SDS 383D, Exercises 3: Linear smoothing and Gaussian processes

Jan-Michael Cabrera

February 27, 2019

Curve fitting by linear smoothing

(A) Linear smoothing Suppose we want to estimate the value of the regression function y^* at some new point x^*, denoted $\hat{f}(x^*)$. Assume for the moment that $f(x)$ is linear, and that y and x have already had their means subtracted, in which case $y_i = \beta x_i + \epsilon_i$.

Show that for the one-predictor case, your prediction $\hat{y}^* = \hat{f}(x^*) = \hat{\beta} x^*$ may be expressed as:

$$\hat{f}(x^*) = \sum_{i=1}^{n} w(x_i, x^*) y_i$$

We can express $\hat{\beta}$ in matrix notation as,

$$\hat{\beta} = (X^T X)^{-1} X^T Y.$$

Plugging this in to find our predictor, $\hat{y}^*$ we get,

$$\begin{aligned} \hat{y}^* &= x^* (X^T X)^{-1} X^T Y \\ &= \frac{X^T Y}{X^T X} \\ &= \frac{\sum_{i=1}^n x_i x^* y_i}{\sum_{i=1}^n x_i^2} \end{aligned}$$

The result is of the form $\hat{f}(x^*) = \sum_{i=1}^n w(x_i, x^*) y_i$ with the weights expressed as,

$$w(x_i, x^*) = \frac{x_i x^*}{\sum_{i=1}^n x_i^2}.$$

This resultant *smoother* has constant weights and produces predictions that lie on the line with slope $\hat{\beta}$. If the data is binned and locality preserved, this estimate is similar to a running line average.

$$w_K(x_i, x^*) = \begin{cases} 1/K, & x_i \text{ one of the } K \text{ closest sample points to } x^*, \\ 0, & \text{otherwise.} \end{cases}$$

With *K-nearest-neighbor smoothing* $\hat{y}^*$ is essentially the arithmetic mean of the K y_i's nearest x^*.

(B) A *kernel function* $K(x)$ is a smooth function satisifying

$$\int_{\mathbb{R}} K(x) dx = 1 \,, \quad \int_{\mathbb{R}} x K(x) dx = 0 \,, \quad \int_{\mathbb{R}} x^2 K(x) dx > 0 \,.$$

A very simple example is the uniform kernel,

$$K(x) = \frac{1}{2} I(x) \quad \text{where} \quad I(x) = \begin{cases} 1, & |x| \leq 1 \\ 0, & \text{otherwise}. \end{cases}$$

Another common example is the Gaussian kernel:

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

Kernels are used as weighting functions for taking local averages. Specifically, define the weighting function

$$w(x_i, x^\star) = \frac{1}{h} K\left(\frac{x_j - x^\star}{h}\right),$$

where h is the bandwidth. Using this weighting function in a linear smoother is called *kernel regression*. (The weighting function gives the unnormalized weights; you should normalize the weights so that they sum to 1.)

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import sys
sys.path.append('../../scripts/')
from smoothers import kernel_smoother
```

```
# Create noisy data, y_i = sin(x_i) + e_i
x = np.linspace(0, 2*np.pi, num = 20)
y = np.sin(x) + np.random.normal(scale=0.2, size=x.shape)

# Create vector for fitting purposes
x_star = np.linspace(0, 2*np.pi)

# Instantiate array of bandwidths
h = np.array([0.25, 1.0])
```

```
# Instantiate a list to append kernel_smoother objects
H = []

# Iterates through array of bandwidths and passes the feature vector, response vector, and ↩
    bandwidth to kernel_smoother object
for i in range(len(h)):
    H.append(kernel_smoother(x, y, x_star, h=h[i]))
```

```
# Plots data
plt.figure()
plt.plot(x, y, '.k', label='Noisy response')
plt.plot(x, np.sin(x), '--k', label='True function')
# Iterates over the smoother objects and plots functions for the uniform and gaussian kernels
for i in range(len(h)):
    plt.plot(x_star, H[i].predictor(kernel='uniform'), label='Uniform Kernel, h='+str(h[i]))
    plt.plot(x_star, H[i].predictor(kernel='gaussian'), label='Gaussian Kernel, h='+str(h[i]))
plt.legend(loc=0)
# plt.show()
plt.savefig('figures/kernel_smoother.pdf')
plt.close()
```

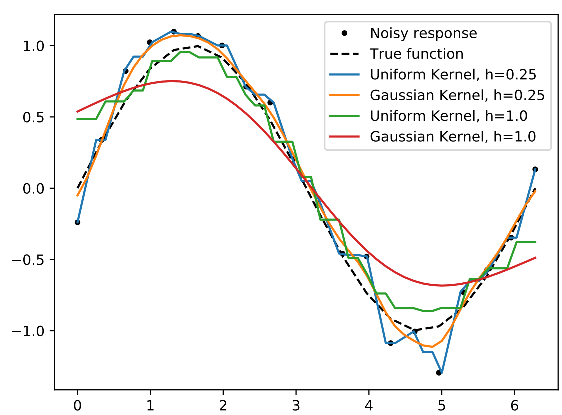

Figure 1: Uniform and Gaussian Kernel smoothing for different bandwidths

A Kernel Smoothing Code

```
from __future__ import division
import numpy as np

class kernels:
    """
    This class contains kernel functions for kernel smoothing
    """
    def __init__(self):
        pass

    def uniform(x):
        """
        Parameters
        ----------
            x: float
                argument to be evaluated

        Returns
        ----------
            k: float
                uniform kernel evaluation at x
                .. math:: k(x) = 1/2 I(x), with I(x) = 1, if |x| \\leq 1, 0 otherwise
        """
        if np.abs(x) <= 1:
            k = 0.5
        else:
            k = 0
        return k

    def gaussian(x):
        """
        Parameters
        ----------
            x: float
                argument to be evaluated

        Returns
        ----------
            k: float
                gaussian kernel evaluation at x
                .. math:: k(x) = \\frac{1}{\\sqrt{2 \\pi}} e^{-x^2/2}
        """
        return 1/np.sqrt(2*np.pi)*np.exp(-x**2/2)

class kernel_smoother:
    """
    This class returns a vector of smoothed values given feature and response vectors
    """
    def __init__(self, x, y, x_star, h=0.5):
        """
        Parameters
        ----------
            x: float
                Feature vector

            y: float
                Response vector

            x_star: float
                Scalar or vector to be evaluated

            h: float (optional)
                Bandwidth
        """
        self.x = x
```

```python
        self.y = y
        self.x_star = x_star
        self.h = h

    def predictor(self, kernel='gaussian'):
        """
        Parameters
        ----------
            kernel: str (optional)
                Kernel type to be used: Available kernels are uniform, gaussian,

        Returns
        ----------
            y_star: float, len(x_star)
                Predictor for x_star
                .. math:: y_i^* = \\sum_{i=1}^n w(x_i, x^*) y_i
                .. math:: w(x_i, x^*) = \\frac{1}{2}K( \\frac{x_i - x^*}{h}), \\sum_{i=1}^n w(x_i, x^*) = 1
        """
        # Instantiate y_star
        y_star = np.zeros(self.x_star.shape)

        # Iterate through each value in y_star
        for i in range(y_star.shape[0]):

            # Instantiate a weights vector
            weights = np.zeros(self.x.shape)

            # Iterates through feature/response vectors
            for j in range(self.x.shape[0]):

                # X = \frac{(x_i - x^*)}{h}
                X = (self.x[j] - self.x_star[i])/self.h

                # w(x_i, x^*) = \frac{1}{h}K(X)
                weights[j] = 1/self.h*getattr(kernels, kernel)(X)

            # Normalizes weights such that \sum_{i=1}^n w(x_i, x^*) = 1
            weights = weights/weights.sum()

            # y_i^* = \sum_{i=1}^n w(x_i, x^*) y_i
            y_star[i] = (weights*self.y).sum()

        return y_star
```