Assignment I

Linear Regression

Generating Synthetic Data

This assignment shows how we can extend ordinary least squares regression, which uses the hypothesis class of linear regression functions, to non-linear regression functions modeled using polynomial basis functions and radial basis functions. The function we want to fit is $y_{\text{true}} = f_{\text{true}}(x) = 6(\sin(x+2) + \sin(2x+4))$. This is a **univariate function** as it has only one input variable. First, we generate synthetic input (data) x_i by sampling $n = 750$ points from a uniform distribution on the interval $[-7.5, 7.5]$.

In [309…

```
# The true function
def f_true(x):
  y = 6.0 * (np.sin(x + 2) + np.sin(2*x + 4))
  return y
```

We can generate a synthetic data set, with Gaussian noise.

In [310…

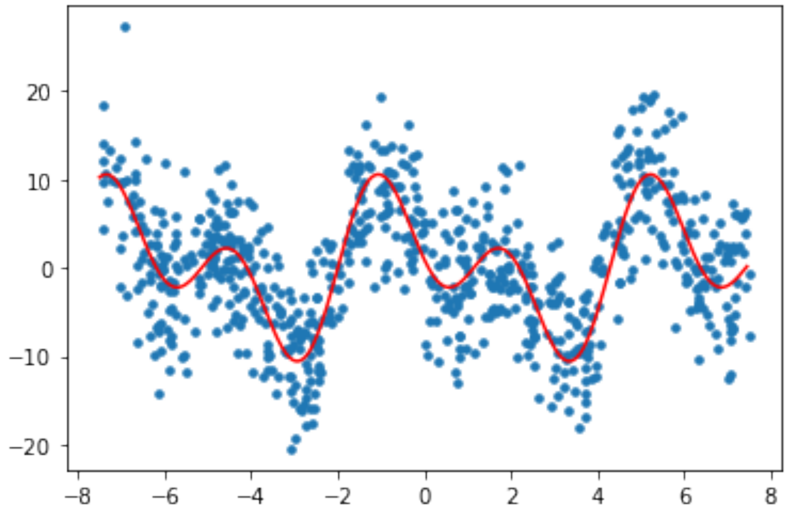
```
import numpy as np                        # For all our math needs
n = 750                                   # Number of data points
X = np.random.uniform(-7.5, 7.5, n)       # Training examples, in one dimensio
e = np.random.normal(0.0, 5.0, n)         # Random Gaussian noise
y = f_true(X) + e                         # True labels with noise
```

Now, we plot the raw data as well as the true function (without noise).

In [311…

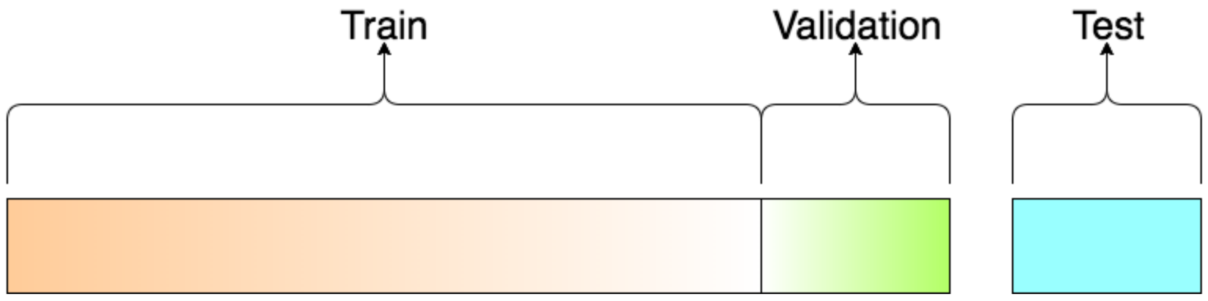
```
import matplotlib.pyplot as plt           # For all our plotting needs
plt.figure()

# Plot the data
plt.scatter(X, y, 12, marker='o')

# Plot the true function, which is really "unknown"
x_true = np.arange(-7.5, 7.5, 0.05)
y_true = f_true(x_true)
plt.plot(x_true, y_true, marker='None', color='r')
```

```
Out[311… [<matplotlib.lines.Line2D at 0x13010aca0>]
```



Recall that we want to build a model to **generalize well on future data**, and in order to generalize well on future data, we need to pick a model that trade-off well between fit and complexity (that is, bias and variance). We randomly split the overall data set ($\mathcal{D}$) into three subsets:

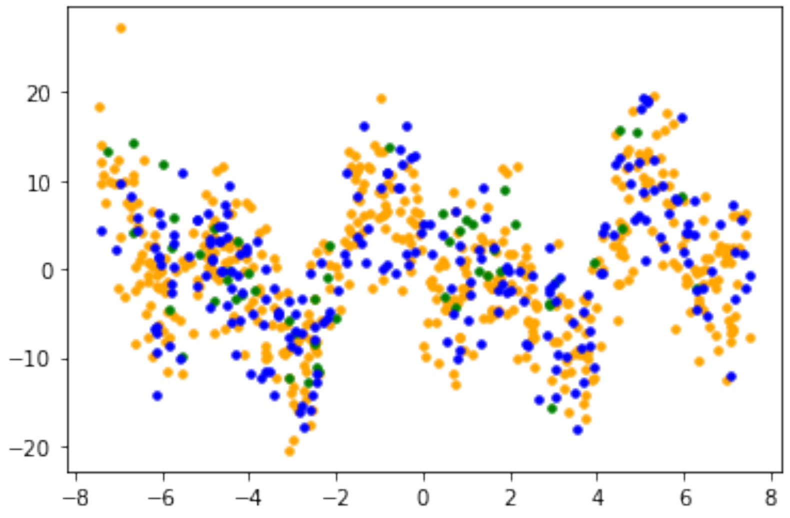
- **Training set**: $\mathcal{D}_{\text{trn}}$ consists of the actual training examples that will be used to **train the model**;
- **Validation set**: $\mathcal{D}_{\text{val}}$ consists of validation examples that will be used to **tune model hyperparameters** (such as $\lambda > 0$ in ridge regression) in order to find the best trade-off between fit and complexity (that is, the value of λ that produces the best model);
- **Test set**: $\mathcal{D}_{\text{tst}}$ consists of test examples to **estimate how the model will perform on future data**.



For this example, let us randomly partition the data into three non-intersecting sets: $\mathcal{D}_{\text{trn}} = 60\%$ of $\mathcal{D}$, $\mathcal{D}_{\text{val}} = 10\%$ of $\mathcal{D}$ and $\mathcal{D}_{\text{tst}} = 30\%$ of $\mathcal{D}$.

In [312…
```
# scikit-learn has many tools and utilities for model selection
from sklearn.model_selection import train_test_split
tst_frac = 0.3  # Fraction of examples to sample for the test set
val_frac = 0.1  # Fraction of examples to sample for the validation set

# First, we use train_test_split to partition (X, y) into training and test s
X_trn, X_tst, y_trn, y_tst = train_test_split(X, y, test_size=tst_frac, rando

# Next, we use train_test_split to further partition (X_trn, y_trn) into trai
X_trn, X_val, y_trn, y_val = train_test_split(X_trn, y_trn, test_size=val_fra

# Plot the three subsets
plt.figure()
plt.scatter(X_trn, y_trn, 12, marker='o', color='orange')
plt.scatter(X_val, y_val, 12, marker='o', color='green')
plt.scatter(X_tst, y_tst, 12, marker='o', color='blue')
```

Out[312…
```
<matplotlib.collections.PathCollection at 0x130fc8850>
```



1. **Regression with Polynomial Basis Functions**, 30 points.

This problem extends **ordinary least squares regression**, which uses the hypothesis class of *linear regression functions*, to *non-linear regression functions* modeled using **polynomial basis functions**. In order to learn nonlinear models using linear regression, we have to explicitly **transform the data** into a higher-dimensional space. The nonlinear hypothesis class we will consider is the set of d-degree polynomials of the form $f(x) = w_0 + w_1 x + w_2 x^2 + \ldots + w_d x^d$ or **a linear combination of polynomial basis function**:

$$f(x) = [w_0,\, w_1,\, w_2 \ldots, w_d]^T \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^d \end{bmatrix}.$$

The monomials $\{1,\, x,\, x^2,\, \ldots,\, x^d\}$ are called **basis functions**, and each basis function x^k has a corresponding weight w_k associated with it, for all $k = 1, \ldots, d$. We transform each univariate data point x_i into into a multivariate (d-dimensional) data point via $\phi(x_i) \rightarrow [1,\, x_i,\, x_i^2,\, \ldots,\, x_i^d]$. When this transformation is applied to every data point, it produces the **Vandermonde matrix**:

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^d \\ 1 & x_2 & x_2^2 & \ldots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^d \end{bmatrix}.$$

---

a. (10 points)

Complete the Python function below that takes univariate data as input and computes a Vandermonde matrix of dimension d. This transforms one-dimensional data into d-dimensional data in terms of the polynomial basis and allows us to model regression using a d-degree polynomial.

In [313…

```python
# X float(n, ): univariate data
# d int: degree of polynomial
def polynomial_transform(X, d):
    # convert data to np.array
    X = np.array(X)

    # power univariate data to d-th and transpose
    return np.transpose(np.array([np.power(X, i) for i in range(d + 1)]))
```

---

b. (10 points)

Complete the Python function below that takes a Vandermonde matrix Φ and the labels $\mathbf{y}$ as input and learns weights via **ordinary least squares regression**. Specifically, given a Vandermonde matrix Φ, implement the computation of $\mathbf{w} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{y}$. *Remember that in Python, @ performs matrix multiplication, while * performs element-wise multiplication. Alternately, numpy.dot also performs matrix multiplication.*

In [314…

```
# Phi float(n, d): transformed data
# y   float(n,  ): labels
def train_model(Phi, y):
    Phi_t = np.transpose(Phi) # transpose phi
    # comptue weight
    return np.dot(np.dot(np.linalg.inv(np.dot(Phi_t, Phi)),Phi_t), y)
```

---

c. (5 points)

Complete the Python function below that takes a Vandermonde matrix Φ, corresponding labels $\mathbf{y}$, and a linear regression model $\mathbf{w}$ as input and evaluates the model using **mean squared error**. That is, $\epsilon_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \Phi_i)^2$.

In [315…

```
# Phi float(n, d): transformed data
# y   float(n,  ): labels
# w   float(d,  ): linear regression model
def evaluate_model(Phi, y, w):
    # wT@phi -> converted to phi@w
    # this is due to the piece-wise subtraction
    return np.sum(np.power((y - np.dot(Phi, w)), 2)) / len(y)
```

---

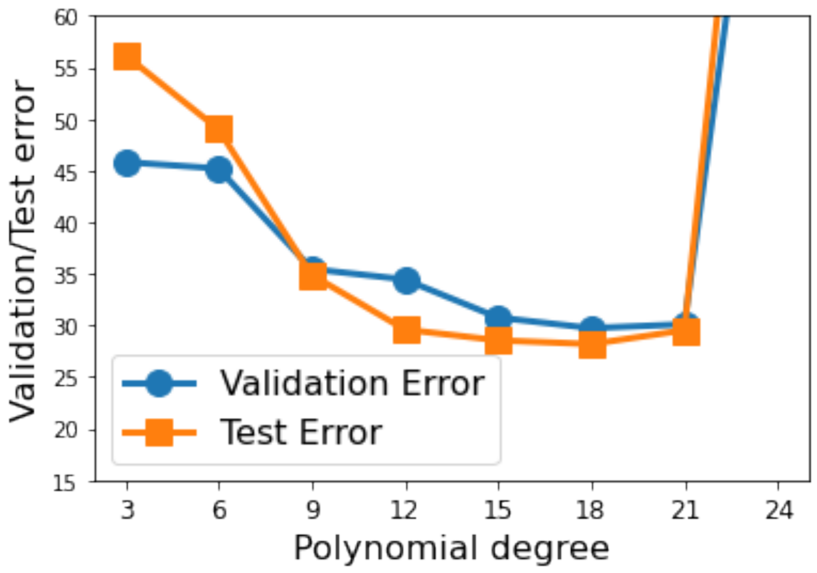
d. (5 points, **Discussion**)

We can explore the **effect of complexity** by varying $d = 3, 6, 9, \cdots, 24$ to steadily increase the non-linearity of the models. For each model, we train using the transformed training data (Φ, whose dimension increases) and evaluate its performance on the transformed validation data and estimate what our future accuracy will be using the test data.

From plot of d vs. validation error below, which choice of d do you expect will generalize best?

In [316…
```
w = {}                  # Dictionary to store all the trained models
validationErr = {}      # Validation error of the models
testErr = {}            # Test error of all the models

for d in range(3, 25, 3):  # Iterate over polynomial degree
    Phi_trn = polynomial_transform(X_trn, d)                 # Transform trai
    w[d] = train_model(Phi_trn, y_trn)                       # Learn model on
    Phi_val = polynomial_transform(X_val, d)                 # Transform vali
    validationErr[d] = evaluate_model(Phi_val, y_val, w[d])  # Evaluate model

    Phi_tst = polynomial_transform(X_tst, d)           # Transform test data
    testErr[d] = evaluate_model(Phi_tst, y_tst, w[d])  # Evaluate model on te

# Plot all the models
plt.figure()
plt.plot(validationErr.keys(), validationErr.values(), marker='o', linewidth=
plt.plot(testErr.keys(), testErr.values(), marker='s', linewidth=3, markersiz
plt.xlabel('Polynomial degree', fontsize=16)
plt.ylabel('Validation/Test error', fontsize=16)
plt.xticks(list(validationErr.keys()), fontsize=12)
plt.legend(['Validation Error', 'Test Error'], fontsize=16)
plt.axis([2, 25, 15, 60])
```

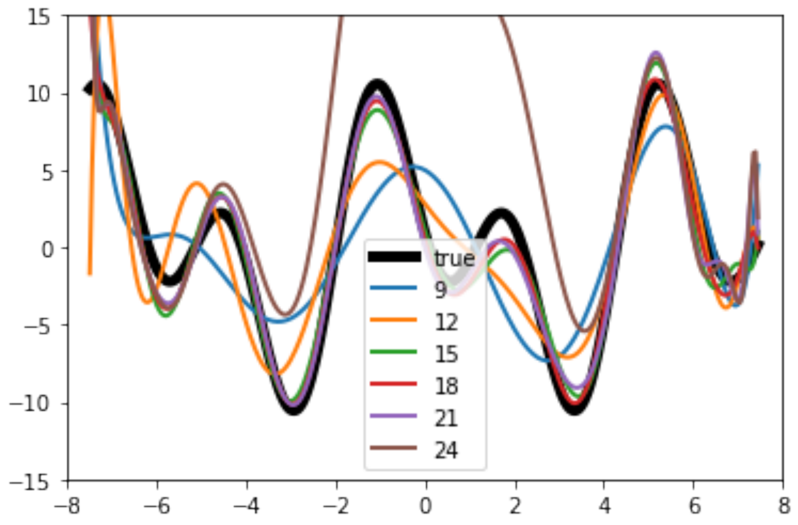
Out[316…
```
(2.0, 25.0, 15.0, 60.0)
```



Finally, let's visualize each learned model.

In [317…
```
plt.figure()
plt.plot(x_true, y_true, marker='None', linewidth=5, color='k')

for d in range(9, 25, 3):
  X_d = polynomial_transform(x_true, d)
  y_d = X_d @ w[d]
  plt.plot(x_true, y_d, marker='None', linewidth=2)

plt.legend(['true'] + list(range(9, 25, 3)))
plt.axis([-8, 8, -15, 15])
```

Out[317…
```
(-8.0, 8.0, -15.0, 15.0)
```



2. **Regression with Radial Basis Functions**, 70 points

In the previous case, we considered a nonlinear extension to linear regression using a linear combination of polynomial basis functions, where each basis function was introduced as a feature $\phi(x) = x^k$. Now, we consider Gaussian radial basis functions of the form:

$$\phi(\mathbf{x}) = e^{-\gamma\,(x-\mu)^2},$$

whose shape is defined by its center μ and its width $\gamma > 0$. In the case of polynomial basis regression, the user's choice of the dimension d determined the transformation and the model. For radial basis regression, we have to contend with deciding how many radial basis functions we should have, and what their center and width parameters should be. For simplicity, let's assume that $\gamma = 0.1$ is fixed. Instead of trying to identify the number of radial basis functions or their centers, we can treat **each data point as the center of a radial basis function**, which means that the model will be:

$$f(x) = [w_1,\, w_2,\, w_3 \ldots, w_n]^T \begin{bmatrix} e^{-\gamma\,(x-x_1)^2} \\ e^{-\gamma\,(x-x_2)^2} \\ e^{-\gamma\,(x-x_2)^2} \\ \ldots \\ e^{-\gamma\,(x-x_n)^2} \end{bmatrix}.$$

This transformation uses radial basis functions centered around data points $e^{-\gamma\,(x-x_i)^2}$ and each basis function has a corresponding weight w_i associated with it, for all $i = 1, \ldots, n$. We transform each univariate data point x_j into into a multivariate (n-dimensional) data point via $\phi(x_j) \rightarrow [\ldots, e^{-\gamma\,(x_j-x_i)^2},\, \ldots]$. When this transformation is applied to every data point, it produces the **radial-basis kernel**:

$$\Phi = \begin{bmatrix} 1 & e^{-\gamma\,(x_1-x_2)^2} & e^{-\gamma\,(x_1-x_3)^2} & \ldots & e^{-\gamma\,(x_1-x_n)^2} \\ e^{-\gamma\,(x_2-x_1)^2} & 1 & e^{-\gamma\,(x_2-x_3)^2} & \ldots & e^{-\gamma\,(x_2-x_n)^2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ e^{-\gamma\,(x_n-x_1)^2} & e^{-\gamma\,(x_n-x_2)^2} & e^{-\gamma\,(x_n-x_3)^2} & \ldots & 1 \end{bmatrix}.$$

---

a. (15 points)

Complete the Python function below that takes univariate data as input and computes a radial-basis kernel. This transforms one-dimensional data into n-dimensional data in terms of Gaussian radial-basis functions centered at each data point and allows us to model nonlinear (kernel) regression.

In [318…
```
# X float(n, ): univariate data
# B float(n, ): basis functions
# gamma float : standard deviation / scaling of radial basis kernel
def radial_basis_transform(X, B, gamma=0.1):
    B = np.array(B)
    # compute radial basis
    return np.exp(-gamma * np.square(np.array([B - x for x in X])))
```

---

b. (15 points)

Complete the Python function below that takes a radial-basis kernel matrix Φ, the labels $\mathbf{y}$, and a regularization parameter $\lambda > 0$ as input and learns weights via **ridge regression**. Specifically, given a radial-basis kernel matrix Φ, implement the computation of $\mathbf{w} = \left(\Phi^T\Phi + \lambda I_n\right)^{-1} \Phi^T\mathbf{y}$.

In [320…
```
# Phi float(n, d): transformed data
# y   float(n,  ): labels
# lam float      : regularization parameter
def train_ridge_model(Phi, y, lam):
    # transpose phi
    Phi_t = np.transpose(Phi)
    return np.dot(np.dot(
        np.linalg.inv(np.dot(Phi_t, Phi) + np.dot(lam, np.identity(Phi.shape[
```

---

c. (30 points)

As before, we can explore the tradeoff between **fit and complexity** by varying $\lambda \in [10^{-3}, 10^{-2} \cdots, 1, \cdots 10^3]$. For each model, train using the transformed training data (Φ) and evaluate its performance on the transformed validation and test data. Plot two curves: (i) λ vs. validation error and (ii) λ vs. test error, as above.

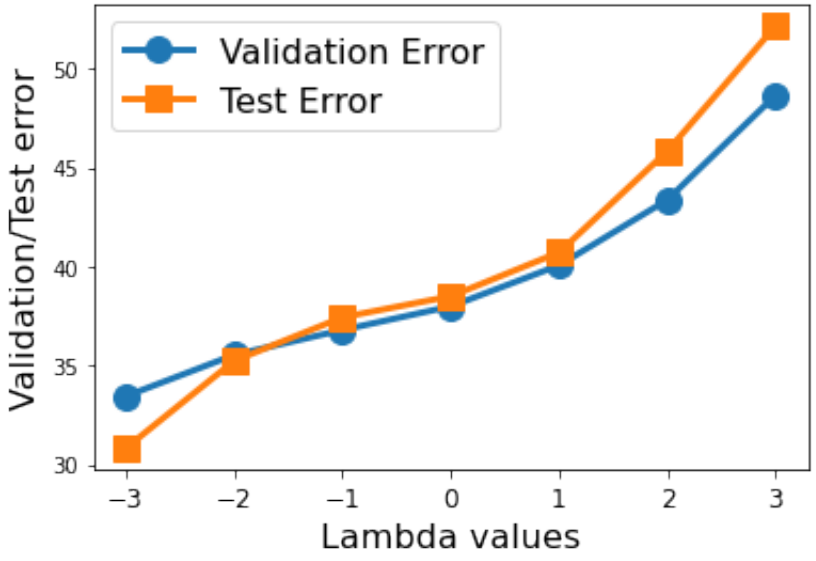
What are some ideal values of λ?

In [322…

```
w = {}                    # Dictionary to store all the trained models
validationErr = {}        # Validation error of the models
testErr = {}              # Test error of all the models

Phi_trn = radial_basis_transform(X_trn, X_trn, gamma=0.1)
Phi_val = radial_basis_transform(X_val, X_trn, gamma=0.1)
Phi_tst = radial_basis_transform(X_tst, X_trn, gamma=0.1)
for lam in range(-3, 4):  # lambdas
    w[lam] = train_ridge_model(Phi_trn, y_trn, 10 ** lam)
    validationErr[lam] = evaluate_model(Phi_val, y_val, w[lam])  # Evaluate m

    testErr[lam] = evaluate_model(Phi_tst, y_tst, w[lam])  # Evaluate model o

# Plot lambda vs validation error
plt.figure()
plt.plot(validationErr.keys(), validationErr.values(), marker='o', linewidth=
plt.plot(testErr.keys(), testErr.values(), marker='s', linewidth=3, markersiz
plt.xlabel('Lambda values', fontsize=16)
plt.ylabel('Validation/Test error', fontsize=16)
plt.xticks(list(validationErr.keys()), fontsize=12)
plt.legend(['Validation Error', 'Test Error'], fontsize=16)
```

Out[322…

```
<matplotlib.legend.Legend at 0x130bf9a60>
```

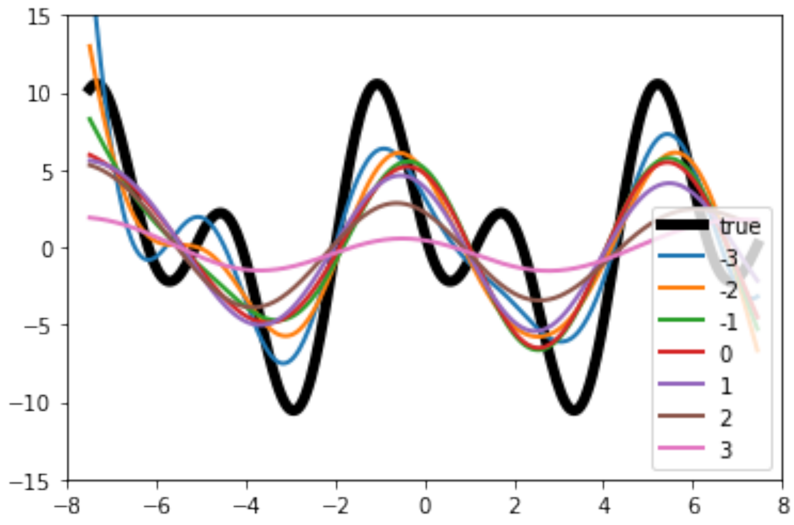


d. (10 points, **Discussion)**

Plot the learned models as well as the true model similar to the polynomial basis case above. How does the linearity of the model change with λ?

```
In [327…
plt.figure()
plt.plot(x_true, y_true, marker='None', linewidth=5, color='k')

for lam in range(-3, 4):
    X_d = radial_basis_transform(x_true, X_trn, gamma = 0.1)
    y_d = X_d @ w[lam]
    plt.plot(x_true, y_d, marker='None', linewidth=2)

plt.legend(['true'] + list(range(-3, 4)), loc = 'lower right')
plt.axis([-8, 8, -15, 15])
```

```
Out[327… (-8.0, 8.0, -15.0, 15.0)
```



Look at the plot, the lambda at -3 is most close to the true value. This is due to the larger lambda value that leads to larger oscilation. Hence, it is predicted that the smaller value may lead to the higher accuracy prediction.