Stats Notes

Maximum Likelihood

Say that we take samples from a probability density function $f(x;\theta)$, where x is an observed value, and θ some parameter. One problem to consider would be estimating θ given a set of measurements x_i - to do this, we can define the *likelihood* as:

$$L(\theta) = f(\boldsymbol{x};\theta)$$

For random samples (drawn from the same distribution), this is simply:

$$L(\theta) = \prod_i f(x_i;\theta)$$

If the samples are drawn from different distributions, then:

$$L(\theta) = \prod f_i(x_i;\theta)$$

θ might be a vector of parameters - if we **minimise** the likelihood with respect to these parameters, then we can find a *maximum likelihood estimate*, $\hat{\theta}(\boldsymbol{x})$.
We can also find the same result by maximising the *log-likelihood*, i.e. $\ell(\theta) = \ln L(\theta)$. This may be more convenient.

Estimators

Any function of observations X_i, $T(X_i)$, may act as an *estimator* for some parameter (e.g. the maximum likelihood). The mean-squared error of an estimator is:

$$\mathrm{MSE}(T) = E[(T-\theta)^2]$$

Here, $E[\cdot]$ signifies an expectation value. Furthermore, the bias is given:

$$b(T) = E(T) - \theta$$

In both cases, θ is the parameter which the estimator estimates. An unbiased estimator has $b = 0$.

Regression

We can model a linear fit for a dataset y_i as being drawn from a set of random variables, Y_i:

$$Y_i = \alpha + \beta x_i + \epsilon_i$$

Here, x_i are known constants, α, β unknown paramters, and ϵ_i a set of 'random errors'. In fact, this is a normal distribution, with $Y_i \sim N(\alpha + \beta x_i, \sigma^2)$, and we can assume that σ^2 is known. Then, the log-likelihood is:

$$\ell(\alpha,\beta) = -\frac{n}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_i \big(y_i - \alpha - \beta x_i\big)^2$$

The first term is just a constant, so we find that *maximising* the likelihood is **equivalent** to *minimising* the sum of squares:

$$S(\alpha,\beta) = \sum_i \big(y_i - \alpha - \beta x_i\big)^2$$

As such, the MLEs of α and β are known as the *least-squares estimators*.
In the general case, a linear model depends on any number of explanatory variables, $x_j^{(i)}$:

$$Y_j = \beta_0 + \sum_{i=1} \beta_i x_j^{(i)}$$

Then, the least-squares estimates are given by minimising:

$$S(\boldsymbol{\beta}) = \sum_j \left(y_j - \beta_0 + \sum_{i=1} \beta_i x_j^{(i)} \right)$$

We have thus far assumed that σ^2 is the same for all measurements, but if this is not the case, then we can perform a *weighted* least-squares minimisation:

$$\sum_i w_i \big(y_i - \alpha - \beta x_i\big)^2$$

Here, $w_i \propto \sigma_i^{-2}$, so points with smaller uncertainty will be reflected more.

Joint Probability Distribution

If we are working with multiple random variables (e.g. x, y), then we can represent their measurements as a *joint* probability distribution, $f(x, y)$. This may include some correlation between the two, i.e. loss of symmetry.
We can recover the *marginal* distributions, which describe the probability of a single measurement, by integrating the joint distribution over the other measurement:

$$f_X(x) = \int f(x, y)\ \mathrm{d}y \qquad\qquad f_Y(y) = \int f(x, y)\ \mathrm{d}x$$

Multivariate Normal Distribution

The multivariate normal distribution is a generalisation of the univariate normal distribution to arbitrary higher dimensions, i.e. a p-dimensional column vector $X = (X_1, \cdots, X_p)^T$. It allows for nonzero correlations to exist between vector elements (important!).
For said p-dimensional random column vector, X, we say that X has a multivariate normal distribution, with a vector of means $\mu = (\mu_1, \cdots, \mu_p)^T$, and a $p \times p$ *covariance* matrix, Σ, so that the *joint* pdf of X is:

$$f(x) = \frac{1}{(2\pi)^{\frac{p}{2}} \, |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}\big(x - \mu\big)^T \Sigma^{-1} \big(x - \mu\big) \right)$$

Here, $x \in \mathbb{R}^p$, i.e. a real column vector.
This covariance matrix, Σ, is key! If the measurements were all independent, then the off diagonal terms would clearly be zero, but by choosing them to be nonzero, we allow for a level of correlation between the different dimensions (this will be important). Specifically, we can interpret:

$$\mathbb{E}[X_i] = \mu_i \qquad \mathrm{var}(X_i) = \Sigma_{ii} \qquad \begin{array}{r}\mathrm{cov}(X_i, X_j) = \Sigma_{ij} \\ (i \neq j)\end{array}$$

cov (X_i, X_j), the covariance of two dimensions X_i and X_j, is a measure of their respective (linear) correlation, i.e. the tendency in their mutual linear relationship. A value of zero would (likely) indicate that they were independent (or non-linearly correlated). If the covariance is equal to the variance, it would indicate that the variables were *identical.*
Formally, we can define the *correlation* between these variables as:

$$\mathrm{cor}(X_i, X_j) \equiv \frac{\mathrm{cov}(X_i, X_j)}{\sqrt{\mathrm{var}(X_i)\,\mathrm{var}(X_j)}} = \frac{\Sigma_{ij}}{\sqrt{\Sigma_{ii}\Sigma_{jj}}}$$

Simulating a Multivariate Distribution

An important process to understand is that of drawing samples from a multivariate normal distribution - evidently, this will be somewhat complicated, given that we have to account for correlation between dimensions.
Here, I will outline a common method for achieving this:

1. Given the covariant matrix, Σ, we must find a matrix A which satisfies:

$$AA^T = \Sigma$$

   If Σ is positive-definite, we can use the *Cholesky decomposition*, i.e. $\text{chol}(\Sigma) = A^T$. We may also use a *SVD decomposition.*

2. Next, we want to generate a vector of independent standard normal variables, i.e. $\boldsymbol{z} \in \mathbb{R}^n$, where $z \sim N(0, 1)$.

3. Finally, our random vector is given by:

$$\boldsymbol{x} = \boldsymbol{\mu} + A\boldsymbol{z}$$

Given a set of observations from a multivariate normal distribution, we can make parameter estimations as

$$\hat{\mu} = \frac{1}{n} \sum_i x_i$$

for the mean, and

$$\hat{\Sigma} = \frac{1}{n} \sum_i (x_i - \hat{\mu})(x_i - \hat{\mu})^T.$$

Keep in mind that x_i and $\hat{\mu}$ are *vector* quantities. The above for $\hat{\Sigma}$ is biased; an unbiased estimator is:

$$\boldsymbol{S} = \frac{1}{n-1} \sum_i (x_i - \hat{\mu})(x_i - \hat{\mu})^T$$

This is the sample covariance matrix. We may also find it useful to calculate the sample correlation matrix:

$$\boldsymbol{R}_{ij} = \frac{\boldsymbol{S}_{ij}}{\sqrt{\boldsymbol{S}_{ii}\boldsymbol{S}_{jj}}}$$

Bayesian Inference

Say that we have a joint pdf, $f(y, z)$ - then the distributions of y, z *given* a fixed value of z, y are, respectively:

$$f(z \mid y) = \frac{f(y, z)}{f(y)} \qquad\qquad f(y \mid z) = \frac{f(y, z)}{f(z)}$$

Substituting one into the other, we have Bayes' theorem (for continuous random variables):

$$f(z \mid y) = \frac{f(y \mid z) f_Z(z)}{f_Y(y)}$$

The respective marginal densities are given by integrating as before.
Now, say that we have some unknown parameter θ - we have a probability model for data $\boldsymbol{x}$, $f(\boldsymbol{x} \mid \theta)$, which is conditional on the value of θ. Suppose that, before observing $\boldsymbol{x}$, we summarise our expectations (beliefs) about θ in a *prior* density, $\pi(\theta)$. Since $\boldsymbol{x}$ is observed (i.e. fixed), $f(\boldsymbol{x})$ is a constant.
Via Bayes' theorem, we can then *infer* a *posterior* density, $f(\theta \mid \boldsymbol{x})$, i.e. our updates beliefs about θ, given our obversations of $\boldsymbol{x}$:

$$\pi(\theta \mid \boldsymbol{x}) = \frac{f(\boldsymbol{x} \mid \theta) \cdot \pi(\theta)}{f(\boldsymbol{x})} \propto f(\boldsymbol{x} \mid \theta) \cdot \pi(\theta)$$

We can then find the constant of proportionality through normalising the posterior density.
Useful quantities can be drawn from the posterior density, e.g. mode, mean, median, variance, etc.

These help us to classify our beliefs about the value of θ.
The Bayesian equivalent of a confidence interval is a *credible* (or *posterior*) interval. Specifically, a $100(1-\alpha)\%$ credible set for θ is a subset C of Θ, such that:

$$\int_C \pi(\theta \mid \boldsymbol{x}) \, \mathrm{d}\theta = 1 - \alpha$$

That is, $P(\theta \in C \mid \boldsymbol{x}) = 1 - \alpha$.
In the case of multiple parameters, the same techniques apply, but we must interpret the posterior density as a *joint* density. As such, to find the marginal posterior of a single parameter (e.g. ψ), we integrate over all over components of $\pi(\psi, \lambda, \cdots)$.

Prediction

Now, let X_{n+1} be some future observation, and $\boldsymbol{x}$ our previously observed data. We will assume that, conditional on θ, X_{n+1} has density $f(x_{n+1} \mid \theta)$, independent of $X_1, X_2, \cdots, X_n$.
The density of X_{n+1} given $\boldsymbol{x}$, $f(x_{n+1} \mid \boldsymbol{x})$, is known as the *posterior predictive density.* We can express this as:

$$\begin{aligned} f(x_{n+1} \mid \boldsymbol{x}) &= \int f(x_{n+1}, \theta \mid \boldsymbol{x}) \, \mathrm{d}\theta \\ &= \int f(x_{n+1} \mid \theta, \boldsymbol{x}) \pi(\theta \mid \boldsymbol{x}) \, \mathrm{d}\theta \end{aligned}$$

The final line results from the identity that $f(u, v|w) = f(u|v, w)f(v|w)$.
Due to independence, though, we can write that $f(x_{n+1} \mid \theta, \boldsymbol{x}) = f(x_{n+1} \mid \theta)$. Then:

$$f(x_{n+1} \mid \boldsymbol{x}) = \int f(x_{n+1} \mid \theta) \pi(\theta \mid \boldsymbol{x}) \, \mathrm{d}\theta$$

As such, given $\boldsymbol{x}$, the predictive density is found from combining the density for x_{n+1} under the model (i.e. $f(x_{n+1} \mid \theta)$) with the posterior density.

Choosing a Prior

One potential source of ambiguity is in the choosing of the prior - this represents our beliefs about the parameters 'prior' to collecting data. We might:

1. Use the 'guess' of an expert - say that a scientific expert anticipates a parameter θ to be around 10, e.g. $\theta \in (5, 17)$ with a probability of 0.95.
   We might ask several different experts for their beliefs, and repeat analysis with this set of priors.

2. If we have little prior knowledge, we might seek a form expressing 'prior ignorance'. For example, if we were unsure of a probability, we might consider a uniform prior, e.g. $\theta \sim U(0, 1)$ (though this may lead to problems when there are a large number of parameters).

A prior, $\pi(\theta)$, is called 'proper' if $\int \pi(\theta) \, \mathrm{d}\theta = 1$, and improper if the integral can not be normalised (e.g. divergent). Improper priors can lead to a proper posterior, which may be used for inference. However, improper posteriors can **not** be used for inference.
A particular choice of prior is a *Jeffreys prior*, which has form

$$\pi(\theta) \propto i(\theta)^{\frac{1}{2}},$$

where we define

$$i(\theta) = -\mathbb{E}\left(\frac{\mathrm{d}^2}{\mathrm{d}\theta^2} \log f(X_1 \mid \theta)\right)$$

This retains ignorance on both the parameter, θ, and *log-odds*, $\phi = \log(1/(1-\theta))$.
For a vector parameter, Jeffreys prior can be extended as:

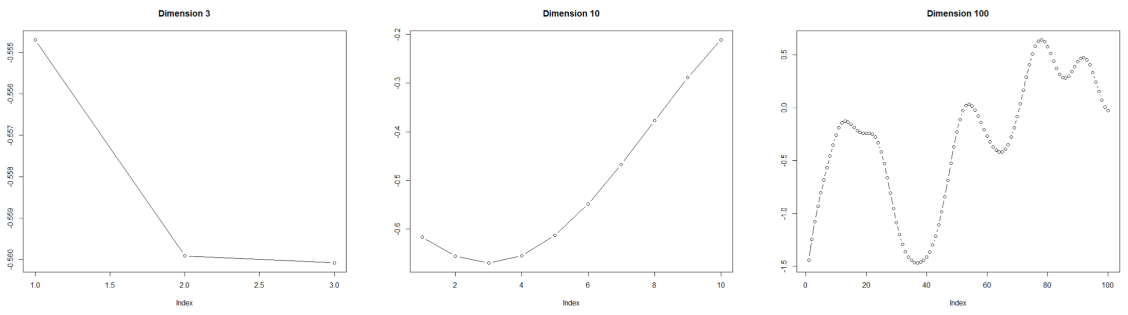
$$\pi(\theta) \propto |I(\theta)|$$

The RHS here denotes the root of the determinant of the 'information matrix'. A simpler (and more common) approach is to find the Jeffreys prior for each component individually, then use the product as the whole prior (i.e. assume prior independence).

Gaussian Processes

Introduction

We can define a multivariate (d-dimensional) normal distribution as some $N_d(\mu, \Sigma)$, where μ is a mean vector, and Σ a covariance matrix. This gives a vector output, where μ specifies the mean of each dimension, and Σ the variance and covariance *between* dimensions.
If Σ is purely diagonal, then the dimensions will be drawn independently, but if we incorporate some large correlation, then by plotting the dimensions sequentially, we see that it begins to resemble a smooth function:



We might then consider an infinite-dimensional multivariate gaussian, and use this to approximate a function on a continuous interval, where each dimension corresponded to some $f(x_i)$, i.e. the value of the function at some given x_i.
Formally, we write a *Gaussian process* as:

$$y(\cdot) \sim \text{GP}(\mu(\cdot), k(\cdot, \cdot))$$

$\mu(\cdot)$ is the mean function, which we will usually take to be zero, and which gives $\mathbb{E}[y(x)] = \mu(x)$. $k(\cdot, \cdot)$ is a *covariance function*, defined so that $\text{cov}(y(x), y(x')) = k(x, x')$. Any finite-dimensional segment of a full Gaussian process will be a multivariate distribution.
There are some restrictions on the form of the covariance function, it must be:

- **Symmetric**, so that $k(x, y) = k(y, x)$ for any x, y.
- **Positive semi-definite**, so that the matrix $K_{ij} = k(x_i, x_j)$ is positive semi-definite.

As we assume the mean is zero, we see that the covariance function will characterise the GP to a great extent.

Covariance Functions

A typical choice for the covariance function is a *squared-exponential* (*RBF*) kernel. This has a typical form:

$$k_{\text{SE}}(x, x') = \sigma_f^2 \exp\left(-\frac{|x - -x'|^2}{2\ell^2}\right)$$

Here, σ_f^2 is an associated kernel variance, and ℓ is a characteristic *lengthscale*. For most purposes, this is the kernel we will use, though there are other choices (see relevant materials).

Gaussian Process Regression

Consider a set of noisy observations, y, with known covariates x. Specifically, let us work with the case where this noise, ε, is gaussian, with a variance σ_n^2, such that $\varepsilon \sim N(0, \sigma_n^2)$. Then, for any two *observed* points, their covariance is given as:

$$\text{cov}(y_i, y_j) = k(x_i, x_j) + \sigma_n^2 \delta_{ij}$$

In the general case, we can write that:

$$\text{cov}(y) = K(x, x) + \sigma_n^2 \hat{\mathbb{1}}$$

To clarify notation, $K(x, x)$ is a matrix containing the covariance of every point with every other point, so that $K(x_i, x_j) = \text{cov}(y_i, y_j)$ - this is a function of x as the kernel (necessarily) has no dependence on y.
Now, the goal of the regression is to find some set of points, f', which approximate the underlying function over *every* point x'. We can show that the joint distribution of y and f' is given:

$$\begin{bmatrix} y \\ f' \end{bmatrix} \sim \left(0, \begin{bmatrix} K(x,x) + \sigma_n^2 \hat{\mathbb{1}} & K(x,x') \\ K(x',x) & K(x',x') \end{bmatrix}\right)$$

Finally, we can show that this leads to the following conditional distribution for f', i.e. our prediction:

$$f' \mid x, x', y \sim N(\bar{f}', \text{cov}(f'))$$

Here, we have defined the mean vector, $\bar{f}'$, and covariance matrix, $\text{cov}(f')$, from known quantities, as:

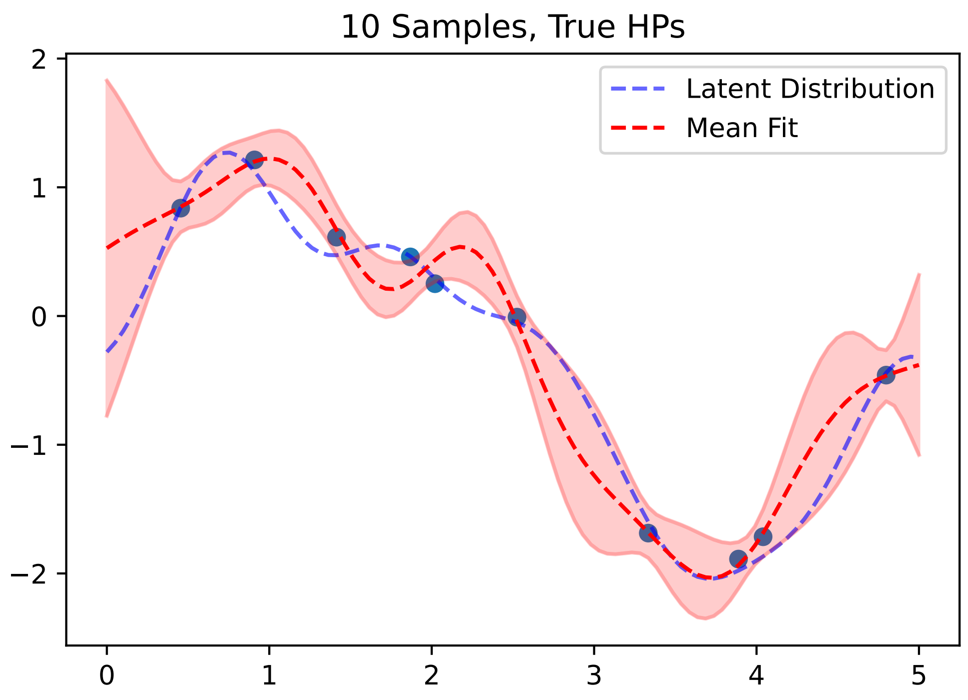
$$\boxed{\bar{f}' \equiv K(x', x) \left[K(x, x) + \sigma_n^2 \hat{\mathbb{1}}\right]^{-1} y}$$

$$\boxed{\text{cov}(f') \equiv K(x', x') - K(x', x) \left[K(x, x) + \sigma_n^2 \hat{\mathbb{1}}\right]^{-1} K(x, x')}$$

We can estimate a fitted function using some measure of the point distributions, e.g. the mean fit would be constructed from the mean vector. Furthermore, we can estimate the uncertainty on this fit from the covariance matrix - specifically, the diagonal elements give the variance (σ_i^2) of each fitted point.
An implementation of this regression algorithm, using numpy and scipy, can be found in my main programming file, `GPs.py`.

Hyperparameters

The main reference for this section will be section 5.4.1 of Rasmussen and Williams, Gaussian Processes.
For testing the principles of GP regression, we have the luxury of supplying our models with the exact (i.e. known) specifications of the kernel parameters, and noise variance. As such, we can achieve 'pretty good' fits straight out of the box; e.g. the following, where we fit random samples taken from a RBF GP, with $\sigma_f^2 = 1$, $\ell = 0.5$, with noise σ_n^2, using a GP with the same specifications:

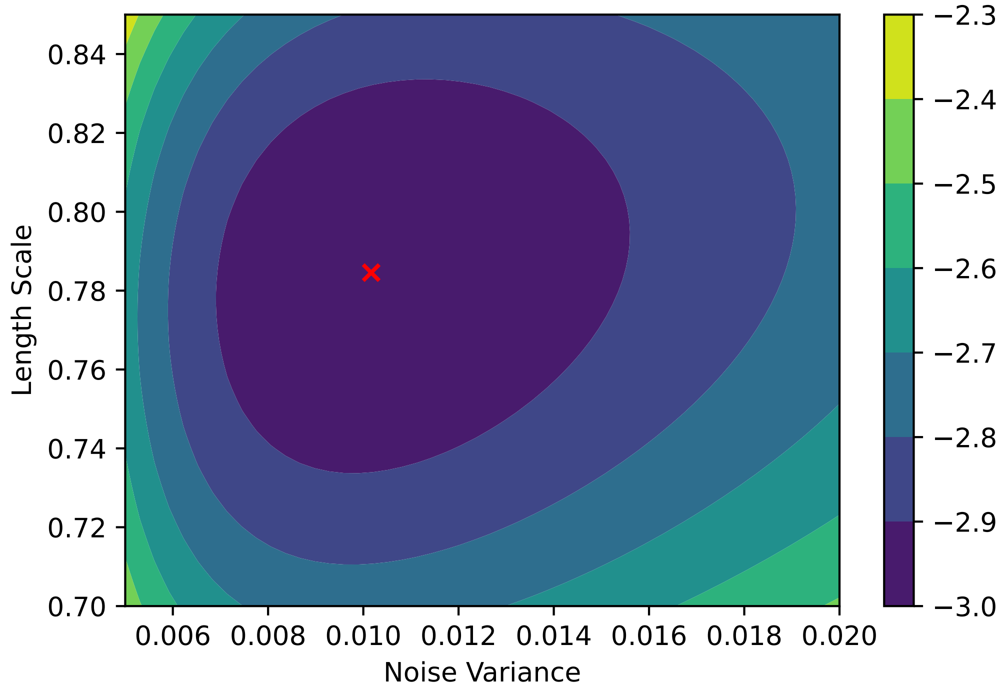


In practice, though, we will not actually know these 'exact' specifications (if they exist at all), and will need to estimate a set of hyperparameters for the dataset. One quantity which proves particularly useful in this regard is the *marginal likelihood* of observing the data $p(y \mid x)$, given a particular model specification. The log-form is simplest to work with (defining $K_{xx} \equiv K(x, x)$):

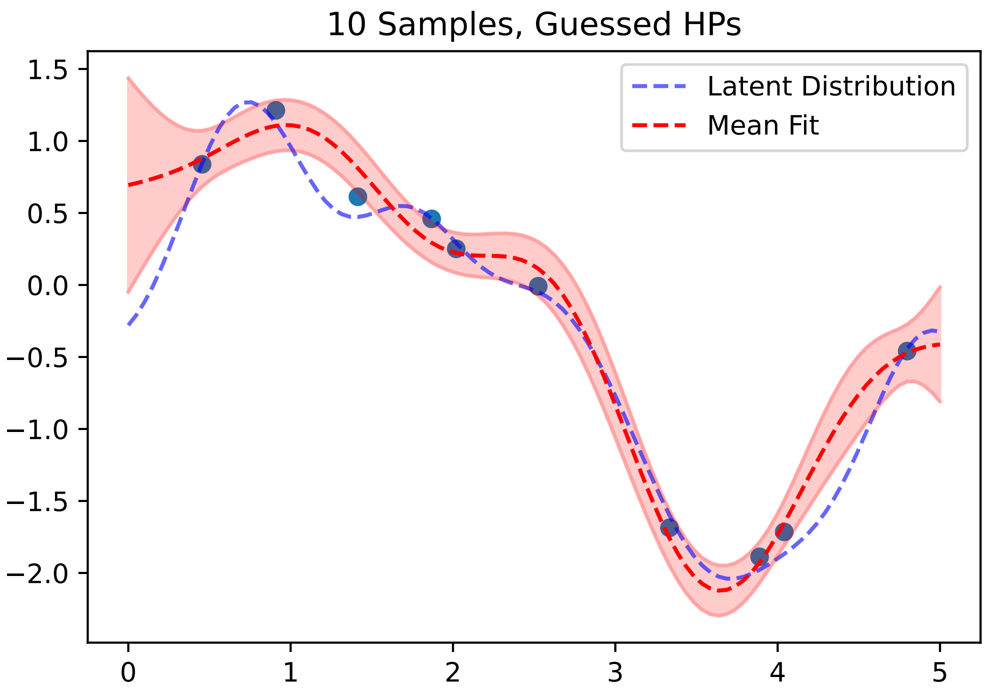
$$\log p(y \mid x) = -\frac{1}{2} y^T \left[K_{xx} + \sigma_n^2 \hat{\mathbb{1}}\right]^{-1} y - \frac{1}{2} \log \left|K_{xx} + \sigma_n^2 \hat{\mathbb{1}}\right| - \frac{n}{2} \log 2\pi$$

The final term has no dependence on the HPs, so we are only really interested in the first two. Maximising this log-likelihood (or, equivalently, minimising the negative log-likelihood) with respect to the hyperparameters will then yield an estimate.

For the same case as above, if we take $\sigma_f^2 = 1$, then we can minimise the negative log-likelihood with respect to ℓ and σ_n^2 to find estimates for these hyperparameters:



This can be achieved numerically (as it is above). In this specific example (seed 140725984), we find MLE estimates $\ell \simeq 0.785$ and $\sigma_n^2 \simeq 0.0102$, which are quite close to the 'actual' values. Regression under these specification yields a fairly good fit, in fact:



This being said, the effectiveness of this method varies - the log-likelihood may have local maxima, and the noise variance (particularly) is prone to being largely underestimated or overestimated.
A log-likelihood minimisation routine is implemented in the `GPs.py` program (though the plotting is quite inefficient).

Intro to GPJax

Basic Functionality

GPJax (**documentation here**) is a python package, built on the JAX structure, which greatly simplifies the amount of programming required to model GPs. Working in JAX (in contrast to numpy) is relatively straightforward, though there are some differences in, e.g., in-place modification (we must use `.at()`).
Generally, we would only require the following packages to be imported:

```
import jax.numpy as jnp
import jax.random as jr
from jax import config
import gpjax as gpx
[import matplotlib.pyplot as plt]
```

The first two constitute necessary JAX functions, the third is for changing *one very important setting*, the fourth is the package itself, and while `matplotlib` is technically optional, there is really no time when we will not be using it.
Specifically, the important setting that we must specify is:

```
config.update("jax_enable_x64",True)
```

JAX normally works with 32-bit floats, but for GPJax, we require that they are 64-bit! As such, this should always be specified at the start of the program.

A First Program

In this section, I will walk through my `GPjax_intro.py` file, which itself follows a tutorial available **here**. This should act as a reference for future programs.

```
[18]    key = jr.key(123)
```

The `jax.random` key acts as a seed for all random sampling; we should set this at the start.

```
[21]    N = 100
[22]    noise = 0.3
```

We specify the number of samples (100), and the noise standard deviation, $\sigma_n = 0.3$, so that $\varepsilon \sim N(\mathbf{0}, \sigma_n^2 \hat{\mathbb{1}})$.

```
[26]    key, subkey = jr.split(key)
```

We will be generating multiple random samples, so we may find it helpful to split our initial key (and use one for each).

```
[29]    x = jr.uniform(key=key,minval=-3.,maxval=3.,shape=(N,))
[30]    x = x.reshape(-1,1)
```

We generate N randomly selected covariates for our observations, sampling from a uniform distribution, with $x \in (-3, 3)$, with our first key. We must reshape the output to be a 2D array.

```
[33]    fun = lambda x: jnp.sin(3*x) + jnp.cos(2*x)
[35]    signal = fun(x)
[37]    y = signal + jr.normal(subkey,shape=signal.shape) * noise
```

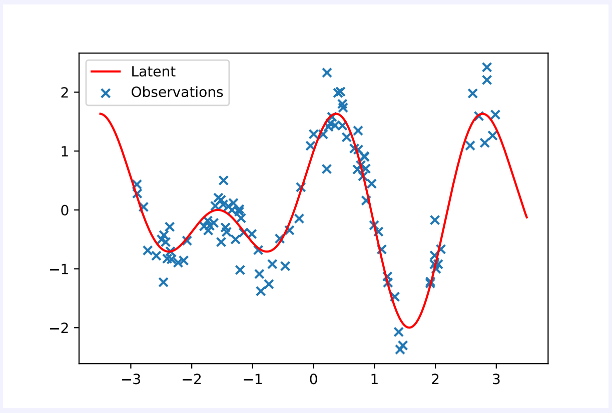
Define our latent function, e.g. $f(x) = \sin 3x + \cos 2x$, and take observations, y, with added gaussian noise.

```
[40]    D = gpx.Dataset(X=x,y=y)
```

Our first GPJax function - we place our observed data within a `Dataset` object. Note that this will not work if x is a 1D array.

```
[43]    x_lat = jnp.linspace(-3.5,3.5,500).reshape(-1,1)
[44]    y_lat = fun(x_lat)
```

Generate our 'latent' covariates (which will also be used for the prior), as well as the corresponding values of the latent function, which we might then plot:

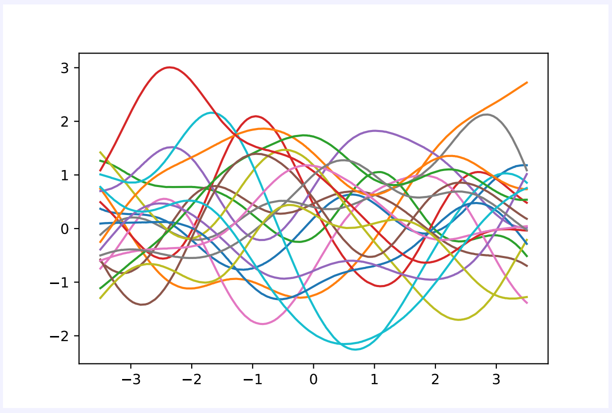


```
[53]    kernel = gpx.kernels.RBF()
[54]    meanf = gpx.mean_functions.Zero()
[56]    prior = gpx.gps.Prior(mean_function=meanf,kernel=kernel)
```

As an example, we will try to model these observations using a zero-mean GP, with an RBF kernel. These are all special objects within the GPJax package (as above). In line 56, we define this GP as our prior distribution, $f(\cdot) \sim N(0, \Sigma_p)$.

```
[60]    prior_dist = prior.predict(x_lat)
[61]    sample = prior_dist.sample(key=key,sample_shape=(1,))
```

If we wanted to draw a sample from this prior distribution, this is how we would do it. In fact, if we changed `(1,)` to some `(n,)`, we could generate as many as we liked:



```
[72]    likelihood = gpx.likelihoods.Gaussian(num_datapoints=D.n)
```

To construct a posterior, we would also require a likelihood. A simple choice is a gaussian, with noise parameter α - the value of this parameter is stored within the object, and initialised to 1. `D.n` simply returns the number of datapoints in `D`.

```
[77]    posterior = prior * likelihood
```

We construct the posterior exactly as we would write it on paper, 'multiplying' our prior object by the likelihood object. All (hyper)parameters are stored within, initialised to 1.

```
opt_posterior, history = gpx.fit_scipy(
model = posterior,
objective = lambda p,d: -gpx.objectives.conjugate_mll(p,d),
train_data = D
)
```

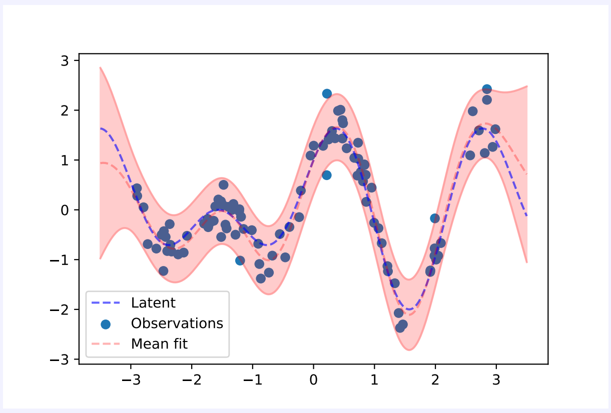
To estimate the hyperparameters, we perform a minimisation of the negative marginal log-likelihood; this is how we would do that! `opt_posterior` is the updated posterior, containing the fitted HPs, and `history` gives the values of the MLL in each step of the minimisation routine.

```
latent_dist = opt_posterior.predict(x_lat,train_data=D)
pred_dist = opt_posterior.likelihood(latent_dist)
```

Finally, we can construct our predictive distribution - from what I can understand, the first line ports over the prior hyperparameters (kernel variance and lengthscale), and the second ports over the likelihood hyperparameters (i.e. noise variance).

```
pred_mean = pred_dist.mean
pred_std = jnp.sqrt(pred_dist.variance)
```

From this object, we can extract the mean values of every points, as well as the associated error, and plot as usual:



Questions

I - *For the covariance function, or kernel, k of a GP, what information does it represent about any two points in the domain?*

The covariance function, k, of a GP dictates the covariance between any two given points, so that:

$$\text{cov}\left(f(x_1), f(x_2)\right) = k(x_1, x_2) = k(x_2, x_1)$$

That is to say, a large positive covariance between two random variables means that, if one is measured to be large (as compared to the mean), then the other will also likely be large, and vice versa.
In the context of a GP, this essentially specifies the form of the function generated.

II - *Assume that you know the exact GP model specification. Given some observation with additive Gaussian noise, how would you use those to predict the expected value (and uncertainty) of an unobserved location?*

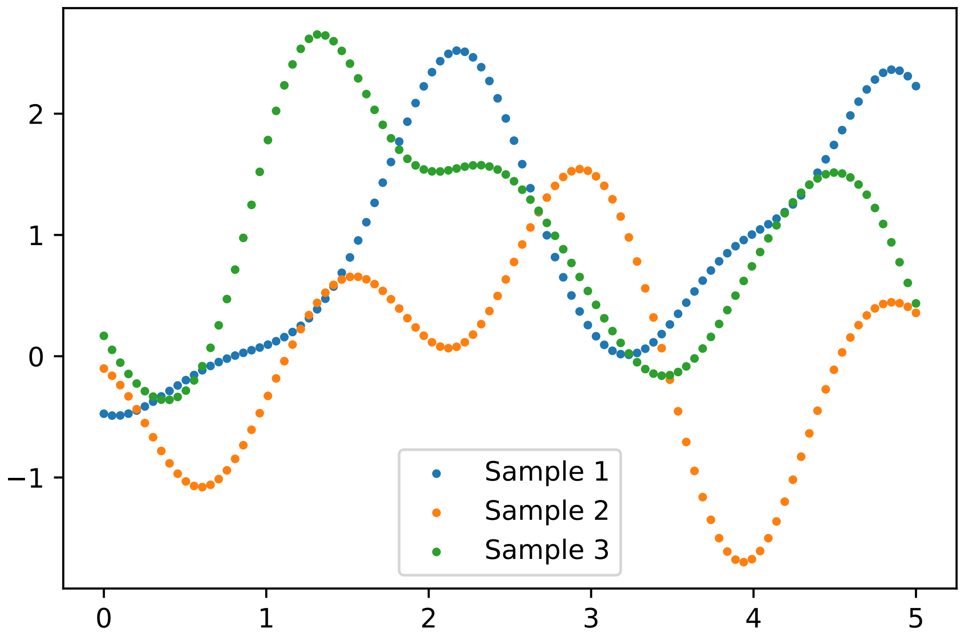
...

III - *Draw samples from a GP with one-dimensional input, zero-mean, RBF kernel with variance 1 and lengthscale 0.5.*

An RBF kernel with variance 1, and lengthscale 0.5, is mathematically given as:

$$k_{RBF}(x, x') = \exp\left(\frac{|x - x'|^2}{2}\right)$$

Using numpy and scipy (for multivariate sampling), I have plotted three samples from this GP, with $x \in (0, 5)$ and $N = 100$:



IV - *Do GP regression with observations made using an arbitrary sample from the above GP model (with additive, i.i.d. Gaussian noise of variance* 0.01*)*

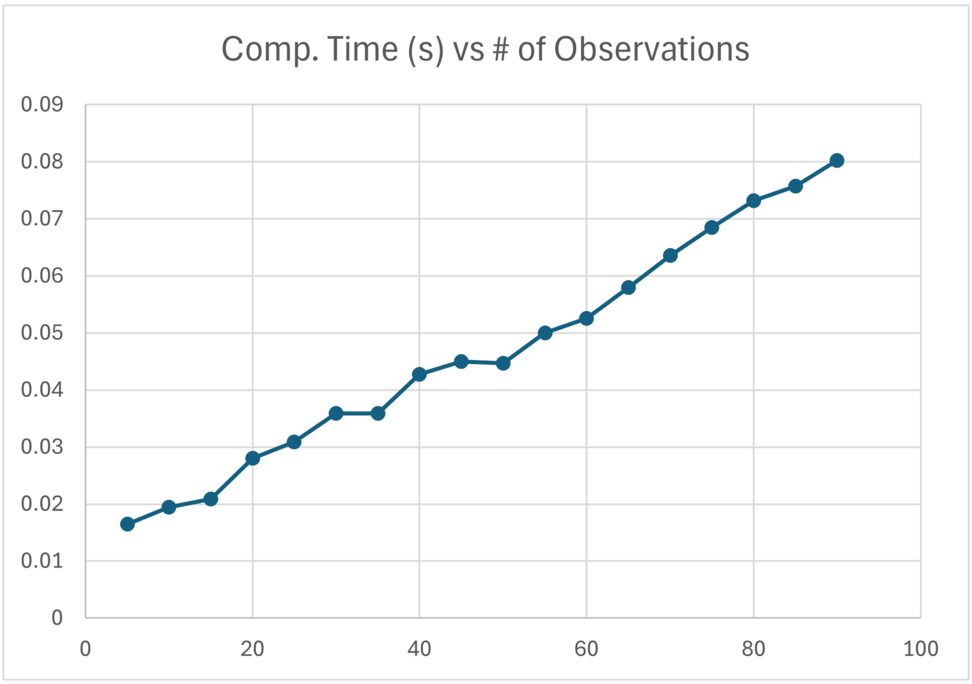
This has been demonstrated in the previous section. See `GPs.py` for the implementation.

V - *How would the computational cost of predicting the distribution of the GP at a new location scale as the number of observations increases? Show mathematically and support it with evidence from numerical experiments.*

The main cost in prediction will come from inversion of the $n \times n$ matrix, where n is the number of observations:

$$\bar{f}_i' = K(x_i', x) \cdot \left[K(x, x) + \sigma_n^2 \hat{\mathbb{1}}\right]^{-1} \cdot y$$

$K(x_i', x)$ and y will be row and column vectors of length n, respectively.
In the documentation for `np.linalg.inv()`, it states a time complexity $O(n^3)$. However, numerical observations seem to indicate a linear scaling:


Comp. Time (s) vs # of Observations