

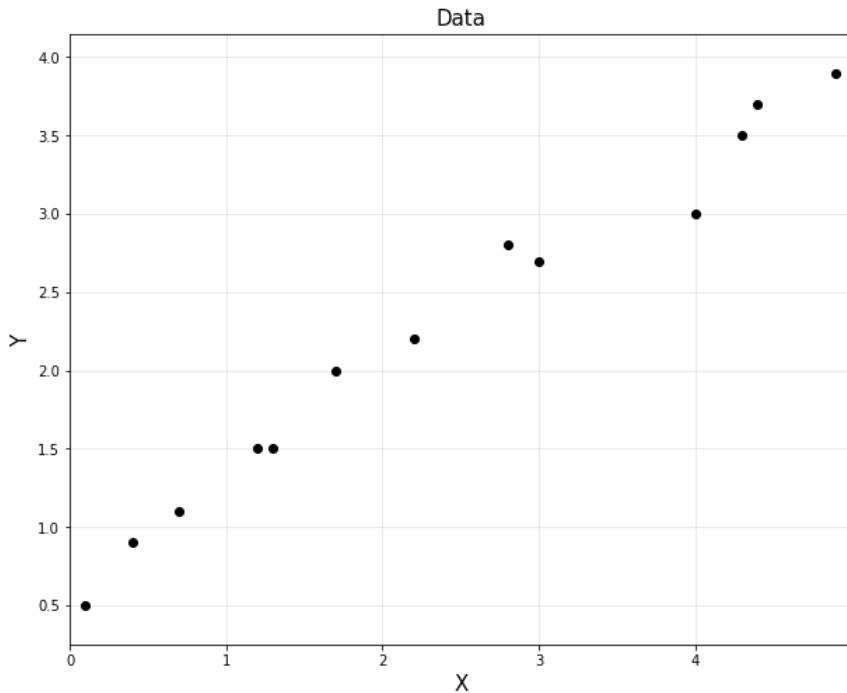


Regression 1

**Prof. Seungchul Lee
Industrial AI Lab.**

Assumption: Linear Model

$$\hat{y}_i = f(x_i ; \theta) \text{ in general}$$

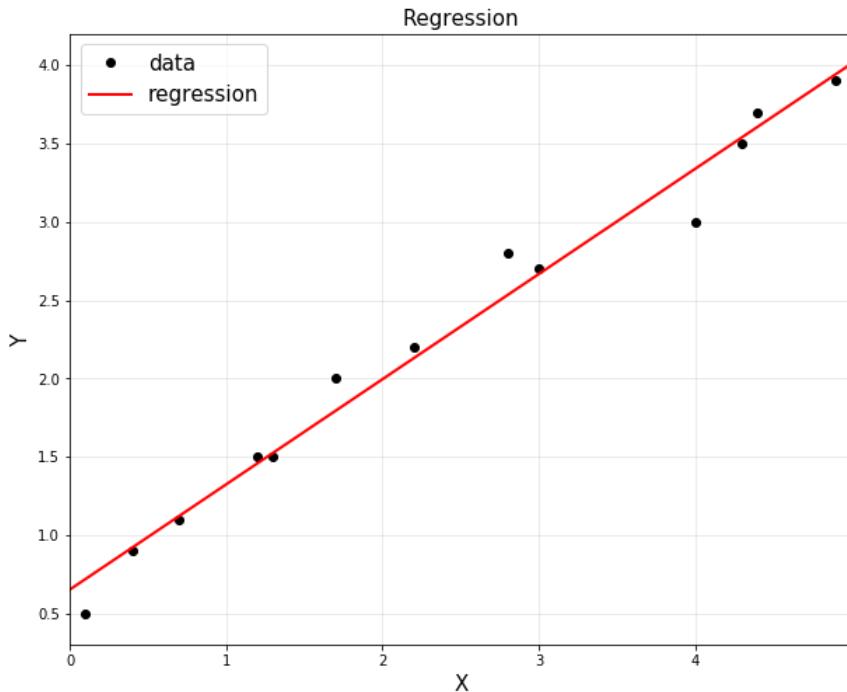


- In many cases, a linear model is used to predict y_i

$$\hat{y}_i = \theta_1 x_i + \theta_2$$

Assumption: Linear Model

$$\hat{y}_i = f(x_i ; \theta) \text{ in general}$$



- In many cases, a linear model is used to predict y_i

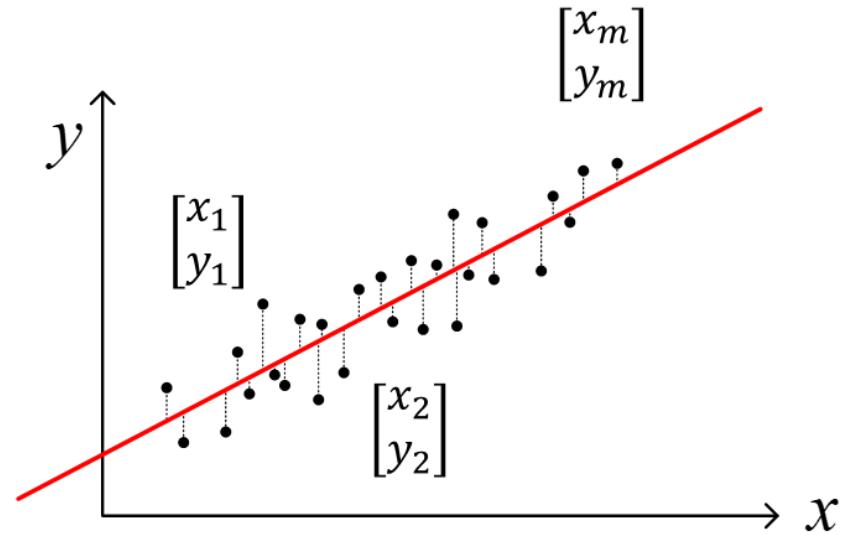
$$\hat{y}_i = \theta_1 x_i + \theta_2$$

Linear Regression

- $\hat{y}_i = f(x_i, \theta)$ in general
- In many cases, a linear model is assumed to predict y_i

Given $\begin{cases} x_i : \text{inputs} \\ y_i : \text{outputs} \end{cases}$, Find θ_0 and θ_1

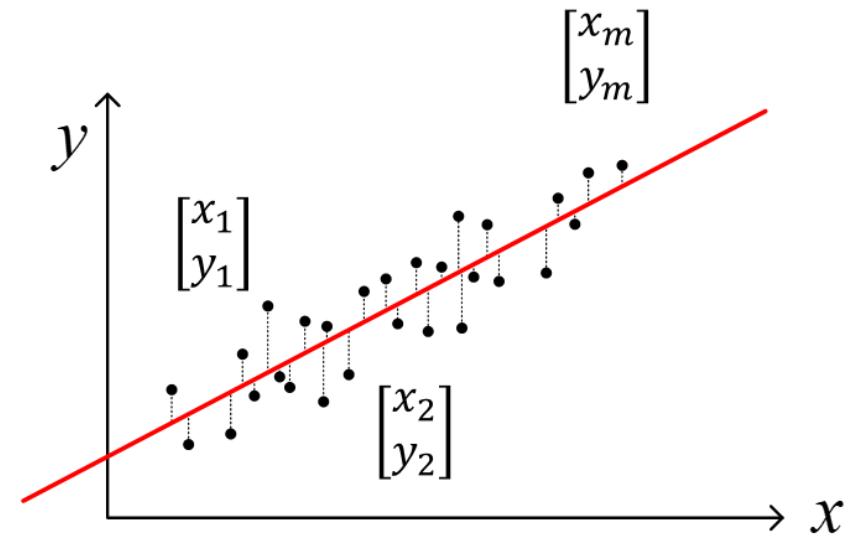
$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \approx \hat{y}_i = \theta_0 + \theta_1 x_i$$



- \hat{y}_i : predicted output
- $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$: model parameters

Linear Regression as Optimization

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \approx \hat{y}_i = \theta_0 + \theta_1 x_i$$



- How to find model parameters $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$
- Optimization problem

$$\hat{y}_i = \theta_0 + \theta_1 x_i$$

such that

$$\min_{\theta_0, \theta_1} \sum_{i=1}^m (\hat{y}_i - y_i)^2$$

$$\min_X \|E\|^2 = \min_X \|AX - B\|^2$$

$$X^* = (A^T A)^{-1} A^T B$$

$$B^* = AX^* = A(A^T A)^{-1} A^T B$$

Re-cast Problem as Least Squares

- For convenience, we define a function that maps inputs to feature vectors, ϕ

$$\hat{y}_i = \theta_0 + x_i\theta_1 = 1 \cdot \theta_0 + x_i\theta_1$$

$$= [1 \quad x_i] \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 \\ x_i \end{bmatrix}^T \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

$$= \phi^T(x_i)\theta$$

feature vector $\phi(x_i) = \begin{bmatrix} 1 \\ x_i \end{bmatrix}$

$$\min_X \|E\|^2 = \min_X \|AX - B\|^2$$

$$X^* = (A^T A)^{-1} A^T B$$

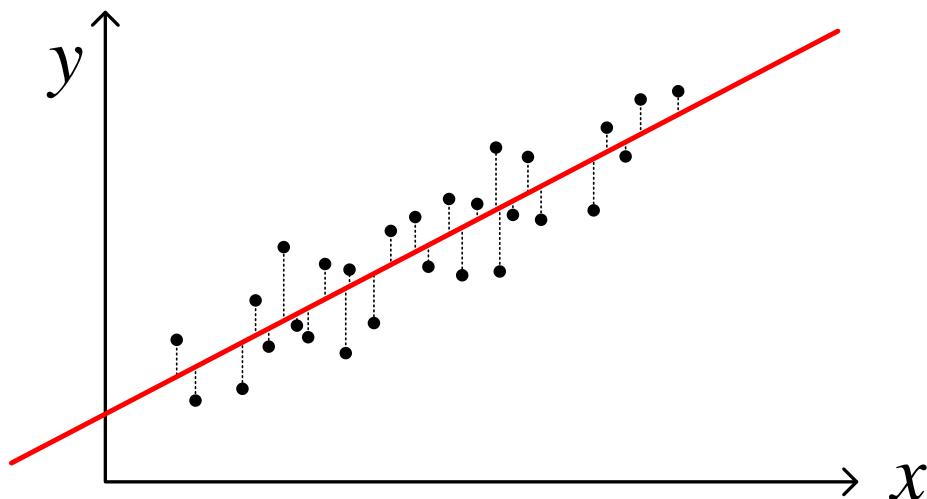
$$B^* = AX^* = A(A^T A)^{-1} A^T B$$

$$\Phi = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix} = \begin{bmatrix} \phi^T(x_1) \\ \phi^T(x_2) \\ \vdots \\ \phi^T(x_m) \end{bmatrix} \implies \hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_m \end{bmatrix} = \Phi\theta$$

Optimization

$$\min_{\theta_0, \theta_1} \sum_{i=1}^m (\hat{y}_i - y_i)^2 = \min_{\theta} \|\Phi\theta - y\|_2^2 \quad \left(\text{same as } \min_x \|Ax - b\|_2^2 \right)$$

$$\text{solution } \theta^* = (\Phi^T \Phi)^{-1} \Phi^T y$$



Optimization: Note

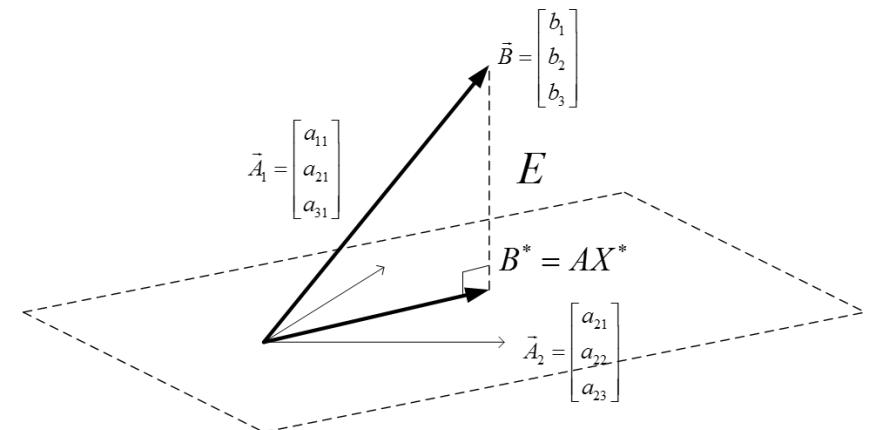
input $x_i \rightarrow$ feature $\begin{bmatrix} 1 \\ x_i \end{bmatrix} \rightarrow$ predicted output \hat{y}_i

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

over-determined or projection

$\uparrow \quad \uparrow \quad \uparrow \quad \uparrow$
 $\vec{A}_1 \quad \vec{A}_2 \quad \vec{x} \quad \vec{B}$

$$A(= \Phi) = [\vec{A}_1 \ \vec{A}_2]$$



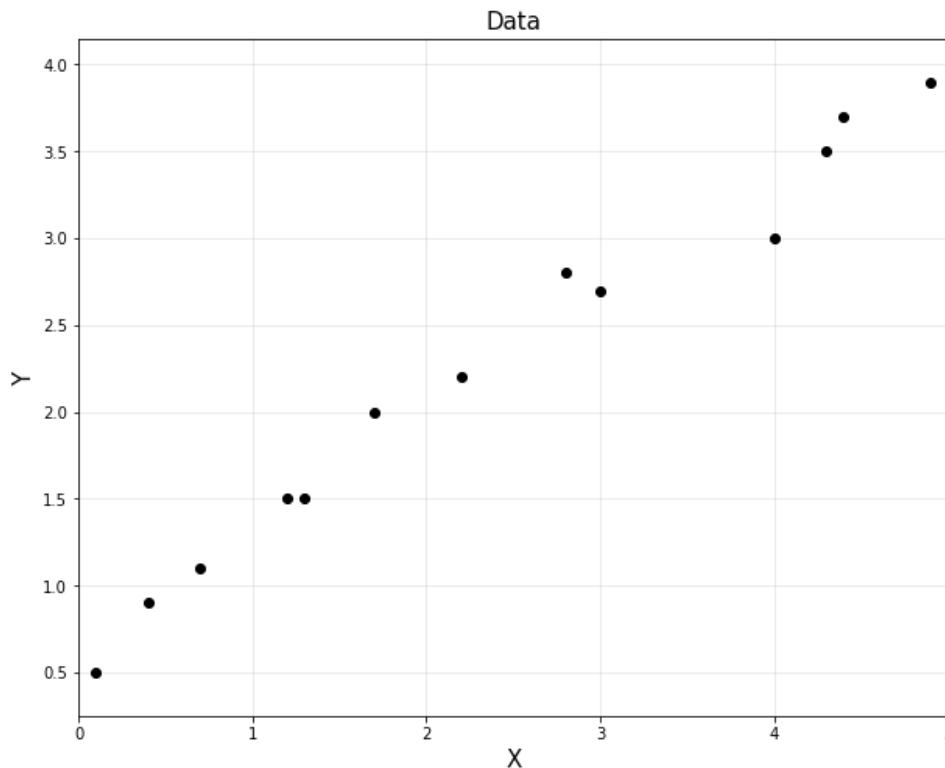
the same principle in a higher dimension

1. Solve using Linear Algebra

- known as *least square*

$$\theta = (A^T A)^{-1} A^T y$$

```
# data points in column vector [input, output]
x = np.array([0.1, 0.4, 0.7, 1.2, 1.3, 1.7, 2.2, 2.8, 3.0, 4.0, 4.3, 4.4, 4.9]).reshape(-1, 1)
y = np.array([0.5, 0.9, 1.1, 1.5, 1.5, 2.0, 2.2, 2.8, 2.7, 3.0, 3.5, 3.7, 3.9]).reshape(-1, 1)
```



1. Solve using Linear Algebra

- known as *least square*

$$\theta = (A^T A)^{-1} A^T y$$

```
m = y.shape[0]
#A = np.hstack([np.ones([m, 1]), x])
A = np.hstack([x**0, x])
A = np.asmatrix(A)

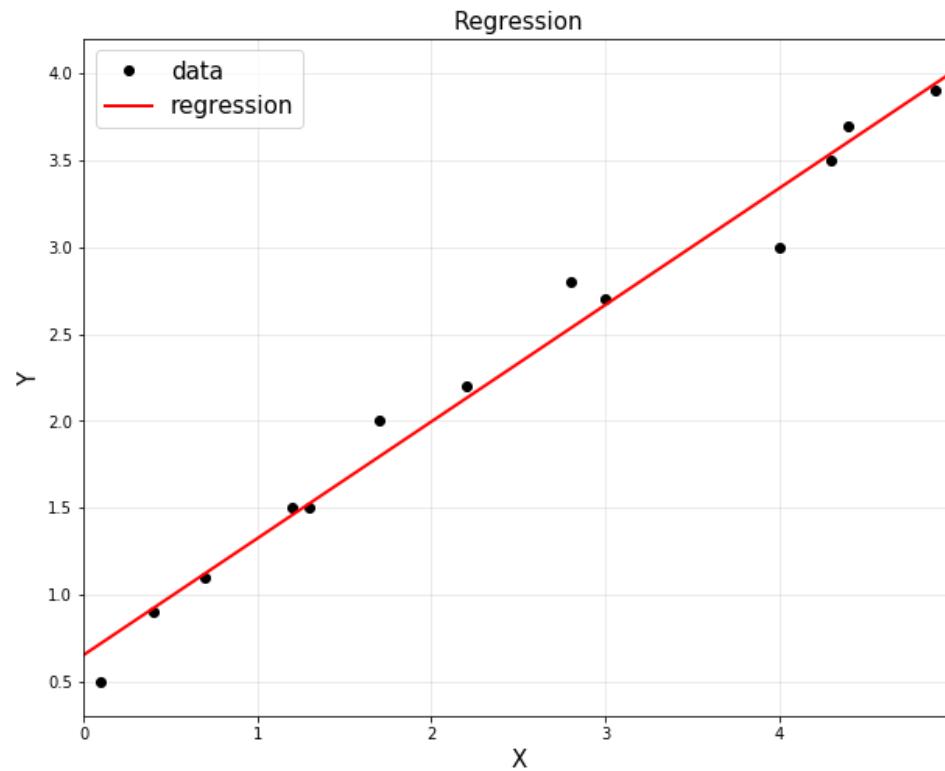
theta = (A.T*A).I*A.T*y

print('theta:\n', theta)
```

```
theta:
[[0.65306531]
 [0.67129519]]
```

```
# to plot
plt.figure(figsize=(10, 8))
plt.title('Regression', fontsize=15)
plt.xlabel('X', fontsize=15)
plt.ylabel('Y', fontsize=15)
plt.plot(x, y, 'ko', label="data")

# to plot a straight line (fitted line)
xp = np.arange(0, 5, 0.01).reshape(-1, 1)
yp = theta[0,0] + theta[1,0]*xp
```



2. Solve using Gradient Descent

$$\begin{aligned}f &= (A\theta - y)^T(A\theta - y) = (\theta^T A^T - y^T)(A\theta - y) \\&= \theta^T A^T A \theta - \theta^T A^T y - y^T A \theta + y^T y\end{aligned}$$

$$\min_{\theta} \|\hat{y} - y\|_2^2 = \min_{\theta} \|A\theta - y\|_2^2$$

$$\nabla f = A^T A \theta + A^T A \theta - A^T y - A^T y = 2(A^T A \theta - A^T y)$$

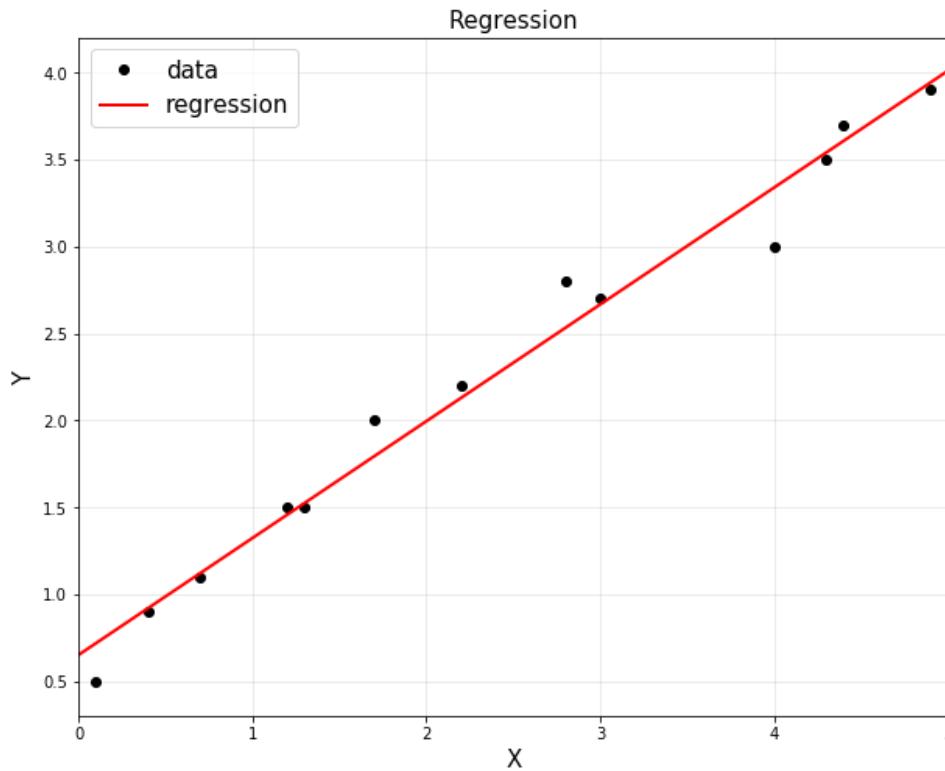
$$\theta \leftarrow \theta - \alpha \nabla f$$

```
theta = np.random.randn(2,1)
theta = np.asmatrix(theta)

alpha = 0.001

for _ in range(1000):
    df = 2*(A.T*A*theta - A.T*y)
    theta = theta - alpha*df

print (theta)
```



3. Solve using CVXPY Optimization

```
theta2 = cvx.Variable([2, 1])
obj = cvx.Minimize(cvx.norm(A*theta2-y, 2))
cvx.Problem(obj,[]).solve()

print('theta:\n', theta2.value)
```

```
theta:
[[0.65306531]
[0.67129519]]
```

$$\min_{\theta} \|\hat{y} - y\|_2 = \min_{\theta} \|A\theta - y\|_2$$

3. Solve using CVXPY Optimization

```
theta2 = cvx.Variable([2, 1])
obj = cvx.Minimize(cvx.norm(A*theta2-y, 2))
cvx.Problem(obj,[]).solve()

print('theta:\n', theta2.value)
```

```
theta:
[[0.65306531]
[0.67129519]]
```

$$\min_{\theta} \|\hat{y} - y\|_2 = \min_{\theta} \|A\theta - y\|_2$$

- By the way, do we have to use only L_2 norm? No.
 - Let's use L_1 norm

```
theta1 = cvx.Variable([2, 1])
obj = cvx.Minimize(cvx.norm(A*theta1-y, 1))
cvx.Problem(obj).solve()

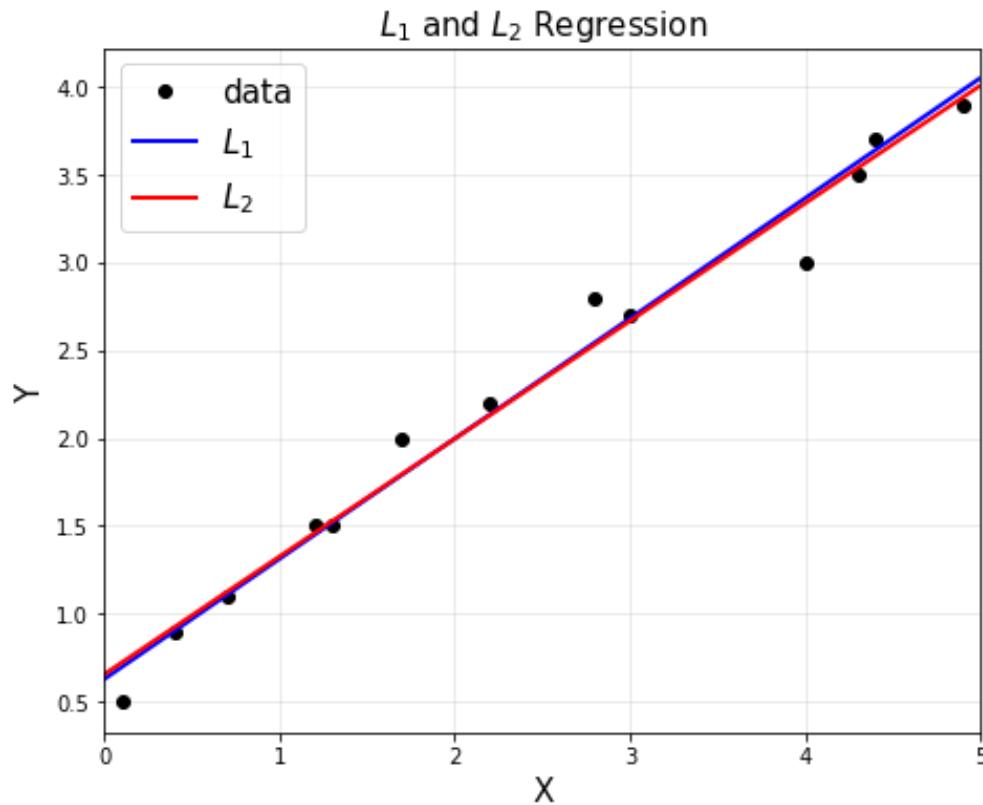
print('theta:\n', theta1.value)
```

```
theta:
[[0.628129]
[0.68520147]]
```

$$\min_{\theta} \|\hat{y} - y\|_1 = \min_{\theta} \|A\theta - y\|_1$$

L_2 Norm vs. L_1 Norm

- L_1 norm also provides a decent linear approximation.



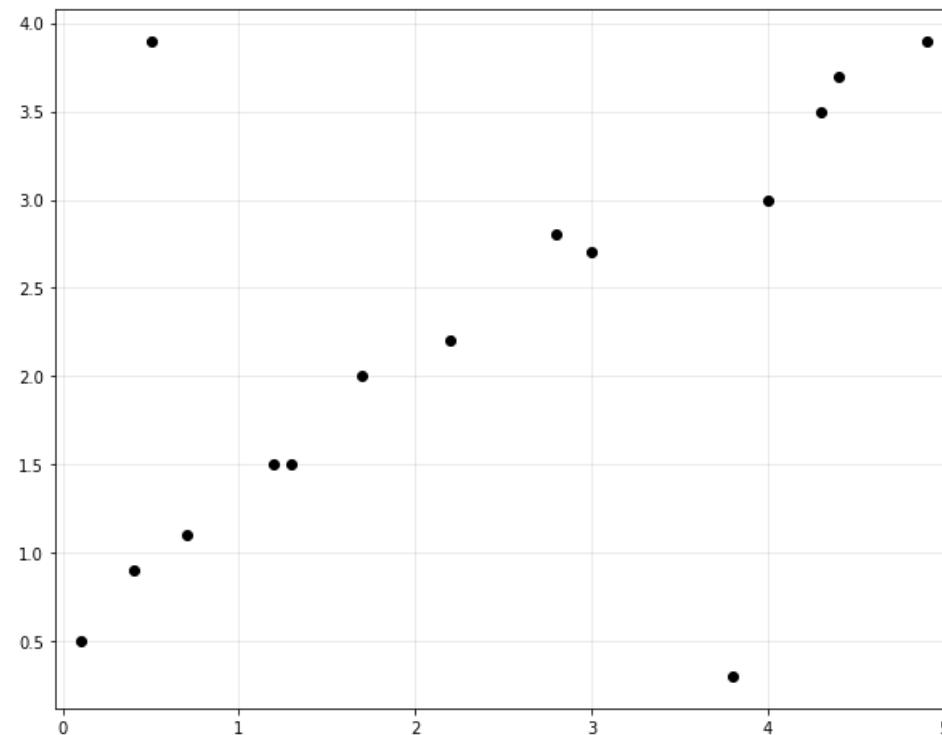
Regression with Outliers

- L_1 norm also provides a decent linear approximation.
- **What if outliers exist?**
 - Fitting with the different norms
 - source:
 - Week 9 of Computational Methods for Data Analysis by Coursera of Univ. of Washington
 - Chapter 17, online book [available](#)

Regression with Outliers

```
# add outliers
x = np.vstack([x, np.array([0.5, 3.8]).reshape(-1, 1)])
y = np.vstack([y, np.array([3.9, 0.3]).reshape(-1, 1)])

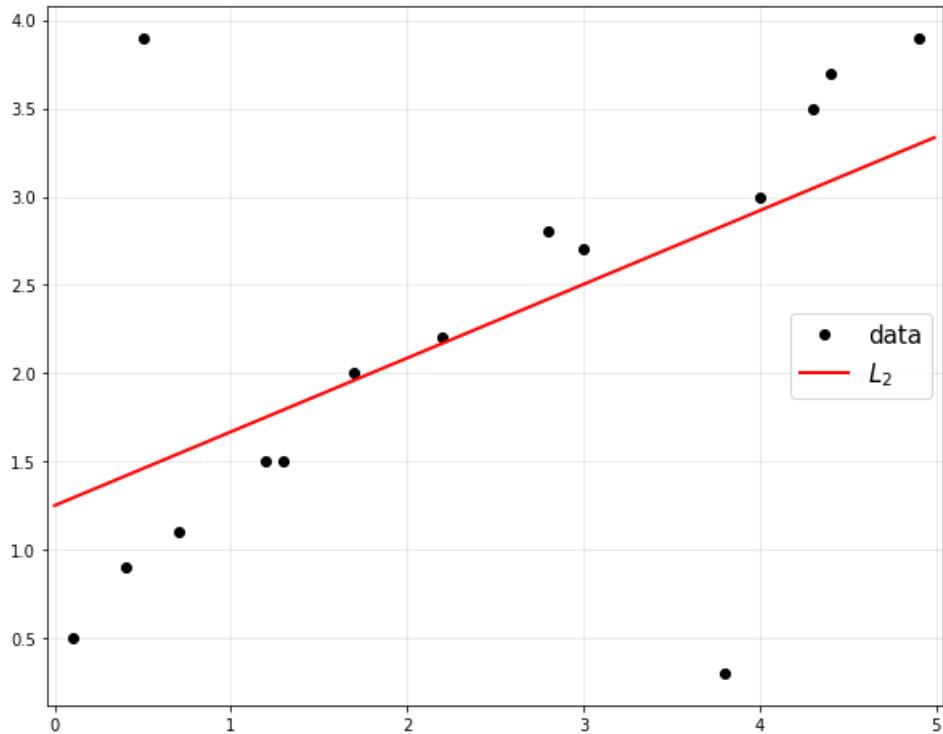
A = np.hstack([x**0, x])
A = np.asmatrix(A)
```



L_2 Norm vs. L_1 Norm

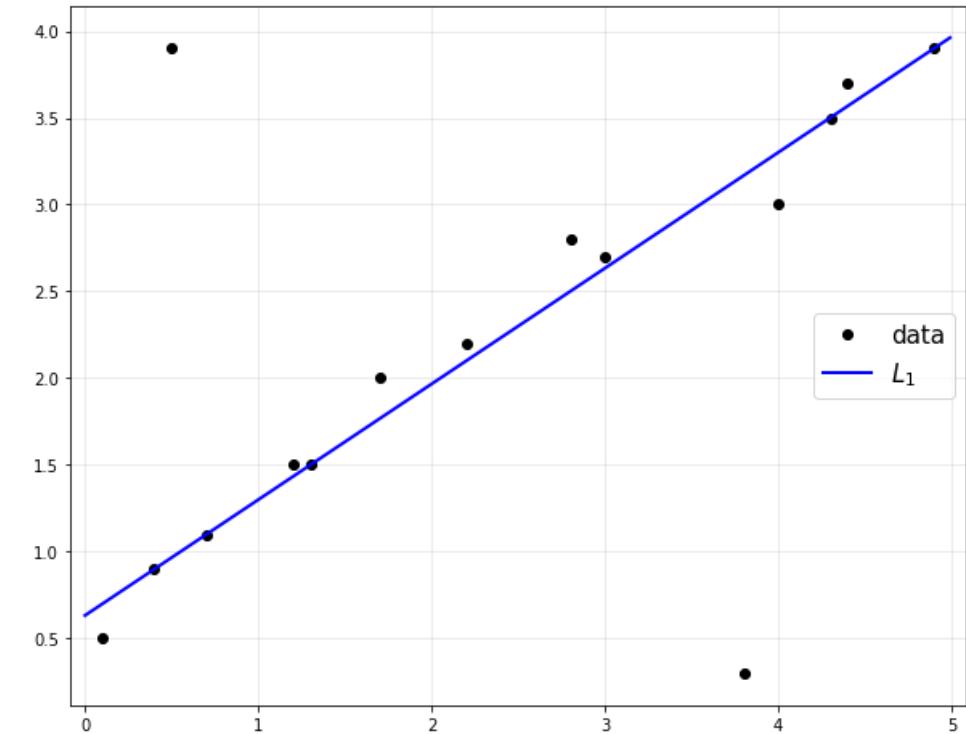
$$\min_{\theta} \|A\theta - y\|_2$$

```
theta2 = cvx.Variable([2, 1])
obj2 = cvx.Minimize(cvx.norm(A*theta2-y, 2))
prob2 = cvx.Problem(obj2).solve()
```



$$\min_{\theta} \|A\theta - y\|_1$$

```
theta1 = cvx.Variable([2, 1])
obj1 = cvx.Minimize(cvx.norm(A*theta1-y, 1))
prob1 = cvx.Problem(obj1).solve()
```

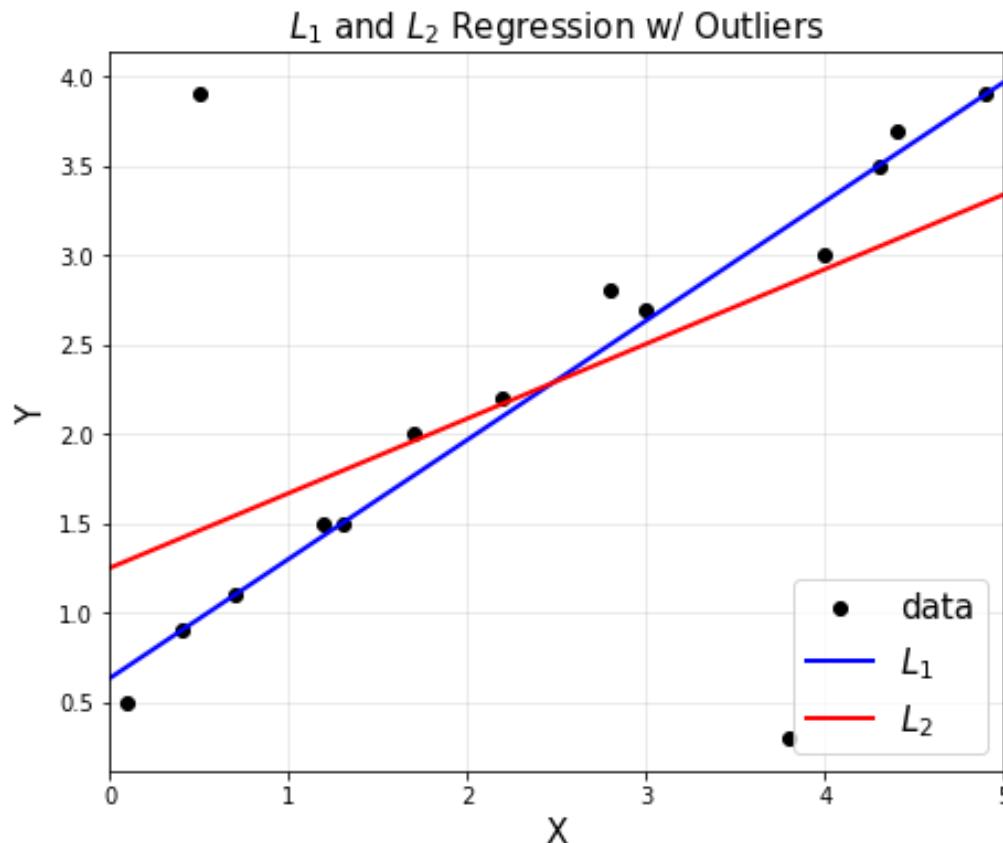


Think About What Makes Different

- It is important to understand what makes them different

$$\min_{\theta} \|A\theta - y\|_1$$

$$\min_{\theta} \|A\theta - y\|_2$$

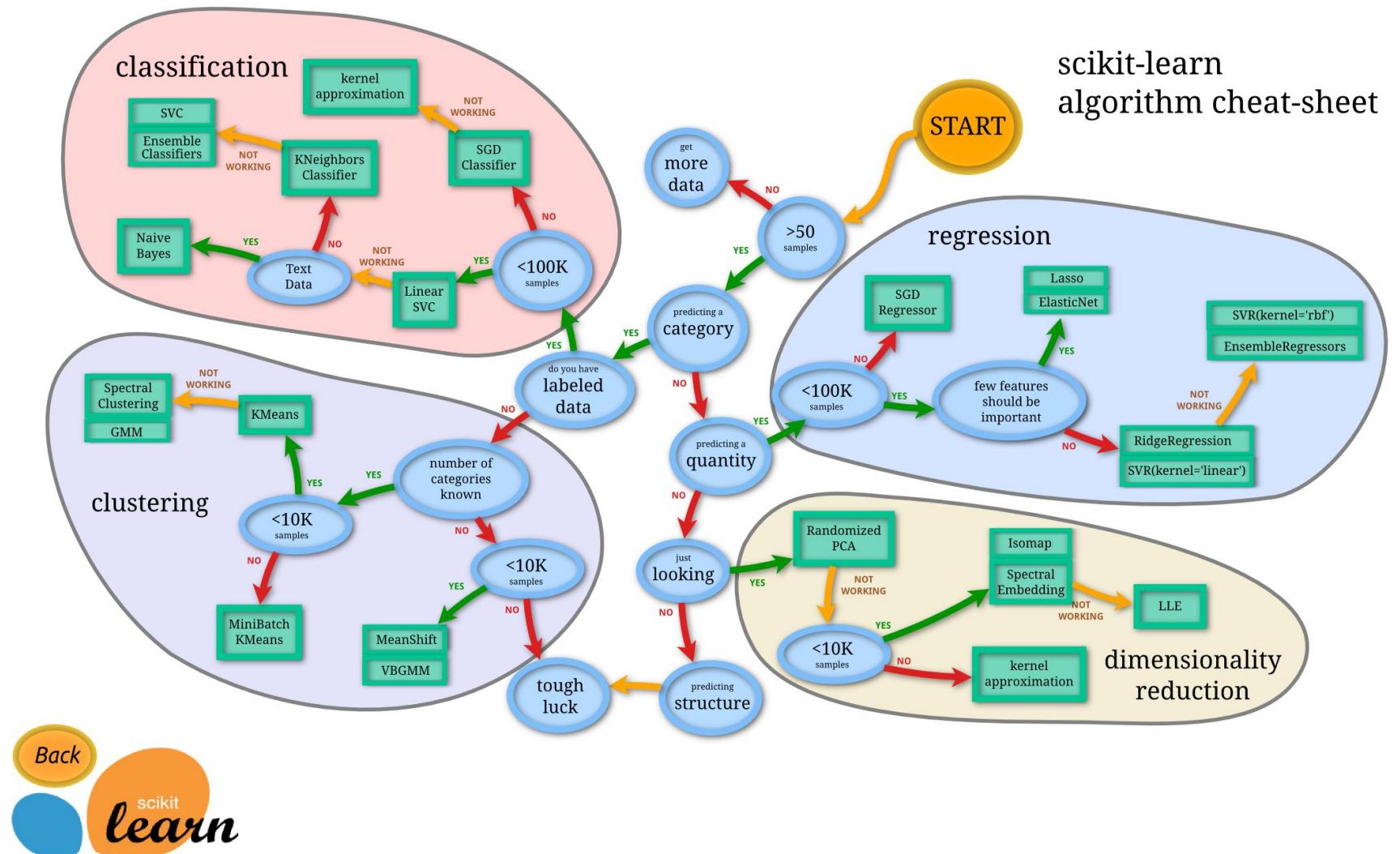


Scikit-Learn

- Machine Learning in Python
- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license
- <https://scikit-learn.org/stable/index.html#>



Scikit-Learn



Scikit-Learn: Regression

```
from sklearn import linear_model
```

```
reg = linear_model.LinearRegression()  
reg.fit(x, y)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,  
normalize=False)
```

```
reg.coef_
```

```
array([[0.67129519]])
```

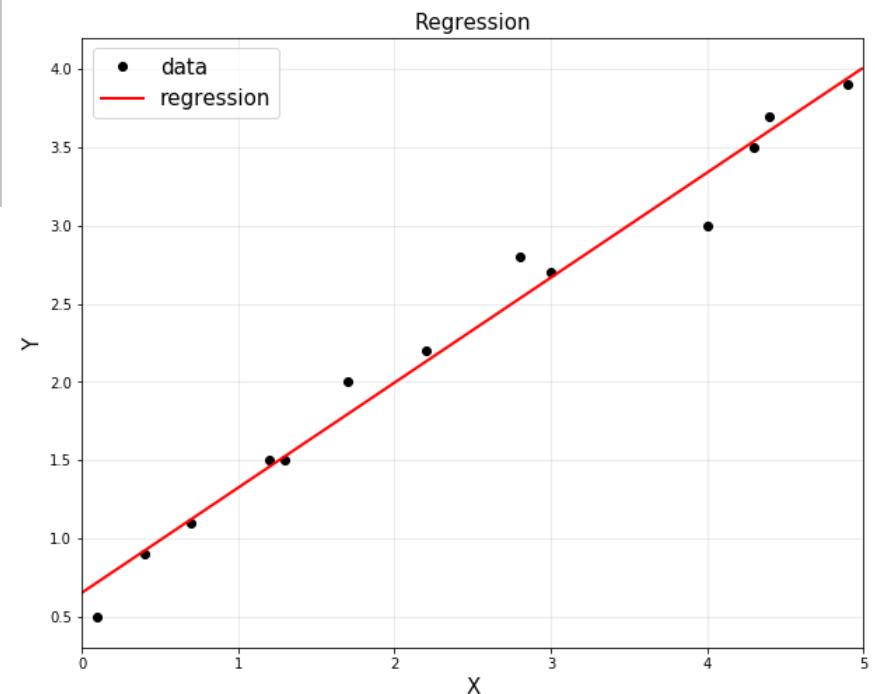
```
reg.intercept_
```

```
array([0.65306531])
```

Scikit-Learn: Regression

```
# to plot
plt.figure(figsize=(10, 8))
plt.title('Regression', fontsize=15)
plt.xlabel('X', fontsize=15)
plt.ylabel('Y', fontsize=15)
plt.plot(x, y, 'ko', label="data")

# to plot a straight line (fitted line)
plt.plot(xp, reg.predict(xp), 'r', linewidth=2, label="regression")
plt.legend(fontsize=15)
plt.axis('equal')
plt.grid(alpha=0.3)
plt.xlim([0, 5])
plt.show()
```



Multivariate Linear Regression

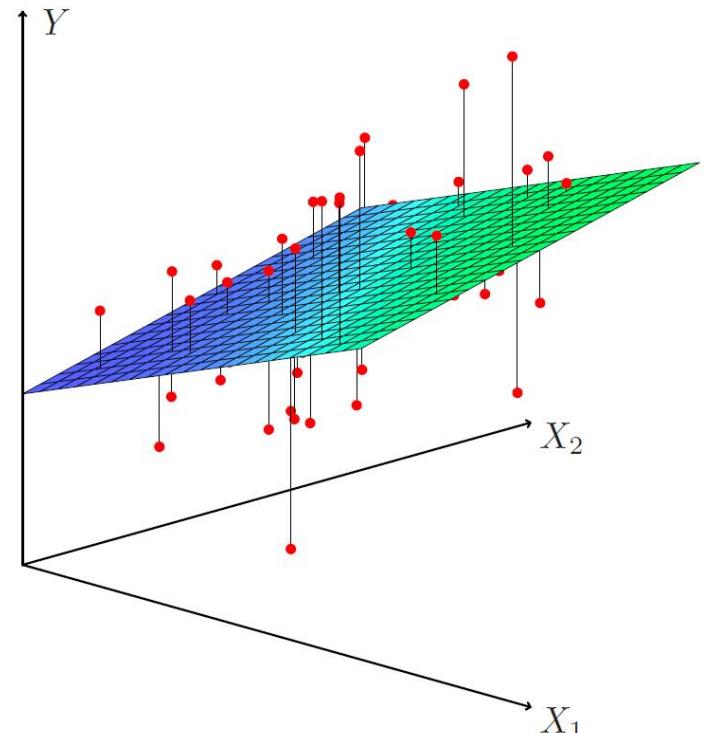
Multivariate Linear Regression

- Linear regression for multivariate data

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

$$\phi(x^{(i)}) = \begin{bmatrix} 1 \\ x_1^{(i)} \\ x_2^{(i)} \end{bmatrix} \implies \theta^* = (\Phi^T \Phi)^{-1} \Phi^T y$$

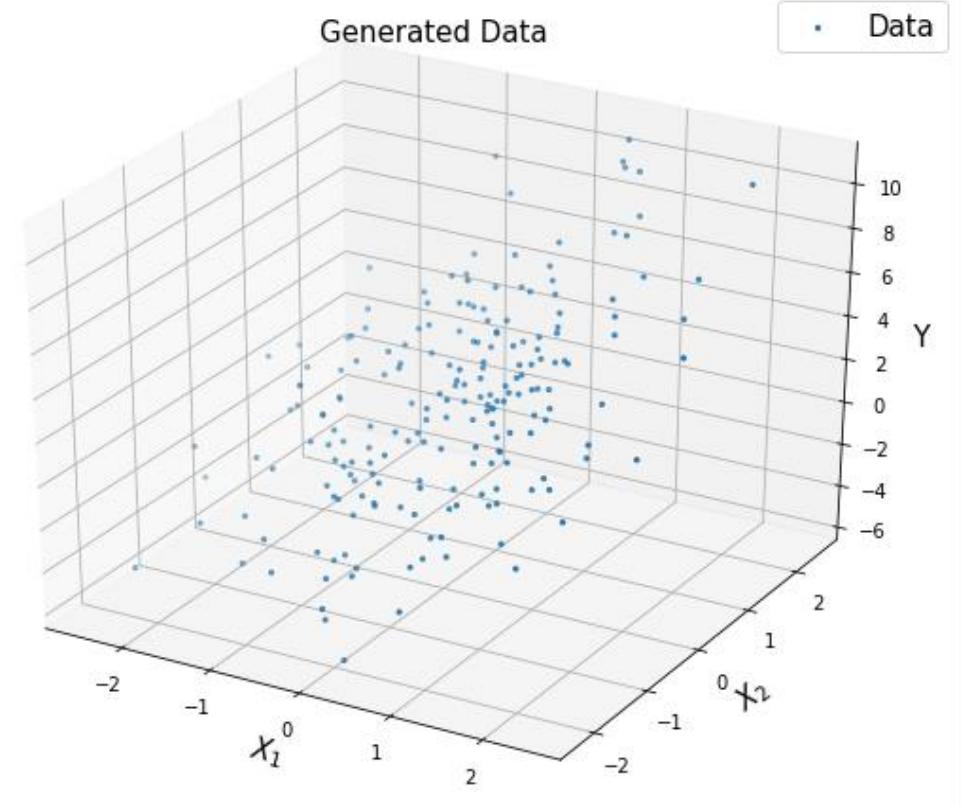
$$\Phi = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} \\ \vdots & & \\ 1 & x_1^{(m)} & x_2^{(m)} \end{bmatrix} \implies \hat{y} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(m)} \end{bmatrix} = \Phi \theta$$



- Same in matrix representation

Multivariate Linear Regression

```
# y = theta0 + theta1*x1 + theta2*x2 + noise  
  
n = 200  
x1 = np.random.randn(n, 1)  
x2 = np.random.randn(n, 1)  
noise = 0.5*np.random.randn(n, 1);  
  
y = 2 + 1*x1 + 3*x2 + noise
```



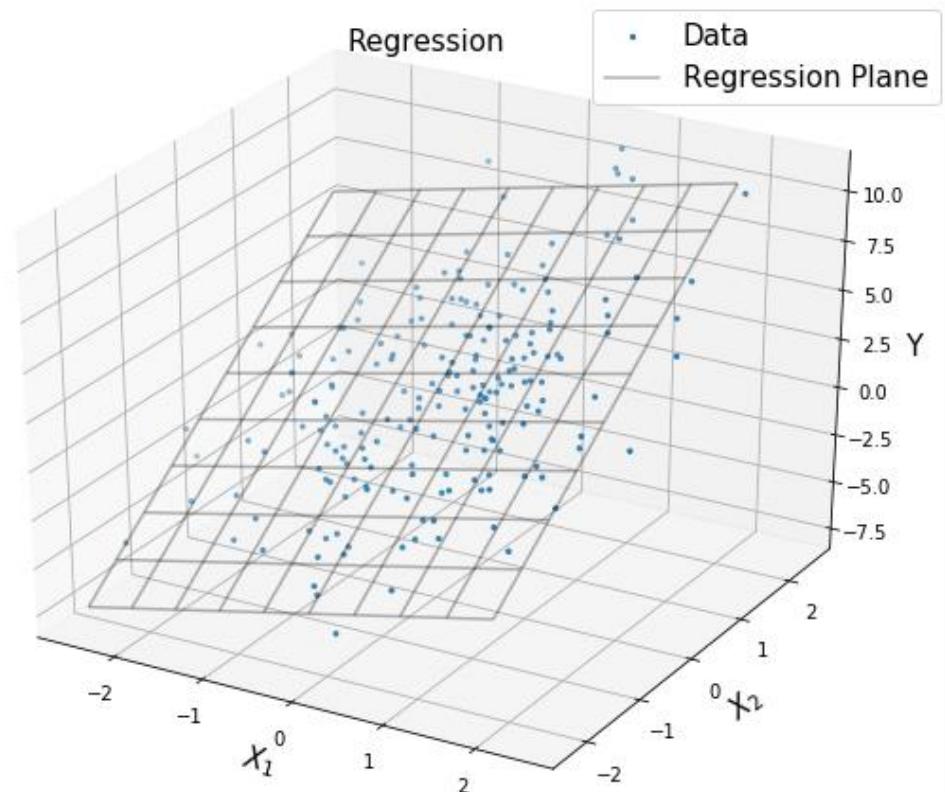
Multivariate Linear Regression

$$\Phi = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} \\ \vdots & & \\ 1 & x_1^{(m)} & x_2^{(m)} \end{bmatrix} \implies \hat{y} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(m)} \end{bmatrix} = \Phi\theta$$

$$\implies \theta^* = (\Phi^T \Phi)^{-1} \Phi^T y$$

```
A = np.hstack([np.ones((n, 1)), x1, x2])
A = np.asmatrix(A)
theta = (A.T*A).I*A.T*y

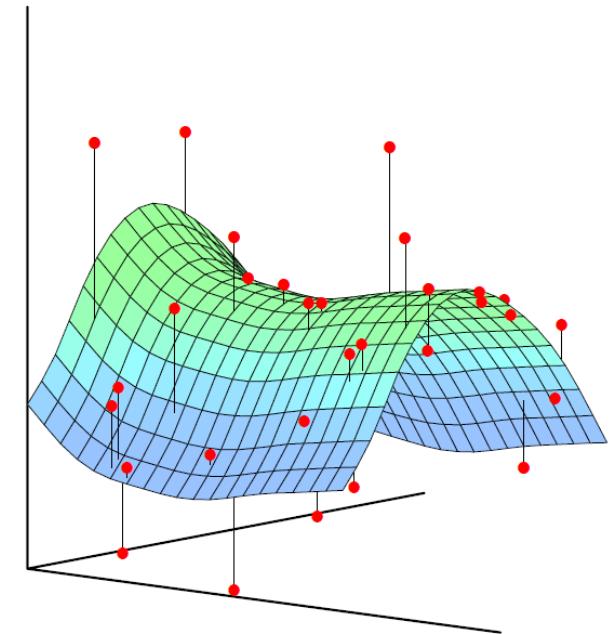
X1, X2 = np.meshgrid(np.arange(np.min(x1), np.max(x1), 0.5),
                      np.arange(np.min(x2), np.max(x2), 0.5))
YP = theta[0,0] + theta[1,0]*X1 + theta[2,0]*X2
```



Nonlinear Regression

Nonlinear Regression

- Linear regression for non-linear data
- Same as linear regression, just with non-linear features
- Method 1: constructing explicit feature vectors
 - polynomial features
 - Radial basis function (**RBF**) features
- Method 2: implicit feature vectors, **kernel trick**



Nonlinear Regression

- Polynomial (here, quad is used as an example)

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \text{noise}$$

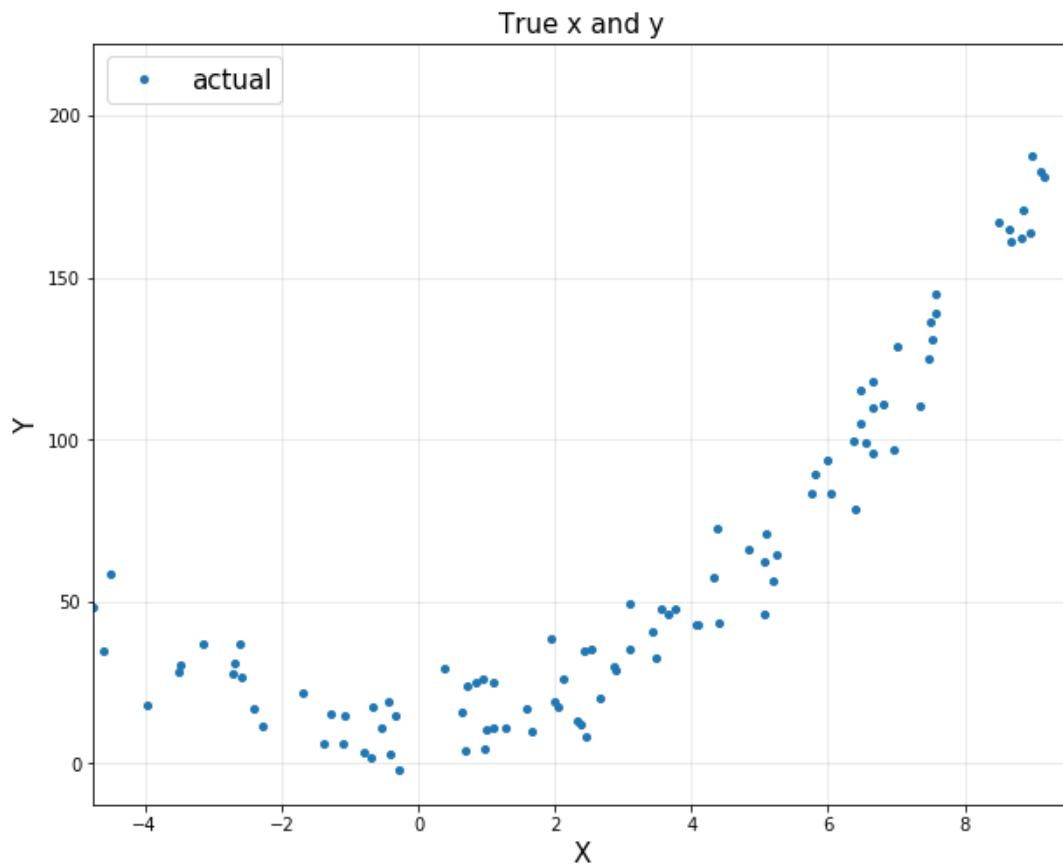
$$\phi(x_i) = \begin{bmatrix} 1 \\ x_i \\ x_i^2 \end{bmatrix}$$

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & & \\ 1 & x_m & x_m^2 \end{bmatrix} \implies \hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_m \end{bmatrix} = \Phi\theta$$

$$\implies \theta^* = (\Phi^T \Phi)^{-1} \Phi^T y$$

Polynomial Regression

```
# y = theta0 + theta1*x + theta2*x^2 + noise  
  
n = 100  
x = -5 + 15*np.random.rand(n, 1)  
noise = 10*np.random.randn(n, 1)  
  
y = 10 + 1*x + 2*x**2 + noise
```

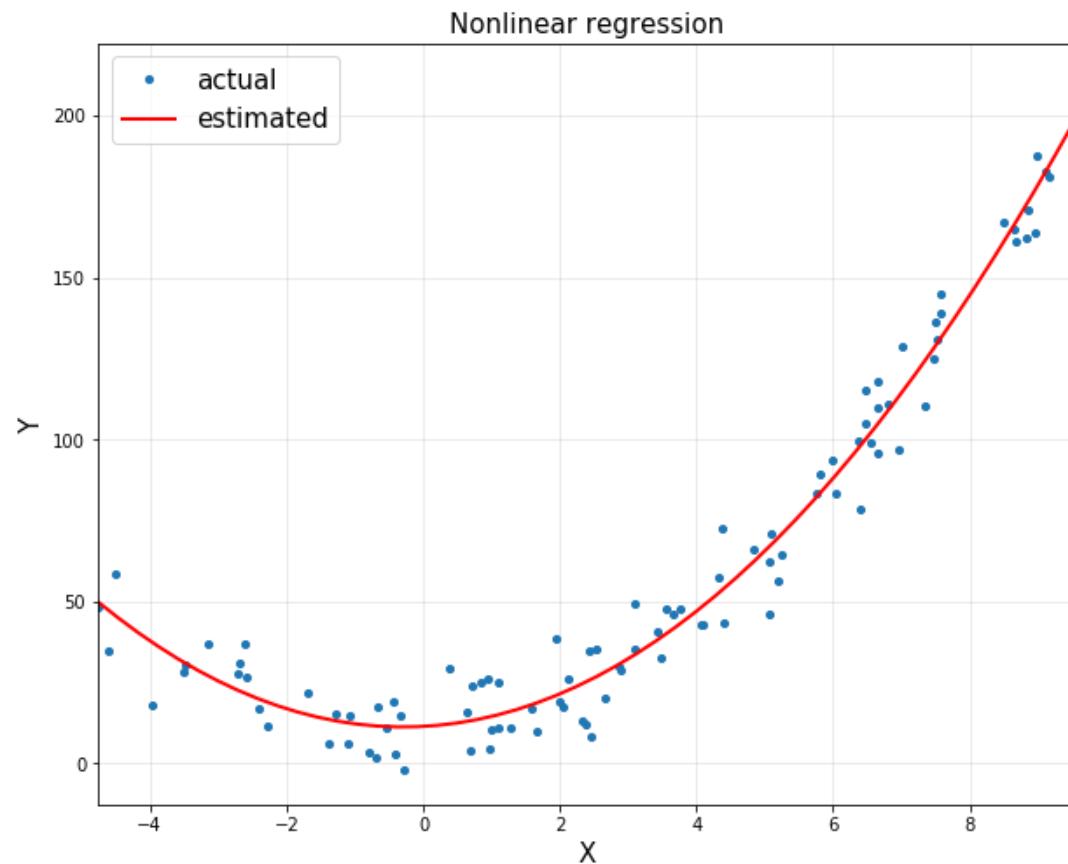


Polynomial Regression

$$\theta = (A^T A)^{-1} A^T y$$

```
A = np.hstack([x**0, x, x**2])
A = np.asmatrix(A)

theta = (A.T*A).I*A.T*y
print('theta:\n', theta)
```



Function Approximation

- Select coefficients among a well-defined function (basis) that closely matches a target function in a task-specific way

Recap: Nonlinear Regression

- Polynomial (here, quad is used as an example)

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \text{noise}$$

$$\phi(x_i) = \begin{bmatrix} 1 \\ x_i \\ x_i^2 \end{bmatrix}$$

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & & \\ 1 & x_m & x_m^2 \end{bmatrix} \implies \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \\ 1 & x_m & x_m^2 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} \Rightarrow \begin{bmatrix} | & | & | \\ b_0(x) & b_1(x) & b_2(x) \\ | & | & | \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}$$

$$\implies \theta^* = (\Phi^T \Phi)^{-1} \Phi^T y$$

Different perspective:

- Approximate a target function as a linear combination of basis

$$\hat{y} = \sum_{i=0}^d \theta_i b_i(x) = \Phi \theta$$

Construct Explicit Feature Vectors

- Consider linear combinations of fixed nonlinear functions
 - Polynomial
 - Radial Basis Function (RBF)

$$\hat{y} = \sum_{i=0}^d \theta_i b_i(x) = \Phi\theta$$

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & & \\ 1 & x_m & x_m^2 \end{bmatrix} \implies \hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_m \end{bmatrix} = \Phi\theta$$

Polynomial Basis

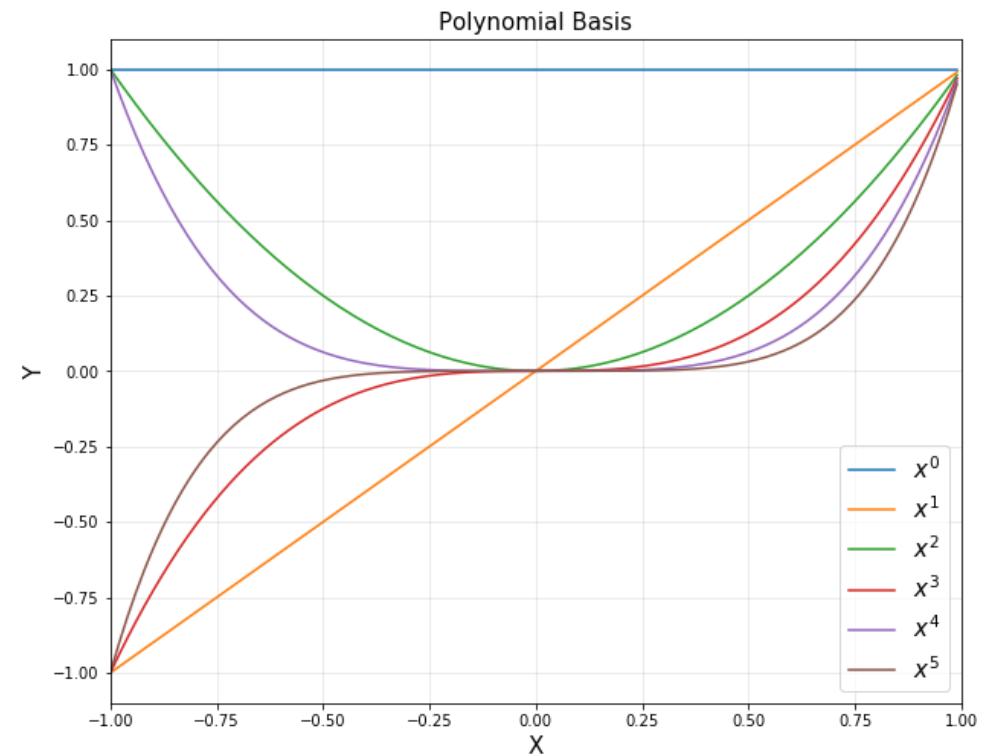
1) Polynomial functions

$$b_i(x) = x^i, \quad i = 0, \dots, d$$

```
xp = np.arange(-1, 1, 0.01).reshape(-1, 1)
polybasis = np.hstack([xp**i for i in range(6)])

plt.figure(figsize=(10, 8))

for i in range(6):
    plt.plot(xp, polybasis[:,i], label = '$x^{}$'.format(i))
```



RBF Basis

2) Radial Basis Functions (RBF) with bandwidth σ and k RBF centers $\mu_i \in \mathbb{R}^n$, $i = 1, 2, \dots, k$

$$b_i(x) = \exp\left(-\frac{\|x - \mu_i\|^2}{2\sigma^2}\right)$$

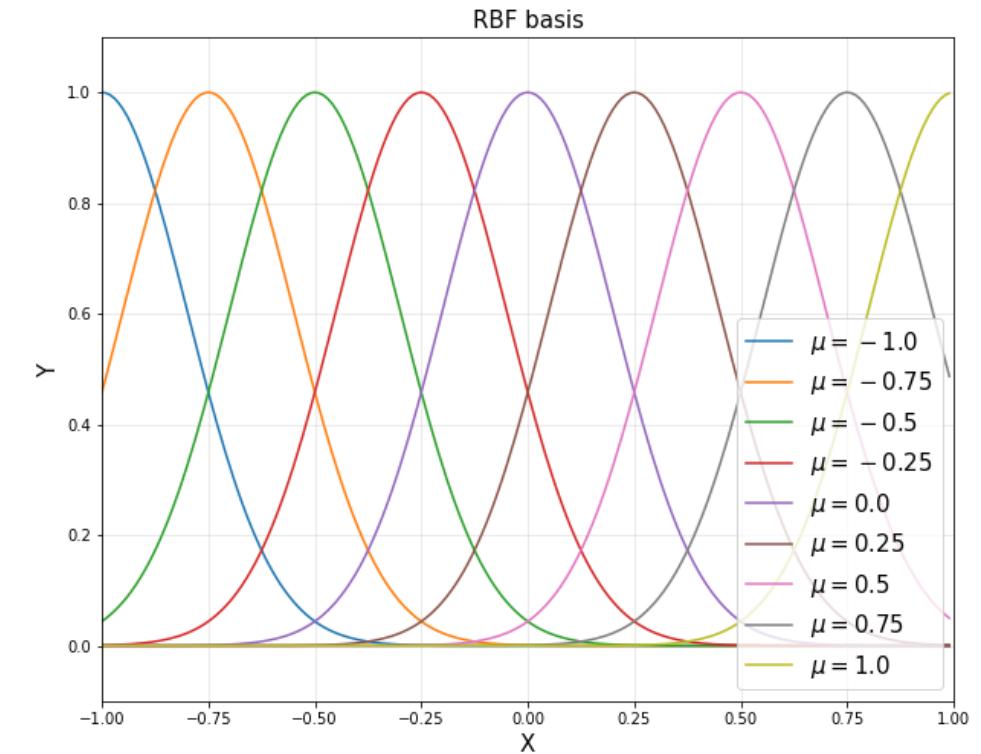
```
d = 9

u = np.linspace(-1, 1, d)
sigma = 0.2

rbfbasis = np.hstack([np.exp(-(xp-u[i])**2/(2*sigma**2)) for i in range(d)])

plt.figure(figsize=(10, 8))

for i in range(d):
    plt.plot(xp, rbfbasis[:,i], label='\mu = {}'.format(u[i]))
```



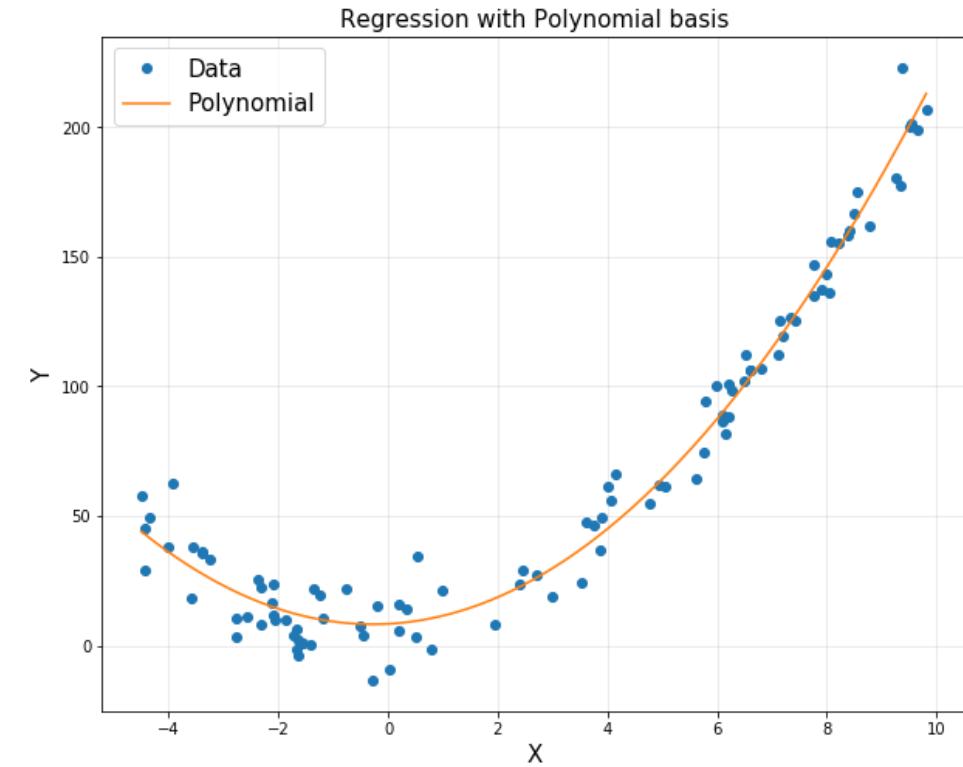
Nonlinear Regression with Polynomial Functions

```
xp = np.arange(np.min(x), np.max(x), 0.01).reshape(-1, 1)

d = 3
polybasis = np.hstack([xp**i for i in range(d)])
polybasis = np.asmatrix(polybasis)

A = np.hstack([x**i for i in range(d)])
A = np.asmatrix(A)

theta = (A.T*A).I*A.T*y
yp = polybasis*theta
```



Nonlinear Regression with RBF Functions

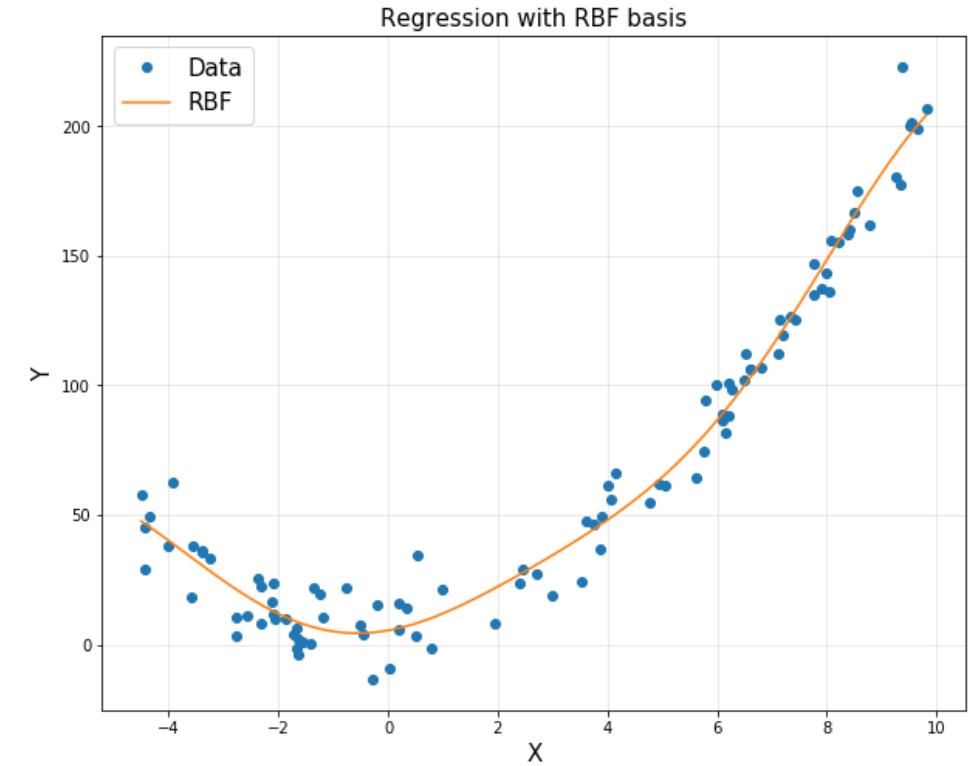
```
xp = np.arange(np.min(x), np.max(x), 0.01).reshape(-1, 1)

d = 6
u = np.linspace(np.min(x), np.max(x), d)
sigma = 4

rbfbasis = np.hstack([np.exp(-(xp-u[i])**2/(2*sigma**2)) for i in range(d)])
A = np.hstack([np.exp(-(x-u[i])**2/(2*sigma**2)) for i in range(d)])

rbfbasis = np.asmatrix(rbfbasis)
A = np.asmatrix(A)

theta = (A.T*A).I*A.T*y
yp = rbfbasis*theta
```



Summary: Linear Regression

- Though linear regression may seem limited, it is very powerful, since the input features can themselves include non-linear features of data
- Linear regression on non-linear features of data
- For least-squares loss, optimal parameters still are

$$\theta^* = (\Phi^T \Phi)^{-1} \Phi^T y$$



Regression 2

**Prof. Seungchul Lee
Industrial AI Lab.**

Linear Regression: Advanced

- Overfitting
- Regularization (Ridge and Lasso)

Overfitting: Start with Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

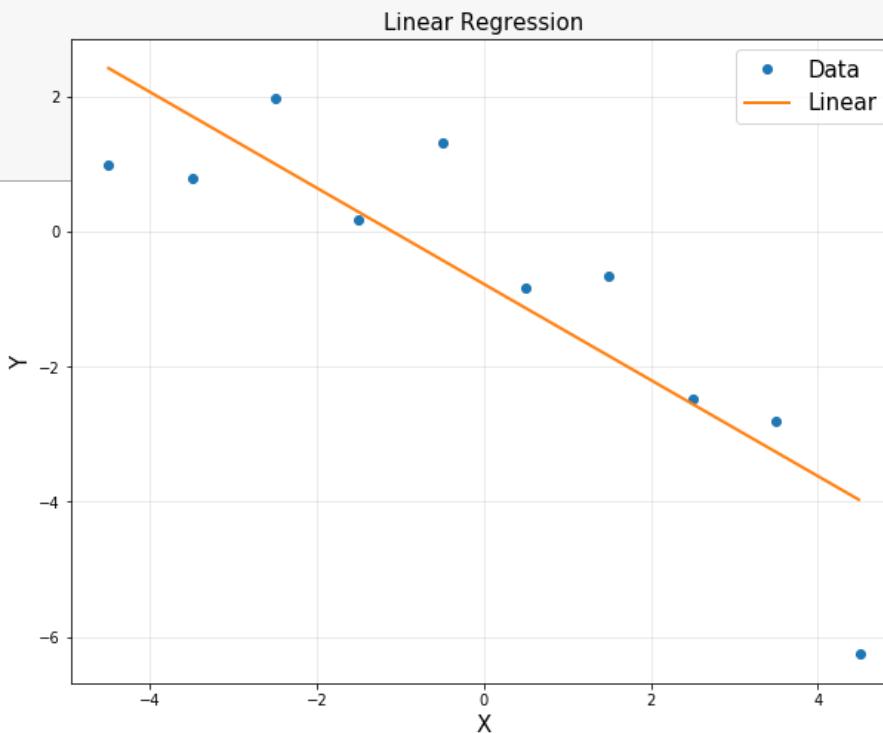
# 10 data points
n = 10
x = np.linspace(-4.5, 4.5, 10).reshape(-1, 1)
y = np.array([0.9819, 0.7973, 1.9737, 0.1838, 1.3180, -0.8361, -0.6591, -2.4701, -2.8122, -6.2512]).reshape(-1, 1)

plt.figure(figsize=(10, 8))
plt.plot(x, y, 'o', label = 'Data')
plt.xlabel('X', fontsize = 15)
plt.ylabel('Y', fontsize = 15)
plt.grid(alpha = 0.3)
plt.show()
```

```
A = np.hstack([x**0, x])
A = np.asmatrix(A)

theta = (A.T*A).I*A.T*y
print(theta)
```

```
[[ -0.7774      ]
 [ -0.71070424]]
```



Recap: Nonlinear Regression

- Polynomial (here, quad is used as an example)

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \text{noise}$$

$$\phi(x_i) = \begin{bmatrix} 1 \\ x_i \\ x_i^2 \end{bmatrix}$$

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & & \\ 1 & x_m & x_m^2 \end{bmatrix} \implies \hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_m \end{bmatrix} = \Phi\theta$$

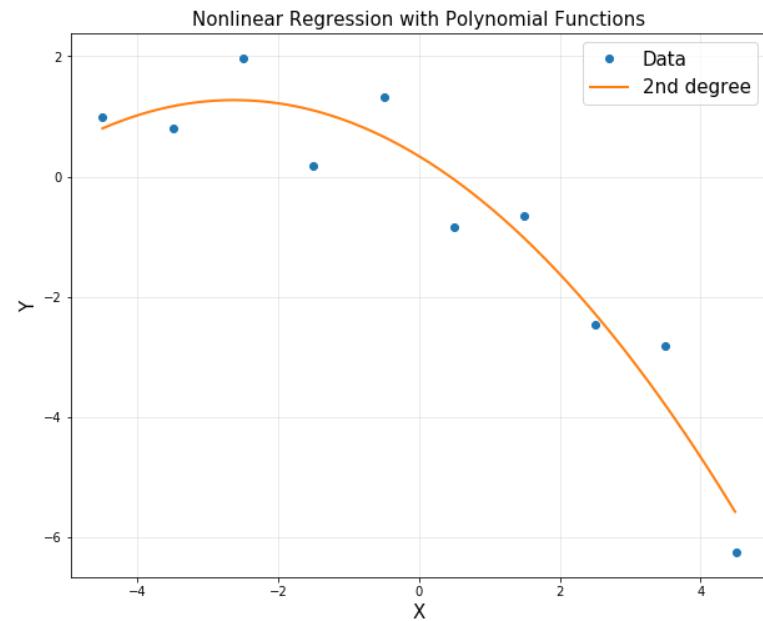
$$\implies \theta^* = (\Phi^T \Phi)^{-1} \Phi^T y$$

Nonlinear Regression

```
A = np.hstack([x**0, x, x**2])
A = np.asmatrix(A)

theta = (A.T*A).I*A.T*y
print(theta)
```

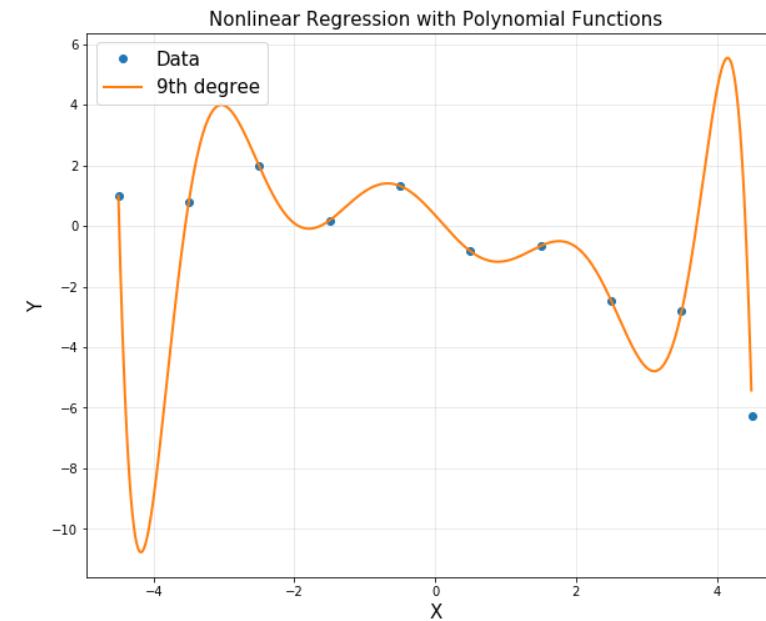
```
[[ 0.33669062]
 [-0.71070424]
 [-0.13504129]]
```



```
A = np.hstack([x**i for i in range(10)])
A = np.asmatrix(A)

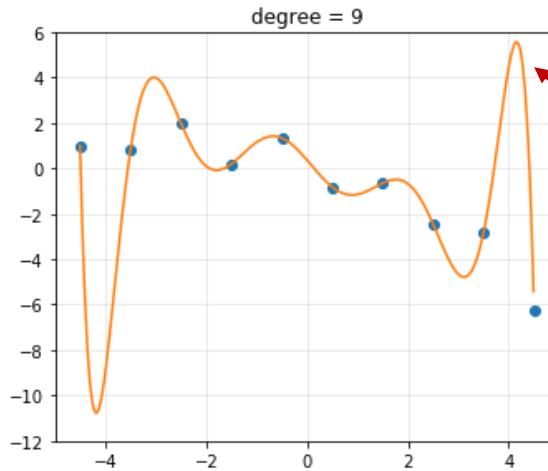
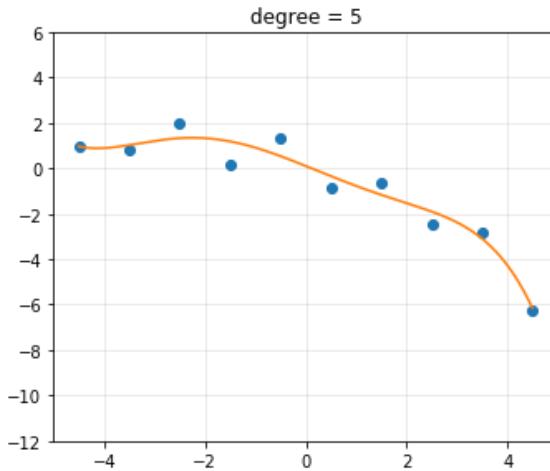
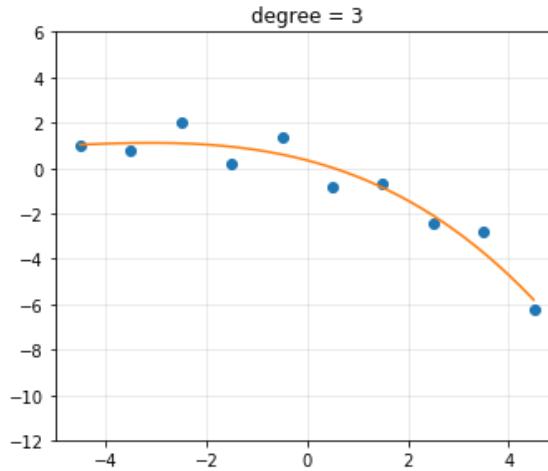
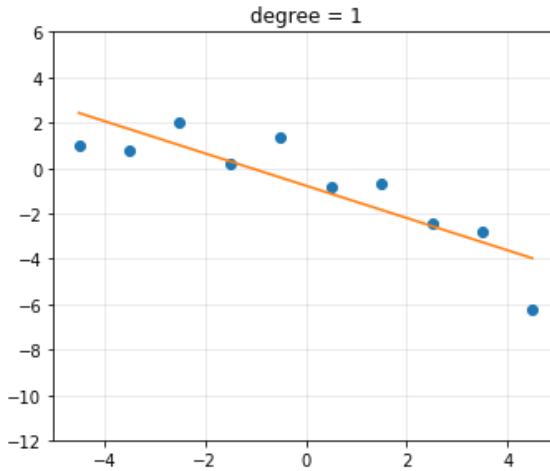
theta = (A.T*A).I*A.T*y
```

10 input points with degree 9 (or 10)



Polynomial Fitting with Different Degrees

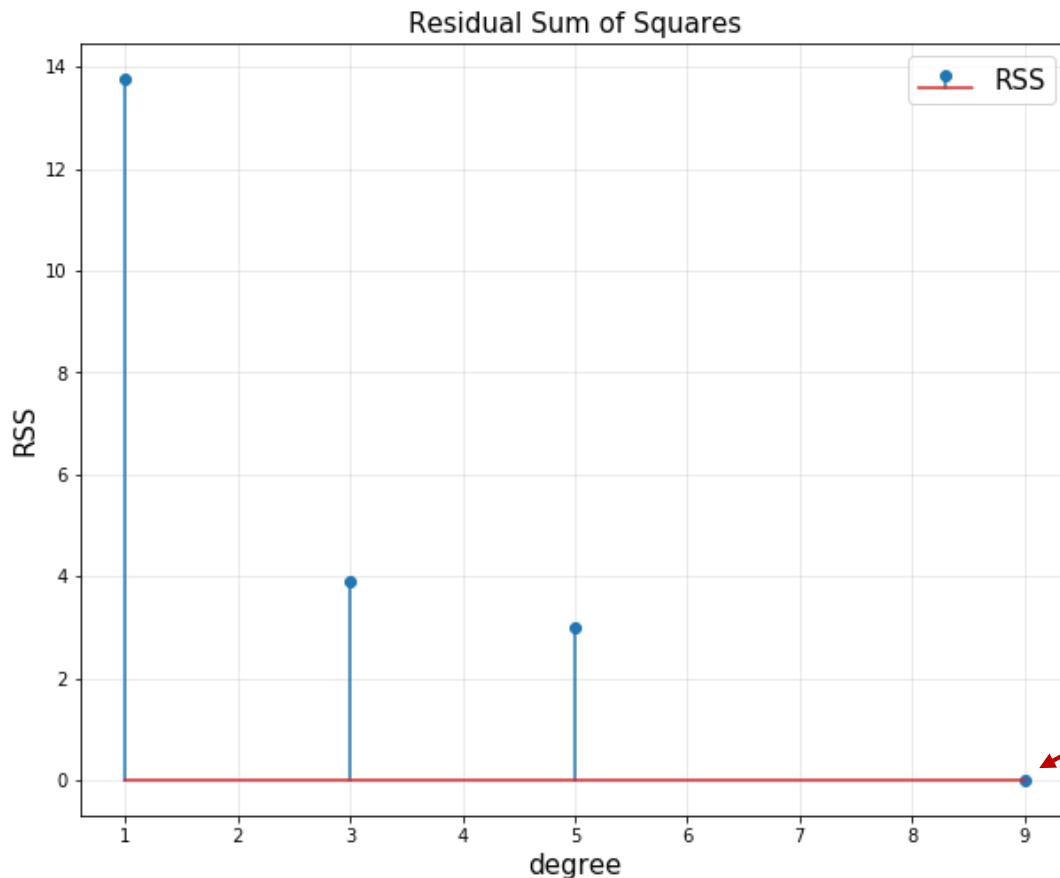
Regression



Low error on input data points,
but high error nearby

Loss

- Loss: Residual Sum of Squares (RSS)



$$\min_{\theta} \|\hat{y} - y\|_2^2$$

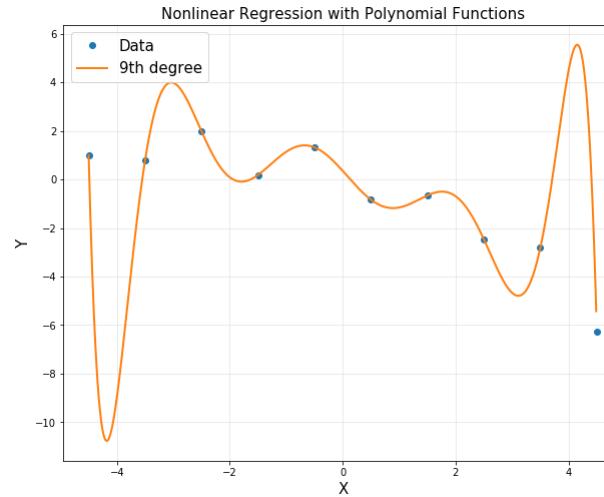
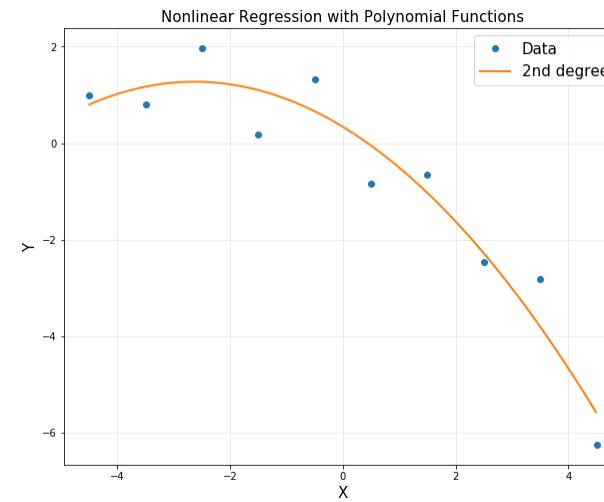
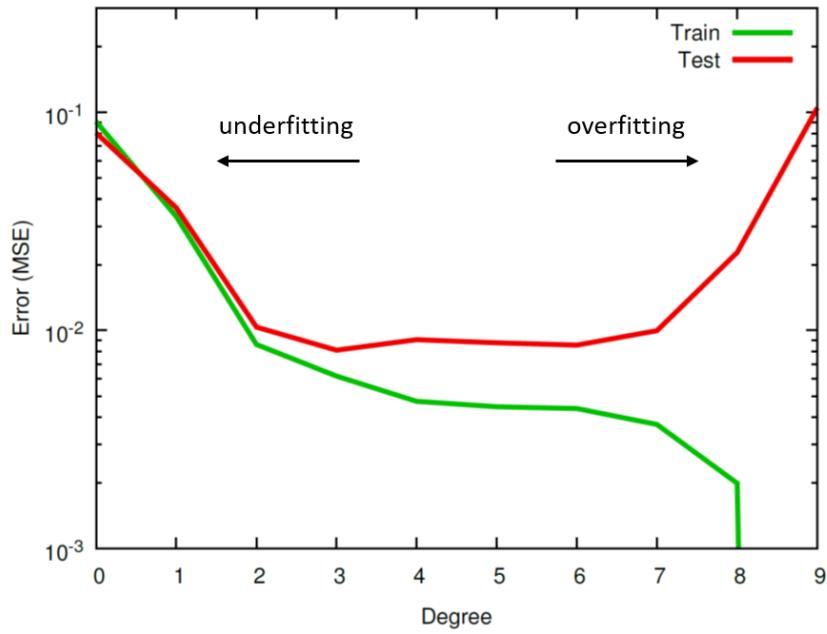
Minimizing loss in training data is often not the best



Low error on input data points, but high error nearby

Issue with Rich Representation

- Low error on input data points, but high error nearby
- Low error on training data, but high error on testing data



Linear Regression with RBF

```
xp = np.arange(-4.5, 4.5, 0.01).reshape(-1, 1)

d = 10
u = np.linspace(-4.5, 4.5, d)
sigma = 0.2

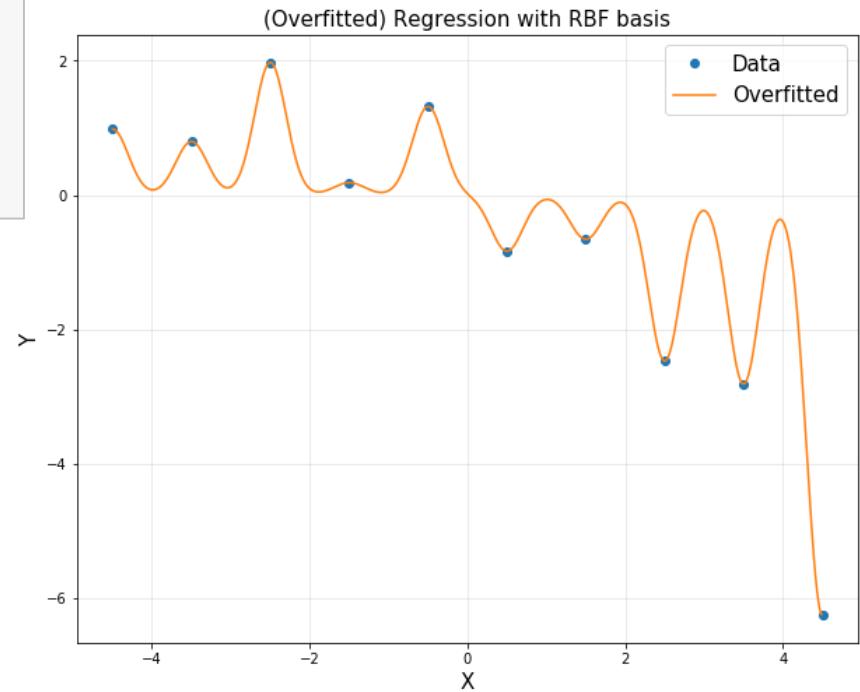
A = np.hstack([np.exp(-(x-u[i])**2/(2*sigma**2)) for i in range(d)])
rbfbasis = np.hstack([np.exp(-(xp-u[i])**2/(2*sigma**2)) for i in range(d)])

A = np.asmatrix(A)
rbfbasis = np.asmatrix(rbfbasis)

theta = (A.T*A).I*A.T*y
yp = rbfbasis*theta
```

$$\theta = (A^T A)^{-1} A^T y$$

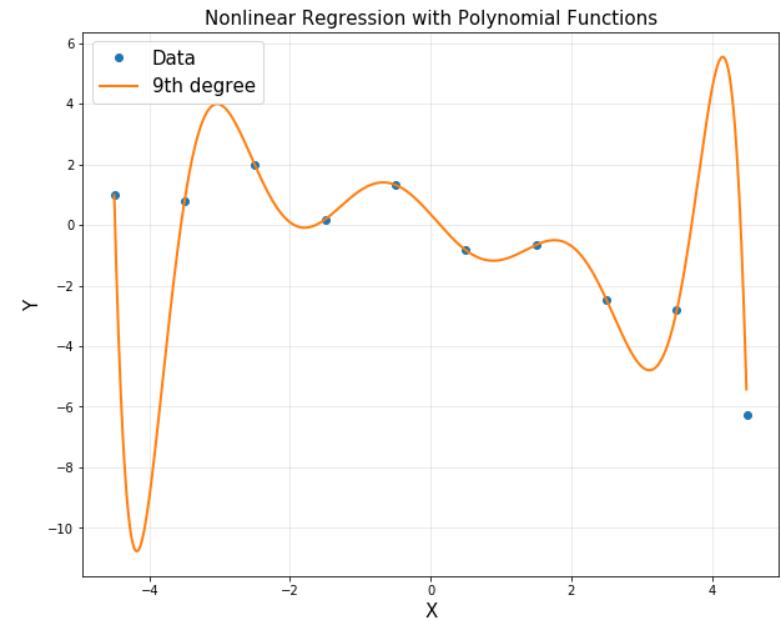
- With many features, our prediction function becomes very expensive
- Can lead to overfitting



Regularization

Issue with Rich Representation

- Low error on input data points, but high error nearby
- Low error on training data, but high error on testing data

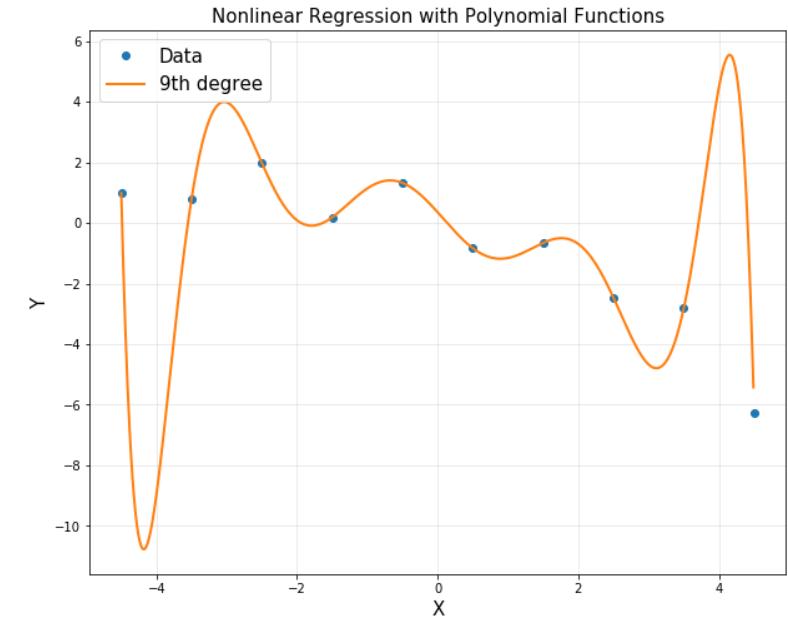


Generalization Error

- Fundamental problem: we are optimizing parameters to solve

$$\min_{\theta} \sum_{i=1}^m \ell(y_i, \hat{y}_i) = \min_{\theta} \sum_{i=1}^m \ell(y_i, \Phi\theta)$$

- But what we really care about is loss of prediction on new data (x, y)
 - also called generalization error
- Divide data into training set, and validation (testing) set



Representational Difficulties

- With many features, prediction function becomes very expressive (model complexity)
 - Choose less expressive function (e.g., lower degree polynomial, fewer RBF centers, larger RBF bandwidth)
 - Keep the magnitude of the parameter small
 - Regularization: penalize large parameters θ

$$\min \|\Phi\theta - y\|_2^2 + \lambda\|\theta\|_2^2$$

- λ : regularization parameter, trades off between low loss and small values of θ

With Less Basis Functions: Fewer RBF Centers

```
d = [2, 4, 6, 10]
sigma = 1

plt.figure(figsize=(12, 10))

for k in range(4):
    u = np.linspace(-4.5, 4.5, d[k])

    A = np.hstack([np.exp(-(x-u[i])**2/(2*sigma**2)) for i in range(d[k])])
    rbfbasis = np.hstack([np.exp(-(xp-u[i])**2/(2*sigma**2)) for i in range(d[k])])

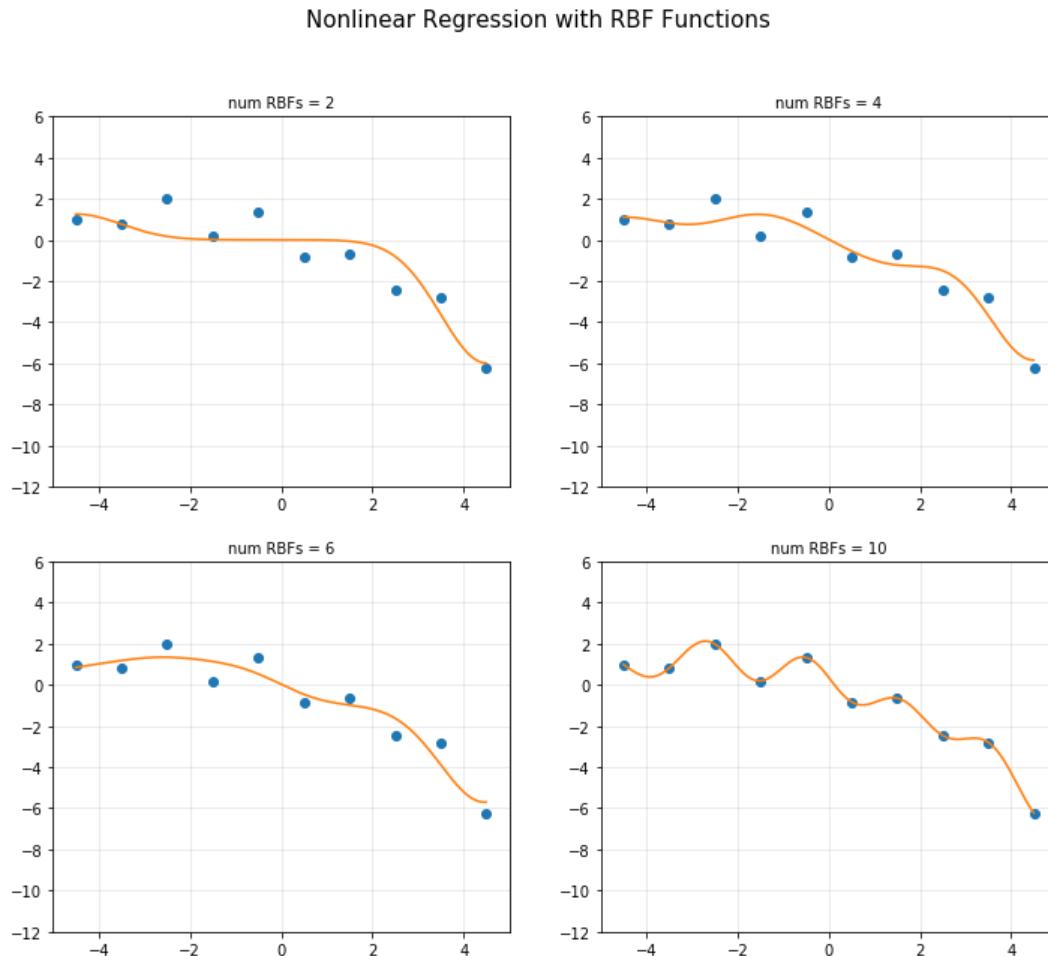
    A = np.asmatrix(A)
    rbfbasis = np.asmatrix(rbfbasis)

    theta = (A.T*A).I*A.T*y
    yp = rbfbasis*theta

    plt.subplot(2, 2, k+1)
    plt.plot(x, y, 'o')
    plt.plot(xp, yp)
    plt.axis([-5, 5, -12, 6])
    plt.title('num RBFs = {}'.format(d[k]), fontsize = 10)
    plt.grid(alpha = 0.3)
```

With Less Basis Functions: Fewer RBF Centers

- Least-squares fits for different numbers of RBFs



Representational Difficulties

- With many features, prediction function becomes very expressive (model complexity)
 - Choose less expensive function (e.g., lower degree polynomial, fewer RBF centers, larger RBF bandwidth)
 - Keep the magnitude of the parameter small
 - Regularization: penalize large parameters θ

$$\min \|\Phi\theta - y\|_2^2 + \lambda\|\theta\|_2^2$$

- λ : regularization parameter, trades off between low loss and small values of θ

Regularization (Shrinkage Methods)

- Often, overfitting associated with very large estimated parameters
- We want to balance
 - how well function fits data
 - magnitude of coefficients

$$\text{Total cost} = \underbrace{\text{measure of fit}}_{RSS(\theta)} + \lambda \cdot \underbrace{\text{measure of magnitude of coefficients}}_{\lambda \cdot \|\theta\|_2^2}$$

$$\implies \min \|\Phi\theta - y\|_2^2 + \lambda \|\theta\|_2^2$$

- multi-objective optimization
- λ is a tuning parameter

Regularization (Shrinkage Methods)

- the second term, $\lambda \cdot \|\theta\|_2^2$, called a shrinkage penalty, is small when $\theta_1, \dots, \theta_d$ are close to zeros, and so it has the effect of shrinking the estimates of θ_j towards zero
- the tuning parameter λ serves to control the relative impact of these two terms on the regression coefficient estimates
- known as a *ridge regression*

RBF: Start from Rich Representation

```
d = 10
u = np.linspace(-4.5, 4.5, d)

sigma = 1

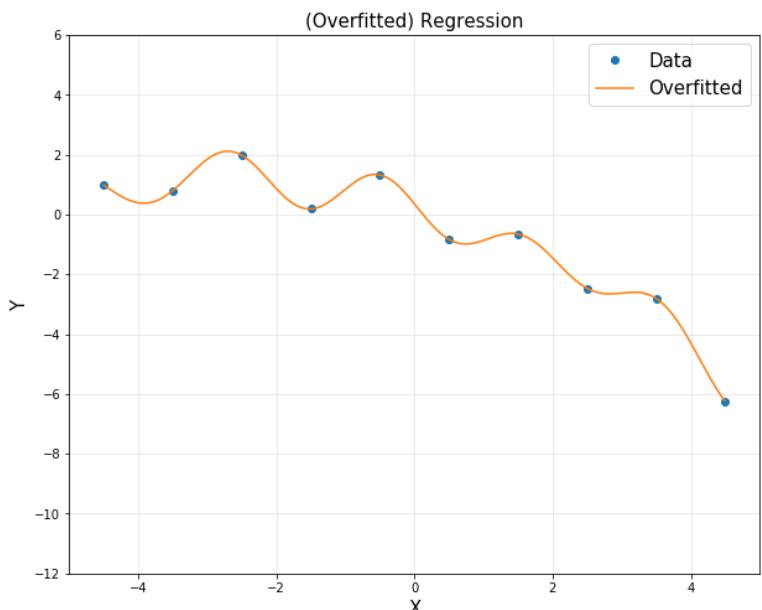
A = np.hstack([np.exp(-(x-u[i])**2/(2*sigma**2)) for i in range(d)])
rbfbasis = np.hstack([np.exp(-(xp-u[i])**2/(2*sigma**2)) for i in range(d)])

A = np.asmatrix(A)
rbfbasis = np.asmatrix(rbfbasis)

theta = cvx.Variable([d, 1])
obj = cvx.Minimize(cvx.sum_squares(A*theta-y))
prob = cvx.Problem(obj).solve()

yp = rbfbasis*theta.value
```

$$\min \|\Phi\theta - y\|_2^2$$



RBF with Regularization

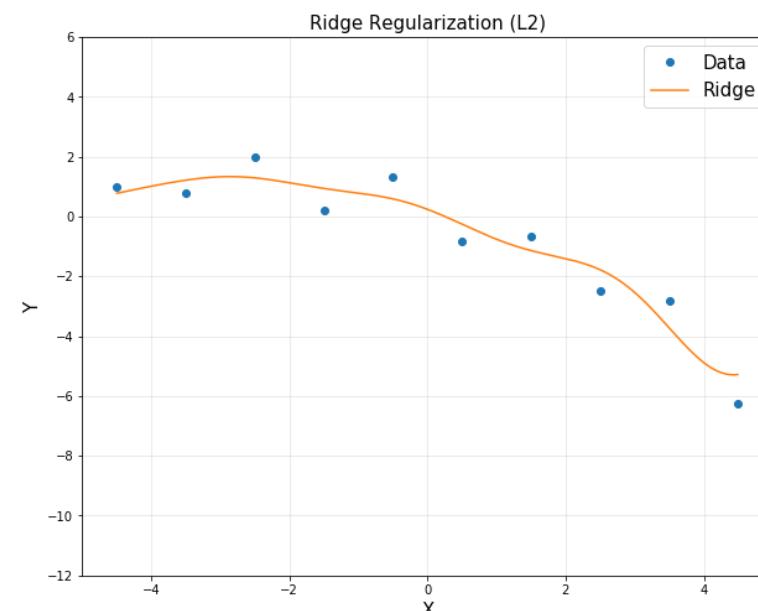
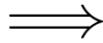
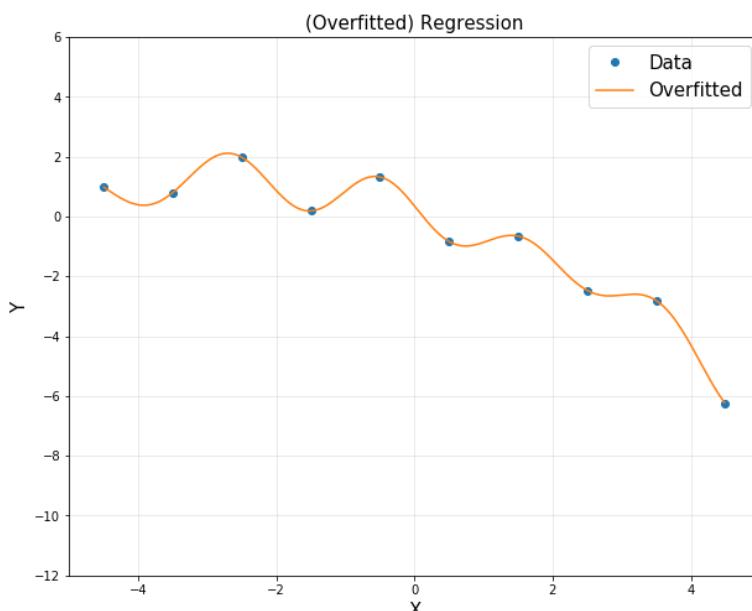
- Start from rich representation. Then, regularize coefficients θ

```
# ridge regression

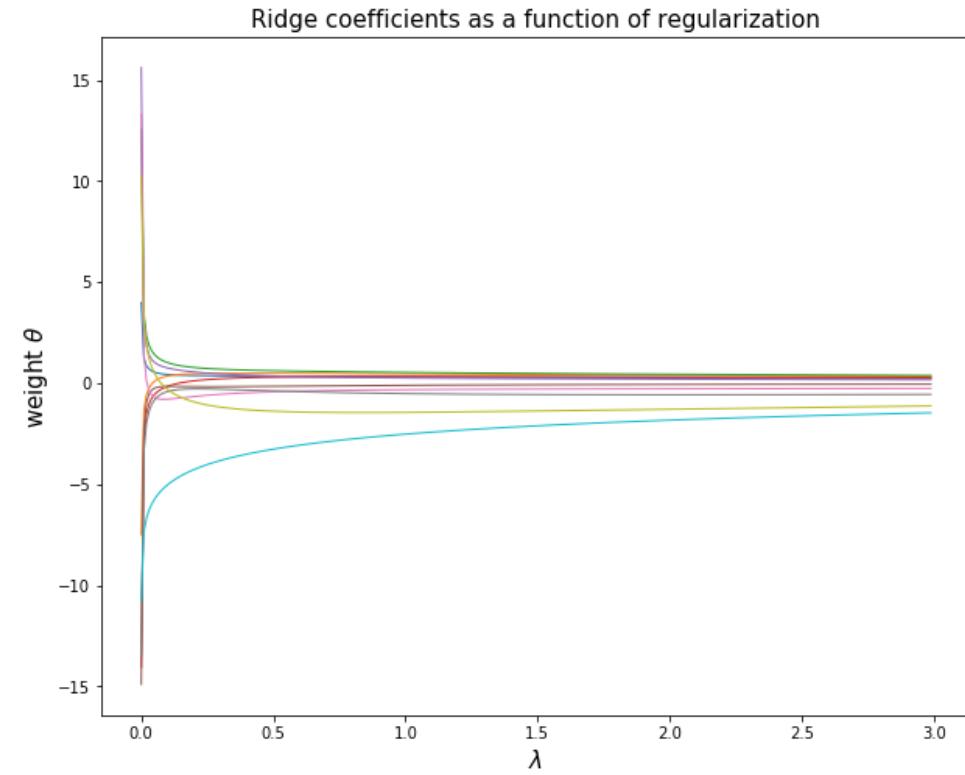
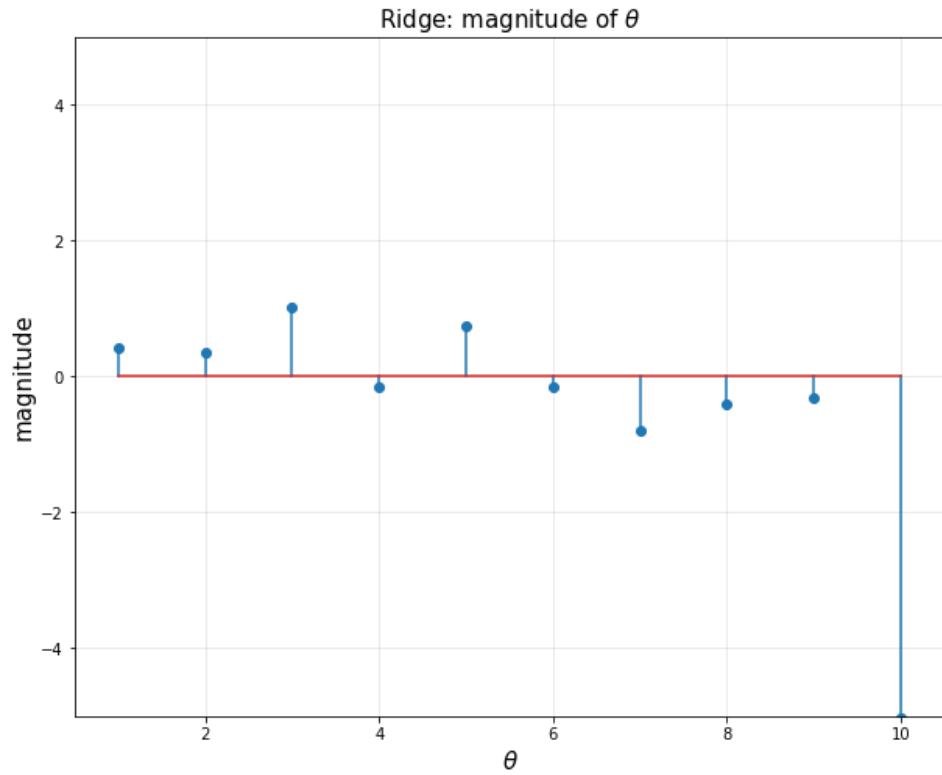
lamb = 0.1
theta = cvx.Variable([d, 1])
obj = cvx.Minimize(cvx.sum_squares(A*theta - y) + lamb*cvx.sum_squares(theta))
prob = cvx.Problem(obj).solve()

yp = rbfbasis*theta.value
```

$$\min \|\Phi\theta - y\|_2^2 + \lambda\|\theta\|_2^2$$



Coefficients θ



Let's Use L_1 Norm

- Ridge regression

$$\text{Total cost} = \underbrace{\text{measure of fit}}_{RSS(\theta)} + \lambda \cdot \underbrace{\text{measure of magnitude of coefficients}}_{\lambda \cdot \|\theta\|_2^2}$$
$$\implies \min \|\Phi\theta - y\|_2^2 + \boxed{\lambda \|\theta\|_2^2}$$

- Try this loss instead of ridge...

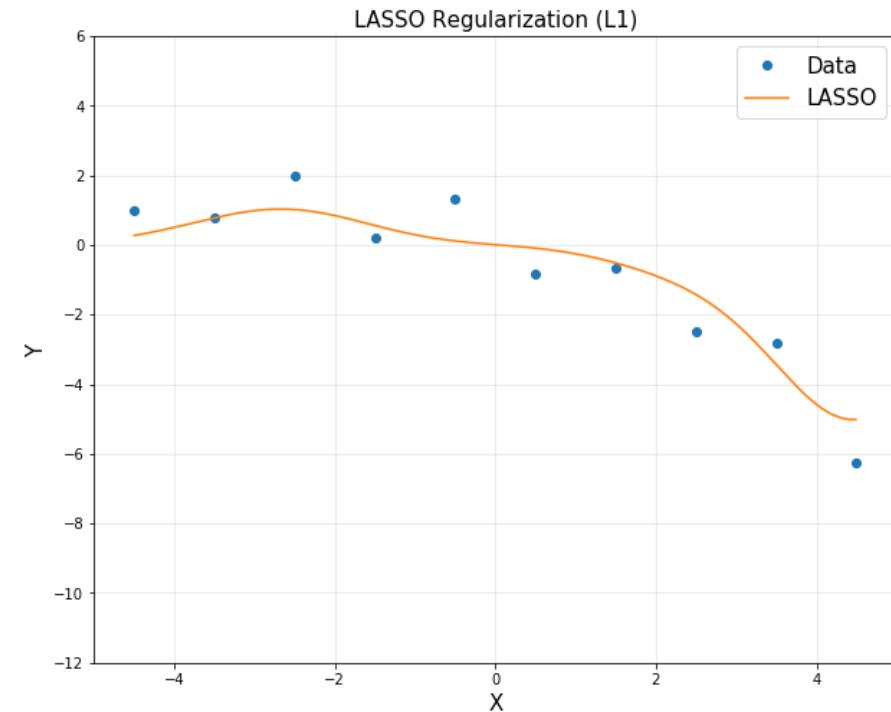
$$\text{Total cost} = \underbrace{\text{measure of fit}}_{RSS(\theta)} + \lambda \cdot \underbrace{\text{measure of magnitude of coefficients}}_{\lambda \cdot \|\theta\|_1}$$
$$\implies \min \|\Phi\theta - y\|_2^2 + \boxed{\lambda \|\theta\|_1}$$

- λ is a tuning parameter = balance of fit and sparsity
- Known as *LASSO*
 - least absolute shrinkage and selection operator

RBF with LASSO

```
# LASSO regression  
  
lamb = 2  
theta = cvx.Variable([d, 1])  
obj = cvx.Minimize(cvx.sum_squares(A*theta - y) + lamb*cvx.norm(theta, 1))  
prob = cvx.Problem(obj).solve()  
  
yp = rbfbasis*theta.value
```

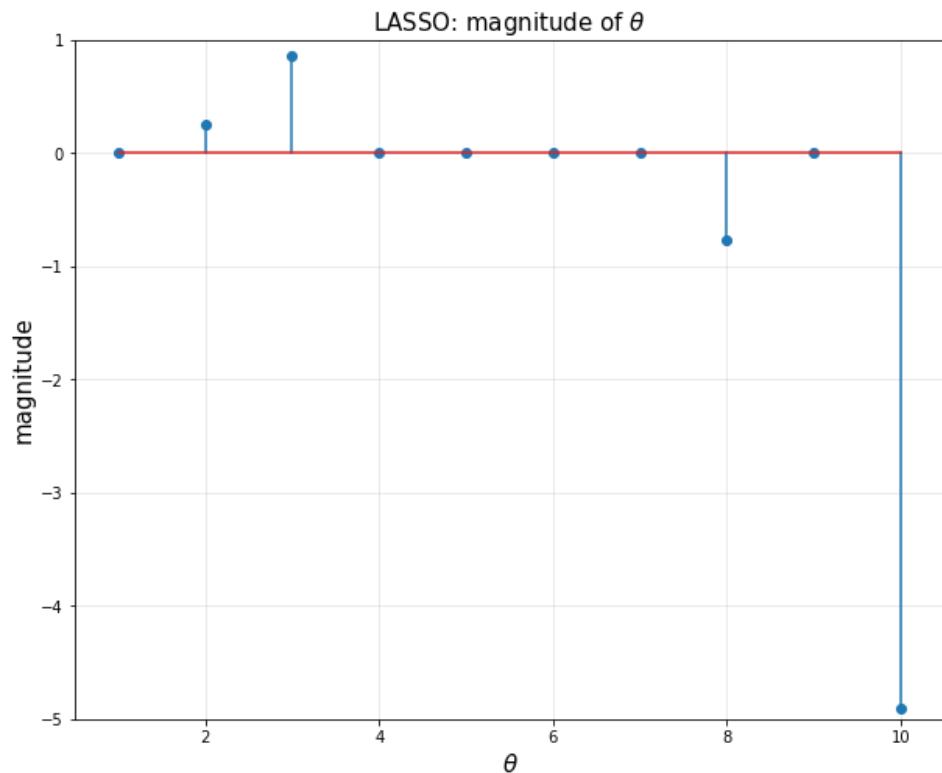
$$\min \|\Phi\theta - y\|_2^2 + \lambda\|\theta\|_1$$



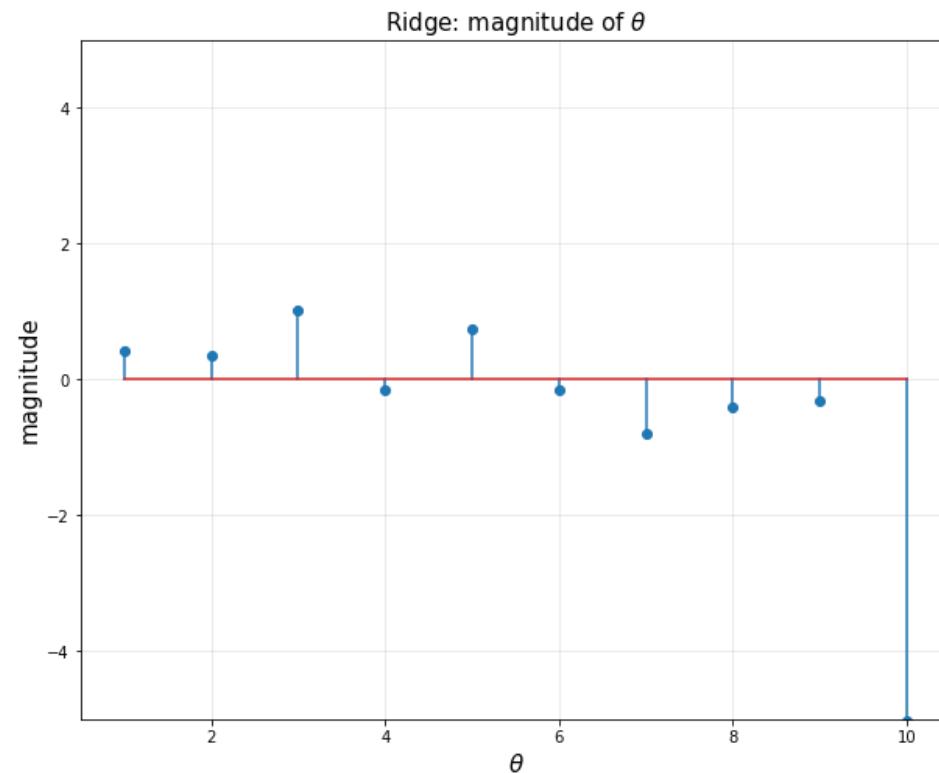
- Approximated function looks similar to that of ridge regression

Coefficients θ with LASSO

- Non-zero coefficients indicate ‘selected’ features



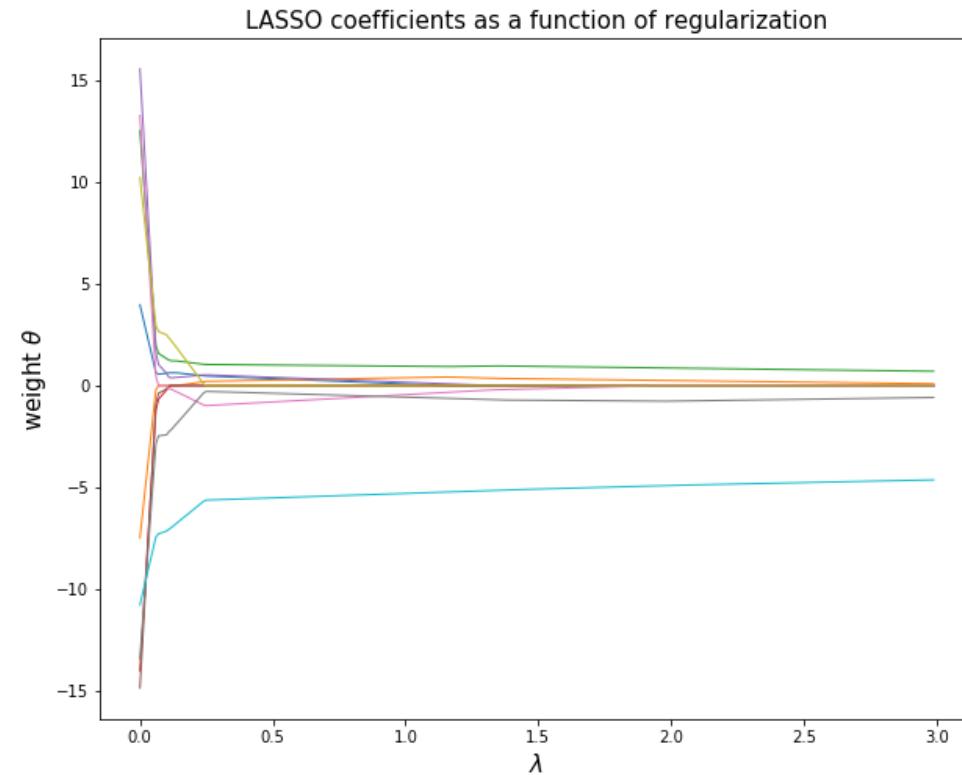
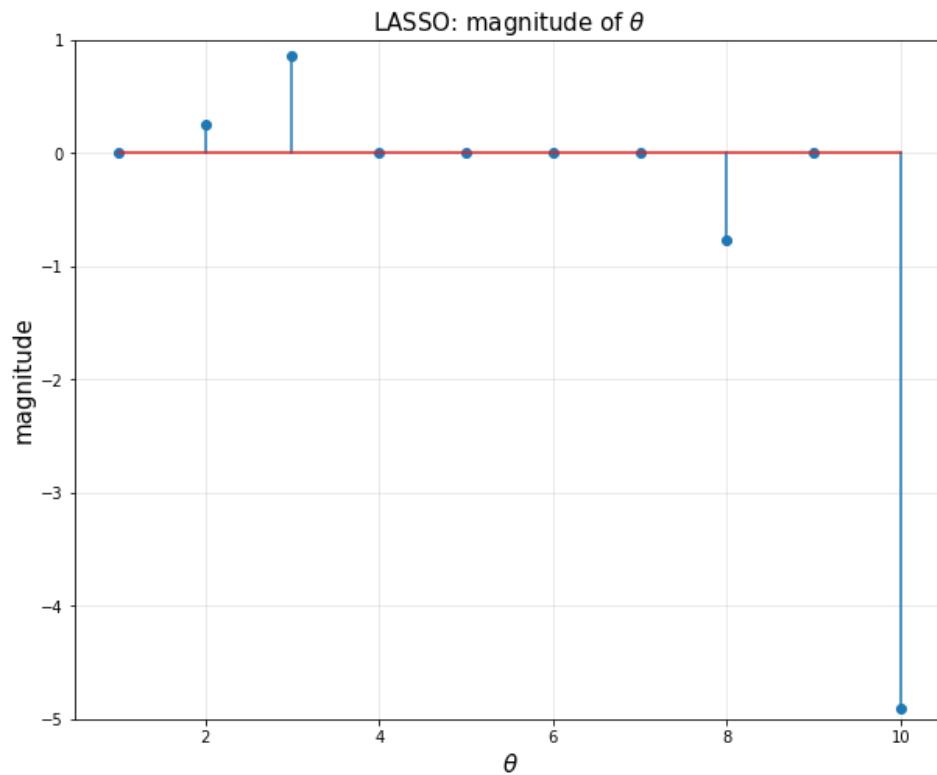
LASSO



Ridge

Coefficients θ with LASSO

- Non-zero coefficients indicate ‘selected’ features



Sparsity for Feature Selection using Lasso

- Least squares with a penalty on the L_1 norm of the parameters
- Start with full model (all possible features)
- ‘Shrink’ some coefficients exactly to 0
 - *i.e.*, knock out certain features
 - The L_1 penalty has the effect of forcing some of the coefficient estimates to be exactly equal to zero
- Non-zero coefficients indicate ‘selected’ features

Regression with Selected Features

```
# reduced order model
# we will use only theta 2, 3, 8, 10

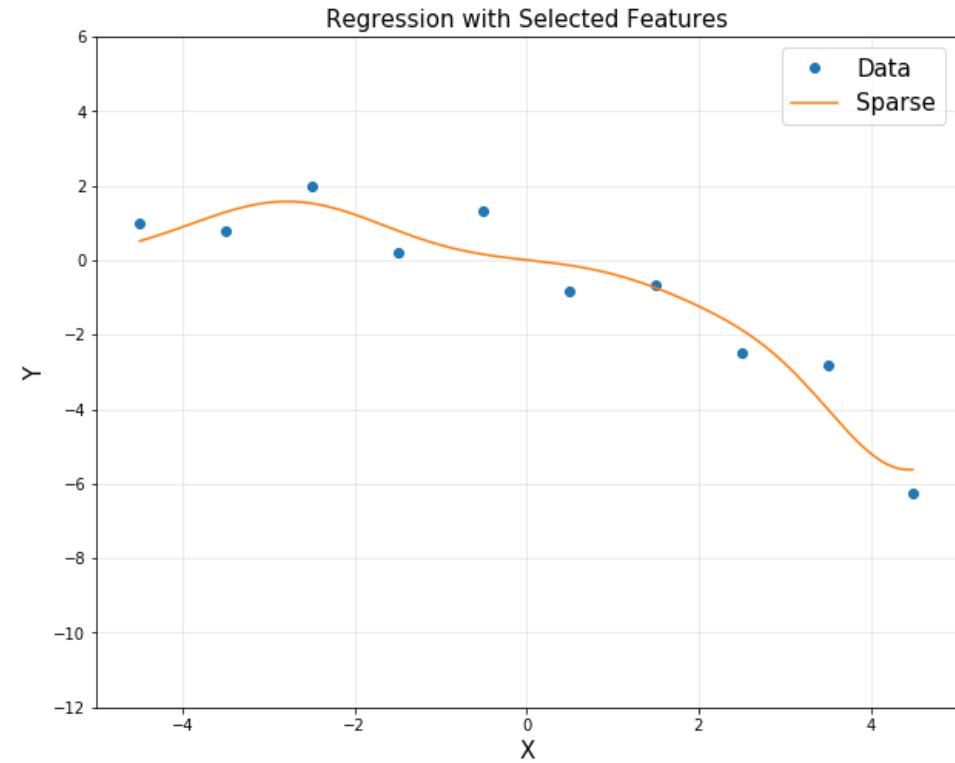
d = 4
u = np.array([-3.5, -2.5, 2.5, 4.5])
sigma = 1

rbfbasis = np.hstack([np.exp(-(xp-u[i])**2/(2*sigma**2)) for i in range(d)])
A = np.hstack([np.exp(-(x-u[i])**2/(2*sigma**2)) for i in range(d)])

rbfbasis = np.asmatrix(rbfbasis)
A = np.asmatrix(A)

theta = cvx.Variable([d, 1])
obj = cvx.Minimize(cvx.norm(A*theta-y, 2))
prob = cvx.Problem(obj).solve()

yp = rbfbasis*theta.value
```



LASSO vs. Ridge

- Another equivalent forms of optimizations

$$\min \|\Phi\theta - y\|_2^2 + \lambda \|\theta\|_1$$

$$\min \|\Phi\theta - y\|_2^2 + \lambda \|\theta\|_2^2$$

\implies

$$\begin{aligned} & \min_{\theta} \|\Phi\theta - y\|_2^2 \\ \text{subject to } & \|\theta\|_1 \leq s_1 \end{aligned}$$

$$\begin{aligned} & \min_{\theta} \|\Phi\theta - y\|_2^2 \\ \text{subject to } & \|\theta\|_2 \leq s_2 \end{aligned}$$

LASSO vs. Ridge

- Another equivalent forms of optimizations

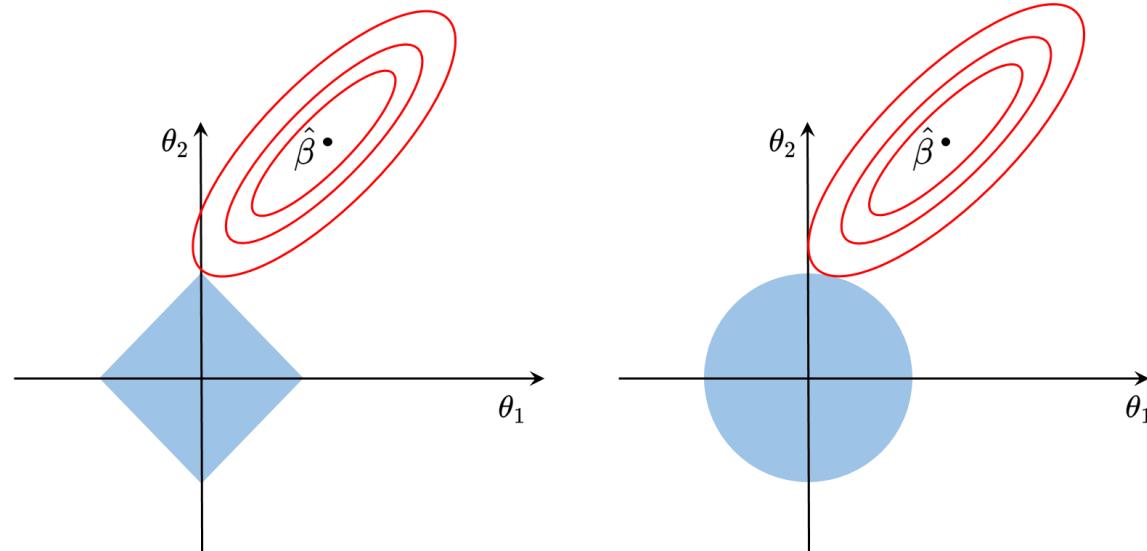
$$\min \|\Phi\theta - y\|_2^2 + \lambda \|\theta\|_1$$

$$\min \|\Phi\theta - y\|_2^2 + \lambda \|\theta\|_2^2$$

\implies

$$\begin{aligned} & \min_{\theta} \|\Phi\theta - y\|_2^2 \\ \text{subject to } & \|\theta\|_1 \leq s_1 \end{aligned}$$

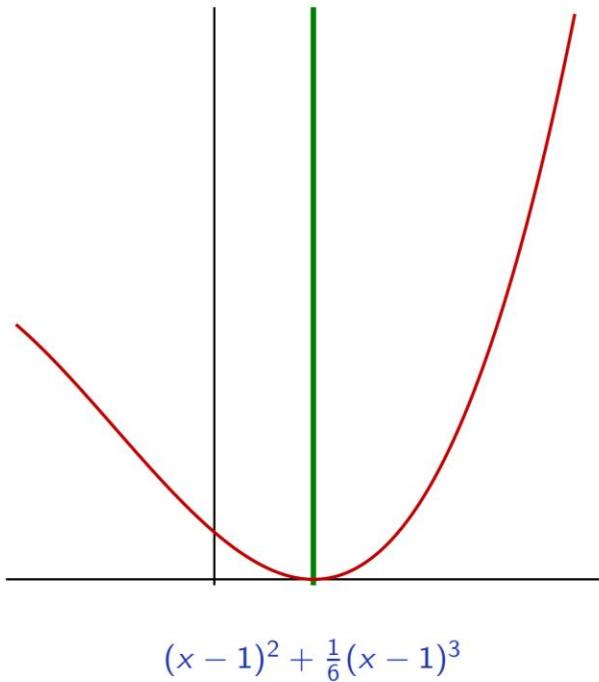
$$\begin{aligned} & \min_{\theta} \|\Phi\theta - y\|_2^2 \\ \text{subject to } & \|\theta\|_2 \leq s_2 \end{aligned}$$



L2 Regularizers: Simple Example

Increasing the λ parameter moves the optimal closer to 0, and away from the optimal for the loss alone.

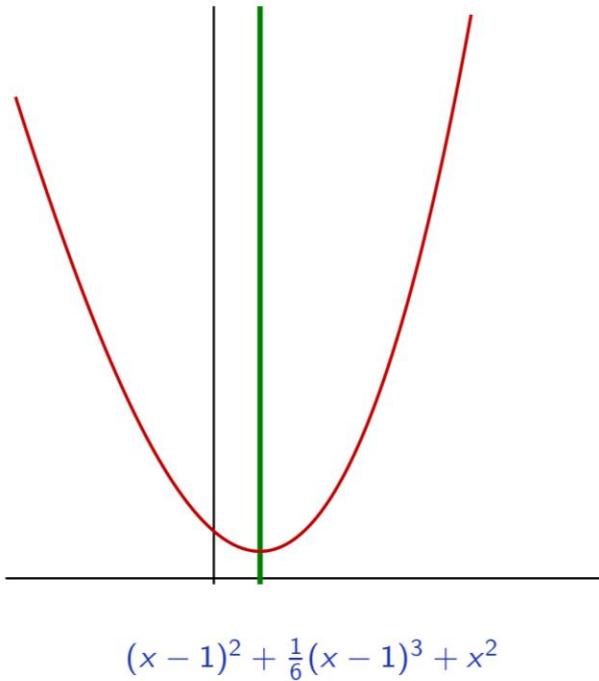
Since the derivative of $\|x\|_2^2$ is zero at zero, the optimal will never move there if it was not already there.



L2 Regularizers: Simple Example

Increasing the λ parameter moves the optimal closer to 0, and away from the optimal for the loss alone.

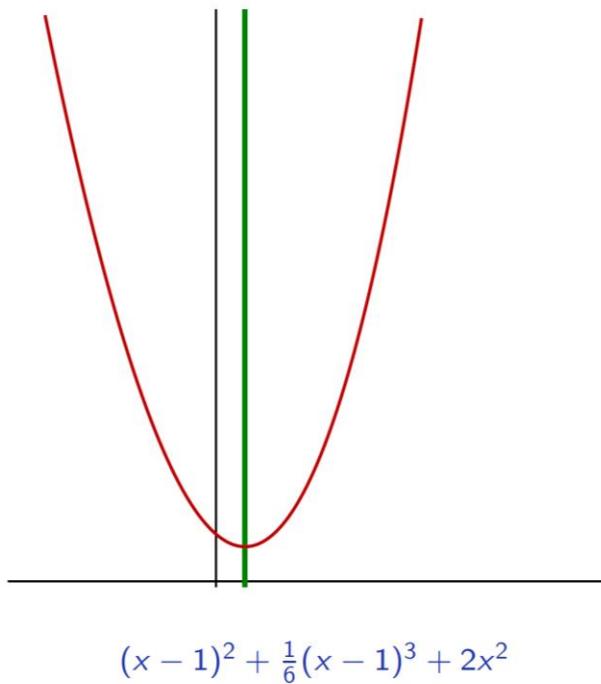
Since the derivative of $\|x\|_2^2$ is zero at zero, the optimal will never move there if it was not already there.



L2 Regularizers: Simple Example

Increasing the λ parameter moves the optimal closer to 0, and away from the optimal for the loss alone.

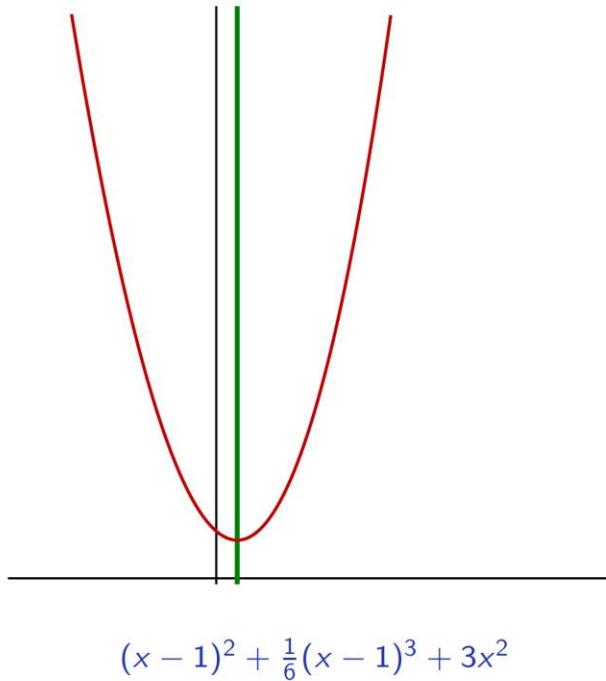
Since the derivative of $\|x\|_2^2$ is zero at zero, the optimal will never move there if it was not already there.



L2 Regularizers: Simple Example

Increasing the λ parameter moves the optimal closer to 0, and away from the optimal for the loss alone.

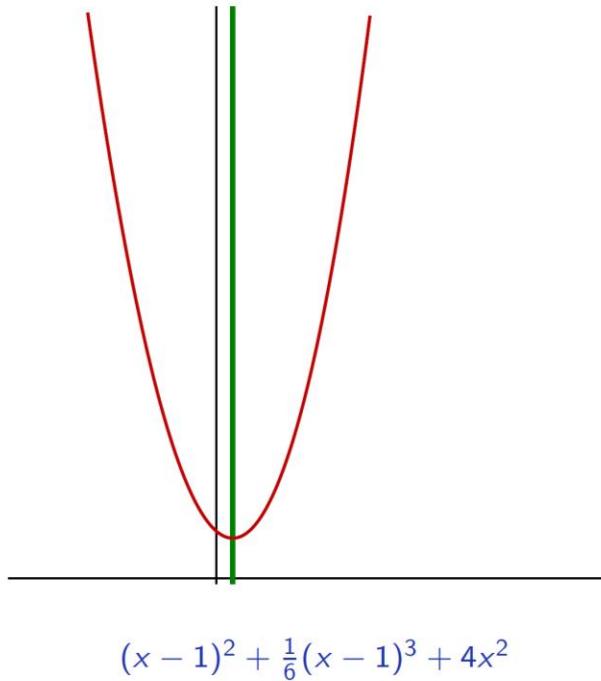
Since the derivative of $\|x\|_2^2$ is zero at zero, the optimal will never move there if it was not already there.



L2 Regularizers: Simple Example

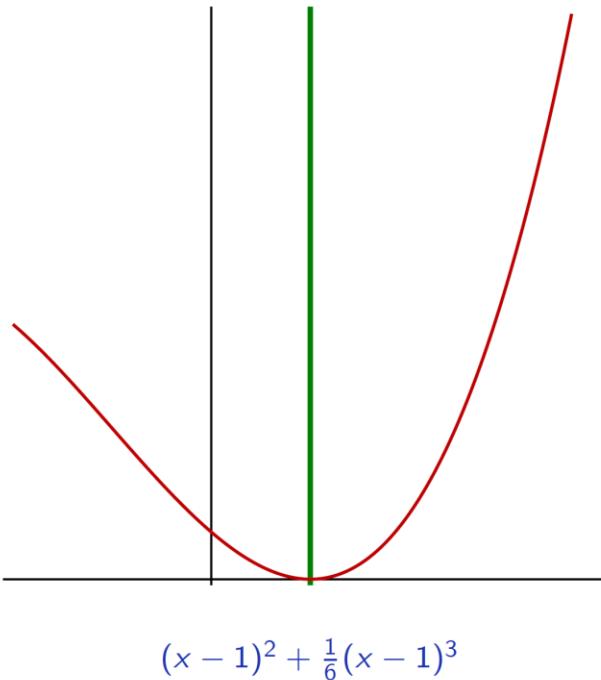
Increasing the λ parameter moves the optimal closer to 0, and away from the optimal for the loss alone.

Since the derivative of $\|x\|_2^2$ is zero at zero, the optimal will never move there if it was not already there.



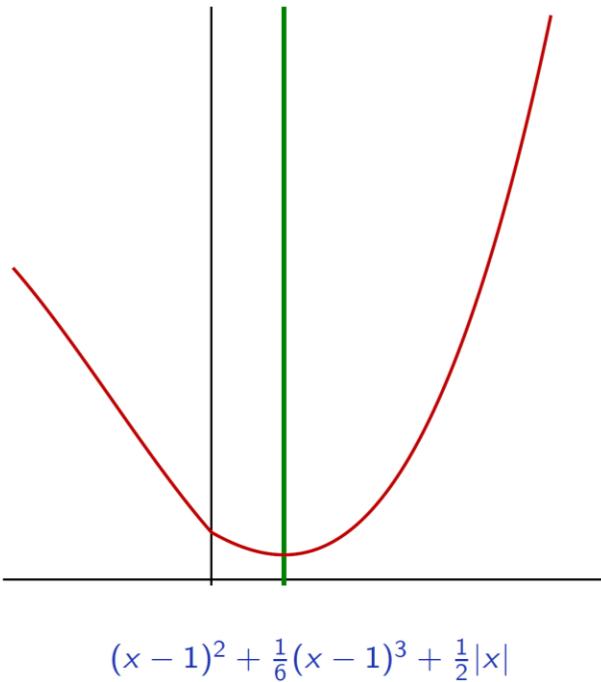
L1 Regularizers: Simple Example

Increasing the λ parameter moves the optimal closer to 0, and away from the optimal for the loss without penalty.



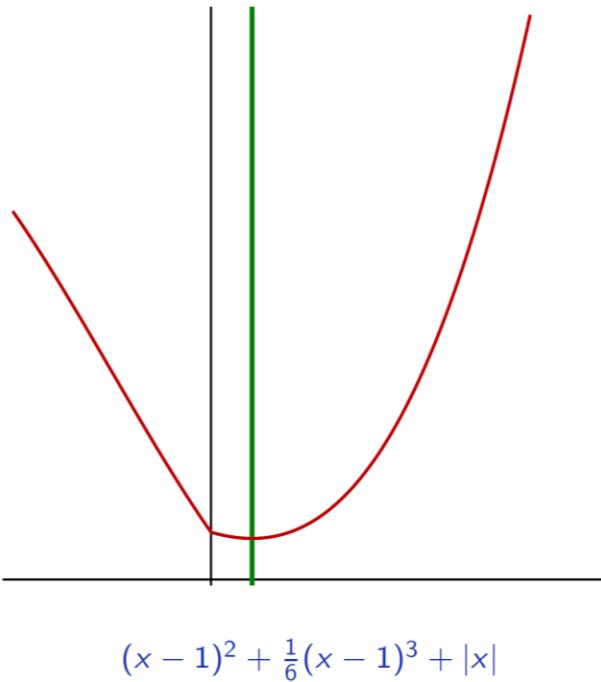
L1 Regularizers: Simple Example

Increasing the λ parameter moves the optimal closer to 0, and away from the optimal for the loss without penalty.



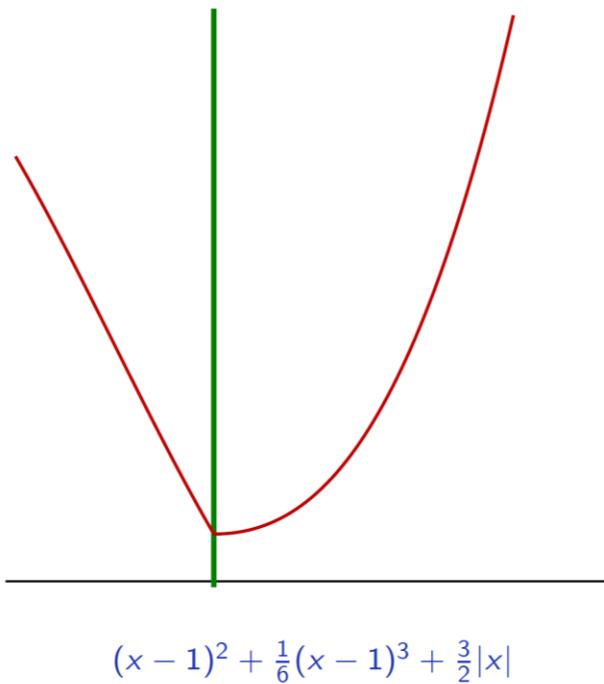
L1 Regularizers: Simple Example

Increasing the λ parameter moves the optimal closer to 0, and away from the optimal for the loss without penalty.



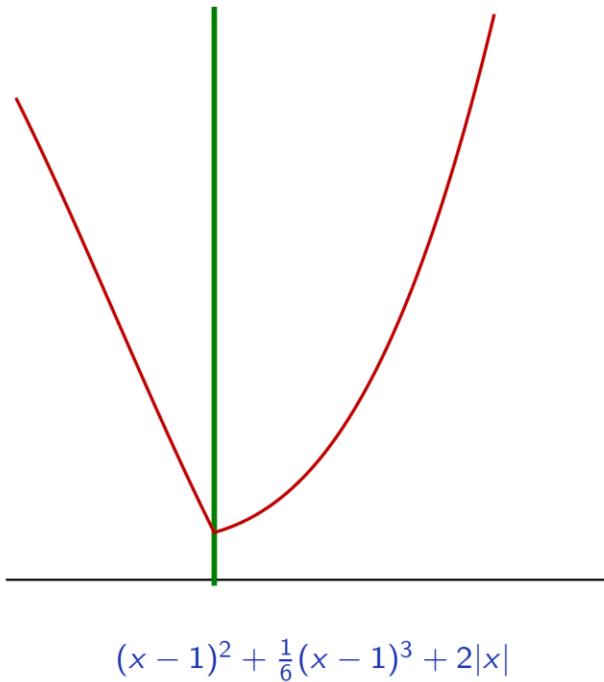
L1 Regularizers: Simple Example

Increasing the λ parameter moves the optimal closer to 0, and away from the optimal for the loss without penalty.



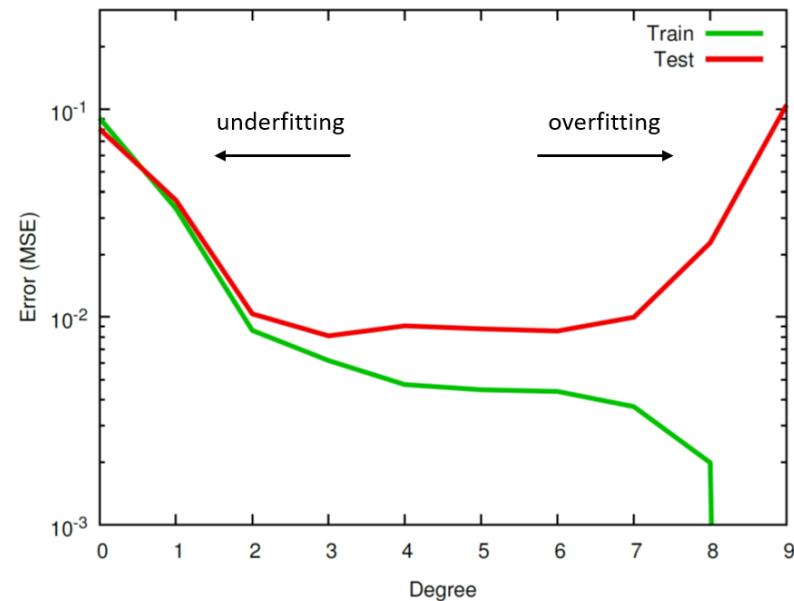
L1 Regularizers: Simple Example

Increasing the λ parameter moves the optimal closer to 0, and away from the optimal for the loss without penalty.



Evaluation

- Adding more features will always decrease the loss
- How do we determine when an algorithm achieves “good” performance?
- A better criterion:
 - Training set (e.g., 70 %)
 - Testing set (e.g., 30 %)



- Performance on testing set called *generalization* performance