Standard Operating Procedure For Performing Bayesian Linear Regression in Python

Harry Sullivan

September 2022

1 Purpose and Scope

This document serves as a standard operating procedure (SOP) for performing Bayesian linear regression (BLR). The goals of this document are as follows: users will be familiar with the mathematics and theory behind BLR, users will be able to implement a simple example of BLR in python, and users will be able to take this knowledge and extended it to their own personal modeling situations. In order to use this document successfully users will need to be familiar with gradients and their minimization properties from calculus III. Along with this, a basic understanding of statistics is required. Users are expected to understand the properties of normal distributions and be able to find their mean and variance. Lastly a basic understanding of Python and specifically the packages Numpy and Matplotlib.

2 Materials

This section will cover the installations that are required before proceeding with this SOP.

2.1 Python

To get started, ensure that Jupyter notebook with Python 3 is installed on the machine. This can be easily verified by opening a terminal inside the Jupyter notebook editor and typing "python" and presing enter. Once the Jupyter notebook versions are verified, create a new Python 3 notebook file and open it.

2.2 Imports

Please import the following packages inside the first cell of the notebook.

1. numpy
2. matplotlib.pyplot

3. the matplotlib.pyplot figure class

```
In [1]:
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib.pyplot import figure
```

Figure 1: The imports cell in Jupyter notebook

If these packages are not installed please refer to https://numpy.org/install/ for the Numpy installation, and refer to https://matplotlib.org/stable/users/installing/index.html for the Matplotlib installation.

3 The Outline of the Problem

A common problem in modeling is learning how to choose the correct model parameters. Performing classic optimization can lead to over fitting. Bayesian inference is intended to solve this issue. To put it technically, the goal of Bayesian regression is to calculate the joint probability distribution of the models parameters given some experimental data. All while utilizing a prior probability distribution that will serve as a regularizer.

At this point, readers may be asking themselves the question "What is a prior? And why do I need one to make a guess on the model parameters?" Prior distributions arise from Bayes rule. Bayes rule is simply another way to write the definition of conditional probability. Taking the definition of conditional probability, Bayes rule can be derived as follows.

$$P(A \cap B) = P(B|A)P(A) \tag{1}$$

This formula can be understood from the point of view of sets. Consider the diagram.

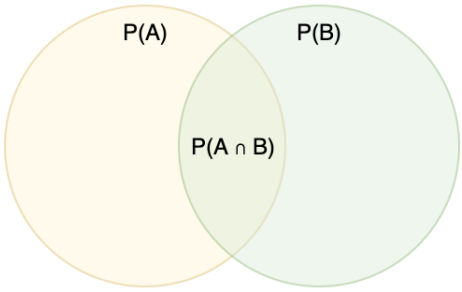

Figure 2: A set theory diagram of the the probability space where two event sets A and B overlap.

This formula is encoding exactly what is happening in the diagram. Choose a point inside A's event space, then calculate A's probability there. With the point known then calculate B's probability at that point. With both of these, just combine the independent events with multiplication as dictated by the axioms of probability. It is know that all the points inside B but not in A have a probability of 0 from the view of A. It is also known that if a point is chosen inside A and not in B, the probability from B's view will be 0. This is exactly extracting the middle section of the Venn diagram. This gives us the intuition that $P(A \cap B)$ can be written two ways.

$$P(A \cap B) = P(B \cap A) = P(A|B)P(B) \tag{2}$$

This shows the following equivalence to be true.

$$P(B|A)P(A) = P(A|B)P(B) \tag{3}$$

$$\implies \frac{P(B|A)P(A)}{P(B)} = P(A|B) \tag{4}$$

$$\tag{5}$$

Each of these terms has a name. $P(A) =$ "Prior", $P(B|A) =$ "Likelihood", $P(B) =$ "Normalization Factor", and $P(A|B) =$ "Posterior". In English, the equation says the prior times the likelihood will give the posterior up to a normalization factor.

If the experimental data is chosen to be B then the quantity $P(A|B)$ represents the probability of the model parameters A given the data B. The quantity $P(A)$ corresponds to the prior probability. It is required in this alternate definition of conditional probability. Throughout this SOP the model in question will be of the form

$$f(x) = \sum_{n=0}^{N} c_n x^n \tag{6}$$

Where the set $\{c_n\}$ is a set of constants that are determined by Bayesian inference. We can prescribe a prior probability distribution on each of these parameters. In other words, the set $\{c_n\}$ corresponds to the A within the Bayes rule formula shown above, and the resulting outputs of $f(x)$ make the set corresponding to B in Bayes rule.

$$\{c_n\} \iff \{P(c_n)\} \tag{7}$$

These distributions are meant to encapsulate the information that is already know about the model. Certain modeling situations require limitations on the

parameters. Utilizing classical minimization of squared error has no way of preventing unreasonable parameters. The use of a prior allows the user to communicate to the optimizer these exact limitations. A physical situation where a limitation is obvious could be when making an inference on a distance. Distances are strictly positive quantities, but the optimizer simply can't know that unless it is told to un-weight those answers. With these prior distributions in tandem with experimental data points Bayes rule can be utilized to make an inference on the set $\{c_n\}$. The conditional probability of the parameters can be written as follows.

$$P(\{c_n\}|\mathbf{y}_{exp}) \propto P(\mathbf{y}_{exp}|\{c_n\}) \prod_n P(c_n) \tag{8}$$

With this equation written, all that is left is to optimize its results to find the best set of parameters. This SOP will outline the exact process of coding such a result. Note the fact that the above expression is not written in vector form, however it can be. In vector form it looks like the following.

$$P(\mathbf{c}_n|\mathbf{y}_{exp}) \propto P(\mathbf{y}_{exp}|\mathbf{c}_n)P(\mathbf{c}_n) \tag{9}$$

Where the bolded lowercase input implies a vector and the bolded uppercase implies a matrix. Instead of a factored prior we now consider a joint distribution as the prior. Note there is not a matrix written in the above equation but there will be one later.

4 Generating the Data

Bayesian inference is meant to encapsulate experimental data's error. This means that this tutorials data must also have some error. This section will serve as a method of generating synthetic data. The assumption taken within this model is that the error ind pendent and identically distributed. Written as a mathematical formula it looks like the following.

$$\mathbf{y}_{\text{exp}} = f(\mathbf{x}) + \boldsymbol{\epsilon} \tag{10}$$

$$\epsilon \sim \mathcal{N}(0, \sigma) \tag{11}$$

Where epsilon is a normal random variable.

4.1

First, create a Jupyter notebook cell and define a function named f with only one input x. This will serve as the true function referenced in equation 6. For this purposes of this SOP. Choose the true function to be

$$f(x) = -0.3 + 0.5x \tag{12}$$

```
In [2]:
def f(x):
    return -0.3 + 0.5*x
```

Figure 3: The proper python implementation of the true function

With these values chosen as the constants c_0 and c_1, it will define a 2 dimensional search space. For each value of c_0 and c_1 an appropriate weight will be assigned based off the posterior equation given above. If the algorithm works, then it should find a maximum probability at $c_0 = -0.3$ and $c_1 = 0.5$.

4.2

To simulate experimental data points, generate a column vector of x values pulled from a uniform distribution in the range $[-1, 1]$. Then find the image of these points by chugging each point through f(x).

```
In [3]:
# Numpy has a built in uniform distribution sampler. The .T converts it into a column vector
x_gt = np.random.uniform(-1,1,20).T
y_gt = f(x_gt) #Numpy will handle the vector inputs automatically
```

Figure 4: How to use Numpy's convenient probabilistic functions.

4.3

The data generated is exact. Experimental data has noise, to encapsulate this feature add some Gaussian noise at each point with a mean of 0 and variance 0.2. An example of this can be seen in the following steps figure.

4.4

Run a 100 point linspace over the range $[-1, 1]$ and evaluate its image under f(x). Then plot both x_{gt} vs y_{exp} and X vs Y.

In [4]:

```
figure(figsize=(8, 6), dpi=80) # This make the figure larger and gives a higher resolution

y_exp = y_gt + np.random.normal(0,0.2,20).T

X = np.linspace(-1,1,100) # This linspace will create 100 evenly spaced points from -1 to 1
Y = f(X)

# The x values are not ordered, implying that a scatter plot must be used
plt.scatter(x_gt,y_exp,label="Synthetic Data",color="green", alpha =0.5) # Alpha corresponds to transparency
plt.plot(X,Y,label="True Function")
plt.show()
```

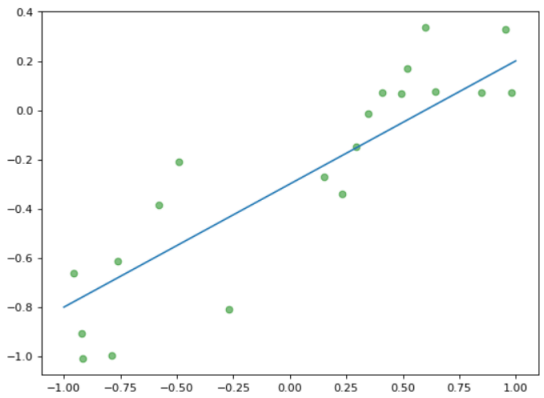

Figure 5: How to add noise to the synthetic data along with an appropriate visualization.

5 How to Create and Sample the Prior Distribution

With the model and the data created, the actual optimization procedure can be started. This section will outline how to create the function $P(\mathbf{c_n})$. Because there is only two variables to make the inference on, only two prior distributions are required. These prior distributions are meant to encapsulate the information the user already knows about the inferred variables. This is done by making the probability of unwanted answers to 0 or make them be quite small. Within the use of this SOP the choice of prior will be a joint distribution between both variables. The function for this joint dist. will be a multivariate normal. This is due to the ease of calculation that the Gaussian form provides.

5.1

Create a python function that represents the formula seen below

$$P(\mathbf{x}) = \frac{1}{\sqrt{\det|2\pi\boldsymbol{\Sigma}|}} \exp(-0.5(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})) \tag{13}$$

Importantly noting the use of bolded and capitalized symbols. An important part of this step is allowing vector inputs as well as vector dot product to the python function. Within python the dot product is done using the @ symbol. Another important aspect is the use of the log-determinant rather than the regular determinant. When using the regular one, floating point and singular matrix errors can arise.

In [5]:
```python
def multivariate_normal (x, μ, Σ):
    sign,logdet = np.linalg.slogdet(Σ*2*np.pi)
    prefactor = 1/(np.sqrt(sign*np.exp(logdet)))
    return prefactor*np.exp((-0.5)*((x - μ).T @ np.linalg.inv(Σ) @ (x - μ)))
```

Figure 6: A vector input multivariate normal function in python

5.2

Now define the prior distributions mean and variance. As an example, say it is known that the parameters that are to be inferred do not often stray far from 0. Implying the correct choice for the mean is $[0, 0]$ and the variance as the identity times 2. With them defined, create a function that will evaluate the prior probability of a vector of $\mathbf{c}_n$.

5.3

Create a grid of points over the range -1 to 1 on both inputs. With this grid evaluate the prior probability on each. Plot the results with a heat map.

In [15]:
```python
figure(figsize=(7, 6), dpi=80)

α = 2 # The prior variance parameter
Σ_prior = np.identity(2)*α
μ_prior = np.array([0,0]) # The prior mean

def prior(c):
    return multivariate_normal(c,μ_prior,Σ_prior)

# Create a grid over the inputs to the prior
c0_range = np.linspace(-1,1,100)
c1_range = np.linspace(-1,1,100)

# Init place to store prior data
prior_c_out = np.zeros((100,100))

# Loop over grid and evaluate prior at each point
for i in range(len(c0_range)):
    for j in range(len(c1_range)):
        prior_c_out[i][j] = prior(np.array([c0_range[i],c1_range[j]]))

# Plot heatmap for prior distribution
plt.contourf(c0_range,c1_range,prior_c_out.T,100)
plt.title("Prior Heatmap")
plt.scatter([-0.3],[0.5],label="Ground Truth",color="red",marker="+")
plt.xlabel("$c_0$")
plt.ylabel("$c_1$")
plt.colorbar()
plt.show()
```

Figure 7: How to implement and plot the prior

5.4

Using the priors mean and variance, sample it using numpy's multivariate normal sampling function. Make 20 draws and plot each one of them on the same plot. At this point, It may be useful to create a function that takes in a set $\{c_n\}$ as well as a vector of x values that returns their linear combination.

In [23]:
```
figure(figsize=(7, 5), dpi=80)

def y(x,c):
    return c[0] + c[1]*x

# Sample from the prior
c_samples = np.random.multivariate_normal(μ_prior, Σ_prior,20)

# Plot samples
for sample in c_samples:
    plt.plot(np.linspace(-1,1,100),y(np.linspace(-1,1,100),sample),color="tab:blue",alpha=0.5)

plt.plot(np.linspace(-1,1,100),f(np.linspace(-1,1,100)),label="True Function",color="red")
plt.legend()
plt.show()
```

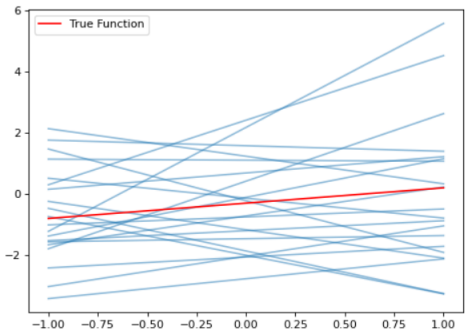

Figure 8: Samples from the prior dist.

Notice how some of these answers are close to the correct function. The likelihood will serve as a way to narrow these answers even further.

6 How to Create The Likelihood and the Posterior

This section will explain why the likelihood looks the way it does as well as how to implement it within the code. This function will vary compared to the prior. It needs to have the experimental data as well as the appropriate inputs for each experimental data point. The role of this function is to evaluate the probability that current model can produce the experimental data. A common metric to grade this is the normal distribution. This will be a great choice for the likelihood because the prior is also normal. It allows for an analytical solution of the maximum probability.

6.1

Most of the work has already been done with respect to the likelihood. All that remains is choosing the variance parameter within the likelihood. For the use of this tutorial, choose $\frac{1}{25}$ times the identity as the variance. This choice reflects the uncertainty in the observed data. In general it is not valid to simply choose a variable here, however inferring it is beyond the scope of this SOP. For the mean, simply set it as the experimental data. Then the input to this normal distribution is the current function evaluations at each input of the experimental data.

$$P(\mathbf{y}_{exp}|\mathbf{c}_n) = \mathcal{N}(\mathbf{\Phi}^T\mathbf{c}_n|\mathbf{y}_{exp}, \boldsymbol{\beta}) \tag{14}$$

$$\boldsymbol{\beta} = \frac{1}{25}\mathbf{I} \tag{15}$$

At this point within the SOP it becomes useful to define the design matrix $\mathbf{\Phi}$. Phi is just another way of writing the inputs of the model. Phi is a M by N matrix where M corresponds to the number of terms in the model polynomial and N is the number of inputs to the model. In other words, N is the number of experimental data points. In our specific case Phi will be:

$$\mathbf{\Phi} = \begin{bmatrix} 1 & 1 & \dots & 1 \\ x_0 & x_1 & \dots & x_n \end{bmatrix} \tag{16}$$

Remembering that $\mathbf{c}_n$ is a M by 1 matrix, allowing us to evaluate every experimental data point with the expression $\mathbf{\Phi}^T\mathbf{c}_n$ which has shape N by 1.

In [8]:
```
def LH_c(c,X,Y):
    model_out = np.array(y(X,c)).reshape(len(X),1)
    return multivariate_normal(model_out, Y, (1/25)*np.identity(len(X)))
```

Figure 9: Likelihood Function Implemented in Python

6.2

The last step is to create the posterior function. Considering the poster equation presented earlier in equation 9, notice how there is a proportionality symbol rather than an equal sign. This is due to the difficult of calculating the normalization factor. This is not an issue because the normalization factor is not a function of the weights $\mathbf{c}_n$. Within python, please make another function with inputs of experimental outputs, experimental inputs, as well as a set of weights.

```
In [9]:
def posterior(c,x_inputs,experimental_data):
    return LH_c(c,x_inputs,experimental_data)*prior(c)
```

Figure 10: Posterior Function Implemented in Python

7 Derivation of Maximum Posterior Estimate

This section will be the most technical section of the SOP. It will involve vector calculus on Gaussian forms. To start, note the fact that Gaussian forms always have a quadratic inside the exponential. If given an exponential of a quadratic form, it is always a Gaussian. This is helpful because of how the posterior is defined. Due to the properties of exponentials, when the prior and posterior are multiplied they will sum within the exponent. This implies that the posterior will have a quadratic form within an exponential, meaning the posterior is also Gaussian.

7.1

A preliminary example is required to prove some properties of quadratic forms in the exponential. Consider the multivariate Gaussian form.

$$\frac{1}{\sqrt{\det|2\pi\boldsymbol{\Sigma}|}}\exp(-0.5(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})) \tag{17}$$

Expand the quadratic inside the exponential.

$$\frac{1}{\sqrt{\det|2\pi\boldsymbol{\Sigma}|}}\exp(-0.5(\mathbf{x}^T\boldsymbol{\Sigma}^{-1}\mathbf{x}-2\mathbf{x}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}+\text{Other terms independent from x}) \tag{18}$$

Looking at the above equation its easy to see, the only thing that can sit between x transpose and x is the sigma matrix. The same can be said for the linear term after it. This allows for the identification of the mean and variance of an exponential quadratic form.

7.2

The next few step will outline the derivation of the posterior. First write the logarithm of the posterior as follows.

$$\log(P(\mathbf{c}_n|\mathbf{y}_{exp})) = \log(P(\mathbf{c}_n)) + \log(P(\mathbf{y}_{exp}|\mathbf{c}_n)) \tag{19}$$

$$= \log\left(\frac{1}{Z}\right) - 0.5(\mathbf{c}_n-\boldsymbol{\mu}_{\text{Prior}})^T\boldsymbol{\Sigma}_{\text{Prior}}^{-1}(\mathbf{c}_n-\boldsymbol{\mu}_{\text{Prior}}) - 0.5(\boldsymbol{\Phi}^T\mathbf{c}_n-\mathbf{y}_{exp})^T\boldsymbol{\beta}^{-1}(\boldsymbol{\Phi}^T\mathbf{c}_n-\mathbf{y}_{exp}) \tag{20}$$

Where the last step is done by substituting the multivariate Gaussian formula for each distribution. Z corresponds to the prefactor in front of the Gaussians that is unnecessary to discuss as it does not depend on $\mathbf{c}_n$. Note the total expression still corresponds to a scalar. If you are ever unsure doing vector calculations, check the shape first.

7.3

Now simplify this equation into a Gaussian form with $\mathbf{c}_n$ as the input.

$$= \log\left(\frac{1}{Z'}\right) - 0.5\left(\mathbf{c}_n^T\mathbf{\Sigma}_{\text{Prior}}^{-\mathbf{1}}\boldsymbol{c}_n - 2\mathbf{c}_n^T\mathbf{\Sigma}_{\text{Prior}}^{-\mathbf{1}}\boldsymbol{\mu}_{\text{Prior}} + \mathbf{c}_n^T\mathbf{\Phi}\boldsymbol{\beta}^{-1}\mathbf{\Phi}^T\mathbf{c}_n - 2\mathbf{c}_n^T\Phi\boldsymbol{\beta}^{-1}\mathbf{y}_{exp}\right) \tag{21}$$

$$= \log\left(\frac{1}{Z'}\right) - 0.5\left(\mathbf{c}_n^T\left(\mathbf{\Sigma}_{\text{Prior}}^{-\mathbf{1}} + \Phi\boldsymbol{\beta}^{-1}\Phi^T\right)\boldsymbol{c}_n - 2\mathbf{c}_n^T\left(\mathbf{\Sigma}_{\text{Prior}}^{-\mathbf{1}}\boldsymbol{\mu}_{\text{Prior}} + \Phi\boldsymbol{\beta}^{-1}\mathbf{y}_{exp}\right)\right) \tag{22}$$

Where any terms that are left out do not depend on $\mathbf{c}_n$ and can be lumped into Z'

7.4

Now the final step is using this form to identify the mean and variance of the posterior. When considering the multivariate Gaussian form, the only possible matrix that can sit between $\mathbf{c}_n^T$ and $\mathbf{c}_n$ is the variance matrix. Allowing us to make the definition:

$$\mathbf{\Sigma}_{\text{Post.}}^{-1} = \mathbf{\Sigma}_{\text{Prior}}^{-\mathbf{1}} + \Phi\boldsymbol{\beta}^{-1}\Phi^T \tag{23}$$

With the uncertainty known, the mean can also be identified with the same trick as follows:

$$\boldsymbol{\mu}_{\text{Post.}} = \mathbf{\Sigma}_{\text{post}}\left(\mathbf{\Sigma}_{\text{Prior}}^{-\mathbf{1}}\boldsymbol{\mu}_{\text{Prior}} + \Phi\boldsymbol{\beta}^{-1}\mathbf{y}_{exp}\right) \tag{24}$$

Because the prior mean is 0 a simplification is possible.

$$\boldsymbol{\mu}_{\text{Post.}} = \mathbf{\Sigma}_{\text{post}}\left(\Phi\boldsymbol{\beta}^{-1}\mathbf{y}_{exp}\right) \tag{25}$$

Due to the properties of Gaussian forms, all we need to identify is the mean and variance to know the whole distribution. Checking the shapes of these quantities, the mean is a M by 1 and the uncertainty is an M by M. This is as expected.

8 Python Implementation of Maximum Posterior Estimate

To calculate the posterior mean and and variance within python simply use the formulas derived in the previous steps. Add a term that will indicate a subset of the data points so you can view how the inference changes when considering less vs more experimental data points.

8.1

Define an integer N that ranges from 1 to 20. Then create a column vector of the first N experimental Y values. Then create a row vector of of their inputted x values (usually should be a column vector as seen within the math above, however matplotlib will get mad if an input a column is used rather than a row). Then define the likelihoods inverse variance matrix. With that completed, create the design matrix Phi. Then code the posterior mean and variance formulas as derived above. Then print them to see how well we inferred the parameters.

In [10]:

```
N = 20 # Change this to see how well the inference does when changing the number of data points.

y_exp_sub = y_exp[:N].reshape(N,1) # Reshape will ensure its a column vector, .T has issues sometimes and reshape
x_gt_sub = x_gt[:N]

β_inv = 25*np.identity(N)

Φ = np.array([x_gt[:N] for i in range(2)])
Φ[0] = np.ones(len(Φ[0]))

Σ_post_inv = Σ_prior + Φ @ β_inv @ Φ.T
Σ_post = np.linalg.inv(Σ_post_inv)
μ_post = Σ_post @ (Φ @ β_inv @ y_exp_sub)

print("Posterior Mean Estimate:\n" + str(μ_post))
print("Posterior Variance Estimate:\n" + str(Σ_post))
```

```
Posterior Mean Estimate:
[[-0.28855544]
 [ 0.47847243]]
Posterior Variance Estimate:
[[0.00201984 0.0004821 ]
 [0.0004821  0.00835896]]
```

Figure 11: Posterior mean and and variance evaluation implemented in python

8.2

This is great and all but how does the actual probability distribution look? To find out, first create a place to store the posterior function values similarly to the prior evaluation done before. Then loop over every c_n value and evaluate the posterior using the function defined above. Then plot its heat map.

In [11]:

```
figure(figsize=(7, 6), dpi=80)

# Init place to store prior data
post_c_out = np.zeros((100,100))

# Loop over grid and evaluate prior at each point
for i in range(len(c0_range)):
    for j in range(len(c1_range)):
        post_c_out[i][j] = posterior(np.array([c0_range[i],c1_range[j]]),x_gt_sub,y_exp_sub)


# Plot heatmap for prior distribution
plt.contourf(c0_range,c1_range,post_c_out.T,100)
plt.title("Posterior Heatmap")
plt.scatter([-0.3],[0.5],label="Ground Truth",color="red",marker="+")
plt.scatter(μ_post[0],μ_post[1],label="MAP Estimate",color="blue",marker="+")
plt.xlabel("$c_0$")
plt.ylabel("$c_1$")
plt.colorbar()
plt.legend()
plt.show()
```

Figure 12: Posterior heat map implemented in python

8.3

We can see how well our Bayesian inference is doing by sampling the posterior. Do this similarly to the prior sampling but replace it with the posterior mean and variance. Then plot the samples as similarly to prior sample plotting. This time also plot the data points considering with a scatter plot.

In [12]:

```
figure(figsize=(7, 5),dpi=80)

# Sample from the posterior
c_samples = np.random.multivariate_normal(np.squeeze(μ_post), Σ_post,20)

# Plot samples
for sample in c_samples:
    plt.plot(np.linspace(-1,1,100),y(np.linspace(-1,1,100),sample),color="tab:blue",alpha=0.5)

plt.plot(np.linspace(-1,1,100),f(np.linspace(-1,1,100)),label="True Function",color="red")
plt.scatter(x_gt[:N],y_exp[:N],label="Synthetic Data",color="green", alpha =0.5) # Alpha corresponds to transparency
plt.legend()
plt.show()
```

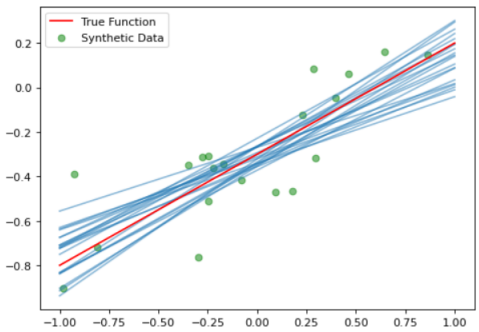

Figure 13: Posterior Predictive Plot in python

Try changing the N variable and re running the last 3 cells to see how the number of data points effects the result.