Bayesian Statistics with Python, No Resampling Necessary

Charles Lindsey[‡*]

Abstract—TensorFlow Probability is a powerful library for statistical analysis in Python. Using TensorFlow Probability's implementation of Bayesian methods, modelers can incorporate prior information and obtain parameter estimates and a quantified degree of belief in the results. Resampling methods like Markov Chain Monte Carlo can also be used to perform Bayesian analysis. As an alternative, we show how to use numerical optimization to estimate model parameters, and then show how numerical differentiation can be used to get a quantified degree of belief. How to perform simulation in Python to corroborate our results is also demonstrated.

Index Terms—Bayesian statistics, resampling, maximum likelihood, numerical differentiation

Introduction

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 $\boldsymbol{\theta}$ 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 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 numerically stable.

* Corresponding author: charles.lindsey@revionics.com
‡ Revionics, an Aptos Company

Now, as described in section 4.1 of [1] , we can approximate $\ln \pi(\boldsymbol{\theta}|\mathbf{x})$ using a second order Taylor Expansion around $\widehat{\boldsymbol{\theta}}$.

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

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. By proper we mean that the function corresponds to a probability density function. Our asymptotic results are depending on probabilities integrating to 1.

We could get a quantified degree of beief by using resampling methods like Markov chain Monte Carlo (MCMC) [1] directly. We would have to use fewer assumptions. However, resampling can be computationally intensive.

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].

We can perform optimization over the real line by creating unconstrained parameters from the original parameters of interest. 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 $\psi = \log\sigma$. The parameter 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 [2] on g to get the posterior distribution of the constrained parameters.

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

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

We could use Numpy's **matmul** function to multiply the component matrices together. The **inv** function in the **lingalg** library could be used to invert the Hessian. So referring the to the gradient of g as **dg**, the following python code could be used to compute the constrained covariance.

```
np.matmul(
    np.matmul(dg,
              np.linalg.inv(hessian)),
    np.transpose(dg))
```

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 and Delta 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.

This can be accomplished with automatic differentiation [3], which will calculate the derivatives analytically. We can also perform numerical differentiation to get the Hessian and the gradient of the constraint function g. This will be less accurate than an analytic expression, but may be less computationally intensive in large models.

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 we would use.

Numerical Differentiation

So numeric derivatives can be very pragmatic, and flexible. How do you compute them? Are they accurate? We use section 5.7 of [4] as a guide.

The derivative of the function f with respect to x is

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

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 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

$$\begin{aligned} f(x+h) &= f(x) + ((x+h)-x)f'(x) + .5((x+h)-x)^2 f''(\varepsilon) \\ &= f(x) + -hf'(x) + .5h^2 f''(\varepsilon) \end{aligned}$$

where ε is between x and $x+h$.

Now we can rearrange to get

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

The right hand side is the truncation error, ε_t since it's linear in h, the bandwidth we call the this approximation a first order method.

We can do second-order approximations for $f(x+h)$ and $f(x-h)$ and get a more accurate second order method of approximation for $f'(x)$.

$$\begin{aligned} f(x+h) &= f(x) + ((x+h)-x)f'(x) \\ &\quad + \frac{((x+h)-x)^2 f''(x)}{2!} + \frac{((x+h)-x)^3 f'''(\varepsilon_1)}{3!} \\ f(x-h) &= f(x) + ((x-h)-x)f'(x) \\ &\quad + \frac{((x-h)-x)^2 f''(x)}{2!} + \frac{((x-h)-x)^3 f'''(\varepsilon_2)}{3!} \end{aligned}$$

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

Then we have

$$\frac{f(x+h) - f(x-h)}{2h} - f'(x) = h^2 \frac{f'''(\varepsilon_1) + f'''(\varepsilon_2)}{12}$$

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

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.

$$\varepsilon_r \approx \varepsilon_f \frac{|\, f(x)\,|}{h}$$

where ε_f is the fractional accuracy with which f is computed. This is generally machine accuracy. If we are using NumPy [5] this would be

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

Minimizing the roundoff error and truncation error, we obtain

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

where $(f/f''')^{1/3}$ is shorthand for the ratio of $f(x)$ and the sum of $f'''(\varepsilon_1) + f'''(\varepsilon_2)$.

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 this 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.

$$x_c = \mid x \mid$$

Then we would use

$$h = \varepsilon_f^{1/3} \mid x \mid$$

But what if x is 0? This is simple to handle, we just add $\varepsilon_f^{1/3}$ to $x_c = \mid x \mid$

$$h = \varepsilon_f^{1/3} (\mid x \mid + \varepsilon_f^{1/3})$$

Now, [4] also suggests 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 $temp - h$.

In Python, the code would simply be

```
temp = x + h
h = temp - x
```

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! The probability is 0 or 1. A credible interval (Bayesian confidence interval) can meet this criteria.

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

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

Then we have

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

Now, $(\gamma - \hat{\gamma})/\hat{\sigma}_\gamma^2$ 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

$$\begin{aligned} \frac{l-\hat{\gamma}}{\hat{\sigma}_\gamma} &= -z_{\alpha/2} \\ \frac{u-\hat{\gamma}}{\hat{\sigma}_\gamma} &= z_{\alpha/2} \end{aligned}$$

then we have a valid Bayesian confidence interval.

Simple calculation shows that the solutions are

$$\begin{aligned} l &= -z_{\alpha/2}\hat{\sigma}_\gamma + \hat{\gamma} \\ u &= z_{\alpha/2}\hat{\sigma}_\gamma + \hat{\gamma} \end{aligned}$$

The $z_{\alpha/2}$ quantile can be easily generated using **scipy.stats** from SciPy [6]. We would use the **norm.ppf** function.

In Python, we would have

```
z_alpha_2 = scipy.stats.norm.ppf(1-alpha/2)
l = -z_alpha_2*se_gamma_hat + gamma_hat
u = z_alpha_2*nsd_gamma_hat + gamma_hat
```

We can also adjust the intervals for inference on many parameters by using Bonferroni correction [7].

Now we know how to estimate the posterior mode. We also know how to estimate the posterior variance after computing the posterior mode. And we have seen how confidence intervals are made based on this posterior variance, mode, and the normal approximation to the posterior. Let's discuss some tools that will enable us to perform these operations.

TensorFlow Probability

Now we will introduce TensorFlow Probability, a Python library that we can use to perform the methods we have been discussing. TensorFlow Probability is library built using TensorFlow, a leading software library for machine learning and artificial intelligence [8].

TensorFlow Probability is a probabilistic programming language. This lets us build powerful models in a modular way and estimate them automatically. At the heart of TensorFlow Probability is the **Distribution** class. In theory, a probability distribution is the set of rules that govern the likelihood of how a random variable (vector, or even general tensor) takes its values.

In TensorFlow Probability, distribution rules for scalars and vectors are parametrized, and these are expanded for higher dimensions as independent samples. A distribution object corresponds to a random variable or vector. The parts of a Bayesian model can be represented using different distribution objects for the parameters and observed data.

Example Distribution

As an example, let's examine a linear regression with a χ^2 prior for the intercept a and a normal prior for the slope β. Our observed outcome variable is y with a normal distribution and the predictor is x.

$$y_i \sim \text{Normal}(x_i\beta + \alpha, 1)$$

We can store the distribution objects in a dictionary for clear organization. The prior distribution of β is Normal with mean 1

and variance 1, $N(1,1)$. We use the **Normal** distribution subclass to encode its information in our dictionary.

```
tfd = tfp.distributions
dist_dict = {}
dist_dict['beta'] = tfd.Normal(1,1)
```

The β parameter can range over the real line, but the intercept, α should be nonnegative. The **Chi2** distribution sublcass has support on only the nonegative reals. However, if we are performing optimization on the α parameter, we may take a step where it became negative. We can avoid any complications like this if we use a **TransformedDistribution**. Transformed distributions can be used together with a **Bijector** object that represents the transforming function.

For α, we will model an unconstrained parameter, $\alpha^u = \log \alpha$. The natural logarithm can take values over the real line.

```
tfb = tfp.bijectors
dist_dict['unconstrained_alpha'] = \
tfd.TransformedDistribution(tfd.Chi2(4),tfb.Log())
```

We can use the **sample** method on the distribution objects we created to see random realizations. Before we do that we should set the seed, so that we can replicate our work.

```
tf.random.set_seed(132)
sample_ex=dist_dict['unconstrained_alpha'].sample(10)
sample_ex
```

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([ 2.050956  , 0.56120026,  1.8559402,
       -0.05669071, ... ], dtype=float32)>
```

We see that the results are stored in a **tf.Tensor** object. This has an easy interface with NumPy, as you can see by the **numpy** component. We see that the unconstrained α, α^u takes positive and negative values.

We can evaluate the density, or it's natural logarithm using class methods as well. Here is the log density for the sample we just drew.

```
dist_dict['unconstrained_alpha'].log_prob(sample_ex)
```

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([-1.1720479 , -1.1402813 , -0.8732692 ,
       -1.9721189 , ...], dtype=float32)>
```

Now we can get α from α^u by using a callable and the **Deterministic** distribution.

```
dist_dict['alpha'] = \
    lambda unconstrained_alpha: \
        tfd.Deterministic(\
            loc= tfb.Log().inverse( \
                unconstrained_alpha))
```

Now we've added all of the parameters to **dist_dict**. We just need to handle the observed variables y and x. In this example x is **exogenous**, which means it can be treated as fixed and nonrandom in estimating α and β in the model for y. y is **endogenous**, which means it is a response variable in the model, the outcome we are trying to estimate.

We will define x separately from our dictionary of distributions. For the example we have to generate values of x, but once this is done we will treat it as fixed and exogenous

The observed variable x will have a standard normal distribution. We will start by giving it a sample size of 100.

```
n = 100
x_dist = tfd.Normal(tf.zeros(n),1)
x = x_dist.sample()
```

The distribution of y, which would give us the likelihood, can be formulated using a callable function of the parameters and the fixed value of x we just obtained.

```
dist_dict['y'] = \
    lambda alpha, beta: \
        tfd.Normal(loc = alpha + beta*x,scale=1)
```

With a dictionary of distributions and callables indicating their dependencies, we can work with the joint density. This will correspond to the posterior distribution of the model, augmenting the priors with the likelihood.

The **JointDistributionNamed** class takes a dictionary as input and behaves similarly to a regular univariate distribution object. We can take samples, which are returned as dictionaries keyed by the parameter and observed variable names. We can also compute log probabilities, which gives us the posterior density.

```
posterior = tfd.JointDistributionNamed(dist_dict)
```

Now we have a feel for how TensorFlow Probability can be used to store a Bayesian model. We have what we need to start performing optimization and variance estimation.

Maximum A Posteriori (MAP) with SciPy

We can use SciPy's implementation of the Limited memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) [9] algorithm to estimate the posterior mode. This is a Quasi-Newton optimization method that does not need to store the entire Hessian matrix during the algorithm, so it can be very fast. If the Hessian was fully stored we could just use it directly in variance estimation, but it would be slower. We do to take advantage of automatic differentiation to calculate the score function, the first derivative of the posterior. TensorFlow Probability provides this through the **value_and_gradient** function of its **math** library.

We will use **minimize** from the **optimize** SciPy library, which operates on a loss function that takes a vector of parameters as input. We will optimize on **unconstrained_alpha** and **beta**, the unconstrained space parameters of the model. In the joint distribution representation, they are separate tensors. But in optimization, we will need to evaluate a single tensor.

We will use the first utility function from the **bayes_mapvar** library, which will be available with this paper, to accomplish this. The **par_vec_from_dict** function unpacks the tensors in a dictionary into a vector.

Within our loss function, we must move the vector of input parameters back to a dictionary of tensors to be evaluated by TensorFlow probability. The **par_dict_from_vec** function moves the unconstrained parameters back into a dictionary, and the constrained parameters are generated by the **get_constrained** function. Then the posterior density is evaluated by augmenting this dictionary of constrained parameters with the observed endogenoous variables. The get_constrained function is also used to get the final posterior model estimates from the SciPy optimization.

Variance Estimation with SciPy

Once the posterior mode is estimated we can estimate the variance. The first step is calculating the bandwidths. The **get_bandwidths** function handles this.

```
def get_bandwidths(unconstrained_par_vec):
    abspars = abs(unconstrained_par_vec)
    epsdouble = np.finfo(float).eps
```

Statistic	Mean	S.D.
α_{MAP} mean	4.141	2.475
α_{MCMC} mean	3.989	2.765
α_{MAP} S.E.	0.037	0.004
α_{MCMC} S.E.	0.041	0.001
α A.D. Reject	0.042	0.201
β_{MAP} mode	1.013	0.504
β_{MCMC} mean	1.022	1.003
β_{MAP} S.E.	0.029	0.001
β_{MCMC} S.E.	0.041	0.002
β A.D. Reject	0.045	0.207

TABLE 1
Simulation Results, $n_{pre} = 1000$, $n_{post} = 600$, $n_{MCMC} = 500$.

```
epsdouble = epsdouble**(1 / 3)
scale = epsdouble * (abspars + epsdouble)
scaleparmstable = scale + abspars
return scaleparmstable - abspars
```

With the bandwidths calculated, we step through the parameters and create the Hessian and Delta matrices that we need for variance estimation. The **get_hessian_delta_variance** function use numeric differentation to calculate the Hessian, based on numeric derivaties of the automatic derivatives computed by TensorFlow probability for the score function. The Delta matrix is calculated using numeric differentation of the constrained parameter functions.

Simulation

We evaluated our methodology with a simulation based on the α and β parameter setting discussed earlier. This was an investigation into how well we estimated the posterior mode, variance, and distribution using the methods of TensorFlow Probability, SciPy, and bayes_mapvar.

To evaluate the posterior distributions of the parameters we used the MCMC capabilities of TensorFlow Probability. Particulary the the No-U-Turn Sampler [10]. We were careful to thin the sample based on effective sample size so that autocorrelation would not be a problem. This was accomplished using TensorFlow Probability's **effective_sample_size** function from its **mcmc** library.

We drew $n_{pre} = 1000$ observations from the unconstrained prior parameter distribution for α_i and β_i. For each of these prior draws, we drew a posterior sample of $\mathbf{y}_i$ and $\mathbf{x}_i$. $\mathbf{y}_i$ and $\mathbf{x}_i$ were $n_{post} = 600$ samples based on each *alpha_i* and β_i. The posterior mode and variance were estimated, and $n_{MCMC} = 500$ posterior draws from MCMC were made. The mean was used in the MCMC draws since it sould coincide with the mode if our assumptions are correct.

To check the distributional results, we used the Anderson-Darling test [11]. This is given by **anderson** in **scipy.stats**. We stored a record of whether the test rejects normality at the .05 significance level for each of the n_{pre} draws. This test actually checks the mean and variance assumptions as well, since it compares to a standard normal and we are standardizing based on the MAP and **get_hessian_delta_variance** estimates.

The results of the simulation are shown in 1.We use Standard Error (S.E.) to refer to the 1000 estimates of posterior standard deviations from **get_hessian_delta_variance** and the MCMC sample standard deviations. The Standard Deviation (S.D.) column represents the statistics calculated over the 1000 estimates. The standard errors are not far from each other, and neither are the modes and means. The rejection rates for the Anderson Darling test are not far from .05 either.

Statistics	Lower	Upper
α AD Reject	0.030	0.056
β A.D. Reject	0.033	0.060

TABLE 2
A.D. Confidence Intervals, $n_{pre} = 1000$, $n_{post} = 600$, $n_{MCMC} = 500$.

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 will use the **proportion_confint** function from **statsmodels** [12]. In 2, we see that .05 is comfortably within intervals for both parameters. Our simulation successfully corroborated our assumptions about the model and the consistency of our method for estimating the posterior mode, variance, and distribution.

Conclusion

We have explored how Bayesian analysis can be performed without resampling and still obtain full inference. With adequate amounts of the data, the posterior mode can be estimated with numeric optimization and the posterior variance can be estimated with numeric or automatic differentation. The asymptotic normality of the posterior distribution enables simple calculation of posterior probabilities and confidence (credible) intervals as well.

Bayesian methods let us use data from past experience, subject matter expertise, and different levels of certainty to solve data sparsity problems and provide a probabilistic basis for inference. Retail Price Optimization benefits from historical data and different granularities of information. Other fields may also take advantage of access to large amounts of data and be able to use these approximation techniques. These techniques and the tools implementing them can be used by practicioners to make their analysis more efficient and less intimidating.

REFERENCES

[1] A. Gelman, J. Carlin, H. Stern, D. Dunson, A. Vehtari, and D. Rubin, *Bayesian Data Analysis, Third Edition*. Chapman & Hall/CRC Texts in Statistical Science, Taylor & Francis, 2013.

[2] G. W. Oehlert, "A note on the delta method," *The American Statistician*, vol. 46, no. 1, pp. 27–29, 1992.

[3] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," 2018.

[4] W. Press and S. Teukolsky, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Numerical Recipes: The Art of Scientific Computing, Cambridge University Press, 2007.

[5] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," vol. 585, no. 7825, pp. 357–362.

[6] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro,

F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[7] C. Bonferroni, "Teoria statistica delle classi e calcolo delle probabilita," *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze*, vol. 8, pp. 3–62, 1936.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[9] R. Fletcher, *Practical Methods of Optimization*. New York, NY, USA: John Wiley & Sons, second ed., 1987.

[10] M. D. Hoffman and A. Gelman, "The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo," 2011.

[11] M. A. Stephens, "Edf statistics for goodness of fit and some comparisons," *Journal of the American Statistical Association*, vol. 69, pp. 730–737, Sep. 1974.

[12] S. Seabold and J. Perktold, "statsmodels: Econometric and statistical modeling with python," in *9th Python in Science Conference*, 2010.