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

Jan-Michael Cabrera

March 4, 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 satisyfing

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

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()
```

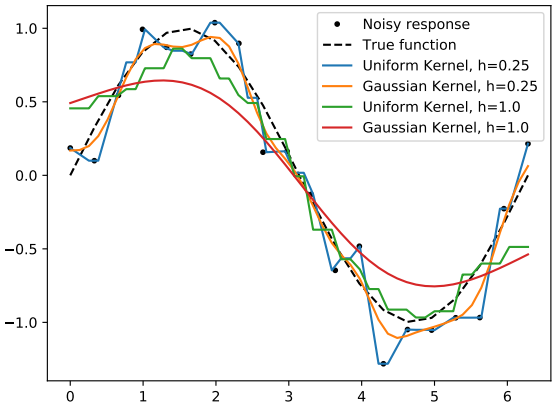

Figure 1: Uniform and Gaussian Kernel smoothing for different bandwidths

Cross validation

(A) Presumably a good choice of h would be one that led to smaller predictive errors on fresh data. Write a function or script that will: (1) accept an old ("training") data set and a new ("testing") data set as inputs; (2) fit the kernel-regression estimator to the training data for specified choices of h; and (3) return the estimated functions and the realized prediction error on the testing data for each value of h. This should involve a fairly straightforward "wrapper" of the function you've already written.

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

```
# Function for testing smoothers
def func(x):
    return np.sin(x)

# Initialize random seed
np.random.seed(3)

# Initialize x-vector
x = np.linspace(0, 2*np.pi, num=20)

# Training data of the form, y_i = f(x_i) + \epsilon_i
y_training = func(x) + np.random.normal(scale=0.2, size=x.shape)

# Testing data of the form, y_i = f(x_i) + \epsilon_i
y_test = func(x) + np.random.normal(scale=0.2, size=x.shape)
```

```
# Initialize list for storing kernel_smoother objects
H =[]

# Array of differing bandwidths to evaluate
h = np.array([0.2, 0.3, 0.4, 0.5, 0.6])

# Initialize MSE vector
mse = np.zeros(len(h))
```

```
# Iterates over bandwidth array and finds approximate MSE for each
for i in range(len(h)):
    H.append(kernel_smoother(x, y_training, x, h=h[i]))
    H[i].predictor()
    mse[i] = H[i].MSE(y_test)

print(mse)
```

```
jancabrera@phoenix$ python cross_validation_a.py
[0.07662882 0.0815833  0.0892882  0.10014996 0.11362704]
```

(B) Imagine a conceptual two-by-two table for the unknown, true state of affairs. The rows of the table are "wiggly function" and "smooth function," and the columns are "highly noisy observations" and "not so noisy observations." Simulate one data set (say, 500 points) for each of the four cells of this table, where the x's take values in the unit interval. Then split each data set into training and testing subsets. You choose the functions. Apply your method to each case, using the testing data to select a bandwidth parameter. Choose the estimate that minimizes the average squared error in prediction, which estimates the mean-squared error:

$$L_n(\hat{f}) = \frac{1}{n}\sum_{i=1}^{n^\star}(y_i^\star - \hat{y}_i^\star)^2\,,$$

where $(y_i^\star, x_i^\star)$ are the points in the test set, and $\hat{y}_i^\star$ is your predicted value arising from the model you fit using only the training data. Does your out-of-sample predictive validation method lead to reasonable choices of h for each case?

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

```
# Function for testing smoothers
def func(x, period = 1):
    return np.sin(x*2*np.pi*period)

# Initialize random seed
np.random.seed(3)

# Initialize x-vector
x = np.linspace(0, 1, num=100)

# Noise level array, low and high
noise = np.array([0.05, 0.25])
noise_label = ['Low', 'High']

# Period of function, high period (wiggly), low period (smooth)
period = np.array([3, 0.5])
period_label = ['Wiggly', 'Smooth']

### Initialize lists for storing the different responses
# Training responses
Y_training = []
# Testing responses
Y_test = []
# True response
T = []
```

```
# Iterates over noise and period arrays to assign data to lists
for n in range(len(noise)):
    for p in range(len(period)):
        Y_training.append(func(x, period=period[p]) + np.random.normal(scale=noise[n], size=x.↩
            shape))
        Y_test.append(func(x, period=period[p]) + np.random.normal(scale=noise[n], size=x.↩
            shape))
        T.append(func(x, period=period[p]))
```

```
# Generates plots
i = 0
for n in range(len(noise)):
    for p in range(len(period)):

        # Instantiate kernel_smoother object
        y_smooth = kernel_smoother(x, Y_training[i], x)
        # Perform initial curve fit
        y_smooth.predictor()

        # Optimize bandwidth given the test data
        y_smooth.optimize_h(Y_test[i])

        # Perform curve fit with optimized bandwidth
        y_smooth.predictor()

        # Predicted values for opimized bandwidth
        y_pred = y_smooth.y_star
```

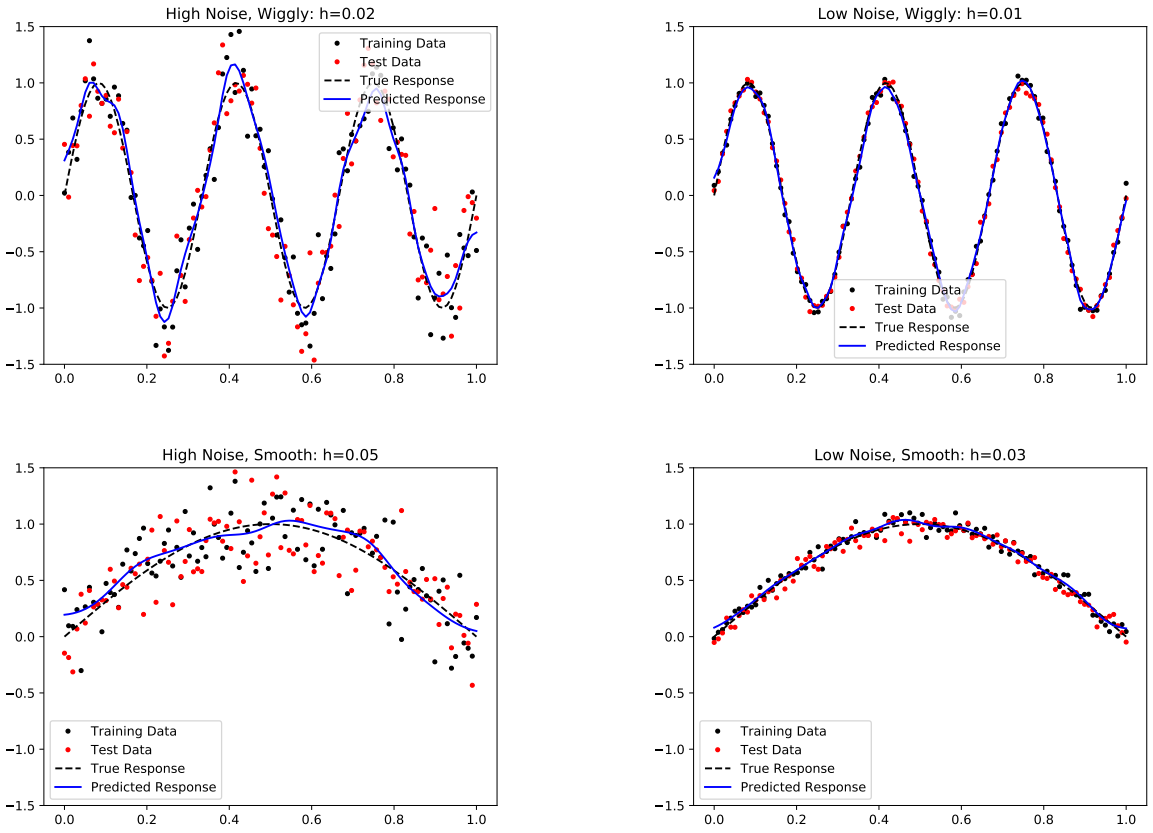

Figure 2: Bandwidths for various functions and noise levels

```
        h_star = y_smooth.h

        plt.figure()
        plt.title(noise_label[n]+' Noise, '+ period_label[p] + ': h={:.2f}'.format(h_star[0]))
        plt.plot(x, Y_training[i], '.k', label='Training Data')
        plt.plot(x, Y_test[i], '.r', label='Test Data')
        plt.plot(x, T[i], '--k', label='True Response')
        plt.plot(x, y_pred, '-b', label='Predicted Response')
        plt.legend(loc=0)
        # plt.show()
        plt.savefig('figures/cross_validation_'+noise_label[n]+'_'+period_label[p]+'.pdf')
        plt.close()
        i += 1
```

Local polynomial regression

Kernel regression has a nice interpretation as a "locally constant" estimator, obtained from locally weighted least squares. To see this, suppose we observe pairs (x_i, y_i) for $i = 1, \ldots, n$ from our new favorite model, $y_i = f(x_i) + \epsilon_i$ and wish to estimate the value of the underlying function $f(x)$ at some point x by weighted least squares. Our estimate is the scalar quantity

$$\hat{f}(x) = a = \arg\min_{\mathbb{R}} \sum_{i=1}^{n} w_i (y_i - a)^2 \,,$$

where the w_i are the normalized weights (i.e. they have been rescaled to sum to 1 for fixed x). Clearly if $w_i = 1/n$, the estimate is simply $\bar{y}$, the sample mean, which is the "best" globally constant estimator. Using elementary

calculus, it is easy to see that if the unnormalized weights are

$$w_i \equiv w(x, x_i) = \frac{1}{h} K\left(\frac{x_i - x}{h}\right),$$

then the solution is exactly the kernel-regression estimator.

(A) A natural generalization of locally constant regression is local polynomial regression. For points u in a neighborhood of the target point x, define the polynomial

$$g_x(u; a) = a_0 + \sum_{k=1}^{D} a_j (u - x)^k$$

for some vector of coefficients $a = (a_0, \ldots, a_D)$. As above, we will estimate the coefficients a in $g_x(u; a)$ at some target point x using weighted least squares:

$$\hat{a} = \arg\min_{R^{D+1}} \sum_{i=1}^{n} w_i \left\{y_i - g_x(x_i; a)\right\}^2,$$

where $w_i \equiv w(x_i, x)$ are the kernel weights defined just above, normalized to sum to one. Derive a concise (matrix) form of the weight vector $\hat{a}$, and by extension, the local function estimate $\hat{f}(x)$ at the target value x. Life will be easier if you define the matrix R_x whose (i, j) entry is $(x_i - x)^{j-1}$, and remember that (weighted) polynomial regression is the same thing as (weighted) linear regression with a polynomial basis.

$$\begin{aligned}
\hat{a} &= \arg\min_{R^{D+1}} \sum_{i=1}^{n} w_i \left\{ y_i - \left[a_0 + \sum_{k=1}^{D} a_j (x_i - x)^k \right] \right\} \\
&= \arg\min_{R^{D+1}} \sum_{i=1}^{n} w_i \left\{ y_i - a_0 - a_1(x_i - x) - a_2(x_i - x)^2 - \ldots - a_D(x_i - x)^D \right\} \\
&= \arg\min_{R^{D+1}} (y - Xa)^T W (y - Xa)
\end{aligned}$$

$$\begin{aligned}
0 &= \frac{d}{da}\left[(y - Xa)^T W (y - Xa)\right] \\
&= \frac{d}{da}\left[(y^T - a^T X^T) W (y - Xa)\right] \\
&= \frac{d}{da}\left[y^T W y - y^T W X a - a^T X^T W y + a^T X^T W X a\right] \\
&= \frac{d}{da}\left[y^T W y - 2 y^T W X a + a^T X^T W X a\right]
\end{aligned}$$

$\frac{\partial (z^T A z)}{dz} = (A + A^T) z$, $\frac{\partial b^T z}{dz} = b$

$$\begin{aligned}
0 &= -2 y^T W X + 2 X^T W X a \\
y^T W X &= X^T W X a
\end{aligned}$$

$$\hat{a} = (X^T W X)^{-1} y^T W X$$

$$\hat{a} = Hy$$

$H = (X^T W X)^{-1} X^T W$

$\hat{f}(x) = \begin{bmatrix} 1 & 0 & \ldots \end{bmatrix} \hat{a}$

(B) From this, conclude that for the special case of the local linear estimator ($D = 1$), we can write $\hat{f}(x)$ as a linear smoother of the form

$$\hat{f}(x) = \frac{\sum_{i=1}^n w_i(x) y_i}{\sum_{i=1}^n w_i(x)},$$

where the unnormalized weights are

$$\begin{aligned} w_i(x) &= K\left(\frac{x - x_i}{h}\right)\{s_2(x) - (x_i - x)s_1(x)\} \\ s_j(x) &= \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right)(x_i - x)^j . \end{aligned}$$

$$\begin{aligned} \hat{f}(x) &= \begin{bmatrix} 1 & 0 \end{bmatrix} \left\{ \begin{bmatrix} (x_1-x)^0 & (x_1-x) \\ \vdots & \vdots \\ (x_n-x)^0 & (x_1-x) \end{bmatrix}^T \begin{bmatrix} w_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & w_n \end{bmatrix} \begin{bmatrix} (x_1-x)^0 & (x_1-x) \\ \vdots & \vdots \\ (x_n-x)^0 & (x_1-x) \end{bmatrix} \right\}^{-1} \begin{bmatrix} (x_1-x)^0 & (x_1-x) \\ \vdots & \vdots \\ (x_n-x)^0 & (x_1-x) \end{bmatrix}^T \begin{bmatrix} w_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & w_n \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \sum_{i=1}^n w_i & \sum_{i=1}^n w_i(x_i - x) \\ \sum_{i=1}^n w_i(x_i - x) & \sum_{i=1}^n w_i(x_i - x)^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum_{i=1}^n w_i y_i \\ \sum_{i=1}^n (x_i - x) w_i y_i \end{bmatrix} \end{aligned}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

$$\begin{aligned} \hat{f}(x) &= \left[\sum_{i=1}^n w_i \sum_{i=1}^n w_i(x_i - x)^2 - \left(\sum_{i=1}^n w_i(x_i - x)\right)^2\right]^{-1} \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \sum_{i=1}^n w_i(x_i - x)^2 & -\sum_{i=1}^n w_i(x_i - x) \\ -\sum_{i=1}^n w_i(x_i - x) & \sum_{i=1}^n w_i \end{bmatrix} \begin{bmatrix} \sum_{i=1}^n w_i y_i \\ \sum_{i=1}^n (x_i - x) w_i y_i \end{bmatrix} \\ &= \left[\sum_{i=1}^n w_i \sum_{i=1}^n w_i(x_i - x)^2 - \left(\sum_{i=1}^n w_i(x_i - x)\right)^2\right]^{-1} \left[\sum_{i=1}^n w_i(x_i - x)^2 \sum_{i=1}^n w_i y_i - \sum_{i=1}^n w_i(x_i - x) \sum_{i=1}^n w_i y_i (x_i - x)\right] \end{aligned}$$

$s_1 = \sum_{i=1}^n w_i(x_i - x)$ $s_2 = \sum_{i=1}^n w_i(x_i - x)^2$

$$\hat{f}(x) = \frac{\sum_{i=1}^n w_i y_i (s_2 - (x_i - x)s_1)}{\sum_{i=1}^n w_i (s_2 - (x_i - x)s_1)}$$

(C) Suppose that the residuals have constant variance σ^2 (that is, the spread of the residuals does not depend on x). Derive the mean and variance of the sampling distribution for the local polynomial estimate $\hat{f}(x)$ at some arbitrary point x. Note: the random variable $\hat{f}(x)$ is just a scalar quantity at x, not the whole function.

$$\begin{aligned} \mathrm{E}\left[\hat{f}(x)\right] &= \mathrm{E}\left[\begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix} H y\right] \\ &= \begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix} H \mathrm{E}[y] \\ &= \begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix} H f(x) \end{aligned}$$

$y_i = f(x_i) + \epsilon_i$

(D) We don't know the residual variance, but we can estimate it. A basic fact is that if x is a random vector with mean μ and covariance matrix Σ, then for any symmetric matrix Q of appropriate dimension, the quadratic form $x^T Q x$ has expectation

$$E(x^T Q x) = \mathrm{tr}(Q\Sigma) + \mu^T Q \mu .$$

Write the vector of residuals as $r = y - \hat{y} = y - Hy$, where H is the smoothing matrix. Compute the expected value of the estimator

$$\hat{\sigma}^2 = \frac{\|r\|_2^2}{n - 2\text{tr}(H) + \text{tr}(H^T H)},$$

and simplify things as much as possible. Roughly under what circumstances will this estimator be nearly unbiased for large n? Note: the quantity $2\text{tr}(H) - \text{tr}(H^T H)$ is often referred to as the "effective degrees of freedom" in such problems.

$$\begin{aligned}
\text{E}\left[\hat{\sigma}^2\right] &= \text{E}\left[\frac{(y - Hy)^T (y - Hy)}{n - 2\text{tr}(H) + \text{tr}(H^T H)}\right] \\
&= \frac{\text{E}[(y - Hy)^T (y - Hy)]}{n - 2\text{tr}(H) + \text{tr}(H^T H)} \\
&= \frac{\text{E}[y^T y] - 2\text{E}[y^T H y] + \text{E}[y^T H^T H y]}{n - 2\text{tr}(H) + \text{tr}(H^T H)}
\end{aligned}$$

$$\begin{aligned}
\text{E}[y^T y] &= \text{tr}(\Sigma) + f(x)^T f(x) \\
&= n\sigma^2 + f(x)^T f(x) \\
\text{E}[y^T H y] &= \text{tr}(H\Sigma) + f(x)^T H f(x) \\
&= \sigma^2 \text{tr}(H) + f(x)^T H f(x) \\
\text{E}[y^T H^T H y] &= \text{tr}(H^T H \Sigma) + f(x)^T H^T H f(x) \\
&= \sigma^2 \text{tr}(H^T H) + f(x)^T H^T H f(x)
\end{aligned}$$

$$\begin{aligned}
\text{E}\left[\hat{\sigma}^2\right] &= \frac{n\sigma^2 + f(x)^T f(x) - 2\sigma^2 \text{tr}(H) - 2 f(x)^T H f(x) + \sigma^2 \text{tr}(H^T H) + f(x)^T H^T H f(x)}{n - 2\text{tr}(H) + \text{tr}(H^T H)} \\
&= \frac{\sigma^2 [n - 2\text{tr}(H) +^2 \text{tr}(H^T H)] + f(x)^T f(x) - 2 f(x)^T H f(x) + f(x)^T H^T H f(x)}{n - 2\text{tr}(H) + \text{tr}(H^T H)} \\
&= \sigma^2 + \frac{\|f(x) - Hf(x)\|_2^2}{n - 2\text{tr}(H) + \text{tr}(H^T H)}
\end{aligned}$$

Unbiased when the difference, $f(x) - Hf(x)$ is small

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
```

```
        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

    def MSE(self, y_test):
        """
        Parameters
        ----------
            y_test: float
                Response vector

        Returns
        ----------
            MSE: float
                Average squared error for y_test given y
                .. math:: MSE \\approx \\frac{1}{n}\\sum(y_test - y_star)**2
        """
        self.y_test = y_test
        return np.mean((y_test - self.y_star)**2)

    def optimize_h(self, y_test):
        """
        Parameters
        ----------
            y_test: float
                Response vector
```

```
        Returns
        ----------
            h_star: float
                Uses BFGS minimization method to minimize the MSE by varying h_star
        """
        self.y_test = y_test
        def func(x):
            Y = kernel_smoother(self.x, self.y, self.x_star, h=x)
            Y.predictor()
            return Y.MSE(self.y_test)
        h_star = minimize(func, self.h)
        self.h = h_star.x
        return h_star.x
```