hw7_arkam2

March 20, 2021

1 STAT 542: Homework 7

1.1 Name - Arka Mitra (netid - arkam2)

1.1.1 Spring 2021, by Ruoqing Zhu (rqzhu)

Due: Tuesday, Mar 23, 11:59 PM CT

1.2 Instruction

About HW7 Question 1 [65 Points] One-dimensional Kernel Regression

Question 2 [35 Points] Multi-dimensional Kernel

1.2.1 About HW7

Kernel regression involves two decisions: choosing the kernel and tuning the bandwidth. Usually, tuning the bandwidth is more influential than choosing the kernel function. Tuning the bandwidth is similar to tuning k in a KNN model. However, this is more difficult in multi-dimensional models. We practice one and two-dimensional kernels that involves these elements.

1.3 Question 1 [65 Points] One-dimensional Kernel Regression

For this question, you should only use the base package and write all the main kernel regression mechanism by yourself. We will use the same `ozone` data in HW6. Again, for Question 1, we only use `time` as the covariate, while in Question 2, we use both `time` and `wind`.

```
library(mlbench)
data(Ozone)

# Wind will only be used for Q2
mydata = data.frame("time" = seq(1:nrow(Ozone))/nrow(Ozone), "ozone" = Ozone$V4, "wind" = Ozon

trainid = sample(1:nrow(Ozone), 250)
train = mydata[trainid, ]
test = mydata[-trainid, ]

par(mfrow=c(1,2))
plot(train$time, train$ozone, pch = 19, cex = 0.5)
plot(train$wind, train$ozone, pch = 19, cex = 0.5)
```

Consider two kernel functions:

Gaussian kernel, defined as $K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}}$

Epanechnikov kernel, defined as $K(u) = \frac{3}{4}(1 - u^2)$ for $|u| \leq 1$.

For both kernel functions, incorporate a bandwidth h. You should start with the Silverman's rule-of-thumb for the choice of h, and then tune h. You need to perform the following:

Using the Silverman's rule-of-thumb, fit and plot the regression line with both kernel functions, and plot them together in a single figure. Report the testing MSE of both methods.

Base on our theoretical understanding of the bias-variance trade-off, select two h values for the Gaussian kernel: a value with over-smoothing (small variance and large bias); a value with under-smoothing (large variance and small bias), and plot the two curves, along with the Gaussian rule-of-thumb curve, in a single figure. Clearly indicate which curve is over/under-smoothing.

For the Epanechnikov kernel, tune the h value (on a grid of 10 different h values) by minimizing the testing data. Plot your optimal regression line.

2 Solution

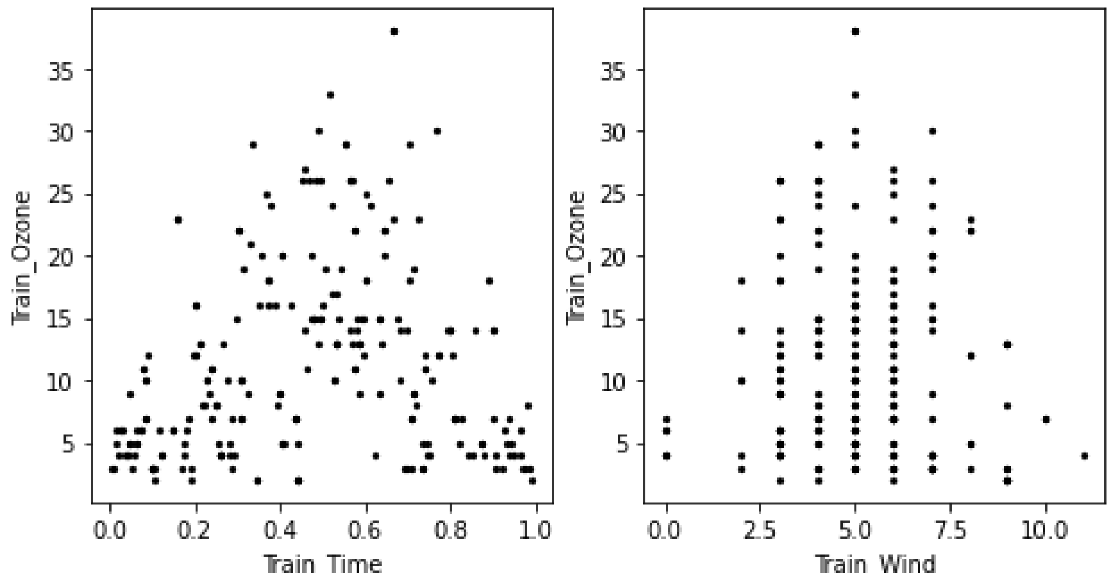
As in HW6, since the `mlbench` library does not exist in Python, we ran the following code to write out the necessary dataset into a .csv file. The rest of the analysis is done on the `oz.csv` file (attached with submission).

```
library(mlbench)
data(Ozone)
mydata = data.frame("time" = seq(1:nrow(Ozone))/nrow(Ozone), "ozone" = Ozone$V4, "wind" = Ozone$
write.csv(mydata, "oz.csv")
```

To check the veracity of the data from R, identical figures to the ones provided are recreated below.

[1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

mydata = pd.read_csv("oz.csv").dropna()
np.random.seed(5)
train_id = np.random.randint(0, len(mydata), 250)
train_df, test_df = mydata.iloc[train_id] , mydata[~mydata.index.isin(train_id)]
train_df = train_df.sort_values('time')

fig, axes = plt.subplots(1,2,figsize = (8,4))
ax1, ax2 = axes[0], axes[1]
ax1.scatter(train_df['time'], train_df['ozone'], s=5, color='k')
ax1.set_xlabel('Train_Time')
ax1.set_ylabel('Train_Ozone')
ax2.scatter(train_df['wind'], train_df['ozone'], s=5, color='k')
ax2.set_xlabel('Train_Wind')
ax2.set_ylabel('Train_Ozone')
```

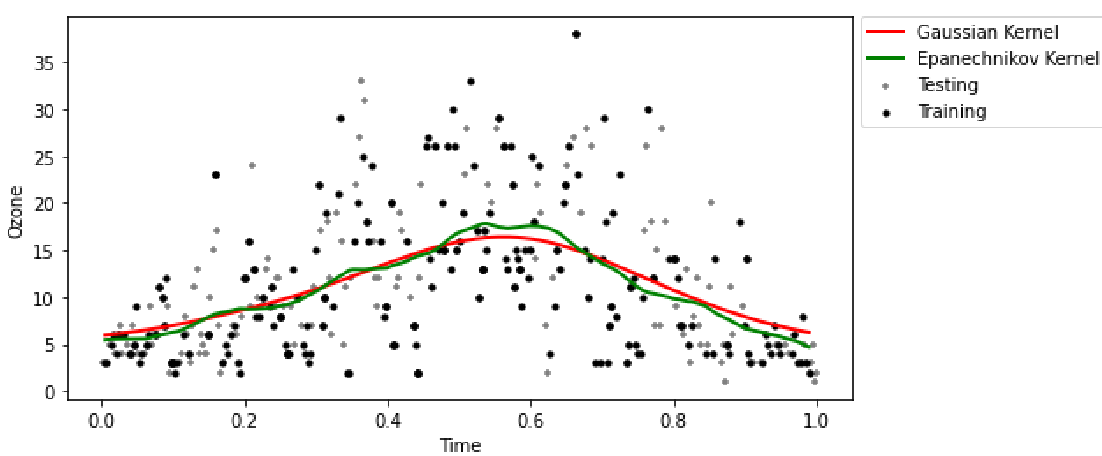
```
plt.show()
```




```
[6]: from sklearn.metrics import mean_squared_error as MSE

def gaussianKernel(x1, x2, h = 1.0):
    '''
    1D Gaussian kernel.

    Variables:
    x1  (Input) :: Array of m points (m x 1)
    x2  (Input) :: Array of n points (n x 1)

    Returns 1D Gaussian Kernel (m x n).
    '''
    x1, x2 = x1.reshape(-1,1), x2.reshape(-1,1)
    eulerdist = (np.sum(x1**2, 1).reshape(-1, 1) + np.sum(x2**2, 1) - 2 * np.
 ↪dot(x1, x2.T)) / h**2
    ker = np.exp(-0.5 * eulerdist)

    return (1/(h * np.sqrt(2*np.pi))) * ker


def epanechnikovKernel(x1, x2, h = 1.0):
    '''
    Gaussian kernel. Computes
    a kernel matrix from points in x1 and x2.
```

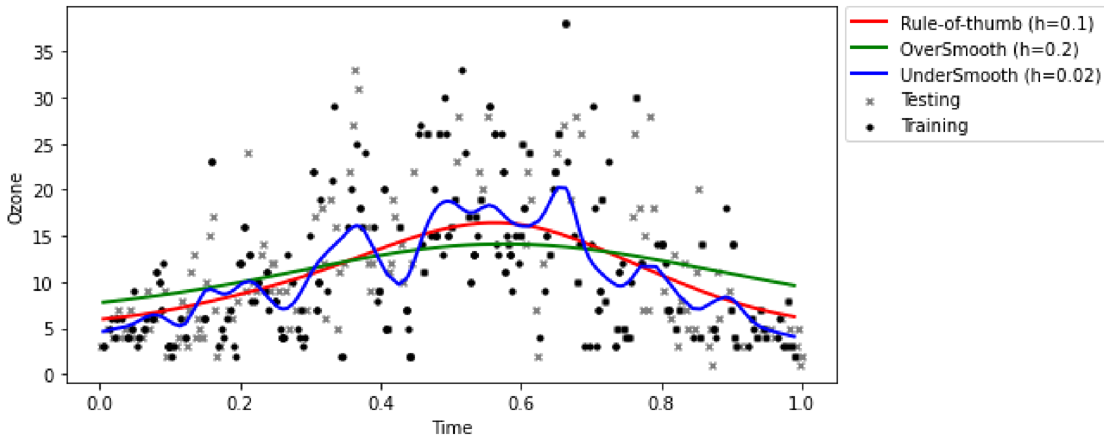
```
    Variables:
    x1 (Input) :: Array of m points (m x 1)
    x2 (Input) :: Array of n points (n x 1)

    Returns 1D Epanechnikov Kernel (m x n).
    '''
    x1, x2 = x1.reshape(-1,1) , x2.reshape(-1,1)
    eulerdist = (np.sum(x1**2, 1).reshape(-1,1) + np.sum(x2**2, 1) - 2 * np.
 ↪dot(x1, x2.T)) / h**2
    eulerdist[eulerdist > 1 ] = 1 #Ensure that distances <= 1

    return 0.75*(1 - eulerdist) / h


#Kernel Regression Function
def kernelRegresion(xtrain, ytrain, xtest, bandwidth = None, kernel =␣
 ↪'Gaussian'):
    '''
    1D kernel regression fit

    Variables:
    xtrain (Input)     :: Training Points (n x 1)
    ytrain (Input)     :: Training Outputs (n x 1)
    xtest (Input)      :: Testing Points (m x 1)
    bandwidth (Input)  :: Parameter for the kernel
            [If none provided, uses Silverman's Estimation(def.)]
    kernel  (Input)    :: Name of the Kernel to be used.
            ['gauss' : Gaussian Kernel,
            'epchen' : Epanechnikov Kernel]
    ytest  (Output) :: Fitted regression on to xtest (m x 1)
    '''
    n = xtrain.shape[0]

    if bandwidth == None:
        h = 1.06 * np.std(xtrain) * n**(-0.2)
    else:
        h = bandwidth

    if kernel == 'gauss':
        kernelmat = gaussianKernel(xtest, xtrain, h)
    elif kernel == 'epchen':
        kernelmat = epanechnikovKernel(xtest, xtrain, h)
    else:
        print("Kernel name can be either 'gauss' or 'epchen'. Check and re-enter!
 ↪")
        return 0
```

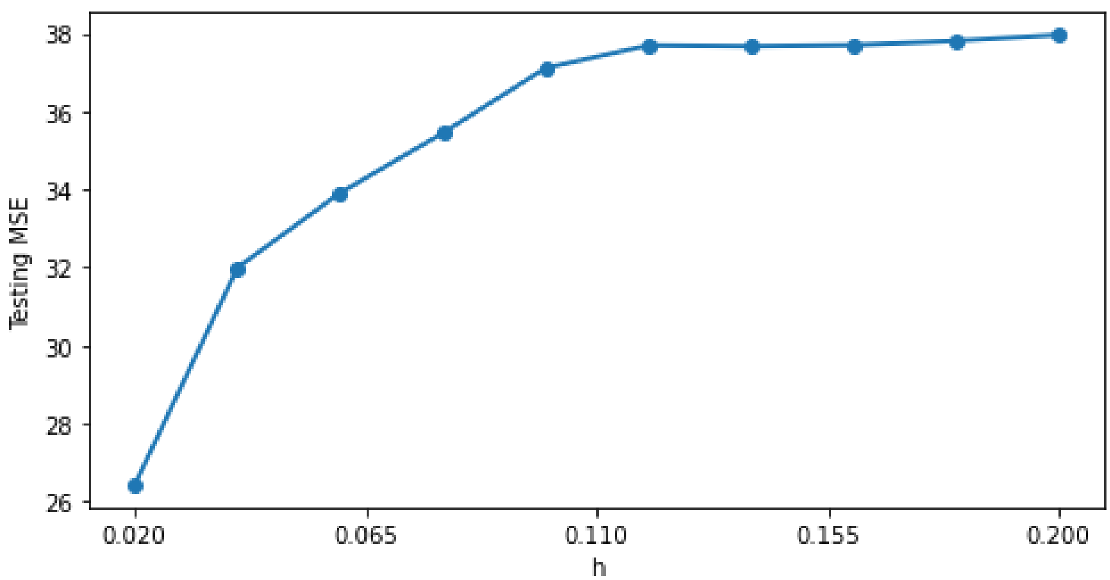
```
    ytest = kernelmat.dot(ytrain) / np.sum(kernelmat, axis = 1)

    return ytest

xtrain, ytrain = np.array(train_df['time']) , np.array(train_df['ozone'])
xtest, ytest = np.array(test_df['time']) , np.array(test_df['ozone'])

y_gaussian_train = kernelRegresion(xtrain, ytrain, xtrain, kernel = 'gauss')
y_epc_train = kernelRegresion(xtrain, ytrain, xtrain, kernel = 'epchen')

y_gaussian_predicted = kernelRegresion(xtrain, ytrain, xtest, kernel = 'gauss')
y_epc_predicted = kernelRegresion(xtrain, ytrain, xtest, kernel = 'epchen')

plt.figure(figsize = (8,4))
plt.scatter(xtest, ytest, s = 15, color = 'gray', marker = '+', label = 'Testing')
plt.scatter(xtrain, ytrain, s = 10, color = 'black', marker = 'o', label = 'Training')
plt.plot(xtrain, y_gaussian_train, label = 'Gaussian Kernel', lw = 2, color='red')
plt.plot(xtrain, y_epc_train, label = 'Epanechnikov Kernel', lw = 2, color='green')

plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0)
plt.xlabel('Time')
plt.ylabel('Ozone')
plt.show()

print("Testing MSE for Gaussian Kernel = ", np.round(MSE(y_gaussian_predicted, ytest),3))
print("Testing MSE for Epanechnikov Kernel = ", np.round(MSE(y_epc_predicted, ytest),3))
```



```
Testing MSE for Gaussian Kernel =  38.132
Testing MSE for Epanechnikov Kernel =  36.91
```

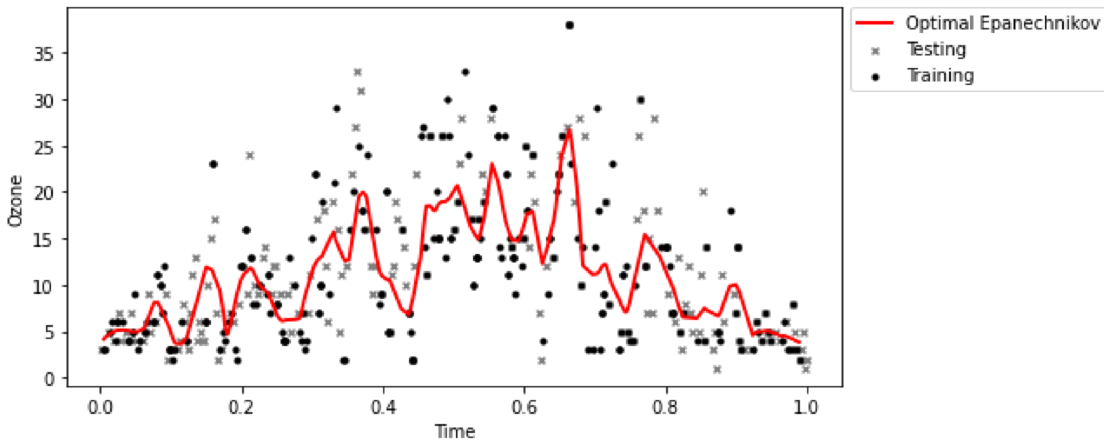
[3]:
```python
h_silverman = 1.06 * np.std(xtrain) * xtrain.shape[0]**(-0.2)
h_over_smooth = 0.2
h_under_smooth = 0.02

y_over_smooth = kernelRegresion(xtrain, ytrain, xtrain, h_over_smooth, kernel =␣
 ↪'gauss')
y_under_smooth = kernelRegresion(xtrain, ytrain, xtrain, h_under_smooth, kernel␣
 ↪= 'gauss')

plt.figure(figsize = (8,4))
plt.scatter(xtest, ytest, s = 15, color = 'gray', marker = 'x', label =␣
 ↪'Testing')
plt.scatter(xtrain, ytrain, s = 10, color = 'black', marker = 'o', label =␣
 ↪'Training')
plt.plot(xtrain, y_gaussian_train, label = 'Rule-of-thumb (h='+str(np.
 ↪round(h_silverman,2))+')', lw = 2,color='r')
plt.plot(xtrain, y_over_smooth, label = 'OverSmooth (h='+str(np.
 ↪round(h_over_smooth,2))+')', lw = 2,color='g')
plt.plot(xtrain, y_under_smooth, label = 'UnderSmooth (h='+str(np.
 ↪round(h_under_smooth,2))+')', lw = 2,color='b')

plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0)
plt.xlabel('Time')
plt.ylabel('Ozone')
plt.show()

#Choose the grid such that h_silverman is at the (approx.) center
h_grid = np.linspace(0.02, 0.20, 10)
testingMSE = []
for h in h_grid:
    y_epc_predicted = kernelRegresion(xtrain, ytrain, xtest, h, kernel =␣
 ↪'epchen')
    testingMSE.append(MSE(y_epc_predicted, ytest))

plt.figure(figsize = (8,4))
plt.plot(h_grid, testingMSE, lw = 2)
plt.scatter(h_grid, testingMSE)
plt.xticks(np.linspace(0.02, 0.20, 5))
plt.xlabel('h')
plt.ylabel('Testing MSE')
plt.show()
```





[4]:
```
y_best_h = kernelRegresion(xtrain, ytrain, xtrain, 0.02, kernel = 'epchen')

plt.figure(figsize = (8,4))
plt.scatter(xtest, ytest, s = 15, color = 'gray', marker = 'x', label =␣
 ↪'Testing')
plt.scatter(xtrain, ytrain, s = 10, color = 'black', marker = 'o', label =␣
 ↪'Training')
plt.plot(xtrain, y_best_h, label = 'Optimal Epanechnikov', lw = 2, color='r')

plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left', borderaxespad=0)
plt.xlabel('Time')
```

```
plt.ylabel('Ozone')
plt.show()
```



2.1 Question 2 (35 Points) Multi-dimensional Kernel

We consider using both time and wind in the regression. We use the following multivariate kernel function, which is essentially a Gaussian kernel with diagonal covariance matrix.

$$K_\lambda(x_i, x_j) = e^{-\frac{1}{2}\sum_{k=1}^{p}\left((x_{ik}-x_{jk})/\lambda_k\right)^2}$$

Based on Silverman's formula, the bandwidth for the kth variable is given by

$$\lambda_k = \left(\frac{4}{p+2}\right)^{\frac{1}{p+4}} n^{-\frac{1}{p+4}} \widehat{\sigma}_k,$$

where $\widehat{\sigma}_k$ is the estimated standard deviation for variable k, p is the number of variables, and n is the sample size. Use the Nadaraya-Watson kernel estimator to fit and predict the ozone level.

Calculate the prediction error and compare this to the univariate model in Question 1. Provide a discussion (you do not need to implement them) on how this current two-dimensional kernel regression can be improved. Provide at least two ideas that could potentially improve the performance.

[5]:
```
#Changes made to the above functions to accommodate for 2D covariates
def gaussianKernel2D(X1, X2, h):
    '''
    2D Gaussian kernel.

    Variables:
    x1  (Input) :: Array of m points (m x 2)
    x2  (Input) :: Array of n points (n x 2)
```

```python
    Returns 2D Gaussian Kernel (m x n).
    '''

    x1, y1 = X1[:,0].reshape(-1,1), X1[:,1].reshape(-1,1)
    x2, y2 = X2[:,0].reshape(-1,1), X2[:,1].reshape(-1,1)

    x_euldist = np.matrix((x1.T - x2)**2) / h[0]**2
    y_euldist = np.matrix((y1.T - y2)**2) / h[1]**2

    return np.asarray(np.exp(-0.5 * (x_euldist + y_euldist)))


def kernelRegresion2D(xtrain, ytrain, xtest, bandwidth = None):
    '''
    2D kernel regression fit

    Variables:
    xtrain (Input)     :: Training Points (n x 1)
    ytrain (Input)     :: Training Outputs (n x 1)
    xtest (Input)      :: Testing Points (m x 1)
    bandwidth (Input)  :: Parameter for the kernel
            [If none provided, uses Silverman's Estimation(def.)]
 ↪
    ytest  (Output) :: Fitted regression on to xtest (m x 1)
    '''

    n = xtrain.shape[0]
    p = xtrain.shape[1]

    if bandwidth == None:
        h = (4/(p+2))**(1/(p+4)) * n**(-1/(p+4)) * np.std(xtrain, 0)
    else:
        h = bandwidth

    kernelmat = gaussianKernel2D(xtrain, xtest, h)
    ytest = kernelmat.dot(ytrain) / np.sum(kernelmat, axis = 1)

    return ytest

xtrain, ytrain = train_df.loc[:, ['time', 'wind']].values , np.
 ↪array(train_df['ozone'])
xtest, ytest = test_df.loc[:, ['time', 'wind']].values , np.
 ↪array(test_df['ozone'])
ypredicted = kernelRegresion2D(xtrain, ytrain, xtest, bandwidth = None)
print("Testing MSE for 2D Gaussian Kernel =", np.round(MSE(ypredicted, ytest),3))
```

```
Testing MSE for 2D Gaussian = 35.076
```

Testing MSE for 2D model (35.076) is lower than MSE for univariate model testing (38.132).

A possible ways to improve the 2-D kernel regression is by performing Cross-Validation to tune λ_k instead of using the default from Silverman's formula will decrease testing MSE.

```
[ ]:
```