The log and exp's cancel each other out.

$$loglike = log(\frac{1}{\sqrt{(2\pi)^n|\Sigma|}}) + -\frac{(x-\mu)^T\Sigma^{-1}(x-\mu)}{2} \tag{5}$$

Divisions in a log can be separated by subtractions and the log of 1 is zero:

$$loglike = -log(\sqrt{(2\pi)^n|\Sigma|}) + -\frac{(x-\mu)^T\Sigma^{-1}(x-\mu)}{2} \tag{6}$$

And we keep doing these little trick until we get to

$$loglike = -(\frac{n}{2}log(2\pi)) + -(\frac{1}{2}log(|\Sigma|)) + -\frac{(x-\mu)^T\Sigma^{-1}(x-\mu)}{2} \tag{7}$$

You can calculate the log determinant using *np.linalg.slogdet( cov )[1]* and the inverse matrix using *np.linalg.inv( cov )*. What you can see here is that there are

two constants: $-(\frac{n}{2}log(2\pi))$ and $-(\frac{1}{2}log(|\Sigma|))$ which can be calculate directly after we calculate the mean and covariance.

The final term $-\frac{(x-\mu)^T\Sigma^{-1}(x-\mu)}{2}$ needs to be computed once we have our input $x$. So what does our class look like?

```
class MVG():

  def train(self, X):
    compute the mean of x and store as a member
    compute the covariance of X and store as a member.
    # once these are computed you can calculate the constants below

  def _precalculations(self,):
    calculate the inverse sigma
    calculate the first constant
    calculate the second constant
    Store the sum of all constants as a member

  def log_likelihood(self, X):
    create an empty vector of the number of samples in X
    calculate the residuals X-mu
    over the samples in x calculate the log likelihood value and
    store in your vector
    return the vector
```

Now once you have coded up your class we will create two objects, one that uses *X0* and another that uses *X1* as the training data. Print out the mean and covariance that is calculated by your class and see how close it is to what you expected (check the previous practical).

Now from your loaded data extract the *Y_0_1* data and labels (*LY_0_1*), store them in variables. Calculate the log likelihood from both of your objects for the the *Y_0_1* data. To do this you will need to create a matrix of size $N \times 2$ where $N$ is the number of samples and 2 is the number of objects we have. Store the outputs from your log likelihoods in this vector. Than you will need to classify this vector by using the argmax call in numpy.

Once you have classified the vector we can then compute the metrics, first calculate and print the accuracy score, then calculate and display the confusion matrix.

# 3 Multi-Class Gaussian Mixture Model

In this exercise we will classify colour pixels using Gaussian mixture models (GMMs). GMMs are an extension of what we have already done in this practical where we are trying to fit $N$ (the number of) Gaussians to some data, where we set $N$. $N$ in this case does not correspond to the number of channels in an image, it is a value we set through trial and error.

Figure 1 gives you an overview of how a 3 component GMM would be fit to 1 dimensional data. First we estimate the mean and covariance of three distributions within our data using the EM steps. Once we have created these

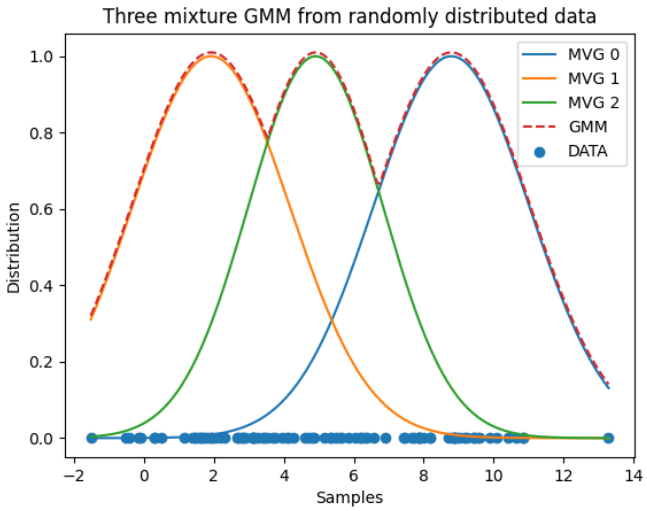


Figure 1: Example of a Gaussian mixture model being fit from 1D data.

distributions (orange, blue, and green) we can then theoretically fit the GMM locations as the maximum value of the three MVGs (red dotted line).

We can then use our GMMs to classify different objects such as colour pixels. Figure 2 shows an example of how we would do this for two GMMs trained on different data. You can see that one set of data is distributed to one end of the plot, this is evident when we look at the GMM lines. If we want to classify a point we will use the density representation at that point, i.e. which one of the two GMMs has the higher probability, we then say that point belongs to that GMM.

Now we will develop a GMM class of our own using *from sklearn.mixture import GaussianMixture*. We will also put this in the *Gaussian.py* library, the pseudo-code is as follows

```
class MultiGMM():
  def __init__(self, n_components):
    store the number of components in a member.
    store the gmms as an empty dictionary (we will use this later
    to store our gmms)

  def fit(self, d):
    d is a dictionary that has the colour name as the key and the
    pixels as the values in a (n,3) matrix
    iterate over the key and value pairs in d
    using your gmms member create a Gaussian Mixture Model using
    the key is the same as d and the input data is the value of d.

  def predict(self, X):
    X is an input of shape (N,3) not a dictionary.
    create a matrix of size (N, len(gmms))
```

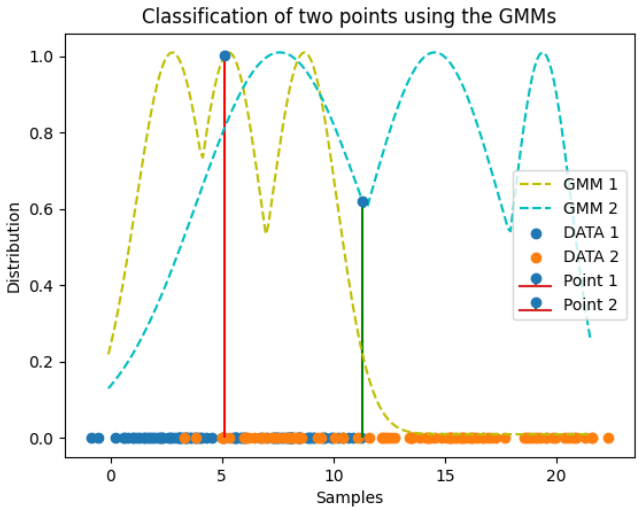


Figure 2: Example classifying two points using two GMMs trained on different data.

```
    for each of your gmms
      store in your created matrix the output from GMM
    score_samples(X)
    classify your matrix by using argmax and return the
    classification vector.
```

Now how will we use this class? First, we need to load in the data, for this we will use the colour snippets dataset in your data folder. The following is the pseudo-code for what we will do with this class:

```
Create an empty train dictionary.

Read in the colour snippets data for each colour folder
    individually and store as an (N,3) matrix.
Use the train_test_split with 0.3 test size to split each colours
    matrix.
Store the training set into a dictionary with the name of the
    colour folder as the key.
Store the testing set in a single matrix (not colour specific)
For each sample in the testing set you should also compute and
    store in a separate variable a label, i.e
    [0,...,1,...,2,...,3,...,4,...,5,...,6,...,7,...]

Create your gmm object.
Fit your gmm object with your training data (dictionary).
Predict your test set (M,3) matrix.

Compute and display the accuracy score using the predictions and
    the labels.
Compute and display the confusion matrix using the predictions and
```

```
    the labels.
```