Quantified Degree of Belief, Posterior Distribution

Some machine learning algorithms output only a single number or decision.

It can be useful to have a measure of confidence in the output of the algorithm, a quantified degree of belief.

Bayesian statistical methods can be used to provide both estimates and confidence for users.

A model with parameters θ governs the process we are investigating.

We begin with a prior belief about the probability distribution of $\boldsymbol{\theta}$, the density $\pi(\boldsymbol{\theta})$.

Then the data we observed gives us a refined belief about the distribution $\boldsymbol{\theta}$. We obtain the posterior density $\pi(\boldsymbol{\theta}|\mathbf{x})$.

We can estimate values of $\boldsymbol{\theta}$, with the posterior mode of $\pi(\boldsymbol{\theta}|\mathbf{x})$, $\widehat{\boldsymbol{\theta}}$.

Then we can estimate the posterior variance of $\boldsymbol{\theta}$, and together with some knowledge of $\pi(\boldsymbol{\theta}|\mathbf{x})$ obtain confidence in our estimate $\widehat{\boldsymbol{\theta}}$.

Normal Approximation to the Posterior

We will use numerical optimization to obtain the posterior mode $\widehat{\boldsymbol{\theta}}$, maximizing the posterior

$\pi(\boldsymbol{\theta}|\mathbf{x})$

The posterior is proportional (where the scaling does not depend on $\boldsymbol{\theta}$) to the prior and likelihood (or density of the data).

$\pi(\boldsymbol{\theta}|\mathbf{x}) \propto L(\boldsymbol{\theta}|\mathbf{x})\pi(\boldsymbol{\theta})$

As in maximum likelihood, we directly maximize the log-posterior, $\log \pi(\boldsymbol{\theta}|\mathbf{x})$ because it is more stable.

Now, as described in section 4.1 of BDA 3 (Bayesian Data Analysis 3rd edn A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari and D. B. Rubin, 2013 Boca Raton, Chapman and Hall–CRC), we can approximate $\ln \pi(\boldsymbol{\theta}|\mathbf{x})$ using a 2nd order Taylor Expansion around $\widehat{\boldsymbol{\theta}}$.

$$\log \pi(\boldsymbol{\theta}|\mathbf{x}) \approx \log \pi(\widehat{\boldsymbol{\theta}}|\mathbf{x}) + (\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}})^T S(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}} + \frac{1}{2}(\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}})^T H(\widehat{\boldsymbol{\theta}})(\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}})$$

Where $S(\boldsymbol{\theta})$ is the score function

$$S(\boldsymbol{\theta}) = \frac{\delta}{\delta \boldsymbol{\theta}} \log \pi(\boldsymbol{\theta}|\mathbf{x})$$

and $H(\boldsymbol{\theta})$ is the Hessian function.

$$H(\boldsymbol{\theta}) = \frac{\delta}{\delta \boldsymbol{\theta}^T} S(\boldsymbol{\theta})$$

We assume that $\widehat{\boldsymbol{\theta}}$ is in the interior of the parameter space (or support) of $\boldsymbol{\theta}$.

Also, $\pi(\boldsymbol{\theta}|\mathbf{x})$ is a continuous function of $\boldsymbol{\theta}$.

Finally the Hessian matrix, $H(\boldsymbol{\theta})$ is negative definite, so $-H(\boldsymbol{\theta})$ is positive definite. This means that we can invert $-H(\boldsymbol{\theta})$ and get a matrix that is a valid covariance.

With these assumptions, as the sample size $n \to \infty$ the quadratic approximation for $\log \pi(\boldsymbol{\theta}|\mathbf{x})$ becomes more accurate. At the posterior mode $\boldsymbol{\theta} = \widehat{\boldsymbol{\theta}}$, $\log \pi(\boldsymbol{\theta}|\mathbf{x})$ is maximized and $0 = S(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}}$.

Given this, we can exponentiate the approximation to get

$$\pi(\boldsymbol{\theta}|\mathbf{x}) \approx \pi(\widehat{\boldsymbol{\theta}}|\mathbf{x}) \exp(\frac{1}{2}(\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}})^T H(\widehat{\boldsymbol{\theta}})(\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}}))$$

So for large n, the posterior distribution of $\boldsymbol{\theta}$ is approximately proportional to a multivariate normal density with mean $\widehat{\boldsymbol{\theta}}$ and covariance $-H(\widehat{\boldsymbol{\theta}})^{-1}$.

$$\boldsymbol{\theta}|x \approx_D N(\widehat{\boldsymbol{\theta}}, -H(\widehat{\boldsymbol{\theta}})^{-1})$$

Another caveat for this result is that the prior should be proper, or at least lead to a proper posterior.
Our asymptotic results are depending on probabilities integrating to 1.

We could get a quantified degree of beief by using resampling methods like MCMC directly.

We would have to use fewer assumptions. However, resampling can be computationally intense.

Parameter Constraints and Transformations

Optimization can be easier if the parameters are defined over the entire real line.

Parameters that do not follow this rule are plentiful. Variances are only positive. Probabilities are in [0,1].

Stan provides the ease of optimizing parameters over the entire real line by creating unconstrained parameters. These are continuous functions of the constrained parameters, which may be defined on intervals of the real line.

For example, the unconstrained version of a standard deviation parameter σ is $\psi = \log \sigma$. ψ is defined over the entire real line.

It will be useful for us to consider the constrained parameters as being functions of the unconstrained parameters.

So $\sigma = exp(\psi)$ is our constrained parameter of ψ.

So the posterior mode of the constrained parameters $\boldsymbol{\theta_c}$ is $\widehat{\boldsymbol{\theta}}_{\boldsymbol{c}} = g(\widehat{\boldsymbol{\theta}})$. We will call g the **constraint** function

Then we can use the delta method on g.

A first-order taylor approximation of $g(\boldsymbol{\theta})$ at $\widehat{\boldsymbol{\theta}}$ yields

$$g(\boldsymbol{\theta}) \approx g(\widehat{\boldsymbol{\theta}}) + \left\{ \frac{\delta}{\delta \widehat{\boldsymbol{\theta}}} g(\widehat{\boldsymbol{\theta}}) \right\} (\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}})$$

Remembering that $\boldsymbol{\theta}$ is approximately normal, the rules about linear transformations for multivariate normal random vectors tell us that

$$\boldsymbol{\theta_c}|x = g(\boldsymbol{\theta})|x \approx_D N\left[g(\widehat{\boldsymbol{\theta}}), \left\{\frac{\delta}{\delta\widehat{\boldsymbol{\theta}}}g(\widehat{\boldsymbol{\theta}})\right\}^T \left\{-H(\widehat{\boldsymbol{\theta}})^{-1}\right\} \left\{\frac{\delta}{\delta\widehat{\boldsymbol{\theta}}}g(\widehat{\boldsymbol{\theta}})\right\}\right]$$

This involved a first-order approximation of g.

Earlier we used a second order approximation for taking the numeric derivative. Why would we just do a first-order here?

Traditionally the delta-method is taught and used as only a first-order method.

Usually the functions used in the delta method are not incredibly complex. It is *good enough* to to use the first-order approximation.

Hessian approximation

To be able to use the normal approximation, we need $\widehat{\boldsymbol{\theta}}$, $H(\widehat{\boldsymbol{\theta}})^{-1}$, and $\frac{\delta}{\delta\widehat{\boldsymbol{\theta}}}g(\widehat{\boldsymbol{\theta}})$.

As mentioned before, we use numerical optimization to get $\widehat{\boldsymbol{\theta}}$. Ideally, we would have analytic expressions for H and the derivatives of g.

In PyStan, no estimate of the Hessian is exposed.

RStan estimates a Hessian using numerical differentiation on the analytic gradient. See:

https://stackoverflow.com/questions/27202395/how-do-i-get-standard-errors-of-maximum-likelihood-estimates-in-stan

https://cran.r-project.org/web/packages/rstan/rstan.pdf

We can also perform numerical differentiation in Python to get the Hessian and the gradient of the constraint function g. This will be less accurate than an analytic expression, and will also normally be more computationaly intensive.

But "*Once you learn how to take one numeric derivative, you can take the numeric derivative of anything*". So using numerical differentiation is a very flexible technique that we can easily apply to all the models that we would use in Stan.

Numerical Differentiation

So numeric derivatives can be very pragmatic, and flexible.

How do you compute them? Are they accurate? We use the following reference as a guide.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. Numerical Recipes 3rd Edition: The Art of Scientific Computing (3rd. ed.). Cambridge University Press, USA.

Let's look at the definition of a derivative.

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

To approximate $f'(x)$ numerically, couldn't we just plugin a small value for h and compute the scaled difference?

Yes. And that is basically what happens.

We do do a little more work to choose h and use a second-order approximation instead of a first-order.

We can see that the scaled difference is a first-order approximation by looking at the Taylor series expansion around x.

Taylor's theorem with remainder gives

$$f(x+h)$$
$$= f(x) + ((x+h)-x)f'(x) + .5((x+h)-x)^2 f''(\epsilon)$$
$$= f(x) + -hf'(x) + .5hf''(\epsilon)$$

where ϵ is between x and $x+h$.

We can rearrange to get

$$\frac{f(x+h)-f(x)}{h} - f'(x) = .5hf''(\epsilon)$$

This is linear in h.

We can do second-order approximations for $f(x+h)$ and $f(x-h)$

$$f(x+h) = f(x) + ((x+h)-x)f'(x) + \frac{((x+h)-x)^2 f''(x)}{2!} + \frac{((x+h)-x)^3 f'''(\epsilon_1)}{3!}$$
$$f(x-h) = f(x) + ((x-h)-x)f'(x) + \frac{((x-h)-x)^2 f''(x)}{2!} + \frac{((x-h)-x)^3 f'''(\epsilon_2)}{3!}$$

where ϵ_1 is between x and $x+h$ and ϵ_2 is between $x-h$ and x.

Then we have

$$\frac{f(x+h)-f(x-h)}{2h} - f'(x) = h^2 \frac{f'''(\epsilon_1)+f'''(\epsilon_2)}{12}$$

This is quadratic in h.

The first term takes equal input from both sides of x, so we call it a centered derivative.

So we choose a small value of h and plug it into

$$\frac{f(x+h)-f(x-h)}{2h}$$

to approximate $f'(x)$.

Our derivation used a single input function f. The idea applies to partial derivatives of multi-input functions as well. The inputs that you aren't taking the derivative with respect to are treated as fixed parts of the function.

Choosing a Bandwidth, *h*

In practice, second order approximation actually involves two sources of error.

Roundoff error, ϵ_r arises from being unable to represent x and h or functions of them with exact binary represetation.

$$\epsilon_r \approx \epsilon_f \frac{|f(x)|}{h}$$

where ϵ_f is the fractional accuracy with which f is computed. This is generally machine accuracy

$\epsilon_f = \text{np.finfo(float).eps}$

The remainder that we showed earlier, $h^2 \frac{f'''(\epsilon_1) + f'''(\epsilon_2)}{12}$ is referred to as **truncation** error.

Minimizing

$$\epsilon_r + \epsilon_t \approx \epsilon_f \frac{|f(x)|}{h} + h^2 \frac{f'''(\epsilon_1) + f'''(\epsilon_2)}{12}$$

we obtain

$$h \sim \epsilon_f^{1/3} \left(\frac{f}{f'''} \right)^{1/3}$$

where $\left(\frac{f}{f'''} \right)^{1/3}$ is shorthand for the ratio of $f(x)$ and the sum of $f'''(\epsilon_1) + f'''(\epsilon_2)$.

We use shorthand here because because we are not going to approximate f''' (we are already approximating f'), so there is no point in writing it out.

Call the shorthand

$$\left(\frac{f}{f'''} \right)^{1/3} = x_c$$

the curvature scale, or characteristic scale of the function f.

There are several algorithms for choosing an optimal scale.

The better the scale chosen, the more accurate the approximation is.

A good rule of thumb, which is computationally quick, is to just use the absolute value of x.

$x_c = | x|$

Then we would use

$h = \epsilon_f^{1/3} \ | x|$

But what if x is 0?

This is simple to handle, we just add $\epsilon_f^{1/3}$ to $x_c = | x|$

$h = \epsilon_f^{1/3}(| x \ | + \epsilon_f^{1/3})$

Now, Press et al. also suggest performing a final sequence of assignment operations that ensures x and $x + h$ differ by an exactly representable number. You assign $x + h$ to a temporary variable $temp$. Then h is assigned the value of $temp - h$.

Using some dummy values for x and h, the code would look like

In [1]:
```
x=32
h=.5
```

In [2]:
```
temp = x + h
h = temp - x
```

Estimating Variance after Optimization

Now let's apply what we have learned to estimating $H(\widehat{\boldsymbol{\theta}})^{-1}$ and $\frac{\delta}{\delta\widehat{\boldsymbol{\theta}}}g(\widehat{\boldsymbol{\theta}})$.

Pystan will provide an estimate of the constrained posterior mode $g(\widehat{\boldsymbol{\theta}})$ by using the **optimizing()** function. We'll refer to this value as **opt_parameters**.

Now to get started on estimating $H(\widehat{\boldsymbol{\theta}})^{-1}$ we have to call PyStan's **sampling()** function.

We use 1 iteration and specify **[opt_parameters]** in the **init** option and use "Fixed_param" in the **algorithm** option. Let's call the object that **sampling()** returns **samp**.

This will allow us to compute the probability and gradient for the unconstrained posterior mode $\widehat{\boldsymbol{\theta}}$.

We get the unconstrained posterior mode, $\widehat{\boldsymbol{\theta}}$ by using the **unconstrain_pars()** function with argument **opt_parameters** on **samp**. We refer to this value as **unconstrained_pars**.

The **grad_log_prob()** function can be applied to **samp** with argument **unconstrained_pars** to get the score function for the unconstrained parameters.

The **constrain_pars()** function can be used to compute the constrained parameters for any given input unconstrained parameters.

We loop over the unconstrained parameters $\theta_1, \ldots, \theta_K$.

For iteration i, we calculate the bandwidth h_i for parameter θ_i, as discussed in the previous section with $x = \theta_i$. We also create two copies of the **unconstrained_pars** vector, **parm_plus** and **parm_minus**.

parm_plus is updated by adding h_i to θ_i and **parm_minus** is updated by subtracting h_i from θ_i. Then we use the **grad_log_prob()** function to compute the score at **parm_plus** and the score at **parm_minus**.
These are stored in **grad_plus** and **grad_minus**. Column i of the Hessian is computed as (**grad_plus**-**grad_minus**)/$2h_i$.

We use the **constrain_pars** function to compute the constrained parameters at **parm_plus** and the constrained parameters at **par_minus**. These are stored as **cons_plus** and **cons_minus**. Column i of $\frac{\delta}{\delta\widehat{\boldsymbol{\theta}}}g(\widehat{\boldsymbol{\theta}})$ is computed as (**cons_plus**-**cons_minus**)/$2h_i$.

After looping over the parameters, we compute the matrix product $\left\{\frac{\delta}{\delta\widehat{\boldsymbol{\theta}}}g(\widehat{\boldsymbol{\theta}})\right\}^T \left\{-H(\widehat{\boldsymbol{\theta}})^{-1}\right\} \left\{\frac{\delta}{\delta\widehat{\boldsymbol{\theta}}}g(\widehat{\boldsymbol{\theta}})\right\}$, our estimate of the covariance of $\boldsymbol{\theta_c} = g(\boldsymbol{\theta})$.

Let's define a Python function that will get this done. This functionality may be broken into several functions later, and customization in the optimization will be provided.

In [ ]:
```
```

First we define some functions that let us suppress output.

In [3]:
```
import os
```

In [4]:
```
# from https://stackoverflow.com/questions/11130156/suppress-stdout-stderr-print
class suppress_stdout_stderr(object):
    '''
    A context manager for doing a "deep suppression" of stdout and stderr in
    Python, i.e. will suppress all print, even if the print originates in a
    compiled C/Fortran sub-function.
       This will not suppress raised exceptions, since exceptions are printed
    to stderr just before a script exits, and after the context manager has
    exited (at least, I think that is why it lets exceptions through).

    '''
    def __init__(self):
        # Open a pair of null files
        self.null_fds = [os.open(os.devnull, os.O_RDWR) for x in range(2)]
        # Save the actual stdout (1) and stderr (2) file descriptors.
        self.save_fds = (os.dup(1), os.dup(2))

    def __enter__(self):
        # Assign the null pointers to stdout and stderr.
        os.dup2(self.null_fds[0], 1)
        os.dup2(self.null_fds[1], 2)

    def __exit__(self, *_):
        # Re-assign the real stdout/stderr back to (1) and (2)
        os.dup2(self.save_fds[0], 1)
        os.dup2(self.save_fds[1], 2)
        # Close the null files
        os.close(self.null_fds[0])
        os.close(self.null_fds[1])
```

In [5]:
```
import pystan
import numpy as np
import warnings
warnings.filterwarnings('ignore')
```

In [6]:
```
def estimate_map(stan_model,stan_data):
    sm = stan_model
    optimizing_fit = sm.optimizing(stan_data,algorithm="Newton",init='0')
    opt_parameters = optimizing_fit
    parkeys = list(opt_parameters.keys())
    test_grad_fit = sm.sampling(stan_data,                \
            iter=1,chains=1,warmup=0,init=[opt_parameters],check_hmc_diagnostics
                                algorithm="Fixed_param")
    unconstrained_pars = test_grad_fit.unconstrain_pars(opt_parameters)
    nparm = unconstrained_pars.shape[0]
    # note that constrained and unconstrained might have different dimensions.
    nparmc = len(parkeys)
    pars = unconstrained_pars


    Hessian = 0*np.ones((nparm,nparm))
    Delta = 0*np.ones((nparmc,nparm))
```

```
    epsdouble = np.finfo(float).eps
    epsdouble = epsdouble**(1/3)
    for i in np.arange(0,nparm):
        scaleparm = abs(pars[i])
        scale = epsdouble*(scaleparm+epsdouble)
        scaleparmstable = scaleparm+scale
        scale = scaleparmstable-scaleparm
        parmplus = pars.copy()
        parmminus = pars.copy()
        parmplus[i]=parmplus[i]+scale
        parmminus[i]=parmminus[i]-scale
        gradplus = test_grad_fit.grad_log_prob(parmplus)
        gradminus = test_grad_fit.grad_log_prob(parmminus)
        Hessian[:,i] = (gradplus-gradminus)/(2*scale)
        consplus = test_grad_fit.constrain_pars(parmplus)
        consminus = test_grad_fit.constrain_pars(parmminus)
        Delta[:,i] = (consplus[0:nparmc]-consminus[0:nparmc])/(2*scale)

    iHessian = -np.linalg.inv(Hessian)
    Cov = iHessian
    Cov = np.matmul(np.matmul(Delta,Cov),Delta)
    return {'mode':opt_parameters,'Cov':Cov,"stan_model":sm}
```

We can provide users standard deviations based on this estimated covariance. This will provide measures of variability for the parameters.

In [ ]:

Estimating Confidence Intervals after Optimization

With the posterior mode, variance, and normal approximation to the posterior. It is simple to create confidence (credible) intervals for the parameters.

Let's talk a little bit about what these intervals are.

For the parameter γ we want a $(1-\alpha)$ interval (u,l) (defined on the observed data generated by a realization of γ) to be defined such that

$$\Pr(\gamma \in (u,l)) = 1-\alpha$$

The frequentist confidence interval does not meet this criteria. γ is just one fixed value, so it is either in the interval, or it isn't!

A credible interval (Bayesian confidence interval) can meet this criteria.

Suppose that we are able to use the normal approximation for $\gamma|\mathbf{x}$

$\gamma|\mathbf{x} \approx_D N(\hat{\gamma}, \hat{\sigma}^2_\gamma)$

$$\begin{aligned} 1-\alpha &= \Pr(l \leq \gamma \leq u|\mathbf{x}) && (1)\\ &= \Pr(l-\hat{\gamma} \leq \gamma - \hat{\gamma} \leq u - \hat{\gamma}|\mathbf{x}) && (2)\\ &= \Pr\left(\frac{l-\hat{\gamma}}{\hat{\sigma}_\gamma} \leq \frac{\gamma-\hat{\gamma}}{\hat{\sigma}_\gamma} \leq \frac{u-\hat{\gamma}}{\hat{\sigma}_\gamma}|\mathbf{x}\right) && (3)\end{aligned}$$

Now, $\frac{\gamma - \hat{\gamma}}{\hat{\sigma}_\gamma^2}$ is $N(0,1)$, standard normal. So we can use the standard normal quantiles in solving for l and u.

The upper $\alpha/2$ quantile of the standard normal distribution, $z_{\alpha/2}$ satisfies

$$\Pr(Z \geq z_{\alpha/2}) = \alpha/2$$

for standard normal Z.

Noting that the standard normal is symmetric, if we can find l and u to satisfy

$$\frac{l - \hat{\gamma}}{\hat{\sigma}_\gamma} = -z_{\alpha/2}$$

$$\frac{u - \hat{\gamma}}{\hat{\sigma}_\gamma} = z_{\alpha/2}$$

then we have a valid Bayesian confidence interval.

Simple calculation shows that the solutions are

$$l = -z_{\alpha/2}\hat{\sigma}_\gamma + \hat{\gamma}$$

$$u = z_{\alpha/2}\hat{\sigma}_\gamma + \hat{\gamma}$$

The $z_{\alpha/2}$ quantile can be easily generated using **scipy.stats**.

We can also adjust the intervals for inference on many parameters by using Bonferroni correction.

Example

Now we know how to estimate the posterior variance after computing the posterior mode. We also know how confidence intervals are made based on this posterior variance, mode, and the normal approximation to the posterior.

Let's perform a simulation to corroborate our results and show how they can be used in practice.

We will see that we can properly estimate the posterior mode and variance, and use the normal approximation for the posterior distribution to generate confidence intervals.

First we import the necessary libraries.

In [7]:
```
import pandas as pd
import scipy.stats as spstats
```

Now let's define our Stan specification for the model. This a linear regression with a χ^2 prior for the intercept α and a normal prior for the slope β.

In [8]:
```
model = """
data {
    int<lower=0> N;
    vector[N] x;
    vector[N] y;
```

```
}

parameters {
    real<lower=0> alpha;
    real beta;
}

model {
    alpha ~ chi_square(4);
    beta ~ normal(1,1);
    y ~ normal(alpha + beta * x,1);
}
"""
with suppress_stdout_stderr():
    stan_model = pystan.StanModel(model_code=model)
```

```
INFO:pystan:COMPILING THE C++ CODE FOR MODEL anon_model_a0d1a455dfb70045e204be5b
ae7f04b9 NOW.
```

We have a constrained parameter and different types of prior distributions here. This should provide robust corroboration that our methods work. Note again that the priors are proper. This is a requirement for our asymptotic approximations.

Now let's define a simulation function that we can call to draw from the model. It takes as argument the sample size **n** and a random state **rs**.

In [9]:

```
def draw(n,rs):

    # draw parameters from priors
    beta = rs.normal(1.0,1)
    alpha = rs.chisquare(4)
    # draw observed data based on drawn parameters
    x = rs.normal(size=n)
    y = alpha + beta * x
    y = rs.normal(y,1)
    data = {'N': n, 'x': x, 'y': y}
    return (data)
```

Once we obtain a simulated draw from the model, we want to check the distribution of the posterior.

If we perform multiple draws from the original model and evaluate them marginally over the draws, we **are** evaluating the posterior.

This can help us determine misspecification of the posterior.

It will not let us directly evalute the posterior distribution though, because we are averaging over multiple draws of the observed data.

So for each draw from the original model, we will use MCMC to perform posterior draws, which will then be used to evaluate the posterior distribution.

The draws provide a posterior sample of α and β.

The posterior sample may need to be thinned to reduce autocorrelation and obtain an independent sample.

Once thinning is completed, we create another parameter, $s_{\alpha\beta} = \alpha + \beta$.

For each posterior sample, we record the mean and standard deviation of α, β, and $s_{\alpha\beta}$ as μ_α, μ_β, and $\mu_{s_{\alpha\beta}}$ and σ_α, σ_β, and $\sigma_{s_{\alpha\beta}}$.

The standard deviation of $s_{\alpha\beta}$, $\sigma_{s_{\alpha\beta}}$ is the square-root of

$$\mathrm{Var}(\alpha + \beta) = \mathrm{Var}(\alpha) + \mathrm{Var}(\beta) + 2\mathrm{Cov}(\alpha, \beta)$$

So evaluating the standard deviation of $s_{\alpha\beta}$ will help us evaluate the covariance of α and β.

We also record our estimates of the posterior mode and variance that we derived using numerical optimization and the approximation methods discussed above. These are $\hat{\mu}_\alpha$, $\hat{\mu}_\beta$, and $\hat{\mu}_{s_{\alpha\beta}}$ and $\hat{\sigma}_\alpha$, $\hat{\sigma}_\beta$, and $\hat{\sigma}_{s_{\alpha\beta}}$.

Note that the mean and mode of the posterior distribution should be equal if it is normal.

So it is reasonable to use μ to denote both.

If our estimates are accurate, and the normal approximation holds, then we should have the following

$$\frac{\alpha - \hat{\mu}_\alpha}{\hat{\sigma}_\alpha} \sim N(0,1)$$

$$\frac{\beta - \hat{\mu}_\beta}{\hat{\sigma}_\beta} \sim N(0,1)$$

$$\frac{s_{\alpha\beta} - \hat{\mu}_{s_{\alpha\beta}}}{\hat{\sigma}_{s_{\alpha\beta}}} \sim N(0,1)$$

We can check this assumption using an Anderson-Darling test. We store a record of whether the test rejects normality at the .05 significance level. These rejection indicators are r_α, r_β, and $r_{s_{\alpha\beta}}$.

We also generate 95% confidence intervals for each parameter. We will check the coverage of the intervals by recording the fraction of the thinned MCMC samples that are cotained within. This will also check the normal approximation.

Using the methods discussed in the last section, the interval endpoints are

$$l_\alpha = -z_{.025}\hat{\sigma}_\alpha + \hat{\mu}_\alpha$$

$$u_\alpha = z_{.025}\hat{\sigma}_\alpha + \hat{\mu}_\alpha$$

$$l_\beta = -z_{.025}\hat{\sigma}_\beta + \hat{\mu}_\beta$$

$$u_\beta = z_{.025}\hat{\sigma}_\beta + \hat{\mu}_\beta$$

$$l_{s_{\alpha\beta}} = -z_{.025}\hat{\sigma}_{s_{\alpha\beta}} + \hat{\mu}_{s_{\alpha\beta}}$$

$$u_{s_{\alpha\beta}} = z_{.025}\hat{\sigma}_{s_{\alpha\beta}} + \hat{\mu}_{s_{\alpha\beta}}$$

The following function performs the posterior sampling and returns the results. **L** denotes the number of posterior samples to draw.

In [10]:
```
def posterior_results(mode,Cov,stan_model,stan_data,L,rs):
    samptest_grad_fit = stan_model.sampling(stan_data,              \
            iter=3000,chains=1,warmup=1000,init=[mode],seed=rs,check_hmc_diagnos
    summary = pd.DataFrame(samptest_grad_fit.summary()['summary'], \
                           columns=samptest_grad_fit.summary()['summary_colnames
    n_eff = np.min(summary['n_eff'])
    N = samptest_grad_fit.sim["iter"]
    N_skip = np.array([np.ceil(N / n_eff)]).astype(int)[0]
    N_needed = N_skip * L + samptest_grad_fit.sim["warmup"]
    if N_needed > N:
        samptest_grad_fit = stan_model.sampling(stan_data,              \
                iter=N_needed,chains=1,warmup=1000,init=[mode],seed=rs,check_hmc

    df = samptest_grad_fit.extract()
    sampalpha  = np.array(df['alpha'][::(N_skip)])
    sampalpha = sampalpha[0:L]
    sampbeta  = np.array(df['beta'][::(N_skip)])
    sampbeta = sampbeta[0:L]
    sampsum = sampalpha + sampbeta
    meanalpha = np.mean(sampalpha)
    meanbeta = np.mean(sampbeta)
    meansum = np.mean(sampsum)
    sealpha = np.std(sampalpha)
    sebeta = np.std(sampbeta)
    sesum = np.std(sampsum)
    adalpha = spstats.anderson((sampalpha-mode['alpha'])/np.sqrt(Cov[0,0]),'norm
    adalpha = adalpha[1][2] < adalpha[0]
    adbeta = spstats.anderson((sampbeta-mode['beta'])/np.sqrt(Cov[1,1]),'norm')
    adbeta = adbeta[1][2] < adbeta[0]

    sesumhat  = np.sqrt(np.matmul(np.ones((1,2)),np.matmul(Cov,np.ones((2,1)))))
    sumest = (sampsum-  (mode['alpha'] + mode['beta']))/sesumhat
    sumest = sumest.flatten()
    adsum = spstats.anderson(sumest,'norm')
    adsum = adsum[1][2] < adsum[0]

    mean = np.array([meanalpha,meanbeta,meansum])
    se = np.array([sealpha,sebeta,sesum])
    ad = np.array([adalpha,adbeta,adsum])

    z025 = spstats.norm.ppf(1-.025)

    lalpha = -z025*np.sqrt(Cov[0,0]) + mode['alpha']
    ualpha = z025*np.sqrt(Cov[0,0]) + mode['alpha']
    lbeta = -z025*np.sqrt(Cov[1,1]) + mode['beta']
    ubeta = z025*np.sqrt(Cov[1,1]) + mode['beta']
    lsum = -z025*sesumhat + mode['alpha'] + mode['beta']
    usum = z025*sesumhat + mode['alpha'] + mode['beta']
    propalpha = np.mean(np.less_equal(sampalpha,ualpha)*np.greater_equal(sampalp
    propbeta = np.mean(np.less_equal(sampbeta,ubeta)*np.greater_equal(sampbeta,l
    propsum = np.mean(np.less_equal(sampsum,usum)*np.greater_equal(sampsum,lsum)
    prop = np.array([propalpha,propbeta,propsum])
    return {'mu':mean, 'sigma': se, 'reject': ad, 'prop': prop}
```

Now let's perform the simulation. We use an observed data sample size of **n=600** based on **draws=1000** prior draws. For each observed data sample we will use **L=500** posterior draws to evaluate the confidence intervals and posterior normal approximation. We store the results in the **results** matrix.

In [11]:
```
n = 600
draws = 1000
L=500
rs = np.random.RandomState(218409)
results = 0*np.ones((draws,18))
```

In [12]:
```
for i in np.arange(0, draws):
    print(i)
    data = draw(n,rs)
    with suppress_stdout_stderr():
        est_fit = estimate_map(stan_model,data)
        post = posterior_results(est_fit['mode'],est_fit['Cov'],est_fit['stan_mo
    results[i,0]= est_fit['mode']['alpha']
    results[i,1] = post['mu'][0]
    results[i,2] = np.sqrt(est_fit['Cov'][0,0])
    results[i,3] = post['sigma'][0]

    results[i,4]= est_fit['mode']['beta']
    results[i,5] = post['mu'][1]
    results[i,6] = np.sqrt(est_fit['Cov'][1,1])
    results[i,7] = post['sigma'][1]

    results[i,8]= est_fit['mode']['alpha'] + est_fit['mode']['beta']
    results[i,9] = post['mu'][2]
    results[i,10] = np.sqrt(np.matmul(np.ones((1,2)),np.matmul(est_fit['Cov'],np
    results[i,11] = post['sigma'][2]

    results[i,12] = post['reject'][0]
    results[i,13] = post['reject'][1]
    results[i,14] = post['reject'][2]
    results[i,15] = post['prop'][0]
    results[i,16] = post['prop'][1]
    results[i,17] = post['prop'][2]
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

```
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
```

85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145

146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206

207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267

268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328

329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389

390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450

451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511

512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572

573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633

634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694

695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755

```
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
```

817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877

878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938

939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

Now we summarize the results.

In [13]:
```
ResultSummary = pd.DataFrame((np.concatenate(( \
                                np.mean(results,axis=0)[:,np.newaxis], \
                                np.std(results,axis=0)[:,np.newaxis]), \
                                                axis=1)),index=[          \
    "Alpha Mode","Alpha Mean Posterior", "SE Alpha Estimate","Alpha SD Posterior
    "Beta Mode","Beta Mean Posterior", "SE Beta Estimate","Beta SD Posterior", \
    "Sum Mode","Sum Mean Posterior", "SE Sum Estimate","Sum SD Posterior", \
     "Alpha AD Reject Rate", "Beta AD Reject Rate","Sum AD Reject Rate", \
     "Alpha CI Proportion", "Beta CI Proportion", "Sum CI Proportion"], \
                                                columns=['Mean',"SD"])
ResultSummary
```

Out[13]:

	Mean	SD
Alpha Mode	4.170080	3.073904
Alpha Mean Posterior	4.170141	3.073850
SE Alpha Estimate	0.040828	0.000207
Alpha SD Posterior	0.040723	0.001481
Beta Mode	0.996798	1.009497
Beta Mean Posterior	0.996799	1.009507
SE Beta Estimate	0.040869	0.001188
Beta SD Posterior	0.040815	0.001952
Sum Mode	5.166878	3.236201
Sum Mean Posterior	5.166940	3.236160
SE Sum Estimate	0.057663	0.001425
Sum SD Posterior	0.057553	0.002457
Alpha AD Reject Rate	0.058000	0.233743
Beta AD Reject Rate	0.051000	0.219998
Sum AD Reject Rate	0.052000	0.222027
Alpha CI Proportion	0.950476	0.010601
Beta CI Proportion	0.949680	0.010935
Sum CI Proportion	0.950114	0.010711

The modes are close the posterior means on average. The same is true for the posterior standard deviation and estimated standard error.

The rejection rates are close to .05, and the proportions of the samples in the 95% confidence intervals are close to .95 We can perform a hypothesis test of whether the rejection rate is .05 by checking whether .05 is in the confidence interval for the proportion. We can test whether the proportions are .95 in an analogous manner.

In [14]:
```
import statsmodels.stats.proportion as smp
```

```
lower, upper = smp.proportion_confint(np.sum(results[:,12:],axis=0), \
                                      draws, method='beta')
CiResults = pd.DataFrame(np.concatenate((lower[:,np.newaxis], \
                                         upper[:,np.newaxis]),axis=1), \
                                         index=["Alpha AD Reject Rate", "Beta AD
     "Alpha CI Proportion", "Beta CI Proportion", "Sum CI Proportion"], \
                                         columns = ('Lower', 'Upper'))
CiResults
```

Out[14]:

	Lower	Upper
Alpha AD Reject Rate	0.044333	0.074336
Beta AD Reject Rate	0.038205	0.066513
Sum AD Reject Rate	0.039077	0.067635
Alpha CI Proportion	0.935145	0.963078
Beta CI Proportion	0.934250	0.962386
Sum CI Proportion	0.934738	0.962764

.05 is in all the rejection rate intervals. .95 is in the all of the confidence interval proportion intervals.

Our simulation was successful.

In [ ]:

In [ ]:

In [ ]: