A3_Response

February 23, 2022

Problem 1

a) Recall Bayesian inference for the Gaussian, in particular that the posterior distribution for the unknown mean is given by $p(\mu|\boldsymbol{x}) \propto p(\boldsymbol{x}|\mu)p(\mu)$, where after completing the square in the exponent we find that $p(\mu|\boldsymbol{x}) = N(\mu|\mu_n, \Sigma_n)$. Additionally, we know the probability density function of the multivariate Gaussian is defined as:

$$N(x|\mu, \Sigma) = \frac{1}{(2\pi)^{d/2}} \frac{1}{|\Sigma|^{1/2}} exp\left\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\}$$

With this knowledge, we can regognize that $p(\boldsymbol{x}|\mu)p(\mu) = c \cdot exp\left\{-\frac{1}{2}(\mu-\mu_n)^T\Sigma_n^{-1}(\mu-\mu_n)\right\}$ (the expressions for μ_n and Σ_n can be found in the question prompt), where the arbitrary constant c abstracts the proportionality of $p(\mu|\boldsymbol{x}) \propto p(\boldsymbol{x}|\mu)p(\mu)$, as well as $p(\mu|\boldsymbol{x})$'s normalization constant.

b) We want to simulate 5 observations from $N(2I, \Sigma) = N(2 \cdot (0\ 1\ 0)^T|\Sigma)$. Before writing the code, let's understand the challenge in this computation, as well as the corresponding solution.

Using the numpy library we are able to generate $\vec{v} \in \mathbb{R}^3$ where each entry comes from the standard normal distribution. This corresponds to generating $\vec{v}$ from a 3-dimensional multivariate Gaussian distribution, $N(\vec{0}, \Sigma)$, where $\Sigma = I_3$. The question, therefore, is how we can transform this into a sample from our desired distribution. We will break the solution into two parts.

Firstly, recall that a linear function of a multivariate Gaussian is also a multivariate Gaussian. In particular, if we let $y = Ax + b$ be a linear transformation of x, where A is any matrix and b is a vector, then

$$y \sim N(A\mu + b, A\Sigma A^T)$$

or rather, y has a Gaussian distribution with mean $A\mu + b$ and covariance matrix $A\Sigma A^T$. Using this knowledge it becomes apparent that to draw samples with our desired mean all we must do is add 2 * $(1\ 1\ 1)^T$ to our original sample. The question now becomes how we construct our desired covariance matrix.

This question leads us to the Cholesky factorization. Taking some symmetrical matrix, B, whose diagonal elements are positive, we can decompose B into $S \times S^T$ where

$$S = \begin{bmatrix} a & 0 & 0 \\ b & c & 0 \\ d & e & f \end{bmatrix}, S^T = \begin{bmatrix} a & b & d \\ 0 & c & e \\ 0 & 0 & f \end{bmatrix}$$

Now, how do we know that we can use this factorization and why will this lead us to our desired Gaussian. Addressing the first question, recall the matrix that we wish to decompose is a covariance matrix, which implies two features. One, it will always be symmetrical. Two, because its diagonal entries represent variances, they will always be positive. Now, re-examine the linear transformation

of a multivariate Gaussian written above. Notice that our new covariance matrix is equal to $A\Sigma A$. Realizing that our original sample has $\Sigma = I_3$, we know our new covariance matrix will actually be equal to AA^T. Therefore, if we find the lower triangular Cholesky factor of our desired covariance matrix, simply multiplying our sample by this factor will lead to our sample being distributed according to a Gaussian with the desired covariance matrix. Let's now begin the actual computation.

[ ]:
```
# Imports
import numpy as np
import plotly.io as pio
import plotly.graph_objects as go
from plotly.subplots import make_subplots
pio.renderers.default = "notebook+pdf"
```

[ ]:
```
def generate_multi_Gaussian(dim, mean_vector, cov_matrix):
    """
    Draws a sample vector distributed according to a multivariate Gaussian␣
 ↪distribution of the user's choosing.
    The Gaussian's covariance matrix is computed using Cholesky Decomposition.

    ...

    Parameters
    ----------
    dim : int
        the dimension of your desired vector (its length)
    mean_vector : numpy.ndarray
        The desired mean vector of the multivariate Gaussian
    cov_matrix : numpy.ndarray
        The desired covariance matrix of the multivariate Gaussian

    Returns
    -------
    A sample vector distributed according to a multivariate Gaussian␣
 ↪distribution whose parameters are chosen by the user
    """
    rand_data = np.random.normal(size = (dim, 1))
    L = np.linalg.cholesky(cov_matrix)
    return np.matmul(L, rand_data) + mean_vector
```

[ ]:
```
def build_post_mean(prior_mean, prior_cov, nature_cov, data):
    n = data.shape[1]
    mean = np.mean(data, axis = 1)
    x_bar = np.array([[mean[0]]])
    for i in range(1, len(mean)):
        to_add = np.array([[mean[i]]])
        x_bar = np.append(x_bar, to_add, axis = 0)
```

```
    inv_term = np.linalg.inv(prior_cov + (1/n)*nature_cov)
    first_term = np.matmul(prior_cov, np.matmul(inv_term, x_bar))
    second_term = (1/n) * np.matmul(nature_cov, np.matmul(inv_term, prior_mean))
    return first_term + second_term
```

[ ]:
```
def build_post_cov_mat(prior_cov, nature_cov, data):
    n = data.shape[1]
    inv_term = np.linalg.inv(prior_cov + (1/n)*nature_cov)
    return np.matmul(prior_cov, np.matmul(inv_term, (1/n) * nature_cov))
```

[ ]:
```
# b) simulation

# Generating 5 observations
sigma = np.array([[1, .3, 0], [.3, .9, .2], [0, .2, .8]])
mean_vector = np.array([[2, 2, 2]]).T
data = generate_multi_Gaussian(3, mean_vector, sigma)
for i in range(4):
    data = np.append(data, generate_multi_Gaussian(3, mean_vector, sigma), axis␣
 ↪= 1)
print("Matrix [x_1 ... x_5] : \n", data, "\n")

# Priors
prior_mean = np.zeros((3, 1))
prior_cov = np.identity(3)

# Generating Posteriors
post_mean = build_post_mean(prior_mean, prior_cov, sigma, data)
post_cov = build_post_cov_mat(prior_cov, sigma, data)
print("Posterior mean vector: \n", post_mean, "\n")
print("Posterior covariance matrix: \n", post_cov, "\n")
```

```
Matrix [x_1 … x_5] :
 [[ 1.80751114  3.54231501  0.62799502  1.35393166  0.76963732]
 [ 2.75707337  2.30625746 -0.07637568  1.1353065   1.74219612]
 [ 4.09689228  1.61599786  3.45503207  1.33196465  1.6944338 ]]

Posterior mean vector:
 [[1.29035784]
 [1.19747713]
 [2.06117677]]

Posterior covariance matrix:
 [[ 0.16454013  0.04253072 -0.00146658]
 [ 0.04253072  0.1493855   0.02933153]
 [-0.00146658  0.02933153  0.1369196 ]]
```

The results for (b) are printed (and labeled) above. I put the five samples into a 3×5 matrix,

where each column is a sample.

c) To re-compute the posterior distribution with our previous posterior as the new prior, all we have to do is pass the (old) posterior data, generated in part b, into the respective "build_post_mean" and "build_post_cov_mat" functions. The results will be labeled and printed in the code output.

[ ]:
```
post_mean2 = build_post_mean(post_mean, post_cov, sigma, data)
post_cov2 = build_post_cov_mat(post_cov, sigma, data)

print("Updated posterior mean vector: \n", post_mean2, "\n")
print("Updated posterior covariance matrix: \n", post_cov2, "\n")
```

```
Updated posterior mean vector:
 [[1.43582013]
 [1.36252976]
 [2.2329755 ]]

Updated posterior covariance matrix:
 [[ 0.09022596  0.02504816 -0.00046385]
 [ 0.02504816  0.08156733  0.01700801]
 [-0.00046385  0.01700801  0.07375911]]
```

d) To compute the new posterior parameters we will simply duplicate the columns of the data matrix from part a and then pass this updated matrix, along with the prior parameters as defined in part a's code, into the respective "build_post_mean" and "buld_post_cov_mat" functions. The results will be labeled and printed in the code output.

[ ]:
```
data_x2 = np.repeat(data, 2, axis = 1)

data_x2_post_mean = build_post_mean(prior_mean, prior_cov, sigma, data_x2)
data_x2_post_cov = build_post_cov_mat(prior_cov, sigma, data_x2)

print("Updated posterior mean vector: \n", data_x2_post_mean, "\n")
print("Updated posterior covariance matrix: \n", data_x2_post_cov, "\n")
```

```
Updated posterior mean vector:
 [[1.43582013]
 [1.36252976]
 [2.2329755 ]]

Updated posterior covariance matrix:
 [[ 0.09022596  0.02504816 -0.00046385]
 [ 0.02504816  0.08156733  0.01700801]
 [-0.00046385  0.01700801  0.07375911]]
```

e) Observing the outputs for **(c)** and **(d)** we can see that the posterior mean vectors and covariance matrices are the same. To use Bishop's words (although not referring to this exact experiment, but equally valid) this experiment "demonstrates the sequential nature of Bayesian learning in which

the current posterior distribution forms the prior when a new data point is observed…[n]ote that this is exactly the same posterior distribution as would be obtained by combining the original prior with the likelihood function for the two data [sets]" (pg. 154).

Problem 2 Firstly, recall that our previous solution methodology, whether using least-squares or solving directly with matrix algebra, resulted in the following formulation to calculate the weights:

$$w = (\Phi^T\Phi)^{-1}\Phi^T y$$

We run into problems, however, if $\Phi^T\Phi$ is singular, or rather, its columns are not linearly independent. When this is the case we are no longer able to compute the inverse of this matrix product, and we must instead approximate the values of w. I will detail two ways to make this approximation, placing more emphasis on the second method.

The first is by using singular value decomposition and what is known as the Moore-Penrose inverse. Singular value decomposition states that for **any** $m \times n$ matrix A, we can decompose A as follows:

$$A = U\Sigma V^T$$

where U is orthogonal, Σ is diagonal, and V is orthogonal. Note that the matrix product $U\Sigma V^T$ is invertible.

Using SVD we can then define the Moore-Penrose inverse $A^\dagger = (U\Sigma V^T)^{-1} = V\Sigma^{-1}U^T$.

Using this methodology, define $\Phi^\dagger = V\Sigma^{-1}U^T$, and left multiply both sides of $y = \Phi w$ by this pseudo-inverse generalization:

$$\begin{aligned}
&\Phi^\dagger(y) = \Phi^\dagger(\Phi w) \\
&\Rightarrow \Phi^\dagger y \approx w \\
&\Rightarrow w \approx V\Sigma^{-1}U^T y
\end{aligned}$$

This linear algebra technique will give a good approximation for our desired weights.

The second, and more widely used, way to approximate our weights is with Bayesian regression. As this assignment states to "show [our] work as if [we] were explaining [our] solution to another student," I will simply provide a high-level overview of this technique and how I'd use it in this instance:

1. Assign a Gaussian prior to the weights.
2. Use Bayes's Rule to multiply $w's$ likelihood function with the assigned Gaussian prior in order to determine a posterior distribution, which is also Gaussian in nature.

Observing the posterior's parameters

$$\begin{aligned}
\boldsymbol{m}_n &= S_N(\boldsymbol{S}_0^{-1}\boldsymbol{m}_0 \ + \ \beta\boldsymbol{\Phi}^T t) \\
\boldsymbol{S}_N^{-1} &= \boldsymbol{S}_0^{-1} + \beta\boldsymbol{\Phi}^T\boldsymbol{\Phi}
\end{aligned}$$

take particular note that all of this matrix algebra is computable. Therefore, while we do not obtain an exact solution for w, we do get an approximation.

Like Bayesian regression, the use of "regularization allows complex models to be trained on data sets of limited size…by limiting the effective model complexity" (Bishop pg. 145) Speaking to ridge regression specifically, firstly recall the solution that it gives:

$$\boldsymbol{w} = (\lambda\boldsymbol{I} + \boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T\boldsymbol{t}$$

Now, taking $\lambda = \frac{\alpha}{\beta}$ where α and β are the precision parameters for two Gaussian distributions with means of zero (for the weight prior and training noise, respectively), we find that the ridge regression approximate solution for the weights:

$$\boldsymbol{w} = (\frac{\alpha}{\beta}\boldsymbol{I} + \boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T\boldsymbol{t}$$

is necessarily equal to the mean of the weights' posterior Gaussian distribution found when using Bayesian linear regression.

Thus, by restricting the weight's linear combination's outcome, we are able to approximate a solution to our underdetermined system. Whether or not we pass through the five observations is dependent upon our hyperparameter solution and the number of weights that we apply the shrinkage to; however, for the instance where our weight approximation is equal to those given by Bayesian regression, we can expect that the polynomial regression model will not fit the observations perfectly (and that is a good thing if we refer back to the bias-variance decomposition).

Problem 3

See graphs below. Parameters used:

$\alpha = 1 \times 10^{-3}$

$s = 0.1$

$\beta = \frac{1}{0.3^2}$

$\mu_0 = \vec{0}$

$S_0 = \alpha^{-1}I$

[ ]:
```
def fun(x):
    return np.sin(2 * np.pi * x)
```

[ ]:
```
def gaussian_basis_function_matrix(X):
    """
    Turns a matrix whose j = 2...m columns are just a repeat of column j = 1␣
↪into a gaussian basis function matrix
    Using hyper-parameter s = 0.1
    """

    s = 0.1
    U = np.array([np.linspace(0, 1, X.shape[1])]).T
    U = np.tile(U, X.shape[0]).T
    power_arr = np.full((X.shape[0], X.shape[1]), 2)
    return np.exp( (-1/(2*(s**2))) * np.power((X - U), power_arr))
```

[ ]:
```
def sinusoidal_simulation(n, S_0, num_Gaussians):
    """
    This function performs the necessary Bayesian regression / predictive␣
↪distribution logic required for
    figures 3.8 and 3.9
```

```
    S_0 is the prior covariance matrix
    num_Gaussians is the number of gaussians desired to fit the data (note an␣
↪additional 1s column will also be added)
    n is the size of the training data set
    """

    to_return = {}

    beta = 1 / (.3**2) # Corresponds with variance = 0.3^2
    x = np.random.random((n, 1)) # Training Points
    t = fun(x) + (1/beta) * np.random.normal(size = (n, 1)) # Target Data

    # Saving training data to plot
    to_return['x'] = x
    to_return['t'] = t

    # Constructing the linear basis function
    X = np.tile(x, num_Gaussians)
    I0 = gaussian_basis_function_matrix(X)
    I0 = np.insert(I0, 0, np.array(np.ones(I0.shape[0])), axis = 1)

    # Computing posterior parameters
    Sn_inv = np.linalg.inv(S_0) + beta * np.matmul(I0.T, I0)
    m_n = beta * np.matmul(np.linalg.inv(Sn_inv), np.matmul(I0.T, t))

    # Computing from predictive distribution
    xx = np.array([np.linspace(0, 1, 100)]).T
    XX = np.tile(xx, num_Gaussians)
    gauss_XX = gaussian_basis_function_matrix(XX)
    gauss_XX = np.insert(gauss_XX, 0, np.array(np.ones(gauss_XX.shape[0])),␣
↪axis = 1)

    # The following matrix multiplication is not how the book defines it, but I␣
↪perform it in this manner as for
    # computing the prediction points we can actually use this matrix product␣
↪(as opposed to the matrix vector
    # products required to gather the variances)
    yy = np.matmul(m_n.T, gauss_XX.T)

    # Computing the variance at each testing point
    yy_vars = []
    for x_i in xx:
        phi = np.tile(np.array([x_i]), num_Gaussians).T
        phi = np.exp( (-1/(2*(0.1**2))) * (phi - np.array([np.linspace(0, 1,␣
↪num_Gaussians)]).T) ** 2 )
        phi = np.insert(phi, 0, [1], axis = 0)
```

```
        phi_var = (1 / beta) + np.matmul(phi.T, np.matmul(np.linalg.
↪inv(Sn_inv), phi))
        yy_vars.append(phi_var.item(0))

    yy_vars = np.array([yy_vars]).T

    # Figure 3.9

    # Drawing Samples from posterior weight distribution using Cholesky␣
↪factorization
    # sampling method implemented previously
    for sample in range(1, 6):
        weights = generate_multi_Gaussian(10, m_n, np.linalg.inv(Sn_inv))
        to_return[f'yy_{sample}'] = np.matmul(gauss_XX, weights)

    # Saving estimation data
    to_return['xx'] = xx
    to_return['yy'] = yy
    to_return['yy_lower'] = yy - np.sqrt(yy_vars)
    to_return['yy_upper'] = yy + np.sqrt(yy_vars)

    return to_return
```

[ ]:
```
# Priors (mu_knot = 0)
alpha_inv = 1 / (1*10**(-3))
S_0 = alpha_inv * np.identity(10)

# Running Simulation

n = [1, 2, 4, 25] # number of generated data points
sims = {}
i = 1
for pts in n:
    sims[f'sim {i}'] = sinusoidal_simulation(pts, S_0, 9)
    i += 1
```

Figure 3.8

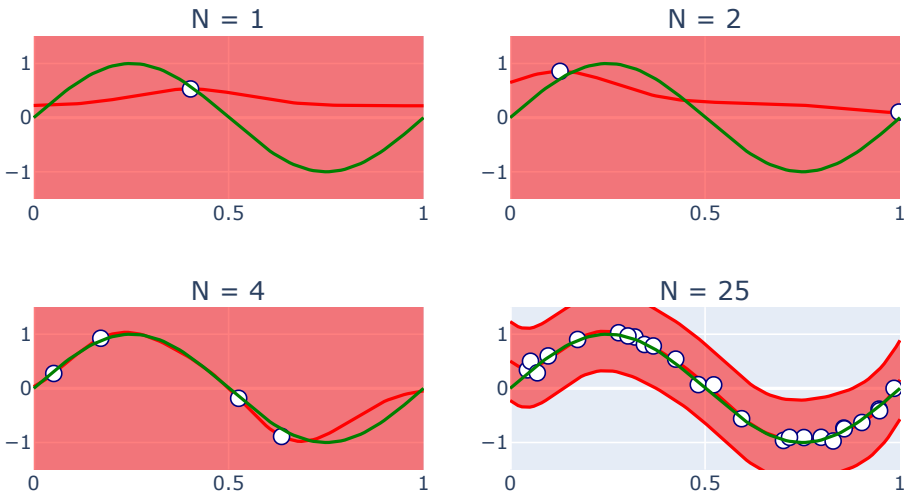
[ ]:
```
fig = make_subplots(
    rows = 2, cols = 2,
    subplot_titles = [
        'N = 1',
        'N = 2',
        'N = 4',
        'N = 25'
    ]
```

```
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 1']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 1']['yy_upper'])).flatten(),
        fill=None,
        mode='lines', line_color='red'
    ),
    row = 1, col = 1)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 1']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 1']['yy_lower'])).flatten(),
    fill='tonexty', # fill area between trace0 and trace1
    mode='lines', line_color='red'
    ),
    row = 1, col = 1)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 1']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 1']['yy'])).flatten(), line=dict(color="red")),
    row = 1, col = 1
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 1']['x'])).flatten(), y = (np.
 ↪asarray(sims['sim 1']['t'])).flatten(), mode = 'markers', marker=dict(
            color='White',
            size=10,
            line=dict(
                color='Navy',
                width=1
            )
        )),
    row = 1, col = 1
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 1']['xx'])).flatten(), y = (np.
 ↪asarray(fun(sims['sim 1']['xx']))).flatten(), line=dict(color="green")),
    row = 1, col = 1
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 2']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 2']['yy_upper'])).flatten(),
```

```
        fill=None,
        mode='lines', line_color='red',
    ),
    row = 1, col = 2)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 2']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 2']['yy_lower'])).flatten(),
    fill='tonexty', # fill area between trace0 and trace1
    mode='lines', line_color='red'
    ),
    row = 1, col = 2)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 2']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 2']['yy'])).flatten(), line=dict(color="red"), ),
    row = 1, col = 2
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 2']['x'])).flatten(), y = (np.
 ↪asarray(sims['sim 2']['t'])).flatten(), mode = 'markers', marker=dict(
            color='White',
            size=10,
            line=dict(
                color='Navy',
                width=1
            )
        ),),
    row = 1, col = 2
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 1']['xx'])).flatten(), y = (np.
 ↪asarray(fun(sims['sim 1']['xx']))).flatten(), line=dict(color="green")),
    row = 1, col = 2,
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 3']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 3']['yy_upper'])).flatten(),
        fill=None,
        mode='lines', line_color='red',
    ),
    row = 2, col = 1)

fig.add_trace(
```

```
    go.Scatter(x = (np.asarray(sims['sim 3']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 3']['yy_lower'])).flatten(),
    fill='tonexty', # fill area between trace0 and trace1
    mode='lines', line_color='red'
    ),
    row = 2, col = 1)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 3']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 3']['yy'])).flatten(), line=dict(color="red")),
    row = 2, col = 1
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 3']['x'])).flatten(), y = (np.
 ↪asarray(sims['sim 3']['t'])).flatten(), mode = 'markers', marker=dict(
            color='White',
            size=10,
            line=dict(
                color='Navy',
                width=1
            )
        ),),
    row = 2, col = 1
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 1']['xx'])).flatten(), y = (np.
 ↪asarray(fun(sims['sim 1']['xx']))).flatten(), line=dict(color="green")),
    row = 2, col = 1,
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 4']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 4']['yy_upper'])).flatten(),
        fill=None,
        mode='lines', line_color='red',
    ),
    row = 2, col = 2)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 4']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 4']['yy_lower'])).flatten(),
    fill='tonexty', # fill area between trace0 and trace1
    mode='lines', line_color='red'
    ),
```

```
    row = 2, col = 2)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 4']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 4']['yy'])).flatten(), line=dict(color="red")),
    row = 2, col = 2
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 4']['x'])).flatten(), y = (np.
 ↪asarray(sims['sim 4']['t'])).flatten(), mode = 'markers', marker=dict(
            color='White',
            size=10,
            line=dict(
                color='Navy',
                width=1
            )
        ),),
    row = 2, col = 2
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 1']['xx'])).flatten(), y = (np.
 ↪asarray(fun(sims['sim 1']['xx']))).flatten(), line=dict(color="green")),
    row = 2, col = 2,
)

fig.update_layout(
    {
        'yaxis' : {'range': [-1.5, 1.5]},
        'yaxis2' : {'range': [-1.5, 1.5]},
        'yaxis3' : {'range': [-1.5, 1.5]},
        'yaxis4' : {'range': [-1.5, 1.5]},
        'xaxis' : {'range': [0, 1]},
        'xaxis2' : {'range': [0, 1]},
        'xaxis3' : {'range': [0, 1]},
        'xaxis4' : {'range': [0, 1]},
        'height' : 750,
        'width' : 1000,
        'showlegend' : False
    }
)
fig.show()
```



For reference:

The green curves correspond to the function $\sin(2\pi x)$ from which the data points were generated (with the addition of Gaussian noise). The data sets of size N are represented by the white/blue circles, the red curve shows the mean of the corresponding Gaussian predictive distribution, and the red shaded region spans one standard deviation either side of the mean.

(Bishop pg. 157)

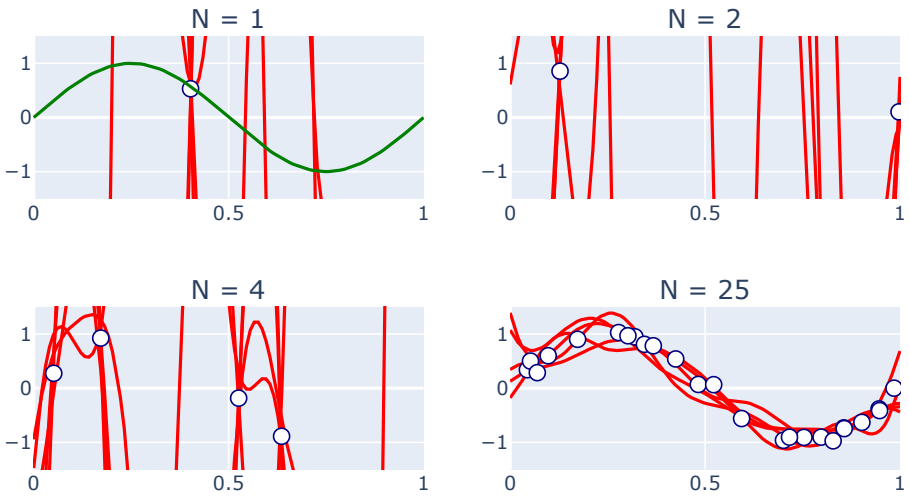
Figure 3.9

[ ]:
```
fig = make_subplots(
    rows = 2, cols = 2,
    subplot_titles = [
        'N = 1',
        'N = 2',
        'N = 4',
        'N = 25'
    ]
)

for i in range(1, 6):
    fig.add_trace(
        go.Scatter(x = (np.asarray(sims['sim 1']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 1'][f'yy_{i}'])).flatten(), line=dict(color="red")),
        row = 1, col = 1
```

```
    )


fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 1']['x'])).flatten(), y = (np.
 ↪asarray(sims['sim 1']['t'])).flatten(), mode = 'markers', marker=dict(
            color='White',
            size=10,
            line=dict(
                color='Navy',
                width=1
            )
        )),
    row = 1, col = 1
)

fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 1']['xx'])).flatten(), y = (np.
 ↪asarray(fun(sims['sim 1']['xx']))).flatten(), line=dict(color="green")),
    row = 1, col = 1
)

for i in range(1, 6):
    fig.add_trace(
        go.Scatter(x = (np.asarray(sims['sim 2']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 2'][f'yy_{i}'])).flatten(), line=dict(color="red")),
        row = 1, col = 2
    )


fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 2']['x'])).flatten(), y = (np.
 ↪asarray(sims['sim 2']['t'])).flatten(), mode = 'markers', marker=dict(
            color='White',
            size=10,
            line=dict(
                color='Navy',
                width=1
            )
        )),
    row = 1, col = 2
)

for i in range(1, 6):
    fig.add_trace(
        go.Scatter(x = (np.asarray(sims['sim 3']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 3'][f'yy_{i}'])).flatten(), line=dict(color="red")),
```

```
        row = 2, col = 1
    )


fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 3']['x'])).flatten(), y = (np.
 ↪asarray(sims['sim 3']['t'])).flatten(), mode = 'markers', marker=dict(
            color='White',
            size=10,
            line=dict(
                color='Navy',
                width=1
            )
        )),
    row = 2, col = 1
)

for i in range(1, 6):
    fig.add_trace(
        go.Scatter(x = (np.asarray(sims['sim 4']['xx'])).flatten(), y = (np.
 ↪asarray(sims['sim 4'][f'yy_{i}'])).flatten(), line=dict(color="red")),
        row = 2, col = 2
    )


fig.add_trace(
    go.Scatter(x = (np.asarray(sims['sim 4']['x'])).flatten(), y = (np.
 ↪asarray(sims['sim 4']['t'])).flatten(), mode = 'markers', marker=dict(
            color='White',
            size=10,
            line=dict(
                color='Navy',
                width=1
            )
        )),
    row = 2, col = 2
)

fig.update_layout(
    {
        'yaxis' : {'range': [-1.5, 1.5]},
        'yaxis2' : {'range': [-1.5, 1.5]},
        'yaxis3' : {'range': [-1.5, 1.5]},
        'yaxis4' : {'range': [-1.5, 1.5]},
        'xaxis' : {'range': [0, 1]},
        'xaxis2' : {'range': [0, 1]},
        'xaxis3' : {'range': [0, 1]},
```

```
        'height' : 750,
        'width' : 1000,
        'xaxis4' : {'range': [0, 1]},
        'showlegend' : False
    }
)
fig.show()
```



Here are plots of the function $y(x, \boldsymbol{w})$ using samples from the posterior distributions over w corresponding to the plots in figure 3.8