CMSE 381, Fundamental Data Science Methods

November 2, 2025

Homework 6

Lowell Monis

Instructor: Dr. Mengsen Zhang

Question 1

For this question, you will use the polynomial toy data set from the course webpage. This is a very simple data set (check Acknowledgements for generation code), you will just predict `y` using `X`. We've learned three models in class this week.
- Polynomial Regression
- Step Functions
- Cubic splines

For each of these, do the following.

1. Identify the hyperparameter that is relevant to be chosen in that model (degree, number of cuts, etc.).
2. Use k-fold CV to find the best choice of that hyperparameter.
3. Train the model on all of the data using that chosen hyperparameter.

Finally, make a plot of the data, along with all three of the learned models plotted on top. What do you notice? Is one a better (or worse choice) than the others? Which would you choose and why?

First, I will load the data.

[2]:
```
data=pd.read_csv('../data/polynomial-toydata.csv').rename({'Unnamed: 0': ''},␣
 ↪axis=1).set_index('')
data.head()
```

[2]:
```
           x          y

0 -2.629139  -4.940311
1  2.556429  72.467439
2  0.587945   6.763137
3 -0.612074  12.080070
4 -4.595832 -92.688351
```

Polynomial regression The polynomial regression model to predict y using x would be:

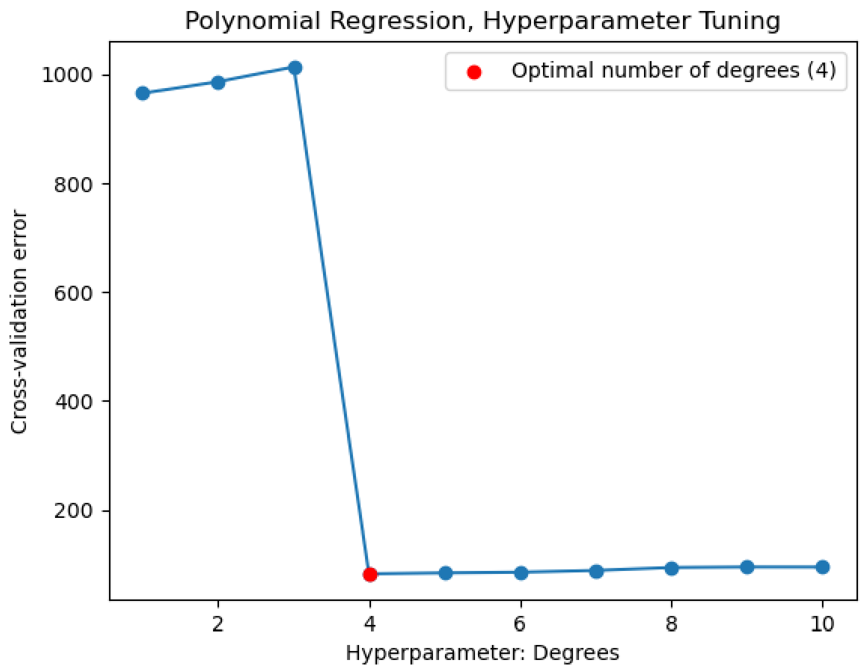
$$y = \sum_{p=0}^{n} \beta_p x^p + \epsilon$$

We need to find the optimal degree of this polynomial, n, which will be the hyperparameter that is to be chosen.

To tune the hyperparameter, I will use k-fold cross-validation. I am choosing to use 10-fold cross-validation over 5-fold due to the small size of the data set, ensuring enough training data exists to make a good choice. We will do cross-validation for degrees 1 through 10. With that being said, let's tune and train this model.

```
[3]: X = data.x.values.reshape(-1, 1)
     y = data.y
     degs = list(range(1,11))
     kf = KFold(n_splits=10, shuffle=True, random_state=381)
     mean_scores_poly=[]
     for d in degs:
         model=Pipeline([
             ('poly', PolynomialFeatures(degree=d, include_bias=False)),
             ('linear', LinearRegression())
         ])
         scores=cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=kf)
         mean_scores_poly.append(-np.mean(scores))
     deg_opt=np.argmin(mean_scores_poly)+1
     mse_opt=min(mean_scores_poly)
     print(deg_opt, mse_opt, sep=', ')
     plt.plot(degs, mean_scores_poly, '-o')
     plt.title('Polynomial Regression, Hyperparameter Tuning')
     plt.scatter(deg_opt, mse_opt, c='red', zorder=2, label=f'Optimal number of␣
      ↪degrees ({deg_opt})')
     plt.xlabel('Hyperparameter: Degrees')
     plt.ylabel('Cross-validation error')
     plt.legend()
     plt.show()
```

```
4, 83.34086255120299
```



The most optimal hyperparameter is 4 degrees, which has the lowest error at around 83.34. We can now proceed with training this model using 4 degrees on all the data together.

```
[4]: polymodel=Pipeline([
         ('poly', PolynomialFeatures(degree=deg_opt, include_bias=False)),
         ('linear', LinearRegression())
     ])
     polymodel.fit(X,y)
     print(polymodel.score(X, y))
```

```
0.9690772744894203
```

When trained on the entire data set with an optimal hyperparameter of 4 degrees, the model can explain 96.91% of the variance in the data. The model can be written out as follows:

```
[5]: polymodel.named_steps['linear'].coef_, polymodel.named_steps['linear'].intercept_
```

```
[5]: (array([-22.08618093,  -3.84263289,   5.75682445,   0.96852444]),
      18.535353194447353)
```

$$y = 18.54 - 22.09x - 3.84x^2 + 5.76x^3 + 0.97x^4 + \epsilon$$

Step functions The step function model to predict y using x with k bins and step function C_p would be:

$$y = \beta_0 + \sum_{p=1}^{k} \beta_p C_p(x)$$

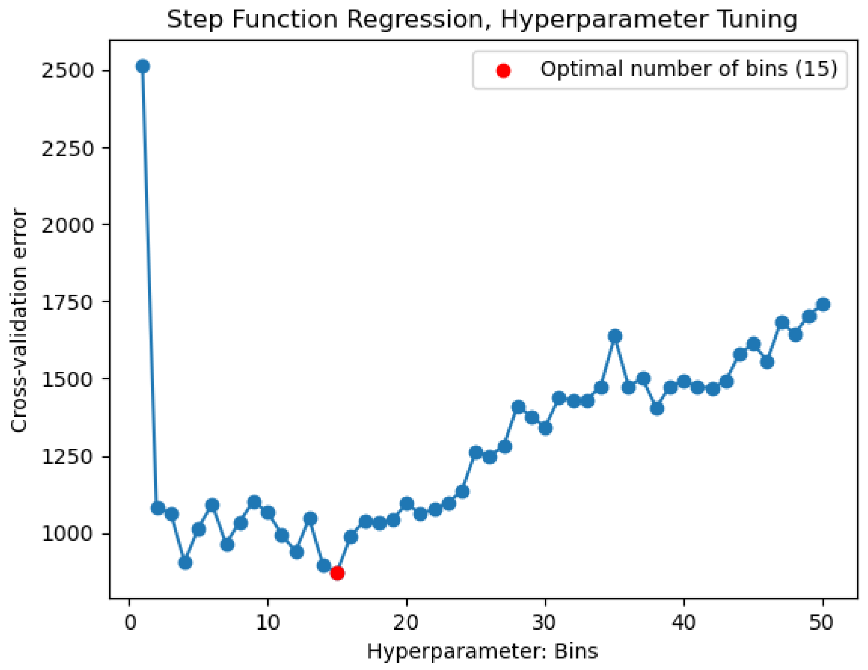
We need to find the optimal number of cuts to make to the data, $k-1$, to get the optimal number of bins k, which will be the hyperparameter that is to be chosen.

To tune the hyperparameter, I will use 10-fold cross-validation again. We will do cross-validation for bin sizes from 1 through 50. With that being said, let's tune and train this model. Here, I am creating a pipeline to account for data leakage during cross-validation.

```
[6]: bins=list(range(1,51))
     mean_scores_step=[]
     for k in bins:
         def create_bins(X):
             df_cut = pd.cut(X, k, right=False)
             return pd.get_dummies(df_cut)
         pipeline = Pipeline([
             ('binner', FunctionTransformer(create_bins)),
             ('linear', LinearRegression())
         ])
         scores = cross_val_score(pipeline, data.x, y,␣
      ↪scoring='neg_mean_squared_error', cv=kf)
         mean_scores_step.append(-np.mean(scores))
```

```
bin_opt=np.argmin(mean_scores_step)+1
mse_opt=min(mean_scores_step)
print(bin_opt, mse_opt, sep=', ')
plt.plot(bins, mean_scores_step, '-o')
plt.title('Step Function Regression, Hyperparameter Tuning')
plt.scatter(bin_opt, mse_opt, c='red', zorder=2, label=f'Optimal number of bins␣
 ↪({bin_opt})')
plt.xlabel('Hyperparameter: Bins')
plt.ylabel('Cross-validation error')
plt.legend()
plt.show()
```

```
15, 871.6642539510378
```



The most optimal hyperparameter is 15 bins, which has the lowest error at around 871.66. We can now proceed with training this model using 15 bins on all the data together.

[7]:
```
def create_bins(X):
    df_cut = pd.cut(X, bin_opt, right=False)
    return pd.get_dummies(df_cut)
stepmodel = Pipeline([
    ('binner', FunctionTransformer(create_bins)),
    ('linear', LinearRegression())
])
stepmodel.fit(data.x,y)
print(stepmodel.score(data.x, y))
```

```
0.9208085567238795
```

When trained on the entire data set with an optimal hyperparameter of 15 bins to the data, the model can explain 92.08% of the variance in the data. The model can be written out as follows:

[8]:
```python
stepmodel.named_steps['linear'].coef_, stepmodel.named_steps['linear'].intercept_
```

[8]:
```
(array([-23.69505503, -70.49670739, -81.19145303, -58.26902733,
        -34.32390938,  -6.71095761,  20.72330385,  31.5653355 ,
         42.25922457,  31.9720076 ,  20.23373814,   4.49238439,
          5.68624646,  29.23598051,  88.51888876]),
 -5.415060382228832)
```

$$y = -5.42 - 23.70C_1(x) - 70.50C_2(x) - 81.19C_3(x) - 58.30C_4(x) - 34.32C_5(x) - 6.71C_6(x) + 20.72C_7(x) + 31.57C_8(x) + 4$$

Cubic splines The cubic spline model to predict y using x with k knots would be:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \sum_{j=1}^{k} \beta_{j+3} \cdot (x - t_j)_+^3$$

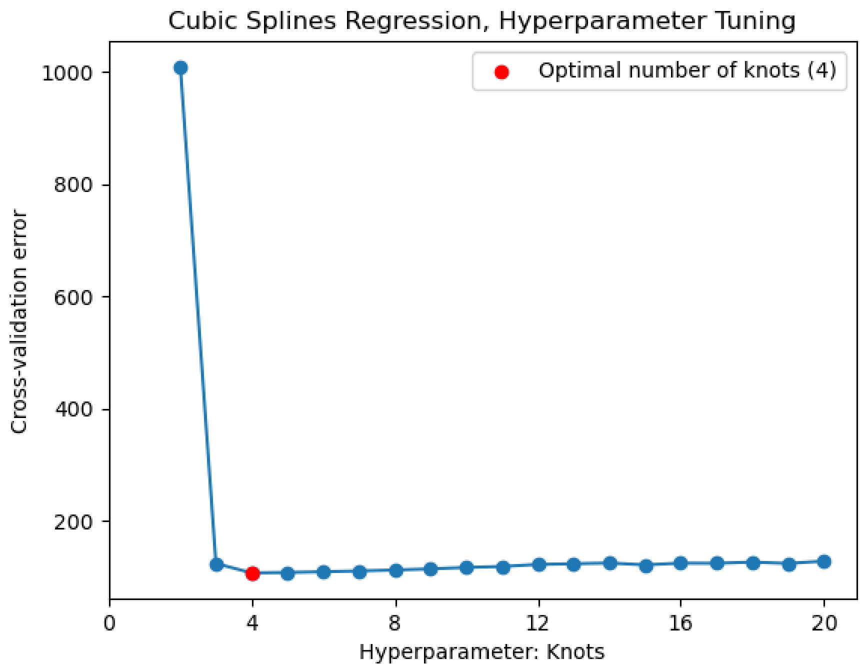
Here, $(x - t_j)_+^3$ is a truncated power basis function.

We need to find the optimal number of knots k, which will be the hyperparameter that is to be chosen.

To tune the hyperparameter, I will use 10-fold cross-validation again. We will do cross-validation for knots from 2 through 20. With that being said, let's tune and train this model.

[9]:
```python
knots = list(range(2, 21))
mean_scores_spline = []
for k in knots:
    pipeline = Pipeline([
        ('spline', SplineTransformer(n_knots=k, degree=3)),
        ('linear', LinearRegression())
    ])
    scores = cross_val_score(pipeline, X, y,
                             scoring='neg_mean_squared_error', cv=kf)
    mean_scores_spline.append(-np.mean(scores))
knot_opt = knots[np.argmin(mean_scores_spline)]
mse_opt = min(mean_scores_spline)
print(knot_opt, mse_opt, sep=', ')
plt.plot(knots, mean_scores_spline, '-o')
plt.title('Cubic Splines Regression, Hyperparameter Tuning')
plt.scatter(knot_opt, mse_opt, c='red', zorder=2, label=f'Optimal number of␣
 ↪knots ({knot_opt})')
plt.xlabel('Hyperparameter: Knots')
plt.ylabel('Cross-validation error')
plt.xticks(list(range(0,22,4)))
plt.legend()
plt.show()
```

```
4, 106.64176902909915
```



The most optimal hyperparameter is 4 knots, which has the lowest error at around 106.64. We can now proceed with training this model using 4 knots on all the data together.

```
[10]: splinemodel = Pipeline([
          ('spline', SplineTransformer(n_knots=knot_opt, degree=3)),
          ('linear', LinearRegression())
      ])
      splinemodel.fit(X,y)
      print(splinemodel.score(X, y))
```

```
0.9688712254776765
```

The knots are given as follows:

```
[11]: splinemodel.named_steps['spline'].bsplines_[0].t
```

```
[11]: array([-14.78258432, -11.83848986,  -8.8943954 ,  -5.95030095,
              -3.00620649,  -0.06211203,   2.88198243,   5.82607689,
               8.77017135,  11.7142658 ])
```

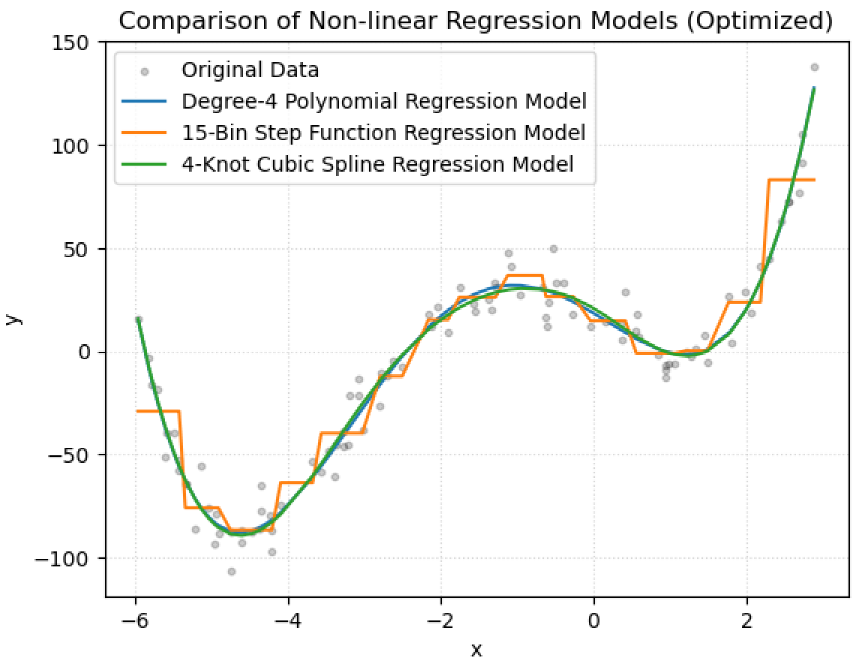
When trained on the entire data set with an optimal hyperparameter of 4 knots to the data, the model can explain 96.89% of the variance in the data. The following are the coefficients:

```
[12]: splinemodel.named_steps['linear'].coef_, splinemodel.named_steps['linear'].
      ↪intercept_
```

```
[12]: (array([ 733.32322492, -560.95209715, -310.15905741, -259.97581236,
              -430.55197577,  828.31571777]),
       318.99191572969033)
```

Bring it all together Now, let's plot all three learned models overlayed on the data.

```
[13]: X_plot = pd.DataFrame(X).sort_values(by=0)
      plt.scatter(X,y, s=10, label='Original Data', c='black', alpha=0.2)
      plt.plot(X_plot,polymodel.predict(X_plot), label='Degree-4 Polynomial Regression␣
      ↪Model')
      plt.plot(data.x.sort_values(),stepmodel.predict(data.x.sort_values()),␣
      ↪label='15-Bin Step Function Regression Model')
      plt.plot(X_plot,splinemodel.predict(X_plot), label='4-Knot Cubic Spline␣
      ↪Regression Model')
      plt.title('Comparison of Non-linear Regression Models (Optimized)')
      plt.xlabel('x')
      plt.ylabel('y')
      plt.legend()
      plt.grid(True, linestyle=':', alpha=0.5)
      plt.show()
```



We can clearly rule out the step function model due to its discontinuity and increased error, worsened by the polynomial form of the data (which we know from prior knowledge). There would still be reasons I would use the step function model, which I will address shortly. Let's analyze the smooth models first. Both the degree-4 polynomial and the 4-knot cubic spline provide an excellent fit to the underlying smooth trend of the data, especially due to the data's inherrent polynomial form. The mean squared errors in both the cases are around the same magnitude-wise, with the polynomial having a lower MSE. The polynomial model also has the advantage in terms of R-squared, but the models when plotted look the same. The polynomial model is a single, globally defined function. It captures the overall quartic trend efficiently. The cubic spline model achieves similar low error by dividing the `x` range into segments defined by the knots. The smoothness constraints at the knots (continuous value, first derivative, and second derivative) ensure the curve is visually indistinguishable from the polynomial in this case. The smoothing is easier here due to the fact that the data is derived from a polynomial.

While both performed well here, the textbook highlights a critical difference that often makes splines superior: how flexibility is introduced. Polynomials introduce flexibility globally by increasing the degree. As the textbook notes (ISLP § 7.4.5), using a very high degree (like 15) to achieve flexibility can lead to "undesirable results at the boundaries" (i.e., wildly varying predictions outside the data range). Splines introduce flexibility locally by increasing the number of knots, while keeping the polynomial degree fixed (i.e., cubic). This results in more stable estimates because flexibility is added only where needed, not across the entire function domain.

For this simple dataset, a low-degree polynomial ($d = 4$) was sufficient and avoided the boundary issues seen with higher-degree polynomials, but splines remain the preferred approach for complex, real-world data due to their superior local control over flexibility (8 degrees of freedom compared to 5 in the polynomial in this case). Moreover, the tradeoff with high flexibility in splines is overfitting, which is very possible.

The 15-bin step function clearly provides the worst fit. The model forces the smooth underlying relationship to be approximated by a series of discontinuous, horizontal steps. It introduces a high degree of error because its fundamental structure is a poor match for the smooth, continuously changing function in the data. I would choose step functions if the data was not naturally smooth via its inherent quartic structure, since step functions provide flexibility while also escaping overfitting because of its discontinuous structure.

Personally for this data, I would choose the polynomial model. The error is the lowest for the polynomial model, which would introduce concerns of overfitting, but the model is parsimonious and simple. It describes the entire relationship using a single, simple, globally defined quartic equation. This is simpler to interpret and deploy than a cubic spline. Also, since we have prior knowledge about the data, the polynomial model would be the most natural and efficient way to capture this specific smooth, global structure. This positive, in my belief, neutralizes concerns about overfitting.

Question 2

Part A.

I am learning a step function of some data, and I'm using knots $c_1 = 3$ and $c_2 = 7$. - (i) Write equations for each of the basis functions $C_0(X)$, $C_1(X)$, and $C_2(X)$. Sketch the three functions. - (ii) If the model learned was

$$f(X) = \beta_0 + \beta_1 C_1(X) + \beta_2 C_2(X)$$

with $\beta_0 = 2$, $\beta_1 = 3$, and $\beta_2 = -1$, sketch the graph learned.

Part B.

- I am learning a cubic spline of some data with a single knot at $c_1 = 4$. As noted in class and in ISLP § 7.4.3, we have a basis for learning cubic spline data. I'm going to build a cubic spline with basis functions
  - $b_1(X) = X$
  - $b_2(X) = X^2$
  - $b_3(X) = X^3$
  - $b_4(X) = h(X, 4) = \begin{cases} (X-4)^3 & X > 4 \\ 0 & \text{else} \end{cases}$
- Assume the learned model was

$$f(X) = 3 + b_1(X) - 2b_2(X) + 3b_3(X) - 4b_4(X)$$

  - (i) Write the equation for the piecewise polynomial that this function represents. Draw a graph of the function.
  - (ii) What are the requirements for a piecewise polynomial function to be a cubic spline?
  - (iii) Check that your piecewise polynomial from (i) fits these requirements.

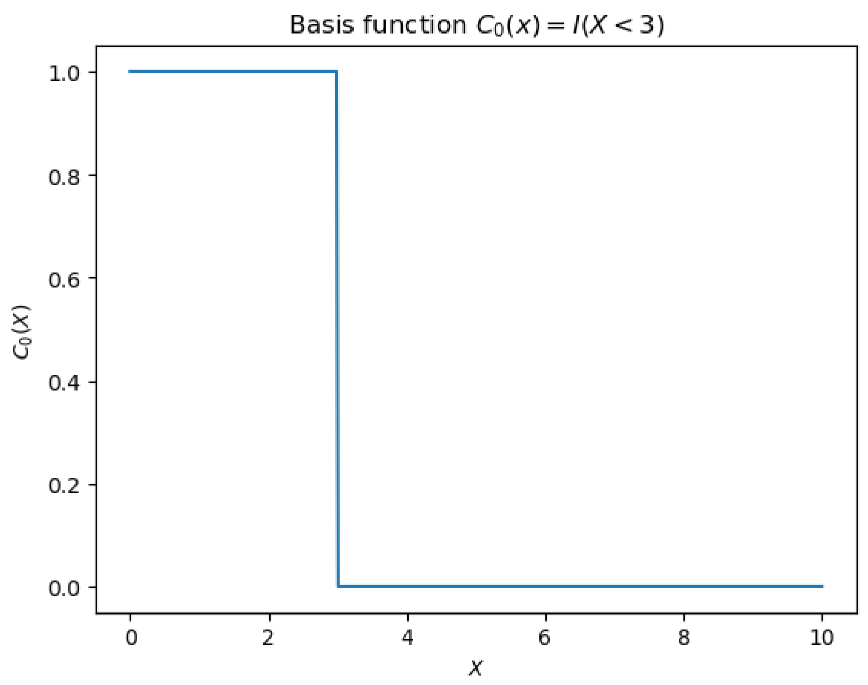
Part A The equations for each of the basis functions are as follows:

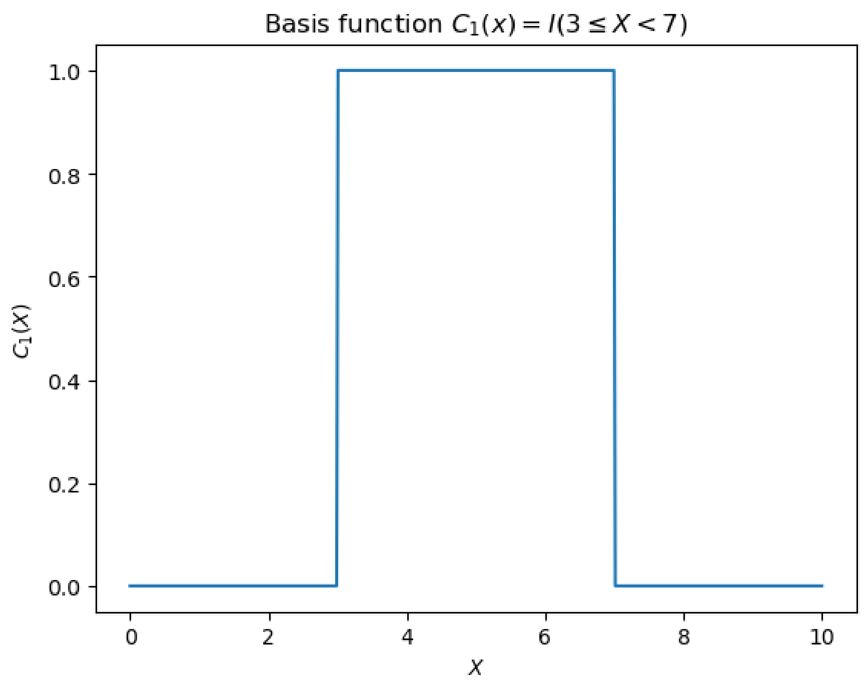
$$C_0(x) = I(X < c_1) = I(X < 3) = \begin{cases} 1, & \text{if } X < 3 \\ 0, & \text{otherwise} \end{cases}$$

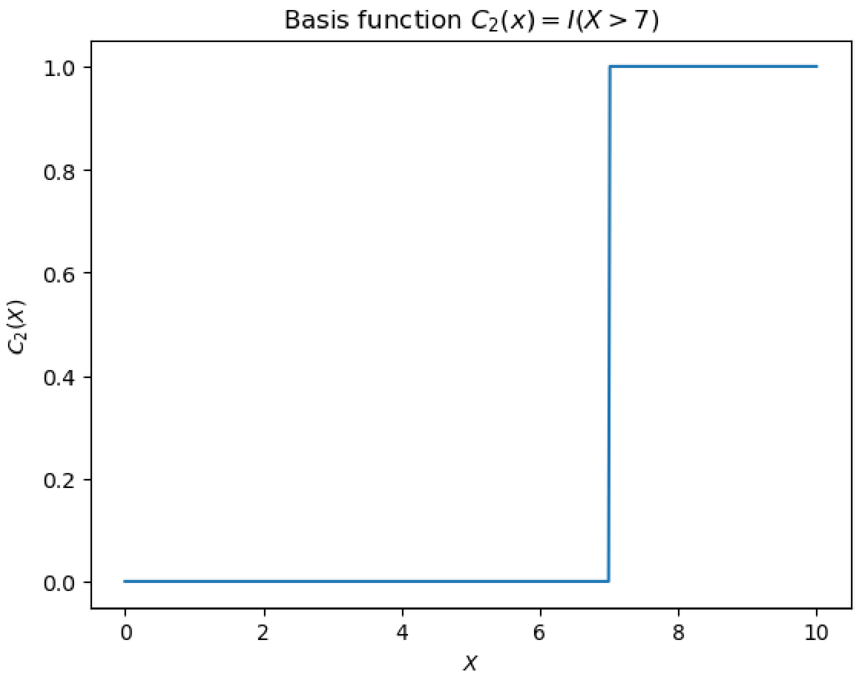
$$C_1(x) = I(c_1 \leq X < c_2) = I(3 \leq X < 7) = \begin{cases} 1, & \text{if } 3 \leq X < 7 \\ 0, & \text{otherwise} \end{cases}$$

$$C_2(x) = I(X \geq c_2) = I(X \geq 7) = \begin{cases} 1, & \text{if } X \geq 7 \\ 0, & \text{otherwise} \end{cases}$$

The plots are sketched as follows:


Basis function $C_0(x) = I(X < 3)$
$C_0(X)$
X
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10

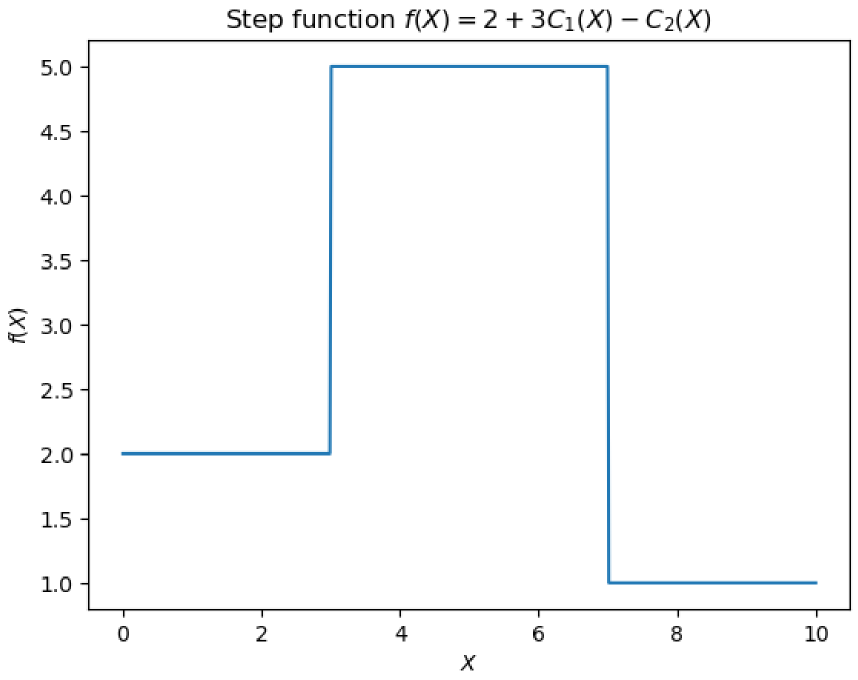

Basis function $C_1(x) = I(3 \leq X < 7)$
$C_1(X)$
X
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10



The step function of the model learned was

$$f(X) = \beta_0 + \beta_1 C_1(X) + \beta_2 C_2(X)$$

with $\beta_0 = 2$, $\beta_1 = 3$, and $\beta_2 = -1$, is sketched below:



Part B We have the basis functions as follows:

$$b_1(X) = X$$
$$b_2(X) = X^2$$
$$b_3(X) = X^3$$
$$b_4(X) = h(X,4) = \begin{cases} (X-4)^3 & X > 4 \\ 0 & \text{else} \end{cases}$$

Considering the learned model is $f(X) = 3 + b_1(X) - 2b_2(X) + 3b_3(X) - 4b_4(X)$, let's substitute the basis functions into the model:

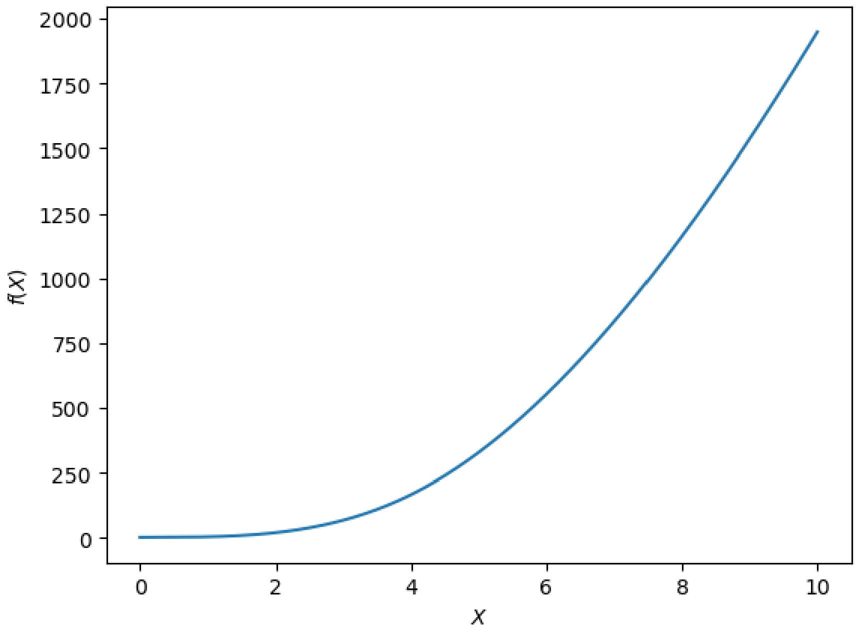
$$f(X) = 3 + X - 2X^2 + 3X^3 - 4h(X,4)$$

For $X > 4$, we need to expand $4h(X,4) = 4(X-4)^3 = 4(X^3 - 12X^2 + 48X - 64) = 4X^3 - 48X^2 + 192X - 256$.

The model has one knot at 4. We can expand this into a piecewise polynomial as follows:

$$f(X) = \begin{cases} 3 + X - 2X^2 + 3X^3, & \text{if } X \leq 4 \\ -X^3 + 46X^2 - 191X + 259 & \text{if } X > 4 \end{cases}$$

I can now draw a graph of the function.



For a piecewise polynomial function to be a cubic spline with knots at $c_1, c_2, \ldots, c_K$, it must satisfy the following:

1. Each piece is a polynomial of degree at most 3 (cubic)
2. The function is continuous at each knot: $\lim_{x \to c_i^-} f(x) = \lim_{x \to c_i^+} f(x) = f(c_i)$ for all knots c_i
3. The first derivative is continuous at each knot: $\lim_{x \to c_i^-} f'(x) = \lim_{x \to c_i^+} f'(x)$ for all knots c_i
4. The second derivative is continuous at each knot: $\lim_{x \to c_i^-} f''(x) = \lim_{x \to c_i^+} f''(x)$ for all knots c_i

In short, a cubic spline must be a piecewise cubic polynomial that is twice continuously differentiable everywhere, including at the knots.

Now, I will verify that this piecewise polynomial satisfies these requirements at the knot $c_1 = 4$.

$$f(X) = \begin{cases} 3 + X - 2X^2 + 3X^3, & \text{if } X \leq 4 \\ -X^3 + 46X^2 - 191X + 259 & \text{if } X > 4 \end{cases}$$

- $f_L(X) = 3 + X - 2X^2 + 3X^3$
- $f_R(X) = -X^3 + 46X^2 - 191X + 259$

Both $f_L(X)$ and $f_R(X)$ are cubic polynomials.

$$f_L(4) = 3 + 4 - 2(4)^2 + 3(4)^3 = 3 + 4 - 32 + 192 = 167$$

$$\begin{gathered} f_R(4) = -(4)^3 + 46(4)^2 - 191(4) + 259 \\ = -64 + 736 - 764 + 259 = 167 \end{gathered}$$

$$f_L(4) = f_R(4) = 167$$

The function is continuous at the knot.

$$\begin{gathered} f_L'(X) = 1 - 4X + 9X^2 \\ f_L'(4) = 1 - 4(4) + 9(4)^2 = 1 - 16 + 144 = 129 \end{gathered}$$

$$\begin{gathered} f_R'(X) = -3X^2 + 92X - 191 \\ f_R'(4) = -3(4)^2 + 92(4) - 191 = -48 + 368 - 191 = 129 \end{gathered}$$

$$f_L'(4) = f_R'(4) = 129$$

The first derivative is continuous at the knot.

$$f''_L(X) = -4 + 18X$$
$$f''_L(4) = -4 + 18(4) = -4 + 72 = 68$$

$$f''_R(X) = -6X + 92$$
$$f''_R(4) = -6(4) + 92 = -24 + 92 = 68$$

$$f''_L(4) = f''_R(4) = 68$$

The second derivative is continuous at the knot.

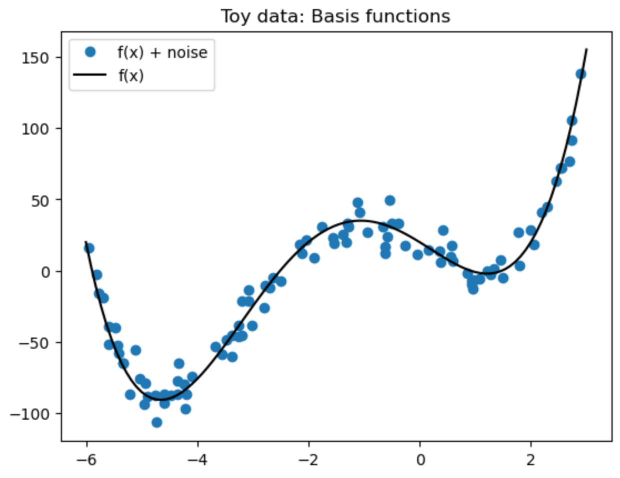
Thus, the piecewise polynomial satisfies all four requirements for a cubic spline. Therefore, this piecewise polynomial is indeed a cubic spline. ■

Acknowledgements

I would like to attribute the use of Gemini to sanity check my code for Question 1, while creating the cross-validation for the step function regressor. I also learned that there is data leakage while cross-validating and ask the large language model to give me ways to deal with the cross-validation, which I implement myself using my prior knowledge on pipelines, as well as `sklearn.preprocessing.FunctionTransformer()`, which the LLM taught me about.

The following code was used to generate the polynomial toy data. Attribution is being provided to the Department of Computational Mathematics, Science, and Engineering, and Dr. Elizabeth Munch.

[17]:
```
# Set the random seed for reproducibility
np.random.seed(42)

# Define the polynomial
f = lambda x: (x+2)*(x-2)*(x+6)*x+20

# Generate data
x_data = np.random.uniform(-6, 3, 100)
y_data = f(x_data) + np.random.normal(0, 10, size=x_data.shape)

# Generate data for plotting
x_plot = np.linspace(-6, 3, 100)
y_plot = f(x_plot)

# Create a DataFrame to store the data
toy_data = pd.DataFrame({'x': x_data, 'y': y_data})

plt.plot(x_data,y_data, 'o', label = 'f(x) + noise')
plt.plot(x_plot, y_plot, 'k', label = 'f(x)')
plt.legend()
plt.title('Toy data: Basis functions')
```


Toy data: Basis functions
● f(x) + noise
f(x)