

Methods 3: Multilevel Statistical Modeling and Machine Learning

Week 7: *Linear regression revisited (machine learning)*

November 9, 2021

by: Lau Møller Andersen

These slides are distributed according to the CC
BY 4.0 licence:

<https://creativecommons.org/licenses/by/4.0/>



Last time – Mid-way evaluation

- 1) Write something you liked about the course so far
- 2) Write something you did not like about the course so far
- 3) What would you change?

Summary

summary of student feedback

LIKE

Good amount of practical exercises (n=3)
Hands-on coding (n=8)
GitHub introduction (n=3)
Lau is answering questions (n=9)
That we go into depth (n=1)
Peer-review (n=1)
Good readings (n=5)
Involvement of students (n=2)
Lectures are interesting (n=2)
Good structure of classes (n=4)
Lau is engaged (n=4)
Exercises tied with papers (n=1)
Allow more time for clarification (n=1)
Easy to get help (n=3)
Pace too slow (n=1)
Explaining lecture goals (n=1)
Demystifying math (n=2)

DID NOT LIKE

Portfolio too challenging or too many questions (n=8)
GitHub is not worth it (n=1)
Math is hard - Explain math more (n=5)
Coding is hard (n=2)
Too many slides, not enough time (n=4)
Questions not clear enough in portfolio (n=9)
Connection between lecture and readings (n=5)
Bad structure of classes (n=1)
Exercises that have not been explicitly covered in class (n=5)
Not clear why we are doing the assignments (n=1)
That Lau is not in all classes (n=2)
Methods 1, 2 and 3 are not well connected (n=2)
Lau talking too fast and not loud enough (n=1)

CHANGE

Closer match between lectures and classes (n=2)
List main concepts in syllabus (n=1)
Better deadlines needed (n=5)
More explanation on portfolios (n=1)
More feedback on exercises (n=3)
Programming together (n=1)
Go through papers that exercises are based on (n=1)

Total n = 29

https://github.com/ualsbombe/github_methods_3/blob/main/week_07/mid-way_evaluation.pdf

Summary – what you liked

- Top 3
 - 1) Lau is answering questions (n=9)
 - 2) Good readings (n=5)
 - 3) Lau is engaged (n=4) Good structure of classes (n=4)

Total n = 29

Summary – what you did not like

- Top 3
 - 1) Questions not clear enough in portfolio (n=9)
 - 2) Portfolio too challenging or too many questions (n=8)
 - 3) Math is hard – explain math more (n=5)
Connection between lecture and readings (n=5)
Exercises not explicitly covered in class (n=5)

Total n = 29

Summary – to change

CHANGE

Closer match between lectures and classes (n=2)

List main concepts in syllabus (n=1)

Better deadlines needed (n=5)

More explanation on portfolios (n=1)

More feedback on exercises (n=3)

Programming together (n=1)

Go through papers that exercises are based on (n=1)

Summary – what I promise to change

I will be very careful and aim at writing questions that are easy to understand

Exercise for you: In tomorrow's exercise: For each question indicate whether you understood what was required of you.

Summary – what I promise to change

Connection between lectures and class will be closer – we will be following the textbook more closely

Summary – a discussion

Lau is answering questions (n=9)

Too many slides, not enough time (n=4)

This is a classic conundrum (which we should still try to solve)

Summary – a discussion

What is the optimal balance between me going through the slides and me answering questions?

Discuss for 3-5 minutes

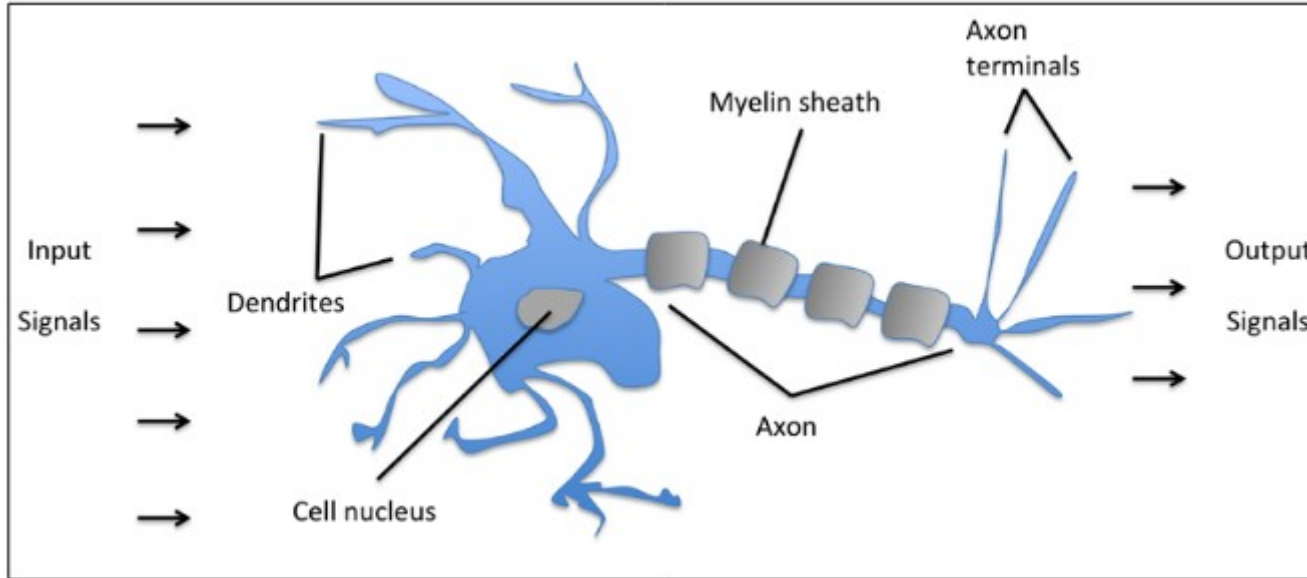
What should our balance be?

Learning goals

Linear regression revisited (machine learning)

- 1) Learning some early classification methods
- 2) Learning how linear regression (with biasing penalties) can be constructed and cross-validated
- 3) Understanding that biasing in-sample solutions can improve out-of-sample predictions

Black box idea



$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

$$z = w_1 x_1 + \dots + w_m x_m$$

Question: what do x , w and z correspond to in the above picture of the *Perceptron*?

(p. 18: Raschka, 2015)

Prediction/classification rule

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

Perceptron fires

Perceptron doesn't fire

θ is a pre-specified threshold

(Raschka, 2015)

Prediction/classification rule

$$w_0 = -\theta$$

$$x_0 = 1$$

$$z = w_0 x_0 + w_1 x_1 + \dots + w_m x_m = \mathbf{w}^T \mathbf{x}$$

$$z = -\theta + w_1 x_1 + \dots + w_m x_m = \mathbf{w}^T \mathbf{x}$$

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Perceptron fires

Perceptron doesn't fire

(Raschka, 2015)

Perceptron classification

Bonus info

$$f(x) = a + \frac{b-a}{1+e^{-\frac{c-x}{d}}}$$

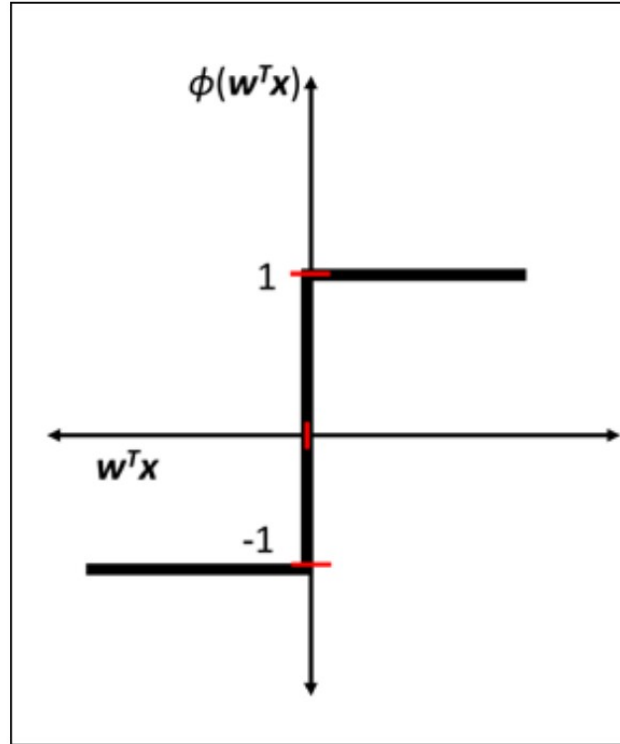
Practical
exercise 5...

$$a = -1$$

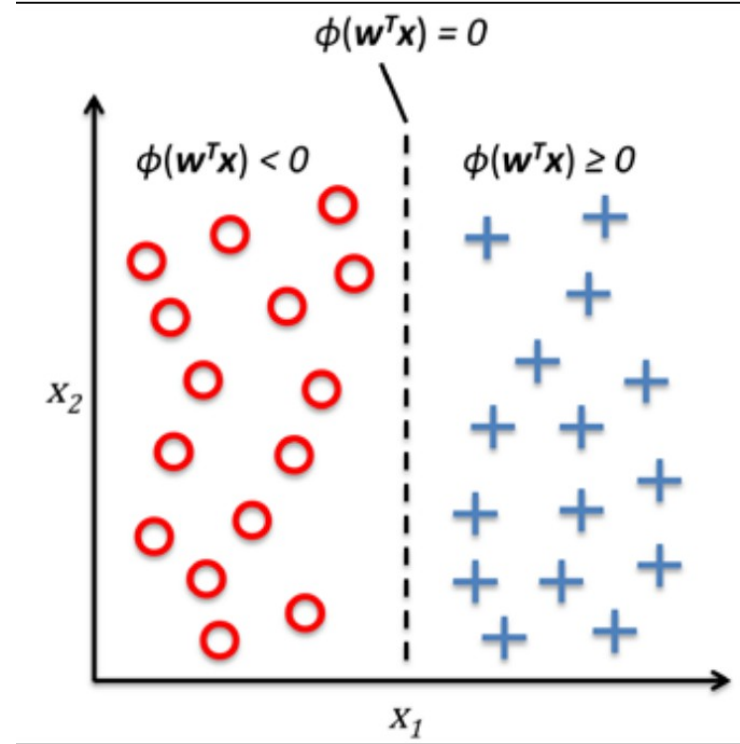
$$b = 1$$

$$c = 0$$

d going to 0



(p. 21: Raschka, 2015)



We want to find $w^T x$ that
achieves this separation

In *Python*

```
class Perceptron(object):  
    """ Perceptron classifier  
  
    Parameters  
    -----  
    eta : float  
        Learning rate (between 0.0 and 1.0)  
    n_iter : int  
        Passes over the training dataset.  
  
    Attributes  
    -----  
    w_ : 1d-array  
        Weights after fitting.  
    errors_ : list  
        Number of misclassifications in every epoch.  
  
    """
```

Special definition that indicates what the object (*Perceptron*) can be initialised with

```
def __init__(self, eta=0.01, n_iter=10):  
    self.eta = eta  
    self.n_iter = n_iter
```

```
ppn = Perceptron(eta=0.1, n_iter=10)
```

Specifying methods of *Perceptron*


1. Initialize the weights to 0 or small random numbers.
2. For each training sample $\mathbf{x}^{(i)}$ perform the following steps:
 1. Compute the output value \hat{y} .
 2. Update the weights.

```
def fit(self, X, y):  
    """ Fit training data.  
  
    Parameters  
    -----  
    X : {array-like}, shape = [n_samples, n_features]  
        Training vectors, where n_samples  
        is the number of samples and  
        n_features is the number of features.  
    y : array-like, shape = [n_samples]  
        Target values.  
  
    Returns  
    -----  
    self : object  
  
    """  
    self.w_ = np.zeros(1 + X.shape[1])  
    self.errors_ = []  
  
    for _ in range(self.n_iter):  
        errors = 0  
        for xi, target in zip(X, y):  
            update = self.eta * (target - self.predict(xi))  
            self.w_[1:] += update * xi  
            self.w_[0] += update  
            errors += int(update != 0.0)  
        self.errors_.append(errors)  
    return self
```

Compute the output value \hat{y}



```
def net_input(self, X):  
    """Calculate net input"""  
    return np.dot(X, self.w_[1:]) + self.w_[0]  
  
def predict(self, X):  
    """Return class label after unit step"""  
    return np.where(self.net_input(X) >= 0.0, 1, -1)
```



$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

When we are right

$$\Delta w_j = \eta \left(\overset{\text{real label}}{\boxed{-1}} - \underset{\text{predicted label}}{\boxed{-1}} \right) x_j^{(i)} = 0$$

$$\Delta w_j = \eta \left(\overset{\text{real label}}{\boxed{1}} - \underset{\text{predicted label}}{\boxed{1}} \right) x_j^{(i)} = 0$$

Δw_j : change in weight

η : learning rate

When we are wrong

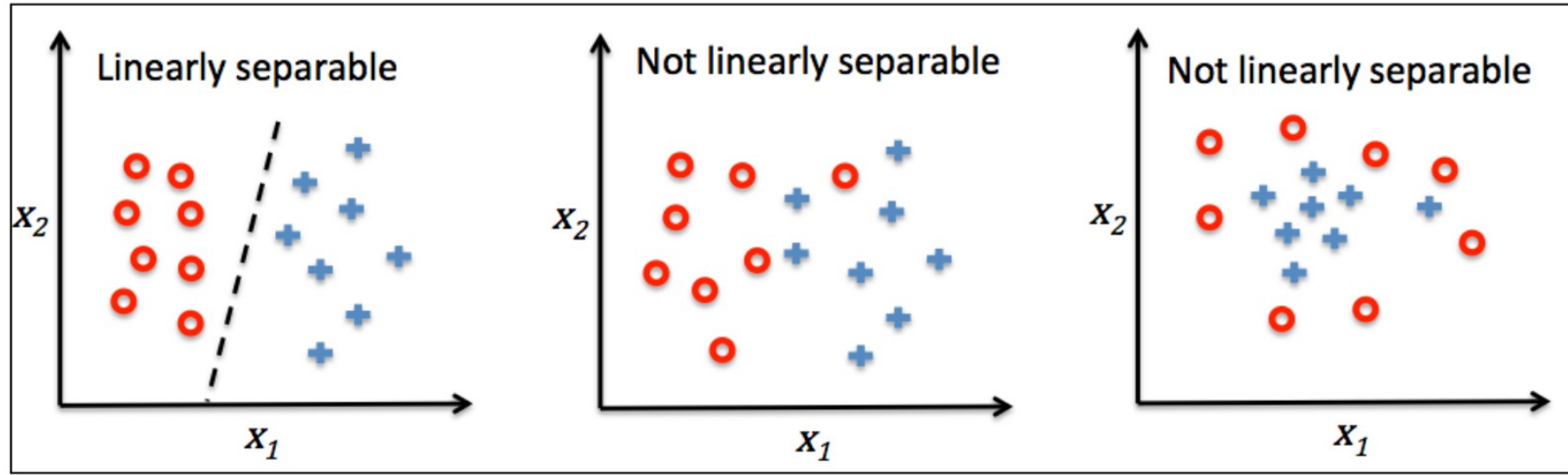
$$\Delta w_j = \eta (\overset{\text{real label}}{\boxed{1}} - \underset{\text{predicted label}}{\boxed{-1}}) x_j^{(i)} = \eta (2) x_j^{(i)}$$

$$\Delta w_j = \eta (\overset{\text{real label}}{\boxed{-1}} - \underset{\text{predicted label}}{\boxed{1}}) x_j^{(i)} = \eta (-2) x_j^{(i)}$$

Δw_j : change in weight

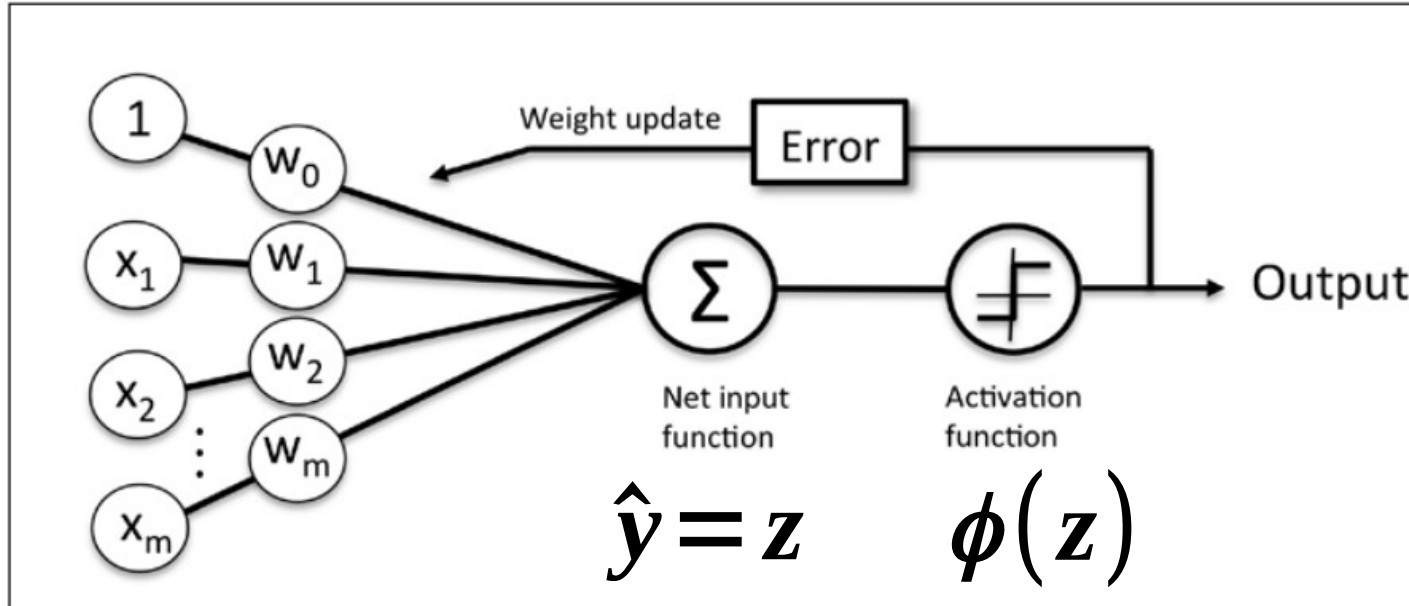
η : learning rate

Convergence only possible when linearly separable



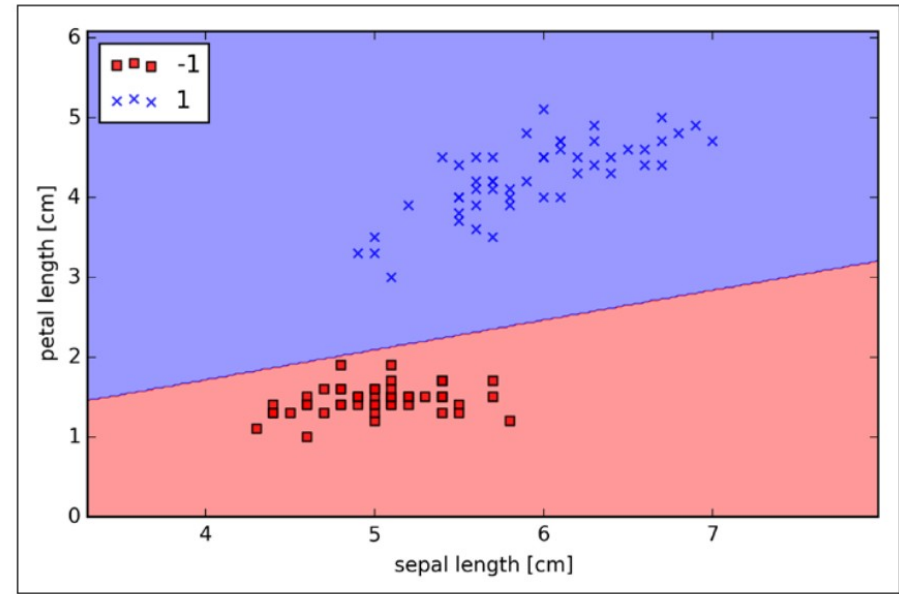
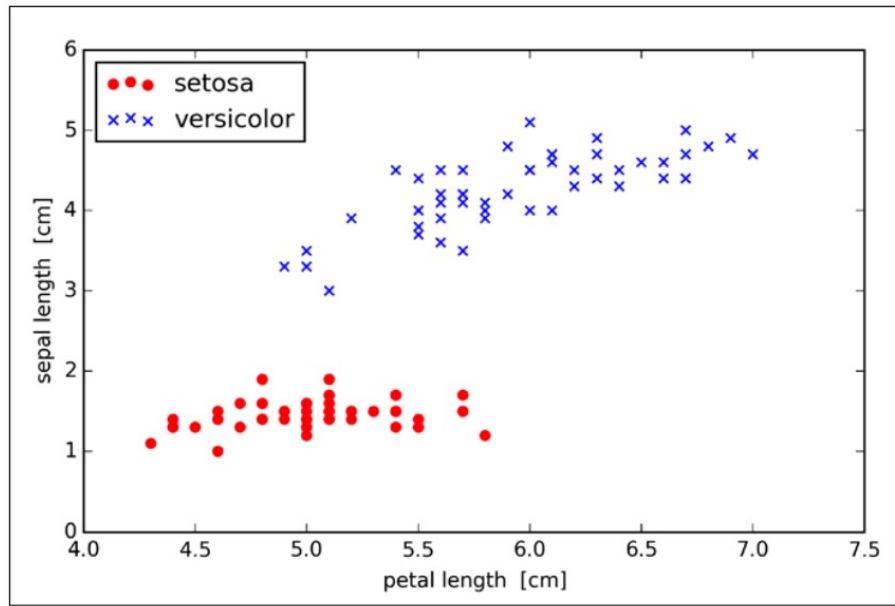
(p. 23: Raschka, 2015)

Perceptron: Graphical summary



(p. 24: Raschka, 2015)

An example



```
In [4]: ppn.w_
Out[4]: array([-0.4 , -0.68,  1.82])
```

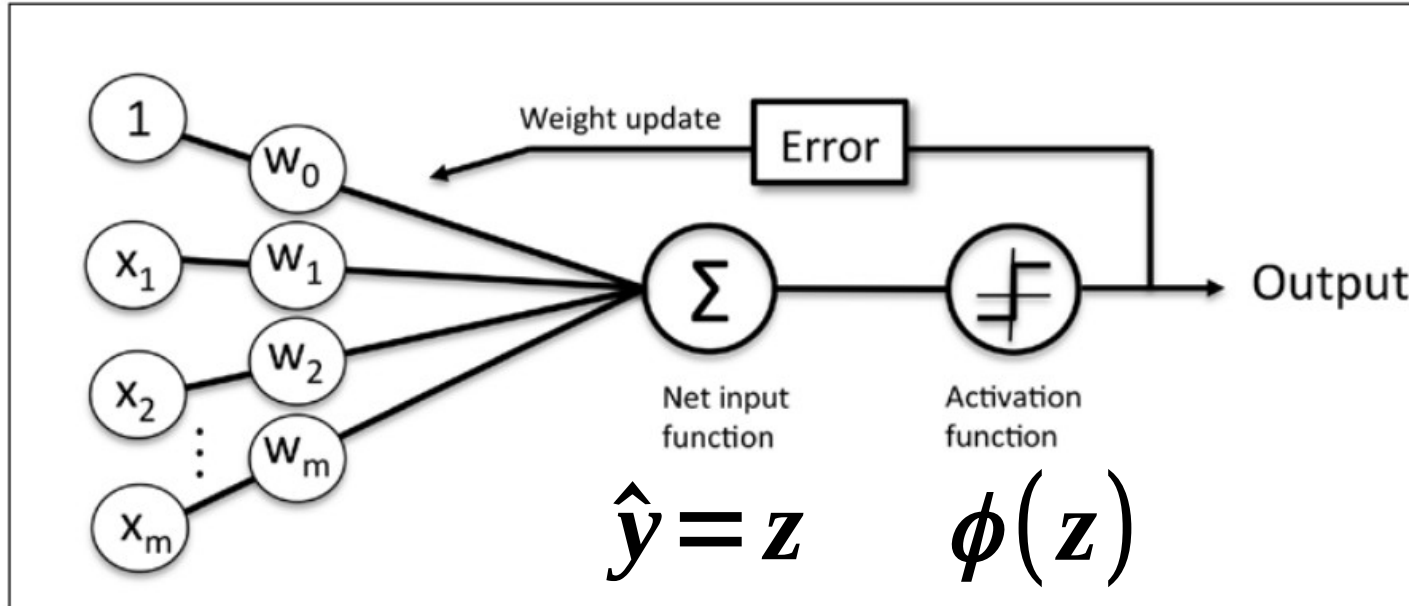
```
In [12]: X[1, :]
Out[12]: array([4.9, 1.4])
```

$$\hat{y} = -0.4 - 0.68 \cdot 4.9 + 1.82 \cdot 1.4 = -1.184$$

```
In [11]: ppn.net_input(X[1, :])
Out[11]: -1.1839999999999975
```

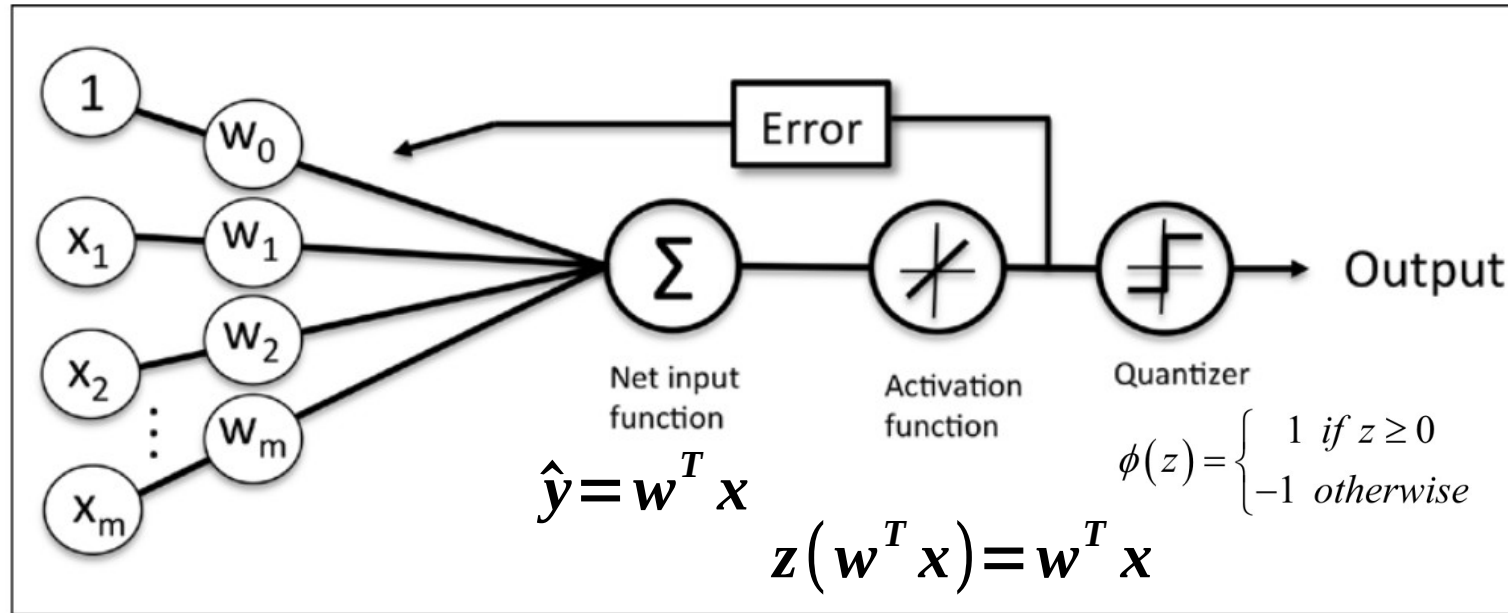
(p. 29 & p. 32: Raschka, 2015)

Perceptron: Graphical summary



(p. 24: Raschka, 2015)

ADaptive Linear NEuron (ADALINE)



$$w^T x = w_0 x_0 + w_1 x_1 + \dots + w_{m-1} x_{m-1} + w_m x_m$$

(p. 33: Raschka, 2015)

ADALINE Gradient descent

```
def __init__(self, eta=0.01, n_iter=50):  
    self.eta = eta  
    self.n_iter = n_iter
```

```
class AdalineGD(object):  
    """ ADaptive Linear NEuron classifier  
  
    Parameters  
    -----  
    eta : float  
        Learning rate (between 0.0 and 1.0)  
    n_iter : int  
        Passes over the training dataset.  
  
    Attributes  
    -----  
    w_ : 1d-array  
        Weights after fitting.  
    errors_ : list  
        Number of misclassifications in every epoch.  
  
    """
```

Methods

$$\hat{y} = w^T x$$

$$z(w^T x) = w^T x$$

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

```
def net_input(self, X):  
    """Calculate net input"""  
    return np.dot(X, self.w_[1:]) + self.w_[0]  
  
def activation(self, X):  
    """Computer linear activation"""  
    return self.net_input(X)  
  
def predict(self, X):  
    """Return class label after unit step"""  
    return np.where(self.activation(X) >= 0.0, 1, -1)
```

The *fit* method

eta: learning rate (a constant)

output: $X\beta = \hat{y}$

errors: $y - \hat{y}$

$X.T.dot(errors)$: $X^T \cdot (y - \hat{y})$

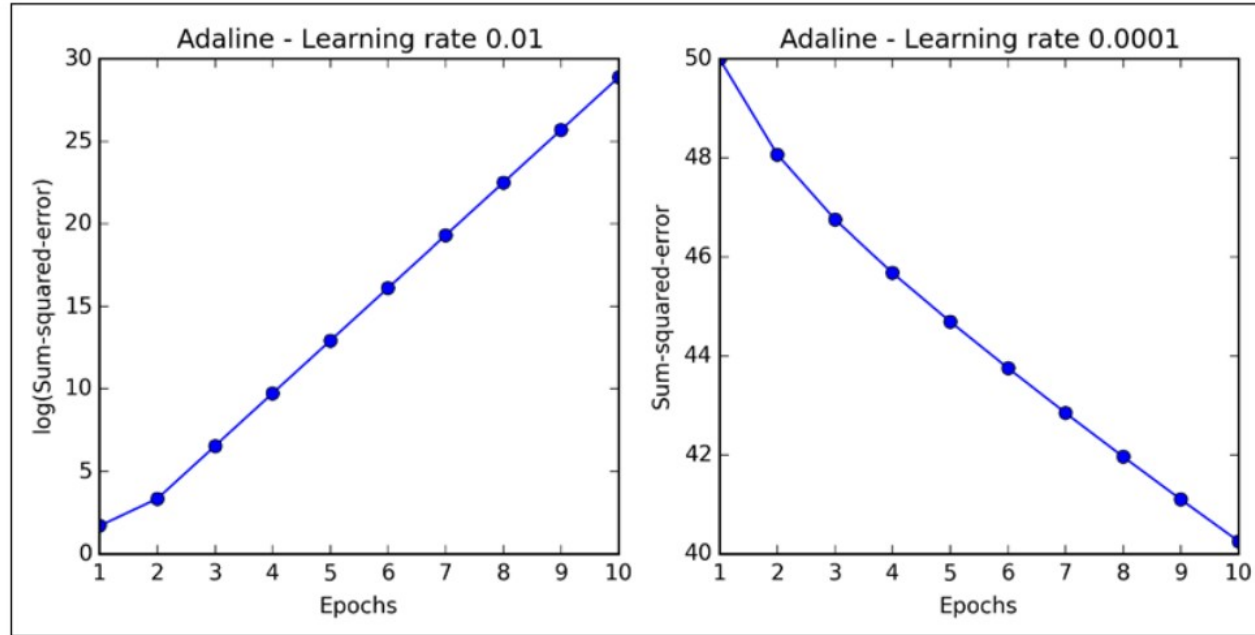
$X^T \cdot (y - \hat{y}) = \Delta w_1 + \Delta w_2 + \dots + \Delta w_{m-1} + \Delta w_m$

cost function: $(\sum (y - \hat{y})^2) / 2$

```
def fit(self, X, y):  
    """ Fit training data.  
  
    Parameters  
    -----  
    X : {array-like}, shape = [n_samples, n_features]  
        Training vectors, where n_samples  
        is the number of samples and  
        n_features is the number of features.  
    y : array-like, shape = [n_samples]  
        Target values.  
  
    Returns  
    -----  
    self : object  
  
    """  
    self.w_ = np.zeros(1 + X.shape[1])  
    self.cost_ = []  
  
    for i in range(self.n_iter):  
        output = self.net_input(X)  
        errors = (y - output)  
        self.w_[1:] += self.eta * X.T.dot(errors)  
        self.w_[0] += self.eta * errors.sum()  
        cost = (errors**2).sum() / 2.0  
        self.cost_.append(cost)  
    return self
```


Learning rate

Overshooting
the global
minimum

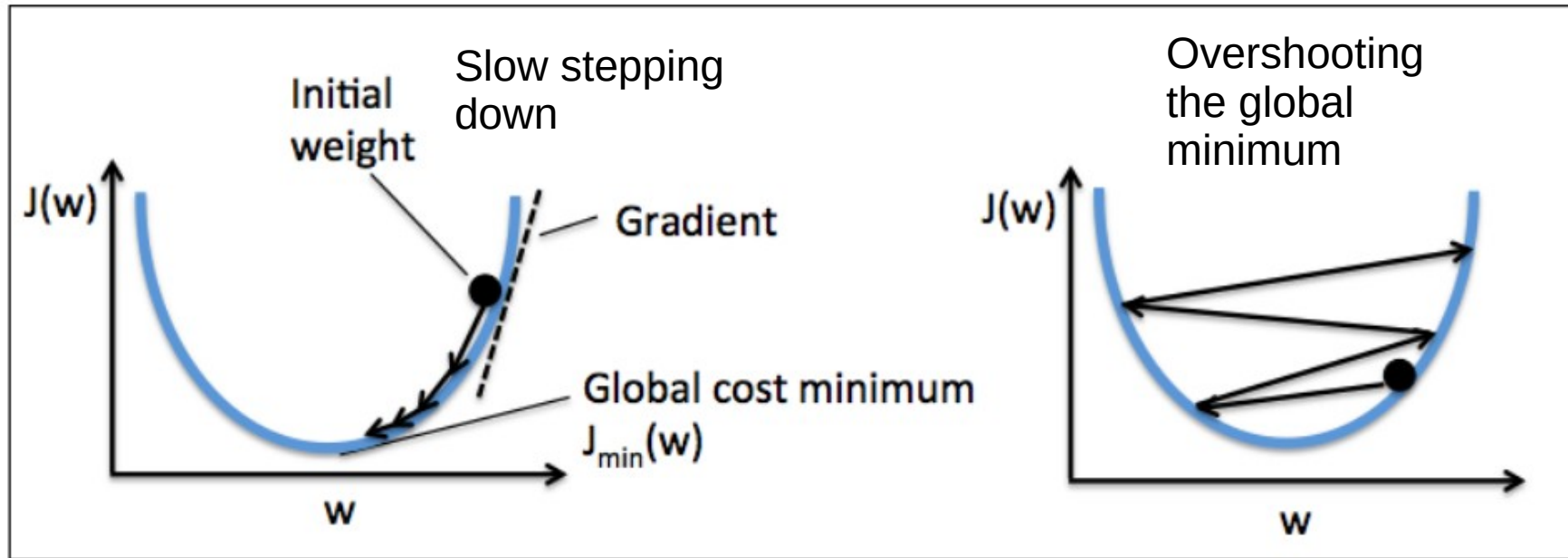


Slow stepping
down

Always check whether it converges

(p. 40: Raschka, 2015)

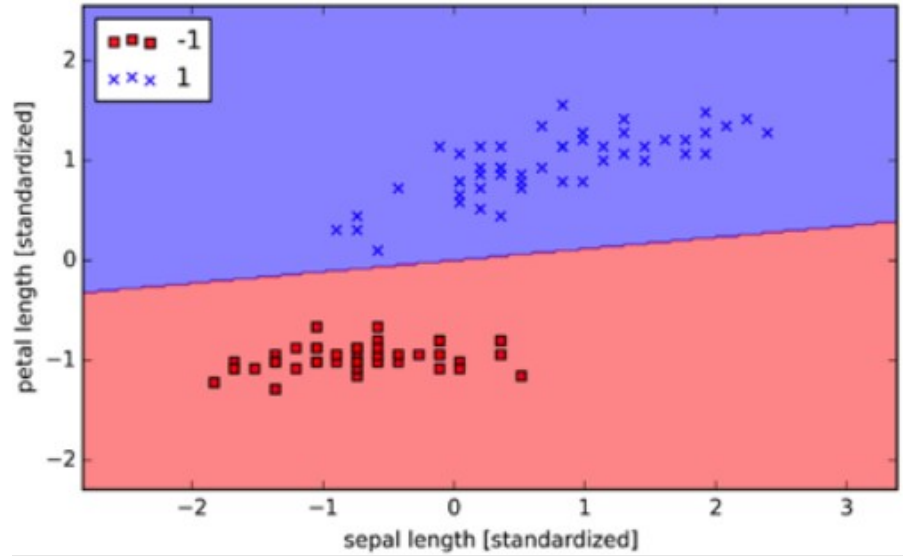
Gradient descent



$$J(w) = (\sum (y - \hat{y})^2) / 2$$

(p. 40: Raschka, 2015)

Adaline - Gradient Descent



(p. 42: Raschka, 2015)

```
In [50]: X_std[1, :]
Out[50]: array([-0.89430898, -1.01435952])
```

```
In [38]: ada.w_
Out[38]: array([ 1.59872116e-16, -1.26256159e-01,  1.10479201e+00])
```

$$\hat{y} \approx 0 - 0.127 \cdot (-0.894) + 1.10 \cdot (-1.014) = -1.01$$

```
In [49]: ada.net_input(X_std[1, :])
Out[49]: -1.0077442758158444
```

How does this relate to linear regression?

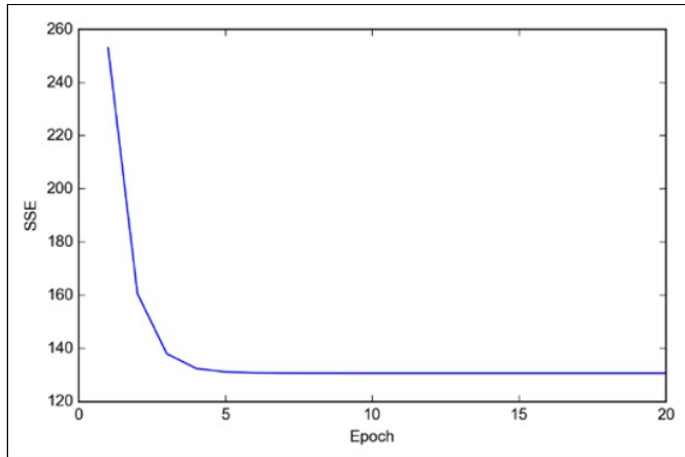
From ADALINE

$$\mathbf{w}^T \mathbf{x} = w_0 x_0 + w_1 x_1 + \dots + w_{m-1} x_{m-1} + w_m x_m$$

Very similar to ADALINE
and when converged will be virtually
identical to the ordinary least
squares solution

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

Convergence

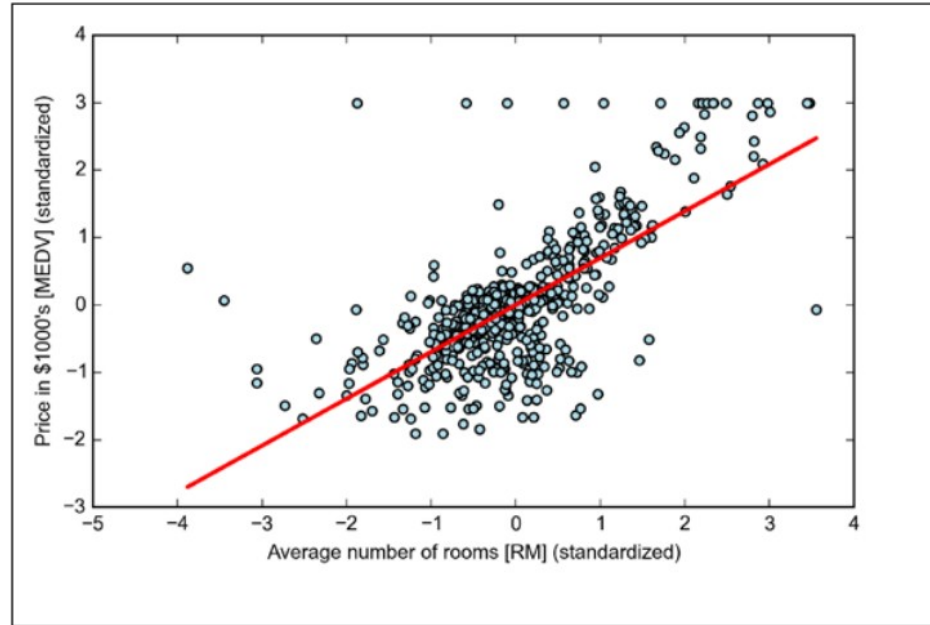


```
class LinearRegressionGD(object):  
  
    def __init__(self, eta=0.001, n_iter=20):  
        self.eta = eta  
        self.n_iter = n_iter  
  
    def fit(self, X, y):  
        self.w_ = np.zeros(1 + X.shape[1])  
        self.cost_ = []  
  
        for i in range(self.n_iter):  
            output = self.net_input(X)  
            errors = (y - output)  
            self.w_[1:] += self.eta * X.T.dot(errors)  
            self.w_[0] += self.eta * errors.sum()  
            cost = (errors**2).sum() / 2.0  
            self.cost_.append(cost)  
        return self  
  
    def net_input(self, X):  
        return np.dot(X, self.w_[1:]) + self.w_[0]  
  
    def predict(self, X):  
        return self.net_input(X)
```

What we have seen before

```
lr = LinearRegressionGD()  
lr.fit(X_std, y_std)
```

(p. 288: Raschka, 2015)



```
In [8]: lr.w_  
Out[8]: array([-4.68958206e-16,  6.95359426e-01])
```

Intercept and slope



Olivier Grisel
@ogrisel

In scikit-learn 1.0, we decided to deprecate the `sklearn.datasets.load_boston` function because the design of this dataset casually assumes that people prefer to buy housing in racially segregated neighborhoods.

(p. 229: Raschka, 2015)

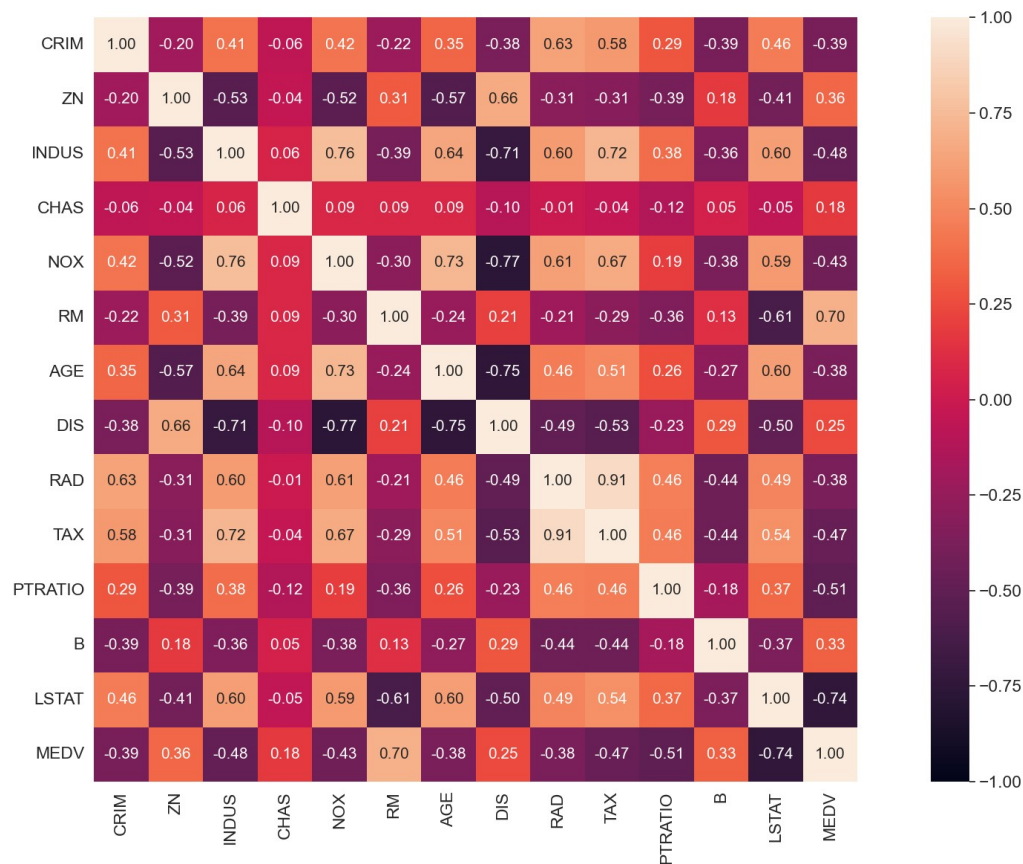
The features of the 506 samples may be summarized as shown in the excerpt of the dataset description:

- **CRIM:** This is the per capita crime rate by town
- **ZN:** This is the proportion of residential land zoned for lots larger than 25,000 sq.ft.
- **INDUS:** This is the proportion of non-retail business acres per town
- **CHAS:** This is the Charles River dummy variable (this is equal to 1 if tract bounds river; 0 otherwise)
- **NOX:** This is the nitric oxides concentration (parts per 10 million)
- **RM:** This is the average number of rooms per dwelling
- **AGE:** This is the proportion of owner-occupied units built prior to 1940
- **DIS:** This is the weighted distances to five Boston employment centers
- **RAD:** This is the index of accessibility to radial highways
- **TAX:** This is the full-value property-tax rate per \$10,000
- **PTRATIO:** This is the pupil-teacher ratio by town
- **B:** This is calculated as $1000(B_k - 0.63)^2$, where B_k is the proportion of people of African American descent by town
- **LSTAT:** This is the percentage lower status of the population
- **MEDV:** This is the median value of owner-occupied homes in \$1000s

Multiple linear regression

$$\begin{aligned} \hat{MEDV} = & w_0 x_0 + w_1 CRIM + w_2 ZN + w_3 INDUS + w_4 CHAS \\ & + w_5 NOX + w_6 RM + w_7 AGE + w_8 DIS \\ & + w_9 RAD + w_{10} TAX + w_{11} PTRATIO + w_{12} B + w_{13} LSTAT + \epsilon \end{aligned}$$

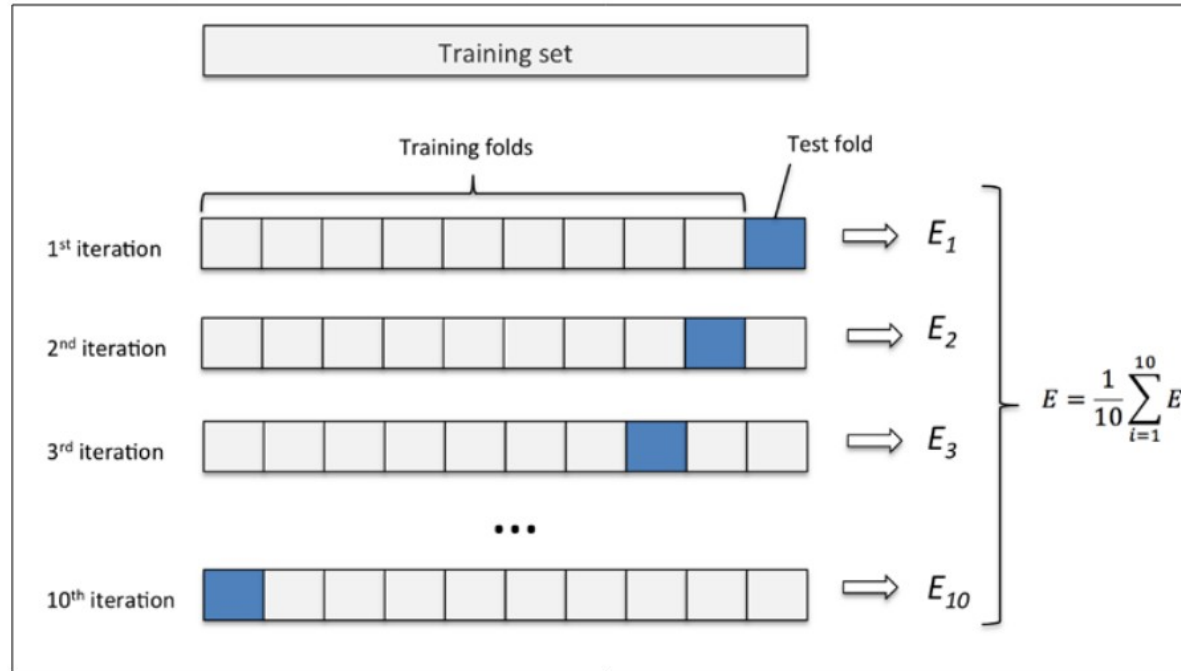
Collinearity – correlation matrix



Because of the collinearity, we know we are prone to overfitting, so we do **out-of-sample** prediction instead of validating our model with traditional measures like R^2 and maximum likelihood

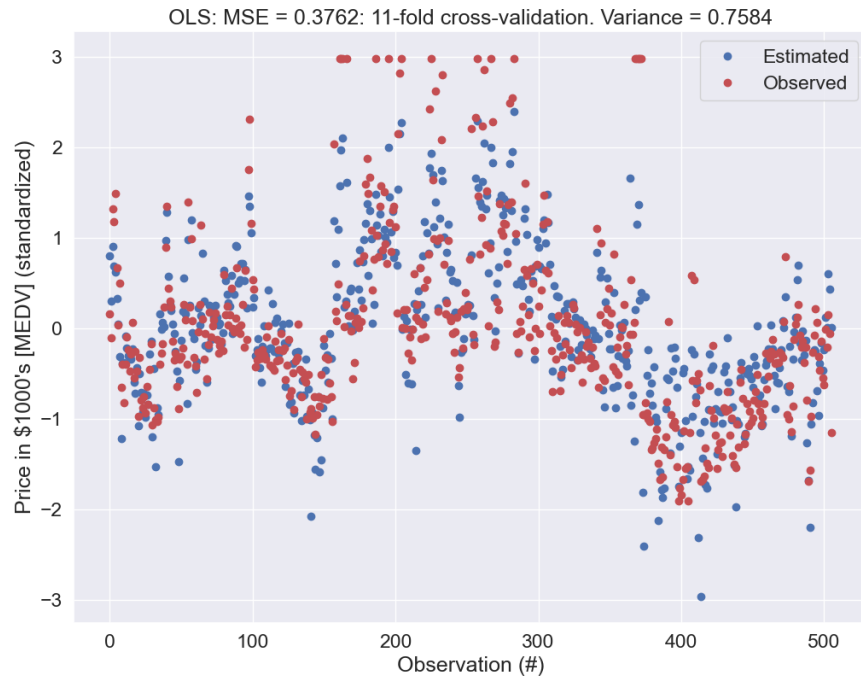
How to choose the **out-of-sample** dataset?

Cross-validation



(p. 176: Raschka, 2015)

```
OLS = LinearRegression()  
OLS.fit(X_std, y_std)  
  
MSE = np.mean(cross_validate(OLS, X_std, y_std, k=11))
```

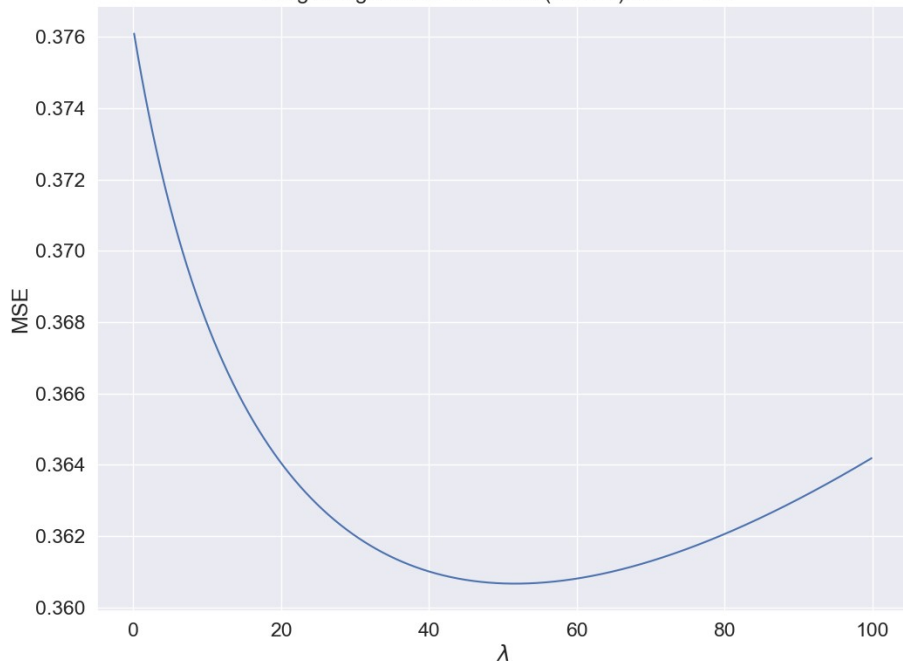


We can impose penalties

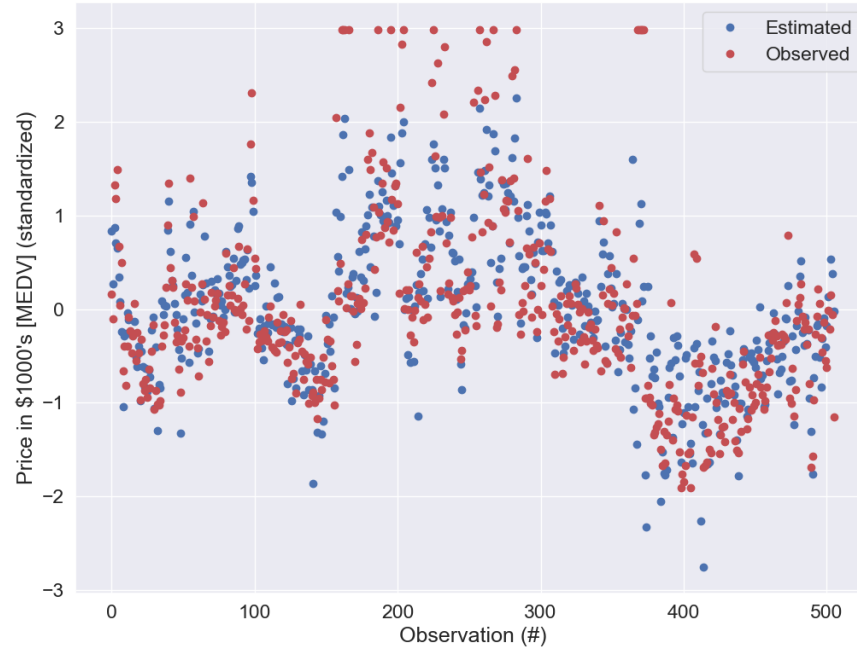
(but not on the intercept)

$$J(w)_{\text{Ridge}} = \sum_{i=1}^n \left(y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda \|w\|_2^2$$

Ridge Regression: min MSE (0.3607) at: $\lambda = 51.7$



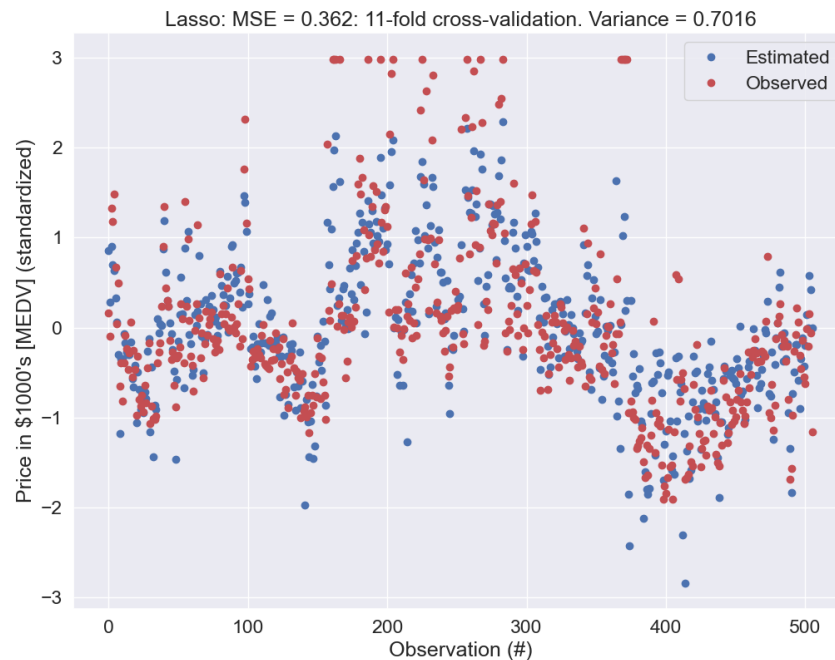
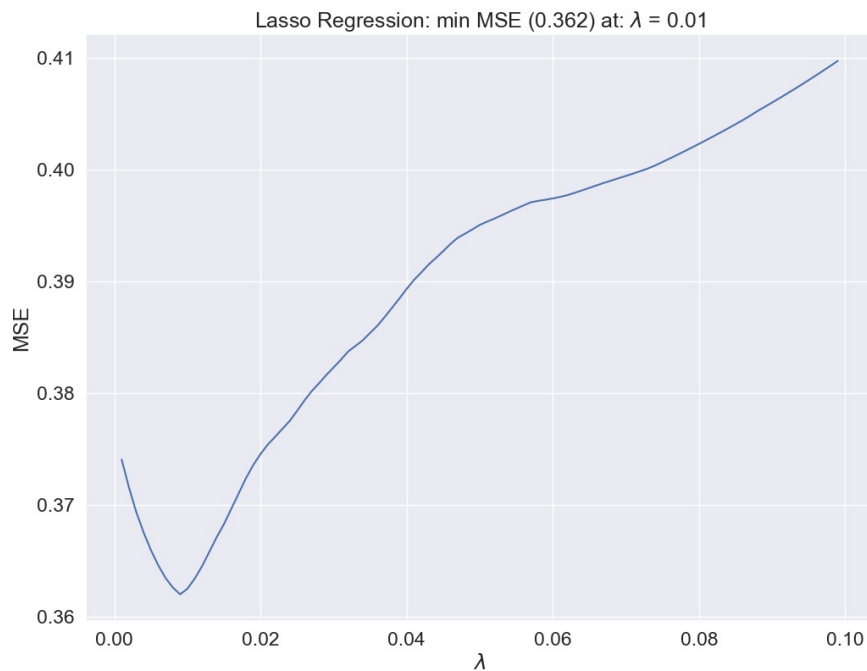
Ridge: MSE = 0.3607: 11-fold cross-validation. Variance = 0.6532



We can impose penalties

(but not on the intercept)

$$J(w)_{LASSO} = \sum_{i=1}^n \left(y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda \|w\|_1$$



Coefficients are shrunk

```
In [131]: OLS.coef_
```

```
Out[131]:
```

```
array([-0.09874812,  0.12473758,  0.02386168,  0.06945318, -0.22231612,  
        0.2911837 ,  0.0325356 , -0.32907266,  0.34212986, -0.28361575,  
       -0.21482076,  0.09763631, -0.43131412])
```

```
In [132]: RR.coef_
```

```
Out[132]:
```

```
array([-0.07536542,  0.08180161, -0.03182673,  0.07855868, -0.13185121,  
        0.31082552,  0.00548938, -0.23491485,  0.11242408, -0.0959682 ,  
       -0.18436614,  0.09154233, -0.36579723])
```

```
In [133]: lasso.coef_
```

```
Out[133]:
```

```
array([-0.07348948,  0.09107936, -0.          ,  0.07003181, -0.17110318,  
        0.30950805,  0.          , -0.29010247,  0.16456993, -0.1319311 ,  
       -0.19668957,  0.09063304, -0.41664587])
```


We can impose penalties

(but not on the intercept)

$$J(w)_{\text{ElasticNet}} = \sum_{i=1}^n \left(y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda_1 \sum_{j=1}^m w_j^2 + \lambda_2 \sum_{j=1}^m |w_j|$$

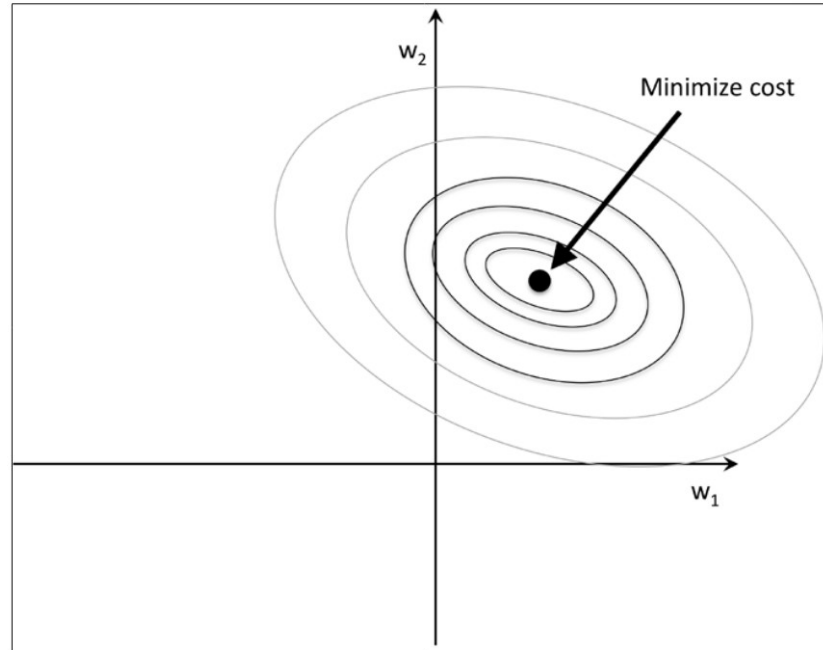
Did you learn?

Linear regression revisited (machine learning)

- 1) Learning some early classification methods
- 2) Learning how linear regression (with biasing penalties) can be constructed and cross-validated
- 3) Understanding that biasing in-sample solutions can improve out-of-sample predictions

Extra slides on regularization

$$J(w) = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$



(p. 113: Raschka, 2015)

L2 regularization

Why is the *budget* round?

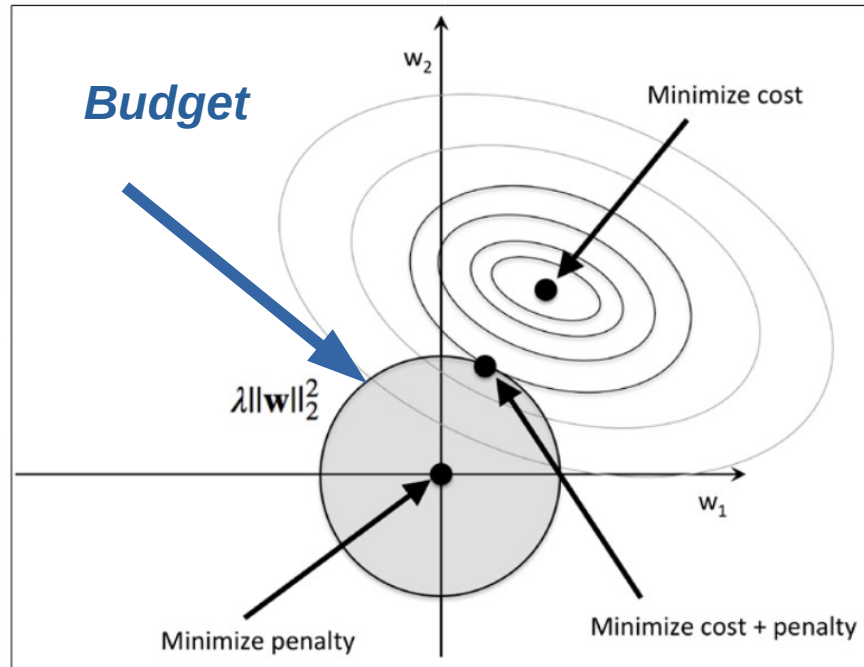
Compare with a circle centred at (0,0)

$$x^2 + y^2 = r^2$$

$$w_1^2 + w_2^2 = r^2$$

$$\|w\|_2 = \sqrt{(w_1^2 + w_2^2)}$$

(p. 114: Raschka, 2015)



$$J(w)_{Ridge} = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \|w\|_2^2$$

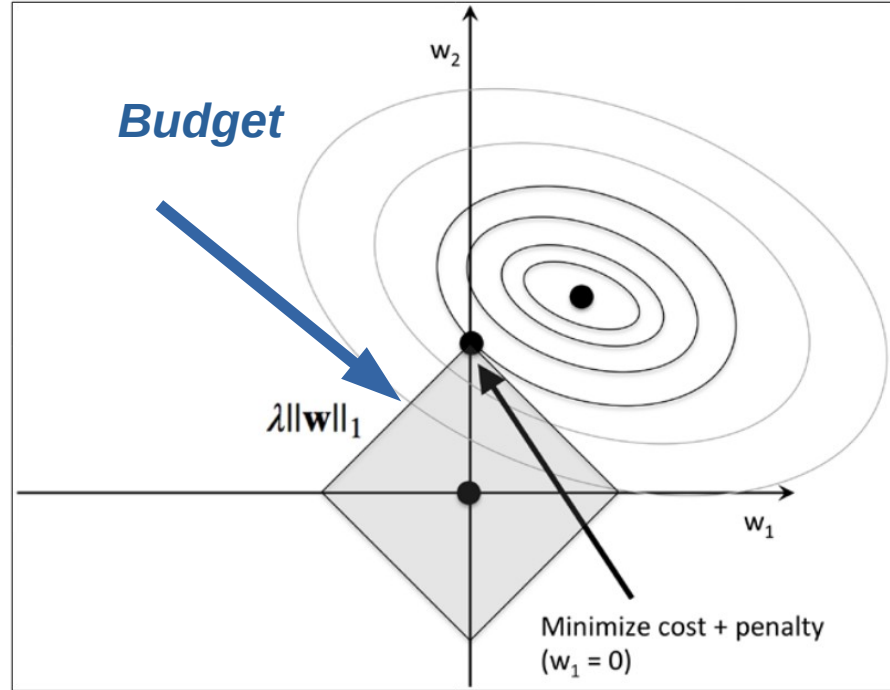
L1 regularization

Why is the *budget* square?

$$\|w\|_1 = |w_1| + |w_2|$$

if $w_1 = \max(w_1)$ then $w_2 = 0$

if $w_2 = \max(w_2)$ then $w_1 = 0$



$$J(w)_{LASSO} = \sum_{i=1}^n \left(y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda \|w\|_1$$

References

- Raschka, S., 2015. Python Machine Learning. Packt Publishing Ltd.