Lecture 5 - Jan 24

Numerical Optimization
Gradient Descent

Recommended Reading

Week 3 Notes on GitHub

Upcoming Deadlines

Homework 1 (Jan 27)

Recall from Calc 1: Optimization

Let $f:[a,b]\to\mathbb{R}$ be a differentiable function

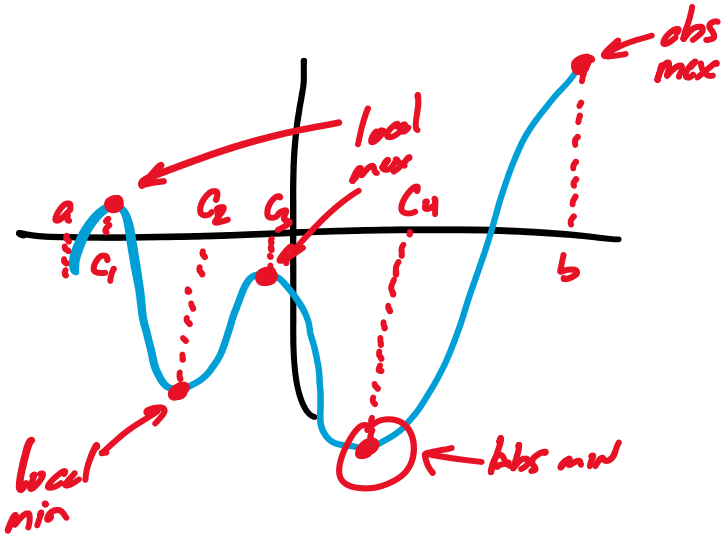
From differential calculus, we know there exists an absolute minimum located at $x=a$, $x=b$, or at some $x=c\in(a,b)$ s.t. $f'(c)=0$.

at a critical value

to find it... ① Find all critical values in (a,b)

② Compare f at $x=a$, $x=b$, and at each critical value

↳ longest output = abs min



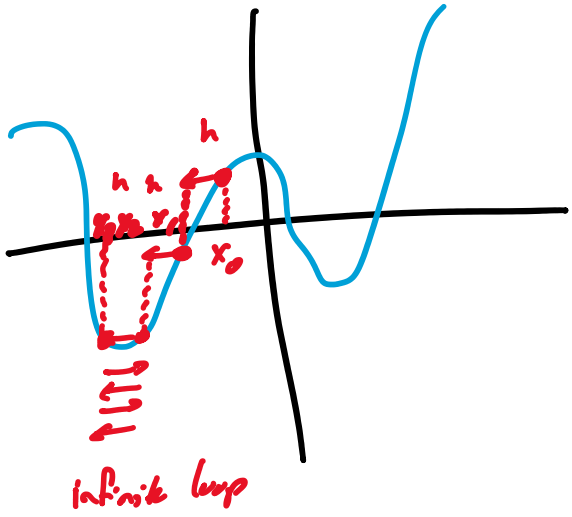
Sometimes computing $f'(x)$ or solving $f'(x)=0$ by hand is tricky. In this case, we can use

Sometimes computing $f(x)$ or solving $f'(x) = 0$ by hand is very difficult. In this case, we can use a numerical approach to approximate f' or find x where $f'(x)$ is approximately 0.

1D Gradient Descent

If we cannot find critical values, we can search for x-values where $|f'(x)| < \varepsilon$ for some fixed small $\varepsilon > 0$.

One popular method is called gradient descent



step size

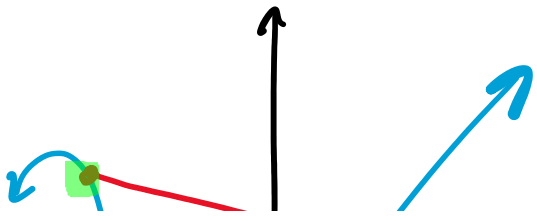
1. choose some x_0
2. if $f'(x_0) > 0$, move left by h
   if $f(x_0) < 0$, move right by h
   to $x_1 = x_0 \pm h$
3. repeat until $|f'| < \varepsilon$ or max steps reached.

tolerance

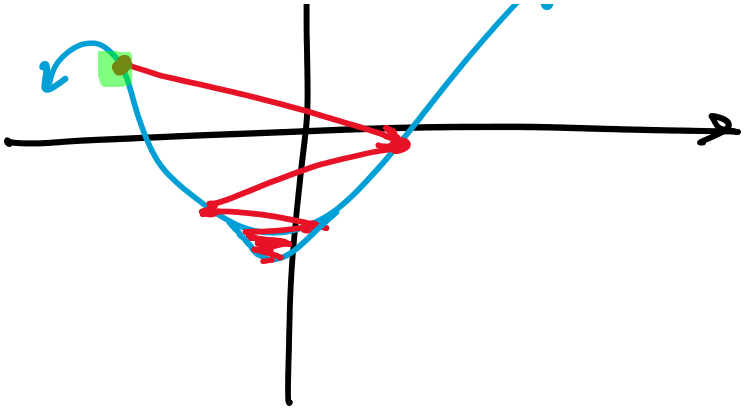
(or if repetitive movement starts)

For tiny h, this is fine. But accuracy is limited by h and many calls to f are required.

To speed it up and improve accuracy, we can make step size proportional to slope. (large steps when it is steep, small steps when it is flatter, e.g. near crit values)



$$\rightarrow x_1 = x_0 - \alpha f'(x_0)$$

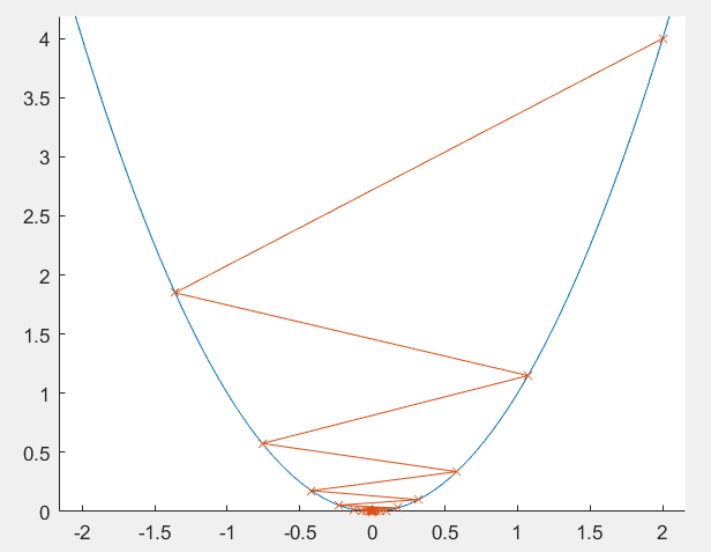


$$\rightarrow x_1 = x_0 - \alpha f'(x_0)$$

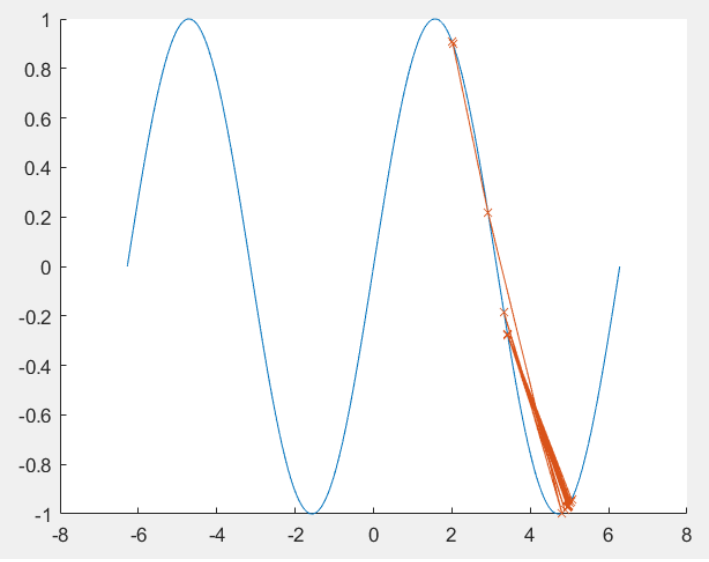
learning rate

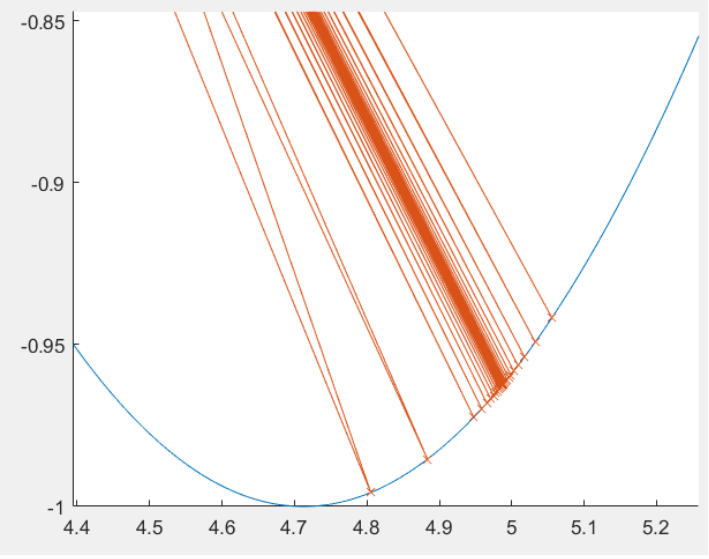
What Can Go Wrong

f(x) = x^2, x0 = 2, alpha = 0.4 (good convergence)

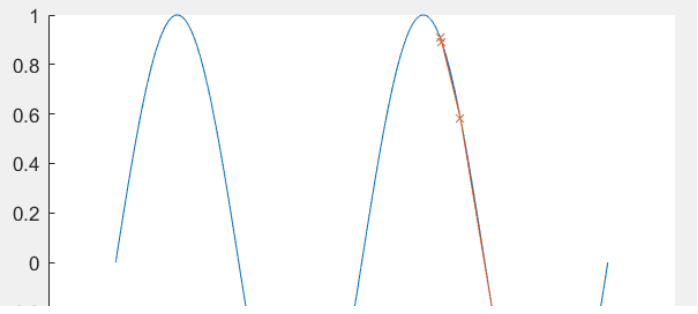


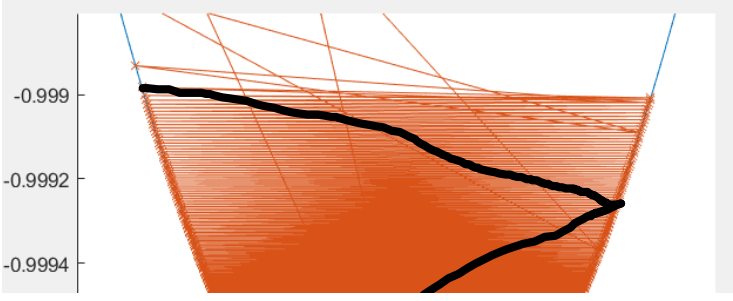
f(x) = sin(x), x0 = 2, alpha = 2 -- alpha is too big, it gets "stuck" (right picture is zoomed-in)

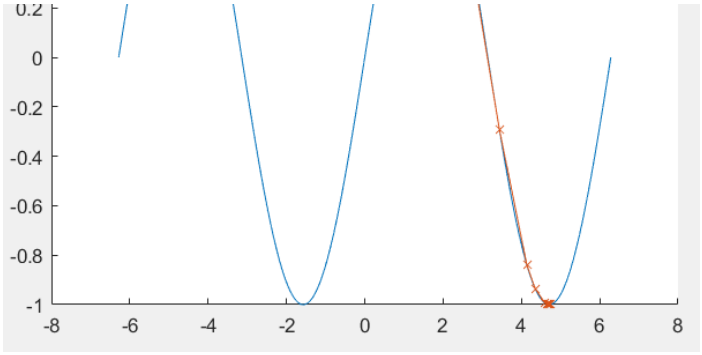


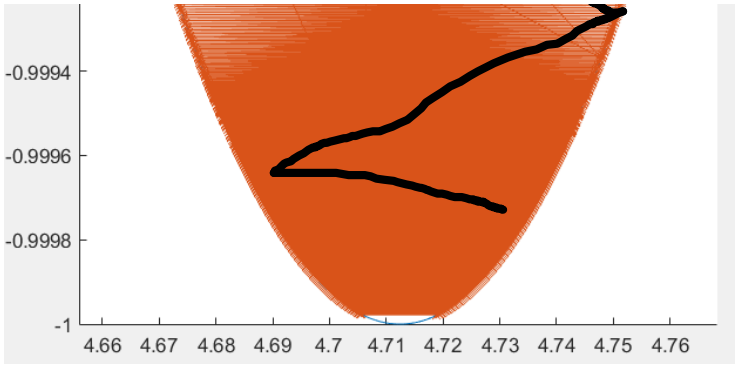


f(x) = sin(x), x0 = 2, alpha = 1 -- alpha still too big, it's working, BUT it's going incredibly slowly

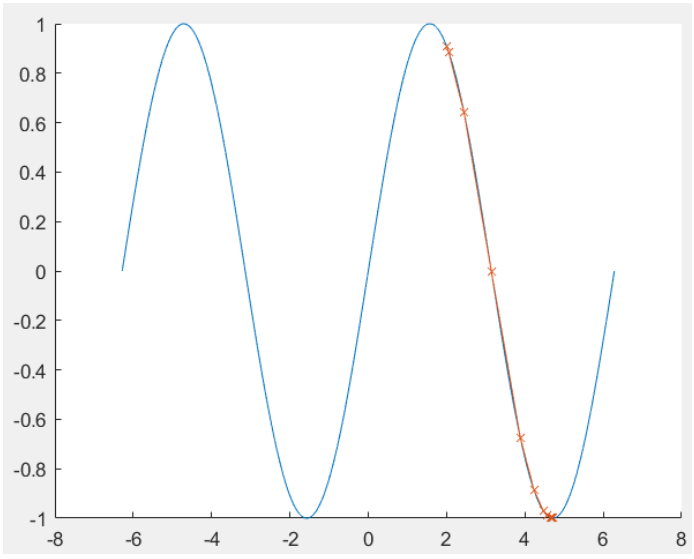


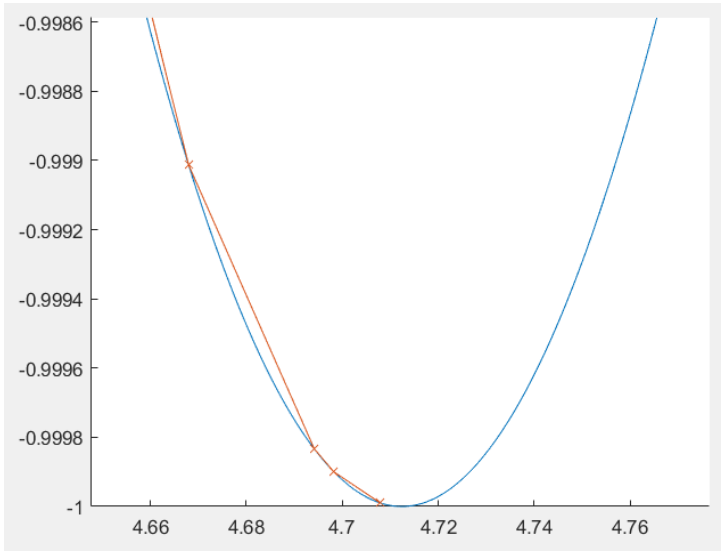




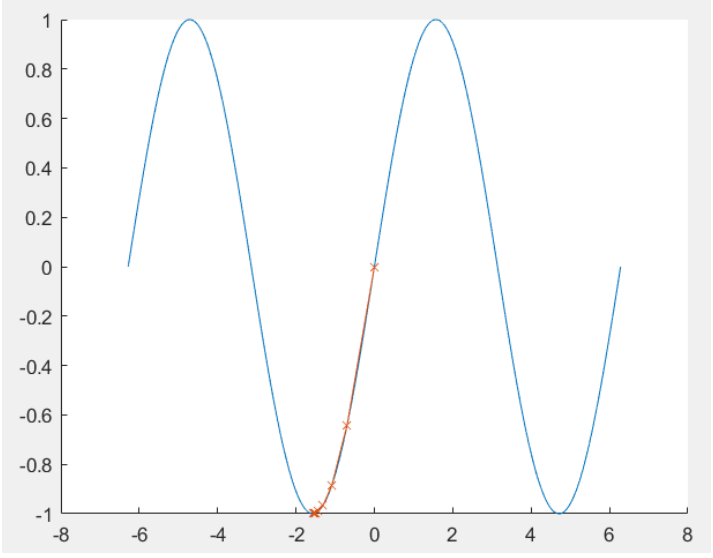


f(x) = sin(x), x0 = 2, alpha = 0.8 -- good convergence!





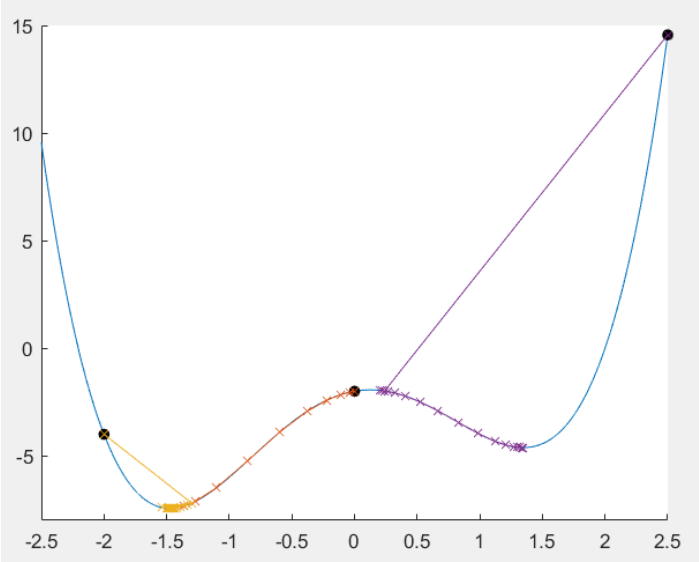
f(x) = sin(x), x0 = 0, alpha = 0.4 -- good convergence, but it goes to a **different** minimum



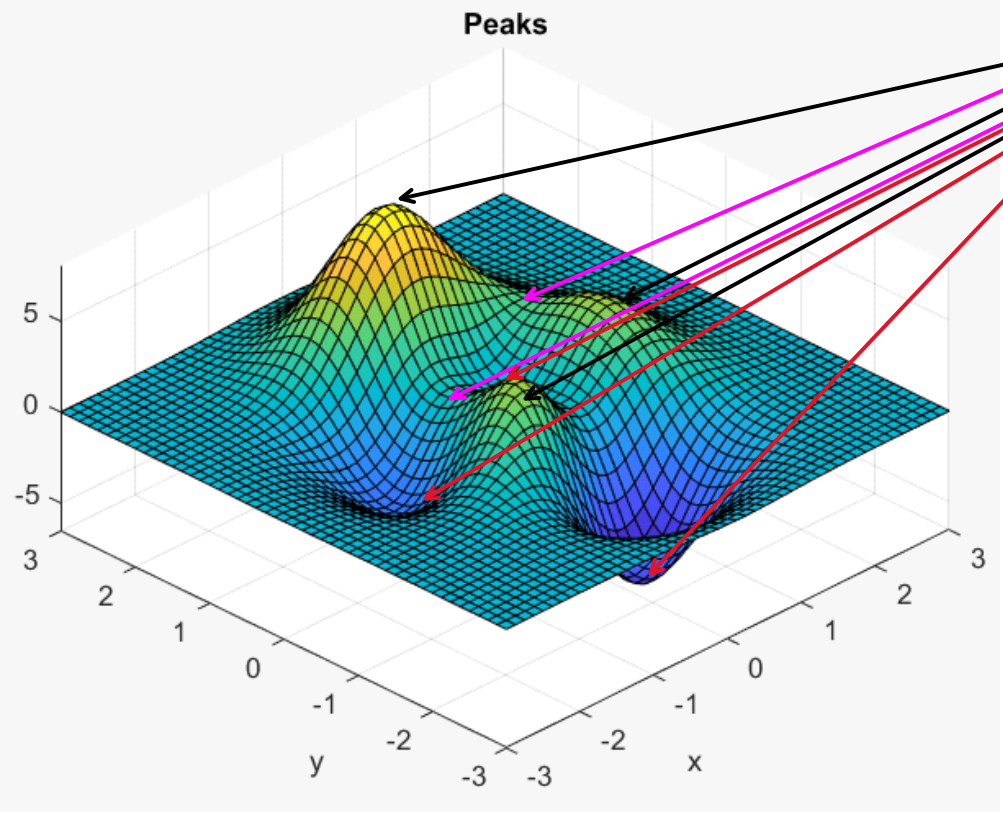
Start at 3 points on the graph of $x^4 - 4x^2 + x - 2$…

x0 = 0, alpha = 0.05
x0 = -2, alpha = 0.05
x0 = 2.5, alpha = 0.05

It converges to different minima **depending on starting point** -- the two on the left hit the global minimum as we want, but the one on the right goes to a local minimum



Multivariate Calculus: Find θ such that $\nabla L(\theta) = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$ (crit. points)



Critical values (where $\nabla L(\theta) = 0$) are "flat spots" on the graph.

Red = local **minima**

Black = local **maxima**

Magenta = **saddle points**

ML: sometimes I can do this by hand (ordinary least squares), but usually I cannot.

Analytic solution Fails → Approximate it numerically

How do I find the critical point(s) numerically?

(1) Make a guess x_0

(1) Make a guess x_0

(2) Take a step downhill to x_1

(3) Take a step downhill to x_2

⋮

(?) Stop (somehow)

If $L: \mathbb{R}^{d+1} \to \mathbb{R}$, the partial derivative of L with respect to θ_i is

$$L_{\theta_i}(\theta) = \frac{\partial L}{\partial \theta_i} = \lim_{h \to 0} \frac{L(\theta_0, \dots, \theta_{i-1}, \theta_i + h, \theta_{i+1}, \dots, \theta_d) - L(\theta)}{h}$$

If we can have derivatives in the directions $\langle 1,0 \rangle$ and $\langle 0,1 \rangle$ ... why not some other direction?

The directional derivative of L in the direction u (unit vector) is

$$D_u L(\theta) = \lim_{h \to 0} \frac{L(\theta + hu) - L(\theta)}{h}$$

Graphics: Directional Derivative

Questions (1) How do I go "downhill"?

Choose a direction u where $D_u L$ is most negative

Choose a direction u where -

(steepest path down)

Theorem: $D_u L(\theta)$ is most negative in the opposite direction of $\nabla L(\theta)$

"Learning rate"

$$\Rightarrow \quad p_1 = p_0 - \alpha \nabla L(p_0)$$

$$p_2 = p_1 - \alpha \nabla L(p_1)$$

$$\vdots$$

(2) How do I know when to stop?

Ideally, when $\nabla L(x_n) = 0$, but this is unlikely or even impossible with code, so lets stop if

$$\nabla L(x_n) \approx 0$$

"tolerance"

For example, if $\|\nabla L(x_n)\| < \varepsilon$ for some small fixed ε.

Method...

(1) Guess x_0

(2) Compute $\nabla L(x_0)$

while $\|\nabla L(x_i)\| > \varepsilon$

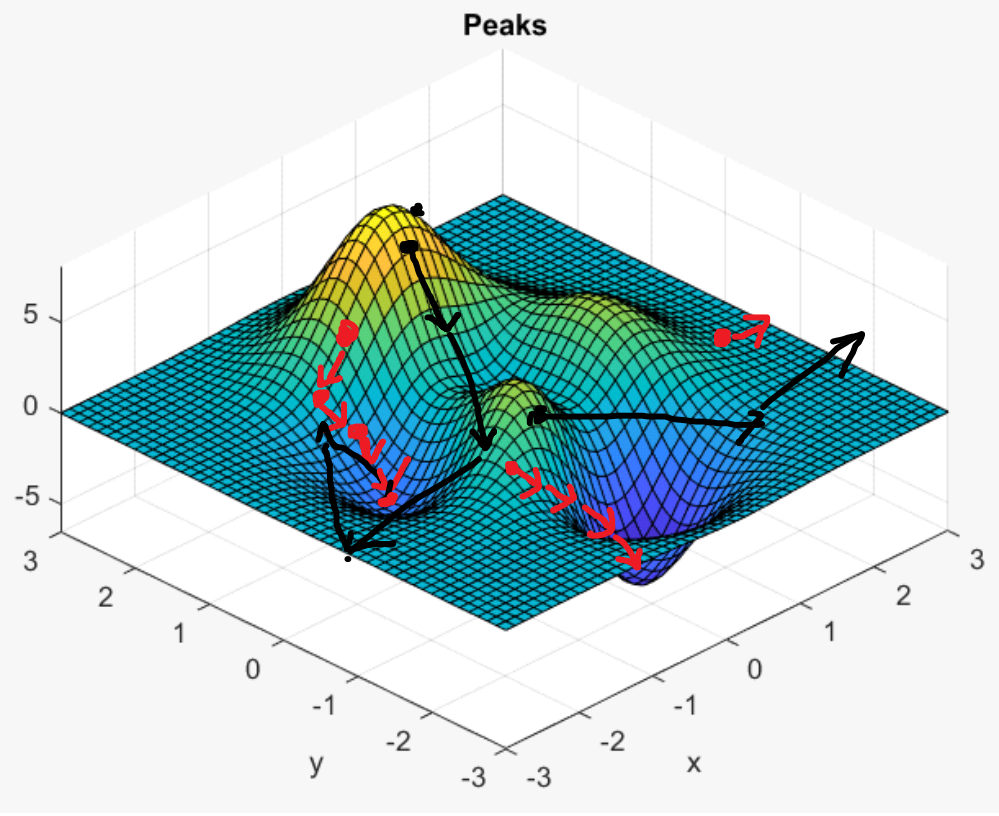
↱ step to $x_{i+1} = x_i - \alpha \nabla L(x_i)$

while

(3) step to $x_{i+1} = x_i - \alpha \nabla L(x_i)$

sometimes we can compute it exactly, sometimes we have to approximate

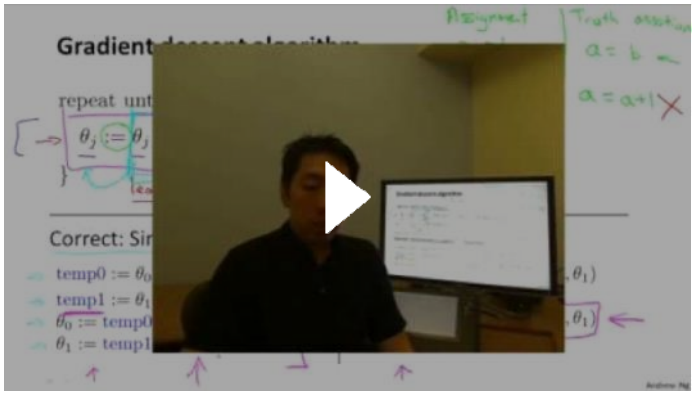
(we do this today)



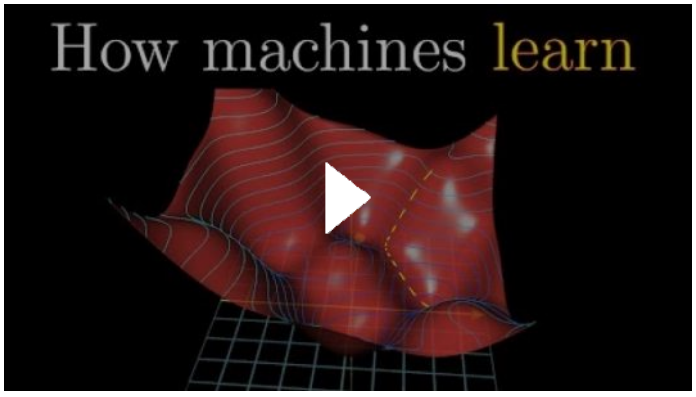
Significance of Gradient Descent

It is worth mentioning just how significant gradient descent is as an optimization method.

- Gradient-based optimization is **THE** most popular numerical approach for optimization
- It solves a wide variety of problems
  - Finding minima of functions with n inputs
  - Finding maxima of functions with n inputs (just use -f(x) to convert maxima to minima)
  - Solving non-linear systems of equations
- It is used for **many** machine learning methods:
  - Calculating coefficients in regression models
    ([Lecture 2.5 — Linear Regression With One Variable | Gradient Descent — [Andrew Ng]])



  - Optimizing weights in neural networks
    ([Gradient descent, how neural networks learn | Deep learning, chapter 2])



- There are many popular implementations in various languages (https://ruder.io/optimizing-gradient-descent/)

Implementing Gradient Descent

Let's import some libraries then write a function for gradient descent.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import scale
```

Gradient descent will need a few inputs:

- A function to minimize f
- A starting point x_0
- A learning rate α
- A small number h (the variable that goes to 0 in the definition of the derivative)
- A small positive value that we can use for a stopping condition for the derivative being sufficiently small (the tolerance)
- A maximum number of iterations

```
def computeGradient(f, x, h):
    n = len(x)
    gradient = np.zeros(n)

    for counter in range(n):
        xUp = x.copy()
        xUp[counter] += h
        gradient[counter] = (f(xUp) - f(x))/h

    return gradient
```

find dimension of x
initialize gradient vector
increment ith element
≈ partial derivative of ith dimension

```
# run gradient descent ant output the
def gradientDescent(f, x0, alpha, h, tolerance, maxIterations):
    # set x equal to the initial guess
    x = x0

    # take up to maxIterations number of steps
    for counter in range(maxIterations):
        # update the gradient
        gradient = computeGradient(f, x, h)

        # stop if the norm of the gradient is near 0
        if np.linalg.norm(gradient) < tolerance:
            #print('Gradient descent took', counter, 'iterations to converge')
            #print('The norm of the gradient is', np.linalg.norm(gradient))
            # return the approximate critical value x
            return x

        # if we do not converge, print a message
        elif counter == maxIterations-1:
            #print("Gradient descent failed")
            #print('The gradient is', gradient)
            # return x, sometimes it is still pretty good
```

stopping conditions

```
        elif counter == maxIterations-1:
            #print("Gradient descent failed")
            #print('The gradient is', gradient)
            # return x, sometimes it is still pretty good
            return x

        # take a step in the opposite direction as the gradient
        x -= alpha*gradient
```

take a step
$x \to x - \alpha \nabla f(x)$

OLS by Gradient Descent

Ideally, we want to use gradient descent for radial basis function expansions, for which we were unable to find an explicit formula for model parameters because the loss function does not have a unique solution, but let's start with a simpler problem: finding an approximate solution for the ordinary least squares problem.

Recall that, in this situation, we have some labeled datapoints $(x_{i1}, \ldots, x_{id}, y_i)$, where we fit a function of the form

$$f_\theta(x_i) = \theta_0 + \sum_{j=1}^{d} \theta_j x_{ij}$$

To find the least squares solution, we aim to solve the minimization problem $\min_\theta L(\theta)$ for $\theta = (\theta_0, \theta_1, \ldots, \theta_d) \in \mathbb{R}^{d+1}$, where the loss function is

$$L(\theta) = \sum_{i=1}^{n} (y_i - f_\theta(x_i))^2 = \sum_{i=1}^{n} \left(y_i - \theta_0 - \sum_{j=1}^{d} \theta_j x_{ij} \right)^2 = \|X\theta - y\|_2^2 = (X\theta - y)^T (X\theta - y)$$

Let's bring in our exact OLS class for linear regression from Week 1.

```
class OrdinaryLeastSquaresExact:
```

In addition, let's write an OLS class using gradient descent instead.

```
class OrdinaryLeastSquaresGradient:

    # fit the model to the data
    def fit(self, X, y, x0, alpha, h, tolerance, maxIterations):
        self.n = X.shape[0]
        self.d = X.shape[1]
        self.h = h
        self.alpha = alpha
        self.initialGuess = x0

        # save the training data
        self.data = np.hstack((np.ones([self.n, 1]), X))

        # save the training labels
        self.outputs = y

        # find the theta values that minimize the sum of squared errors via gradient descent
        X = self.data
        L = lambda theta: ((X @ theta).T - y.T) @ (X @ theta - y)
        self.theta = gradientDescent(L, self.initialGuess, self.alpha, self.h, tolerance, maxIterations)

    # predict the output from testing data
    def predict(self, X):
        # initialize an empty matrix to store the predicted outputs
        yPredicted = np.empty([X.shape[0],1])

        # append a column of ones at the beginning of X
        X = np.hstack((np.ones([X.shape[0],1]), X))

        # apply the function f with the values of theta from the fit function to each testing datapoint (rows of X)
        for row in range(X.shape[0]):
            yPredicted[row] = self.theta @ X[row,]

        return yPredicted
```

set class variables

add column of ones to data matrix

"anonymous function"

create the loss function

gradient descent will minimize loss

predict test outputs

1D OLS Example

```
X = np.array([[6], [7], [8], [9], [7]])
y = np.array([1, 2, 3, 3, 4])

print('FOR THE GRADIENT-BASED ORDINARY LEAST SQUARES CODE \n')

# instantiate an OLS (gradient) object, fit to data, predict data
model = OrdinaryLeastSquaresGradient()
model.fit(X, y, [0, 0], alpha = 0.001, h = 0.001, tolerance = 0.01, maxIterations = 100000)
predictions = model.predict(X)

# print the predictions
print('\nThe predicted y values are', predictions.T[0])

# print the real y values
print('The real y values are', y)

# print the theta values
parameters = model.theta
print('The theta values are', parameters)

# plot the training points
plt.scatter(X, y, label = 'Training Data')

# plot the fitted model with the training data
xModel = np.linspace(6,10,100)
yModel = parameters[0] + parameters[1]*xModel
lineFormula = 'y={:.3f}+{:.3f}x'.format(parameters[0], parameters[1])
plt.plot(xModel, yModel, 'r', label = lineFormula)

# add a legend
plt.legend()

# return quality metrics
print('The r^2 score is', r2_score(y, predictions))
print('The mean absolute error is', mean_absolute_error(y, predictions),'\n')
```

use gradient descent to estimate θ

plotting

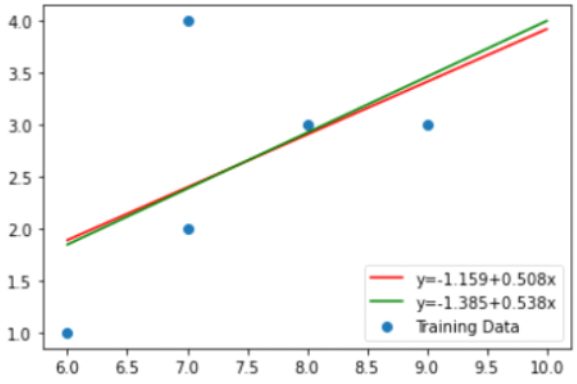
```
print('FOR THE EXACT ORDINARY LEAST SQUARES CODE \n')

# instantiate an OLS (exact) object, fit to data, predict data
model = OrdinaryLeastSquaresExact()
model.fit(X,y)
predictions = model.predict(X)

# print the predictions
print('The predicted y values are', predictions.T[0])

# print the real y values
print('The real y values are', y)

# print the theta values
parameters = model.theta
print('The theta values are', parameters)

# plot the fitted model with the training data
xModel = np.linspace(6,10,100)
yModel = parameters[0] + parameters[1]*xModel
lineFormula = 'y={:.3f}+{:.3f}x'.format(parameters[0], parameters[1])
plt.plot(xModel, yModel, 'g', label = lineFormula)

# add a legend
plt.legend()

# return quality metrics
print('The r^2 score is', r2_score(y, predictions))
print('The mean absolute error is', mean_absolute_error(y, predictions))
```

uses exact formula for θ

plotting

```
# return quality metrics
print('The r^2 score is', r2_score(y, predictions))
print('The mean absolute error is', mean_absolute_error(y, predictions))
```

```
FOR THE GRADIENT-BASED ORDINARY LEAST SQUARES CODE

The predicted y values are [1.88927362 2.39728623 2.90529883 3.41331144 2.39728623]
The real y values are [1 2 3 3 4]
The theta values are [-1.15880201  0.5080126 ]
The r^2 score is 0.28901345877326357
The mean absolute error is 0.6794572453892237

FOR THE EXACT ORDINARY LEAST SQUARES CODE

The predicted y values are [1.84615385 2.38461538 2.92307692 3.46153846 2.38461538]
The real y values are [1 2 3 3 4]
The theta values are [-1.38461538  0.53846154]
The r^2 score is 0.2899408284023671
The mean absolute error is 0.6769230769230786
```



gradient descent learns almost the exact same results

Comments on Gradient Descent

Monday, February 1, 2021 1:12 PM

- We get almost identical results in all of these examples in using a gradient-based method for ordinary least squares.
- It runs a little slower, but not much.
- We must be careful with the h and tolerance hyperparameters to be sure gradient descent will converge.
- Gradient descent in our implementation above does not actually require any derivatives since we only used approximate derivatives.
- If we knew formulas for the derivatives, we could compute them exactly to let the step size be exactly proportional to ∇L. This would drastically reduce the number of times we compute the loss function.
- Gradient descent and related methods are the main driver of many machine learning problems that are based on to minimizing a loss function (least squares and neural networks, among others), although we will later need some variants to reduce the computational burden.

Kernel Smoothing by Gradient Descent

[redacted]

```python
def GaussianKernel(x0, x, lam):
    return (1/lam) * np.exp(-np.linalg.norm(x0 - x) / (2 * lam))

class KernelRegressionGradient:
    def __init__(self, kernel_function, lam, fit_intercept = True):
        self.kernel_function = kernel_function
        self.fit_intercept = fit_intercept
        self.lam = lam

    def predict(self, x0, X, y, theta0, alpha, h, tolerance, maxIterations):
        # find the number of X points
        n = X.shape[0]

        # add a column of ones if needed
        if self.fit_intercept:
            X = np.hstack((np.ones([n,1]), X))

        # construct the kernel matrix
        kernel = np.zeros([n, n])

        # populate the kernel matrix
        for i in range(n):
            kernel[i][i] = self.kernel_function(x0, X[i,:], self.lam)

        L = lambda theta: (X @ theta - y).T @ kernel @ (X @ theta - y)
        self.theta = gradientDescent(L, theta0, alpha, h, tolerance, maxIterations)

        return np.array([1, np.float(x0)]) @ self.theta
```

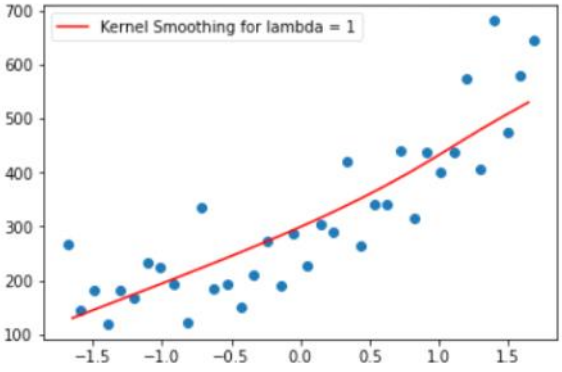
gradient descent version

In the previous class, we put an exact formula here

```python
# read the shampoo sales dataset
data = pd.read_csv('data/shampoo.csv')

# save the targets
y = data['Sales'].to_numpy()

# make a column vector of 0s with n elements
X = np.zeros([y.shape[0], 1])

# convert the vector to (0, 1, 2, ..., n)
X[:,0] = [i for i in range(y.shape[0])]

X = scale(X)

# split the data into train and test sets
(trainX, testX, trainY, testY) = train_test_split(X, y, test_size = 0.25, random_state = 1)
```

```
lam = 1

model = KernelRegressionGradient(GaussianKernel, lam)

# compute the model coordinates
xModel = scale([i for i in range(20)])
yModel = [model.predict(i, trainX, trainY, [0, 0], alpha = 0.001, h = 0.0001, tolerance = 0.01, maxIterations = 100000)
          for i in xModel]

# plot the model
label = 'Kernel Smoothing for lambda = ' + str(np.round(lam, 2))
plt.scatter(X, y)
plt.plot(xModel, yModel, 'r', label = label)
plt.legend()

# apply the functions to the test data and predict with the model
trainPredictions = [model.predict(i, trainX, trainY, [0, 0], alpha = 0.001, h = 0.0001, tolerance = 0.01, maxIterations = 100000)
                    for i in trainX]
testPredictions = [model.predict(i, trainX, trainY, [0, 0], alpha = 0.001, h = 0.0001, tolerance = 0.01, maxIterations = 100000)
                   for i in testX]

# compute the training and test mean absolute error
trainError = mean_absolute_error(trainY, trainPredictions)
testError = mean_absolute_error(testY, testPredictions)

# return quality metrics
print('lambda:', np.round(lam, 3), '\t\tr^2:', np.round(r2_score(trainY, trainPredictions), 3),
      '\t\ttrain MAE:', np.round(trainError, 3), '\t\ttest MAE:', np.round(testError, 3))
```

```
lambda: 1               r^2: 0.719              train MAE: 61.008               test MAE: 41.563
```

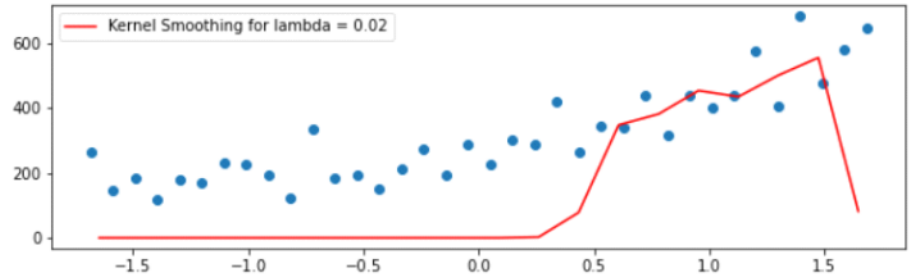


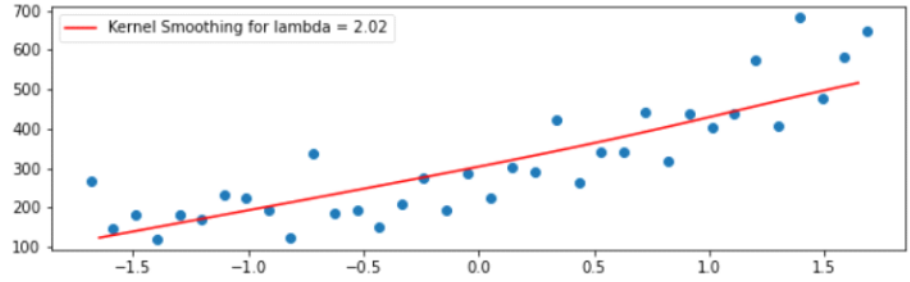
We could run it like last time (below), but it becomes **very** expensive.

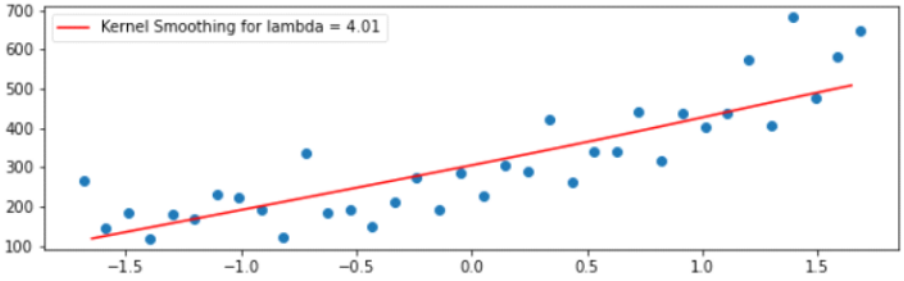
```
lamValues = np.linspace(0.025, 10, 6)

M = lamValues.shape[0]

# allow multiple plots
fig, axes = plt.subplots(nrows = M + 1, figsize = (10, 4*M))

trainMAE = []
testMAE = []

for (j, lam) in enumerate(lamValues):
    print(j)
    model = KernelRegressionGradient(GaussianKernel, lam)

    # compute the model coordinates
    xModel = scale([i for i in range(20)])
    yModel = [model.predict(i, trainX, trainY, [0, 0], alpha = 0.001, h = 0.0001, tolerance = 0.01, maxIterations = 100000)
              for i in xModel]

    # plot the model
    label = 'Kernel Smoothing for lambda = ' + str(np.round(lam, 2))
    axes[j].scatter(X, y)
    axes[j].plot(xModel, yModel, 'r', label = label)
    axes[j].legend()

    # apply the functions to the test data and predict with the model
    trainPredictions = [model.predict(i, trainX, trainY, [0, 0], alpha = 0.001, h = 0.0001, tolerance = 0.01, maxIterations = 100000)
                        for i in trainX]
    testPredictions = [model.predict(i, trainX, trainY, [0, 0], alpha = 0.001, h = 0.0001, tolerance = 0.01, maxIterations = 100000)
                       for i in testX]

    # compute the training and test mean absolute error
    trainError = mean_absolute_error(trainY, trainPredictions)
    testError = mean_absolute_error(testY, testPredictions)
```

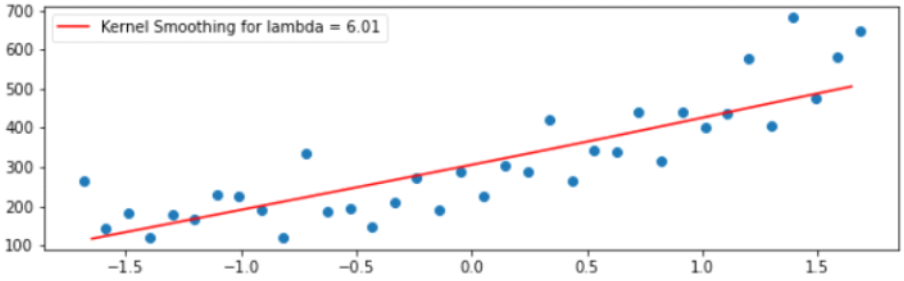
```
    # return quality metrics
    print('lambda:', np.round(lam, 3), '\t\tr^2:', np.round(r2_score(trainY, trainPredictions), 3),
          '\t\ttrain MAE:', np.round(trainError, 3), '\t\ttest MAE:', np.round(testError, 3))

# plot the errors
axes[M].plot(range(M), trainMAE, label = 'Training Mean Absolute Error')
axes[M].plot(range(M), testMAE, label = 'Testing Mean Absolute Error')
axes[M].legend()
```

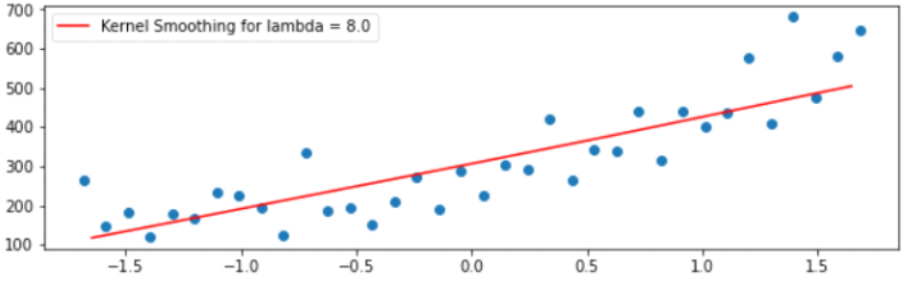
```
0
lambda: 0.025           r^2: -1.368             train MAE: 186.537              test MAE: 189.227
1
lambda: 2.02            r^2: 0.696              train MAE: 63.091               test MAE: 43.103
2
lambda: 4.015           r^2: 0.683              train MAE: 64.169               test MAE: 43.894
3
lambda: 6.01            r^2: 0.678              train MAE: 64.547               test MAE: 44.167
4
lambda: 8.005           r^2: 0.676              train MAE: 64.755               test MAE: 44.304
5
lambda: 10.0            r^2: 0.674              train MAE: 64.881               test MAE: 44.388
```

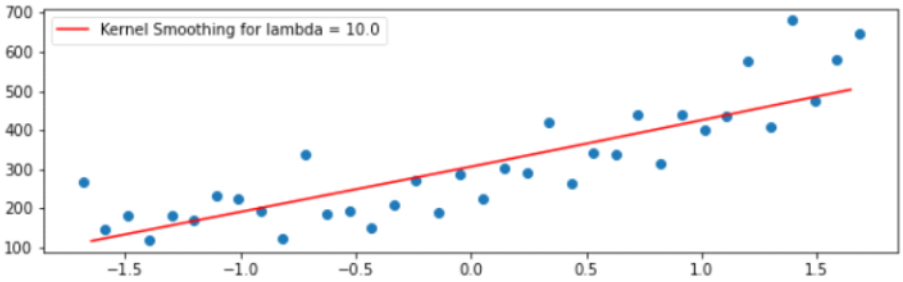

Kernel Smoothing for lambda = 0.02

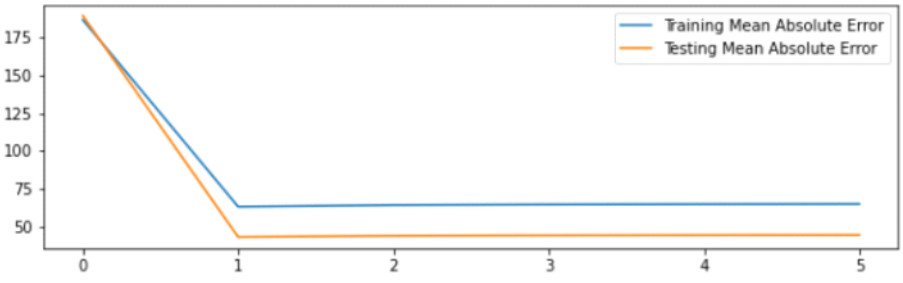

Kernel Smoothing for lambda = 2.02


Kernel Smoothing for lambda = 4.01


Kernel Smoothing for lambda = 6.01


Kernel Smoothing for lambda = 8.0


Kernel Smoothing for lambda = 10.0
700
600
500
400
300
200
100
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Training Mean Absolute Error
Testing Mean Absolute Error
175
150
125
100
75
50
0
1
2
3
4
5