S1: Principles of Data Science

Problem Sheet 4 Solutions

MPhil in Data Intensive Science

Matt Kenzie
mk652@cam.ac.uk

Michealmas Term 2023

Problem Sheet 4

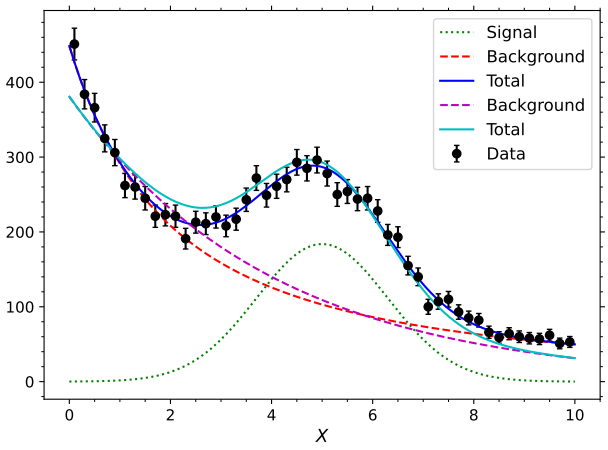
1. I already showed some of this in the lectures. A simple demonstration I put in `code/bootstrapping.py` and show below (with output) at the end.

```python
import numpy as np
np.random.seed(1)
from resample import jackknife, bootstrap

x = np.random.normal(1, 5, size=200)

# biased standard deviation
sdev = np.std( x )

# unbiased standard deviation
sdevc = np.std( x, ddof=1 )

# jackknife
jk_bias = jackknife.bias( np.std, x )
jk_std = jackknife.bias_corrected( np.std, x )

# bootstrap
bt_bias = bootstrap.bias( np.std, x, size=5000 )
bt_std = bootstrap.bias_corrected( np.std, x, size=5000 )

print('Estimate of variance without Bessel Correction: ', sdev )
print('Estimate of variance with Bessel Correction:    ', sdevc )
print('Estiamate of bias:')
print('   jackknife:', jk_bias )
print('   bootstrap:', bt_bias )
print('Bias-corrected estimate:')
print('   jackknife:', jk_std )
print('   bootstrap:', bt_std )
```

The output this produces is:

```
Estimate of variance without Bessel Correction:  4.550229914869617
Estimate of variance with Bessel Correction:     4.5616483265867425
Estiamate of bias:
   jackknife: -0.017533285542813637
   bootstrap: -0.017330823342105006
Bias-corrected estimate:
   jackknife: 4.567763200412401
   bootstrap: 4.567738504663221
```

2. It may be possible to do this analytically. I didn't bother with that. Take a look at the code in `code/bias_coverage.py` in which I investigate with some simulated samples. For reference my true distributions look like the following (with a single high-statistics toy thrown).



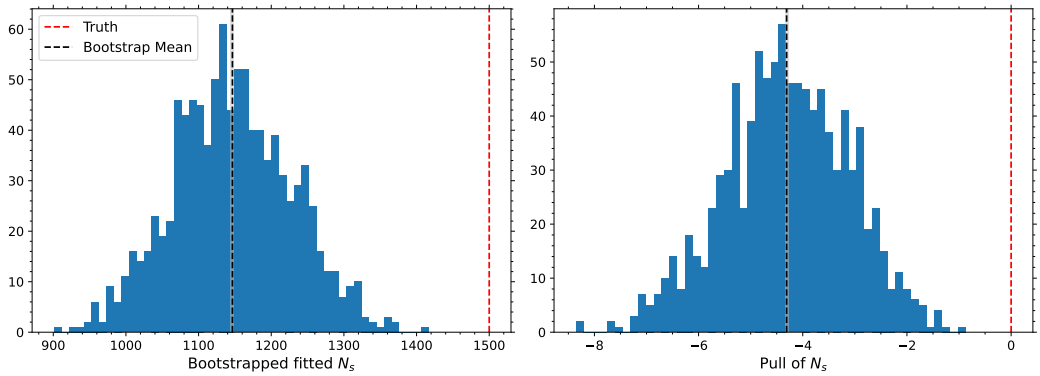
In my example I generate from a "Pareto" distribution of the form

$$f(X, b) = \frac{b}{X^{b+1}} \tag{1}$$

and fit it back with an exponential of the form

$$f(X, \lambda) = \lambda e^{-\lambda X}. \tag{2}$$

When fitting a thousand toys back I see a significant bias in the extracted signal yield, which consquently has a massive effect on the coverage (the true value is never in the interval because the true value is so far away). If the procedure were unbiased then the coverage is still off by a good way. Perhaps you can think about why this is? Here are a couple of plots showing the offset of the fitted values and the pull over an ensemble of bootstrapped samples



The script prints the following output

```
Absolute bias: -353.43967609015317
Percentage bias: -0.23562645072676877
Pull Mean:  -4.31 +/- 0.04
Pull Width: 1.20 +/- 0.03
True coverage of 68.3% error: 0.001 (but because biased)
Coverage even it were unbiased: 0.578 +/- 0.016 (6.7 sigma off)
```

3. The code for this can be found in `code/kl_toy.py`

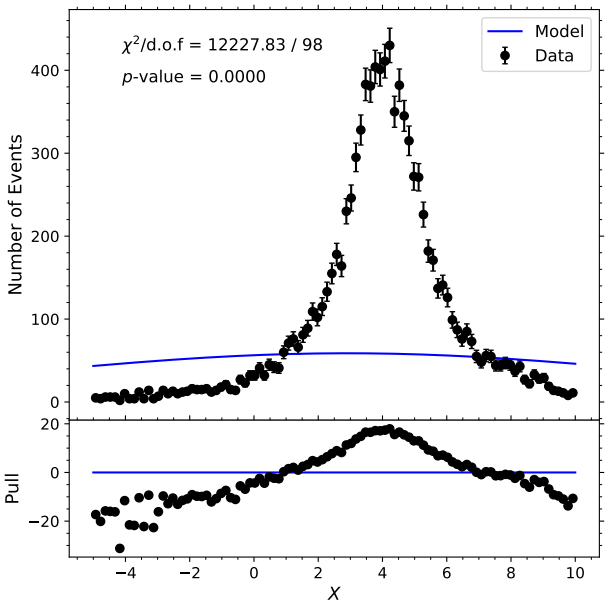
   (a) I find 0.0062

   (b) I find 0.0075

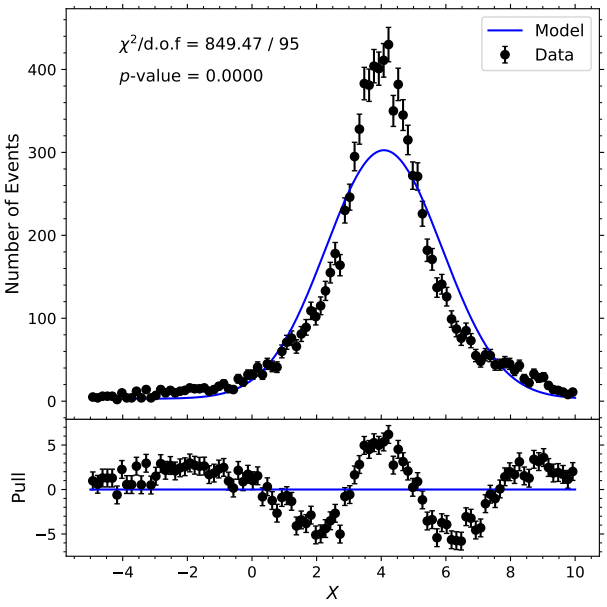
   (c) I find $\hat{f} = 0.102 \pm 0.005$ *i.e.* significantly non-zero. Conclusion is that there is a small but significant number of two pion decays of the K_L. This was the first ever observation of *CP* violation (a violation of the symmetry between matter and antimatter)

4. The code for this can be found in `code/kde.py`

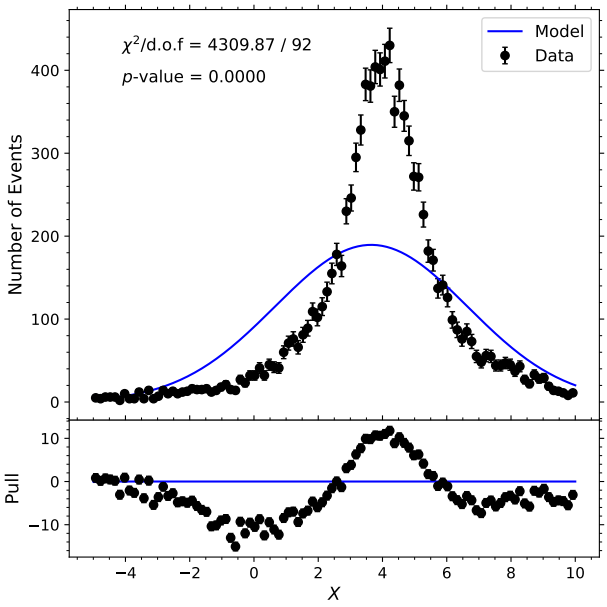
   (a) My fit to a normal distribution gives, $\hat{\mu} = 2.9 \pm 0.1$, $\hat{\sigma} = 10.18 \pm 0.07$. The $\chi^2/\text{d.o.f} = 12227.73/98$ which is clearly dog shit. Thus the conclusion is that this is not normally distributed.



(b) Doesn't really improve with two Gaussians and the fit has to be coerced into converging at all. For this the $\chi^2/\text{d.o.f} = 849.47/95$. Still awful.



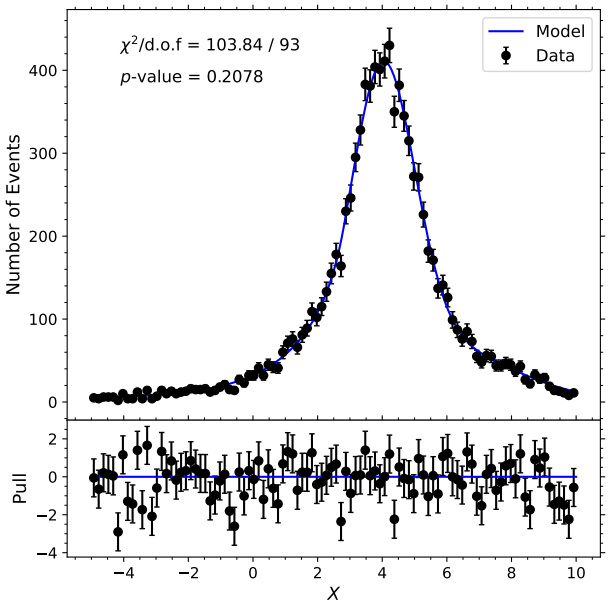
(c) In my case it actually gets worse with three Gaussians as my fit goes a bit haywire trying to converge. This result is also nonsense $\chi^2/\text{d.o.f} = 4309.87/92$.



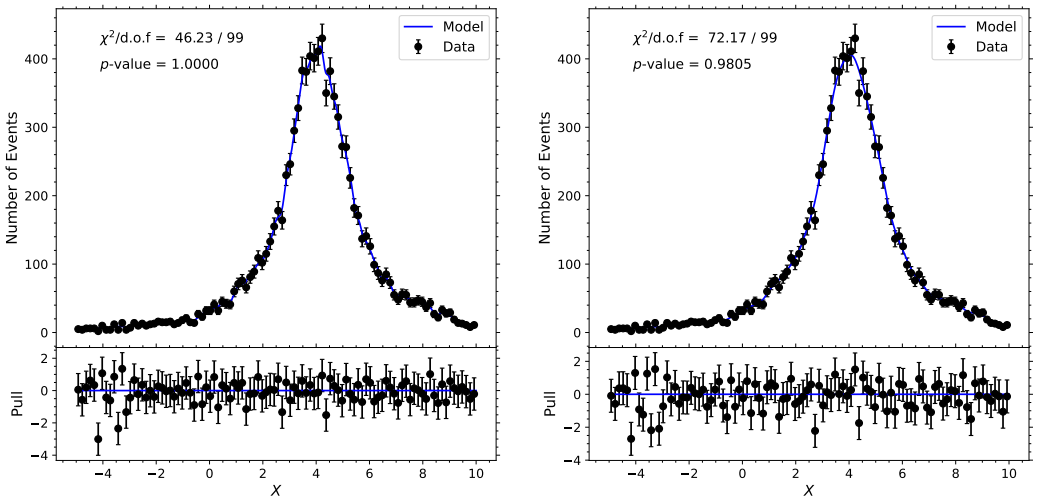
I can find an emprical distribution which does fit but that's only because I know how the sample was generated, namely from a sum of CrystalBall and a Gaussian. When I fit back with this I find:

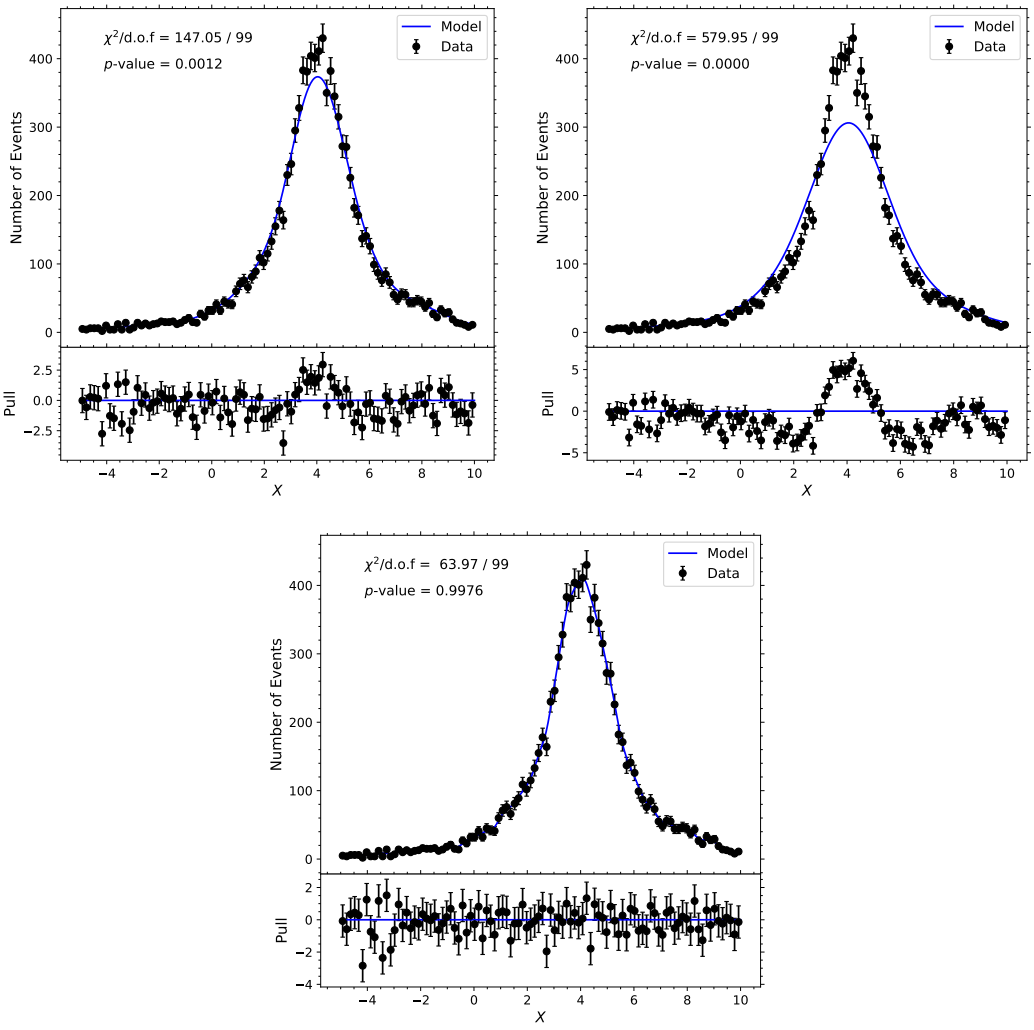
Parameter	Estimate
$\hat{f}$	0.68 ± 0.02
$\hat{\beta}_{CB}$	1.21 ± 0.04
$\hat{m}_{CB}$	1.95 ± 0.05
$\hat{\mu}_{CB}$	4.01 ± 0.02
$\hat{\sigma}_{CB}$	0.98 ± 0.03
$\hat{\mu}_G$	5.1 ± 0.1
$\hat{\sigma}_G$	2.56 ± 0.06

with a $\chi^2/\text{d.o.f} = 103.84/93$, which gives a p-value of $p = 0.21$. Perfectly reasonable.



(d) Below I put the KDE plots (top two are $h = 0.1$ and $h = 0.2$, second row are $h = 0.5$ and $h = 1$, third row is $h = 0.16$ from the "Scott" algorithm. It seems bandwidths of 0.5 and 1 are too wide and we see a bias. Bandwidths of 0.1 and 0.2 both seem reasonable. The Scott algorithm suggests a bandwidth of ~ 0.16. Notice that the χ^2/d.o.f $= 63.97/99$ comes out a little bit too good here so I would argue somewhere around 0.2 is perhaps a bit better.





5. Expectation maximisation on a biased coin flip. My code for this lives in `code/biased_coin_flip.py`. Going to start with an initial guess that the probability to get heads for each coin are $\theta_T = (0.4, 0.6)$. For each set of trials I can work work out the posterior, i.e. the probability that a given set was obtained from either coin A or coin B based on the intial values θ_T. If a given set gives me k heads and $n-k$ tails (where n is the number of flips in a set, in this case $n = 10$) then the likelihood can be written as

$$p(X_i|Z_j) = \frac{n!}{k!(n-k)!}\theta_{T,j}^{k}(1-\theta_{T,j}^{n-k}. \tag{3}$$

So then the posterior is

$$p(Z_j|X_i) = \frac{p(X_i|Z_j)p(Z_j)}{\sum_j p(X_i|Z_j)p(Z_j)}. \tag{4}$$

In this case we are told the coin is picked at random for each trial so we actually know the prior, that $p(Z_A) = p(Z_B) = 0.5$.

A further simplification can be found in this case because in the posterior ratio the factorial parts of the binomial distribution also cancel, thus the posteriors can be more

straightforwardly written as:

$$p(Z_j|X_i) = \frac{\theta_j^k(1-\theta_j)^{n-k}}{\sum_j \theta_j^k(1-\theta_j)^{n-k}}. \tag{5}$$

Recall from the lectures that we now compute the posterior based on some initial guess θ_T and we would then maximise the expectation of the log likelihood, which is the posterior multiplied by log of the "conditional" and "prior". This example is relatively straightforward to do analytically. Using the posterior we can compute the "expected" number of heads and tails in a given set for each coin by multiplying the posterior by the number of heads in that set.

We then want to maximise the expected log of the likelihood to update our initial guess. Again in the coin toss case this is easy. If I knew which coin was used for a given toss, my maximum likelihood estimate of the bias would simply be the number of heads over the numbers of tosses, $\theta = N_H/N$. Of course in this case I don't know what coin is flipped but I have now computed a probability (the posterior) that a given set used coin A or coin B and I also know how many head I would expect for each coin. Thus my maximum likelihood estimate simply becomes, $\hat{\theta_j} = E[N_{H,j}]/E[N_j]$.

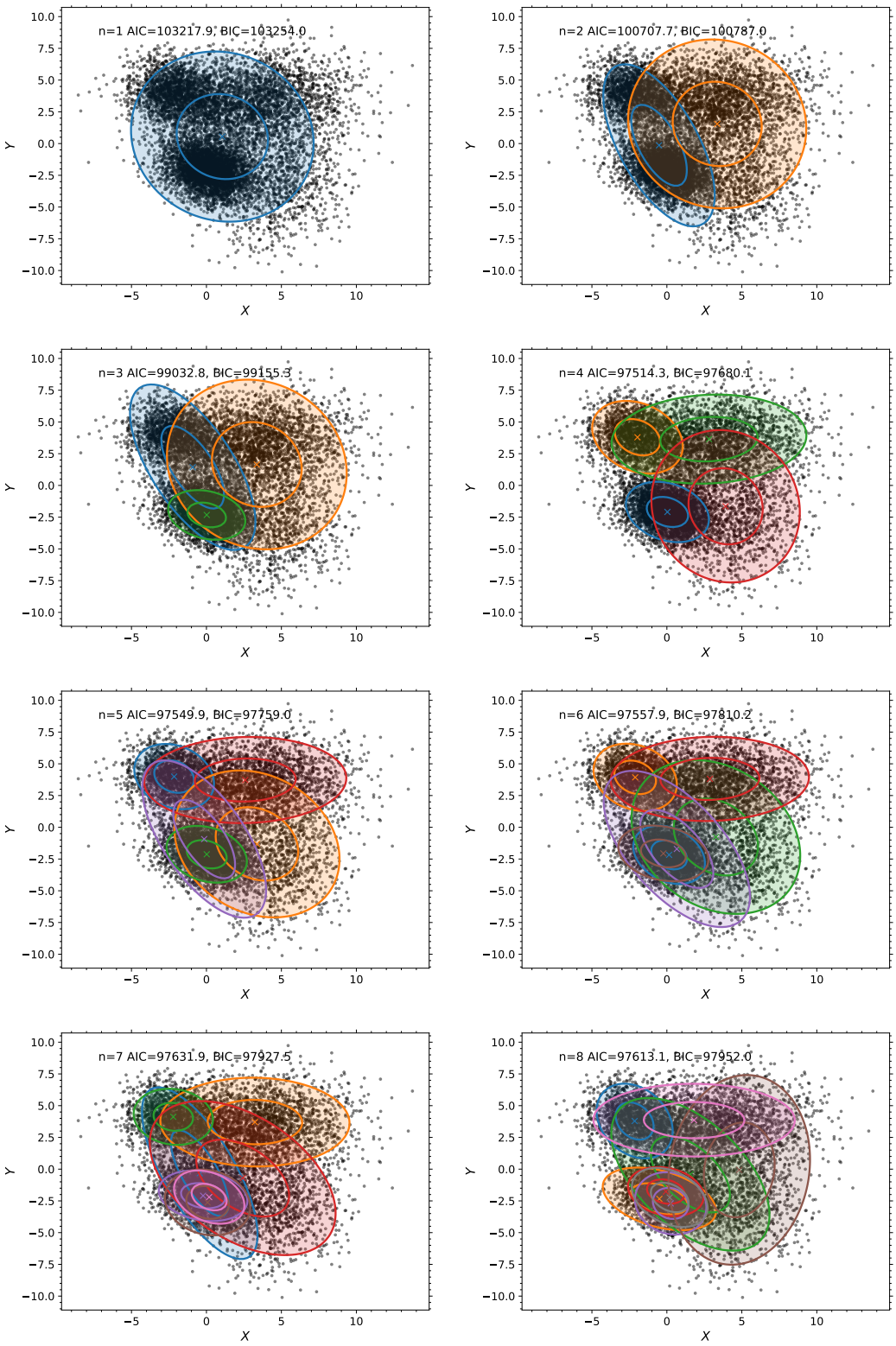
For an initial guess of $\theta_A = 0.4$ and $\theta_B = 0.6$ I can then write out these different values:

Set	Outcomes	Posterior for A	Posterior for B	Expected Heads for A	Expected Heads for B
1	THHHHHHHTH	0.08	0.92	0.65	7.35
2	TTTTTTHTHT	0.92	0.08	1.84	0.16
3	HTTHHHHHHH	0.08	0.92	0.65	7.35
4	TTHTTHHHTH	0.50	0.50	2.50	2.50
5	HTHHTTHHHH	0.16	0.84	1.15	5.85

which would mean I estimate $\hat{\theta}_A = 0.389$ and $\hat{\theta}_B = 0.713$.

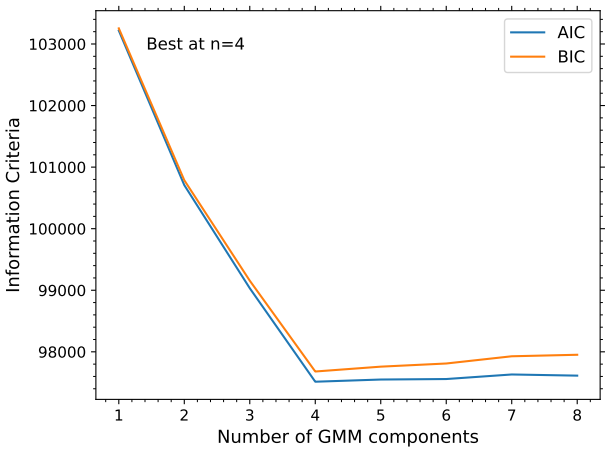
Running this over 8 iterations stops the estimates changing by more than 0.1% and find $\hat{\theta}_A = 0.347$ and $\hat{\theta}_B = 0.742$. For reference the true biases which were generated were $\theta_A = 0.32$ and $\theta_B = 0.64$.

6. The sample is generated from a mixture four two-dimensional Gaussians. My solution is provided in `code/gmm.py`. I try GMMs using the `sklearn.mixture.GaussianMixture` class with components ranging from 1-8. I put the plots for these different configurations below.

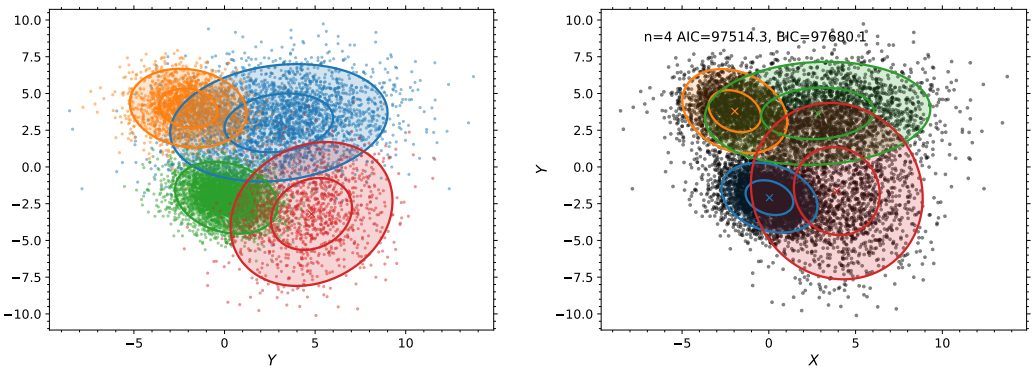


I then also keep track of the AIC (Aikaike Information Criterion) and BIC (Bayesian Information Criterion) score for each GMM and chose the one with the minimum BIC score. In this case this suggests a mixture of four (which is what was generated with). A plot of the AIC and BIC as a function of the number of mixture components is

shown below.

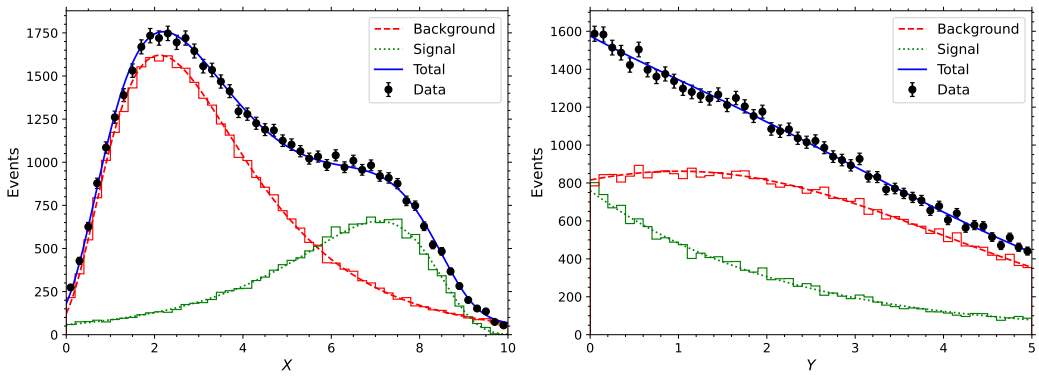


For completeness below I show a comparison between the truth (generated model with data coloured by their component), on the left, next to the GMM with 4 components, on the right.



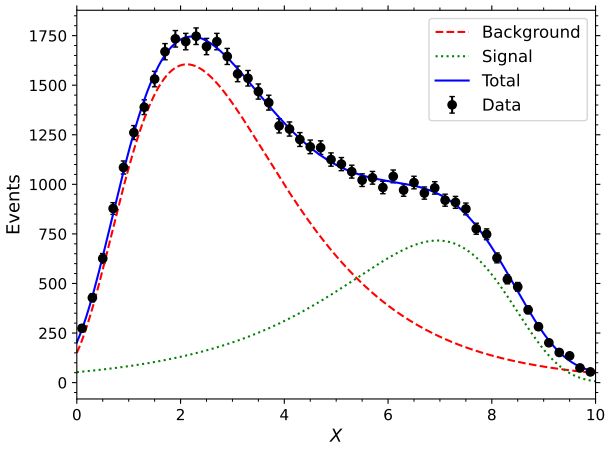
The extracted parameter values (mixture fractions, π_j, means and covariances of each component) are printed by `code/gmm.py`.

7. The code for this can be found in `code/sweight.py`. Note you will need to have installed the `python sweights` package. For reference and my own sanity the following shows the true underlying distributions and the dataset that I generated from them (this is the data stored in `datasets/sweights.npy`.

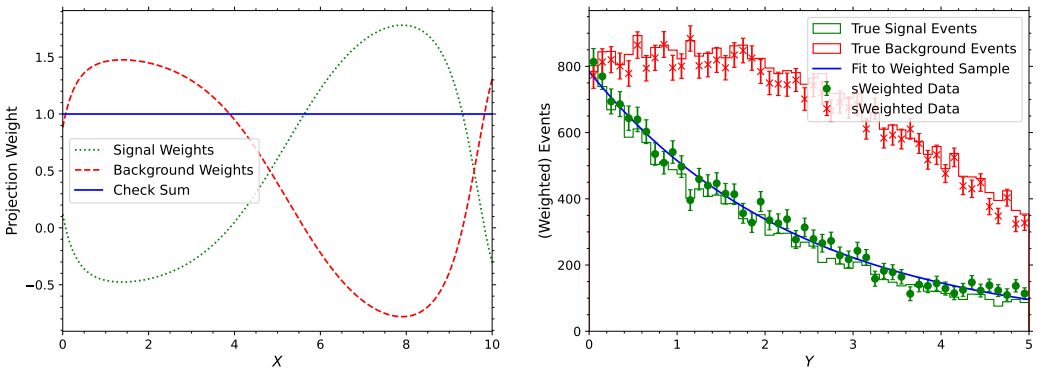


(a) I set up the model as suggested in the question and then run a maximum likelihood fit to the "discriminant" dimension, X. That gives me the best fit values and plot shown below.

Parameter	Estimate
$\hat{N}$	$(50.0 \pm 0.2) \times 10^3$
$\hat{f}$	0.33 ± 0.05
$\hat{c}$	0.6 ± 0.2
$\hat{\mu_s}$	7.7 ± 0.3
$\hat{\sigma_s}$	1.2 ± 0.1
$\hat{s}$	0.46 ± 0.03
$\hat{\mu_b}$	-1.3 ± 0.2
$\hat{\sigma_b}$	4.2 ± 0.3



(b) Now I run the `sweights` package to get the weight functions. The two plots below show these weight functions for the signal and background (left) and then the projection of the data when these respective weight functions are applied (right plot, points with errors) compared to the data for which the underyling relevent component is known (right plot, histogram).



In order to determine the slope parameter of the sWeighted (red points) events I can simply fit that distribution with an exponential (or use any other estimation method I so desire). That is what gives me the $\hat{\lambda} = 2.38 \pm 0.03$.