Lecture 5 pre-video

Vector norms

How we measure distances between vectors

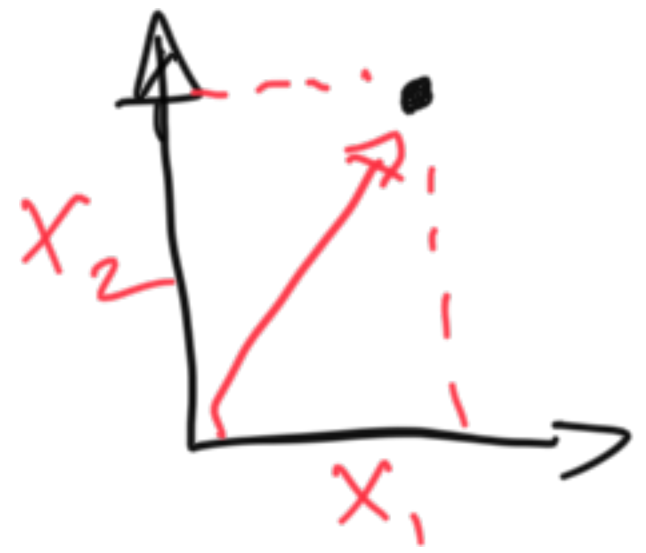
Vector norms

- Named vector norms L1, L2, … named after mathematician Henri Lebesgue (1875-1941)
- A vector norm $p : X \mapsto \mathbb{R}$ has the following properties (where X is a vector space)
    - Triangle inequality $p(x+y) \leq p(x) + p(y) \ \ \forall x, y \in X$
    - Absolute homogeneity $p(sx) = |s| p(x) \ \ \forall s \in \mathbb{R}, x \in X$
    - Positive definiteness $p(x) = 0 \iff x = 0$
    - Non-negativity $p(x) \geq 0 \ \ \forall x \in X$

Vector norms

L2: Euclidean

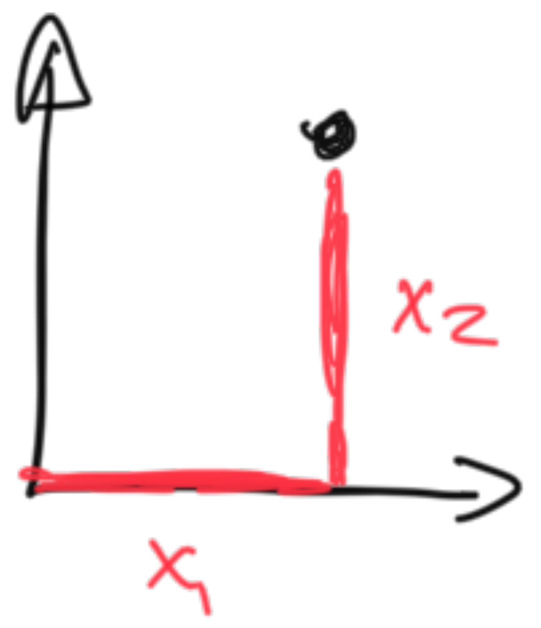
$$\|x\|_2 \Rightarrow \sqrt{x_1^2 + x_2^2 + \dots x_n^2}$$



Vector norms

L1: Absolute value / Manhattan distance

$$\|\mathbf{x}\|_1 = |x_1| + |x_2| + \dots |x_n|$$



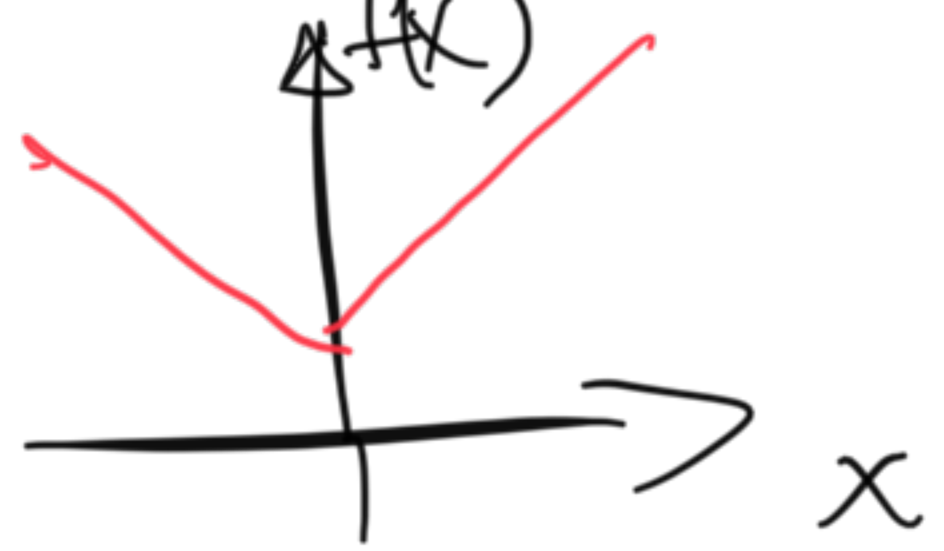
need abs otherwise violate non-negativity for vectors in lower/left quadrants

Vector norms

Application of L1 to an error function, and its derivative

for a linear fct $f(x)$

$$\| f(\boldsymbol{x}) \|_1$$



$$\frac{\partial \| f(\boldsymbol{x}) \|_1}{\partial \boldsymbol{x}} = \operatorname{Sign}(f(\boldsymbol{x})) \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$$

Vector norms

L∞

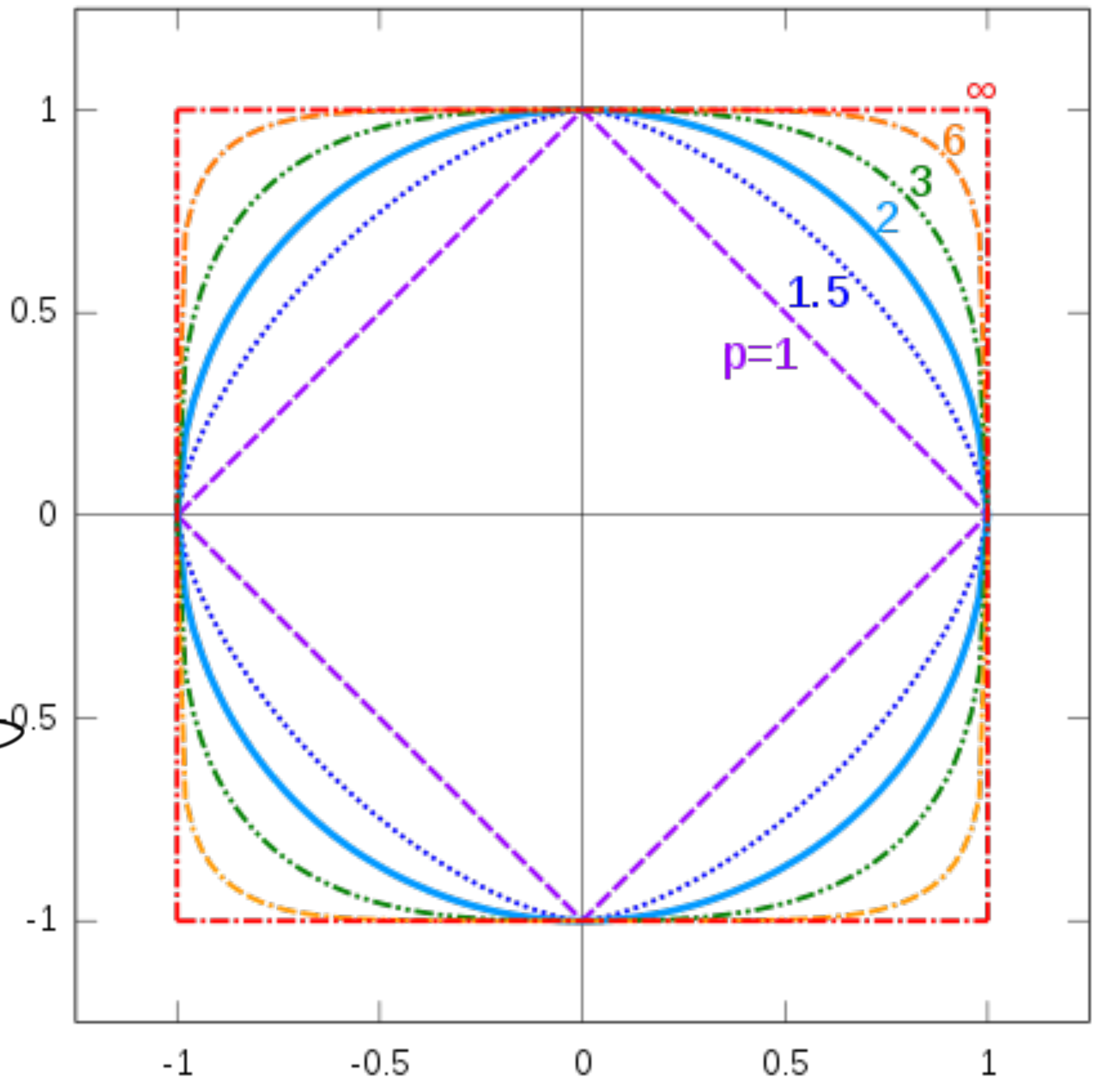
$$\|x\|_\infty = \max\left[x_1, x_2 \cdots x_n\right]$$

Vector norms

L*p*

$$\|x\|_p = \left(x_1^p + x_2^p + \dots x_n^p\right)^{1/p}$$

for the $x \in \mathbb{R}^2$ where x_1, x_2 are in range $[0,1]$ $\longrightarrow$



Lecture #3

Regularization to prevent overfitting + robust regression to minimize outliers

Jason G. Fleischer, Ph.D.
Asst. Teaching Professor
Department of Cognitive Science, UC San Diego

jfleischer@ucsd.edu

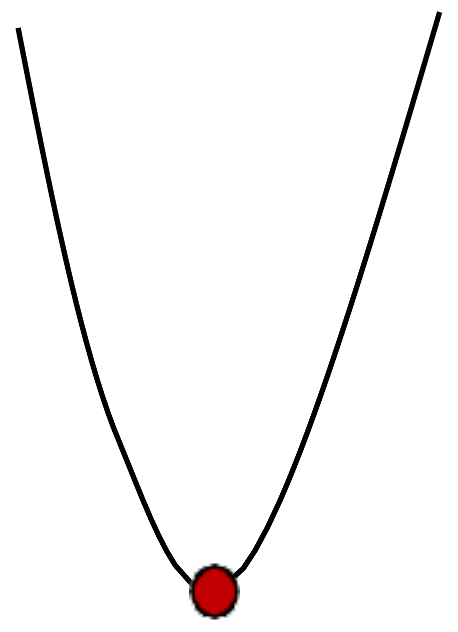
@jasongfleischer

https://jgfleischer.com

Ordinary least squares regression

- Univariate linear regression
- Polynomial Linear Regression
- Multivariate Linear Regression

Least squares estimation



Obtain/train: $f(x, W) = w_0 + w_1 x$

$$W^* = \arg min_W \sum_i (\mathbf{x}_i^T \cdot W - y_i)^2$$

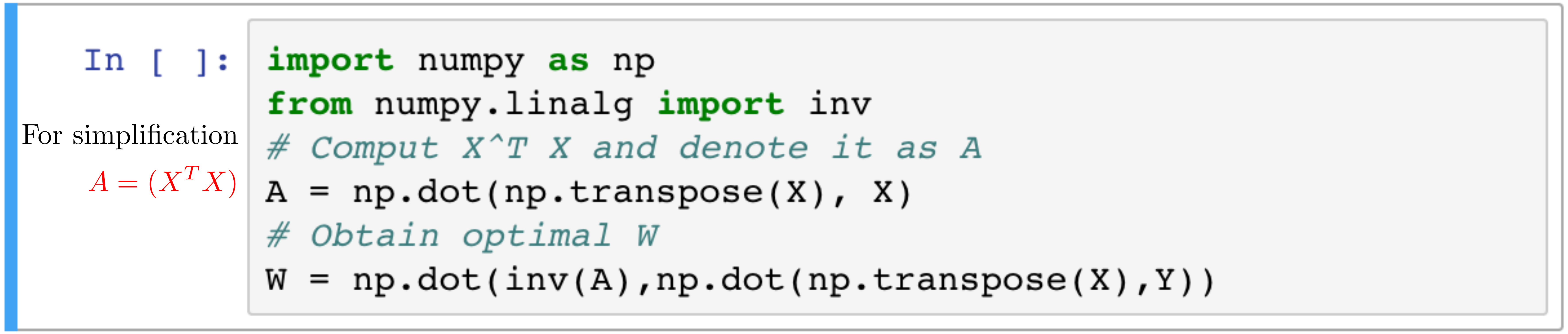
$$W = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} \quad \mathbf{x}_i = \begin{pmatrix} 1 \\ x_i \end{pmatrix}$$

$$W^* = \arg min_W = \arg min_W L(W) = (XW - Y)^T (XW - Y)$$

$$L(W) = W^T X^T X W - W^T X^T Y - Y^T X W + Y^T Y$$

$$\frac{dL(W)}{dW} = 2X^T X W - 2X^T Y = 0$$

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



In [ ]:

For simplification

$A = (X^T X)$

```
import numpy as np
from numpy.linalg import inv
# Comput X^T X and denote it as A
A = np.dot(np.transpose(X), X)
# Obtain optimal W
W = np.dot(inv(A),np.dot(np.transpose(X),Y))
```

Least Squares Solution

$$S_{training} = \{(x_i, y_i), i = 1..n\} = \{(1,1), (3,1.9), (2,1.05), (5,4.1), (4,2.1)\}$$

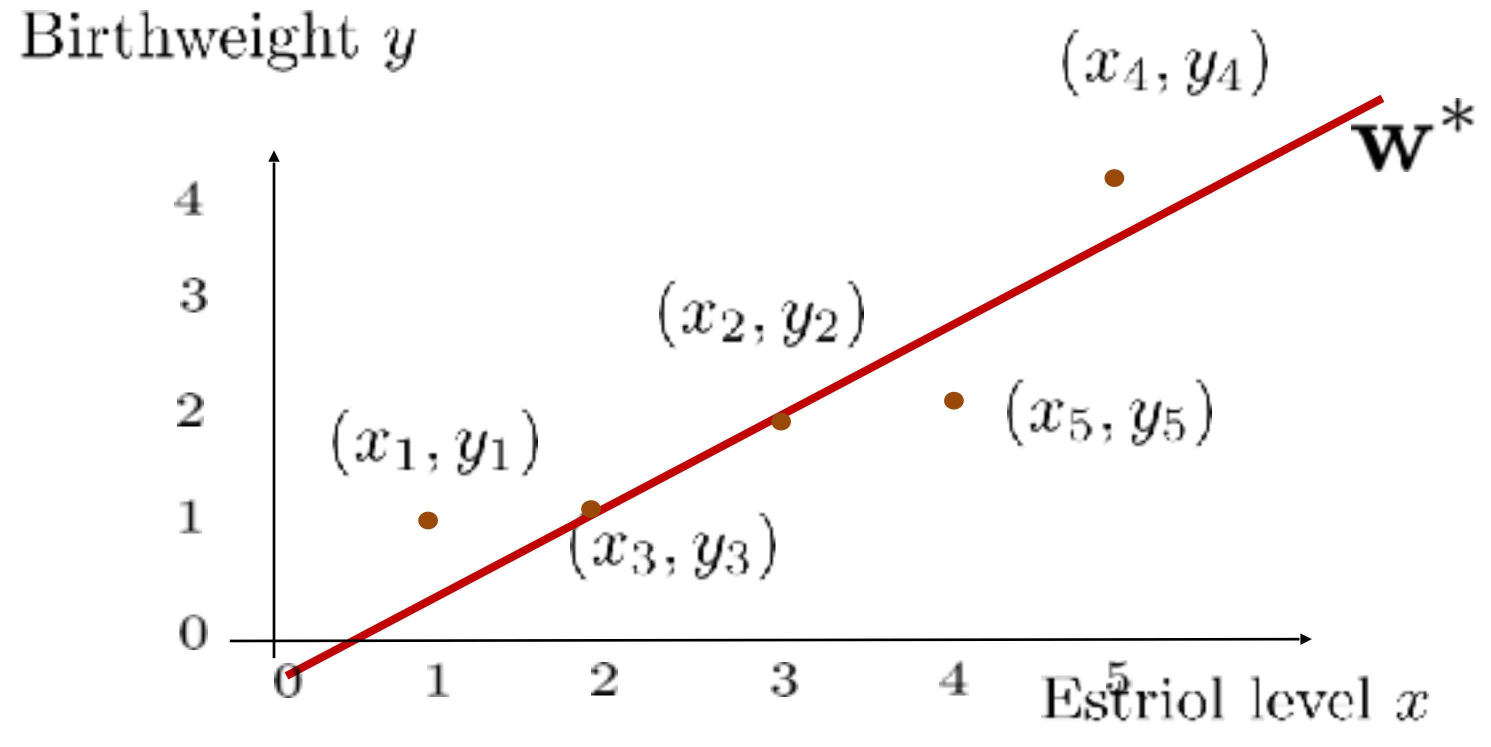
Basic Equations

$$y = w_0 + w_1 x$$

$$1 = w_0 + w_1 \times 1$$
$$1.9 = w_0 + w_1 \times 3$$
$$1.05 = w_0 + w_1 \times 2$$
$$4.1 = w_0 + w_1 \times 5$$
$$2.1 = w_0 + w_1 \times 4$$

Matrix Form

$$Y = XW$$



$$\begin{array}{cccc} Y & = & X & W \\ \begin{pmatrix} 1 \\ 1.9 \\ 1.05 \\ 4.1 \\ 2.1 \end{pmatrix} & & \begin{pmatrix} 1,1 \\ 1,3 \\ 1,2 \\ 1,5 \\ 1,4 \end{pmatrix} & \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} \end{array}$$

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

In python from numpy.linalg import inv

```
W* = np.dot(inv(np.dot(np.transpose(X), X) ),np.dot(np.transpose(X),Y))
```

$$\begin{pmatrix} -0.145 \\ 0.725 \end{pmatrix}$$

$$e_{training}(\mathbf{w}^*) = 0.21$$

Ordinary least squares regression

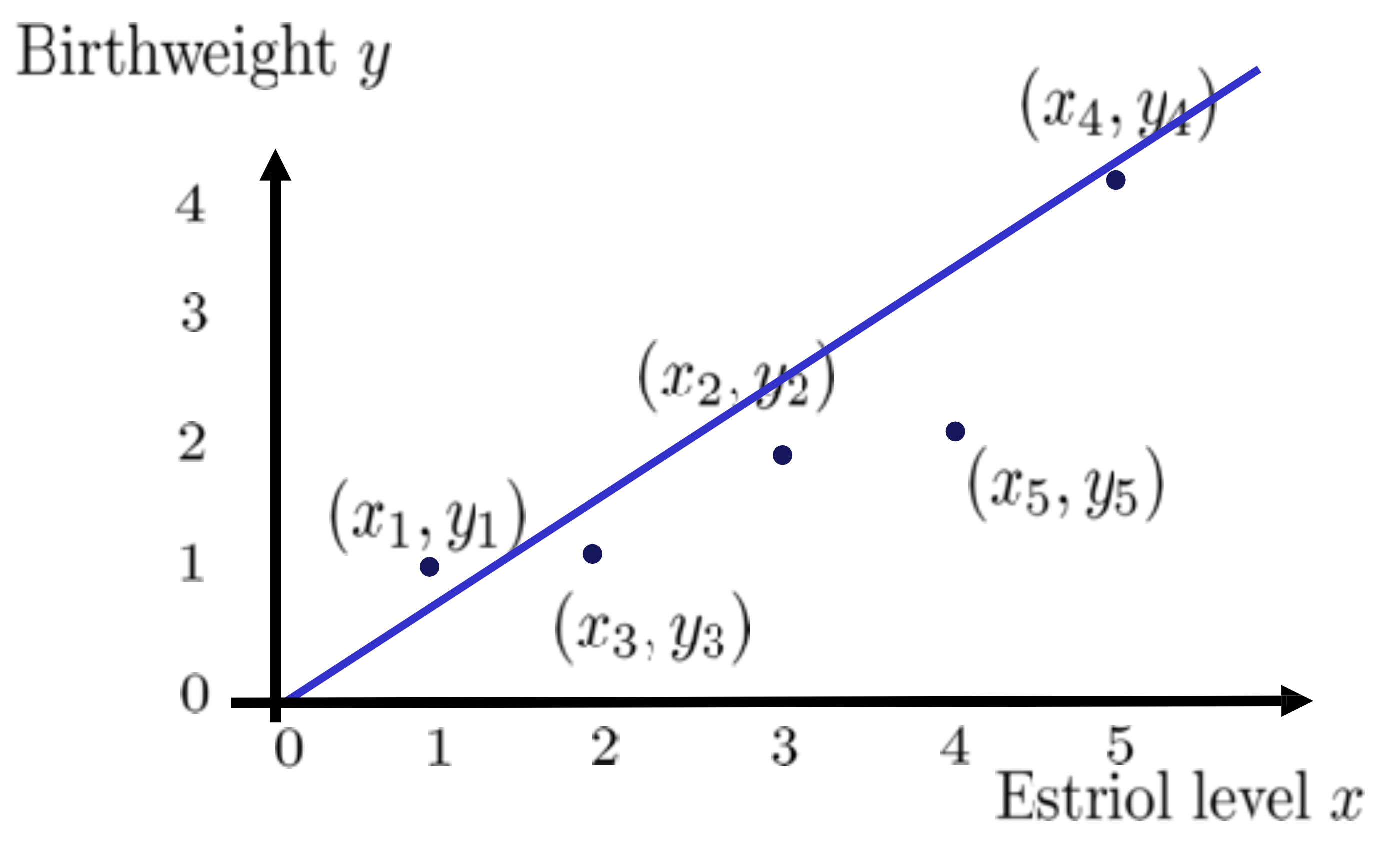
- Univariate linear regression

Now we extend the basic linear regression into more generalized forms.

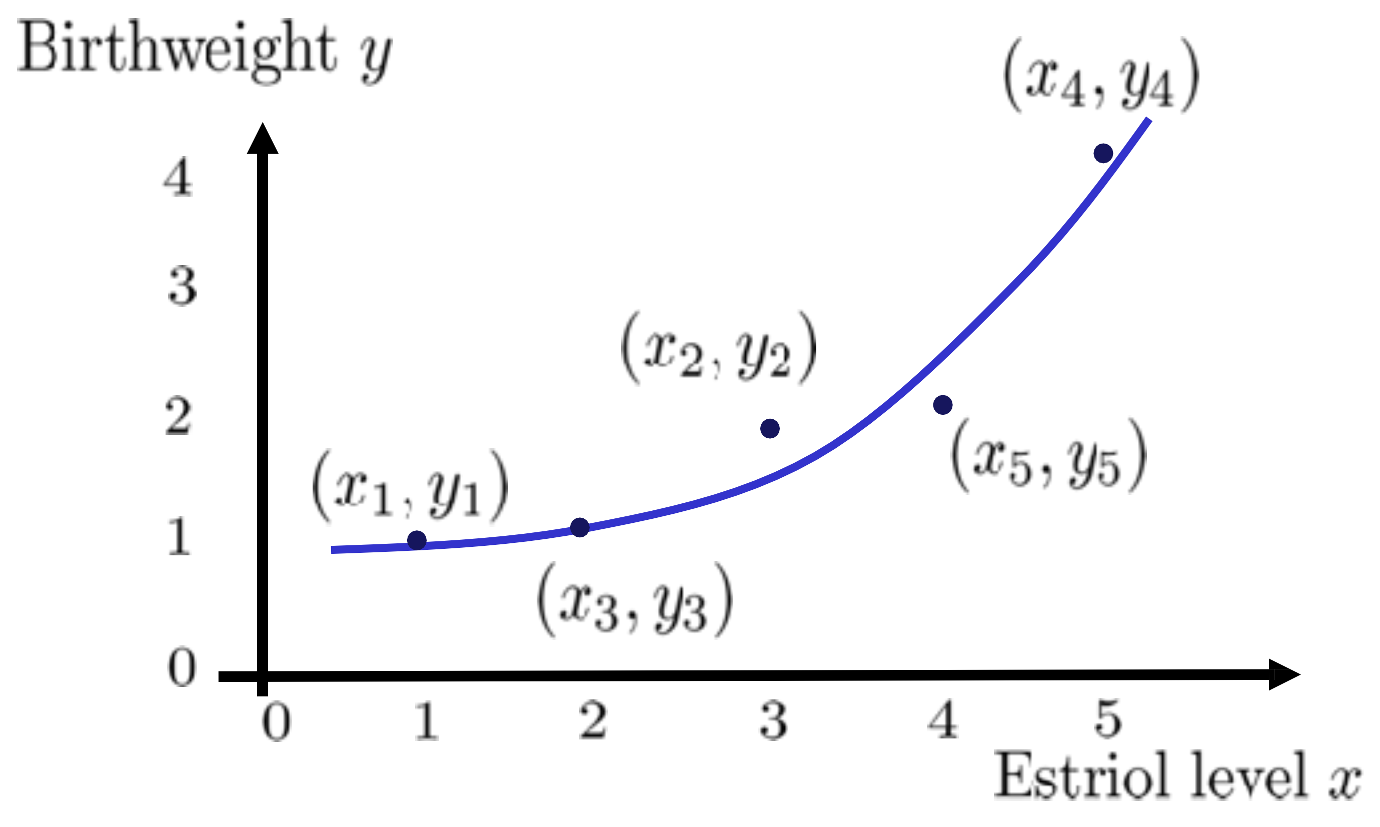
- Polynomial Linear Regression

$$\text{Output: } y = w_0 + w_1 x_1 + w_2 x_1^2 + \ldots + w_q x_m^q$$

- Multivariate Linear Regression



Linear



Polynomial

Polynomial Linear Regression

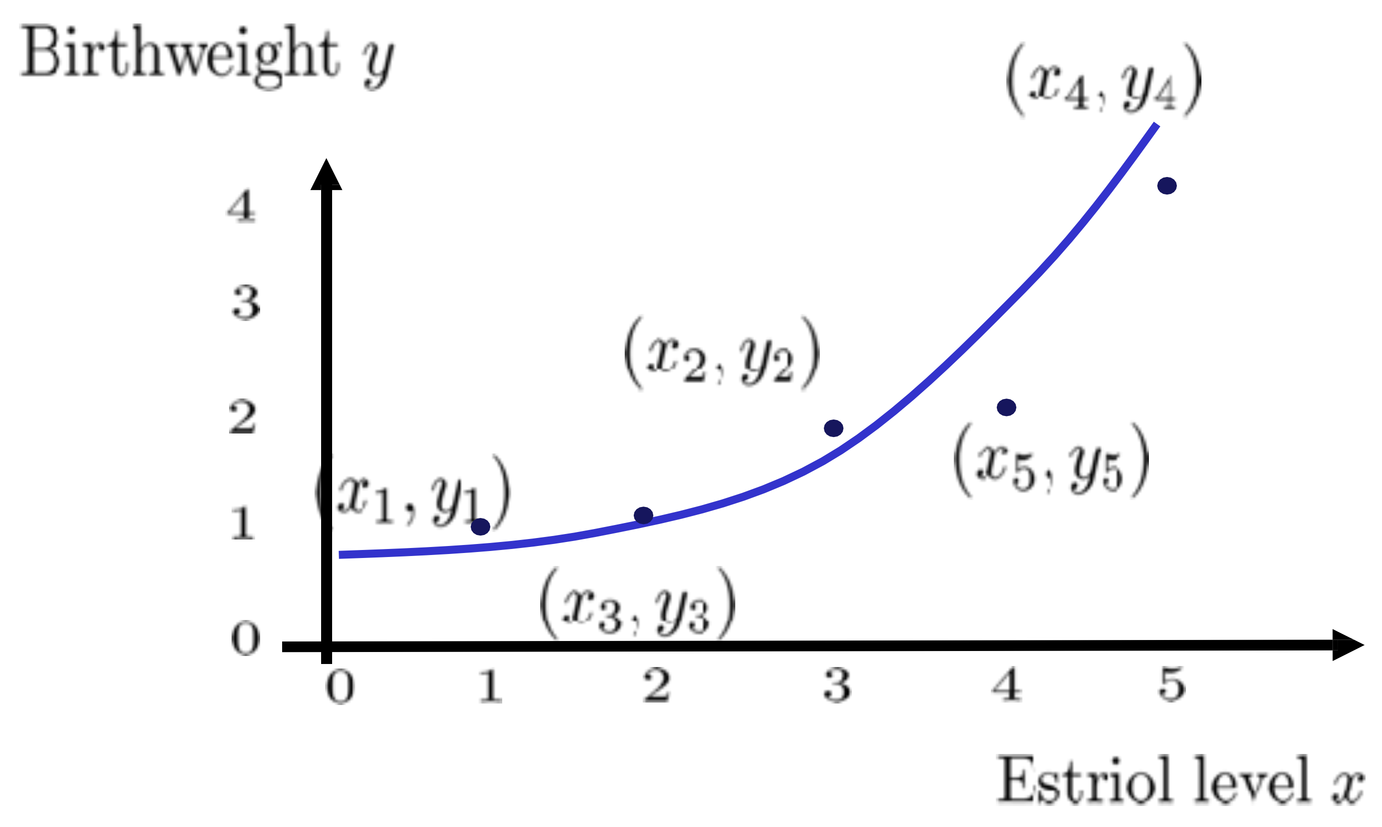
$$S_{training} = \{(x_i, y_i), i = 1..n\}$$

Input: $x,\ x \in R$

Model parameter: $\mathbf{w} = (w_0, w_1, ..., w_d),\ w_i \in R$

Output: $y = w_0 + w_1 x_1 + w_2 x_1^2 + ... + w_q x_m^q$

The combination of terms has your input variable raised to an additional power for each subsequent term.



Polynomial

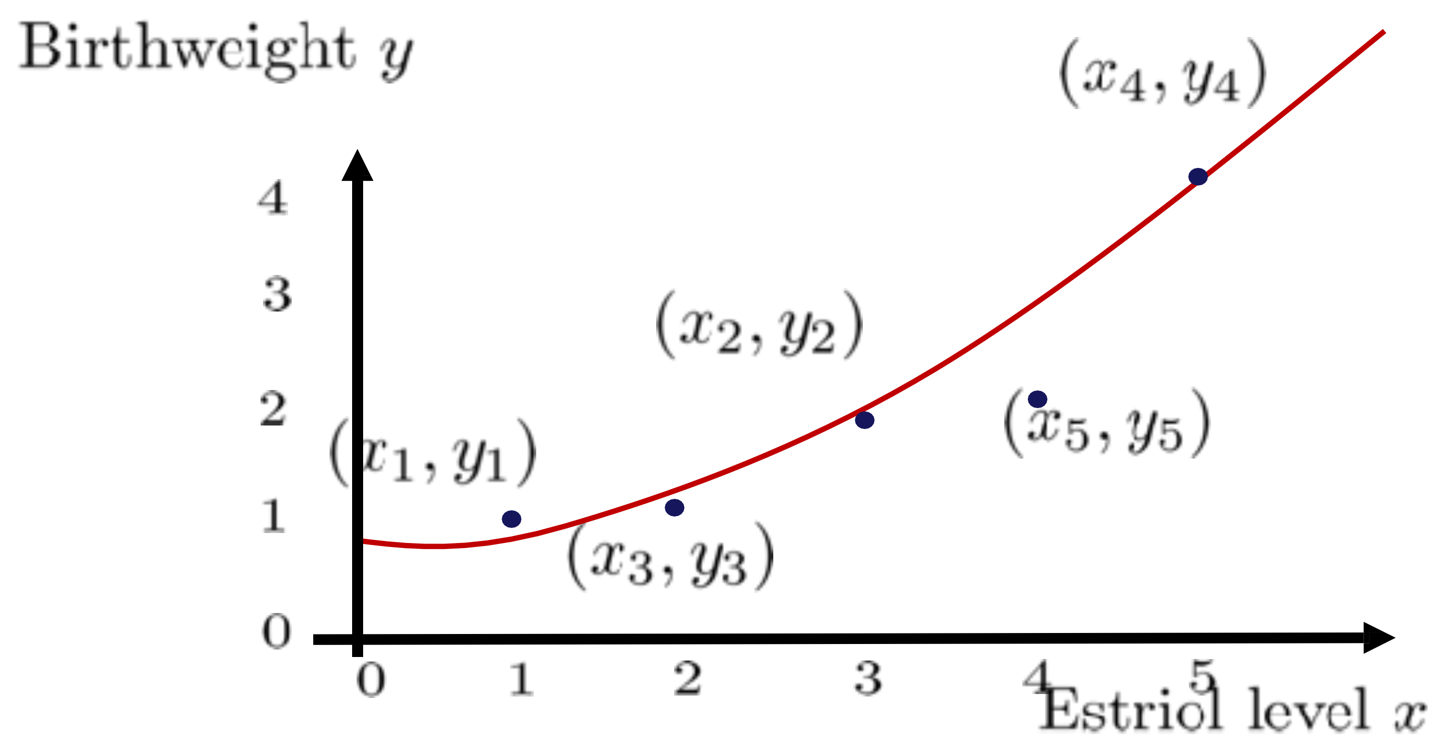
Put Data Into Matrix Form

$$S_{training} = \{(x_i, y_i), i = 1..n\} = \{(1,1), (3,1.9), (2,1.05), (5,4.1), (4,2.1)\}$$

Basic Equations **Matrix Form**

$$y = w_0 + w_1 x + w_2 x^2 \qquad Y = XW$$

$$1 = w_0 + w_1 \times 1 + w_2 \times 1$$
$$1.9 = w_0 + w_1 \times 3 + w_2 \times 9$$
$$1.05 = w_0 + w_1 \times 2 + w_2 \times 4$$
$$4.1 = w_0 + w_1 \times 5 + w_2 \times 25$$
$$2.1 = w_0 + w_1 \times 4 + +w_2 \times 16$$



In python from numpy.linalg import inv

```
W* = np.dot(inv(np.dot(np.transpose(X), X) ),np.dot(np.transpose(X),Y))
```

$$\begin{array}{cccc} Y & = & X & W \\ \begin{pmatrix} 1 \\ 1.9 \\ 1.05 \\ 4.1 \\ 2.1 \end{pmatrix} & & \begin{pmatrix} 1,1,1 \\ 1,3,9 \\ 1,2,4 \\ 1,5,25 \\ 1,4,16 \end{pmatrix} & \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} \end{array}$$

$$\begin{pmatrix} 1.48 \\ -0.67 \\ 0.2321 \end{pmatrix}$$

$$e_{training}(\mathbf{w}^*) = 0.063$$

Ordinary least squares regression

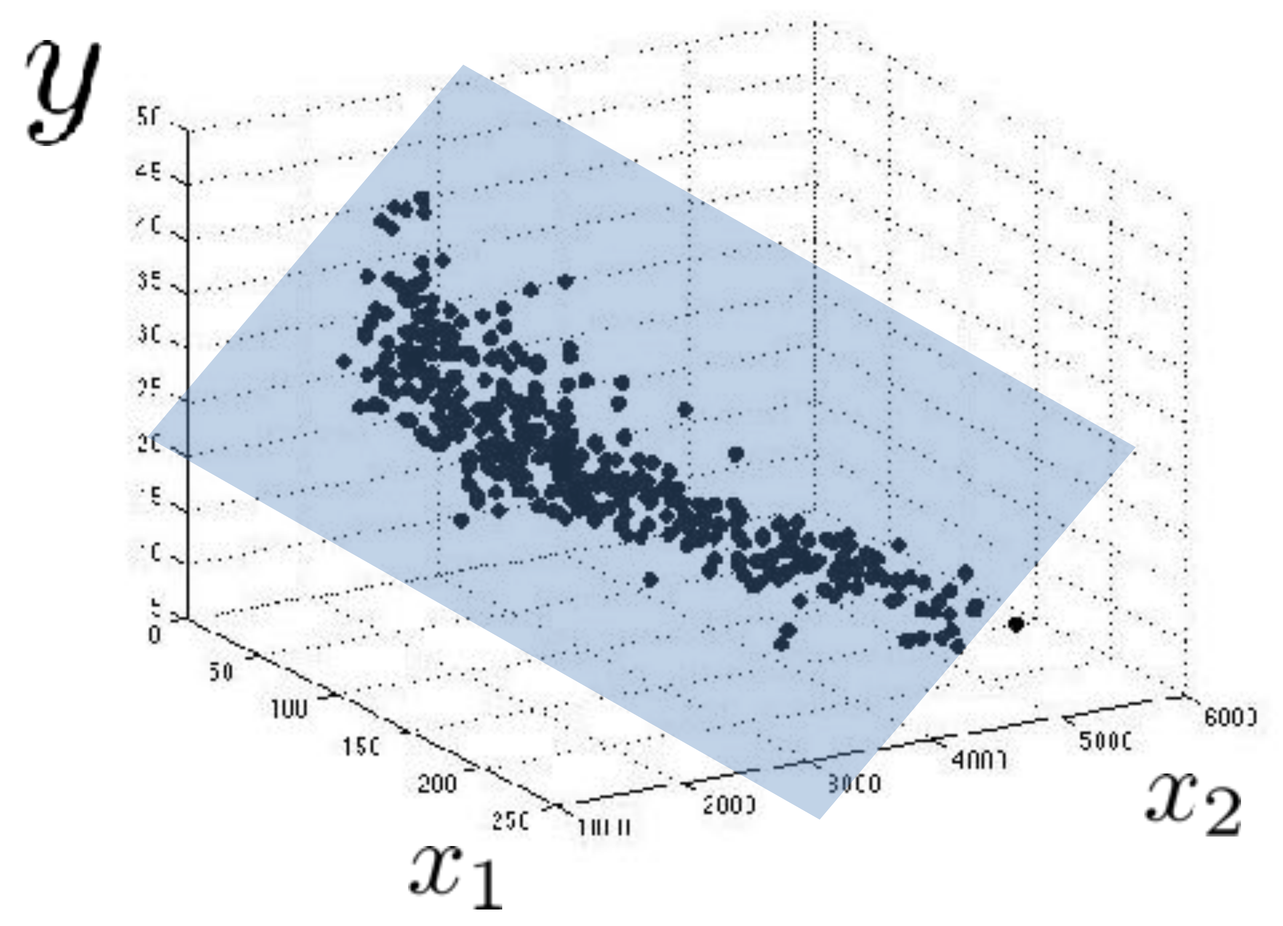
- Univariate linear regression

- Polynomial Linear Regression

- Multivariate Linear Regression

$$\text{Output: } y = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_m x_m$$

Multi-variate Linear Regression



Input: $\mathbf{x} = (x_1, ..., x_m),\ x_i \in R$

Model parameter: $\mathbf{w} = (w_0, w_1, ..., w_m),\ w_i \in R$

Output: $y = w_0 + w_1 x_1 + w_2 x_2 + ... + w_m x_m$

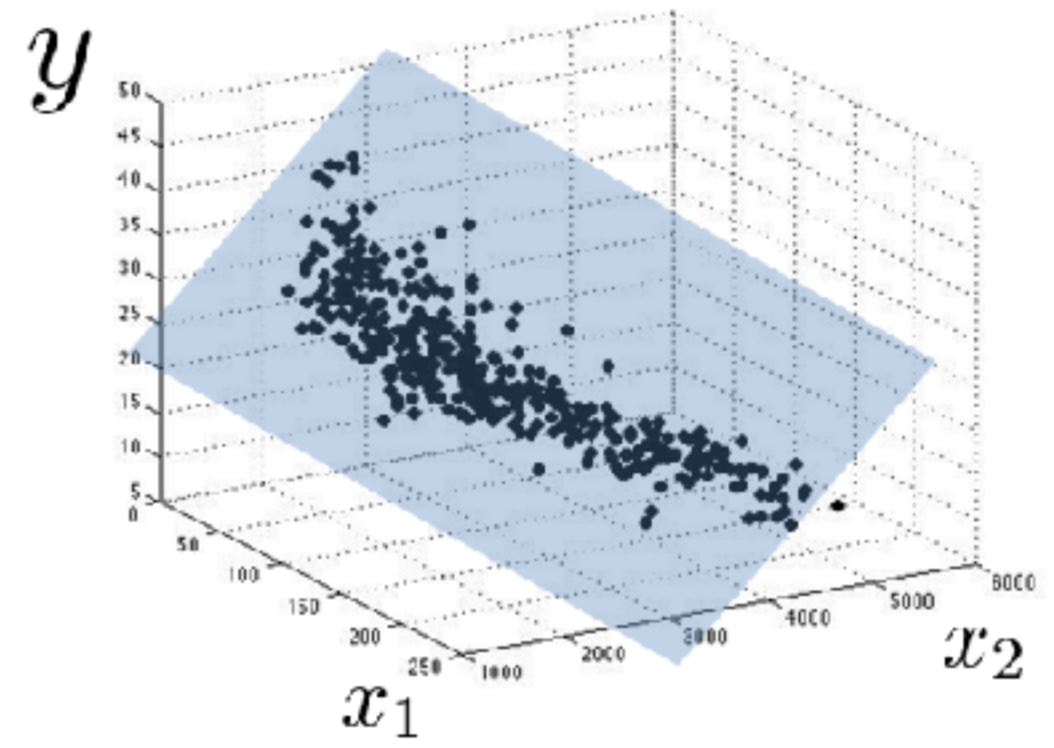
We apply do the least squre fitting method again.

Put Data Into Matrix Form

$$S_{training} = \{((x_{i1}, x_{i2}), y_i), i = 1..n\}$$
$$= \{((1, 0.5), 1), ((3, 0.9), 1.9), ((2, 1.0), 1.05), ((5, 6.7), 4.1), ((4, 2.5), 2.1)\}$$

Basic Equations **Matrix Form**

$$y = w_0 + w_1 x_1 + w_2 x_2 \qquad Y = XW$$



$$1 = w_0 + w_1 \times 1 + w_2 \times 0.5$$
$$1.9 = w_0 + w_1 \times 3 + w_2 \times 0.9$$
$$1.05 = w_0 + w_1 \times 2 + w_2 \times 1.0$$
$$4.1 = w_0 + w_1 \times 5 + w_2 \times 6.7$$
$$2.1 = w_0 + w_1 \times 4 + w_2 \times 2.5$$

$$\begin{array}{cccc} Y & = & X & W \\ \begin{pmatrix} 1 \\ 1.9 \\ 1.05 \\ 4.1 \\ 2.1 \end{pmatrix} & & \begin{pmatrix} 1, 1, 0.5 \\ 1, 3, 0.9 \\ 1, 2, 1.0 \\ 1, 5, 6.7 \\ 1, 4, 2.5 \end{pmatrix} & \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} \end{array}$$

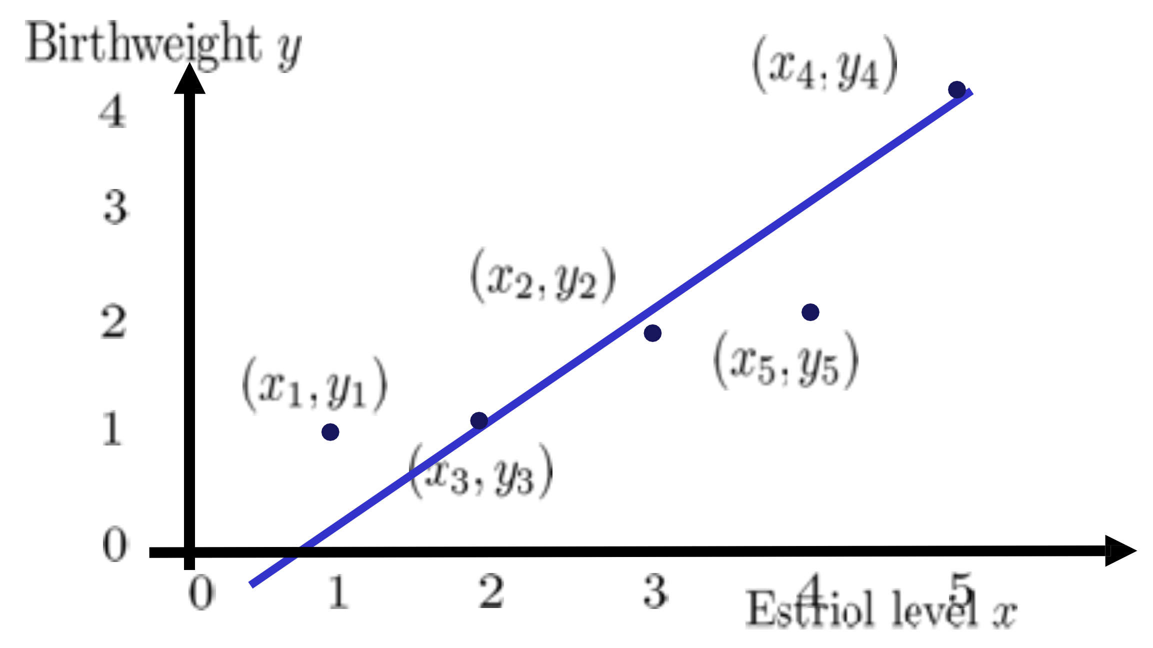
$$W^* = (X^T X)^{-1} X^T Y$$

$$\begin{pmatrix} 0.482 \\ 0.2552 \\ 0.338 \end{pmatrix}$$

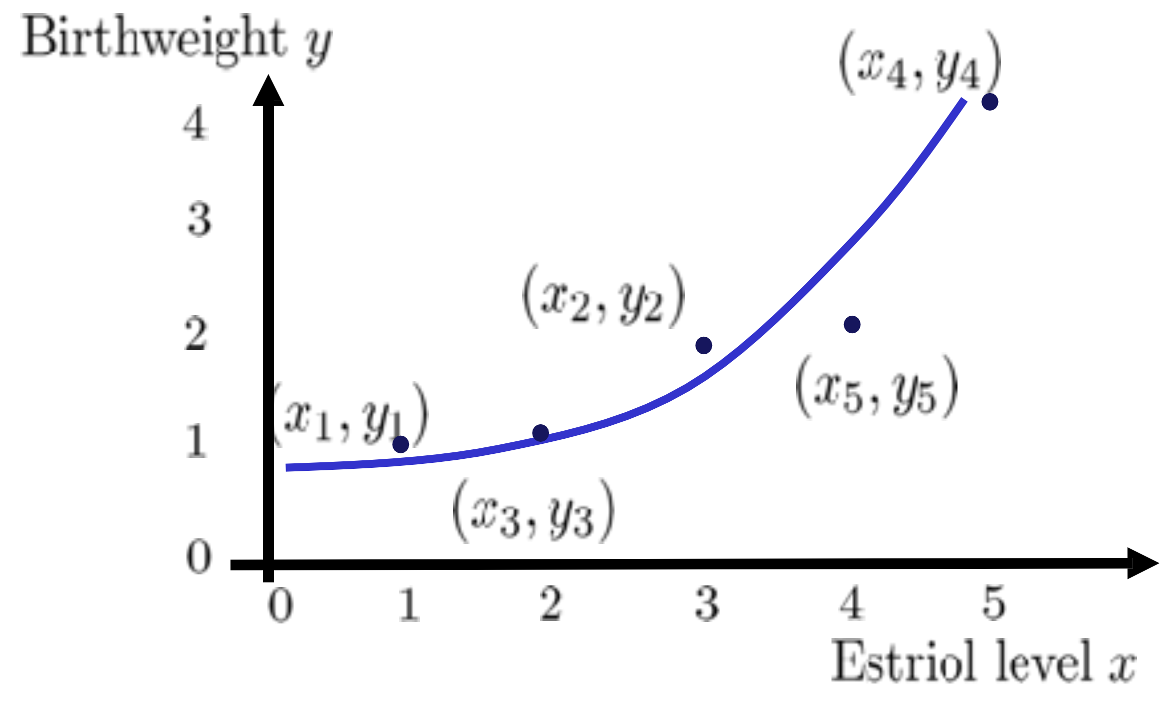
In python from numpy.linalg import inv

```
W* = np.dot(inv(np.dot(np.transpose(X), X) ),np.dot(np.transpose(X),Y))
```

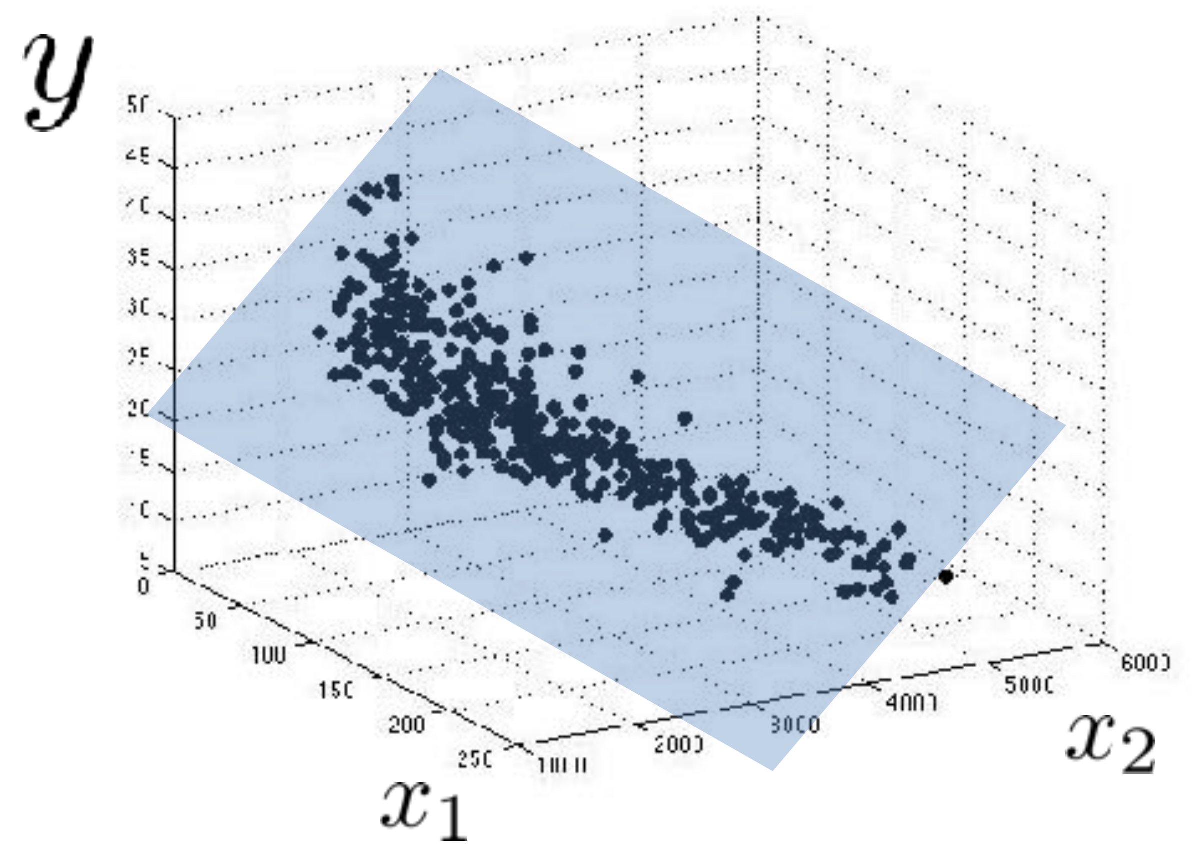
Select your model



Linear



Polynomial



$e_{training}(\mathbf{w}^*) = 0.21$

$e_{training}(\mathbf{w}^*) = 0.063$

$e_{training}(\mathbf{w}^*) = 0.052$

Conclusions for linear regression with the least squares estimation

Linear regression:

- Univariate linear regression

Output: $y = w_0 + w_1 x_1$

- Polynomial linear regression

Output: $y = w_0 + w_1 x_1 + w_2 x_1^2 + ... + w_q x_m^q$

- Multivariate linear regression

Output: $y = w_0 + w_1 x_1 + w_2 x_2 + ... + w_m x_m$

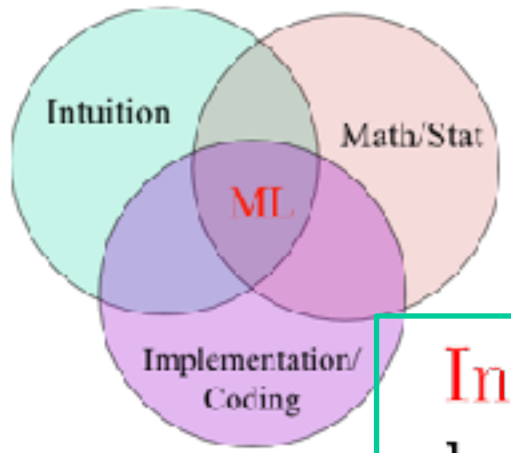
They all share a general form (when generalizing X):

$$Y \qquad = \qquad X \qquad W$$

linear w.r.t. the W!

With an analytical solution:

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



Recap: OLS Regression

Intuition: Linear (polynomial) regression is one of the most widely used machine learning models. Typically, a squared loss is adopted in training to minimize the averaged difference between the ground-truth values and model predictions for all the input data samples. In this case, the loss/objective function is in a quadratic (convex) form w.r.t. the model parameters, hence a convex function with a unique closed-form (analytical) solution by setting the gradient of the loss function to be zero. The learned model predicts a real number (e.g. length, price, weight) for a given input.

Math:

$$\underset{Y}{\begin{pmatrix} 1 \\ 1.9 \\ 1.05 \\ 4.1 \\ 2.1 \end{pmatrix}} = \underset{X}{\begin{pmatrix} 1,1,0.5 \\ 1,3,0.9 \\ 1,2,1.0 \\ 1,5,6.7 \\ 1,4,2.5 \end{pmatrix}} \underset{W}{\begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix}}$$

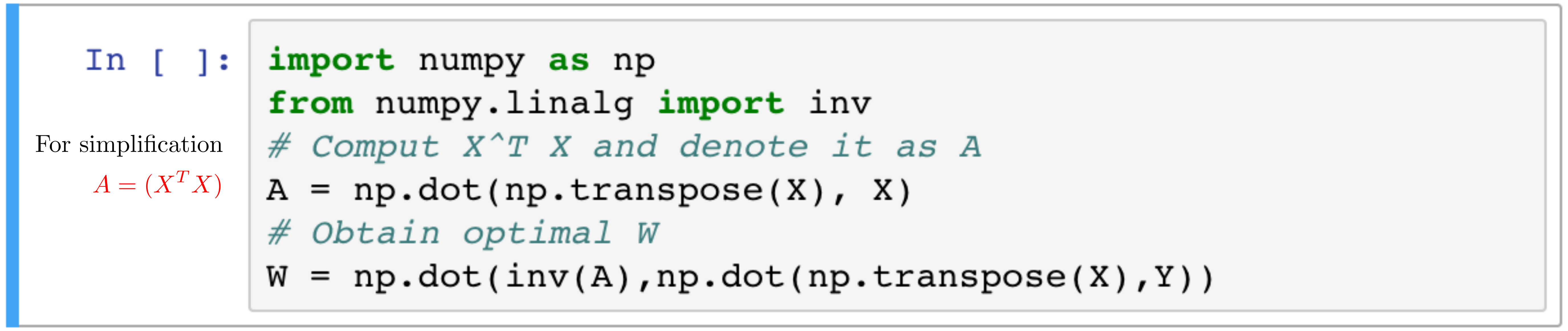
$$W^* = \arg min_W = \arg min_W L(W) = (XW - Y)^T(XW - Y)$$

$$L(W) = W^T X^T XW - W^T X^T Y - Y^T XW + Y^T Y$$

$$\frac{dL(W)}{dW} = 2X^T XW - 2X^T Y = 0$$

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

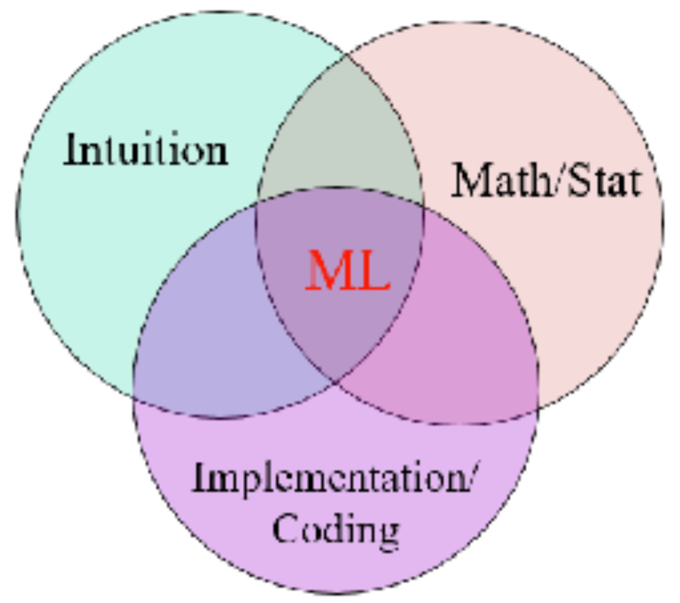
Implementation:



In [ ]:

```
import numpy as np
from numpy.linalg import inv
# Comput X^T X and denote it as A
A = np.dot(np.transpose(X), X)
# Obtain optimal W
W = np.dot(inv(A),np.dot(np.transpose(X),Y))
```

For simplification $A = (X^T X)$



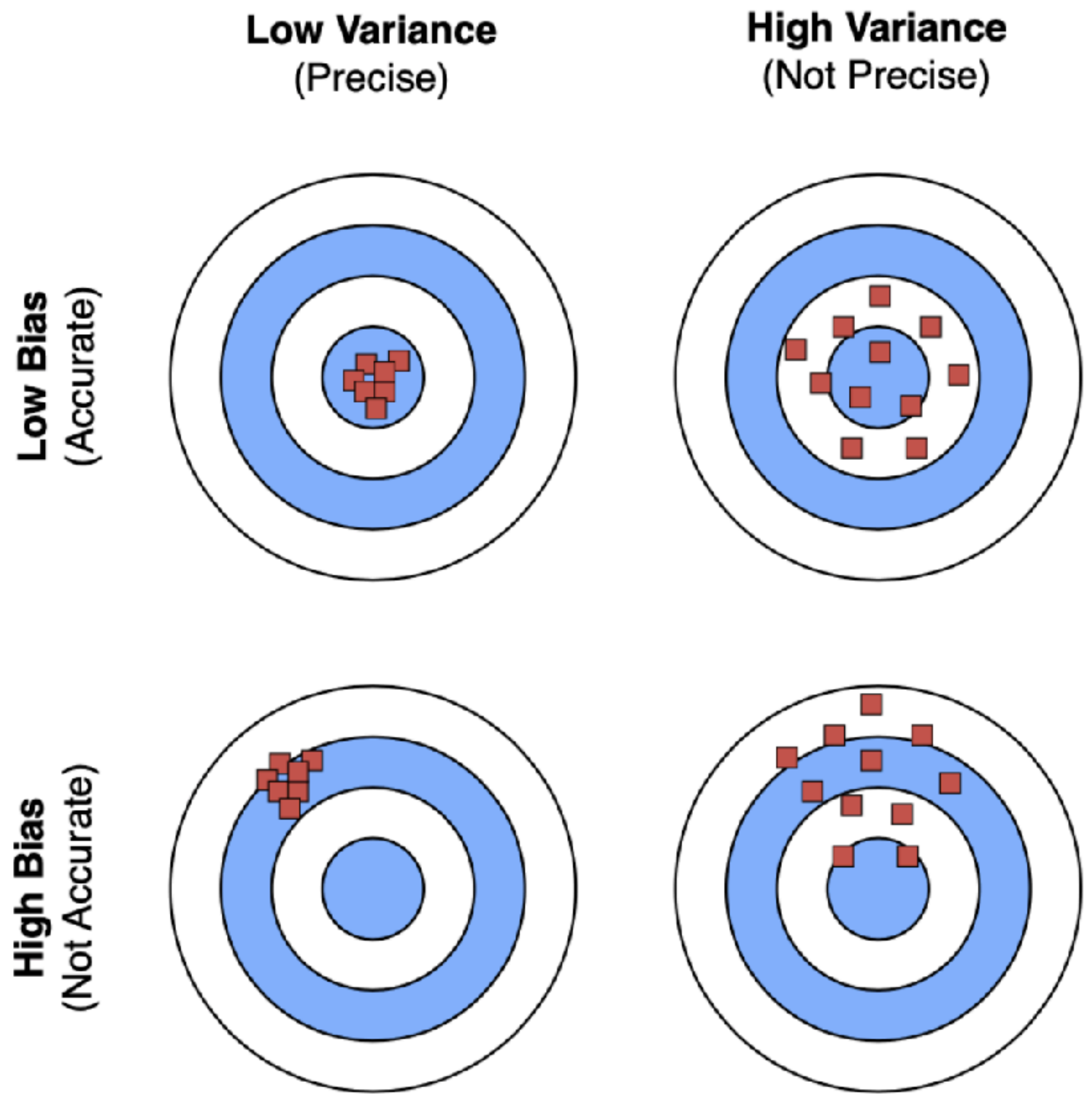
Implementation

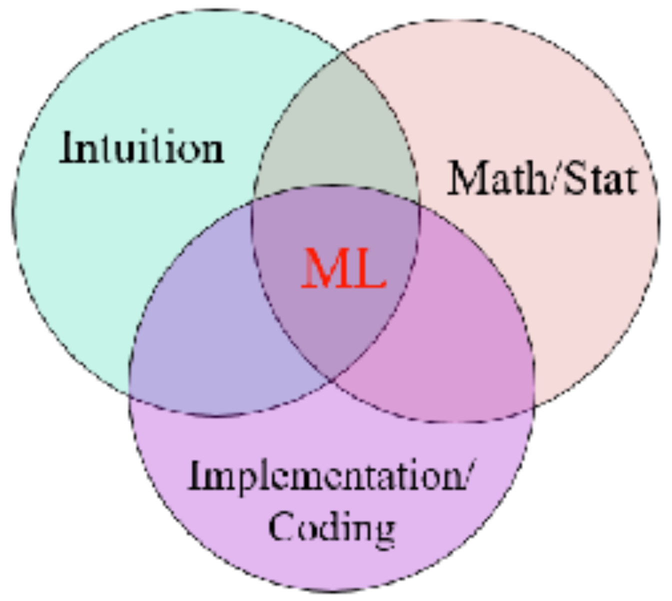
Linear Regression using Ordinary Least Squares

https://colab.research.google.com/github/COGS118A/demo_notebooks/blob/main/lecture_04_linear_regression.ipynb

https://github.com/COGS118A/demo_notebooks.git

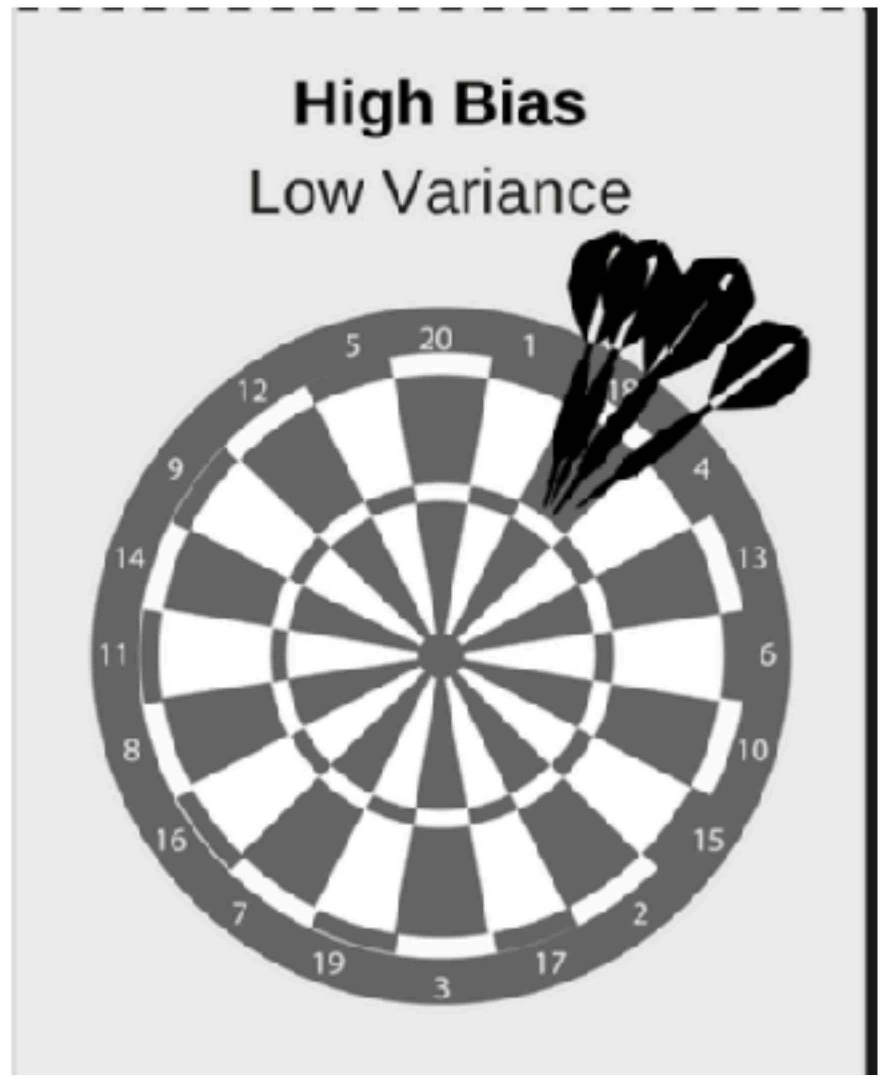
Bias Variance Tradeoff

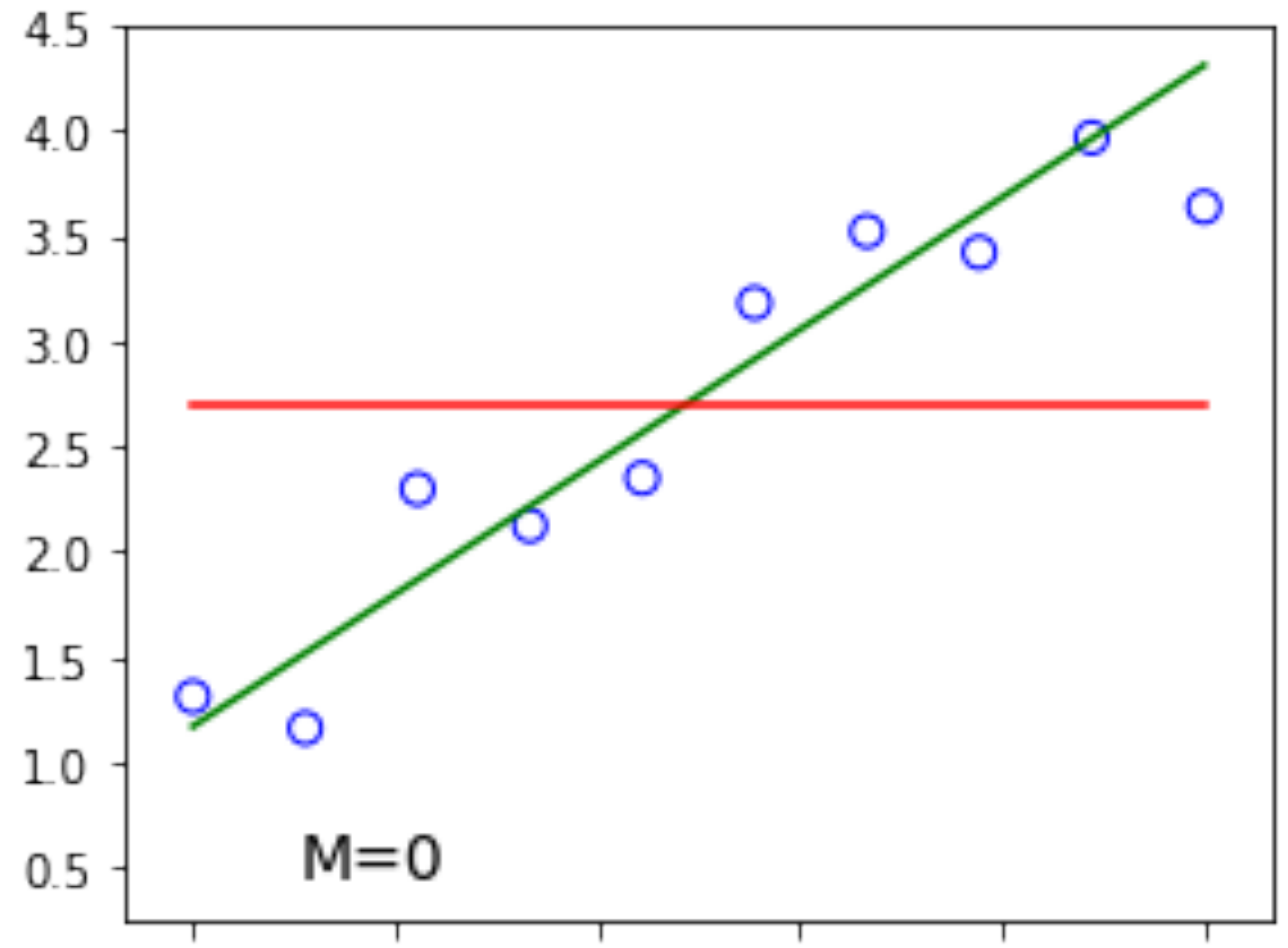




Intuition

Bias-variance tradeoff in model complexity

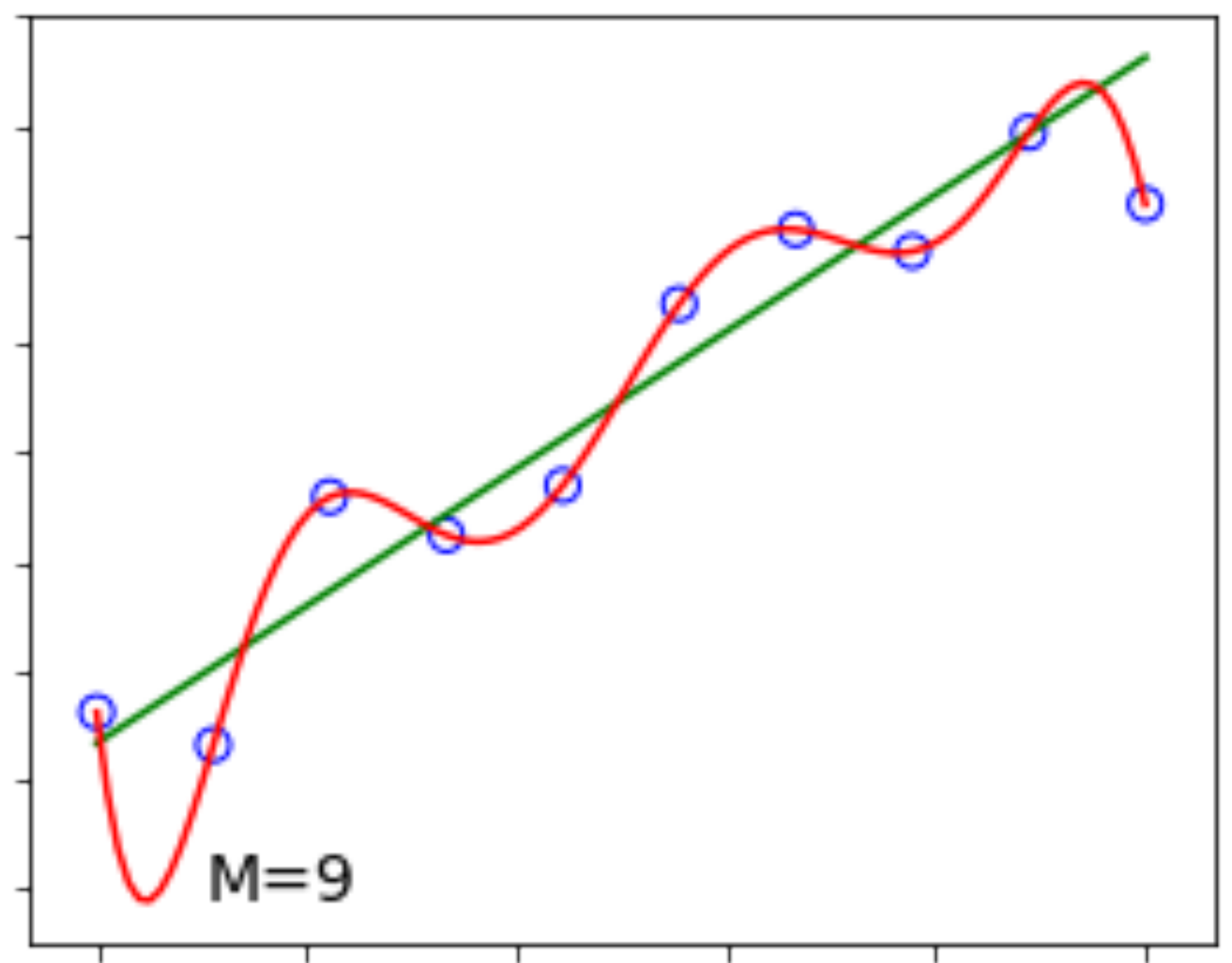




Underfitting

(Model is too simple!)

High Variance
Low Bias

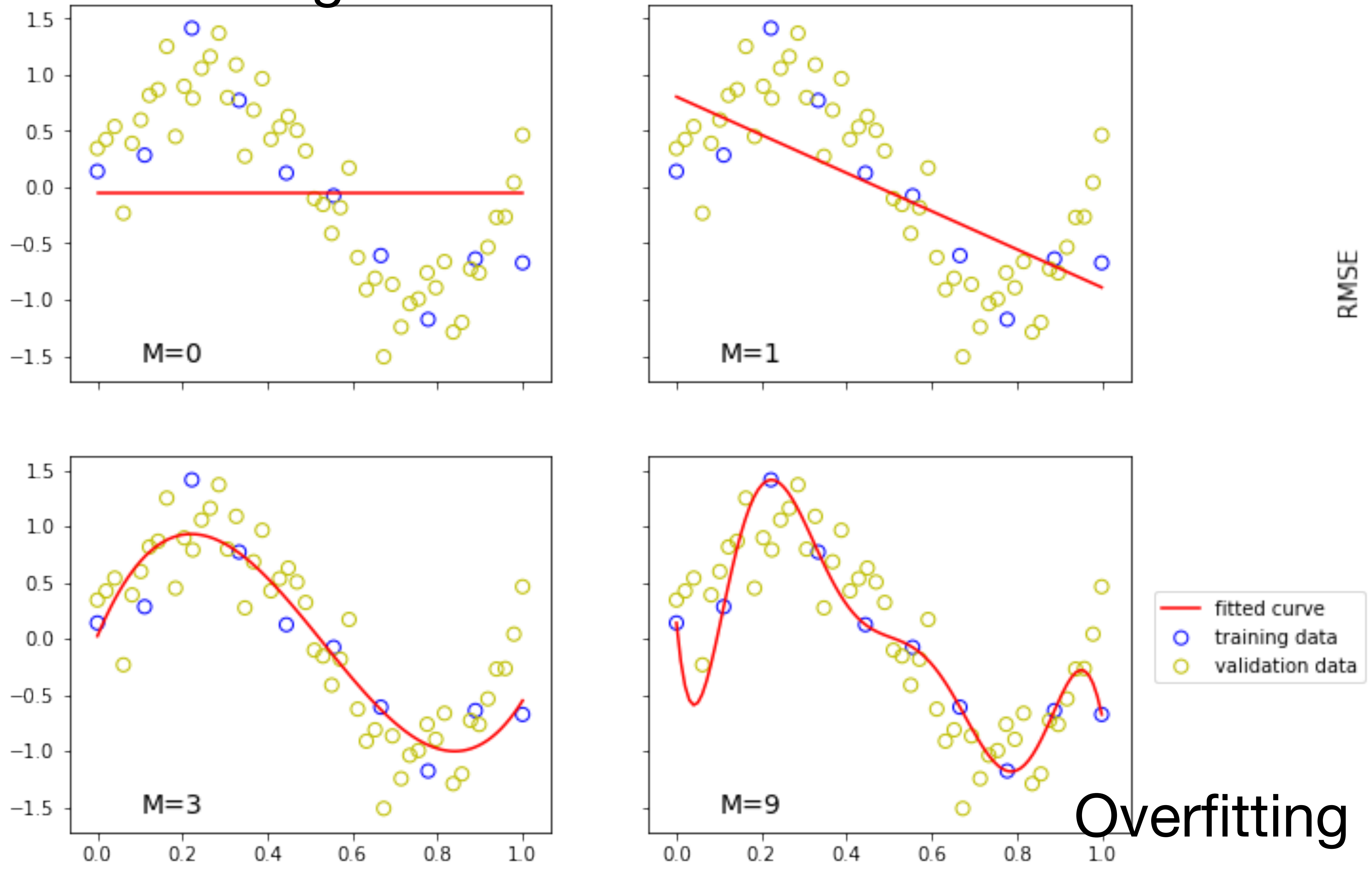


Overfitting

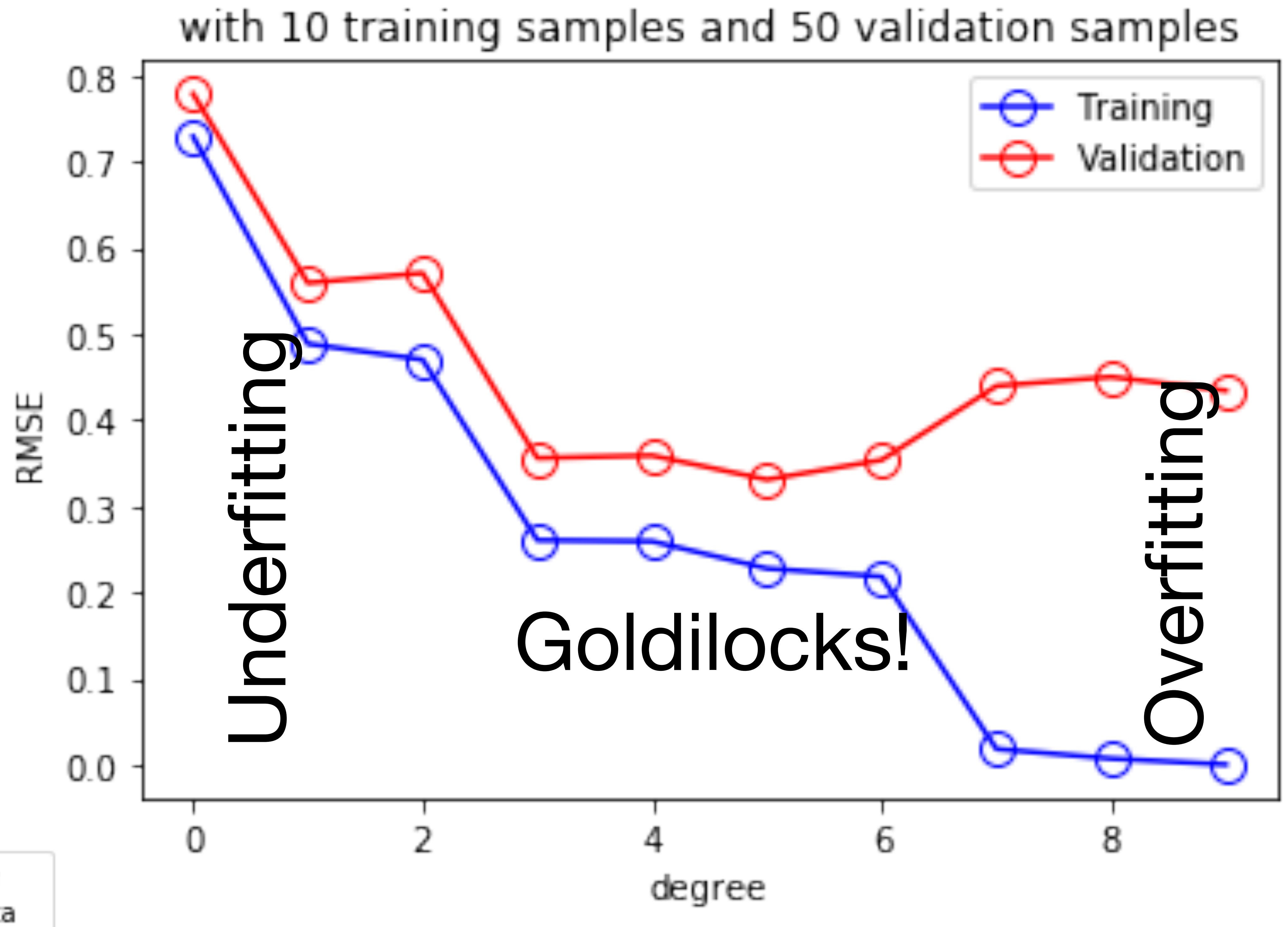
(Model is too complex!)

Training set	-> set the parameters
Validation set	-> try different models, select best
Test set	-> how good is your chosen model

Underfitting



Overfitting

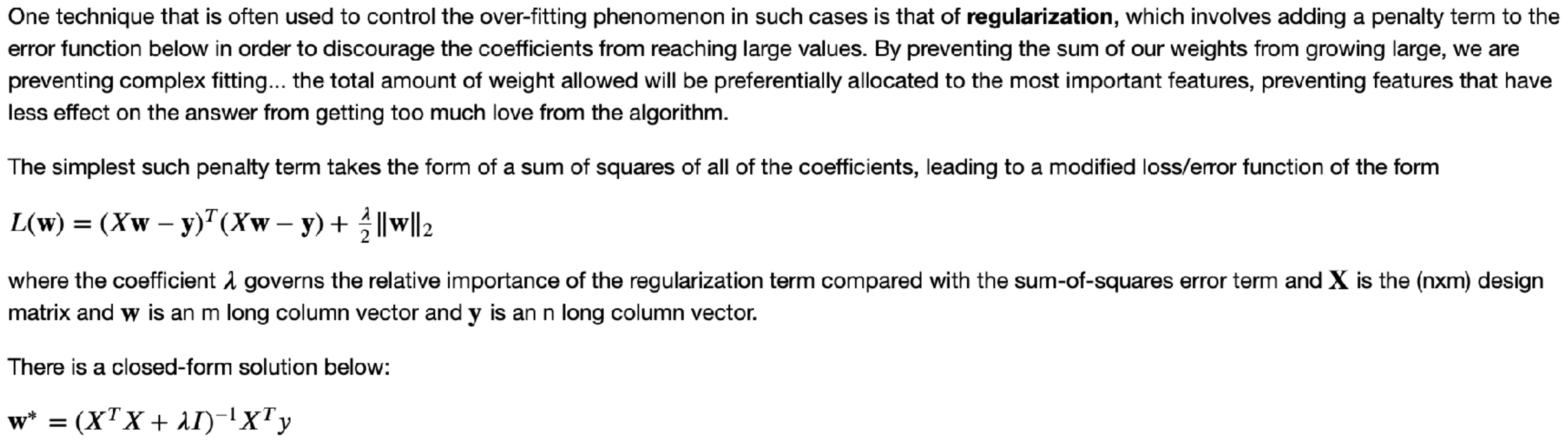


A “validation curve”

https://scikit-learn.org/stable/auto_examples/model_selection/plot_validation_curve.html#plotting-validation-curves

Regularizations reduce overfitting to random noise

Penalize solutions that are too complex



One technique that is often used to control the over-fitting phenomenon in such cases is that of **regularization**, which involves adding a penalty term to the error function below in order to discourage the coefficients from reaching large values. By preventing the sum of our weights from growing large, we are preventing complex fitting... the total amount of weight allowed will be preferentially allocated to the most important features, preventing features that have less effect on the answer from getting too much love from the algorithm.

The simplest such penalty term takes the form of a sum of squares of all of the coefficients, leading to a modified loss/error function of the form

$$L(\mathbf{w}) = (X\mathbf{w} - \mathbf{y})^T(X\mathbf{w} - \mathbf{y}) + \frac{\lambda}{2}\|\mathbf{w}\|_2$$

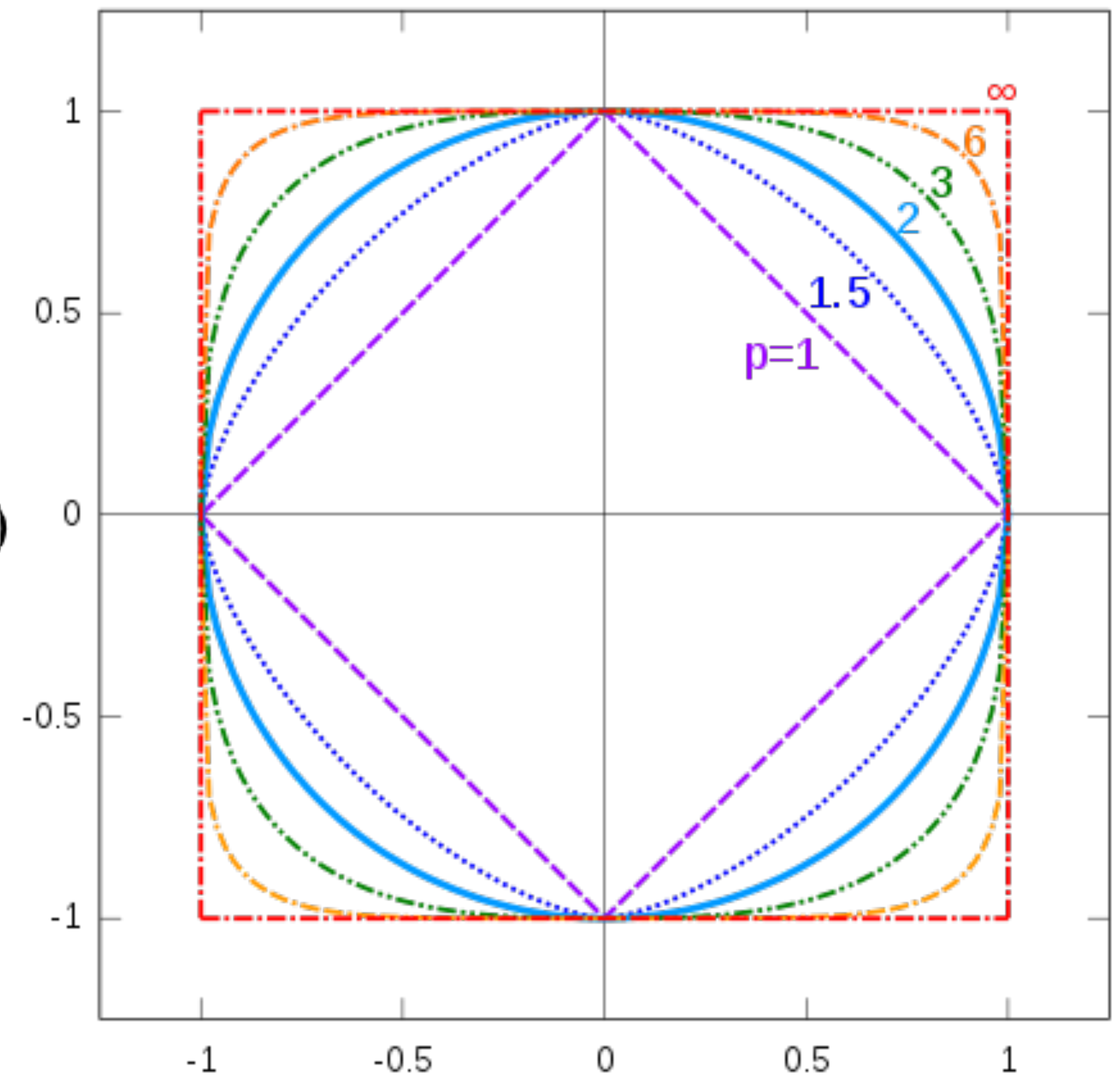
where the coefficient λ governs the relative importance of the regularization term compared with the sum-of-squares error term and $\mathbf{X}$ is the (nxm) design matrix and $\mathbf{w}$ is an m long column vector and $\mathbf{y}$ is an n long column vector.

There is a closed-form solution below:

$$\mathbf{w}^* = (X^T X + \lambda I)^{-1} X^T y$$

Regularizations reduce overfitting to random noise

Penalize solutions that are too complex



You can also do L1 regularization which is often called LASSO regression (least absolute shrinkage and selection operator)

$$L(\mathbf{w}) = (X\mathbf{w} - \mathbf{y})^T(X\mathbf{w} - \mathbf{y}) + \frac{\lambda}{2}\|\mathbf{w}\|_1$$

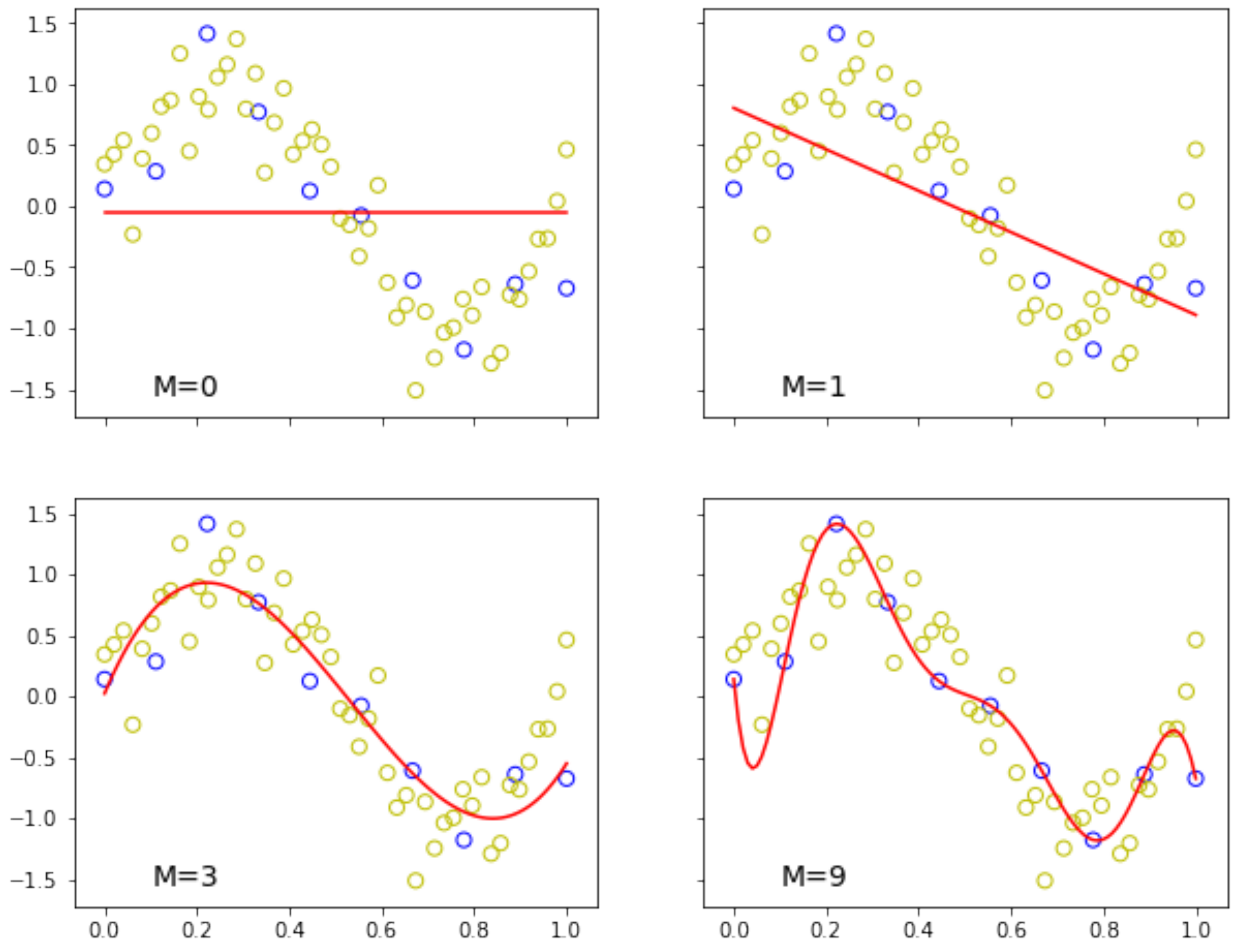
And you can combine the two together in a technique called ElasticNet

$$L(\mathbf{w}) = (X\mathbf{w} - \mathbf{y})^T(X\mathbf{w} - \mathbf{y}) + \frac{\lambda}{2}(\alpha\|\mathbf{w}\|_1 + (1-\alpha)\|\mathbf{w}\|_2)$$

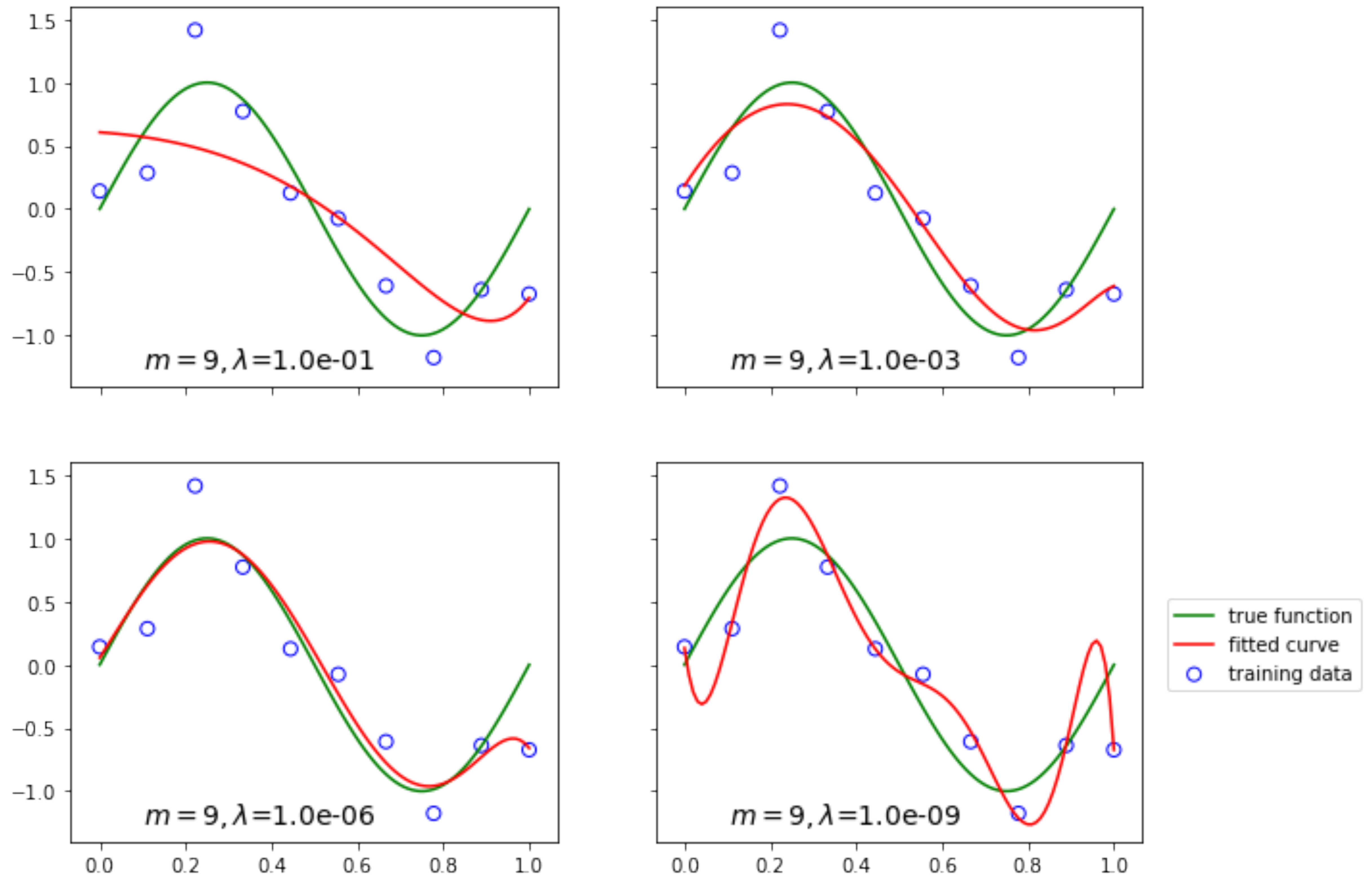
where $\alpha \in [0, 1]$ is a parameter dictating the proportion of L1 to L2 regularization.

Why all these different kinds of regularization? Well L1 tends to produce a *sparse* solution... many weights that are not important are driven towards 0. You use this technique when it seems appropriate to you that less-important factors have no influence on the solution. Whereas L2 limits the total amount of weight evenly, so less-important factors can continue to have less-influence-but-still-some-influence.

One application of L1 regularization: feature selection. Let's say you have data about the expression levels of ~27,000 protein coding genes across a few thousand humans, and you want to dermine if any of the genes have an effect on a disease. Since almost NONE of them will, it's good to use something like a heavy L1 penalty, which will prevent you from picking up too much on random chance associations that may exist in the data.



9th order polynomial regression + L2 regularization



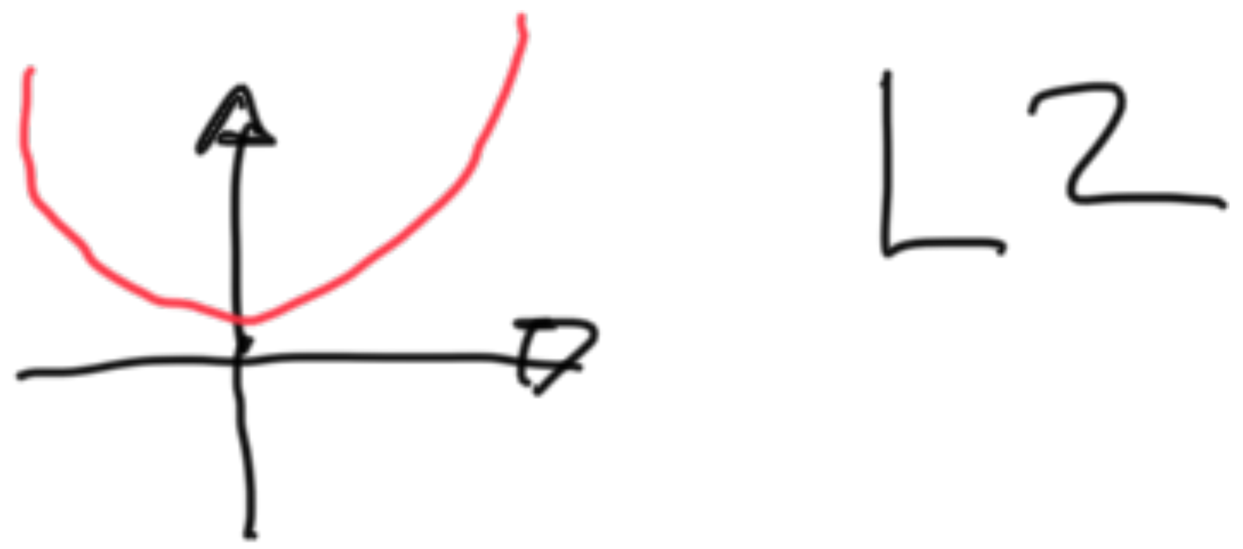
Robust estimation to reject outliers

Estimation and optimization

You can have a different loss function than Residual Sum of Squares!

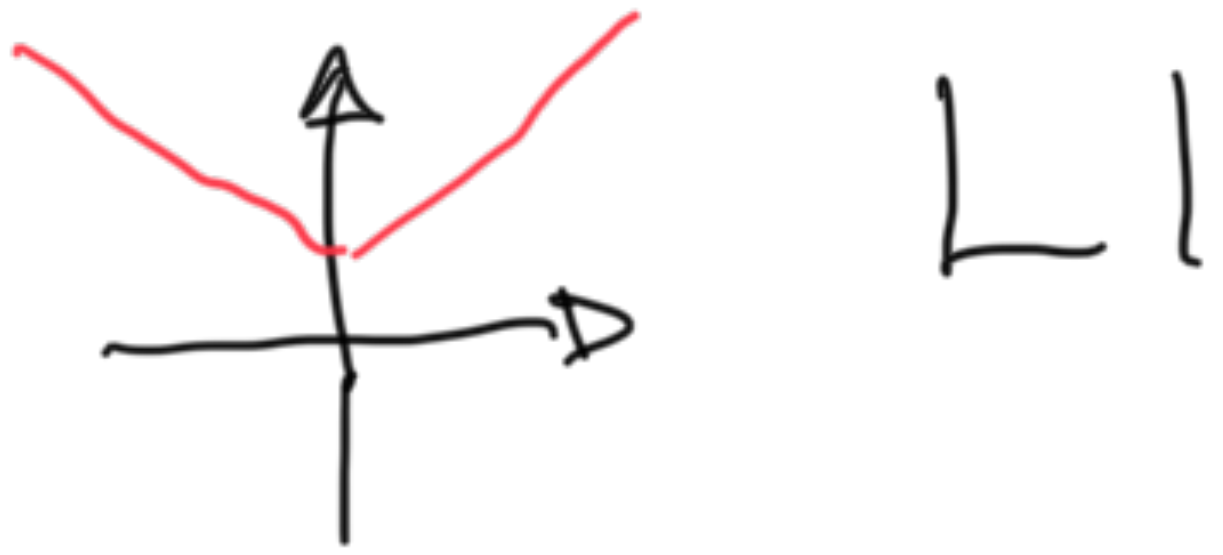
L2 norm: aka RSS!

$$e = \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i; \theta))^2$$



L1 norm:

$$e = \sum_{i=1}^{n} |y_i - f(\mathbf{x}_i; \theta)|$$



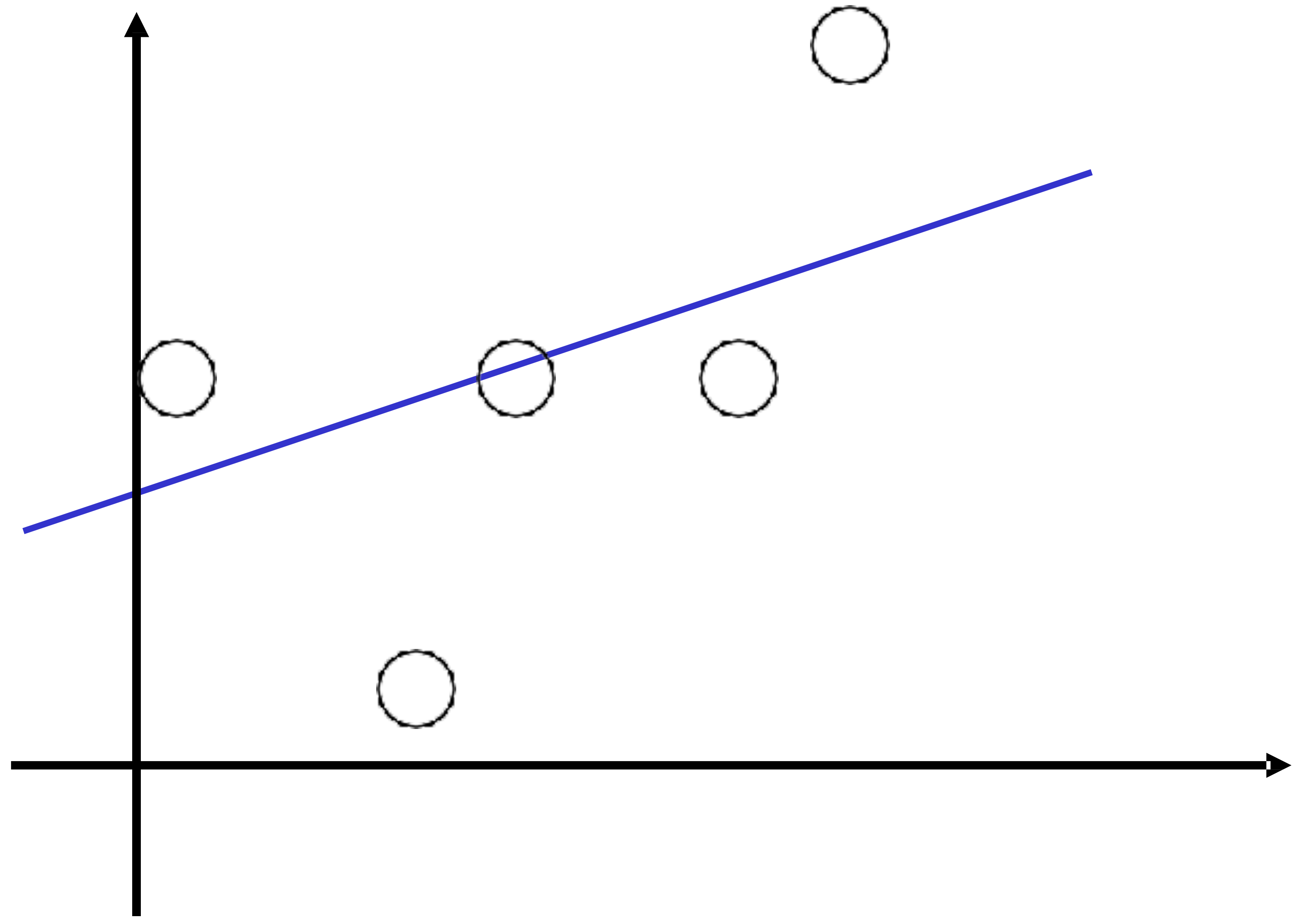
Estimation and optimization

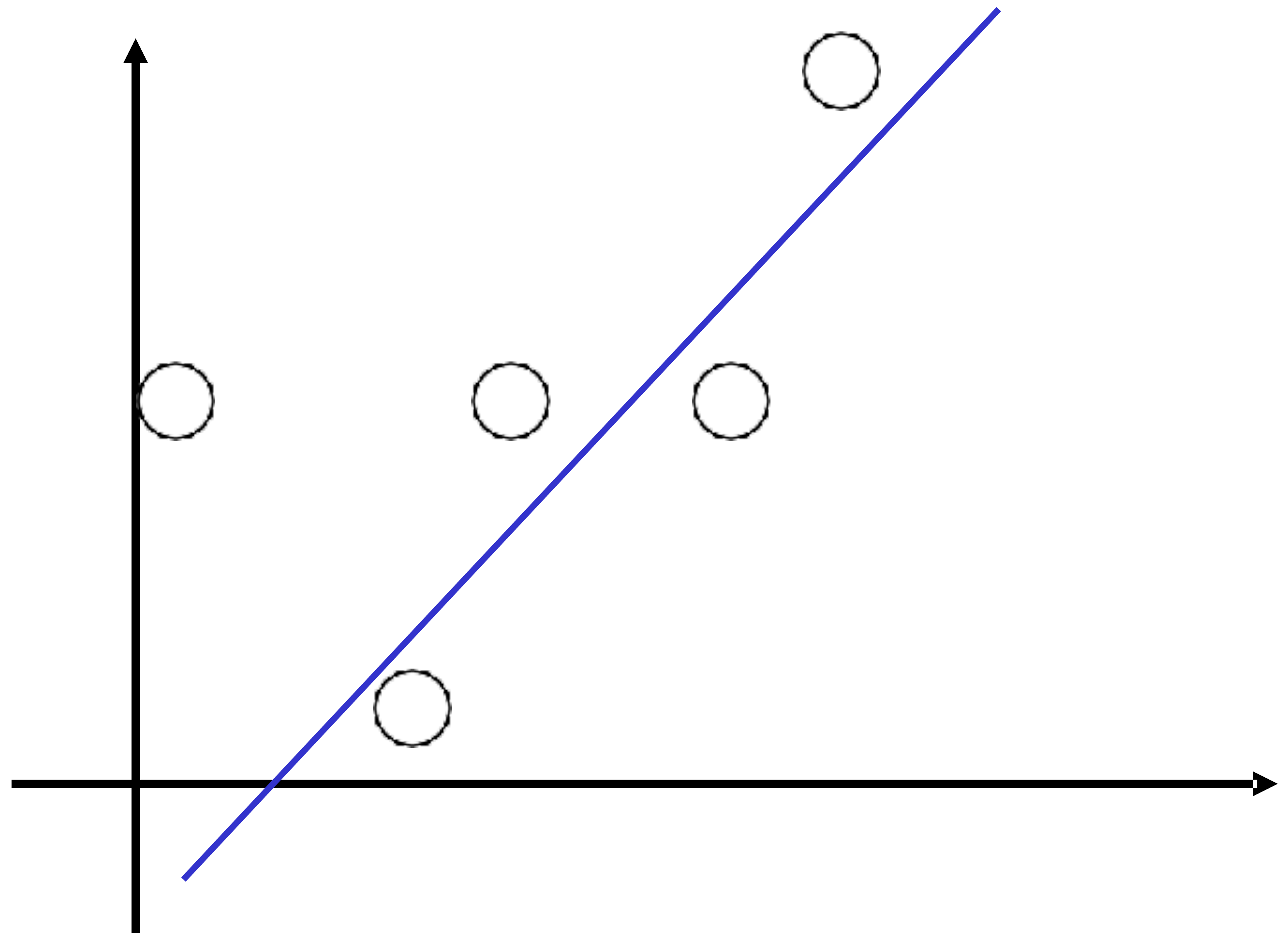
$$S_{training} = \{(\mathbf{x}_i, y_i), i = 1..n\}$$

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \mathscr{L}(\mathbf{w}; \mathbf{x}; \mathbf{y})$$

$$\mathscr{L}_{\mathrm{OLS}}(\mathbf{w}; \mathbf{x}; \mathbf{y}) = \left(\mathbf{y} - f(\mathbf{x}; \mathbf{w})\right)^{\mathrm{T}} \left(\mathbf{y} - f(\mathbf{x}; \mathbf{w})\right)$$
$$= \|\mathbf{y} - f(\mathbf{x}; \mathbf{w})\|_2^2$$

$$\mathscr{L}_{\mathrm{robust}}(\mathbf{w}; \mathbf{x}; \mathbf{y}) = \sum_{i=1}^{n} \left| y_i - f\left(\mathbf{x}_i; \mathbf{w}\right) \right|$$
$$= \|\mathbf{y} - f(\mathbf{x}; \mathbf{w})\|_1$$

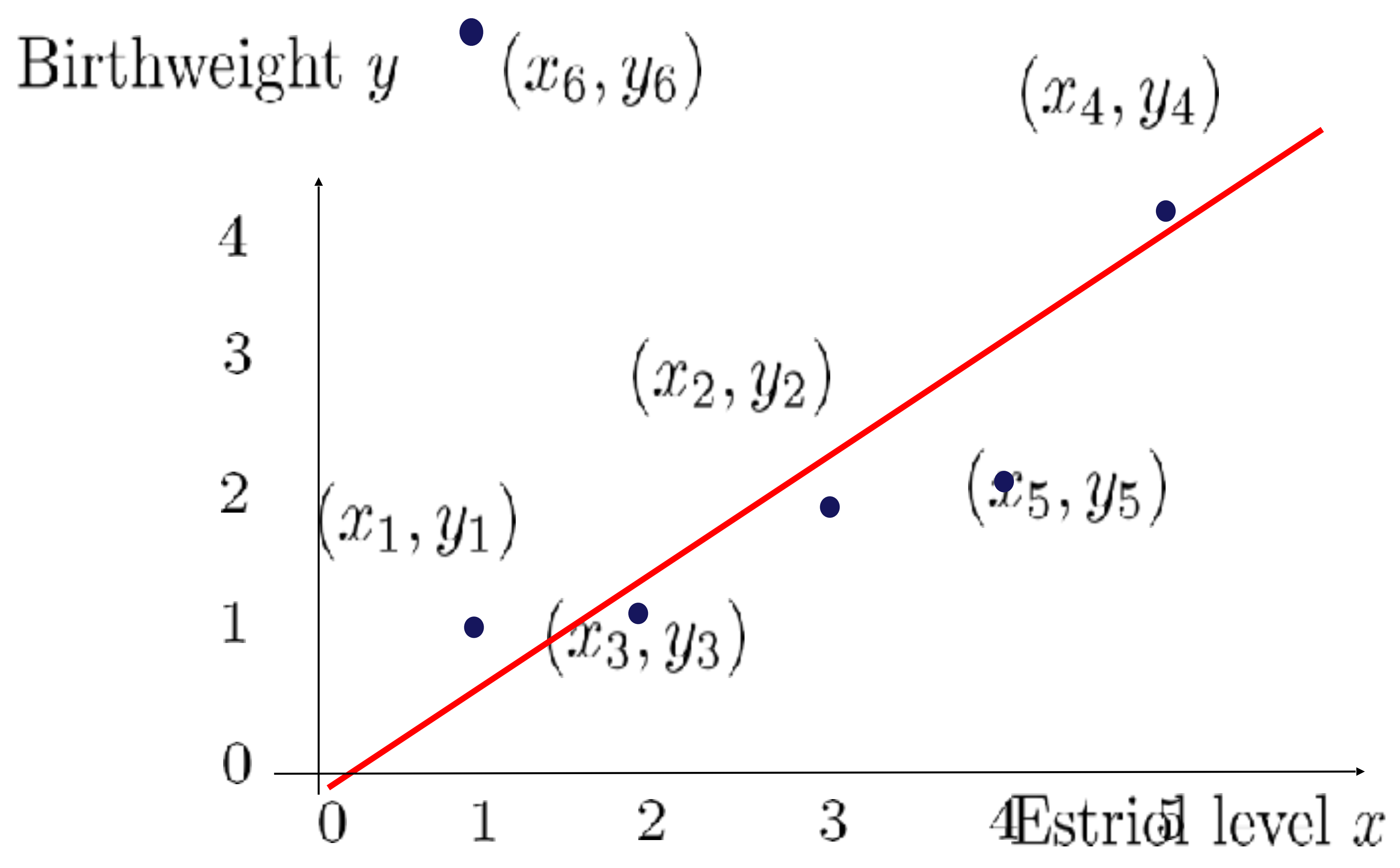




Robust estimation

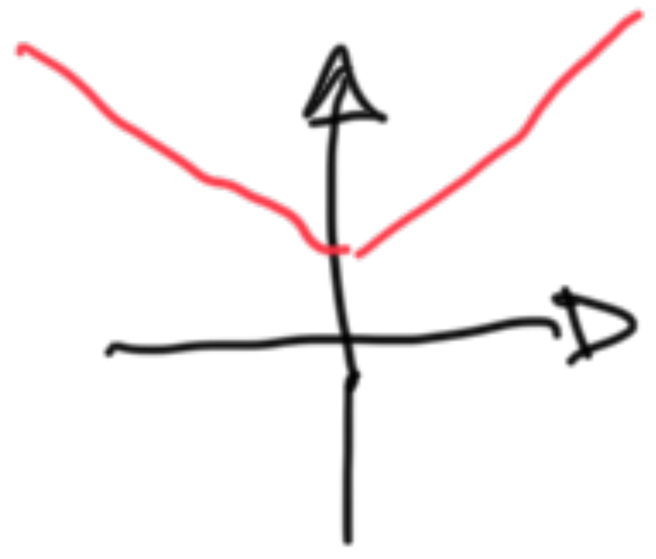
$$Y = X \quad W$$

$$\begin{pmatrix} 1 \\ 1.9 \\ 1.05 \\ 4.1 \\ 2.1 \\ 6.0 \end{pmatrix} = \begin{pmatrix} 1, 1 \\ 1, 3 \\ 1, 2 \\ 1, 5 \\ 1, 4 \\ 1, 1.1 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \end{pmatrix}$$



~~$W^* = (X^T X)^{-1} X^T Y$~~

$$W^* = \arg min_W \sum_i (\mathbf{x}_i^T W - y_i)^2$$



L1

$$W^* = \arg min_W \sum_i |\mathbf{x}_i^T W - y_i|$$

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \mathscr{L}(\mathbf{w}; \mathbf{x}; \mathbf{y})$$

L1

$$S_{training} = \{(x_i, y_i), i = 1..n\}$$

$$\text{E.g. } S_{training} = \{(x_i, y_i), i = 1..n\}$$
$$= \{(1,1), (3,1.9), (2,1.05), (5,4.1), (4,2.1)\}$$

$$Y = \begin{pmatrix} 1 \\ 1.9 \\ 1.05 \\ 4.1 \\ 2.1 \end{pmatrix} \qquad X = \begin{pmatrix} 1,1 \\ 1,3 \\ 1,2 \\ 1,5 \\ 1,4 \end{pmatrix}$$

1. Loss (Cost) Function

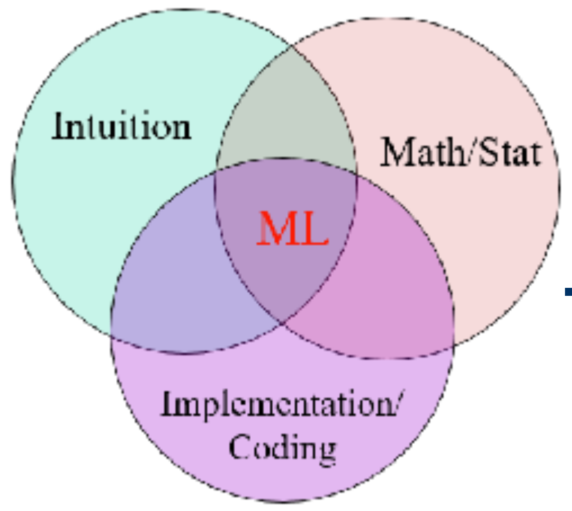
$$\mathscr{L}_{\text{robust}}(\mathbf{y}; \mathbf{x}; \mathbf{w}) = \sum_{i=1}^{n} \left| y_i - f\left(\mathbf{x}_i; \mathbf{w}\right) \right|$$
$$= \|\mathbf{y} - f(\mathbf{x}; \mathbf{w})\|_1$$

2. Obtain the gradient

$$\frac{\partial \mathscr{L}(\mathbf{w})}{\partial \mathbf{w}} = \text{sign}\left(\mathbf{y} - f(\mathbf{x}; \mathbf{w})\right) \mathbf{x}$$

3. Update parameter W

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \lambda_t \frac{\partial \mathscr{L}(\mathbf{w})}{\partial \mathbf{w}}$$



Recap: Robust estimation

Intuition: It's very easy to overfit to either random noise or outliers. Help your ML algorithm reject either form of perturbation by applying robust estimation methods like using an L1 loss function (helps reduce the influence of outliers) or by adding L1 or L2 regularization to any loss function (helps reduce overfitting to noise by penalizing complex and large parameter values)

Math:

L1 loss function
(robust regression)

$$L(\mathbf{w}) = \frac{1}{2}\|(X\mathbf{w} - \mathbf{y})\|_1$$

L1 + L2 regularization
(ElasticNet)

$$L(\mathbf{w}) = \frac{1}{2}(X\mathbf{w} - \mathbf{y})^T(X\mathbf{w} - \mathbf{y}) + \frac{\lambda}{2}(\alpha\|\mathbf{w}\|_1 + (1-\alpha)\|\mathbf{w}\|_2)$$