Assignment_Two_Response

February 9, 2022

1

Suppose that we have n observations $(x_i, y_i), i = 1, \ldots, n$ and want to fit the model $y = f(x; \boldsymbol{w})$ to these observations. Assuming that the noise in the observations is Gaussian with mean 0 and unknown variance σ^2 we can define the probability of observation y_i occurring as $p(y_i) = N(y_i|f(x_i; \boldsymbol{w}), \sigma^2)$. Recall that for the case of a single real-valued variable x, the Gaussian distribution is defined by:

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

We can use this definition to express $p(y_i)$ as:

$$p(y_i) = N(y_i|f(x_i; \boldsymbol{w}), \sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} exp\left\{-\frac{1}{2\sigma^2}(y_i - f(x_i; \boldsymbol{w}))^2\right\} \tag{1}$$

Therefore, given the training data $\{\boldsymbol{x}, \boldsymbol{y}\}$ and the assumption that our observations were drawn independently from (1), we can determine the maximum likelihood values of the unknown parameters $\boldsymbol{w}$ and σ^2 with the following likelihood function

$$\begin{aligned}
p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w}, \sigma^2) = \prod_{i=1}^{n} p(y_i) = \prod_{i=1}^{n} N(y_i|f(x_i, \boldsymbol{w}), \sigma^2) &= \prod_{i=1}^{n} \frac{1}{(2\pi\sigma^2)^{1/2}} exp\left\{-\frac{1}{2\sigma^2}(y_i - f(x_i; w))^2\right\} \\
&= \left(\frac{1}{(2\pi\sigma^2)^{1/2}}\right)^n \prod_{i=1}^{n} exp\left\{-\frac{1}{2\sigma^2}(y_i - f(x_i; w))^2\right\} \\
&= \left(\frac{1}{(2\pi\sigma^2)^{1/2}}\right)^n \times \; exp\left\{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - f(x_i; \boldsymbol{w}))^2\right\}
\end{aligned}$$

As this prior expression would be difficult to maximize, it is more convenient to maximize the logarithm of the likelihood function. Note that because the logarithm is a monotonically increasing function of its argument, maximizing the log of a function is equivalent to maximizing the function itself.

$$\begin{aligned}
ln\left(p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w}, \sigma^2)\right) &= ln\left(\left(\frac{1}{(2\pi\sigma^2)^{1/2}}\right)^n \times \; exp\left\{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - f(x_i; \boldsymbol{w}))^2\right\}\right) \\
&= ln\left(\left(2\pi * \sigma^2\right)^{-\frac{n}{2}}\right) + ln\left(exp\left\{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - f(x_i; \boldsymbol{w}))^2\right\}\right) \\
&= -\frac{n}{2}ln(2\pi) - \frac{n}{2}ln(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - f(x_i; \boldsymbol{w}))^2
\end{aligned}$$

Finding the maximum likelihood estimate of σ^2 is now simply a matter of differentiating our maximum likelihood function with respect to σ^2 and setting that expression equal to zero.

$$\frac{\partial}{\partial\sigma^2}\, lnp(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{w},\sigma^2) = -\frac{n}{2}\times\frac{1}{\sigma^2} + \frac{1}{2}\times\frac{1}{(\sigma^2)^2}\times\sum_{i=1}^{n}(y_i - f(x_i;\boldsymbol{w}))^2 = 0$$
$$\Rightarrow \left(\frac{1}{2(\sigma^2)^2}\sum_{i=1}^{n}(y_i - f(x_i;\boldsymbol{w}))^2 = \frac{n}{2\sigma^2}\right)\times\frac{2(\sigma^2)^2}{n} \Rightarrow \frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i;\boldsymbol{w}))^2 = \sigma^2$$

Before finalizing our expression we should adjust our notation. Because this is the variance that maximizes the likelihood of our training data we will denote it σ^2_{ML}. Additionally, note that the value of σ^2_{ML} depends on the model's weight coefficients. These are values we do not have. Fortunately, if we were to differentiate our maximum likelihood function with respect to $\boldsymbol{w}$ as opposed to σ^2 we would find an equation for w_{ML}, the weights which maximize the likelihood of our training data occurring given the data follows a Gaussian distribution. Because this expression for w_{ML} does not rely on σ^2, we can simply use w_{ML} instead of w in our equation for σ^2_{ML}. Therefore, we finally conclude that

$$\sigma^2_{ML} = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i;w_{ML}))^2$$

2

2.1

We will now begin question 2. In order to evaluate polynomial regresion we will assume the parametric form $y(x) = sin(2\pi \times x)$, where the test and training data are generated with some level of noise $y(x) = sin(2\pi \times x) + \epsilon$ where $\epsilon \sim N(0, \sigma^2)$. We will vary the value of σ^2 to examine the impact of the noise variance on the test MSE as a function of the polynomial order.

I will now walk through the simulation, providing a conceptual overview at the end with the results of the simulation.

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

The python function "$fun(x)$" will generate the sinusoidal data according to the input scalar (or vector) x, which throughout the report will be generated uniformly in the range $(0, 1)$.

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

Let's quickly recap polynomial regression. Assume we have data $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ and we want to fit it with a k^{th} degree polynomial regression function. We can describe this model with the following matrix equation:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^k \\ 1 & x_2 & x_2^2 & \cdots & x_2^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^k \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_k \end{bmatrix}$$

The following "generate_training_matrix" function handles building the data matrix seen in the above equation.

```
[ ]: def generate_training_matrix(k, n, x):
         columns = []
         columns.append(np.ones(n))
         for i in range(1, k + 1):
             columns.append(x**i)
         X0 = np.matrix(data = columns)
         return X0.T
```

I will quickly summarize the following "poly_regression_mse_analysis" function. Essentially, our goal is to determine if increasing the polynomial degree, k, (we will use $k = 9$ for our testing) of the regression model will lead to a larger or smaller mean squared error. In order to come to a computational conclusion we want to build a large number of models, all trained with new synthetic target data, and compute the MSE for each model.

In order to come to a computational conclusion, it is not sufficient to simply build nine regression models (corresponding to $k = 1$ through $k = 9$) and compare each's MSE_{TE}. Because each model is trained against target data containing a random error term ϵ, it becomes necessary to run many iterations of each model, computing a test MSE for each degree for every sample. The following function houses this simulation.

```
[ ]: def poly_regression_mse_analysis(n, nsamp, k, sigma2):
         """
         Runs a simulation to find the expected mean squarred error for approximating␣
       ↪a sin function with a polynomial line of degree = 1...k


         ...

         Parameters
         ----------
         n : int
             The sample size (# of observations in a sample)
         nsamp : int
             The number of samples
         k : int
             The most flexible curve the simulation will fit
         sigma2 : float
             The noise variance
```

```
    Returns
    -------
    MSE_matrix : A numpy (nsamp X k) matrix whose ith row and jth column contains␣
 ↪the MSE for the ith sample and jth degree polynomial; i = 1...nsamp, j = 1...
 ↪k
    """

    MSE_matrix = np.zeros((nsamp, k))
    delta = 1 / (n+1)
    x = np.arange(delta, 1, delta)

    # Begin Simulation
    for i in range(0, nsamp):

      # Generate Synthetic Training Data
      y = fun(x) + np.sqrt(sigma2)*np.random.normal(size = len(x))
      X0 = generate_training_matrix(k, n, x)

      # Generate Synthetic Testing Data
      x_test = np.linspace(0, 1, 100, endpoint = True)
      y_test = fun(x_test) + np.sqrt(sigma2)*np.random.normal(size = len(x_test))

      # Find weight coefficients for p(x) with degree = deg
      # Use the weights to compute estimate for test data
      for deg in range(1, 10):
          X = X0[:, 0:deg+1]
          w = linalg.lstsq(X, y)[0]
          f = w[0] * np.ones(len(x_test))
          for k in range(1, deg+1):
              f = f + w[k]*(x_test**k)

          # Compute MSE for test data (x_test, fun(x_test) + noise)
          mse = (1/len(y_test))*(np.dot((y_test-f), (y_test-f)))
          MSE_matrix[i, deg - 1] = mse

    return MSE_matrix
```

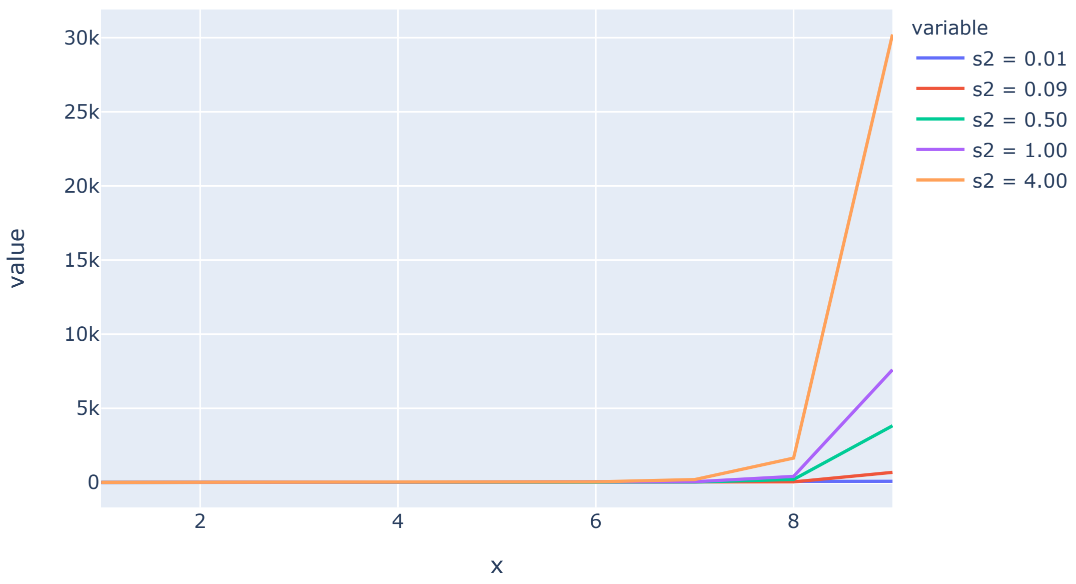
With our simulation built we will now run it with four different noise variances: $\sigma^2 \in \{0.09, 0.5, 1, 4\}$.

```
[ ]: answer1 = poly_regression_mse_analysis(10, 10000, 9, 0.5)
     poly_regression_analysis = pd.DataFrame()
     vars = [0.01, 0.09, 0.5, 1, 4]
     for var in vars:
         poly_regression_analysis[f's2 = {var:.2f}'] =␣
      ↪poly_regression_mse_analysis(10, 10000, 9, var).mean(axis = 0)
     print(poly_regression_analysis)
```

```
# expected_MSE1 = answer1.mean(axis = 0)
# px.line(x = [i for i in range(1, 10)], y = expected_MSE1)
```

```
     s2 = 0.01   s2 = 0.09    s2 = 0.50    s2 = 1.00     s2 = 4.00
0     0.250802    0.349069     0.849841     1.468825      5.135960
1     0.252453    0.364413     0.935672     1.640093      5.804205
2     0.030190    0.163134     0.842348     1.672608      6.648696
3     0.035637    0.214141     1.116673     2.220147      8.837897
4     0.037827    0.338543     1.898412     3.749831     15.020265
5     0.101344    0.896730     5.026083    10.005846     40.131914
6     0.468022    4.240781    23.592212    46.718755    192.870894
7     4.003457   35.697121   201.815860   407.593484   1642.940518
8    76.529628  679.167287  3822.798362  7604.059883  30218.467609
```

```
[ ]: poly_regression_analysis_plot = px.line(poly_regression_analysis, x = [i for i␣
      ↪in range(1, 10)], y = poly_regression_analysis.columns[0::])
     poly_regression_analysis_plot.show()
```



Now, first and foremost, our plots clearly do not follow the desired U-shaped curve. Or rather, the graph does not appear to support the argument that the expected mean squared error decreases and then increases as k (flexibility) increases. We made sure to find $\mathbb{E}[MSE_{TE}]$, not simly a singular MSE_{TE}, for each level of flexility, so this output seems strange. When we also examine the actual values in the table above an answer presents itself. Compare the $\mathbb{E}[MSE_{TE}]$ of $\sigma^2 = 0.01$ and $\sigma^2 = 4$. The former's values drop and then increase, as we expect. The latter's, however, only increase. Of course, we would expect $\mathbb{E}[MSE_{TE}]$ to increase as the variance of the noise term increases, this is the irreducible error. However, looking at our results we can also state that perhaps when we are estimating sinusoidal functions (or at least estimating them with polynomial regression), the error due to variance in the model has a larger impact on the expected test MSE.

3 Question 3.

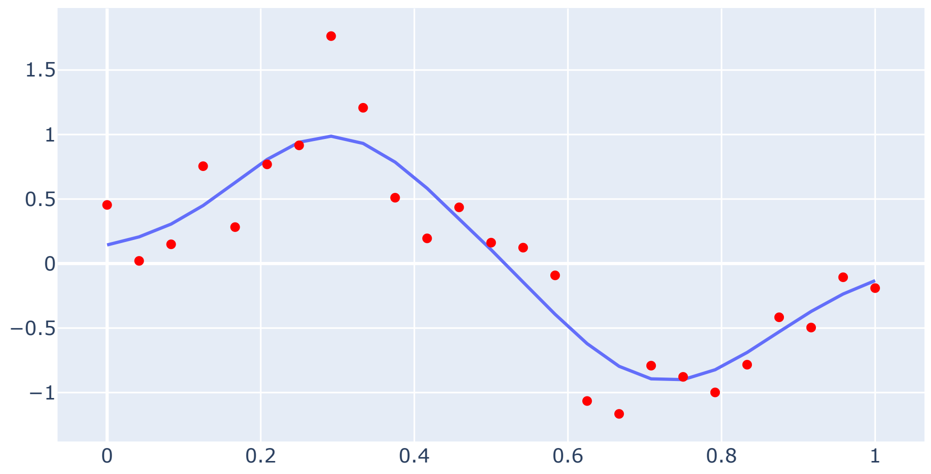
```
[ ]: def gaussian_basis(X, s):
         u = np.linspace(0, 1, X.shape[1], endpoint = True)
         rows = []
         for i in range(0, X.shape[0]):
             rows.append(u)
         U = np.matrix(data = rows)
         power_arr = np.full((X.shape[0], 24), 2)
         return np.exp( (-1/(2*(s**2))) * np.power((X - U), power_arr))
```

```
[ ]: def construct_basis_matrix(num, x):
         columns = []
         for _ in range(0, num):
             columns.append(x)
         basis_matrix = np.matrix(data = columns)
         return basis_matrix.T
```

```
[ ]: x = np.linspace(0, 1, num = 25, endpoint = True)
     Io = construct_basis_matrix(24, x)
     I0 = gaussian_basis(Io, 0.1)
     I0 = np.insert(I0, 0, np.ones(I0.shape[0]), axis = 1)
     I = np.sqrt(0.5)*np.identity(I0.shape[1])
     I0_reg = np.append(I0, I, axis = 0)
     y = fun(x) + 0.3*np.random.normal(size = len(x))
     y_reg = np.append(y, np.zeros(25))
     w = linalg.lstsq(I0_reg, y_reg)
     f = np.matmul(I0, w[0])
```

Before getting into plots involving larger datasets, here is a plot of a single sample.

```
[ ]: fig = px.line(x = x, y = (np.asarray(f)).flatten())
     fig2 = px.scatter(x = x, y = y).update_traces(marker=dict(color='red'))
     fig3 = go.Figure(data = fig.data + fig2.data)
     fig3.show()
     # print(x.shape)
     # print(f[0,:])
```



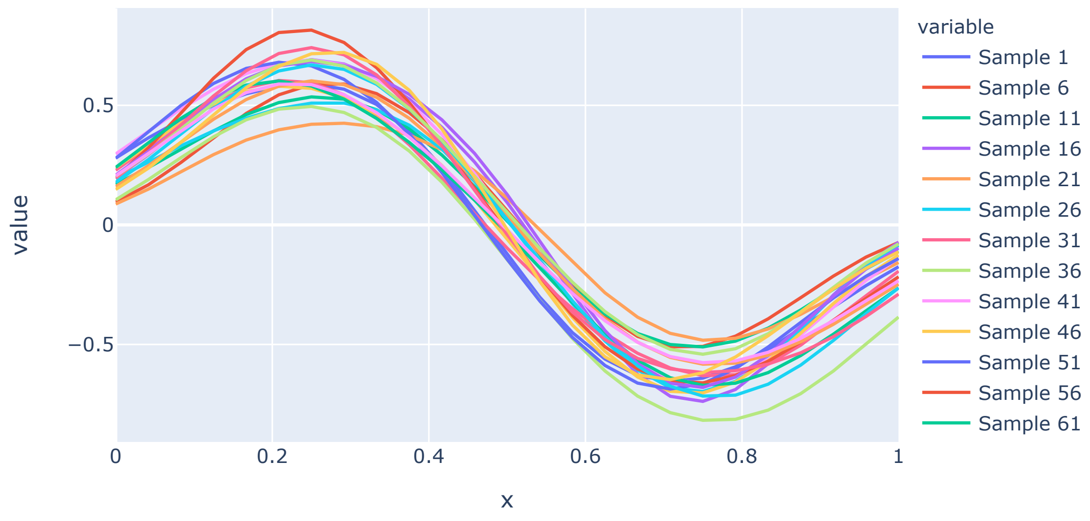
```
[ ]: def linear_basis_regularized_simulation(nsamp, lambd, I0, x_train):

         f_values = pd.DataFrame()
         f_star_values = np.zeros((len(x_train), nsamp))
         f_hat_values = np.zeros((len(x_train), 100))

         x_test = np.linspace(0, 1, num = 100, endpoint = True)
         y_test = fun(x) + 0.3 * np.random.normal(size = len(x_test))

         for n in range(1, nsamp + 1):
             y_train = fun(x_train) + 0.3 * np.random.normal(size = len(x_train))
             f_star_values[:, (n - 1)] = y_train

             # Add regularization
             I = np.sqrt(lambd) * np.identity(I0.shape[1])
             I0_reg = np.append(I0, I, axis = 0)
             y_train_reg = np.append(y_train, np.zeros(I0.shape[1]))

             # Train Model
             w = linalg.lstsq(I0_reg, y_train_reg)

             # Training Predictions
             f_hat = np.matmul(I0, w[0])
             f_values['Sample %d' % n] = (np.asarray(f_hat)).flatten()
```

```
        Io_test = construct_basis_matrix(24, x_test)
        I0_test = gaussian_basis(Io_test, 0.1)
        I0_test = np.insert(I0, 0, np.ones(I0_test.shape(0)), axis = 1)

        # Test Predictions
        f_hat_values[:, (nsamp - 1)] = np.matmul(I0_test, w[0])

    # Finding Ef_i_hat
    expected_f_i_hat = f_hat_values.mean(axis = 1)

    power_matrix = np.full((f_star_values.shape[0], f_star_values.shape[1]), 2)

    f_i_star_sub_Ef_i_hat = f_star_values - np.transpose([expected_f_i_hat] *␣
↪100)
    f_i_star_sub_f_i_hat_2 = np.power(f_i_star_sub_Ef_i_hat, power_matrix)
    expected_f_i_star_sub_f_i_hat2 = f_i_star_sub_f_i_hat_2.mean(axis = 1)
    bias2 = expected_f_i_star_sub_f_i_hat2.mean()

    Ef_i_hat_sub_f_i_hat = np.transpose([expected_f_i_hat] * 100) - f_hat_values
    Ef_i_hat_sub_f_i_hat_2 = np.power(Ef_i_hat_sub_f_i_hat, power_matrix)
    expected_Ef_i_hat_sub_f_i_hat_2 = Ef_i_hat_sub_f_i_hat_2.mean(axis = 1)
    variance = expected_Ef_i_hat_sub_f_i_hat_2.mean()


    # print(f_i_star_sub_f_i_hat.shape)
    # expected_mse_i = np.power((f_star_values - np.
↪transpose([expected_f_i_hat] * 100)), power_matrix)
    # expected_mse_i = expected_mse_i.mean(axis = 1)
    # print(expected_mse_i.mean())

    toReturn = {'f_values' : f_values, 'bias squared' : bias2, 'variance' :␣
↪variance}

    return toReturn
```

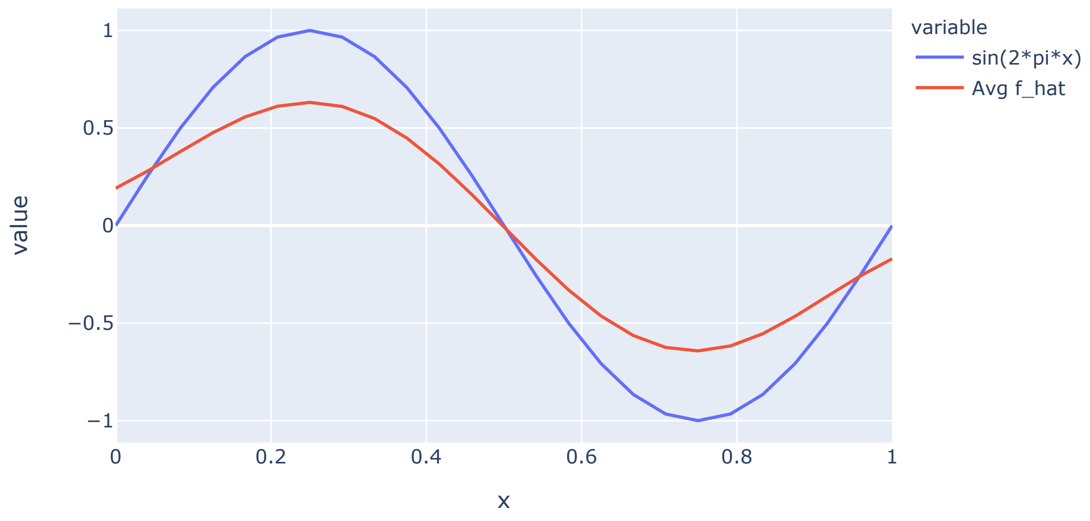
```
[ ]: x_train = np.linspace(0, 1, num = 25, endpoint = True)
     Io = construct_basis_matrix(24, x_train)
     I0 = gaussian_basis(Io, 0.1)
     I0 = np.insert(I0, 0, np.ones(I0.shape[0]), axis = 1)

     df_lambda2 = linear_basis_regularized_simulation(100, np.exp(2.6), I0,␣
     ↪x_train)['f_values']
     df_lambda3 = linear_basis_regularized_simulation(100, np.exp(-0.31), I0,␣
     ↪x_train)['f_values']
     df_lambda4 = linear_basis_regularized_simulation(100, np.exp(-2.4), I0,␣
     ↪x_train)['f_values']
```

```
l2_plot = px.line(df_lambda2, x = x_train, y = df_lambda2.columns[0:100:5], ␣
 ↪title = 'ln(lambda) = 2.6')
df_lambda2_avg = pd.DataFrame()
df_lambda2_avg['sin(2*pi*x)'] = fun(x_train)
df_lambda2_avg['Avg f_hat'] = df_lambda2.iloc[:, 0:100].mean(axis = 1)
l2_avg_plot = px.line(df_lambda2_avg, x = x_train, y = df_lambda2_avg.columns[0:
 ↪2], title = 'ln(lambda) = 2.6')
l2_plot.show()
l2_avg_plot.show()

l3_plot = px.line(df_lambda3, x = x_train, y = df_lambda3.columns[0:100:5], ␣
 ↪title = 'ln(lambda) = -0.31')
df_lambda3_avg = pd.DataFrame()
df_lambda3_avg['sin(2*pi*x)'] = fun(x_train)
df_lambda3_avg['Avg f_hat'] = df_lambda3.iloc[:, 0:100].mean(axis = 1)
l3_avg_plot = px.line(df_lambda3_avg, x = x_train, y = df_lambda3_avg.columns[0:
 ↪2], title = 'ln(lambda) = -0.31')
l3_plot.show()
l3_avg_plot.show()

l4_plot = px.line(df_lambda4, x = x_train, y = df_lambda4.columns[0:100:5], ␣
 ↪title = 'ln(lambda) = -2.4')
df_lambda4_avg = pd.DataFrame()
df_lambda4_avg['sin(2*pi*x)'] = fun(x_train)
df_lambda4_avg['Avg f_hat'] = df_lambda4.iloc[:, 0:100].mean(axis = 1)
l4_avg_plot = px.line(df_lambda4_avg, x = x_train, y = df_lambda4_avg.columns[0:
 ↪2], title = 'ln(lambda) = -2.4')
l4_plot.show()
l4_avg_plot.show()
```

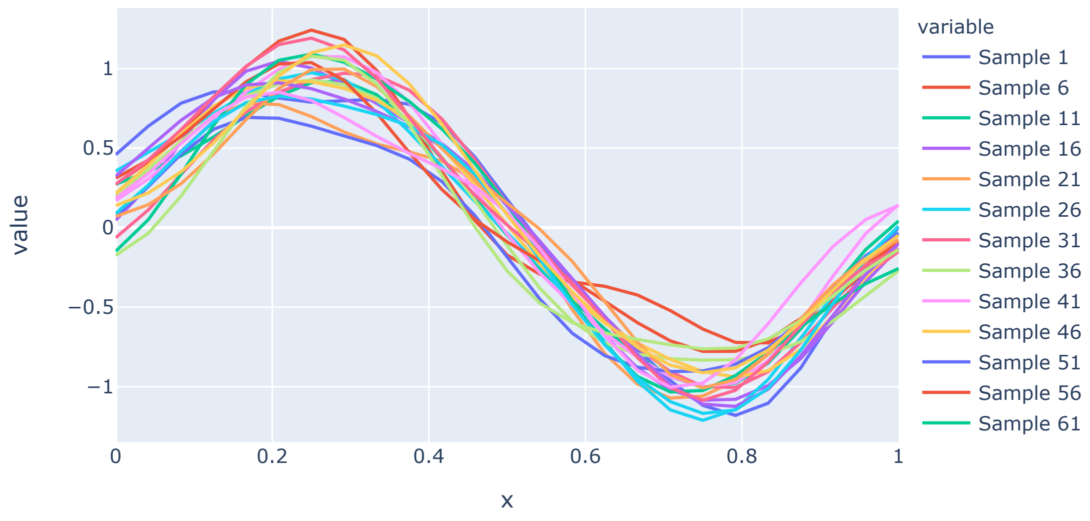
ln(lambda) = 2.6



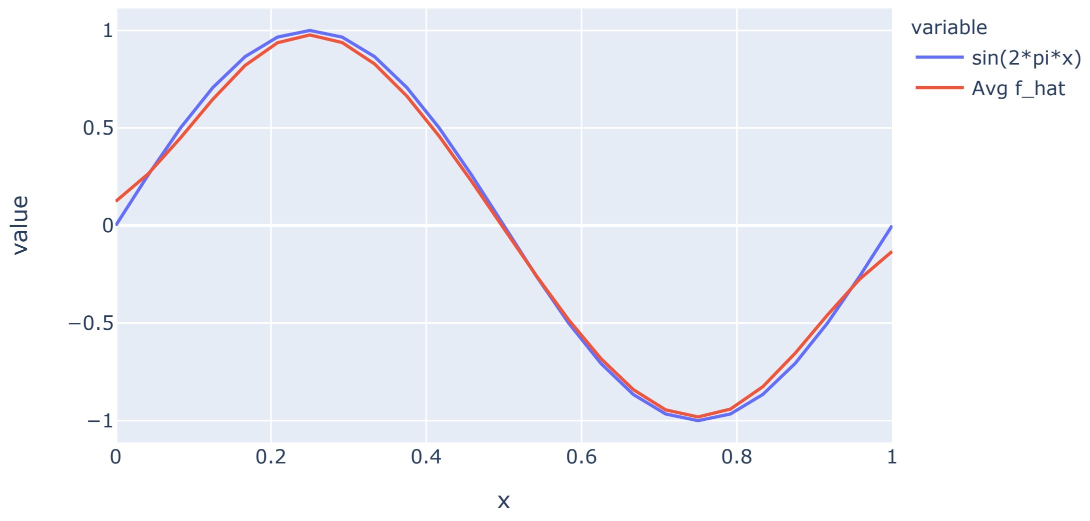
ln(lambda) = 2.6

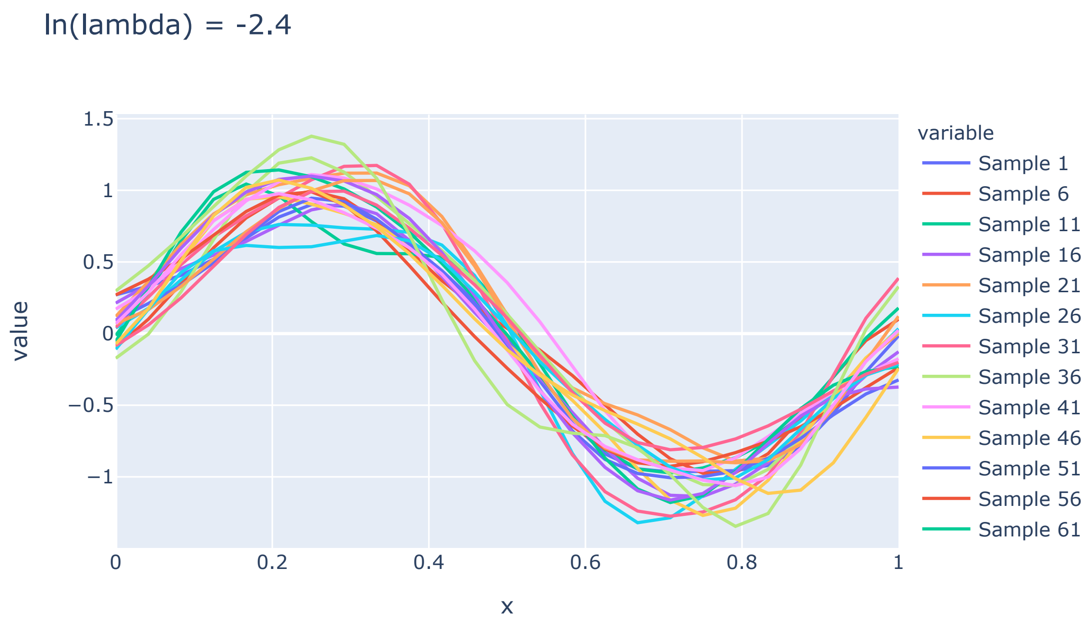


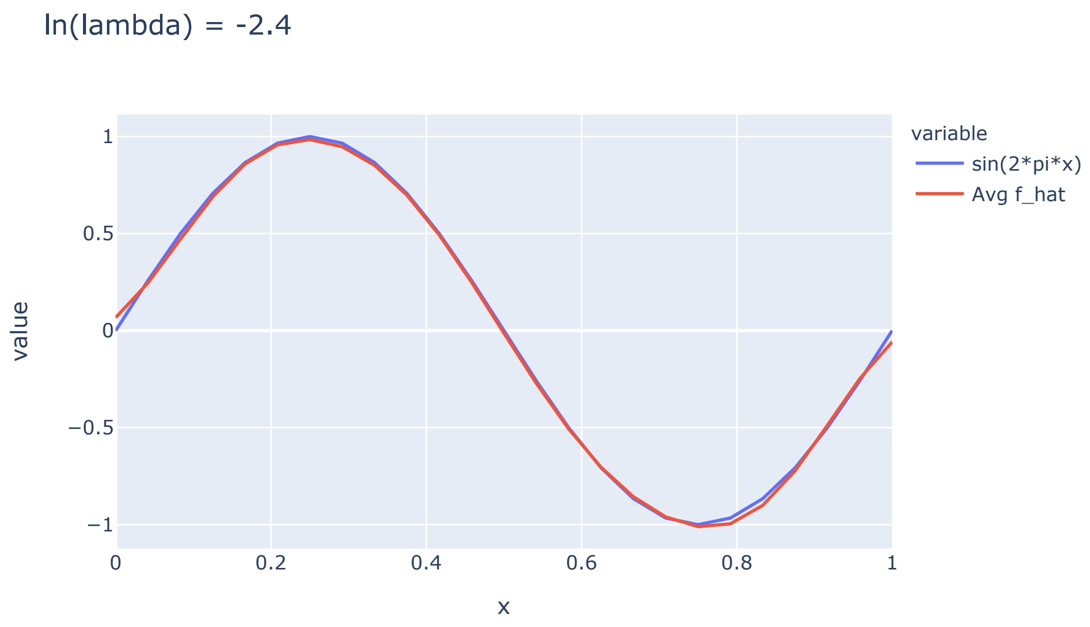
ln(lambda) = -0.31



ln(lambda) = -0.31







Let's examine the above output. Looking at the first graph of 20 samples where $ln\lambda = 2.6$ We see that the prediction functions of the twenty samples are very precise, or rather, the variance between them is small. However, when we examine the corresponding graph of the average of the 100 samples it becomes apparent that our expected regression curve does a poor job estimating the target data. This is clearly an example of the bias-variance trade off. Adding the weight decay term of $ln\lambda = 2.6$ (utilizing ridge regression) forces the weights to be very small, thus guarunteeing low variance between models. However, the models' ability to hit more distanced points decreases, thus increasing the error due to bias.

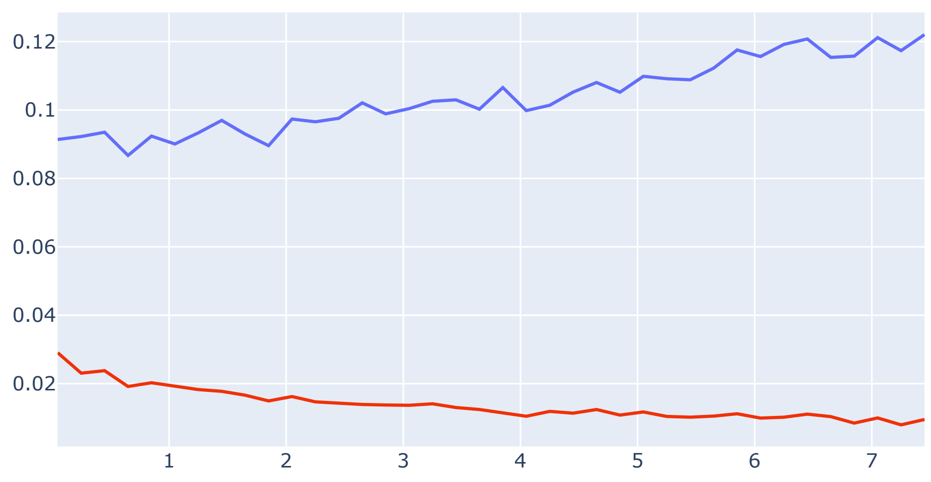
Observing the graphs corresponding to $ln\lambda = -2.4$ we see the opposite effect: the twenty samples vary a good deal from each other, however, they do a very good job of approximating the target data (up to the point where they overfit the training data, causing the test MSE to increase).

The s term governs the spatial scale of the Gaussian basis functions. When s is bigger, the denominator grows, shrinking the power. An exponential raised to a negative power gets bigger when the power term is smaller. Therefore, when s is bigger, so are the basis functions.

4 Question 4

[ ]:
```
lambdas = np.arange(np.exp(-3), np.exp(2) + 0.2, 0.2)
bias2_var_df = pd.DataFrame()
bias2_list = []
var_list = []
for lambd in lambdas:
    value = linear_basis_regularized_simulation(100, lambd, I0, x_train)
    bias2_list.append(value['bias squared'])
    var_list.append(value['variance'])
bias2_var_df['Bias Squared'] = bias2_list
bias2_var_df['Variance'] = var_list
```

[ ]:
```
fig = px.line(bias2_var_df, x = lambdas, y = 'Bias Squared')
fig2 = px.line(bias2_var_df, x = lambdas, y = 'Variance')
fig2['data'][0]['line']['color']='rgb(240, 49, 8)'
fig3 = go.Figure(data = fig.data + fig2.data)
fig3.show()
```



Note that in the above graph the red line is the variance, while the blue line is the bias^2 term.

For the case where $\epsilon \sim N(0, 0.3^2)$ the weight decay term that minimizes the test MSE is $\lambda = 0.7334469562242892$, or rather, when $\ln \lambda = -0.31$. Looking at the plot of test error and $(bias)^2+$ variance in Bishop we can also make a more general conclusion. We can feel confident that the test error is at a minimum when the $(bias)^2+$ variance is at a minimum. Or, when the expected error due to $bias^2$ is equal to the expected error due to variance (both as a function of the weight decay term).