



# HAJ Lecture

Chapter 3. 분류(Classification)  
Chapter 4. 모델 훈련

Hands-On Machine Learning  
with Scikit-Learn, Keras & TensorFlow

| 3.1 MNIST

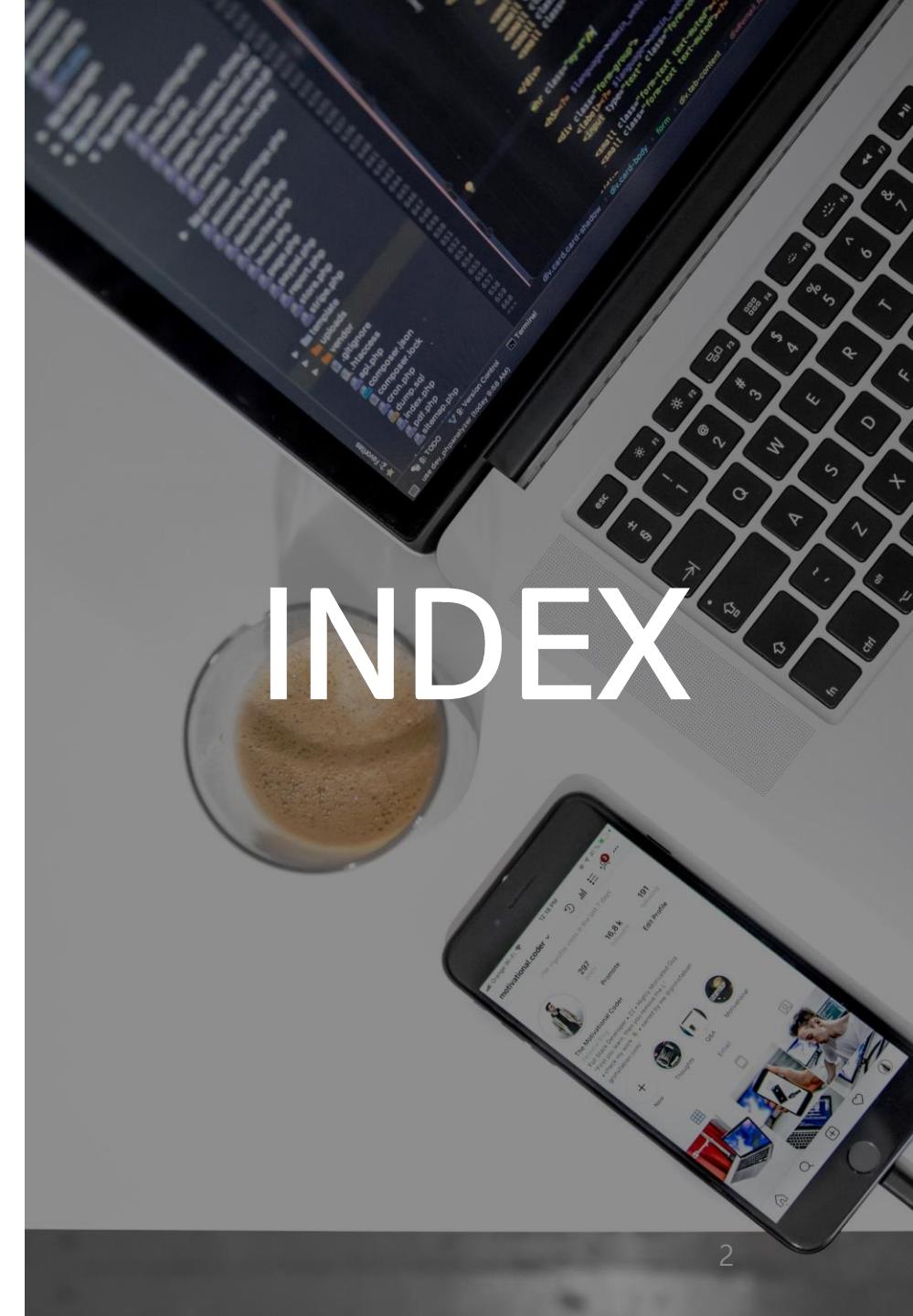
| 3.2 이진 분류기 훈련

| 3.3 성능 측정

| 3.4 다중 분류

| 3.6 다중 레이블 분류

| 3.7 다중 출력 분류



## 3.1 MNIST

미국 고등학생과 인구조사국 직원에 의해 만들어진 손글씨 숫자 이미지  
약 70000여개의 데이터셋

- 머신러닝 분야에서 학습용으로 많이 쓰이는 데이터셋
- Scikit-Learn에서 아래 코드를 통해 내려받을 수 있다.

```
>>> from sklearn.datasets import fetch_openml  
>>> mnist = fetch_openml('mnist_784', version=1)
```

- dictionary 구성: data, target, feature\_names, DESCR, ...
- 이미지 70,000개 & 각 이미지에 대해 784개의 feature

```
>>> X, y = mnist["data"], mnist["target"]  
>>> X.shape  
(70000, 784)  
>>> y.shape  
(70000,)
```

## | 3.2 이진 분류기 훈련

이진 분류기(binary classifier)

두 개의 클래스(class)로 구분할 수 있는 분류 시스템

e.g. 스팸메일 감지, MNIST에서 5인치의 여부 판단

STEP1: 타깃 벡터 만들기

→ '5' vs. '5 아님'

```
y_train_5 = (y_train == 5) # True for all 5s, False for all other digits.  
y_test_5 = (y_test == 5)
```

## | 3.2 이진 분류기 훈련

STEP2: 분류 모델 선택 후 훈련

in Scikit-Learn → SGDClassifier 객체

\* SGD: 확률적 경사 하강법(Stochastic Gradient Descent)

```
from sklearn.linear_model import SGDClassifier
```

```
sgd_clf = SGDClassifier(random_state=42)  
sgd_clf.fit(X_train, y_train_5)
```

- “무작위성(randomness)”를 기본 원리로 사용한다. = 확률적  
random\_state를 고정시키고 실행을 반복해보아라.  
random\_state를 변화시켜 보아라.

## | 3.3 성능 측정

### [1] 교차 검증

in Scikit-learn, `cross_val_score()`

SGDClassifier 모델을 k-겹 교차검증을 통해 평가할 것임.(k=3)

```
>>> from sklearn.model_selection import cross_val_score  
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")  
array([0.96355, 0.93795, 0.95615])
```

결과: 약 95% 이상의 정확도

## 3.3 성능 측정

### [1] 교차 검증

모든 데이터를 “5 아님” 클래스로 분류하는 분류기

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.91125, 0.90855, 0.90915])
```

결과: 약 90% 이상의 정확도(?)

→ 전체 이미지 중 “5”는 10% → 모든 이미지를 “5 아님”으로 판정하면 90% 정확성

→ 정확도를 성능 측정 지표로 하기에는 부적절한 상황 존재

(불균형한 분포의 데이터셋, 어떤 클래스가 다른 것보다 월등히 많거나 적은 경우)

### 3.3 성능 측정

#### [2] 오차 행렬(confusion matrix)

클래스 A의 샘플이 클래스 B로 잘못 분류된 횟수를 세는 것

e.g. 숫자 5의 이미지를 3으로 잘못 분류한 횟수 → [5, 3]

```
>>> from sklearn.metrics import confusion_matrix  
>>> confusion_matrix(y_train_5, y_train_pred)  
array([[53057,  1522],  
       [ 1325, 4096]])
```

		PREDICTION	
		5	5 아님
FACT	5	53057	1522
	5 아님	1325	4096

### 3.3 성능 측정

#### [2] 오차 행렬(confusion matrix)

		PREDICTION	
		5	5 아님
FACT	5	53057	1522
	5 아님	1325	4096

- 행 → 실제 클래스 / 열 → 예측한 클래스
- “5 아님”: negative class                    “5”: positive class
- “53057”: true negative
- “1522”: false positive
- “1325”: false negative
- “4096”: true positive
- 정밀도(precision) =  $\frac{TP}{TP+FP}$  (양성 예측의 정확도)
- 재현율(recall) =  $\frac{TP}{TP+FN}$  (정확하게 감지한 양성 샘플 비율)

### 3.3 성능 측정

#### [3] 정밀도와 재현율

```
>>> from sklearn.metrics import precision_score, recall_score  
>>> precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)  
0.7290850836596654  
>>> recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)  
0.7555801512636044
```

$F_1$  점수 = 정밀도 + 재현율 → 조화 평균(harmonic mean)

$$F_1 = \frac{2}{\frac{1}{\text{(정밀도)}} + \frac{1}{\text{(재현율)}}} = 2 * \frac{\text{(정밀도)} * \text{(재현율)}}{\text{(정밀도)} + \text{(재현율)}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

```
>>> from sklearn.metrics import f1_score  
>>> f1_score(y_train_5, y_train_pred)  
0.7420962043663375
```

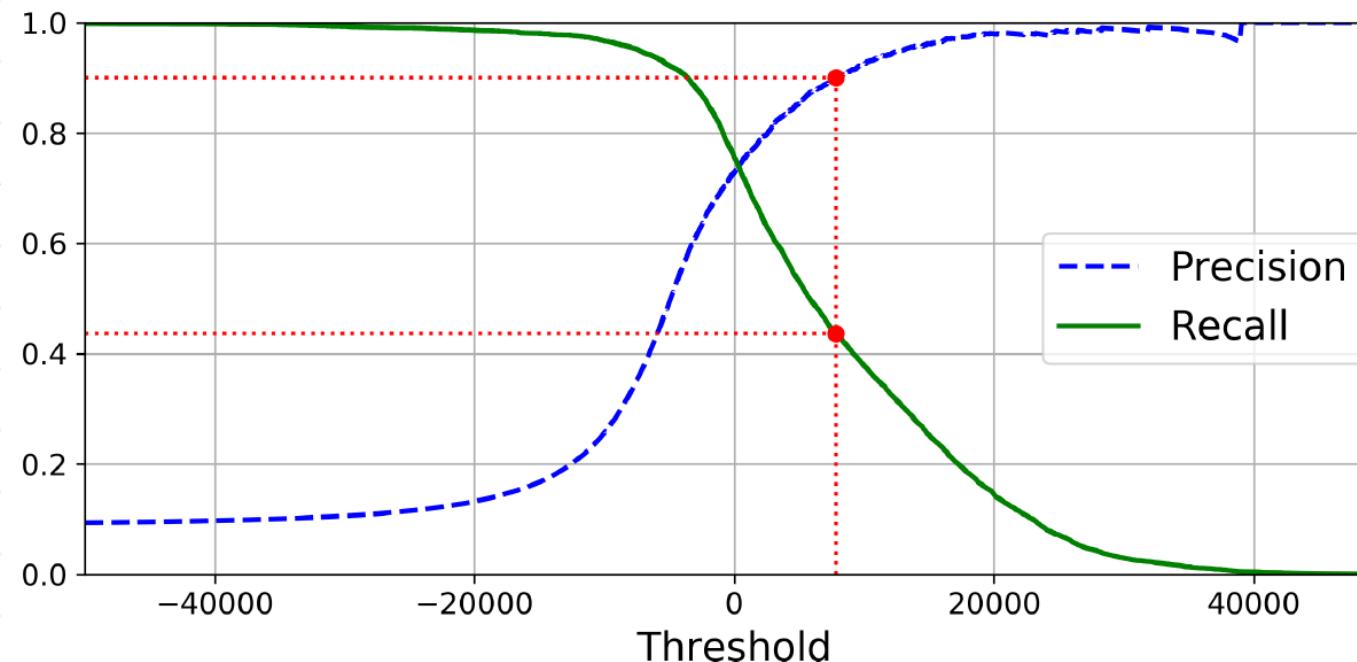
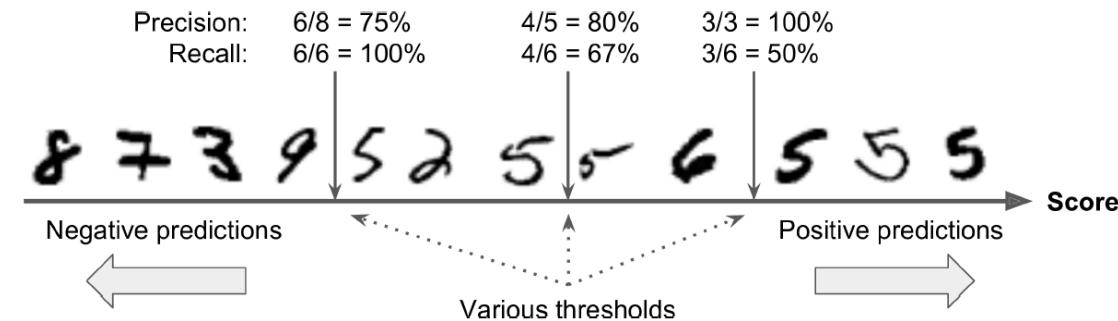
### 3.3 성능 측정

#### [3] 정밀도와 재현율

정밀도와 재현율은 trade-off 관계를 가진다.

임곗값을 올리면? 재현율 DOWN 정밀도 UP

임곗값을 내리면? 재현율 UP 정밀도 DOWN



## | 3.4 다중 분류

### 다중 분류기(multiclass classifier)

두 개 이상의 클래스(class)로 구분할 수 있는 분류 시스템

e.g. 혈액형 분류, MNIST 0~9 숫자 각각 식별

#### 0vR(one-versus-the-rest)

각 분류기의 결정 점수 중에서 가장 높은 클래스로 선택

#### 0v0(one-versus-one)

각 조합마다 이진 분류기를 훈련시키는 것

e.g. 0/1 구별, 0/2 구별 등 → 클래스 N개 →  $N(N-1)/2$  개 분류기

## 3.4 다중 분류

### 0vR(one-versus-the-rest)

각 분류기의 결정 점수 중에서 가장 높은 클래스로 선택

```
>>> sgd_clf.fit(X_train, y_train) # y_train, not y_train_5
>>> sgd_clf.predict([some_digit])
array([5], dtype=uint8)

>>> some_digit_scores = sgd_clf.decision_function([some_digit])
>>> some_digit_scores
array([[-15955.22627845, -38080.96296175, -13326.66694897,
       573.52692379, -17680.6846644 ,  2412.53175101,
      -25526.86498156, -12290.15704709, -7946.05205023,
      -10631.35888549]])
```

## 3.4 다중 분류

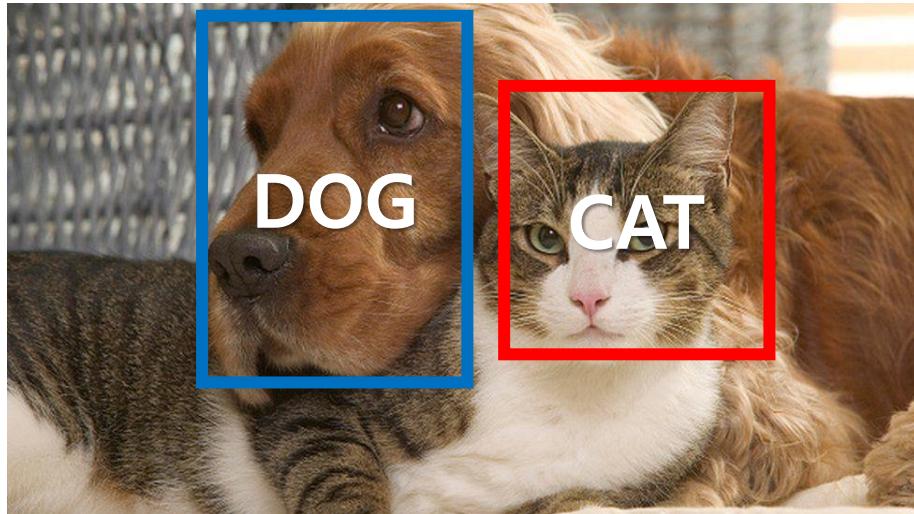
### 0v0(one-versus-one)

각 조합마다 이진 분류기를 훈련시키는 것

```
>>> from sklearn.multiclass import OneVsOneClassifier
>>> ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
>>> ovo_clf.fit(X_train, y_train)
>>> ovo_clf.predict([some_digit])
array([5], dtype=uint8)
>>> len(ovo_clf.estimators_)
45
>>> forest_clf.fit(X_train, y_train)
>>> forest_clf.predict([some_digit])
array([5], dtype=uint8)
>>> forest_clf.predict_proba([some_digit])
array([[0. , 0. , 0.01, 0.08, 0. , 0.9 , 0. , 0. , 0. , 0. , 0.01]])
```

## | 3.6 다중 레이블 분류

하나의 샘플에 여러 개의 클래스를 출력해야 하는 경우에 사용



e.g. 개와 고양이

[개, 고양이] 처럼 이진 꼬리표 여러 개가  
배열의 형태를 띠어야 함.

--> [개, 고양이] = [1, 1]인 경우에 해당

e.g. Alice, Cindy가 있는 사진

→ [Alice, Bob, Cindy] = [1, 0, 1]

## | 3.6 다중 레이블 분류

- e.g. 2개의 label → i) 숫자가 큰 값인가? ( $\rightarrow 7, 8, 9$ )  
ii) 숫자가 홀수인가? ( $\rightarrow 1, 3, 5, 7, 9$ )

```
from sklearn.neighbors import KNeighborsClassifier
```

```
y_train_large = (y_train >= 7)  
y_train_odd = (y_train % 2 == 1)  
y_multilabel = np.c_[y_train_large, y_train_odd]
```

```
knn_clf = KNeighborsClassifier()  
knn_clf.fit(X_train, y_multilabel)
```

```
>>> knn_clf.predict([some_digit])  
array([[False,  True]])
```

각 이미지에 두 개의 타겟 테이블이 담긴 `y_multilabel`

`KNeighborsClassifier` 인스턴스 만들고  
다중 타겟 배열 사용하여 훈련시킨다.

## | 3.7 다중 출력 분류

다중 출력 분류(multioutput classification)

다중 레이블 분류에서 한 레이블이 다중 클래스가 될 수 있도록 일반화한 것  
(→ 값을 두 개 이상 가질 수 있다.)

e.g. 이미지에서 잡음 제거하기

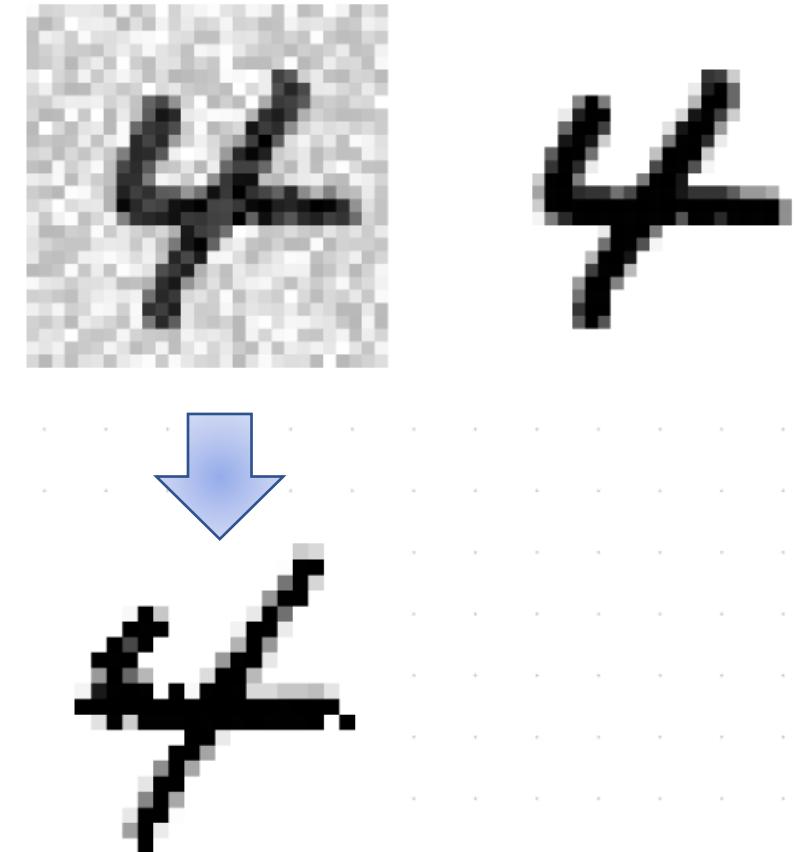
잡음이 많은 숫자 이미지 → 깨끗한 숫자 이미지

이미지 1개는  $255 \times 255$  픽셀이고,  
1 픽셀당 0~255의 값 중 하나를 갖는다.  
→ 다중 출력 분류 시스템에 해당한다.

## 3.7 다중 출력 분류

```
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```



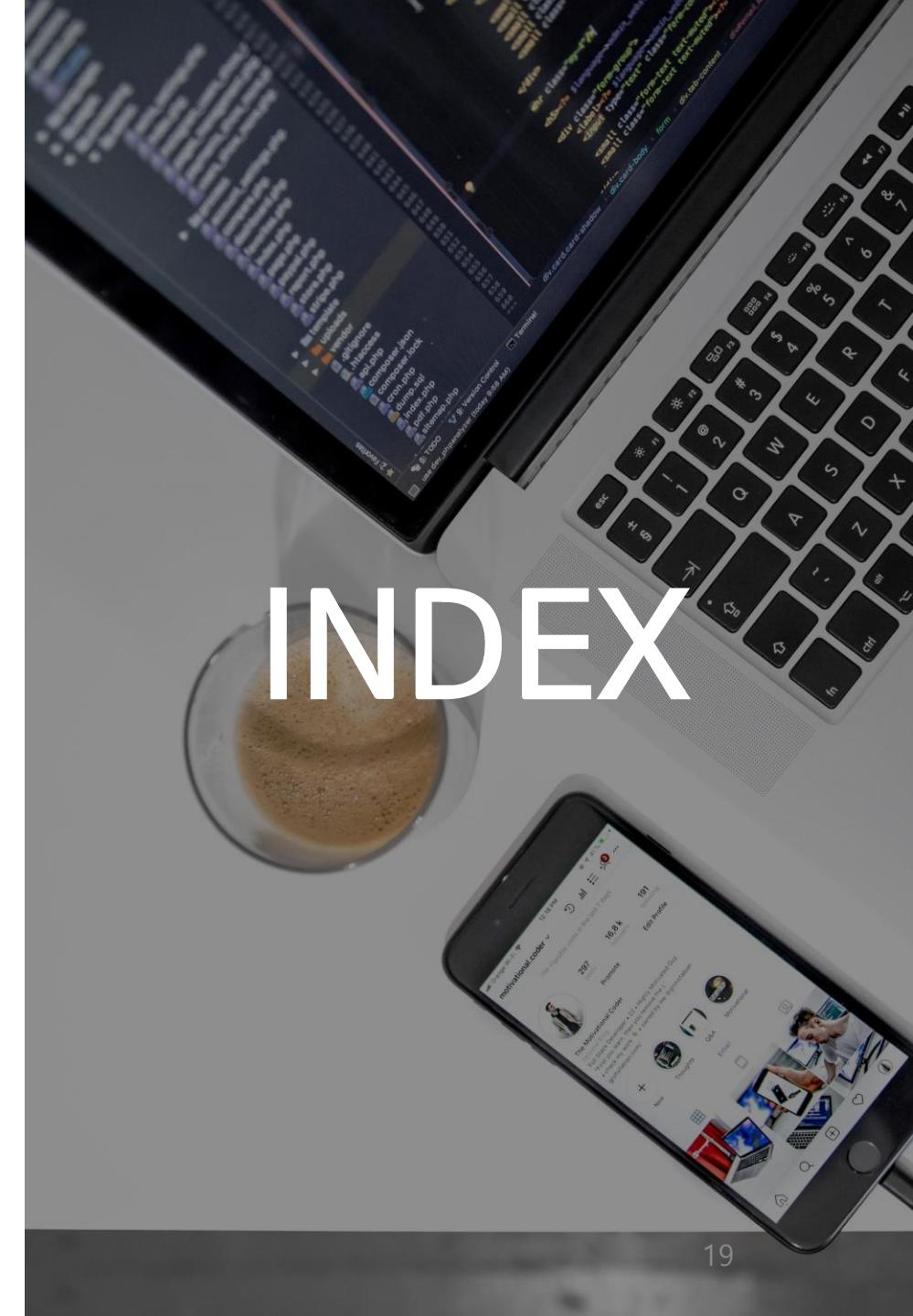
| 4.1 선형 회귀

| 4.2 경사 하강법

| 4.3 다항 회귀

| 4.4 학습 곡선

| 4.6 로지스틱 회귀



## 4.1 선형 회귀

입력 특성의 가중치 합(weighted sum)과 편향(bias)을 더해 예측치(hypothesis)를 만든다.

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

$$\hat{y} = h_{\theta}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

$\hat{y}$ : 예측값

$n$ : 특성의 수

$x_i$ : i번째 특성값

$\theta_j$ : j번째 모델 파라미터

## 4.1 선형 회귀

선형회귀 모델에서의 MSE 비용함수

$$\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m (\theta^T \mathbf{x}^{(i)} - y^{(i)})^2$$

정규 방정식(normal equation)

비용 함수를 최소화하는  $\theta$ 값을 미지수로 하는 방정식(해석적 방법)

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

## 4.1 선형 회귀

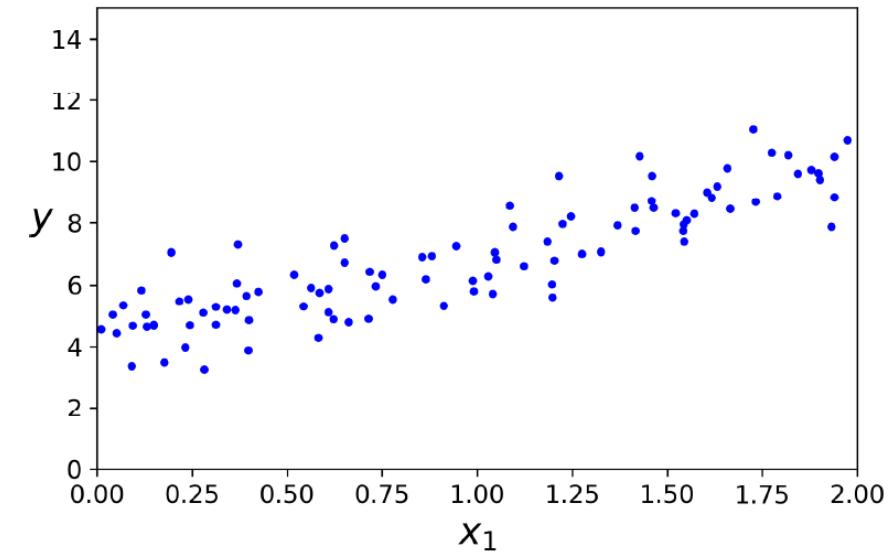
### 정규 방정식(normal equation)

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

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
X_b = np.c_[np.ones((100, 1)), X] # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

>>> theta_best
array([[4.21509616],
       [2.77011339]])
```



## 4.1 선형 회귀

Scikit-learn을 이용한 선형 회귀 모델 만들기

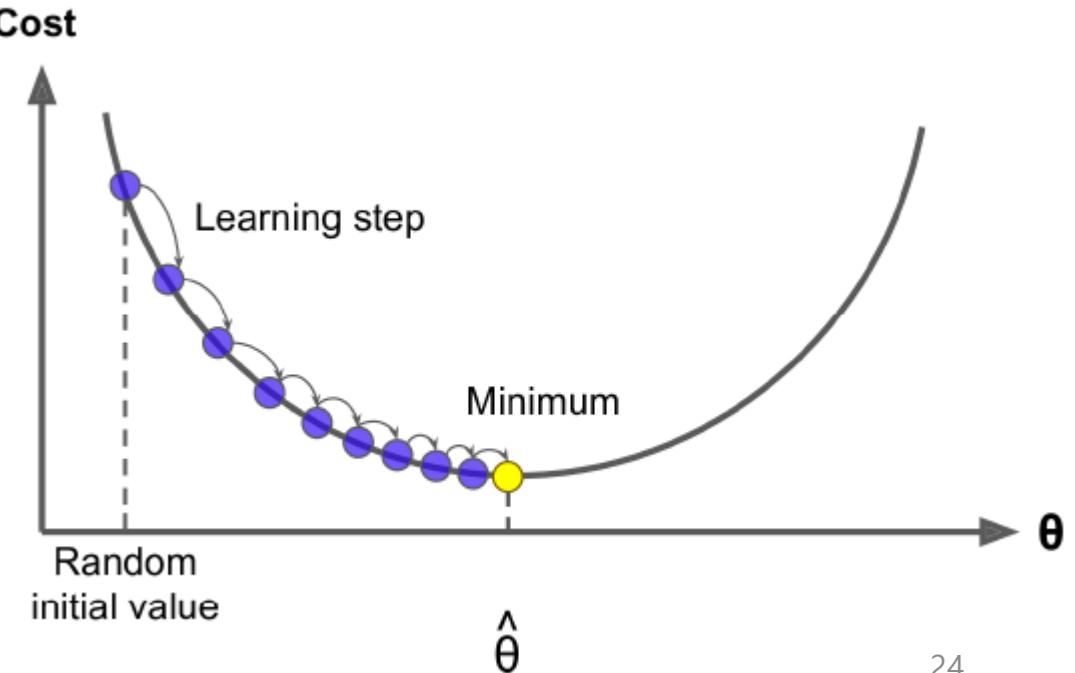
```
>>> from sklearn.linear_model import LinearRegression  
>>> lin_reg = LinearRegression()  
>>> lin_reg.fit(X, y)  
>>> lin_reg.intercept_, lin_reg.coef_  
(array([4.21509616]), array([[2.77011339]]))  
>>> lin_reg.predict(X_new)  
array([[4.21509616],  
       [9.75532293]])  
  
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)  
>>> theta_best_svd  
array([[4.21509616],  
       [2.77011339]])
```

## 4.2 경사 하강법

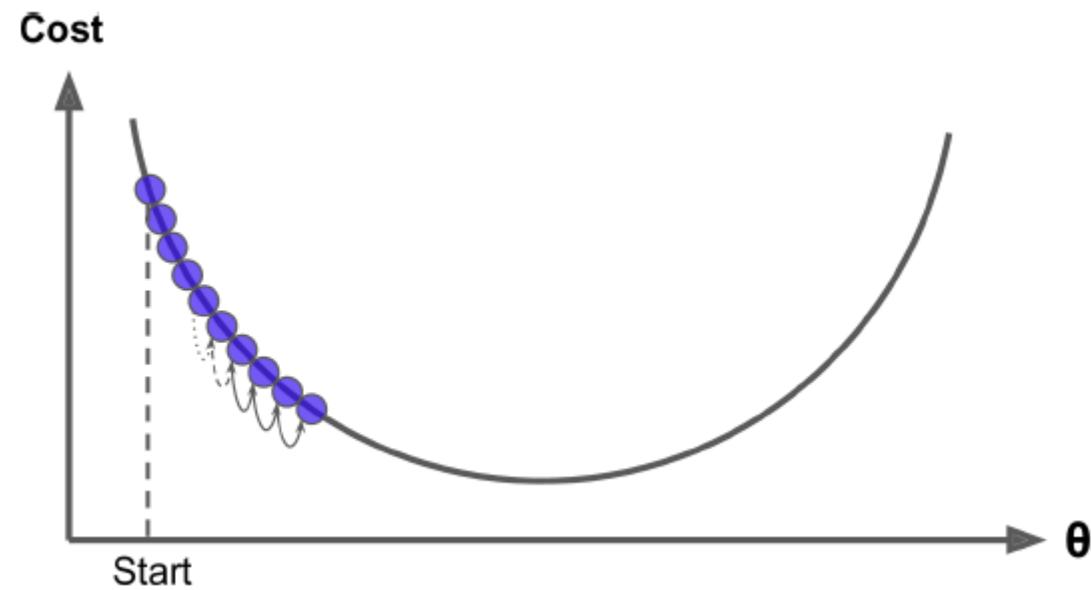
비용 함수(loss function)을 최소화하기 위해 반복해서  
파라미터를 조정해나가는 것

- 최적의 해법을 찾을 수 있는 일반적인 최적화 알고리즘
- 파라미터 벡터에 대해 비용함수의 현재 gradient를 계산
- gradient가 감소하는 방향으로 진행한다.

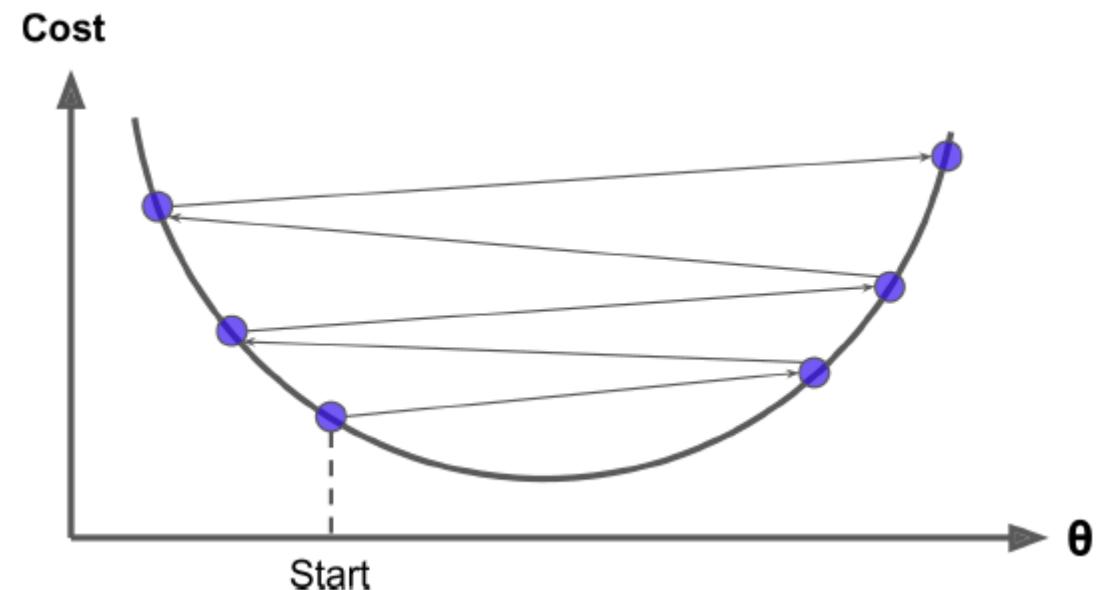
- 학습률(learning rate)
  - 스텝의 크기
  - 학습률이 작으면 반복횟수 증가



## 4.2 경사 하강법



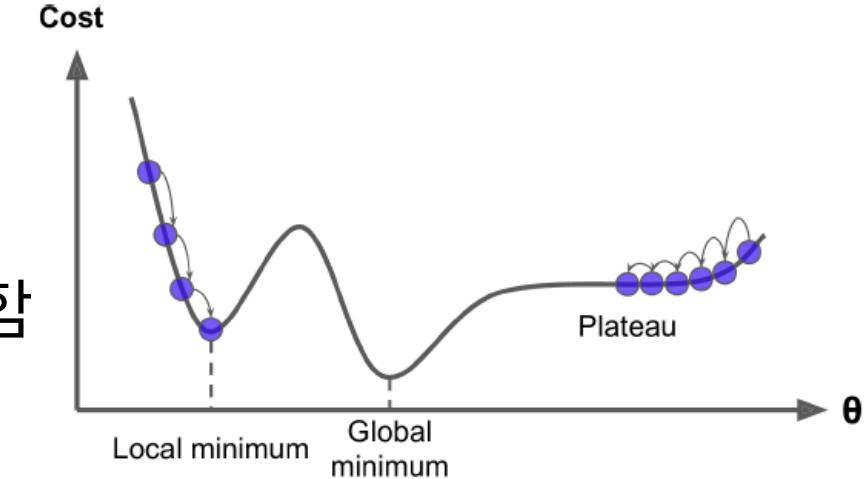
(학습률이 작은 경우)



(학습률이 큰 경우)

## 4.2 경사 하강법

- 모든 비용함수가 매끈한 그릇 모양인 것은 아님  
→ 패인 곳, 산마루, 평지 모양 등 경우의 수는 무한함
- 이에 따라 전역 최솟값(global minimum)보다 덜 좋은 지역 최솟값(local minimum)에 수렴할 가능성이 존재한다.
- 선형회귀에 한해서는 비용 함수가 전체 구간에서 볼록 함수임  
→ 어떤 두 점을 선택해도 곡선을 가로지르는 경우는 존재하지 않음  
→ 전역 최솟값에 수렴함이 보장되는 경우에 해당한다.



## 4.2 경사 하강법

### [1] 배치 경사 하강법

각 모델 파라미터에 대해 비용 함수의 gradient를 계산해야 한다.

편도함수(partial derivative)

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

그레디언트 벡터(gradient vector)

$$\nabla_{\theta} \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\theta - \mathbf{y})$$

## 4.2 경사 하강법

### [1] 배치 경사 하강법

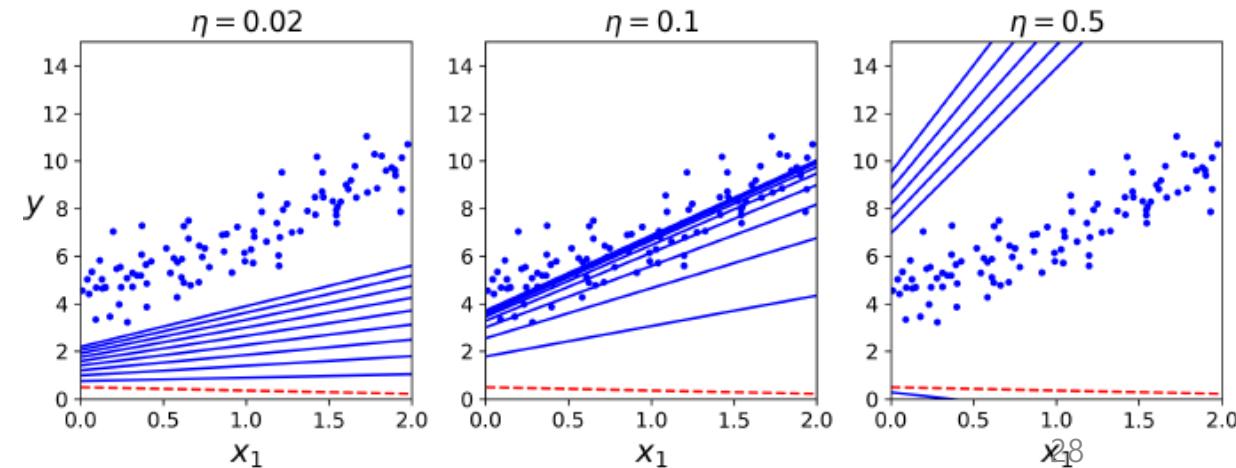
#### gradient descent step

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

```
eta = 0.1 # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1) # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```



## | 4.2 경사 하강법

### [2] 확률적 경사 하강법

매 스텝에서 한 개의 샘플을 부작위로 선택하고 그 하나의 샘플에 대한 Gradient를 계산한다.

- 각각의 반복에서 다루는 데이터의 양이 적기 때문에 속도가 빠르다.
- 무작위적이므로 불안정하다.
- 해결법: 초기에는 학습률을 높게 잡는다.(for 빠른 수렴, 지역 최솟값에 빠지지X)  
점차 작게 줄여 전역 최솟값에 도달하도록 유도(: 담금질 기법)

## 4.2 경사 하강법

### [2] 확률적 경사 하강법

```
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

## 4.2 경사 하강법

### [2] 확률적 경사 하강법

in Scikit-learn, SGDRegressor 객체

```
from sklearn.linear_model import SGDRegressor  
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)  
sgd_reg.fit(X, y.ravel())  
  
>>> sgd_reg.intercept_, sgd_reg.coef_  
(array([4.24365286]), array([2.8250878]))
```

## 4.3 다항 회귀

다항 회귀(polynomial regression)

다항식 형태의 회귀기법

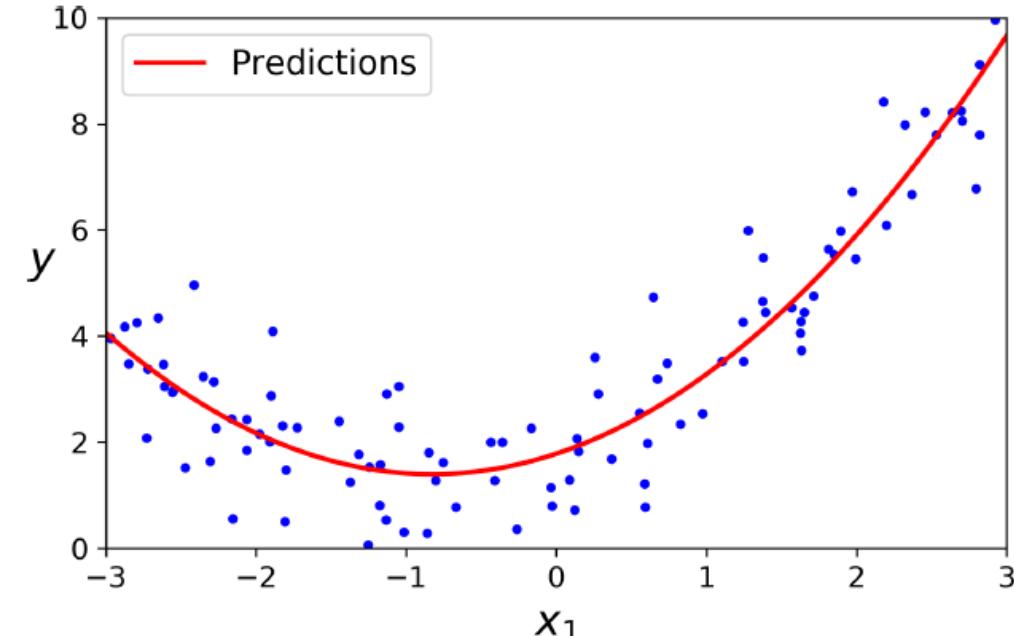
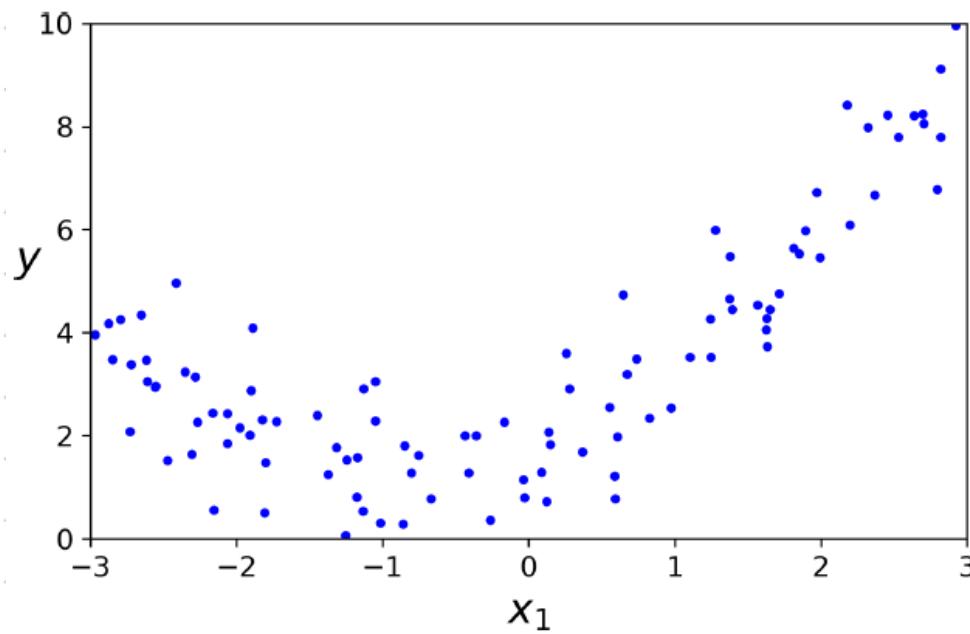
in Scikit-learn, PolynomialFeatures 객체

```
>>> from sklearn.preprocessing import PolynomialFeatures  
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)  
>>> X_poly = poly_features.fit_transform(X)  
>>> X[0]  
array([-0.75275929])  
>>> X_poly[0]  
array([-0.75275929, 0.56664654])
```

## 4.3 다항 회귀

### 다항 회귀(polynomial regression)

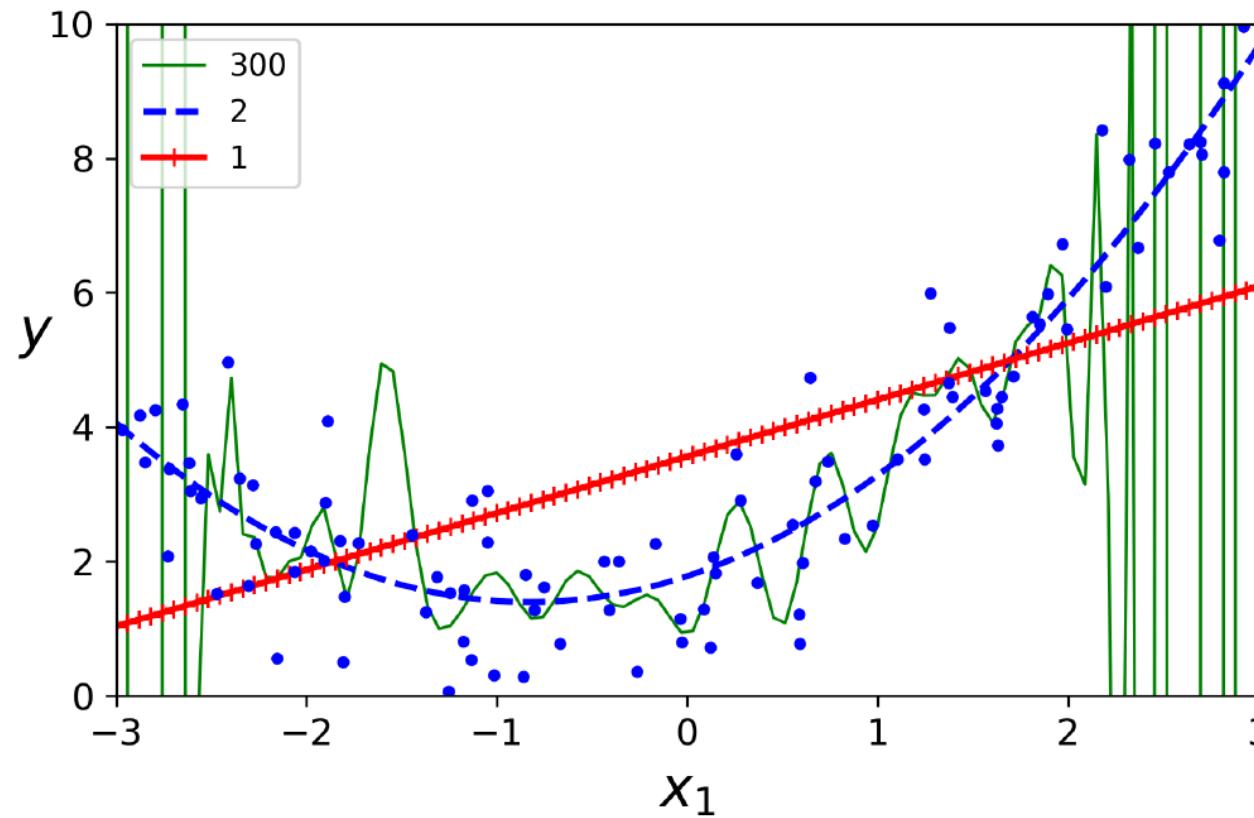
```
>>> lin_reg = LinearRegression()  
>>> lin_reg.fit(X_poly, y)  
>>> lin_reg.intercept_, lin_reg.coef_  
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```



## 4.4 학습 곡선

고차 다항 회귀일수록 보통의 선형회귀에 비해

훨씬 더 훈련 데이터에 잘 맞추려 하는 경향을 보인다.



- i) 1차 다항식  
→ 과소적합(under-fitting)
- ii) 300차 다항식  
→ 과대적합(over-fitting)

## 4.4 학습 곡선

### 학습 곡선(learning curve)

X: 훈련 세트 크기(반복) → Y: 훈련세트/검증세트의 모델 성능

질문) 어떻게 이 곡선을 도출시키는가?

훈련 세트에서 서브 세트의 크기를 달리하여 여러 번 훈련시켜 도출함.

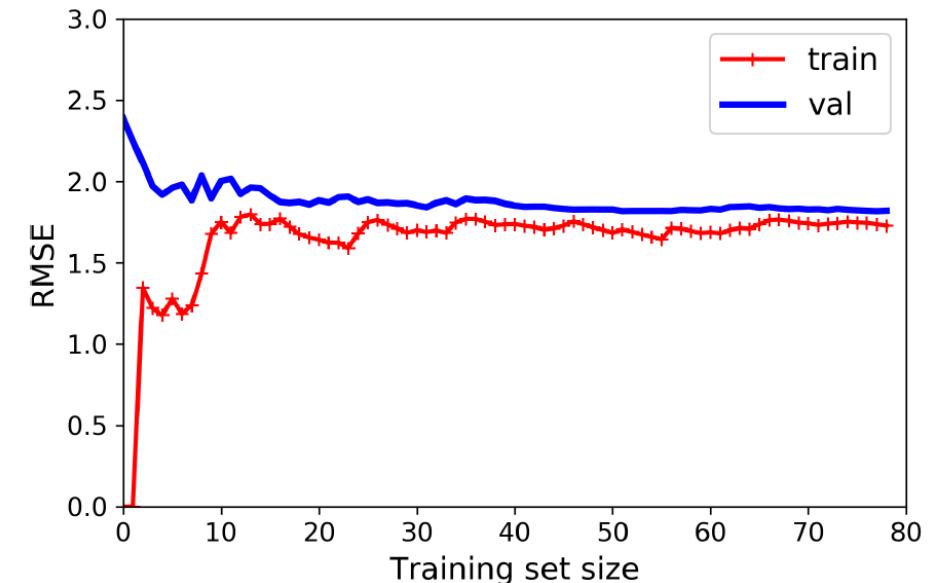
```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

## 4.4 학습 곡선

### 1) 과소적합 사례

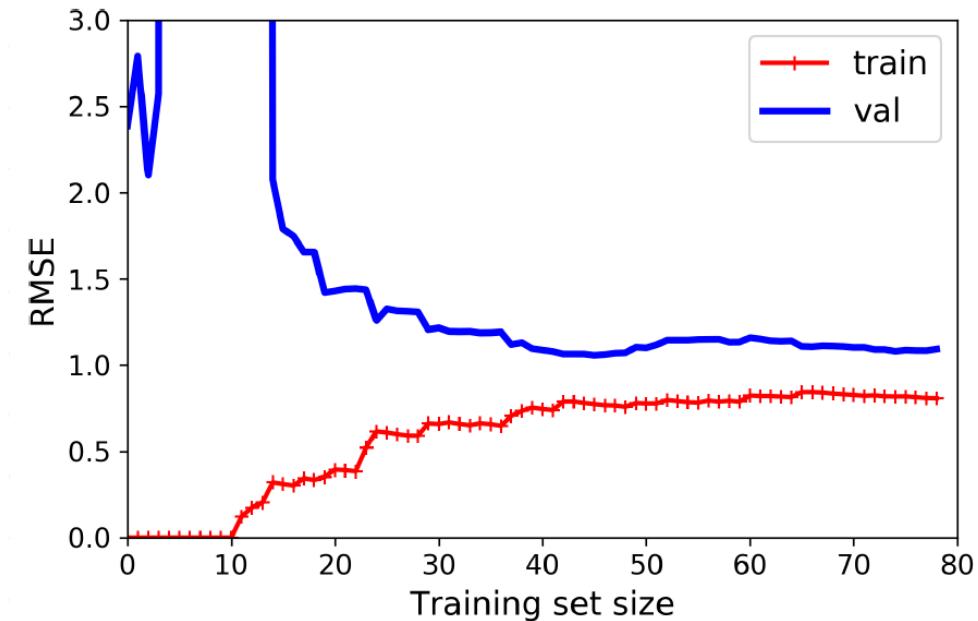
- 1~2개의 데이터에서는 완벽하게 작동함
- 곡선이 어느 정도 평편해질 때까지 오차는 계속하여 상승
- 훈련 세트에 샘플이 추가되어도 이후에는 더 이상 평균 오차가 달라지지 않음
- 두 곡선이 수평한 구간을 형성한 채, 높은 오차에서 머물러 있음.



## 4.4 학습 곡선

### 2) 과대적합 사례

- 훈련 데이터의 오차가 이전보다 낮아짐
- 두 곡선 사이의 공간이 존재한다.
  - 훈련 데이터에서의 모델 성능이 검증 데이터에서보다 현저히 좋다.
  - (: 과대적합 모델의 특성)
- 더 큰 훈련세트를 사용하면 두 곡선의 거리가 점차 좁아지는 경향을 보임
- 줄일 수 없는 오차? → 데이터 자체의 잡음 때문



## | 4.6 로지스틱 회귀

### 로지스틱 회귀(Logistic Regression)

샘플이 특정 클래스에 속할 확률을 추정하는데 사용되는 기법

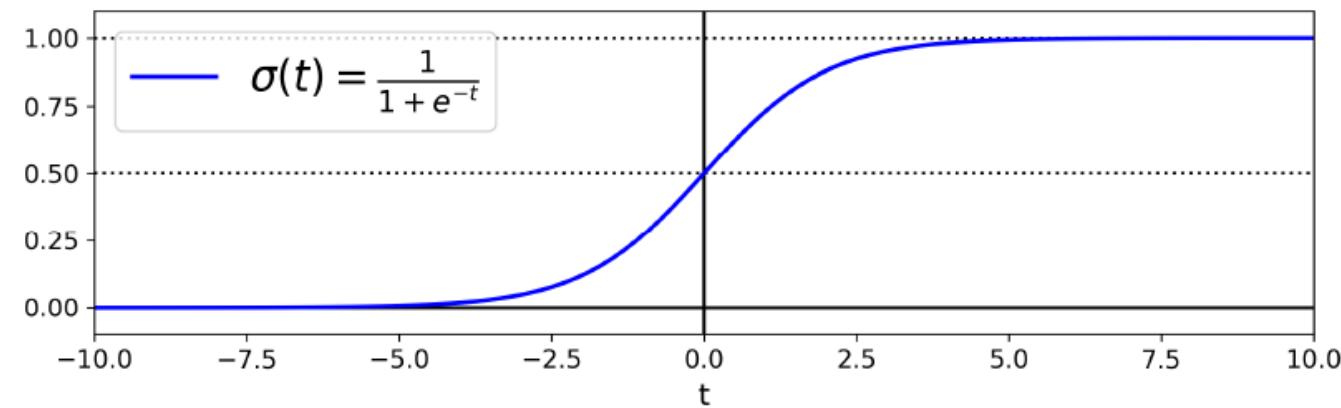
e.g. 스팸메일 감지, 코로나 양성/음성 등

→ 추정 확률이 50%가 넘으면 모델은 샘플이 해당 클래스에 속한다고 판정함

로지스틱(logistic): 0과 1 사이의 값을 출력하는 시그모이드 함수

로지스틱 회귀모델에서의 확률 추정

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\mathbf{x}^T \boldsymbol{\theta})$$



## 4.6 로지스틱 회귀

### 비용 함수

목적: 양성샘플에 대해서는 높은 확률을,  
음성샘플에 대해서는 낮은 확률을 추정하는 모델의 파라미터 찾기

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

하나의 훈련샘플에 대한 비용 함수

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

전체 훈련세트에 대한 비용 함수

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m (\sigma(\theta^T \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

비용 함수의 편미분

## 4.6 로지스틱 회귀

### Iris Dataset

붓꽃 데이터셋, 3개의 품종으로 이루어짐

꽃잎, 꽃받침의 너비와 길이, 약 150개



목적: 꽃잎의 너비를 기반으로 Iris-Versicolor 종을 감지하는 모델 설계

```
>>> from sklearn import datasets  
>>> iris = datasets.load_iris()  
>>> list(iris.keys())  
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']  
>>> X = iris["data"][:, 3:] # petal width  
>>> y = (iris["target"] == 2).astype(np.int) # 1 if Iris-Virginica, else 0
```

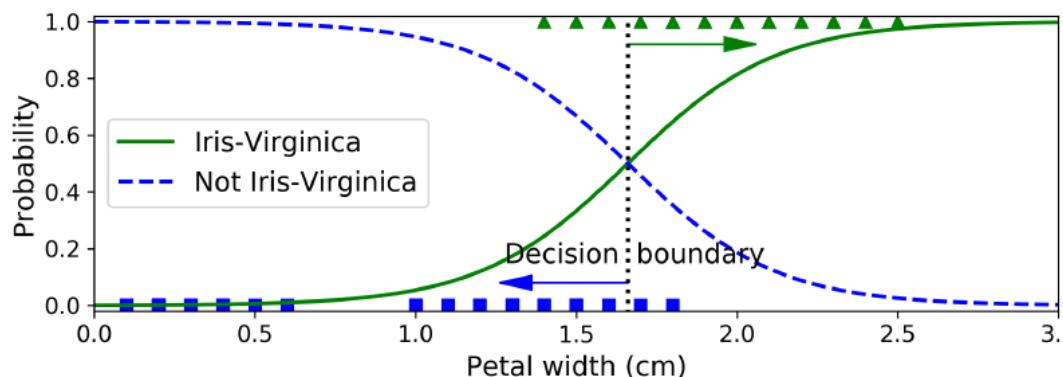
## 4.6 로지스틱 회귀

### 로지스틱 회귀 모델 수립

```
from sklearn.linear_model import LogisticRegression  
  
log_reg = LogisticRegression()  
log_reg.fit(X, y)
```

꽃잎의 너비가 0~3cm인 꽃에 대해 모델의 추정 확률 계산

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)  
y_proba = log_reg.predict_proba(X_new)  
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")  
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica")
```



```
>>> log_reg.predict([[1.7], [1.5]])  
array([1, 0])
```

## 4.7 소프트맥스 회귀

### 소프트맥스 회귀(Softmax Regression)

로지스틱 회귀 모델의 일반화된 개념

→ 다중 클래스 지원

→ 샘플  $\mathbf{x}$ 가 주어지면 각 클래스  $k$ 에 대한 점수를 계산하고 그 점수에 소프트맥스 함수를 적용하여 각 클래스의 확률을 추정한다.

$$s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)}$$

클래스  $k$ 에 대한 소프트맥스 점수

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

소프트맥스 함수

$$\hat{y} = \operatorname{argmax}_k \sigma(s(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x}) = \operatorname{argmax}_k \left( (\boldsymbol{\theta}^{(k)})^T \mathbf{x} \right)$$

소프트맥스 회귀 분류기의 예측

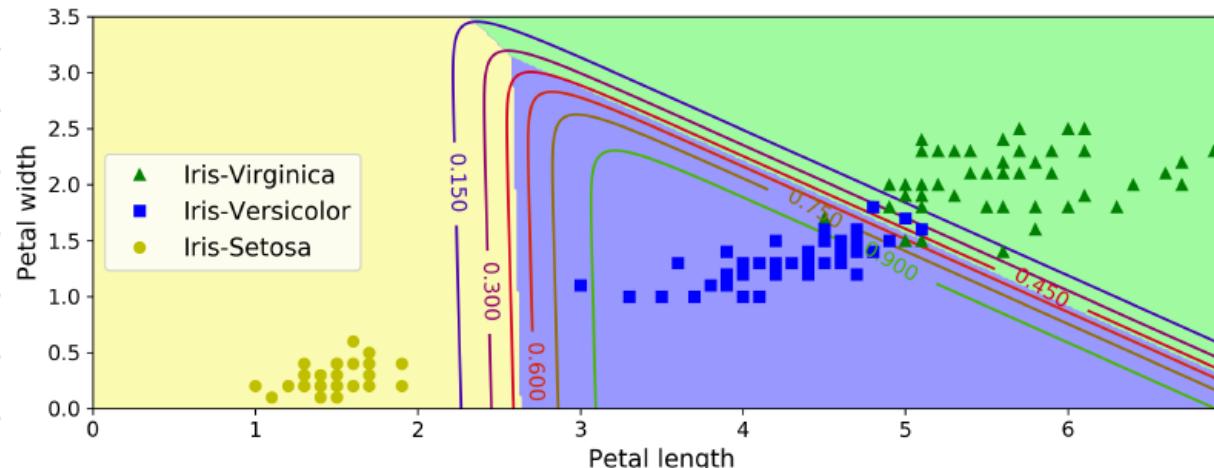
## 4.7 소프트맥스 회귀

LogisticRegression 객체에서 `multi_class = "multinomial"`

```
X = iris["data"][:, (2, 3)] # petal length, petal width
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial", solver="lbfgs", C=10)
softmax_reg.fit(X, y)

>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```





A close-up photograph of a Christmas tree branch. The branch is adorned with numerous small, glowing ornaments in shades of blue, green, yellow, and red. The background is dark, making the bright lights stand out. The overall effect is a festive and magical atmosphere.

Thank You