Chapter 4

Advanced Topics

We have now covered the bulk of the course regarding statistical inference. In this chapter we will cover some more advanced topics which are particularly relevent for data analysis and machine learning.

4.1 Measurement error and forward modelling

Measurement error, sometimes also called *observation error*, is the difference between the measured value of a parameter and the true value. Note that this is not the same as the *bias* which we have discussed above which actually says something about the expected value of the estimate compared to the true value. We want an unbiased estimator which will asymptotically tend to the true value, however the *measurement error* quantifies the difference between the value we measure on some finite sample and the true value.

There are two potential sources for measurement error. One is the *statistical error* due to random fluctutations in the sample that we have. The other is the *systematic error*, sometimes confusingly referred to as the *statistical bias*.

Measurement error has some important consequences for regression problems, which will be discussed in more detail in the M1 module, and in particular statistical error in the dependent variable of a regression is not problematic aside from increasing the χ^2 value.

4.1.1 Forward modelling

Forward is an important concept in the application of machine learning. What forward modeling means is using a model to *predict* what you might expect to observe and then comparing this to what you actually do observe. This is how we train machine learning algorithms in supervised approaches. We build a model, which can start with a simple guess, we then use this model to predict an outcome. We then want to minimise the difference between the predicted outcome and the true outcome on a training sample. This difference can be parameterised in some of the ways we have seen above, *i.e.* it could be a least squares of a maximum likelihood.

Consider a training sample containing a feature set of N data points, $X = \{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N$, where the $\vec{x}$ notation symbolises that each data point contains a vector of features. The training sample will also need to contain the target value which we would like to be able to

predict, $Y = \{Y_1, Y_2, \ldots, Y_N$. We can then define some function (built in any way we like) which produces an output given the values of the feature vector,

$$y_i = f(\vec{x}_i; \vec{\theta}), \tag{4.1}$$

where in this notation $\vec{\theta}$ are the parameters of the model. If you've been paying attention in the M1: Machine Learning module you should realise that for typical regression problems these will be sets of weights and biases, w_j and b_j. In order to train our algorithm (find the best function) we will minimise the difference between the predicted output y_i and the target output Y_i using a numerical approach by adjusting the values of the parameters within $\vec{\theta}$:

$$\chi^2 = \sum_i^N (y_i - Y_i)^2 = \sum_i^N (f(\vec{x}_i) - Y_i)^2. \tag{4.2}$$

This is an example of forward modelling. When we are performing such methods we need to make sure that we haven't overfitted our function f such that it goes through every single point in the training sample. We can do this by using an adjoint, statistically independent, *validation* sample to check that on average, or in certain regions of the predicted output, that the prediction for the training sample is not wildly different from the precition for validation sample. If they are very different then it suggests that parameters of the function f have been tuned too specifically to certain artefacts or flucutations within the training sample.

4.2 Optimisation

We have seen several examples of optimisation already in this course but let's broaden the discussion a bit to just clarify what is meant by optimisation and what algorithms are out there that can help us.

From a mathematical sense optimisation means that given some function of a set of variables, $f(\vec{x})$, finding the values $\vec{x}_0$ which either minimise or maximise the response. It does not matter if we want to minimise or maximise because we can always simply take the negative of the function. We have seen examples of this when we want to maximise the likelihood or minimise the χ^2.

4.2.1 Optimisation algorithms

If I ask you to think about how to numerically figure out where the minimum of some function is perhaps the first thing that comes to mind is scanning the parameter space. You just plug some values of $\vec{x}$ into your function f and keep track of where the smallest value was. This is fairly trivial in one-dimension, especially if you know roughly where to start and can narrow your scan range. But if $\vec{x}$ has a very high dimensionality this quickly gets out of hand, the number of scan points goes with n^m.

If you sit and think for a few more minutes about how you would design an optimisation algorithm it won't take you long to realise that having knowledge, or at least an estimate, of the gradient of the function, $\partial f/\partial x$, will help you considerably. Of course you can also estimate this numerically as part of your scan procedure. If you test some point to get the

value of $f(\vec{x})$ then move one of the values a bit and get a higher value then you already have an estimate of the gradient (difference in y over difference in x of the two points you've scanned). That allows you to ascertain if you are moving in the right or wrong direction (at least it does in one-dimension if the derivative is not rapidly changing).

There are lots of optimisation algorithms out there and I do not have the time nor the inclination to cover them all. Instead what I will do is point out which of them, and in particular their implementation in different `python` packages, are useful for the kind of problems we will be solving.

Algorithms within `scipy.optimize`

The `scipy.optimize` library contains almost everything you'll need for day-to-day optimisations. It has a variety of different algorithms for searching for both local and global minima. It is normally quite fast and quite effective. One it's drawbacks is that it will typically only return you the result of the minimum point but in some cases we also want to know estimate of the derivative, or in the case of maximum likelihood fits the double derivative which we can use to get estimates of the uncertainties.

The migrad algorithm

You will have seen above that I make extensive use of the `iminuit` package when maximising likelihoods or minimising χ^2. That is because that library is specifically designed for these use cases and also has built in cost functions, which is very useful. The `iminuit` library uses the `migrad` algorithm, which *does* provide an estimate of the second derivative using the Davidson-Fletcher-Powell (DFP) formula [21]. An extension of the DFP formula is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) formula [22] which is one of those implemented in `scipy.optimize`. The `migrad` algorithm will therefore return not only the optimum values but also an estimate of the covariance matrix based on the inverse of the double differential of the likelihood.

Gradient descent

The gradient descent algorithm is one of the most common optimisation algorithms in machine learning. There are different varieties of gradient descent and I will only give a basic overview here. For most applications you will probably make use of the so-called "Adam" optimiser which is version of gradient descent.

The gradient descent method deploys numeric calculations of the gradient in order to "descend" to the minimum of the "loss" or "cost" function. The loss function is the one which describes differences between the true and predicted values of the model, for example the χ^2-like loss function shown in (4.2). The gradient descent algorithm has one free parameter, the so-called *learning rate* or *alpha* value, which defines the step-size for the descent. Different descent algorithms will update the learning rate in different ways as the algorithm descends, it should get smaller as we approach the minimum so that we find the minimum more accurately. When you train ML algorithms the *learning rate* is a hyperparameter of the training, if you select a small value the descent will be more reliable but the training will take much longer. A large value will speed up the trianing but can overshoot the true minimum. A pictoral example is shown in Fig. 4.1.

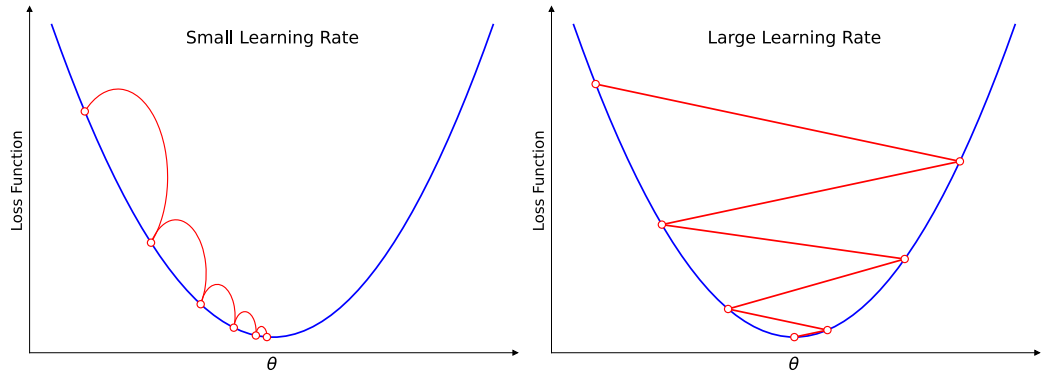

Figure 4.1: Demonstration of the gradient descent algorithm with an adapative learning rate.

4.2.2 Local minima and saddle points

Many optimisation algorithms are designed for local minimisation. In other words they are designed for only one minimum in the local vicinity of where they start. If you have a problem with many local minima then it may be better to use a global optimisation algorithm, for example the "basin hopping" algorithm. It is worth noting that saddle points can also cause issues with minimisation, as the minimiser thinks it has found the minimum (gradient is zero) but in fact it should have kept going.

When training deep neural networks you may also run into difficulties with vanishing gradients (network is no longer learning) and exploding gradients (parameters of the model tend to infinity). There are some specific ways of dealing with these problems but you they are specific to the problem at hand and the particular network you are training. You will have to work it out!

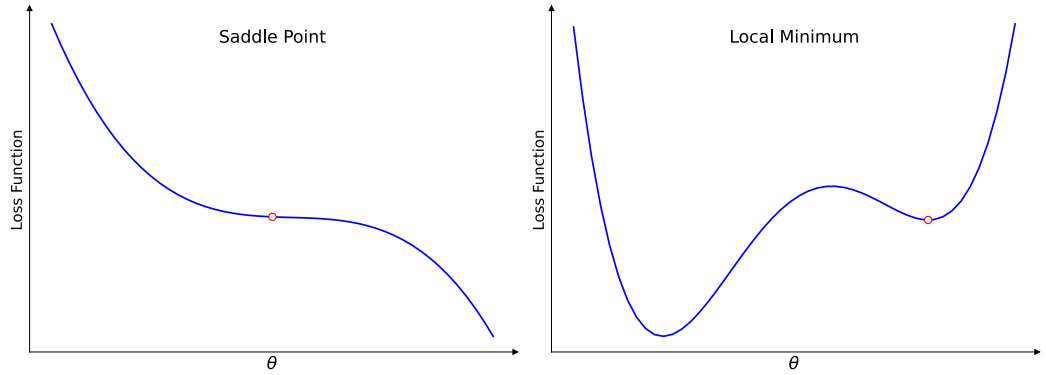

Figure 4.2: Examples of minimisations that get stuck at a saddle point (left) or in a local minima (right).

4.2.3 Parameters in optimisations

Nearly all optimisation algorithms are designed to work best with parameters with are $\mathcal{O}(1)$ so it can help a lot if try to ensure that the parameterisation of your problem, especially if you are struggling with the optimisation steps, is such that the parameters are near one.

Most machine learning algorithms will transform the feature set in such a way that they look standard normally distributed, there are various ways of doing this and several useful standard tools. One of the most common is a quantile transformation. The reason this is done is to keep the network weights and bias values near unity.

Any optimisation algorithm needs to be provided with a starting value. This is another reason for keeping parameterisations such that the values are near one, it's then easy to set the starting values to one. Choosing sensible starting values can be vital for finding the correct minima but you should also be aware that this can cause biases. A good check for optimisation robustness is that convergence can occur for a variety of different starting values. In particular local optimisation algorithms can suffer from getting "stuck" in local minima depending on their starting values. Some examples of a few different optimisation routines built into `scipy.optimize` are shown in Fig. 4.3. In this case you can see a function with multiple minima is very dependent on the starting values and the specific algorithm being chosen. The "brute force" method shown on the right requires a range rather than a starting value and will always find the global minimum but may be rather slow.

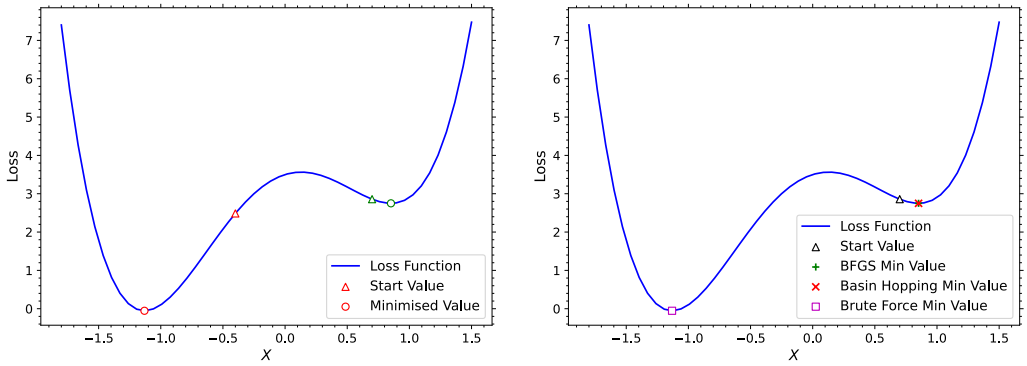

Figure 4.3: A demonstration of different optimisation algorithms for a loss with multiple minima. The optimisation routines used in this case are all from scipy.optimize and include the `minimize`, `basinhopping` and `brute` routines.

4.3 Regularisation

Regularisation refers to specific terms that we add to the loss function in order to steer or change the optimisation outcome. These can include certain constraints, terms that attempt to avoid over-fitting or specific priors we may have. Regularisation terms in a machine learning context are often included to restrict the complexity of the solution and to avoid over-training.

Without any regularisation most machine learning algorithms will simply perfectly learn the data sample they are trained on. This is clearly undesirable because the training sample is only a subset of the data and we want our classifiers / regressors to learn *general* properties not *local* properties specfic to the dataset. This is achieved with a regularisation term, so that the overall loss function takes the form

$$L = \sum_{i}^{N} V(f(x_i), y_i) + \lambda R(f), \tag{4.3}$$

where V is the emprical loss function (*e.g.* likelihood or least-squares) that we have seen before and R is a regularisation term which will typically depend on the complexity of f relative to the number of points in the sample N. The λ parameter is then an input hyperparameter which we can tune to signifiy the important of the regularisation term.

If we consider the plot show below, Fig. 4.4, we can see that two loss function both "fit" the data equally well, in other words their χ^2 difference will be at a minimum. However one is much simpler than the other so we may chose to add a penalty to the loss for the increased complexity.

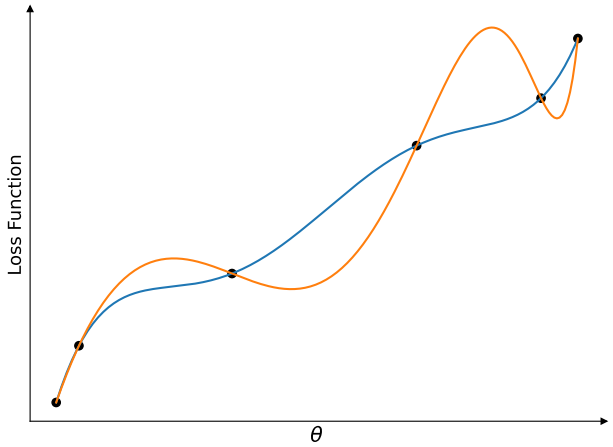

Figure 4.4: A demeonstration of two network predictions which give an equally low empirical loss. However, the orange curve has a much higher degree of complexity so would be penalised by a regularisation term.

We have already seen some examples of regularisation when discussing ML fits. One example is the extended term that is added for EML fits. Other examples may include constraint terms in our likelihood, or systematic terms. In the machine learning context we are often not actually interested in using the likelihood to infer statistical properties (we are just trying to train our algorithm by minisming the likelihood) so we can add penalty terms with rather arbitary meanings as we like.

Another important aspect of regularisation, which I will not cover in this module, but comes in the Advanced Machine Learning module, is the concept of *early stopping*. We may often have a regularisation term that corresponds to the number of epochs we have been training for without a given improvement in the loss function itself.

4.4 Density estimation

Something we often want to do is approximate the probability density of a dataset. We have already seen one method for doing this back in Chapter 1 which is to make a histogram. We can even normalise that histogram so that the product of the entries and the bin widths, summed over all bins, gives unity (so that it really is a probability density). This, however, gives us a discontinous distribution which is a step function at each of the bin boundaries. One method for obtaining a smooth and continous probability distribution is called *Kernal Density Estimation* (KDE) which replaces the step function of the histogram with some

kernel distribution which smooths it out.

If we have a dataset in some variable $X = (X_1, X_2, \ldots, X_N)$ of i.i.d samples drawn from a distribution with an *unknown* density then we can approximate the shape of that density as

$$\hat{f}(X) = \frac{1}{Nh} \sum_{i}^{N} K\left(\frac{x - x_i}{h}\right), \tag{4.4}$$

where K is a probability density function called the *kernal* (more on that in a moment) and h is a tuneable parameter called the *bandwidth*. Quite often the kernel used is simple a Gaussian function and in this sense we then model a dataset as a sum of Gaussians with the same width (given by the bandwidth) but at different locations, depending on the value in the sample. A pictoral example is shown below for a small dataset generated from a uniform distribution.

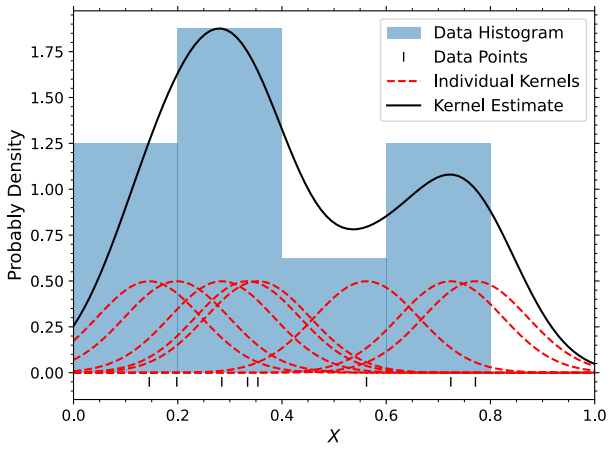

Figure 4.5: A demonstration of generating a smooth, continous density estimate based on a sum of Gaussians (the kernel) at each datapoint. In this case the bandwith (the width of each Gaussian) is set to 0.1.

Kernel density esimation is really useful, and not just for visualisation purposes, it can help with model building, dataset fitting and estimation. The user choice of the kernel type and the kernel bandwidth are however important and will give very different results depending on the choices made. There are many good packages out there for kernel density estimation. It's worth noting that density estimation in this way is a useful visualisation and analytical tool but it can be rather slow, especially for large samples. You of course need to find a balance between a distribution which has too narrow a bandwidth (picks up various fluctutations) but also not too broad (such that it misses important features). Density estimation of this type can be particularly problematic in mutiple dimensions when the different random variables have very different scales, so you should consider a transformation to keep variables $\mathcal{O}(1)$. Furthermore these density estimates have particular issues at physical boundaries and so it is worth considering applying a reflection at a boundary so that the estimate does not tend to zero or infinity at the boundary. A few examples a changing kernels and bandwidths for a larger dataset are shown in the plot below, where in this case I have made use of the `scikit-learn` `KernelDensity` class.

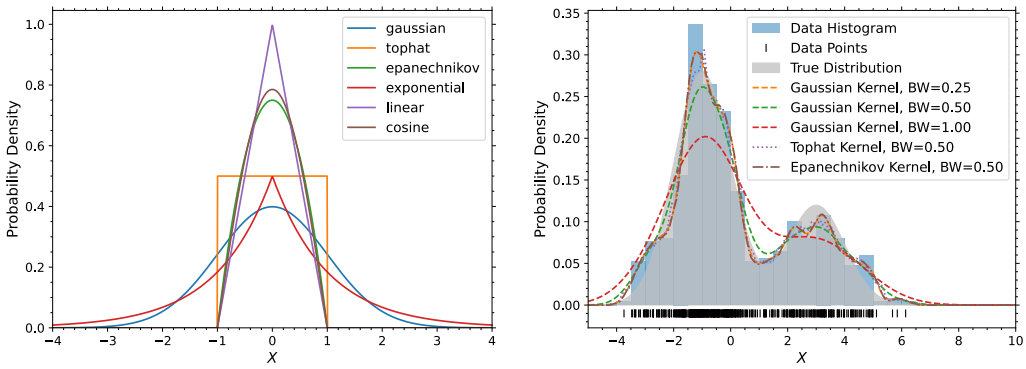

Figure 4.6: Left: an example of various different kernels available in `scikit-learn`. Right: A demonstration of kernel density estimates with different kernels and different bandwidths on a dummy datasets generated from the sum of two disjoint Gaussians.

4.5 Expectation maximisation

In this section I give a brief overview of a very cute method which deploys the maximum likelihood approach but in cases where there may be so called *hidden* or *latent* variables. This allows us to infer properties of dataset but about variables which we cannot necessarily quantify. It has a lot of uses in finance, economics and the social sciences where it is typically hard to actually quantify specific variables. For example if you are trying to determine the "happiness" of a population. You may have the assumption that this is related to their wealth or their health or various other factors but it could well be impossible to actually quantify this. It can be applied to both the cases where the random variable itself is missing or depends on some parameter(s) which we do not know about.

Expectation maximation allows us to iteratively maximise the likliehood without having to have an explicit parameterisation of the parameters therein. Suppose we have some true underlying statistcal model which contains a set of observed data in random variable(s), X, with a set of missing (latent) randomn variables, Y. If our model depends on a set of parameters, $\vec{\theta}$, then we know how to write down the likelihood function

$$L(\vec{\theta}; X, Y) = \prod_i^N p(X_i, Y_i|\vec{\theta}). \tag{4.5}$$

However, in this case we cannot know the probability distribution, $p(X, Y|\vec{\theta})$, because it contains the random variable(s) Y we do not know about. We could instead decide to consider the *marginal likelihood* integrating out the Y random variables, which would be given by

$$L(\vec{\theta}; X) = \prod_i^N p(X_i|\vec{\theta}) = \prod_i^N \int p(X_i, Y|\vec{\theta}) dY = \prod_i^N \int p(X_i|Y, \vec{\theta}) p(Y|\vec{\theta}) dY. \tag{4.6}$$

This would also give a valid maximum likelihood estimate of $\vec{\theta}$ but again we do not know *a priori* the probability distribution for Y, $p(Y|\theta)$.

The expectation maximisation algortihm allows us to find the maximum likelihood estimate of the marginal likelihood, (4.6), using an iterative procedure. We first assume some

distribution for the Y random variables and take an initial guess of the parameters $\vec{\theta}_{\mathrm{T}}$ [1] then proceed to iteratively

(i) **Expectation Step:** determine the *expectation* of the log likelihood given the values $\vec{\theta}_{\mathrm{T}}$

$$T(\vec{\theta}|\vec{\theta}_{\mathrm{T}}) = E\left[\log L(X|Y,\vec{\theta})\right], \tag{4.7}$$

where we assume that Y are distributed according to our initial guess.

(ii) **Maximisation Step:** Find a new set of parameters $\vec{\theta}_{\mathrm{T}}$ which maximise the expectation, $T(\vec{\theta}|\vec{\theta}_{\mathrm{T}})$.

This iteration procedure then repeats until some user defined cut-off point, normally that the determined values $\vec{\theta}_{\mathrm{T}}$ are not significantly changing between iterations, *i.e.* until convergence.

More simply put you can thing of the procedure as

(i) **Expect:** Estimate the expected value for the hidden variable,

(ii) **Maximise:** Optimise the parameter using the ML.

Problem for the classes is the biased coin flip example?

We will take a look at a nice expectation maximisation problem in a moment but let's first talk about expectation maximisation in the explicit case of Gaussian mixture models which we will then make use of for our example.

4.6 Gaussian mixture models

Gaussian mixture models (GMMs) are exactly what they say on the tin. They are probability distributions that are built from a mixture or a (finite) number of Gaussian distributions with unknown parameters. If you think about it this is a very reasonble approximation to almost all kinds of datasets we might encounter. Because in the high-statistics limit things tend to Gaussians and if we observe multiple things going on we can consider this as just a sum of Gaussians.

These are particularly useful for classification algorithms in which we may have an observed datasets that contains an overlap of different types of events (which we want to classify). We only see the entire dataset but can infer from that data that there are multiple different categories of events consistent with having originated from different Gaussians. We can infer Gaussian mixture models from datasets using expectation maximisation and the performance is excellent.

So let's just jump right in with an example which will demonstrate expectation maximation and Gaussian mixture models. Let's suppose I have dataset which is generated from three disjoint two-dimensional Gaussians as shown in the plot below, Fig. 4.7. For example it might be that I have some seismology data but I cannot tell if it comes from the same, two, three or four different underground volcanos. By performing an expectation maximisation I can produce a Gaussian mixture model for the different scenarios and then compare them. The example dataset in question (which as normal I have generated myself) is shown below in Fig. 4.7.

[1] Remember that nearly all optimisation algorithms need an initial guess anyway.

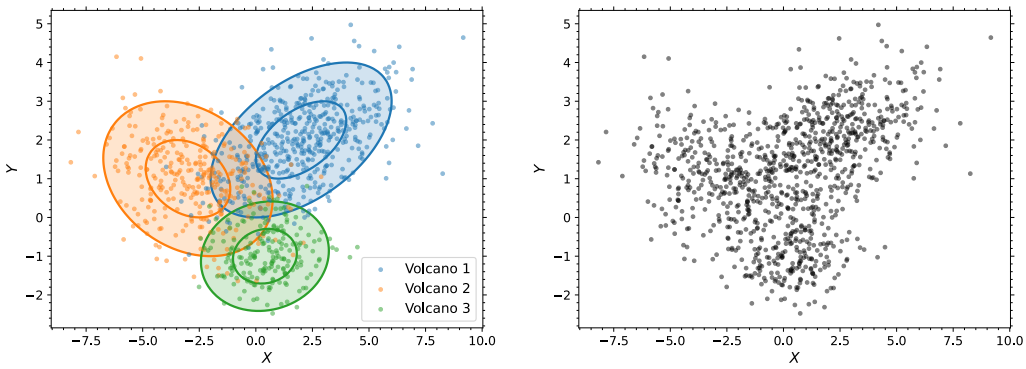

Figure 4.7: The data generated for the Gaussian mixture model example. Left: the 3 coloured contours show the "true" distributions for which the data is generated. The different coloured points show the data generated from the three distinct Gaussians. Right: is the same data but all with a single colour. This represents what we would actually start with in real life and what we perform the expectation maximastion on.

The next set of figures, Fig. 4.8, demonstrate what happens when I perform an expectation maximastion after several different iterations. We have to make an intial guess for the locations of the Gaussians, in this case I put some dummy values, but you can make use of the `scikit-learn` implementation of `GaussianMixture` which uses a `k-means` algorithm to initialise the starting locations.

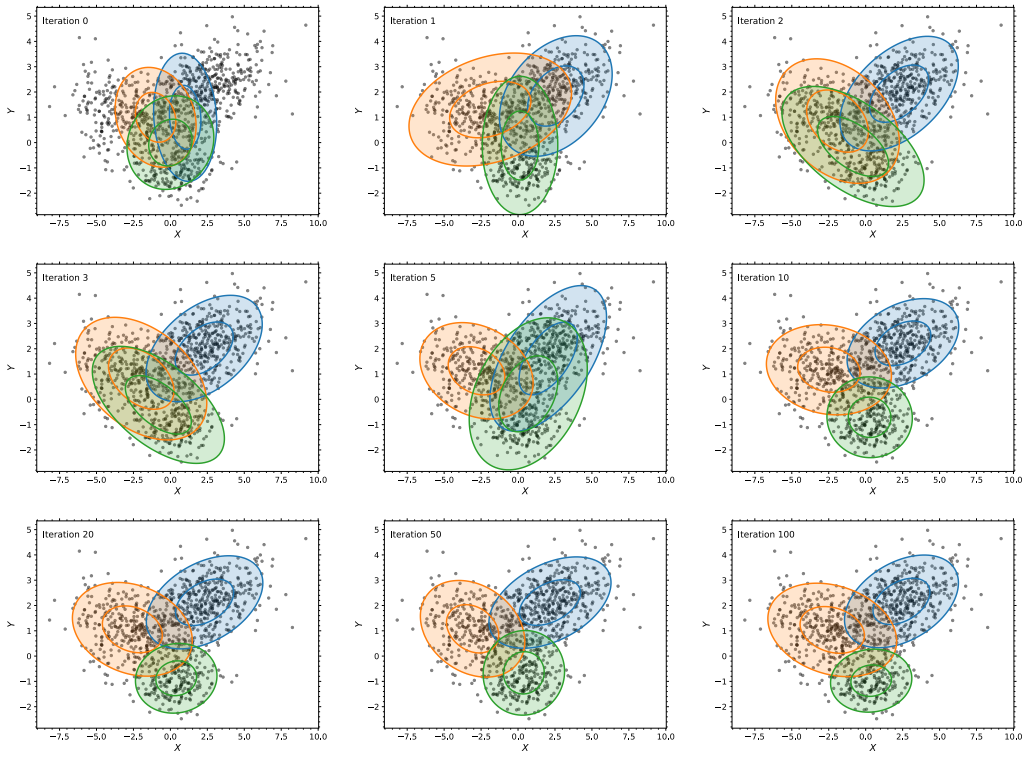

Figure 4.8: A demonstration of the progression of the expectation maximation algorithm in the context of a Gaussian mixture model over various different steps of the iteration. From top-left to bottom-right, at the iteration number 0, 1, 2, 3, 5, 10, 20, 50 and 100.

The final result of the full Gaussian mixture model, after 100 iterations, is shown below in Fig. 4.9, compared to the initial truth model. We can see the truly impressive performance of the expectation maximisation algorithm, especially in the context of Gaussian mixture models. A dataset which (on the right plot of Fig. 4.7) looks seeminly rather randomly can be very nicely fit, with little *a priori* knowledge, with three two-dimensional Gaussians.

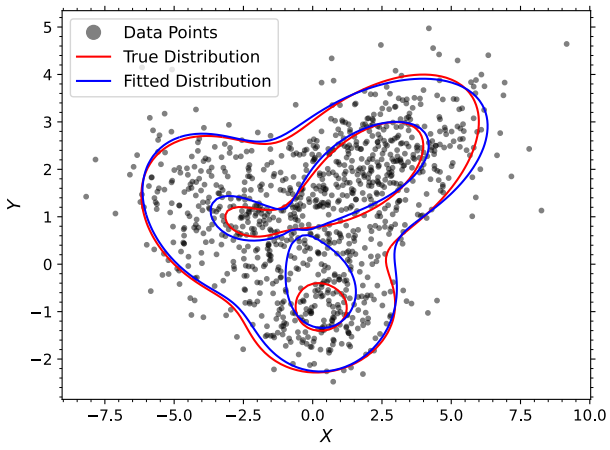

Figure 4.9: A comparison between the truth model (red) and the fitted Gaussian mixture model obtained using the expecation maximisation procedure (blue).