S1 Principles of Data Science Coursework Report

CRSiD: tmb76

University of Cambridge

Section A

(a)

We have the continuous random variable $M \in [5; 5.6]$. Our model is the weighted sum of a background and signal such that:

$$p(M; f, \lambda, \mu, \sigma) = fs(M; \mu, \sigma) + (1 - f)b(M; \lambda) \tag{1}$$

where:

$$s(M; \mu, \sigma) = \tfrac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(M-\mu)^2}{2\sigma^2}}$$
$$b(M; \lambda) = \lambda e^{-\lambda M}$$

We want to show that:

$$I = \int_{-\infty}^{+\infty} p(M; f, \lambda, \mu, \sigma)\, dM = 1 \tag{2}$$

We have:

$$I = \int_{-\infty}^{+\infty} fs(M; \mu, \sigma) + (1 - f)b(M; \lambda), dM$$
$$I = \int_{-\infty}^{+\infty} f \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(M-\mu)^2}{2\sigma^2}}\, dM + \int_{0}^{+\infty} (1 - f)\lambda e^{-\lambda M}\, dM$$

Since the exponential decay distribution is only defined from 0 to $+\infty$. We can first evaluate the 2nd integral:

$$\int_{0}^{+\infty} (1 - f)\lambda e^{-\lambda M}\, dM = (1 - f)[-e^{-\lambda M}]_0^{\infty} = (1 - f)[0 - (-1)] = (1 - f)$$

Then, we evaluate the integral of the signal part of the model, taking the f weight out. We use a change of variable so that $u = \frac{(M-\mu)}{\sigma} \iff du = \sigma dM$, thus:

$$J = \int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(M-\mu)^2}{2\sigma^2}}\, dM = \int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{(u)^2}{2}}\, du$$

We can square multiply this integral by itself, using dummy variables x and y:

$$J^2 = \int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{(x)^2}{2}}\, dx \times \int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{(y)^2}{2}}\, dy$$

$$J^2 = \tfrac{1}{2\pi} \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} e^{-\frac{(x)^2}{2}} e^{-\frac{(y)^2}{2}} \, dx \, dy$$
$$J^2 = \tfrac{1}{2\pi} \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} e^{-\frac{1}{2}(x^2+y^2)} \, dx \, dy$$

From here, we can switch to polar coordinates to be able to evaluate this. We have $x = r\cos(\theta) y = r\sin(\theta)$. This means we need to change the limits to polar equivalents. We will get $r \in [0, \infty]$ and $\theta \in [0, 2\pi]$. Further, $dxdy = rdrd\theta$:

$$J^2 = \tfrac{1}{2\pi} \int_{0}^{2\pi} \int_{0}^{+\infty} e^{-\frac{1}{2}r^2} r \, dr \, d\theta$$
$$J^2 = \tfrac{1}{2\pi} \int_{0}^{2\pi} \left[-e^{-\frac{1}{2}r^2}\right]_0^\infty d\theta$$
$$J^2 = \tfrac{1}{2\pi} \int_{0}^{2\pi} [-0 - (-1)] \, d\theta$$
$$J^2 = \frac{1}{2\pi} \int_{0}^{2\pi} 1 \, d\theta$$
$$J^2 = \frac{1}{2\pi} [\theta]_0^{2\pi}$$
$$J^2 = \frac{1}{2\pi} [2\pi - 0]$$
$$J^2 = 1$$

This then means that $J = \pm 1$ but since $s(M; \mu, \sigma) > 0 \forall M \in [-\infty, \infty]$, $J = 1$. Thus: $I = (1 - f) + f = 1$.

(b)

We want to find an expression of $p(M; \vec{\theta})$ such that it is normalised between α and β. Now because the signal fraction f is such that:

$$\frac{s(M; \mu, \sigma)}{b(M; \lambda)} = \frac{f}{1 - f} \tag{3}$$

This means that we need to normalise the signal and background components separately. This gives:

$$\int_\alpha^\beta f N_s \times s(M; \mu, \sigma) + (1 - f) N_b \times b(M; \lambda) \, dM = 1 \tag{4}$$

with:

$$\frac{1}{N_s} = \int_{\alpha}^{\beta} s(M;\mu,\sigma)\, dM \tag{5}$$

$$\frac{1}{N_b} = \int_{\alpha}^{\beta} b(M;\lambda)\, dM. \tag{6}$$

where N_s and N_b are a function of parameters $\vec{\theta}$. Now, from the definition of the Cumulative Distribution Function (c.d.f.): $F(X') = \int_{-\inf}^{X'} f(X)\, dX$, where X' is a specific value of the random variable X, and $f(X)$ is the p.d.f.(The lower limit is 0 for the exponential decay distribution) [13, pp. 20-24]. This means we can write (5) and (6) as:

$$\frac{1}{N_s} = \int_{\alpha}^{\beta} s(M;\mu,\sigma)\, dM = F_s(\beta) - F_s(\alpha) \tag{7}$$

$$\frac{1}{N_b} = \int_{\alpha}^{\beta} b(M;\lambda)\, dM = F_b(\beta) - F_b(\alpha) \tag{8}$$

We know that the normal and exponential decay distributions have:

$$Normal : F(X) = \frac{1}{2}[1 + erf(\frac{X-\mu}{\sigma\sqrt{2}})] \tag{9}$$

$$Exponential\ Decay : F(X) = 1 - e^{-\lambda X} \tag{10}$$

From equation (7) and (8), we can write the normalisation factors as:

$$\frac{1}{N_s} = \frac{1}{2}[1 + erf(\frac{\beta-\mu}{\sigma\sqrt{2}})] - \frac{1}{2}[1 + erf(\frac{\alpha-\mu}{\sigma\sqrt{2}})] = \frac{1}{2}[erf(\frac{\beta-\mu}{\sigma\sqrt{2}}) - erf(\frac{\alpha-\mu}{\sigma\sqrt{2}})] \tag{11}$$

$$\frac{1}{N_b} = (1 - e^{-\lambda\beta}) - (1 - e^{-\lambda\alpha}) = e^{-\lambda\alpha} - e^{-\lambda\beta} \tag{12}$$

Finally, this gives the total p.d.f. of the model as:

$$p(M;\vec{\theta}) = \frac{f}{2}[erf(\frac{\beta-\mu}{\sigma\sqrt{2}}) - erf(\frac{\alpha-\mu}{\sigma\sqrt{2}})] \times s(M;\mu,\sigma) + (1-f)[e^{-\lambda\alpha} - e^{-\lambda\beta}] \times b(M;\lambda) \tag{13}$$

(c)

In this question, we check that the integral of the p.d.f. between α and β does equal unity. For this specific case, the `scipy.integrate` library [11] is used to numerically integrate the component-wise normalised p.d.f. as described in equation (4). Though N_s and N_b, are computed using the `scipy.integrate` library [11], like in (5) and (6). The components are computed using `scipy.stats` library's `norm.pdf` and `expon.pdf` methods [12] (cf. `funcs.py` file, `pdf_norm` function). Then, the weighted sum of the two is computed. Finally, that p.d.f. is integrated from α to β, with randomly generated $\vec{\theta}$ parameters, using `random.uniform` [1]. The results are shown in the table below:

μ	σ	λ	f	$Integral$
5.2357	0.0190	0.3643	0.5044	1.0
5.5436	0.0105	0.3823	0.6081	1.0
5.3184	0.0214	0.6424	0.6882	1.0
5.3621	0.0221	0.3695	0.3625	1.0
5.3640	0.0231	0.5015	0.2442	1.0

Table 1: Results for different values of $\vec{\theta}$ parameters. (cf. `solve_part_c.py` file)

(d)

For this question, the true values of the parameters were set and used to plot the p.d.f. of the model, along with the signal and background component overlayed. The parameters are assumed to be:

$$\mu = 5.28;\ \sigma = 0.018;\ \lambda = 0.5;\ f = 0.1$$

The `signal_norm`, `background_norm`, and `pdf_norm` functions, described in (c) were used to compute the p.d.f. of the model, signal, and background, normalised for $M \in [5, 5.6]$. The results are shown in the figure below:

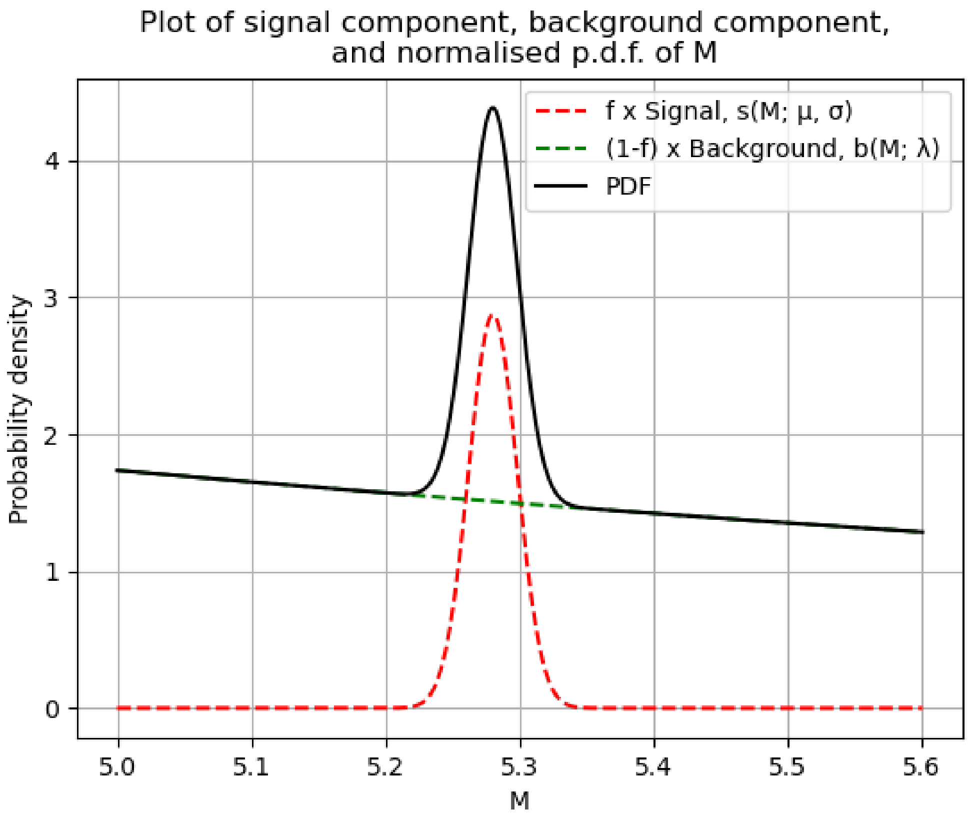

Figure 1: Plot of the total p.d.f. of the model, scaled signal and background, normalised for $M \in [5, 5.6]$.

This shows the impact of the f weight, which considerably reduces the size of the signal component in the total p.d.f..

(e)

The "true" p.d.f., signal and background component are now known and have been visualised. The context is now to generate a dataset from this "true" p.d.f., and to assume this is an observed sample obtained in an experiment. From this sample, the goal is now to try and estimate these "true" parameters, using a chosen estimation method.

To generate the sample, an accept-reject method was used [14]. The accept-reject method requires us to know the p.d.f. function over the M domain, and the maximum value of the p.d.f. in the M domain, y_{max}. Uniformly random values of M between α and β are generated, using `random.uniform` [1]. Then, for each of these values, many uniform random values of y are generated between 0 and y_{max}. If these generated y's are such that $y \leq p(M;\vec{\theta})$ for the generated M then the **M value** is accepted and added to the sample. This is repeated until the sample size is reached (cf. `funcs.py` file, `accept_reject` function).

The chosen estimation method is the Maximum Likelihood Estimation (MLE) method [13, pp. 122-128]. This method relies on the likelihood function, which stands for the probability of observing the data given a set of parameters and is defined as:

$$\mathcal{L}(\vec{\theta}) = \prod_{i=1}^{N} p(M_i;\vec{\theta}) \tag{14}$$

It is a function of the parameters $\vec{\theta}$ only. The goal of the MLE method is to find estimates of the parameters $\vec{\theta}$ that maximise the likelihood function [13, pp. 122-128]. In other words, finding $\hat{\vec{\theta}}$, such that:

$$\frac{\delta\mathcal{L}(\hat{\vec{\theta}})}{\delta\theta} = 0 \tag{15}$$

This result is because maximising the probability of observing M given a certain set $\hat{\vec{\theta}}$ means we have found the set of parameters $\hat{\vec{\theta}}$ that are the most likely to be the ones the data was observed under. Now because summing small numbers is less of a problem as multiplying small numbers in computations, we take the logarithm of the likelihood function, to turn the product into a sum. And taking twice the negative of that gives a negative log-likelihood function:

$$-2\ln\mathcal{L}(\vec{\theta}) = -2\sum_{i=1}^{N} \ln p(M_i;\vec{\theta}) \tag{16}$$

And this is the function to now **minimise**. To do so we use a python package named `iminuit` [4], which possesses a very useful set of minimising tools. The `iminuit` package requires a function to minimise, called a cost function, and a set of initial guesses for the parameters. Luckily, `iminuit` possesses a series of standard cost functions [6]. The one used here is the `BinnedNLL`, Binned Negative Log-Likelihood, which takes in the c.d.f. of a **normalised** probability density and the observed **binned** data we want to fit that density to. With the cost function defined, `iminuit`'s main function `Minuit` [8], a function minimizer and error computer, takes that cost function in along with initial guesses for the parameters. These initial guesses are usually educated guesses at what the true parameters may be. Then we call the `migrad` method, which is a gradient descent method, to conduct the minimization [7].

An important point is that `iminuit.Minuit` reads the function given in the cost function definition to read the parameters it needs to fit for. In other words, using the `pdf_norm` function from (d), `Minuit.migrad()` would minimise the cost function for the $\vec{\theta}$ parameters as well as α and β [4].

Furthermore, in the interest of performance, the `numbastats` library [10] was used to speed up any computation that uses our new p.d.f. function. This library provides `numba`-accelerated statistical function for common probability distributions, mostly following `scipy.stats` library's conventions [12]. So the c.d.f. of the model was defined to only take the $\vec{\theta}$ parameters. It then uses the `numba_stats.truncnorm.cdf` and `.truncexpon.cdf` functions. Running the minimisation, the following parameter estimates and their uncertainties were obtained:

$$\hat{\mu} = 5.27939 \pm 0.00033;$$
$$\hat{\sigma} = 0.01820 \pm 0.0032;$$
$$\hat{\lambda} = 0.490 \pm 0.019;$$
$$\hat{f} = 0.1008 \pm 0.0017$$

We can see that the minimisation performed well, with close estimates and relatively small uncertainties. Those uncertainties were obtained using a Hessian matrix. By taking the inverse of the Hessian matirx, the matrix of 2^{nd} derivatives of the cost function for all free parameters, one can approximate the covariance matrix. Then, taking the diagonal elements gives an approximation of the variance of the estimates. Taking the squareroot gives the standard deviation. With those parameter estimates, the following plot of the data and the fitted model was obtained:

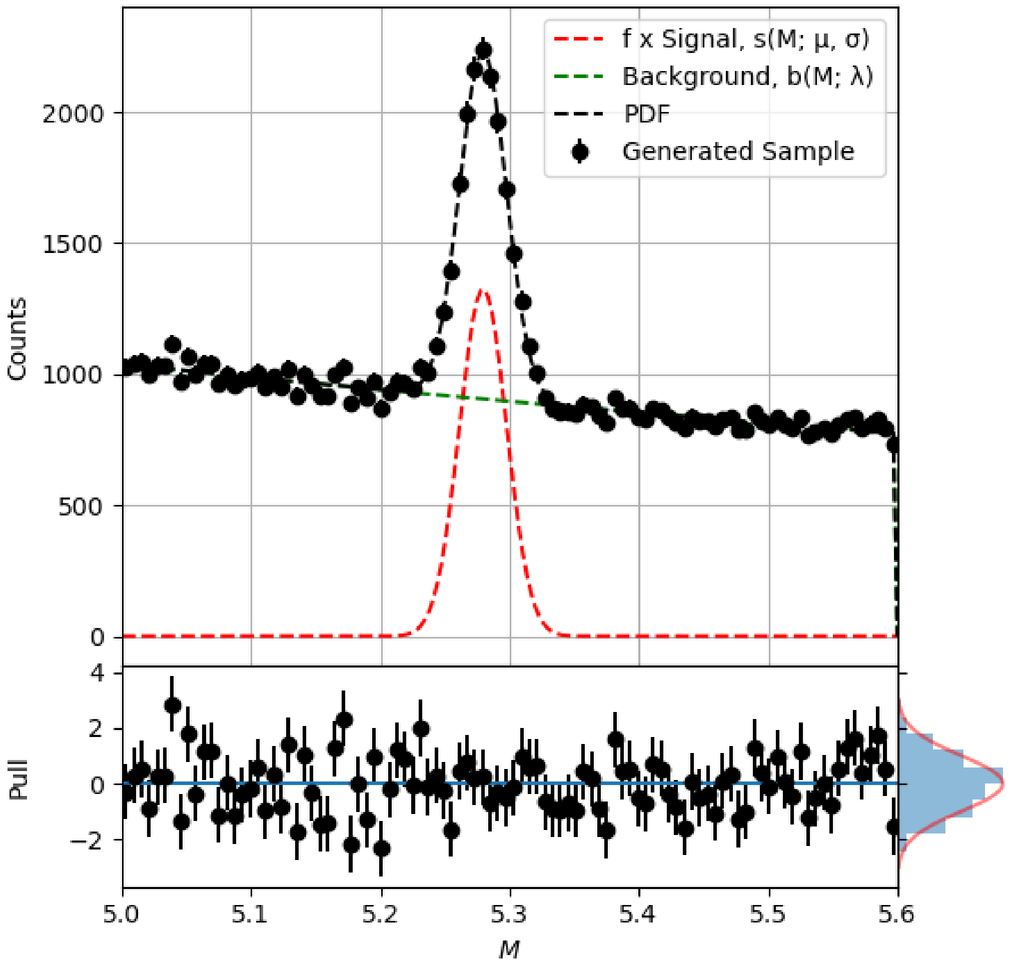

Figure 2: Plots of the binned data, with uncertainties defined by the Poisson distribution ($\sqrt{N}$) and the fitted signal, background, and total p.d.f. model from the estimated parameters (top). And of the pull values for each bin, with their distribution histogram on the right (bottom).

The lower plots are plots of the pull: $pull = \frac{obs-pred}{\sqrt{N}}$, where N is the bin counts. In other words, they are the errors weighted by the uncertainties. The pull plots are a good way to check that the uncertainties are sensible, and is also known as the student's t-distribution [2].

Section B

(f) and (g)

Introduction

In this section, the context is now the discovery of a signal in an observed dataset, similar to searching for a particle using a collider [9]. To classify the signal as discovered, a significance of at least 5 standard deviations of a normal distribution. Let's imagine there is a search for a particle going on and, obviously, data needs to be collected. However, the more data we need to collect, the more expensive the running of the experiment becomes.

The task at hand is to simulate the experiment numerically and find the smallest possible dataset size for which the signal is discovered at least 90% of the time. To do this, a hypothesis test can be conducted with the null hypothesis being that there is only background being observed in our data, and the alternate hypothesis being that there is one signal on top of the background [13, p. 164]. The second task is a similar one, but this time, the experiment is expected to reveal 2 signals in the data. Again, the goal is to find the smallest possible dataset size for which the 2 signals instead of 1 are discovered at least 90% of the time. This means the null hypothesis will be that there is only one signal on top of the background, and the alternate hypothesis will be that there are in fact 2 of them. In the context of hypthesis testing, the discovery threshold can be defined as the p-value with which we can make a decision regarding the null hypothesis [13, p. 164].

Method

For (f):

The general idea is to generate many pretend datsets for multiple sample sizes and conduct hypothesis testing on each of them to obtain a p-value, for which we can confidently reject the null hypothesis, i.e. at our discovery threshold (2.9×10^{-7}). First, a very large dataset (100,000 samples) is generated, from the same p.d.f. as in (e), with the same "true" parameters using the previously described accept-reject method. Sample sizes to test are generated as a base-10 logarithmic scale from 100 to 10000. For each sample size, 1000 datasets are generated by bootstrapping from the larger dataset.

Bootstrapping is a resampling method that consists of sampling with replacement from a dataset to create a new dataset of the same size, or smaller. When doing so from a model, it is called parametric bootstrapping, and can be referred to as "throwing toys" or generating pretend datasets. One other use is to estimate parameters and their uncertainties, by estimating them for each toy and taking the mean and standard deviation of the resulting distributions [13, p. 108].

In this simulation, for each bootstrapped sample, a hypothesis test is conducted. The test is set up with the null hypothesis being that there is only background being observed in our data, and the alternate hypothesis being that there is one signal on top of the background. In other words:

$$H_0 : p(M; \vec{\theta}) = b(M; \lambda)$$

$$H_1 : p(M; \vec{\theta}) = fs(M; \mu, \sigma) + (1 - f)b(M; \lambda)$$

It can be seen that the null hypothesis is simply the case where $f = 0$. Just like when determining the profiled log-likelihood, the parameter can be fixed, and the likelihood function minimised for all other parameters. f can be set to 0, and the negative log-likelihood minimised for μ, σ, and λ. This is possible to do when setting up `Minuit`, f can be fixed at 0, while all other parameters are allowed to float. Then, running `migrad` will minimise the cost function for all other parameters. Then, a second instance of `Minuit` is set up, where f is now allowed to float, just like in (e), and run `migrad` again [8] [7].

Doing so, 2 hypothesis are being fitted to the samples, and we can obtain the minimum negative log-likelihood values for each of those fits [5]. Now the hypothesis test relies on a chosen test statistic. The test statistic used here is the difference between the 2 minimised negative log-likelihood values under each hypothesis, or the negative log-likelihood ratio:

$$T = -2 \ln \frac{\mathcal{L}(f = 0)}{\mathcal{L}(f = \hat{f})} \tag{17}$$

The reason this test is used is backed up by the Neyman-Pearson Lemma, which asserts that a test statistic of this form has the strongest statistical power, for any test size. In other words, it maximises the critical region, under the test-statistic's distribution curve, whose area is the value for which a decision regarding the alternate hypothesis can be made [13, p. 170]. Now, from Wilk's theorem, this test-statistic will be approximately distributed as a χ^2 with number of degrees of freedom (DoF), the difference in the number of free parameters under the alternate and the null

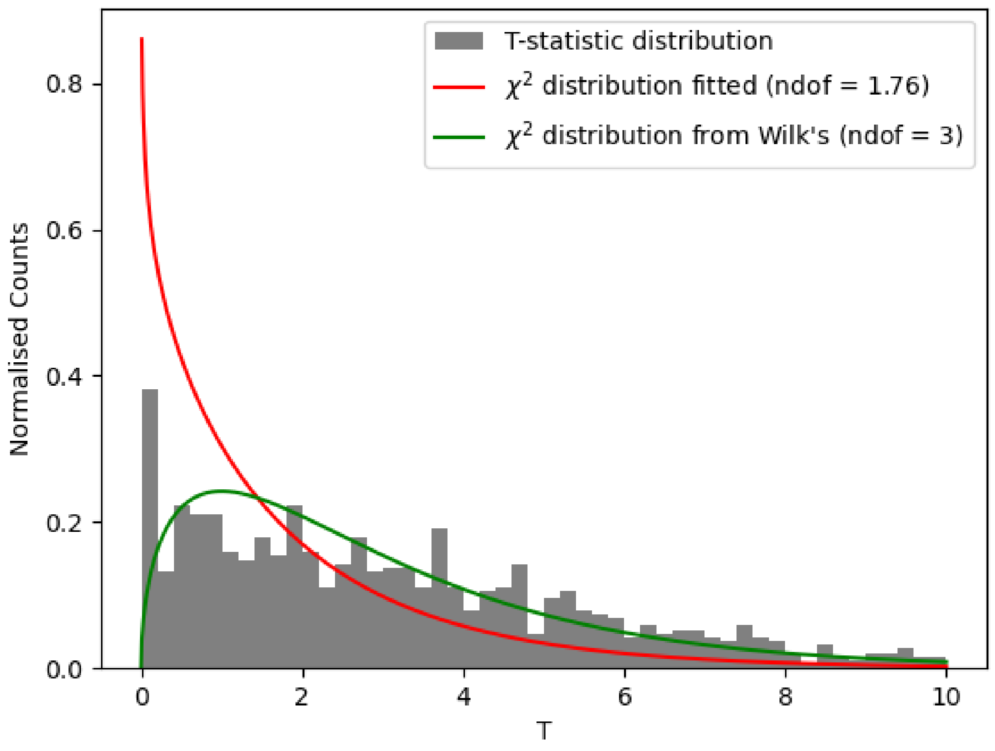

Figure 3: Plot of the fitted χ^2 distribution vs Wilk's theorem over the test-statistic histogram under the null hypothesis, for part (f).

hypotheses [15]. Wilk's theorem applies here because the hypothesis test has been set up so that the null is a subset case of the alternate [3].

For a more robust result, a side simulation was run, similar to (e), but with a dataset generated from the null hypothesis, i.e. $f = 0$. And bootstrapping from it, a distribution of the test-statistic under the null hypothesis was obtained. Using `iminuit`, and defining the cost function as the unbinned NLL of the χ^2 distribution p.d.f., we fit for the number of DoF parameter for that observed distribution [5]. For (f), we obtained a value of 1.76. This is different from Wilk's theorem's expected value of $4 - 1 = 3$ (cf. `ndof_for_art_f_g.py`).

Indeed, the fitted χ^2 distribution fits much worse than using the Wilk's theorem's expected value. Thus, we will use the Wilk's theorem's expected value of number of DoF for part (f). With the number of DoF, the p-value can be computed as 1 minus the area under the χ^2 distribution curve, from $-\inf$ to the test-statistic. This is done using the `scipy.stats.chi2.cdf` function, going back to the definition of the c.d.f. [13, pp. 20-23] [12]. Then, applying the discovery threshold, the number of discoveries is counted and the frequency of discovery is computed.

To then get the uncertainties of those discovery rates, samples are bootstrapped from the of discoveries, a list of 1's if there was a discovery, 0's otherwise. Using this, the discovery rate is computed for each of those toys. From the resulting distribution of discovery rates, the standard deviation is calculated. We thus obtain confidence intervals for the discovery rate.

For (g):

The same method is used, though the data is generated from a different p.d.f. and the hypotheses are appropriately changed. The p.d.f. used is the same as in (e), but with 2 signals on top of the background, each with their own f weight and mean μ, but with the same width σ (cf. `funcs.py` file, `pdf_norm_g`):

$$p(M;\vec{\theta}) = f_1 s(M;\mu_1,\sigma) + f_2 s(M;\mu_2,\sigma) + (1-f_1-f_2)b(M;\lambda) \tag{18}$$

The null hypothesis is now that there is only one signal on top of the background. In other words, the 2^{nd} signal's f_2 weight is set to 0, and μ_2 is technically allowed to float but it has no meaning in this context. The alternate hypothesis is that there are 2 signals on top of the background, and these are described by the following "true" parameters:

$$\mu_1 = 5.28;\ \mu_2 = 5.35;$$
$$\sigma = 0.018;\ \lambda = 0.5;$$
$$f_1 = 0.1;\ f_2 = 0.1$$

The same test statistic is used, and the same method is applied. The number of DoF was now found to be 2.13, closer to the Wilk's theorem's expected value of $6 - 4 = 2$.

In this case, the χ^2 distribution for the fitted number of DoF seems to better fit the data than for the Wilk's theorem's expected value, so we will use the fitted number of DoF for part (g).

Results

For (f):

The plot shows that the discovery rate increases with the sample size, as expected. The simulation stops after it founds 3 sample sizes that have discovery rates above 90%. Here, the minimum sample size appear to be around 800 samples, maybe 750. The error bars are set to represent 3 standard deviations from the mean. Here,

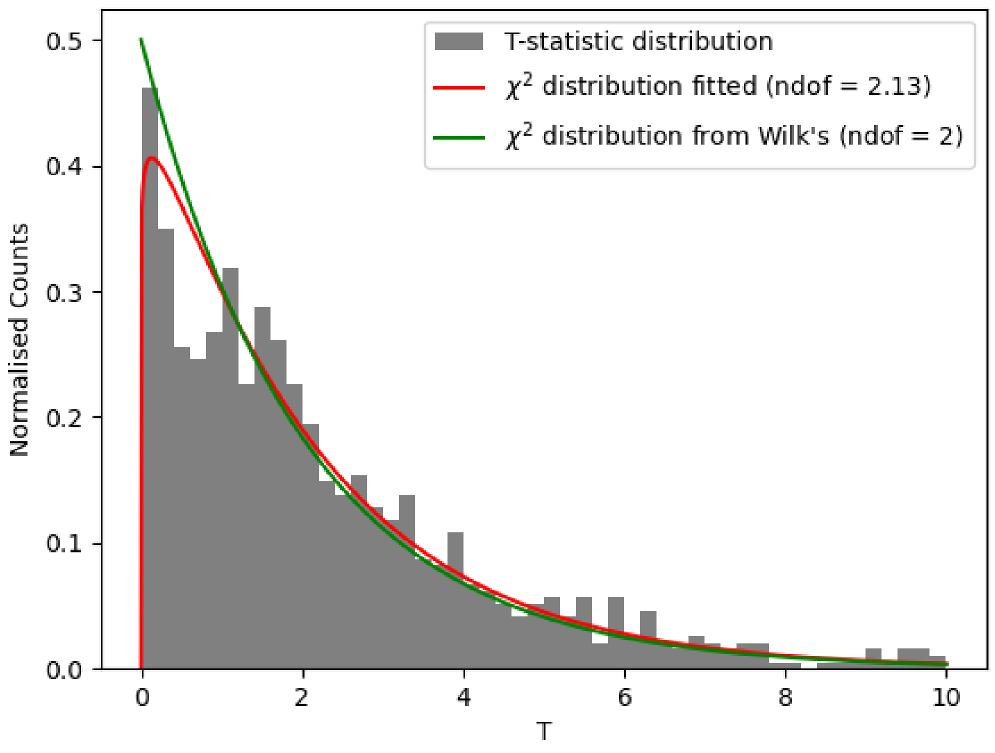

Figure 4: Plot of the fitted χ^2 distribution vs Wilk's theorem over the test-statistic histogram under the null hypothesis, for part (g).

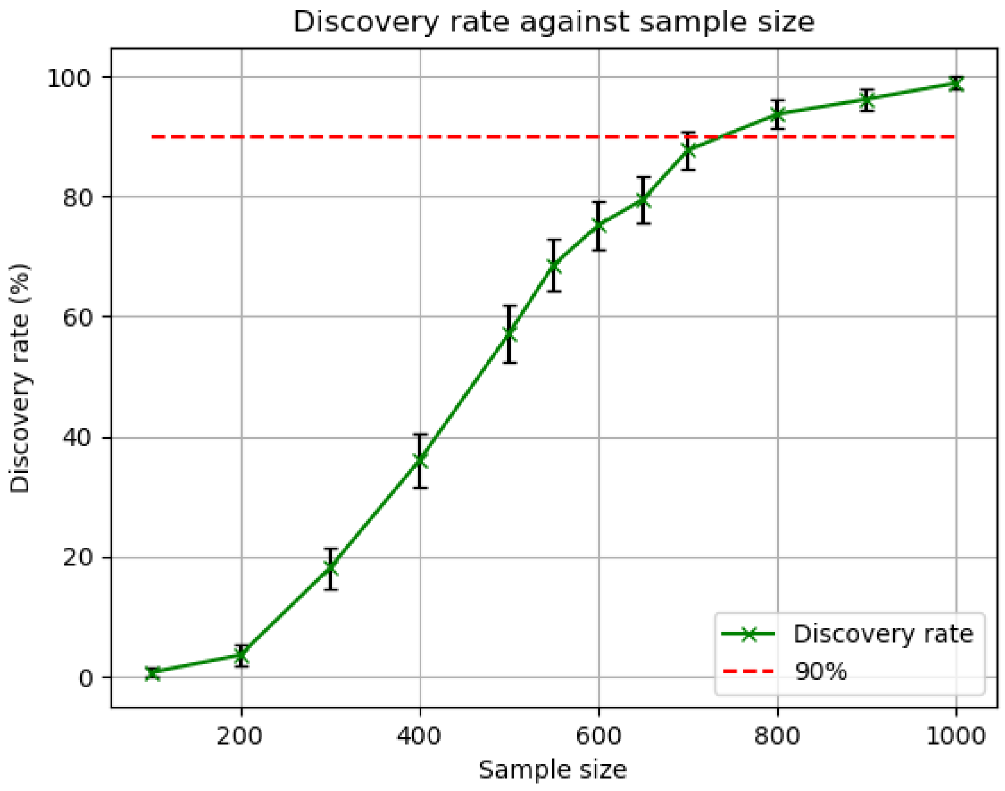

Figure 5: Plot of the discovery rate as a function of the sample size (blue line), with error bars of 3 standard deviations. The red line shows the 90% goal

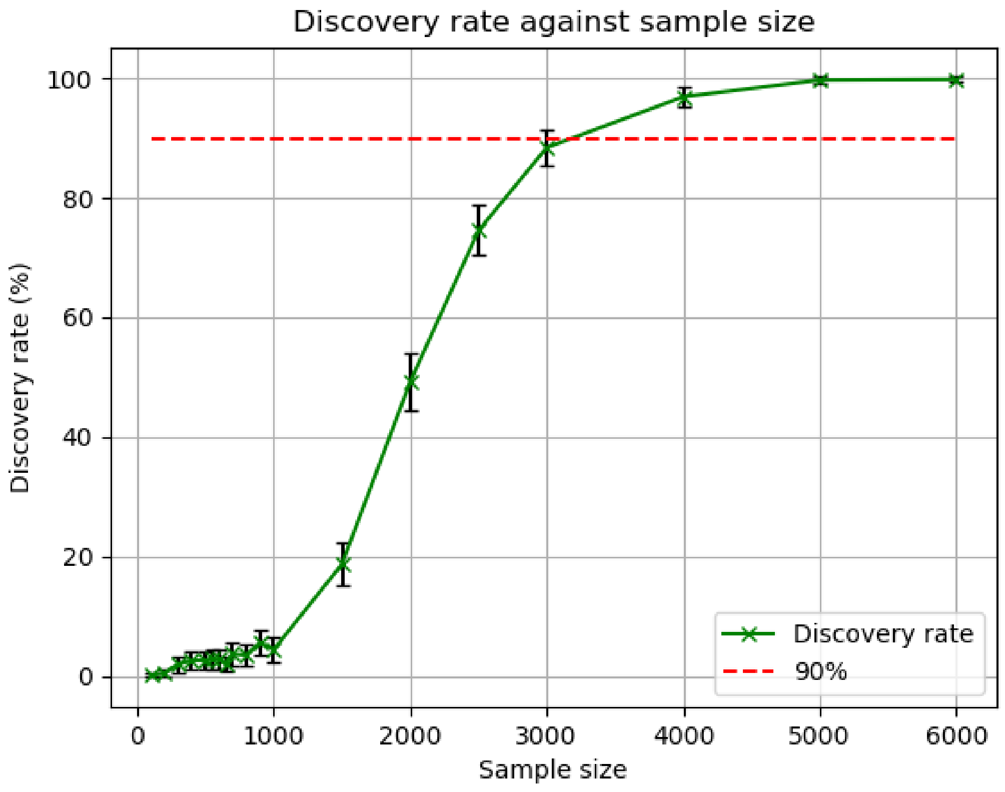

Figure 6: Plot of the discovery rate as a function of the sample size (blue line). The red line shows the 90% goal

it is clear that for 800 samples, there is a large proportion of the discovery rate distribution above the 90% threshold.

For (g):

For this simulation, the discovery rate stays low for much longer than in (f). Comparing the 2 simulations, discerning 2 signals may be much harder to do than simply detect that there is one. This is seen especially when looking at a plot of the 2^{nd} simulation's p.d.f.:

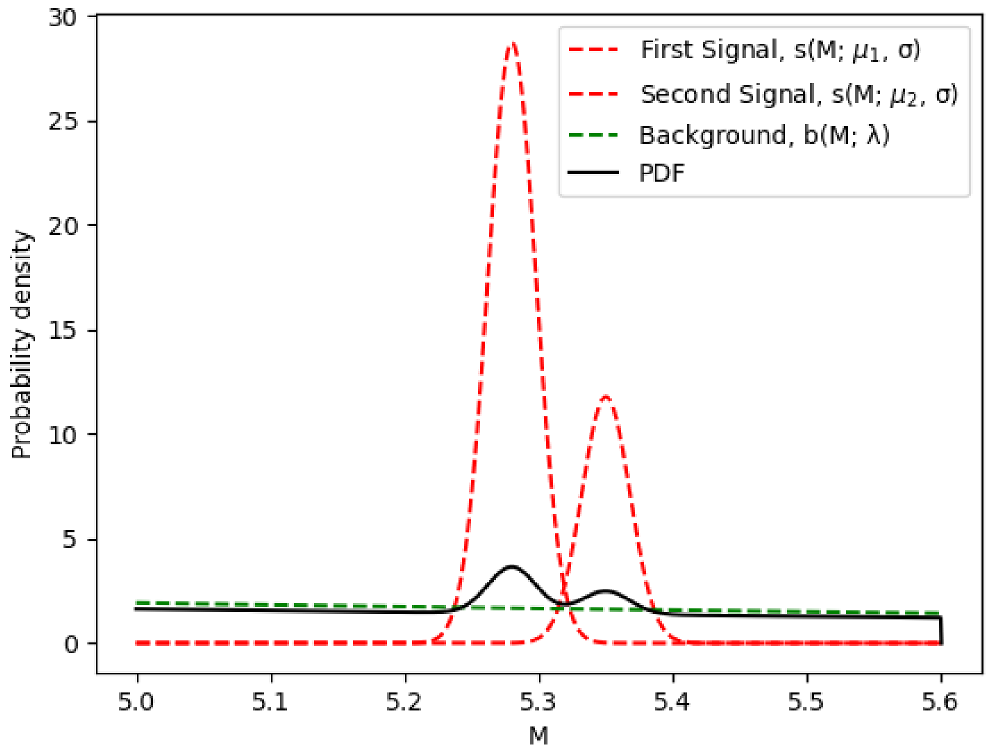

Figure 7: Plot of the total p.d.f. of the model, signal, and background, normalised for $M \in [5, 5.6]$.

One can see the 2nd signal is much smaller than the first, making it harder to detect for smaller sample sizes. In other words, for smaller samples, the 2nd signal is more likely to be hidden in the background, or the 2 signals can be confused as one wider signal. Overall, since the discovery threshold is quite high, it is likely the p-values obtained are still significant enough to reject the null hypothesis for smaller samples.

Discussion

Hence, the simulation shows that the minimum sample size for which the signal is discovered at least 90% of the time is around 700 samples. This is a reasonable result, considering the size of the signal component (Figure 2). For the 2nd simulation, the minimum sample size is slightly above 3000 samples. This increase in required sample size is reasonable considering the size and location of the 2nd signal component (Figure 5)

Bibliography

[1] Python Software Foundation. random.uniform. `https://docs.python.org/3/library/random.html#random.uniform`, Accessed: September 2021.

[2] Thomas Haslwanter. *An Introduction to Statistics with Python*. Springer Cham, 2 edition.

[3] John P. Huelsenbeck and Keith A. Crandall. Phylogeny estimation and hypothesis testing using maximum likelihood. *Annual Review of Ecology and Systematics*, 28:437–466, 1997.

[4] iminuit. iminuit tutorial basics documentation. `https://iminuit.readthedocs.io/en/stable/notebooks/basic.html`, Accessed: September 2021.

[5] iminuit. iminuit tutorial documentation. `https://iminuit.readthedocs.io/en/stable/notebooks/cost_functions.html#Maximum-likelihood-fits`, Accessed: September 2021.

[6] iminuit. iminuit.cost() cost functions documentation. `https://iminuit.readthedocs.io/en/stable/reference.html#module-iminuit.cost`, Accessed: September 2021.

[7] iminuit. iminuit.migrad() minimization documentation. `https://iminuit.readthedocs.io/en/stable/reference.html#iminuit.Minuit.migrad`, Accessed: September 2021.

[8] iminuit. iminuit.minuit documentation. `https://iminuit.readthedocs.io/en/stable/reference.html#minuit`, Accessed: September 2021.

[9] Matthew Kenzie. *Properties of the Higgs-like state around 125 GeV in its decay into two photons at the CMS experiment*. PhD thesis, Imperial Coll., London, 2014.

[10] Numba-Stats. Numba-stats description. `https://pypi.org/project/numba-stats/`, Accessed: September 2021.

[11] SciPy. Scipy - integration functions. `https://docs.scipy.org/doc/scipy/reference/integrate.html`, Accessed: September 2021.

[12] SciPy. Scipy - statistical functions. `https://docs.scipy.org/doc/scipy/reference/stats.html`, Accessed: September 2021.

[13] Larry Wasserman. *All of Statistics: A Concise Course in Statistical Inference.* Springer New York, NY, 2004.

[14] Wikipedia. Rejection sampling. `https://en.wikipedia.org/wiki/Rejection_sampling`, Accessed: September 2021.

[15] S. S. Wilks. The large-sample distribution of the likelihood ratio for testing composite hypotheses. *The Annals of Mathematical Statistics*, 9(1):60–62, 1938.

Appendix

Use of generative AI

Copilot's autocompletion feature was used in coding the project, when writing docstrings for the functions, and when covering repetitive parts of the code, like part (f) and (g)'s hypothesis testing. ChatGPT was used to help in debugging the code, by providing the tracebacks when an error was difficult to understand.

README file

Description

This repositery contains the code and written report for the S1 PDS coursework. The aim was to demonstrate the applications of statistical methods, including coding them.

Contents

Inside this `tmb76/` directory, there are 3 sub-directories to explore. One is the code directory (`src/`), which contains all the code used to generate the results and plots. The other is the plots directory (`Plots/`), which contains all the plots generated by the code. An important note is that the plots direcotry will be created by the code if it does not exist yet. So if there is no plots direcotry in the repository yet, running all the code should lead to creating and filling one. The last one is the report directory, which contains the LaTeX file for the report, as well as the pdf version of it. More importantly, there are an `environment.yml` file and a `Dockerfile`, which uses it, and one is advised to use.

How to run the code

For permissions reasons, the `Dockerfile` is not set up to pull the repository directly as it builds the image. Therefore, one must first pull this repository to their local machine and then are free to build the Docker image from the `Dockerfile`. All code can be run from the command line as follows:

```
    $ python src/solve_part_*.py
```

Listing 1: Running the code for each part

where * can be letters c through to g.

Additionally, there is a `ndof_for_part_f_g.py` file which covers determining the appropriate number of degrees of freedom to be used in parts (f) and (g) (cf. part (f) and (g)'s code). This is also in the `src/` directory, and can therefore be run in a similar fashion:

```
    $ python src/ndof_for_part_f_g.py
```

Listing 2: Running the number of DoF fitting code

Relevant outputs will be printed by the code to the terminal, and any plots generated will be saved to a `Plots/` directory, that will get created if it doesn't exist yet.

This can all be done within a Docker container, by building an image from the `Dockerfile`, and running a container from it.

From there, all files can be run in the same manner as above.

Note on time: The codes for part (c) to part (e) should each take less than a minute to run. For part (f), the code should take less than 5 minutes, with part (g) taking a little longer at The `ndof_for_part_f_g.py` is one of the more time consuming files to run, and could take around 10-15 minutes to run. This is based on running all of these on a MacBook Air M2 (2022, Ventura 13.2.1), with 8 GB of Memory.