분류 (Classification)
모델 훈련 (Model Training)

Seolyoung Jeong, Ph.D.

경북대학교 IT 대학

분류 (Classification)

Contents

3.1 MNIST

- MNIST : 미국에서 손으로 쓴 70,000개의 작은 숫자 이미지를 모은 데이터셋 (머신러닝 분야의 'Hello World')

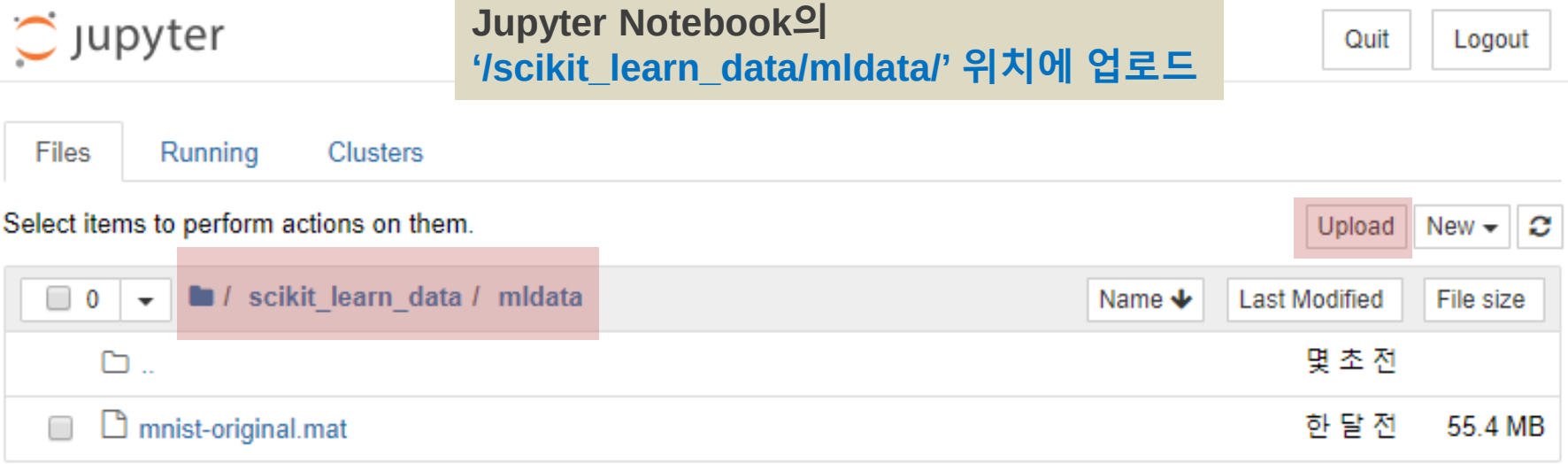
- 목적 : 28x28 픽셀의 필기 숫자 이미지를 어떤 숫자인지 판별

MNIST 데이터셋 다운로드

```
In [1]: from sklearn.datasets import fetch_mldata
        #mnist = fetch_mldata('MNIST original')
        mnist = fetch_mldata('mnist-original')
        mnist
```

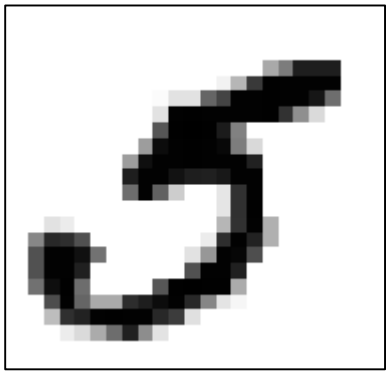
0.22 버전부터는
fetch_openml 함수로 변경

(주석) 다운로드 오류
'mnist-original.mat' 파일을 직접
Jupyter Notebook의
'/scikit_learn_data/mldata/' 위치에 업로드



MNIST 데이터 구조

- **MNIST 딕셔너리 구조**
  - DESCR : 데이터셋에 대한 설명
  - COL_NAMES : label, data
  - target : 레이블 배열
  - data : 2차원 배열 (70000, 784)
    - 70,000개의 이미지 (28x28)
    - 784개 특성



```
Out[1]: {'DESCR': 'mldata.org dataset: mnist-original',
 'COL_NAMES': ['label', 'data'],
 'target': array([0., 0., 0., ..., 9., 9., 9.]),
 'data': array([[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)}
```

MNIST 배열 확인

◆ 배열 값 확인

```
In [2]: X, y = mnist["data"], mnist["target"]
        X.shape
Out[2]: (70000, 784)

In [3]: y.shape
Out[3]: (70000,)

In [4]: 28*28
Out[4]: 784
```

- data (X) : image data 70,000개, 각 이미지에는 784개 특성 (28x28 픽셀)
  - 특성 : 0(흰색) ~ 255(검은색)까지의 픽셀 강도
- target (y) : label 70,000개
  - 각 이미지별 0~9에 해당하는 정답 숫자

MNIST 이미지 확인

- 임의 샘플의 특성 벡터 추출 → 28x28 배열로 reshape 후 확인

In [5]:
```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

some_digit = X[36000]   # 임의의 36,000번째 이미지 샘플
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap = matplotlib.cm.binary,
           interpolation="nearest")
plt.axis("off")

plt.show()
```

In [6]:
```
y[36000]
```

Out [6]: 5.0

MNIST 숫자 이미지

테스트 세트 분리

- **MNIST 데이터셋은 이미 분리되어 있음**
  - 훈련 세트 : 앞쪽 60,000개 이미지
  - 테스트 세트 : 뒤쪽 10,000개 이미지

```
In [7]: X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

- **훈련 세트를 섞어서 모든 교차 검증 폴드가 비슷해지도록... (하나의 폴드라도 특정 숫자가 누락되면 안됨)**

```
In [8]: import numpy as np

        shuffle_index = np.random.permutation(60000)
        X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

  - 경우에 따라, 섞는 것이 좋지 않을 수도 있음
  - 예) 시계열 데이터를 다루는 경우 (예: 주식가격, 날씨 예보)

3.2 이진 분류기 훈련

- **하나의 숫자(예: 숫자 5)만 식별하는 예**
  - 5-감지기 : '5'와 '5 아님' 두 개의 클래스 구분 → 이진 분류기

```
In [9]: y_train_5 = (y_train == 5)
        y_test_5 = (y_test == 5)
```

  - 5는 True, 다른 숫자는 모두 False
- **분류 모델 선택 후 훈련**
  - 모델1) 확률적 경사 하강법 (Stochastic Gradient Descent : SGD) 분류기
    - 장점: 매우 큰 데이터셋을 효율적으로 처리
    - 한번에 하나씩 훈련 샘플을 독립적으로 처리 (온라인 학습에 적합)
    - 확률적 : 무작위성 사용

```
In [10]: from sklearn.linear_model import SGDClassifier

         sgd_clf = SGDClassifier(max_iter=5, random_state=42)
         sgd_clf.fit(X_train, y_train_5)
```

    - 모델을 사용하여 샘플 숫자 식별 (5인지 아닌지)

```
In [11]: sgd_clf.predict([some_digit])

Out[11]: array([ True])
```

3.3 성능 측정

- **회귀모델 평가보다 분류기 평가에 사용할 수 있는 성능 지표가 더 많음!**

- **3.3.1 교차 검증을 사용한 정확도 측정**
  - 훈련 세트를 3개의 폴드로 나누고, 각 폴드에 대해 예측을 만든 후 평가하기 위해 나머지 폴드로 훈련시킨 모델 사용

```
In [12]: from sklearn.model_selection import cross_val_score
         cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
Out[12]: array([0.9662 , 0.95565, 0.9414 ])
```

  - 정확도 94~96%

3.3.1 교차 검증을 사용한 정확도 측정

- **확인용) 모든 이미지를 '5 아님' 클래스로 분류 (임의의 더미 분류기)**

```
In [13]: 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)
```

```
In [14]: never_5_clf = Never5Classifier()
         cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
Out[14]: array([0.9107 , 0.90985, 0.9084 ])
```

  - 정확도 : 90% 정도

→ 이미지의 10% 정도만 숫자 5이므로, 무조건 '5 아님'으로 예측하면 정확히 맞출 확률은 90%임

- **정확도는 분류기의 성능 측정 지표로 선호하지 않음 (특히, 불균형한 데이터셋을 다룰 때...)**

3.3.2 오차 행렬

- **분류기 성능 평가에 더 좋은 방법 : 오차 행렬 조사**
- **클래스 A의 샘플이 클래스 B로 분류된 횟수를 측정**
- **cross_val_predict() 함수**
  - 교차 검증 수행 후 각 테스트 폴드에서 얻은 예측값 반환 (score 아님)

```
In [15]: from sklearn.model_selection import cross_val_predict

         y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)

In [16]: y_train_pred

Out[16]: array([False, False, False, ..., False, False, False])
```

  - confusion_matrix() 함수로 오차 행렬 생성
    - 타깃 클래스 : y_train_5
    - 예측 클래스 : y_train_pred

```
In [17]: from sklearn.metrics import confusion_matrix

         confusion_matrix(y_train_5, y_train_pred)

Out[17]: array([[54129,   450],
                [ 2285,  3136]], dtype=int64)
```

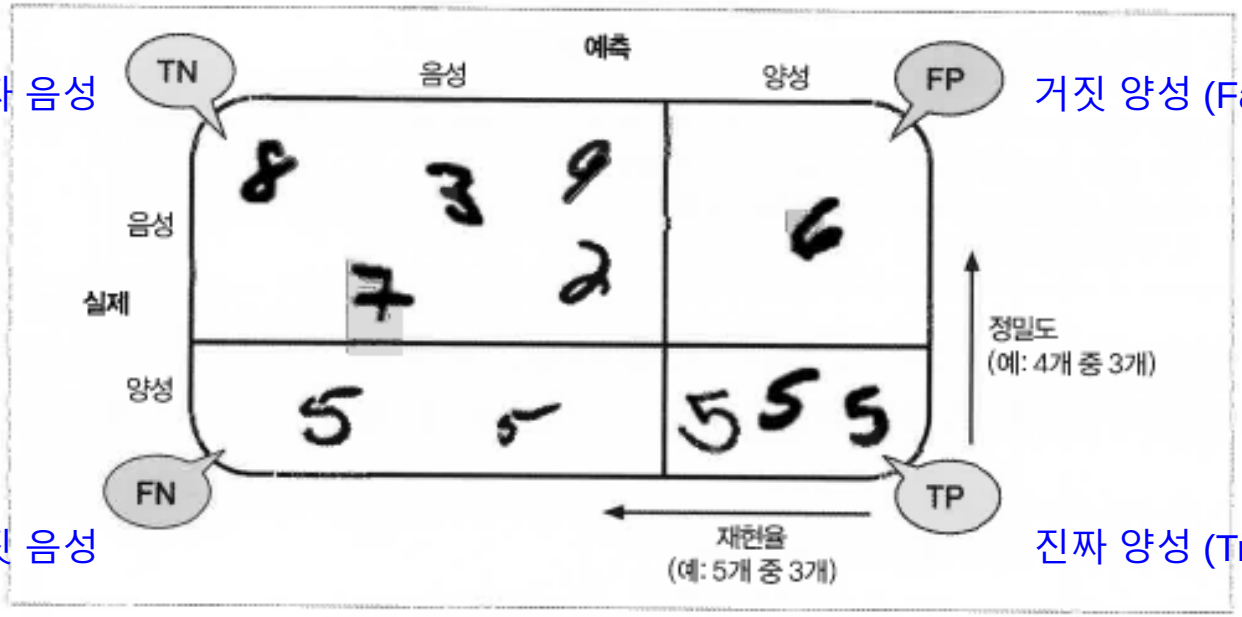
'5 아님' 이미지 →
'5' 이미지 →

'5 아님' 분류 (54129, 450)
'5' 분류 (2285, 3136)

3.3.2 오차 행렬

- 오차 행렬

그림 3-2 오차 행렬



(True Negative) 진짜 음성

거짓 양성 (False Positive)

(False Negative) 거짓 음성

진짜 양성 (True Positive)

- 완벽한 분류기라면 진짜 양성, 진짜 음성만 가지고 있음

```
In [18]: y_train_perfect_predictions = y_train_5

In [19]: confusion_matrix(y_train_5, y_train_perfect_predictions)

Out[19]: array([[54579,     0],
                [    0,  5421]], dtype=int64)
```

3.3.2 오차 행렬

- **정밀도 : 양성 예측의 정확도**
  - 분류기가 양성이라고 판단한 샘플(이미지) 중 실제 양성 샘플 수

$$\text{정밀도} = \frac{TP}{TP+FP}$$

TP : 진짜 양성 수, FP : 거짓 양성 수

- **재현율 : 분류기가 정확하게 감지한 양성 샘플의 비율**
  - 실제 양성인 샘플(이미지) 중에서 양성이라고 판단한 샘플 수
  - 민감도, 진짜 양성 비율

$$\text{재현율} = \frac{TP}{TP+FN}$$

FN : 거짓 음성 수

3.3.3 정밀도와 재현율

```
In [20]: from sklearn.metrics import precision_score, recall_score

         precision_score(y_train_5, y_train_pred)
Out[20]: 0.8745119910764082

In [21]: 3136 / (3136 + 450)
Out[21]: 0.8745119910764082

In [22]: recall_score(y_train_5, y_train_pred)
Out[22]: 0.5784910533111972

In [23]: 3136 / (3136 + 2285)
Out[23]: 0.5784910533111972
```

- 5로 판별된 이미지 중 87%만 정확, 전체 숫자 5에서 57.8%만 감지

◆ F_1 점수 : 정밀도와 재현율의 조화평균

정밀도와 재현율이 비슷한 분류기에서는 F_1 점수가 높음

```
In [24]: from sklearn.metrics import f1_score
         f1_score(y_train_5, y_train_pred)
Out[24]: 0.6963472854446542

In [25]: 3136 / (3136 + (450 + 2285)/2)
Out[25]: 0.6963472854446542
```

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

3.3.3 정밀도와 재현율

- **상황에 따라 정밀도 or 재현율 중요도 다를 수 있음**

- **예) 어린 아이에게 안전한 동영상을 걸러내는 분류기**
  - (재현율 보다는) 높은 정밀도 선호
  - 나쁜 동영상 몇개 노출보다 좋은 동영상이 많이 제외되더라도 (낮은 재현율) 안전한 것들만 노출(높은 정밀도)

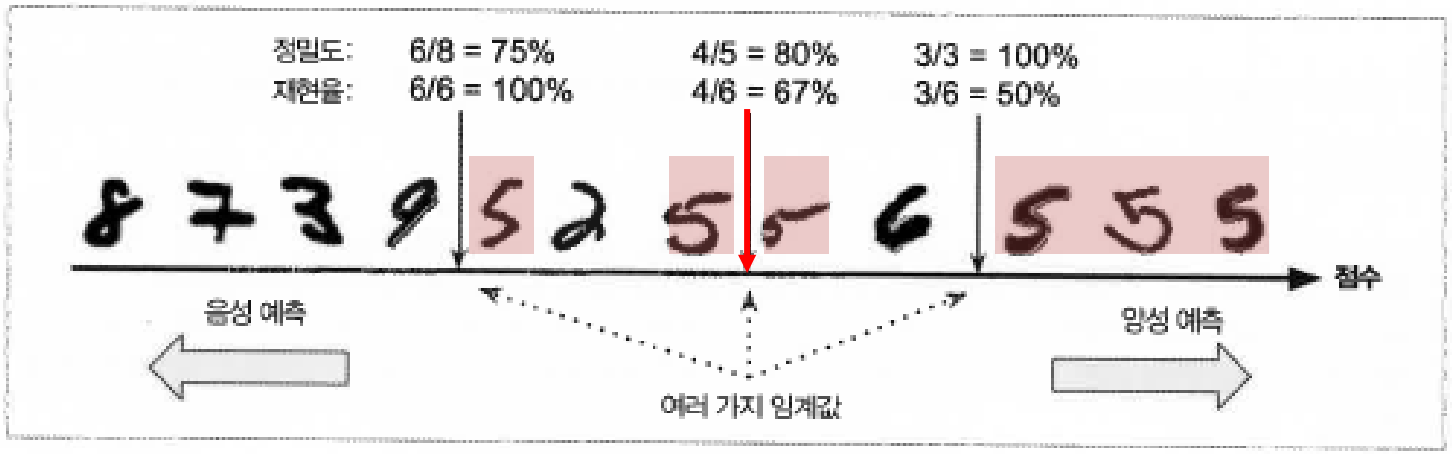
- **예) 감시 카메라를 통해 좀도둑을 잡아내는 분류기**
  - (정밀도 보다는) 높은 재현율 선호
  - 분류기의 재현율이 99%라면 정확도가 30%만 되더라도 괜찮음
  - 잘못된 알람 자주 발생하나, 거의 모든 좀도둑을 잡음

- **정밀도 / 재현율 트레이드오프 관계**
  - 정밀도를 올리면 재현율이 줄고, 그 반대도 마찬가지

3.3.4 정밀도/재현율 트레이드오프

- **현재 분류기(SGD)의 결정 점수**
  - 결정함수(decision function)로 각 샘플 점수 계산
  - 점수 > 임곗값 : 양성 클래스에 할당 (아니면 음성 클래스에 할당)

그림 3-3 결정 임곗값과 정밀도/재현율 트레이드오프



  - 결정 임곗값 : 가운데 화살표
    - 양성 예측 : 진짜 양성(숫자 5) 4개, 거짓 양성(숫자6) 1개
      - 정밀도 80% (5개 중 4개)
    - 실제 숫자 5는 6개, 분류기는 4개만 감지
      - 재현율 67% (6개 중 4개)
  - 임곗값을 높이면 : 정밀도 (3/3) = 100%, 재현율 (3/6) = 50%
  - 임곗값을 내리면 : 정밀도 (6/8) = 75%, 재현율 (6/6) = 100%

3.3.4 정밀도/재현율 트레이드오프

- **사이킷런에서 임곗값을 직접 지정할 수는 없지만, 예측에 사용한 점수 확인 가능**
  - decision_function()에 의한 샘플 숫자의 점수

```
In [26]: y_scores = sgd_clf.decision_function([some_digit])
         y_scores
Out[26]: array([57013.16411891])
```

  - SGDClassifier 임곗값 = 0 → 결과 True

```
In [27]: threshold = 0
         y_some_digit_pred = (y_scores > threshold)
         y_some_digit_pred
Out[27]: array([ True])
```

  - SGDClassifier 임곗값 = 200000 → 결과 False

```
In [28]: threshold = 200000
         y_some_digit_pred = (y_scores > threshold)
         y_some_digit_pred
Out[28]: array([ True])
```

3.3.4 정밀도/재현율 트레이드오프

- **적절한 임곗값 결정**
  - 훈련 세트에 있는 모든 샘플의 점수 구함 (결정 점수)

In [29]:
```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

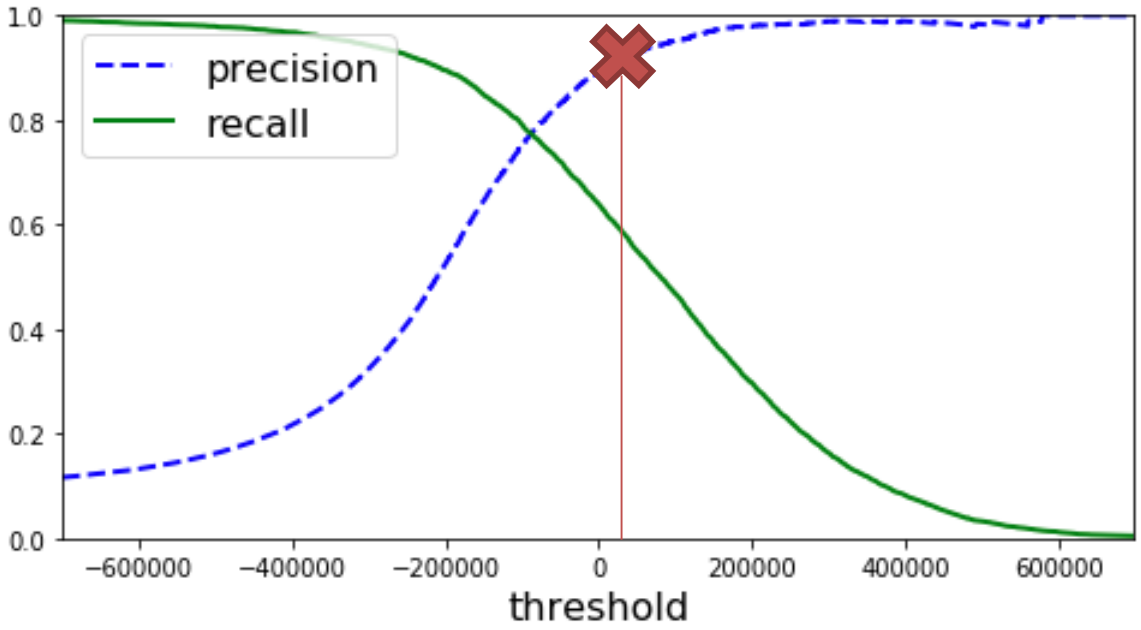
  - 이 점수를 이용하여 가능한 모든 임곗값에 대해 정밀도와 재현율 계산

In [30]:
```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

  - 임곗값 함수로 정밀도 재현율 그래프

In [31]:
```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="precision", linewidth=2)
    plt.plot(thresholds, recalls[:-1], "g-", label="recall", linewidth=2)
    plt.xlabel("threshold", fontsize=16)
    plt.legend(loc="upper left", fontsize=16)
    plt.ylim([0, 1])

plt.figure(figsize=(8, 4))
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.xlim([-700000, 700000])

plt.show()
```

3.3.4 정밀도/재현율 트레이드오프



- 정밀도 90% 달성 분류기

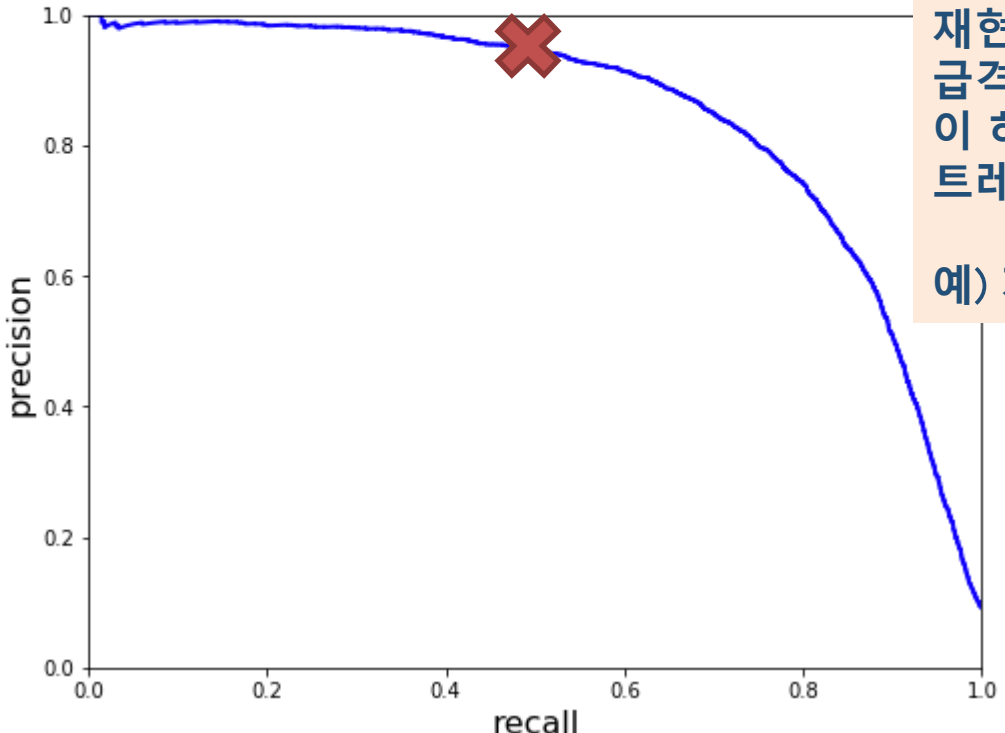
```
In [32]: y_train_pred_90 = (y_scores > 50000)

In [33]: precision_score(y_train_5, y_train_pred_90)
Out[33]: 0.9282162834058422

In [34]: recall_score(y_train_5, y_train_pred_90)
Out[34]: 0.5510053495665006
```

정밀도를 높게 만들더라도, 재현율이 너무 낮다면 전혀 유용하지 않음!

3.3.4 정밀도/재현율 트레이드오프

- 작업에 맞는 최선의 정밀도/재현율 트레이드오프를 만드는 임곗값 선택

In [35]:
```
def plot_precision_vs_recall(precisions, recalls):
    plt.plot(recalls, precisions, "b-", linewidth=2)
    plt.xlabel("recall", fontsize=16)
    plt.ylabel("precision", fontsize=16)
    plt.axis([0, 1, 0, 1])

plt.figure(figsize=(8, 6))
plot_precision_vs_recall(precisions, recalls)
plt.show()
```



재현율 60% 근처에서 정밀도가 급격하게 줄어들기 시작.
이 하강점 직전을 정밀도/재현율 트레이드오프로 선택하는것이 좋다.

예) 재현율 50% 정도인 지점.

3.3.5 ROC 곡선

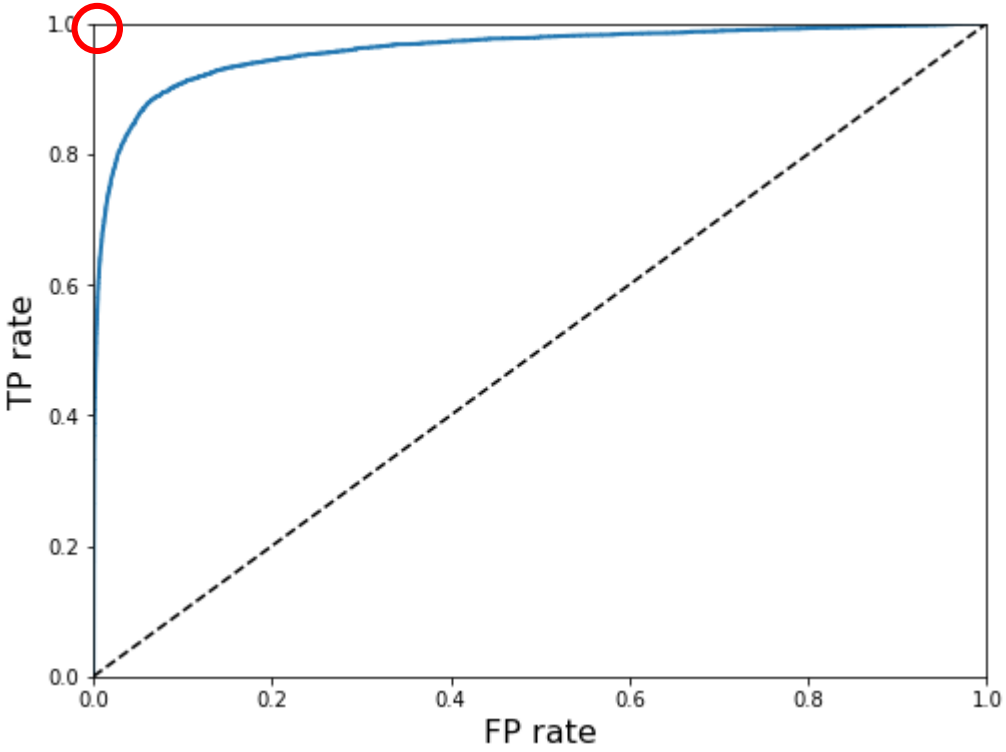
- **ROC (Receiver Operating Charactoeristic) 곡선**
  - 이진 분류 평가에 많이 사용됨
  - 거짓 양성 비율에 대한 진짜 양성 비율(재현율)의 곡선 (정밀도/재현율 곡선과 비슷하게 생김)
  - 거짓 양성 비율(FPR) = 1 – 진짜 음성 비율 (TNR: 특이도)
  - ROC 곡선 : 민감도(재현율)에 대한 1- 특이도 그래프
  - 여러 임곗값에서 TPR, FPR 계산 : roc_curve() 함수

In [36]:
```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

In [37]:
```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('FP rate', fontsize=16)
    plt.ylabel('TP rate', fontsize=16)

plt.figure(figsize=(8, 6))
plot_roc_curve(fpr, tpr)

plt.show()
```

3.3.5 ROC 곡선

- **재현율(TPR)이 높을수록 분류기가 만드는 거짓 양성(FPR)이 늘어남**
  - 점선 : 완전한 랜덤 분류기의 ROC 곡선
  - 좋은 분류기 : 점선에서 최대한 멀리 떨어져 있어야 (왼쪽 위 모서리)



3.3.5 ROC 곡선

- **ROC 곡선 아래 면적 : Area Under the Curve (AUC)**
  - 완벽한 분류기는 ROC의 AUC가 1
  - 완전한 랜덤 분류기는 0.5

```
In [38]: from sklearn.metrics import roc_auc_score

         roc_auc_score(y_train_5, y_scores)
```

```
Out[38]: 0.9592257393320891
```

- **예) RandomForestClassifier vs. SGDClassifier 비교**
  - 훈련 세트의 샘플에 대한 점수

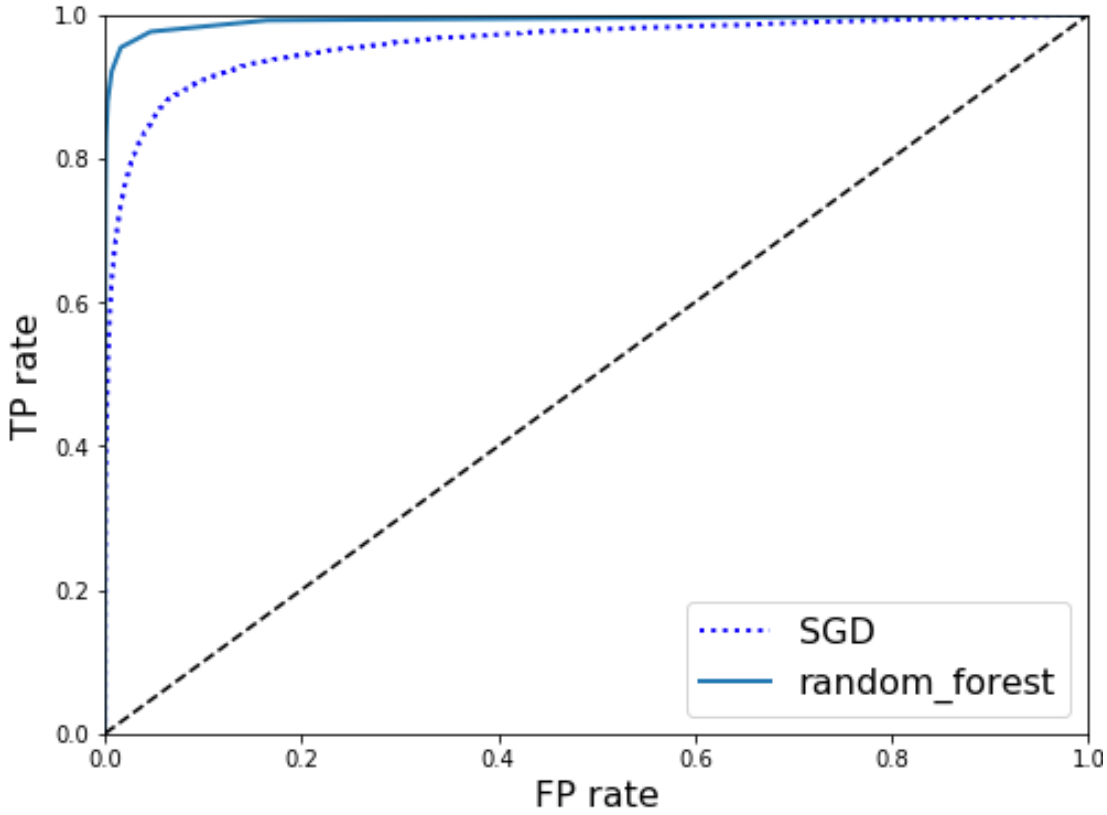
```
In [39]: from sklearn.ensemble import RandomForestClassifier
         forest_clf = RandomForestClassifier(n_estimators=10, random_state=42)
         y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                             method="predict_proba")
```

  - 확률이 아니라, 점수 필요 → 양성 클래스 확률을 점수로 사용

```
In [40]: y_scores_forest = y_probas_forest[:, 1] # 점수는 양성 클래스의 확률입니다
         fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

3.3.5 ROC 곡선

In [41]:
```
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, "b:", linewidth=2, label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "random_forest")
plt.legend(loc="lower right", fontsize=16)

plt.show()
```



In [42]:
```
roc_auc_score(y_train_5, y_scores_forest)
```

Out [42]: 0.9920590644845406

3.4 다중 분류

- **다중 분류기 (다항 분류기) : 둘 이상의 클래스 구별**
- **여러 개 클래스 직접 처리 가능 : RandomForest, NaiveBayes**
- **이진분류만 가능 : SVM, 선형분류기**

- **이진분류기를 이용한 다중분류 구성 방법**
  - 특정 숫자 하나만 구분하는 숫자별 이진분류기 10개 (0~9)를 훈련시켜, 클래스가 10개인 숫자 이미지 분류 시스템 구성
    - 이미지 분류 시 각 분류기의 결정 점수 중 가장 높은 것을 클래스로 선택
    - 일대다(OvA) 전략
  - 0과 1 구별, 0과 2 구별, 1과 2구별 등 각 숫자 조합마다 이진 분류기 훈련
    - 일대일(OvO) 전략
    - 클래스가 N개라면 분류기는 $N \times (N-1) / 2$개 필요
    - MNIST 문제 : 45개 분류기 훈련 필요

3.4 다중 분류

- **다중 클래스 분류 작업에 이진 분류 알고리즘을 선택하면 사이킷런이 자동으로 감지해 OvA (SVM 분류기일 때는 OvO) 적용**
  - 0~9까지의 원래 타깃 클래스(y_train)을 사용하여 SGDClassifier를 훈련시키고 예측 생성
  - 내부에서는 사이킷런이 실제로 10개의 이진 분류기를 훈련시키고 각각의 결정 점수를 얻어 점수가 가장 높은 클래스를 선택
  - decision_function() 함수 : 클래스마다 하나씩, 총 10개의 점수 반환
  - → 가장 높은 점수 : 클래스 5

```
In [43]: sgd_clf.fit(X_train, y_train)
         sgd_clf.predict([some_digit])

Out[43]: array([5.])
```

```
In [44]: some_digit_scores = sgd_clf.decision_function([some_digit])
         some_digit_scores

Out[44]: array([[-134186.14709804, -308537.29543399, -459029.38543142,
                 -169598.9924357 , -544361.54006891,    4166.00984863,
                 -949748.23975407, -339994.78909335, -551579.76797367,
                 -695995.80653337]])
```

3.4 다중 분류

- OvO 혹은 OvA를 강제로 지정하기 위해 OneVsOneClassifier, OneVsRestClassifier를 사용
- SGDClassifier 기반으로 OvO 사용

```
In [45]: from sklearn.multiclass import OneVsOneClassifier
         ovo_clf = OneVsOneClassifier(SGDClassifier(max_iter=5, random_state=42))
         ovo_clf.fit(X_train, y_train)
         ovo_clf.predict([some_digit])
```

```
In [46]: len(ovo_clf.estimators_)
Out[46]: 45
```

- RandomForestClassifier 기반 훈련
  - 직접 샘플을 다중 클래스로 분류
  - 5를 100%확률로 추측

```
In [47]: forest_clf.fit(X_train, y_train)
         forest_clf.predict([some_digit])
Out[47]: array([5.])
```

```
In [48]: forest_clf.predict_proba([some_digit])
Out[48]: array([[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.]])
```

3.4 다중 분류

- 분류기 평가 : 교차 검증 사용

- 모든 테스트 폴트에서 86% 이상

```
In [49]: cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
```

```
Out[49]: array([0.86472705, 0.87324366, 0.86808021])
```

- 입력의 스케일을 조정하면 정확도 90%이상

```
In [50]: from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
```

```
Out[50]: array([0.91011798, 0.91079554, 0.9086363 ])
```

3.5 에러 분석

- **실제 프로젝트라면...**
  - 여러 모델을 시도하고,
  - 가장 좋은 몇 개를 골라,
  - GridSearchCV를 사용해 하이퍼파라미터를 세밀하게 튜닝하고,
  - 가능한 한 자동화
- **가능성이 높은 모델을 하나 선정, 모델의 성능 향상**
  - 만들어진 에러 종류 분석 → 오차행렬 분석

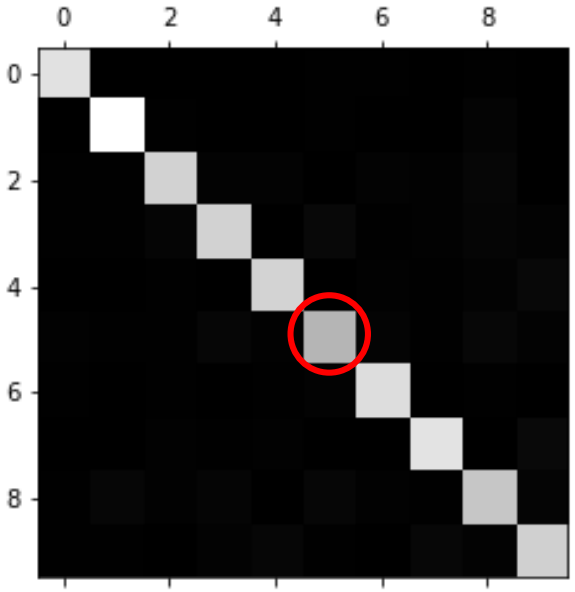
In [51]:
```
y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
conf_mx = confusion_matrix(y_train, y_train_pred)
conf_mx
```

Out[51]:
```
array([[5728,    4,   22,   12,   10,   47,   47,    8,   40,    5],
       [   1, 6492,   46,   22,    6,   39,    6,   10,  107,   13],
       [  50,   35, 5327,   97,   83,   25,   99,   58,  168,   16],
       [  50,   44,  136, 5337,    2,  229,   39,   60,  140,   94],
       [  18,   26,   39,    8, 5369,   11,   55,   29,   83,  204],
       [  71,   45,   33,  178,   67, 4607,  110,   30,  186,   94],
       [  30,   25,   45,    3,   41,   98, 5624,    6,   45,    1],
       [  22,   23,   71,   27,   54,    9,    5, 5801,   15,  238],
       [  47,  165,   66,  146,   13,  162,   58,   27, 5027,  140],
       [  42,   33,   26,   87,  156,   37,    2,  203,   84, 5279]],
      dtype=int64)
```

3.5 에러 분석

- **이미지화 →**
  - 배열에서 가장 큰 값 흰색, 가장 작은 값 검은색으로 정규화
- **대부분의 이미지가 올바르게 분류되었음**
  - 숫자 5의 색상이 다른 색에 비해 조금 어두움
  - 데이터셋에 숫자 5의 이미지가 적거나, 분류기가 숫자 5를 다른 숫자만큼 잘 분류하지 못함

```
In [52]: plt.matshow(conf_mx, cmap=plt.cm.gray)
         plt.show()
```



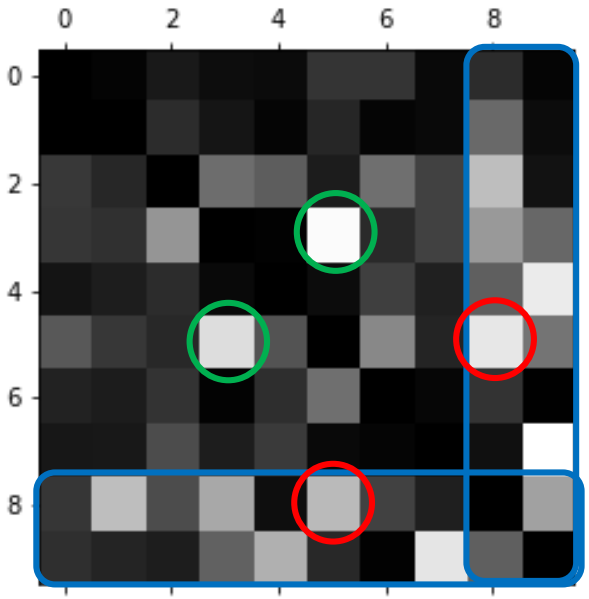
3.5 에러 분석

- 오차 행렬의 각 값을 대응되는 클래스의 이미지 개수로 나누어 에러 비율 비교

In [53]:
```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

- 다른 항목은 그대로 유지, 주대각선만 0으로 채워서 그래프로 그림

In [54]:
```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)

plt.show()
```

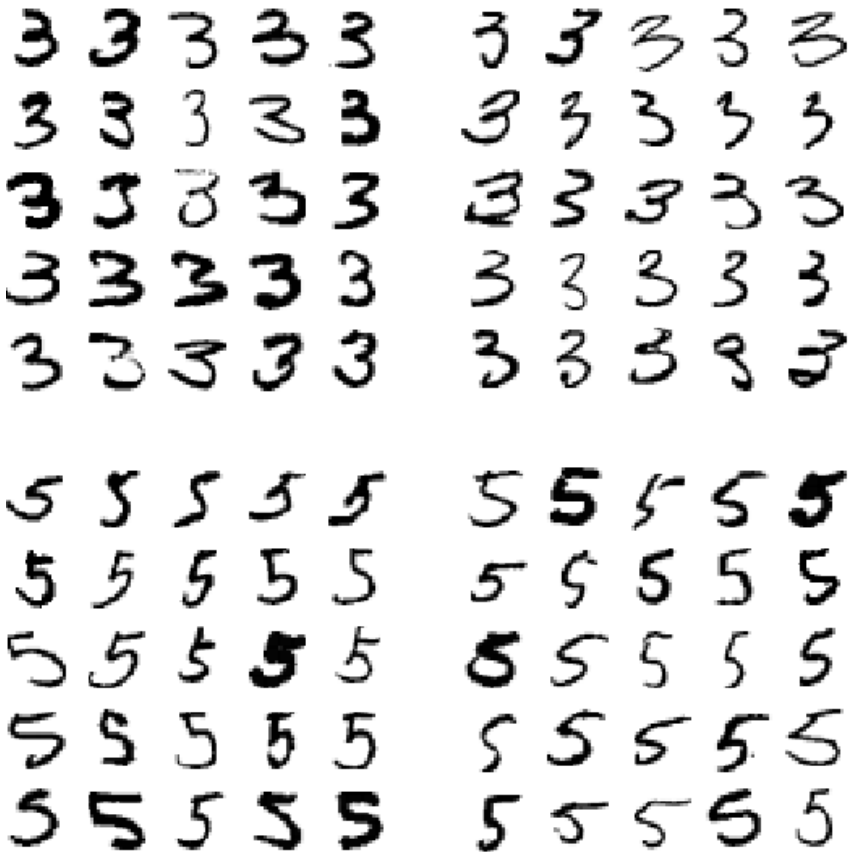


행 : 실제 클래스
열 : 예측한 클래스

- **8과 9의 열이 상당히 밝음**
  → 많은 이미지가 8과 9로 잘못 분류되었음
- **8과 9의 행도 밝음**
  → 숫자 8과 9가 다른 숫자들과 혼돈이 자주 됨
- **클래스 1의 열은 매우 어두움**
  → 대부분의 숫자 1이 정확하게 분류되었음
- **8→5로 잘못 분류된 경우 > 5→8로 잘못 분류된 경우**
- **3→5, 5→3 잘못 분류되는 경우 많음**

3.5 에러 분석

- 3과 5의 샘플 그림



왼쪽 블록 두 개 : 3으로 분류된 이미지
오른쪽 블록 두개 : 5로 분류된 이미지

3과 5는 몇 개의 픽셀만 다름
SGDClassifier를 사용하는 경우, 픽셀에 가중치를 할당하고, 픽셀 강도의 가중치 합을 클래스 점수로 계산
→ 3과 5 쉽게 혼동

분류기는 이미지의 위치나 회전 방향에 매우 민감
3과 5의 에러를 줄이는 방법 예)
이미지를 중앙에 위치시키고, 회전되어 있지 않도록 전처리

3.6 다중 레이블 분류

- **샘플마다 여러 개의 클래스를 출력해야 하는 경우**

- **예) 얼굴 인식 분류기**
  - 같은 사진에 여러 사람이 등장
  - 인식된 사람마다 레이블을 하나씩 할당
  - 분류기 인식 얼굴 : [앨리스, 밥, 찰리] 인 경우
  - 한 사진에 앨리스&찰리 → 분류기는 [1,0,1] 출력

- **다중 레이블 분류: 여러 개의 이진 레이블을 출력하는 분류 시스템**

3.6 다중 레이블 분류

- **각 숫자 이미지에 두 개의 타깃 레이블이 담긴 y-multilabel 배열 생성**
  - 7 이상
  - 홀수 여부

In [56]:
```
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)
```

Out[56]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

In [57]:
```
knn_clf.predict([some_digit])
```

Out[57]:
```
array([[False,  True]])
```

모델 훈련 (Model Training)

Contents

모델 훈련

- 머신러닝 모델, 훈련 알고리즘 → 블랙박스 취급
- 실제로 어떻게 작동하는지는 모름

- 어떻게 작동하는지 잘 이해하고 있으면...
- 적절한 모델, 올바른 훈련 알고리즘, 작업에 맞는 좋은 하이퍼파라미터를 빠르게 찾을 수 있음
- 디버깅이나 에러를 효율적으로 분석 가능
- → 특히 신경망을 이해, 구축, 훈련시키는데 필수

- **모델을 훈련시킨다. =**
  모델이 훈련세트에 가장 잘 맞도록 모델 파라미터를 설정한다.
  - 두 가지 방법) 직접 계산 가능한 공식 사용 /
    반복적 최적화 방식(경사 하강법)을 사용해서 모델 파라미터를 조금씩 바꾸면서 비용 함수를 훈련 세트에 대해 최소화

- **먼저, 모델이 훈련 데이터에 얼마나 잘 들어맞는지 측정**
- **성능 측정 지표 : 평균 제곱근 오차 (RMSE)**
- **즉, RMSE를 최소화하는 θ를 찾아야 함**
  - 실제로는 RMSE보다 평균제곱오차(MSE)를 최소화하는 것이 같은 결과를 내면서 더 간단
  - 선형 회귀 모델의 MSE 비용 함수

$$\mathrm{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\boldsymbol{T}} \cdot \boldsymbol{X}^{(i)} - \boldsymbol{y}^{(i)}\right)^2$$

4.1 선형 회귀 (Linear Regression)

- 가장 간단한 모델 중 하나
- '삶의 만족도' 선형 회귀 모델

$$\text{삶의만족도} = \theta_0 + \theta_1 \text{X1인당 GDP}$$

- 선형 모델
  - 예측값 = 입력 특성의 가중치 합 + 편향(또는 절편)이라는 상수

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

$\hat{y}$: 예측값, n: 특성 수, x_i: i번째 특성값($x_0 = 1$),
θ_j: j번째 모델 파라미터(편향(θ_0)과 가중치($\theta_1, \theta_2 \ldots \theta_n$) 포함)

- 벡터 형태로 표현

$$\hat{y} = \boldsymbol{h_\theta(x) = \theta^T \cdot X}$$

→ 선형 회귀 모델의 예측

4.1.1 정규방정식

- 비용함수를 최소화하는 θ 값을 찾기 위한 해석적인 방법 (수학공식) : 정규방정식

$$\hat{\theta} = \left(X^T \cdot X\right)^{-1} \cdot X^T \cdot y$$

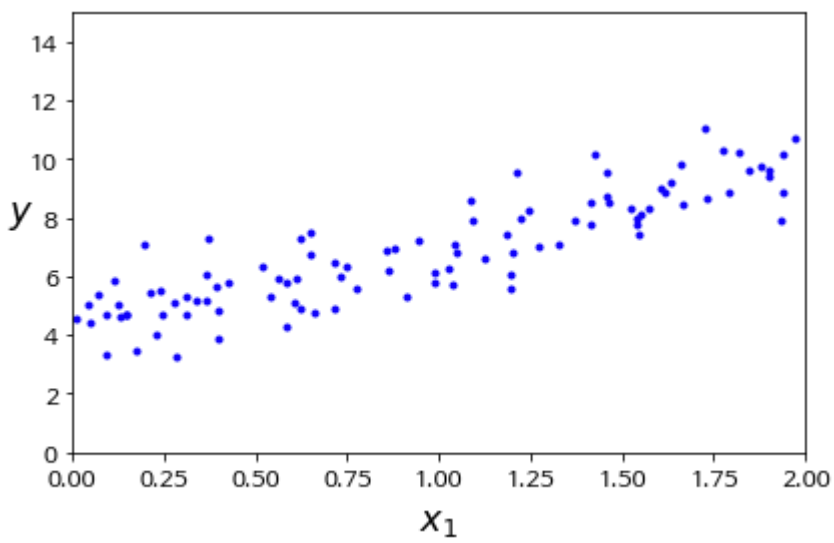
$\hat{\theta}$: 비용함수를 최소화시키는 θ값 벡터

y : $y^{(1)}$부터 $y^{(m)}$까지 포함하는 타깃 벡터

- 테스트 위해 임의의 선형 데이터 생성

In [58]:
```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

In [59]:
```
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])

plt.show()
```



◆ 정규방정식을 사용해 $\hat{\theta}$ 계산

- 역행렬 계산 : numpy 선형대수 모듈 (np.linalg)의 inv() 함수
- 행렬 곱셈 : dot() 함수 사용

In [60]:
```
X_b = np.c_[np.ones((100, 1)), X]  # 모든 샘플에 x0 = 1을 추가합니다.
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

- 값 확인 → 기대값 : θ_0=4, θ_1=3 (노이즈 때문에 정확하지 않음)

In [61]:
```
theta_best
```

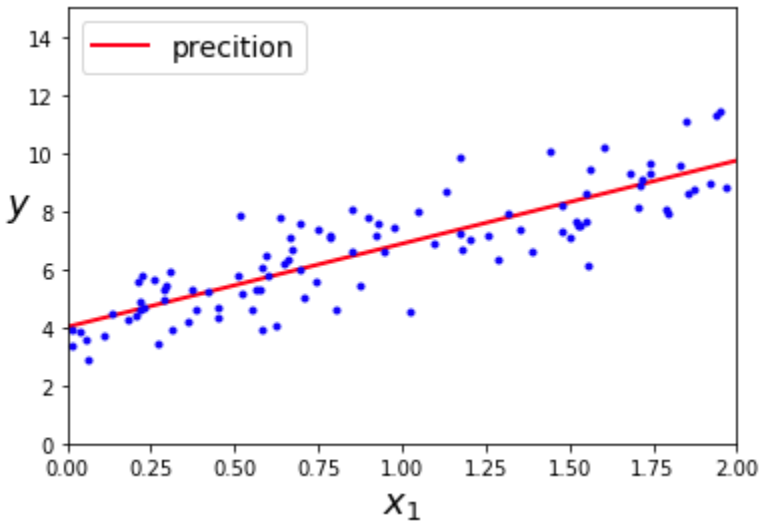
Out[61]:
```
array([[4.01285389],
       [2.86439883]])
```

- $\hat{\theta}$ 을 사용해 예측

In [62]:
```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]  # 모든 샘플에 x0 = 1을 추가
y_predict = X_new_b.dot(theta_best)
y_predict
```

Out[62]:
```
array([[4.01285389],
       [9.74165154]])
```

In [63]:
```
plt.plot(X_new, y_predict, "r-", linewidth=2, label="precition")
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([0, 2, 0, 15])
plt.show()
```



사이킷런의 LinearRegression 사용하는 방법

In [64]:
```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
```

Out[64]:
```
(array([4.01285389]), array([[2.86439883]]))
```

In [65]:
```
lin_reg.predict(X_new)
```

Out[65]:
```
array([[4.01285389],
       [9.74165154]])
```

4.1.2 계산 복잡도

- 정규방정식은 $(n+1)$x$(n+1)$ 크기가 되는 $X^T \cdot X$의 역행렬 계산 (n은 특성 수)

- 역행렬 계산 복잡도 : $O(n^{2.4}) \sim O(n^3)$
- → 특성 수가 2배로 늘어나면 계산 시간이 대략 5.3~8배로 증가

- 훈련 세트의 샘플 수에는 선형적으로 증가 $O(m)$
- 메모리 공간이 허락된다면 큰 훈련 세트도 효율적으로 처리 가능

- 선형 회귀 모델 예측 빠름.
- 예측 계산 복잡도는 샘플수와 특성수에 선형적
- → 예측하려는 샘플이 두배 증가, 걸리는 시간 두배 증가

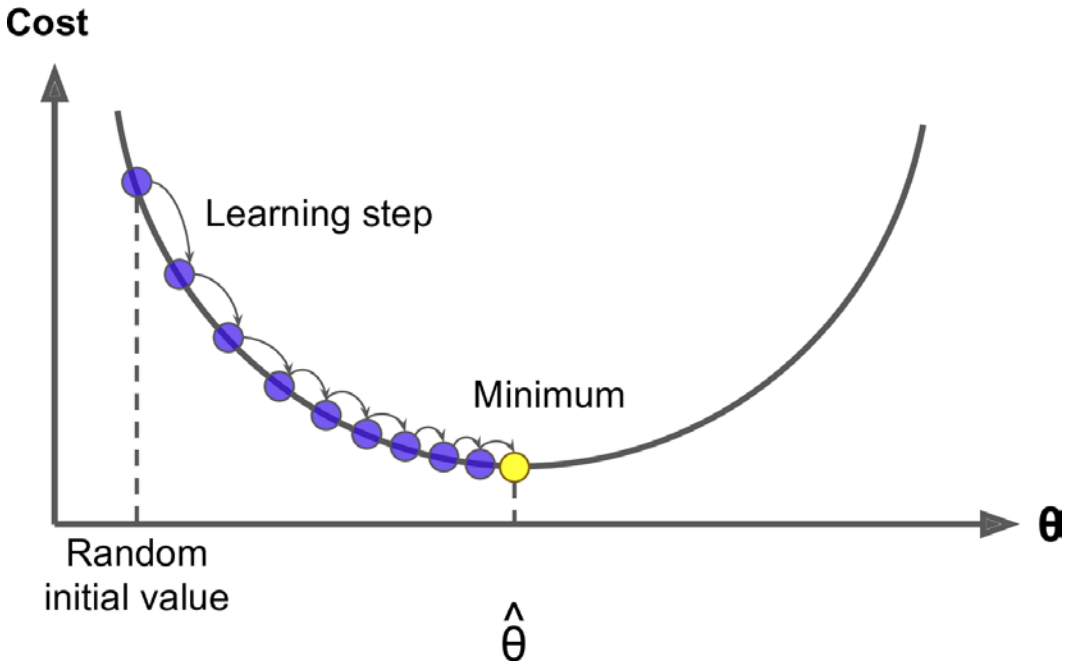
4.2 경사 하강법

- **또 다른 방법으로 선형 회귀 모델 훈련**

- **특성이 매우 많고 훈련 샘플이 너무 많아 메모리에 모두 담을 수 없을 때 적합**

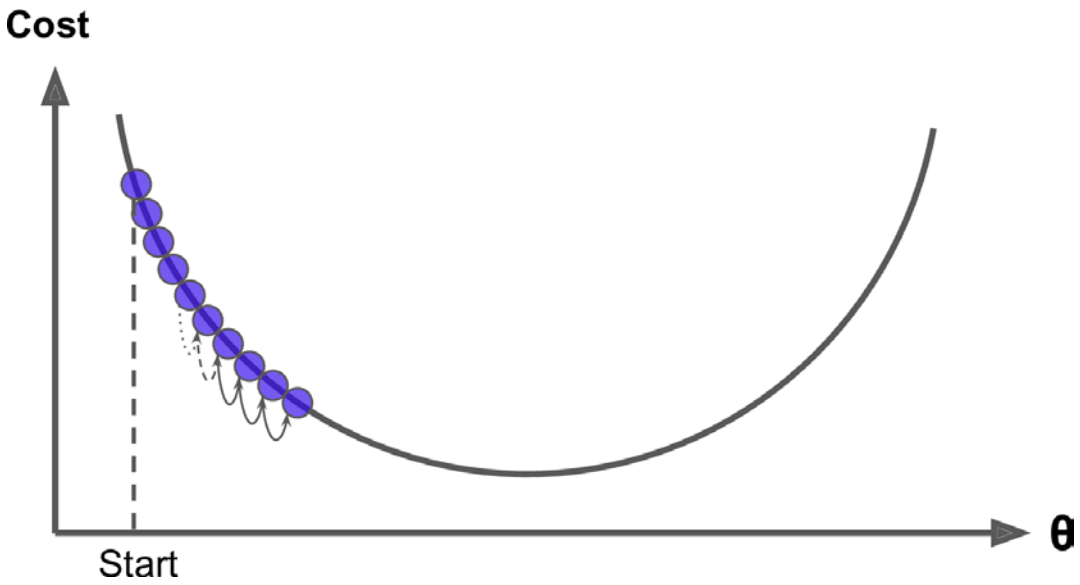
- **경사하강법(Gradient Descent)**
  - 여러 종류의 문제에서 최적의 해법을 찾을 수 있는 매우 일반적인 최적화 알고리즘
  - 기본 아이디어) 비용 함수를 최소화하기 위해 반복해서 파라미터 조정
  - 파라미터 벡터 θ 에 대해 비용 함수의 gradient를 계산하고, gradient가 감소하는 방향으로 반복적으로 θ를 수정
  - gradient=0 이면 최소값에 도달 완료

경사 하강법

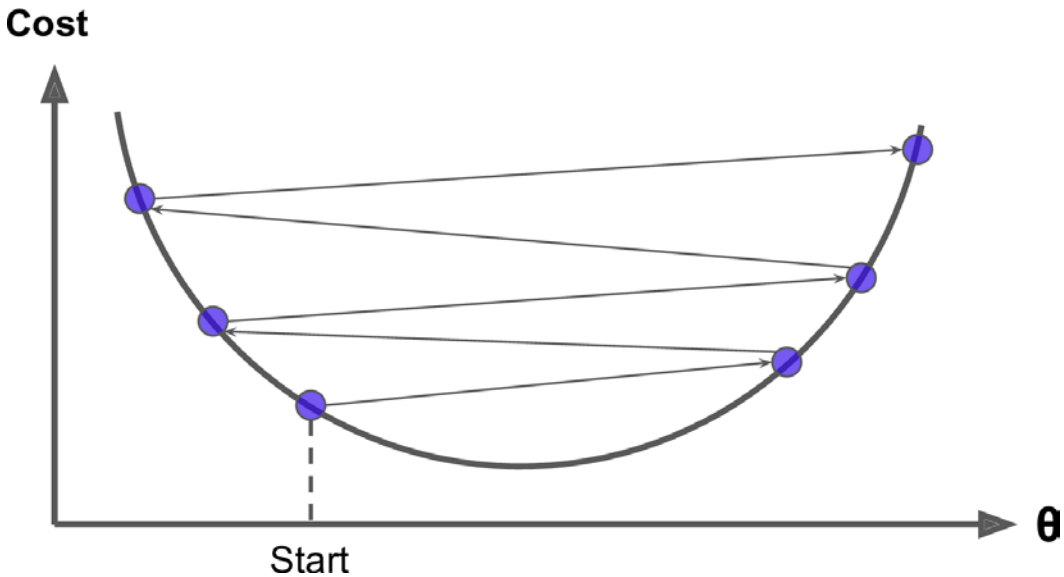


- Learning step의 크기는 학습률 하이퍼파라미터로 결정됨
- 학습률이 너무 작으면 알고리즘이 수렴하기 위해 반복을 많이 진행. 시간이 오래 걸림
- 학습률이 너무 크면 골짜기를 가로질러 반대편으로 건너뛰게 되어 이전보다 더 높은 곳으로 올라갈지도...

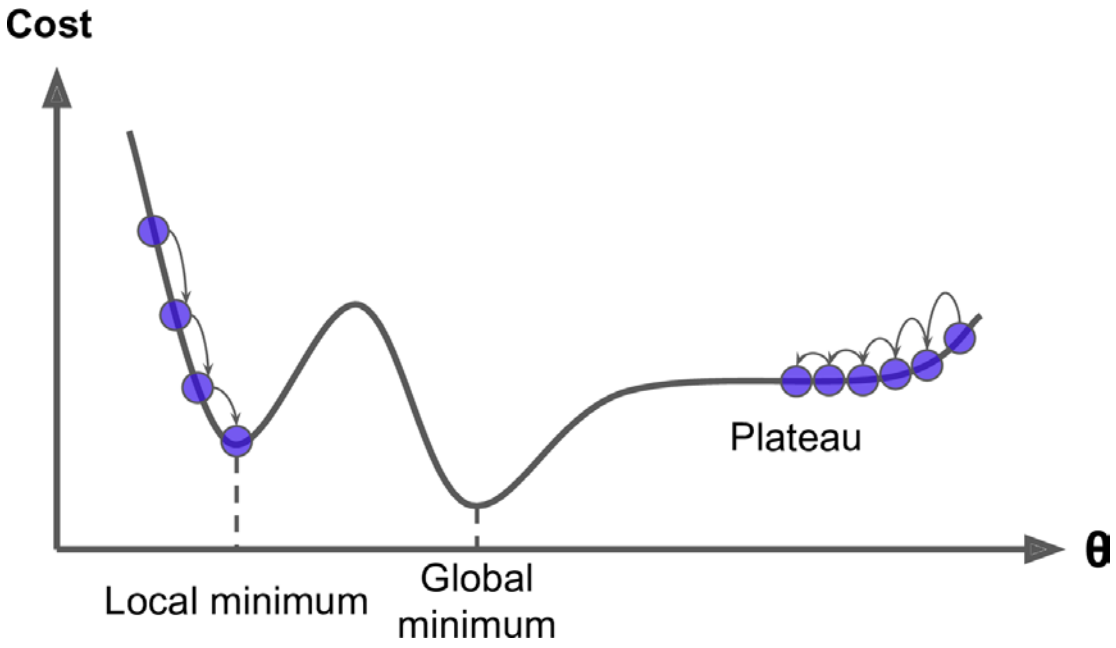
- 학습률이 너무 작을 때



- 학습률이 너무 클 때

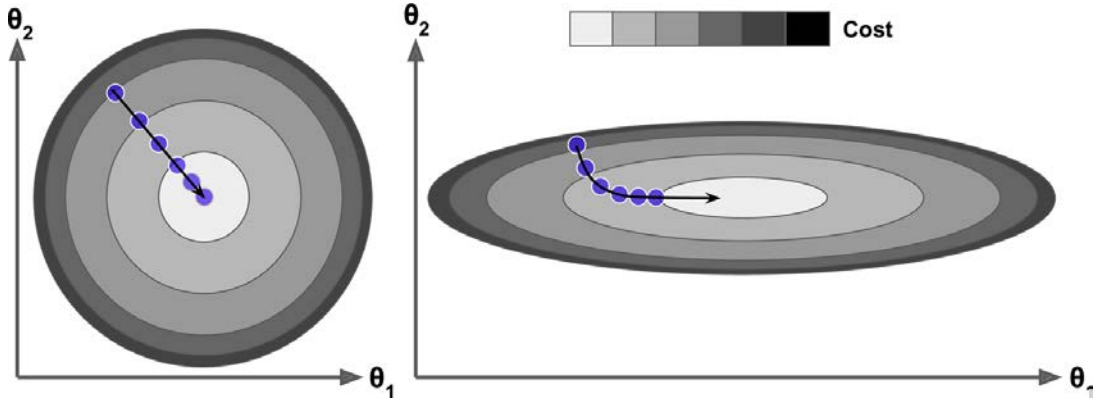


- **모든 비용 함수가 매끈한 그릇 같지는 않음**
  - 패인 곳, 산마루, 평지 등 특이한 지형
  - 최솟값으로 수렴하기 매우 어려움
- **경사 하강법의 문제점 예)**
  - 왼쪽에서 시작 → 전역 최솟값보다 덜 좋은 지역 최솟값에 수렴
  - 오른쪽에서 시작 → 평탄한 지역을 지나기 위해 시간이 오래 걸리고, 일찍 멈추게 됨



- **선형 회귀를 위한 MSE 비용 함수**
  - 곡선에서 어떤 두 점을 선택해 선을 그어도 곡선을 가로지르지 않는 볼록 함수 (convex function)
  - 지역 최솟값이 없고, 하나의 전역 최솟값만 있음
  - 연속된 함수. 기울기가 갑자기 변하지 않음

  → 경사 하강법이 전역 최솟값에 가깝게 접근할 수 있음을 보장
  (학습률이 너무 높지 않고, 충분한 시간이 주어진다면...)
- **특성의 스케일이 매우 다르면 길쭉한 모양**
  - (왼쪽) 특성1=특성2 스케일 : 최솟값으로 곧장 진행. 빠르게 도달
    (오른쪽) 특성1<특성2 스케일 : 시간 오래 걸림



  - scaling 필요 : StandardScaler() 함수 사용

- **모델 훈련이란...**
  - (훈련 세트에서) 비용 함수를 최소화하는 모델 파라미터의 조합을 찾음
  - 모델이 가진 파라미터가 많을수록 공간의 차원을 커지고 검색 어려워짐

4.2.1 배치 경사 하강법

- **경사 하강법 구현**
  - 각 모델 파라미터 θ_j에 대해 비용 함수의 gradient 계산
  - θ_j가 조금 변경될 때 비용 함수가 얼마나 바뀌는지 계산
  - 편도함수(partial derivative)

- **파라미터 θ_j에 대한 비용 함수의 편도 함수**
  - 모든 차원에 대해 기울기 확인

$$\frac{\partial}{\partial \theta_j} \mathrm{MSE}(\mathbf{X}, \mathrm{h}_\theta) = \frac{2}{M} \sum_{i=1}^{m} \left(\theta^T \cdot \boldsymbol{X}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

  - 각각 계산하는 대신 한꺼번에 계산
  - 그래디언트 벡터 : 비용 함수의 편도함수를 모두 담음

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

매 경사 하강법 스텝에서 전체 훈련 세트 X에 대해 계산
→ 배치 경사 하강법
매우 큰 훈련세트에서 아주 느림
하지만, 특성 수에 민감하지 않음

다음에 내려가는 스텝 크기 결정

- θ 에서 $\nabla_\theta MSE$를 뺌
- 학습률 η 사용 → 이전의 그래디언트 벡터 * η
- 경사하강법의 스텝

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

- 알고리즘 구현

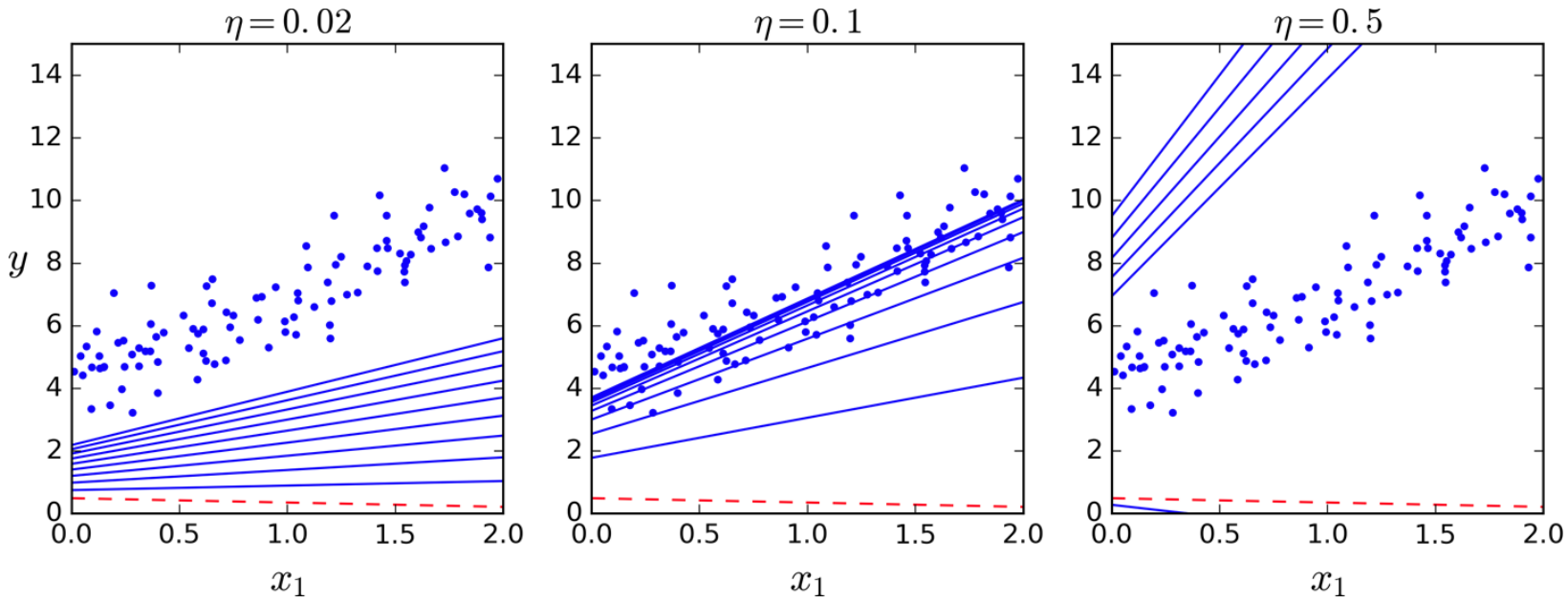
```
In [66]: eta = 0.1
         n_iterations = 1000
         m = 100
         theta = np.random.randn(2,1)

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

In [67]: theta

Out[67]: array([[4.01285389],
                [2.86439883]])
```

정규방정식으로 찾은 것과 동일

```
In [68]: X_new_b.dot(theta)

Out[68]: array([[4.01285389],
                [9.74165154]])
```

학습률 η 변경

- 세 가지 다른 학습률 사용하여 진행한 경사 하강법 스텝 처음 10개 (점선은 시작점)



학습률이 너무 낮음
시간 오래 걸림

학습률 적당
반복 몇 번 만에 최적점 수렴

학습률이 너무 높음
알고리즘이 이리저리 널뛰면서
스텝마다 최적점에서 점점 더
멀어져 발산

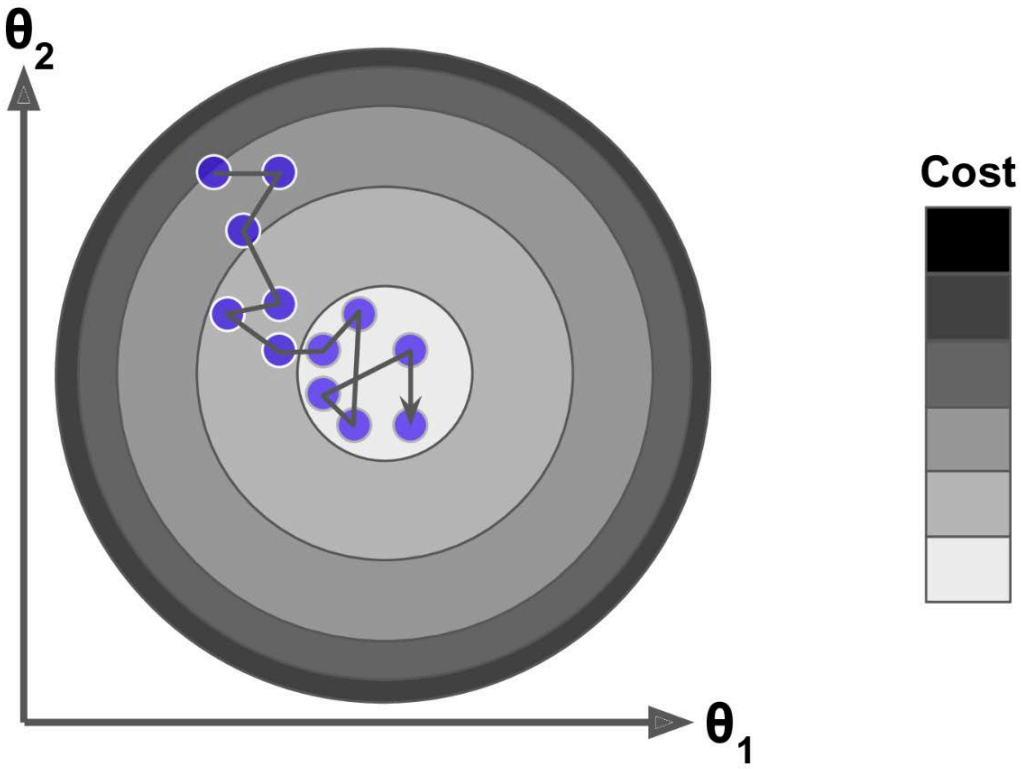
BGD 구현

In [69]:
```
theta_path_bgd = []

def plot_gradient_descent(theta, eta, theta_path=None):
    m = len(X_b)
    plt.plot(X, y, "b.")
    n_iterations = 1000
    for iteration in range(n_iterations):
        if iteration < 10:
            y_predict = X_new_b.dot(theta)
            style = "b-" if iteration > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
        theta = theta - eta * gradients
        if theta_path is not None:
            theta_path.append(theta)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 2, 0, 15])
    plt.title(r"$\eta = {}$".format(eta), fontsize=16)
```

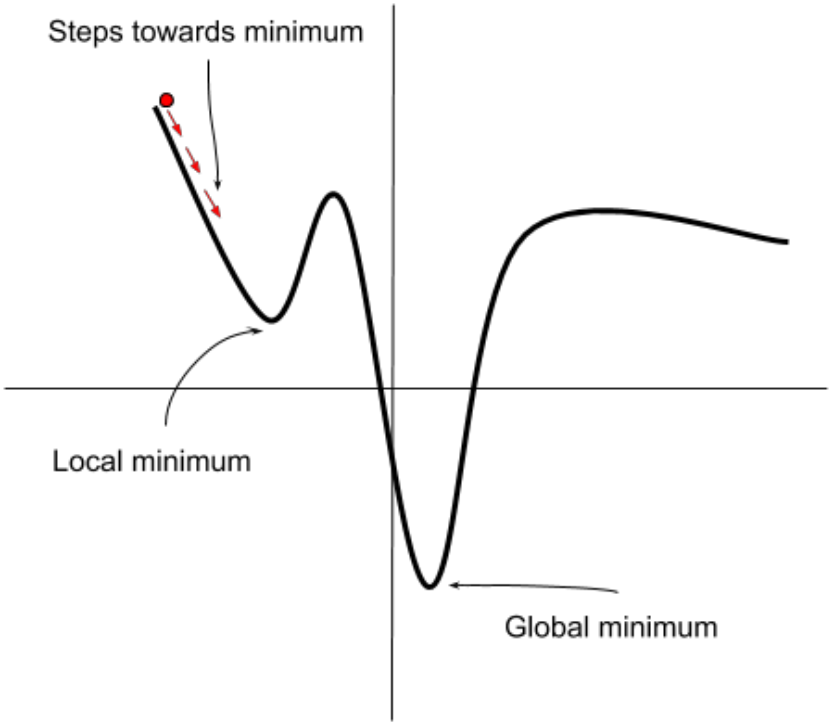
In [70]:
```
np.random.seed(42)
theta = np.random.randn(2,1)  # random initialization

plt.figure(figsize=(10,4))
plt.subplot(131); plot_gradient_descent(theta, eta=0.02)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(132); plot_gradient_descent(theta, eta=0.1, theta_path=theta_path_bgd)
plt.subplot(133); plot_gradient_descent(theta, eta=0.5)

plt.show()
```

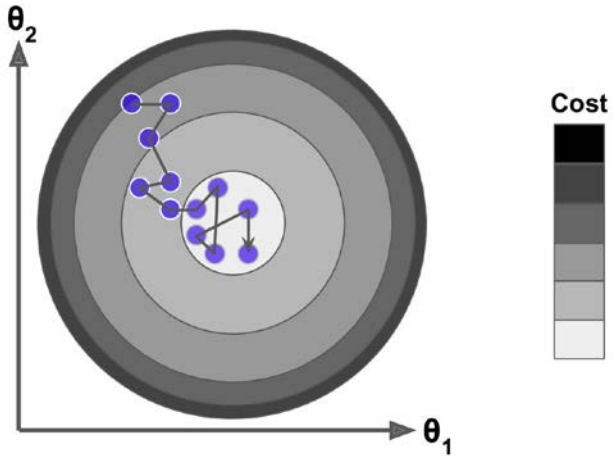
4.2.2 확률적 경사 하강법 (SGD)

- SGD(Stochastic gradient descent)는 매 step에서 한 개의 샘플을 무작위로 선택하고 하나의 샘플에 대한 gradient를 계산
- 매 반복에서 하나의 샘플만 있으면 되므로 매우 큰 훈련세트도 빠르게 훈련시킬 수 있음
- 단점) 확률적이기 때문에 batch gradient descent 보다 불안정



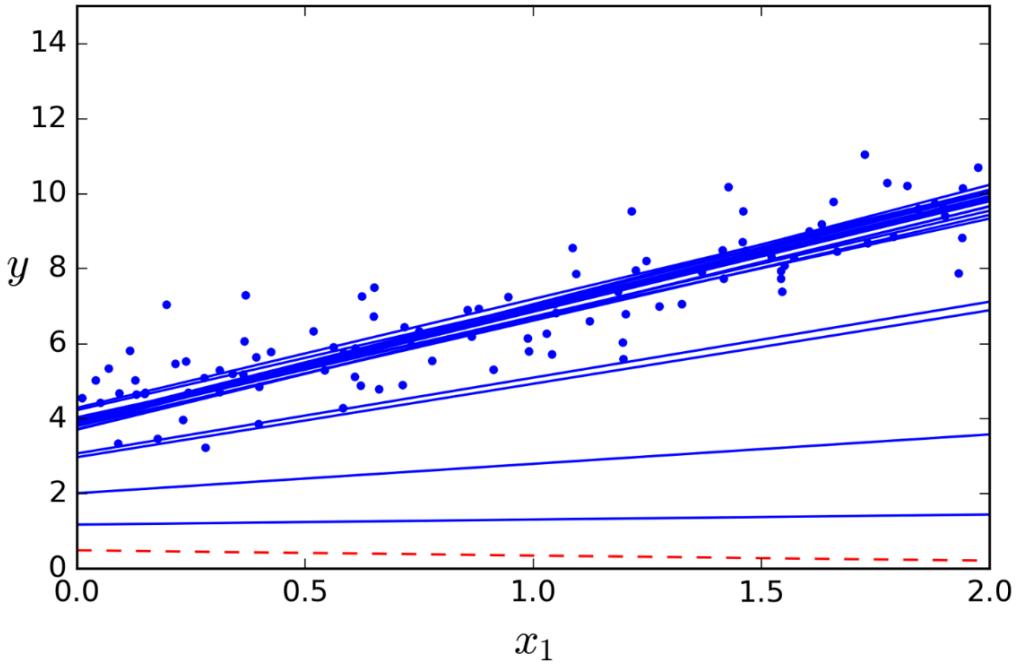
- 만일 cost function이 오른쪽 그림과 같이 매우 불규칙하다면 알고리즘이 지역 최소값을 건너뛸 수 있도록 도와주므로 SGD가 BGD보다 전역 최솟값을 찾을 가능성이 더 높음
- 무작위성은 지역 최소값을 건너뛸 수 있어 좋지만 전역 최소값에 정확히 다다르지 못한다는 단점





- 딜레마를 해결하기 위해 학습률을 점진적으로 감소시키는 방법을 사용
- 매 반복에서 학습률을 결정하는 함수 : **learning (rate) schedule**

- 학습률이 너무 빨리 줄어들면 지역 최솟값에 빠지거나 최솟값까지 가는 도중 멈출 수 있음
- 반면 너무 천천히 줄어들면 최솟값 주변을 오랫동안 머물거나 훈련을 일찍 중지시켜 지역 최솟값에 머무르게 할 수 있음



SGD 구현

In [71]:
```
theta_path_sgd = []
m = len(X_b)
np.random.seed(42)
```

In [72]:
```
n_epochs = 50
t0, t1 = 5, 50  # 학습 스케줄 하이퍼파라미터 learning schedule hyperparameters

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

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

for epoch in range(n_epochs):
    for i in range(m):
        if epoch == 0 and i < 20:
            y_predict = X_new_b.dot(theta)
            style = "b-" if i > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        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
        theta_path_sgd.append(theta)

plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])

plt.show()
```

확인 및 비교

In [73]:
```
theta
```

Out[73]:
```
array([[4.03857488],
       [2.87420599]])
```

In [74]:
```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=5, penalty=None, eta0=0.1, random_state=42)
sgd_reg.fit(X, y.ravel())
```

```
C:₩ProgramData₩Anaconda3₩lib₩site-packages₩sklearn₩linear_model₩stochastic_gradient
on reached before convergence. Consider increasing max_iter to improve the fit.
  ConvergenceWarning)
```

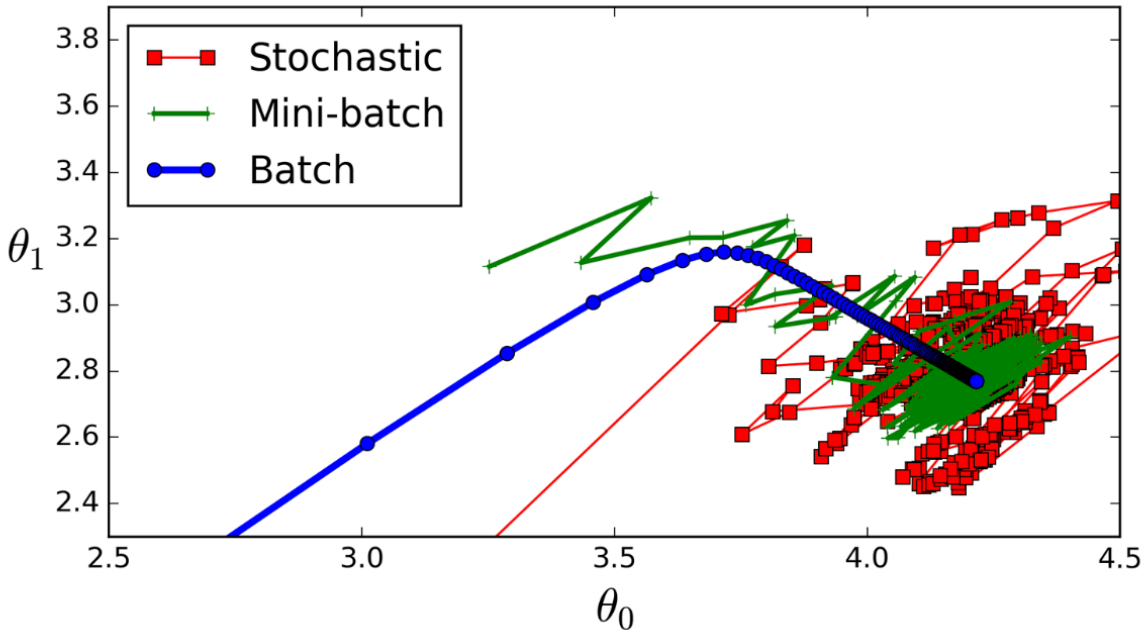
Out[74]:
```
SGDRegressor(alpha=0.0001, average=False, early_stopping=False, epsilon=0.1,
             eta0=0.1, fit_intercept=True, l1_ratio=0.15,
             learning_rate='invscaling', loss='squared_loss', max_iter=5,
             n_iter_no_change=5, penalty=None, power_t=0.25, random_state=42,
             shuffle=True, tol=0.001, validation_fraction=0.1, verbose=0,
             warm_start=False)
```

In [75]:
```
sgd_reg.intercept_, sgd_reg.coef_
```

Out[75]:
```
(array([4.01378262]), array([2.92987705]))
```

4.2.3 미니배치 경사 하강법

- 각 step에서 전체 훈련 세트(batch)나 하나의 샘플(stochastic)을 기반으로 gradient를 계산하는 것이 아니라, 임의의 작은 sample set (mini-batch)에 대해 gradient를 계산



미니배치 구현

In [76]:
```
theta_path_mgd = []

n_iterations = 50
minibatch_size = 20

np.random.seed(42)
theta = np.random.randn(2,1)  # 무작위 초기화

t0, t1 = 200, 1000
def learning_schedule(t):
    return t0 / (t + t1)

t = 0
for epoch in range(n_iterations):
    shuffled_indices = np.random.permutation(m)
    X_b_shuffled = X_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]
    for i in range(0, m, minibatch_size):
        t += 1
        xi = X_b_shuffled[i:i+minibatch_size]
        yi = y_shuffled[i:i+minibatch_size]
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(t)
        theta = theta - eta * gradients
        theta_path_mgd.append(theta)
```

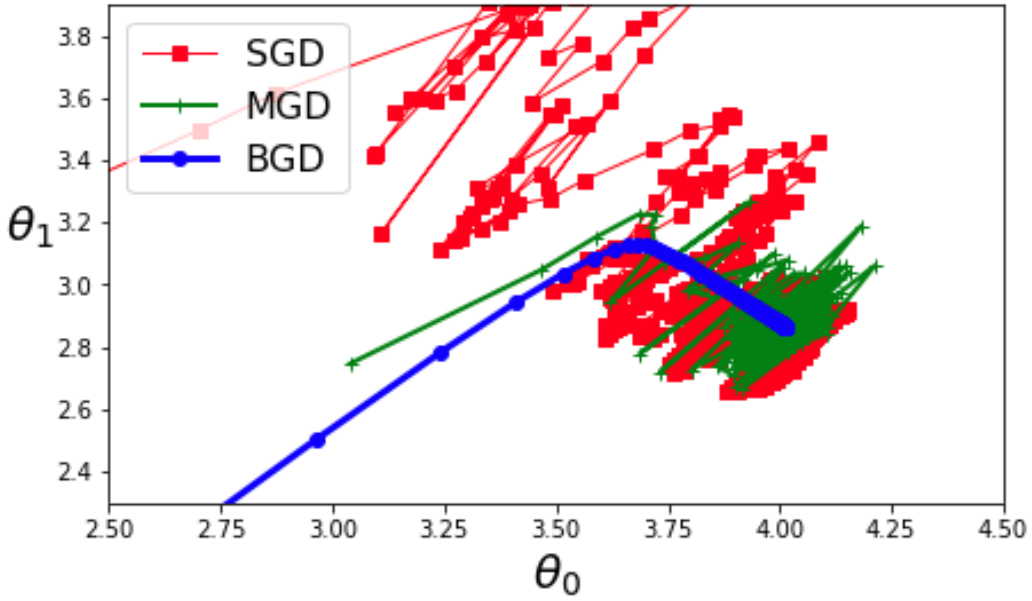
In [77]:
```
theta
```

Out[77]:
```
array([[3.9281083 ],
       [2.74863716]])
```

BGD / SGD / MGD 비교

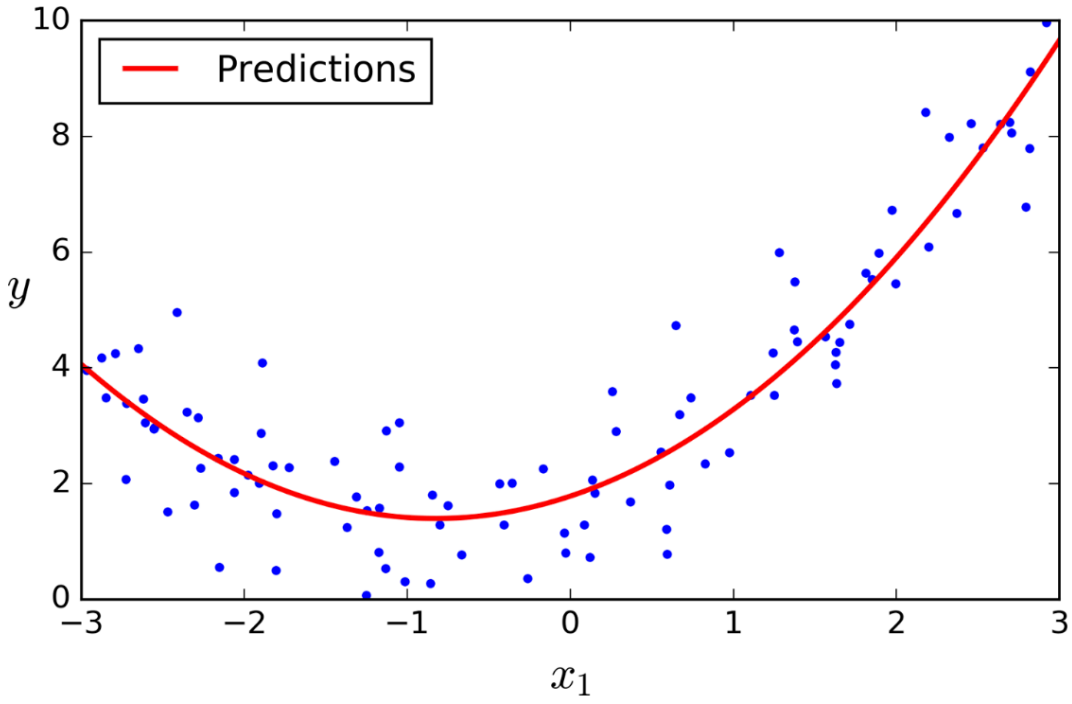
In [78]:
```
theta_path_bgd = np.array(theta_path_bgd)
theta_path_sgd = np.array(theta_path_sgd)
theta_path_mgd = np.array(theta_path_mgd)
```

In [79]:
```
plt.figure(figsize=(7,4))
plt.plot(theta_path_sgd[:, 0], theta_path_sgd[:, 1], "r-s", linewidth=1, label="SGD")
plt.plot(theta_path_mgd[:, 0], theta_path_mgd[:, 1], "g-+", linewidth=2, label="MGD")
plt.plot(theta_path_bgd[:, 0], theta_path_bgd[:, 1], "b-o", linewidth=3, label="BGD")
plt.legend(loc="upper left", fontsize=16)
plt.xlabel(r"$₩theta_0$", fontsize=20)
plt.ylabel(r"$₩theta_1$   ", fontsize=20, rotation=0)
plt.axis([2.5, 4.5, 2.3, 3.9])

plt.show()
```



4.3 다항 회귀 (Polynomial Regression)

- 비선형 데이터를 학습하는데도 선형 모델 사용 가능
- 다항 회귀(Polynomial Regression)
  - 각 특성의 거듭제곱을 새로운 특성으로 추가
  - 확장된 특성을 포함한 data set에 선형 모델을 훈련

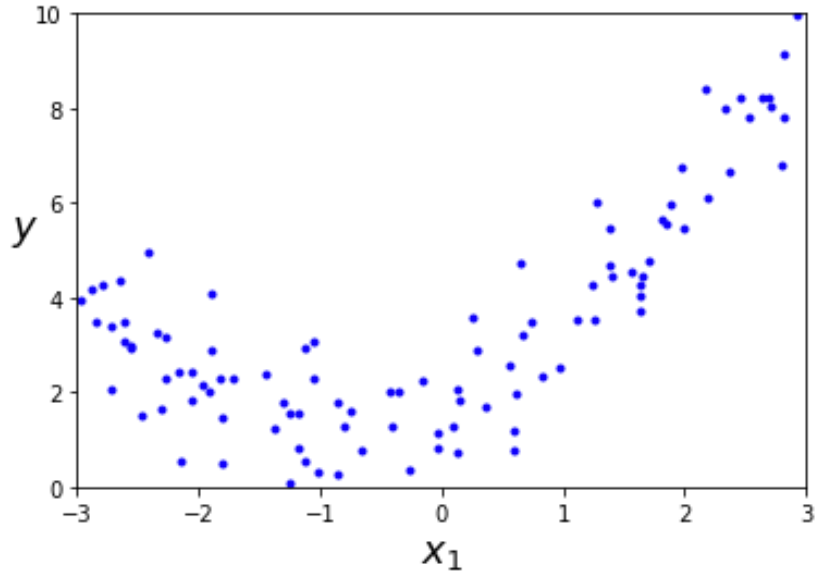


간단한 비선형 데이터 생성

In [80]:
```
import numpy as np
import numpy.random as rnd

np.random.seed(42)
```

In [81]:
```
# 간단한 2차 방정식으로 비선형 데이터 생성 (약간의 노이즈 포함)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

In [82]:
```
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])

plt.show()
```



다항회귀 구현

- 훈련 데이터 변환 (새로운 특성 추가)

In [83]:
```
# 사이킷런의 PolynomialFeatures 사용하여 변환
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0]
```

Out [83]:
```
array([-0.75275929])
```

In [84]:
```
X_poly[0]  # 원래 특성 X 와 특성의 제곱 (추가된 특성)
```

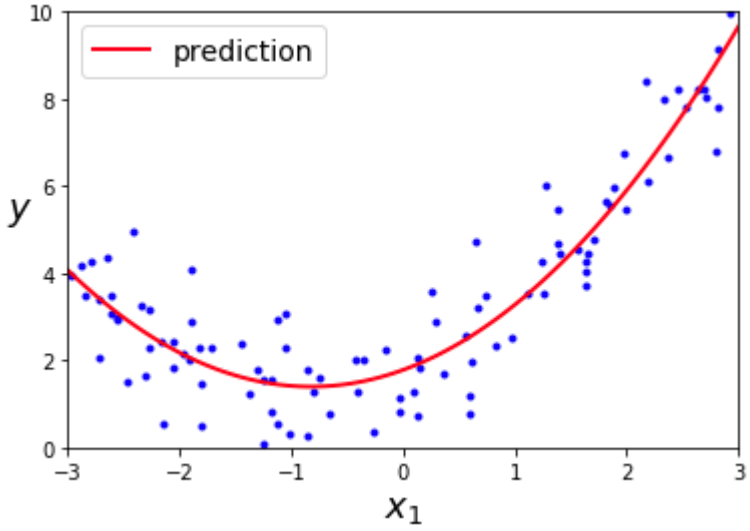
Out [84]:
```
array([-0.75275929,  0.56664654])
```

- 확장된 훈련 데이터에 선형회귀 적용

In [85]:
```
# 확장된 훈련 데이터에 LinearRegression 적용
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

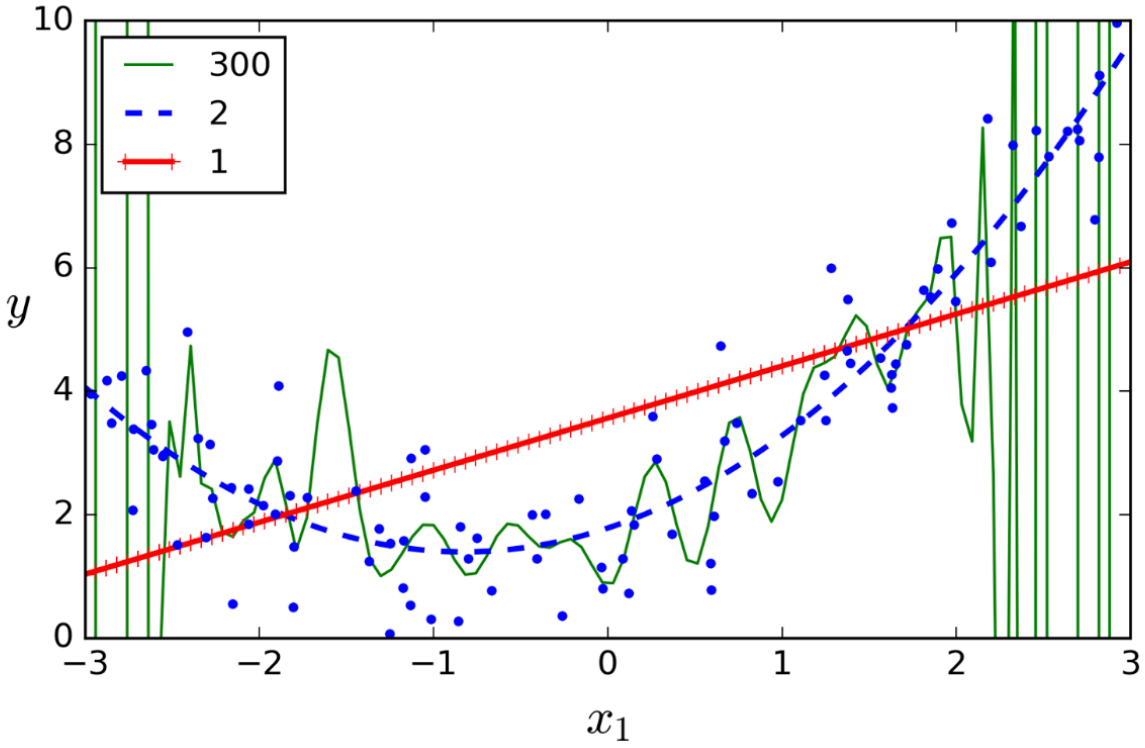
Out [85]:
```
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

In [86]:
```
X_new=np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)
plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="prediction")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([-3, 3, 0, 10])

plt.show()
```



다항 회귀 모델의 과대적합

- 고차 다항 회귀를 적용하면 보통 선형회귀보다 훨씬 더 training data에 잘 맞게 model을 구성하려 할 것임
- 1차, 2차, 300차 다항 회귀 모델을 이전(2차) training data에 적용시킨 결과



- 선형 모델은 underfitting, 300차 회귀 모델은 overfitting이 나타남

구현

In [87]:

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

for style, width, degree in (("g-", 1, 300), ("b--", 2, 2), ("r-+", 2, 1)):
    polybig_features = PolynomialFeatures(degree=degree, include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = Pipeline([
            ("poly_features", polybig_features),
            ("std_scaler", std_scaler),
            ("lin_reg", lin_reg),
        ])
    polynomial_regression.fit(X, y)
    y_newbig = polynomial_regression.predict(X_new)
    plt.plot(X_new, y_newbig, style, label=str(degree), linewidth=width)

plt.plot(X, y, "b.", linewidth=3)
plt.legend(loc="upper left")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])

plt.show()
```

4.4 학습 곡선

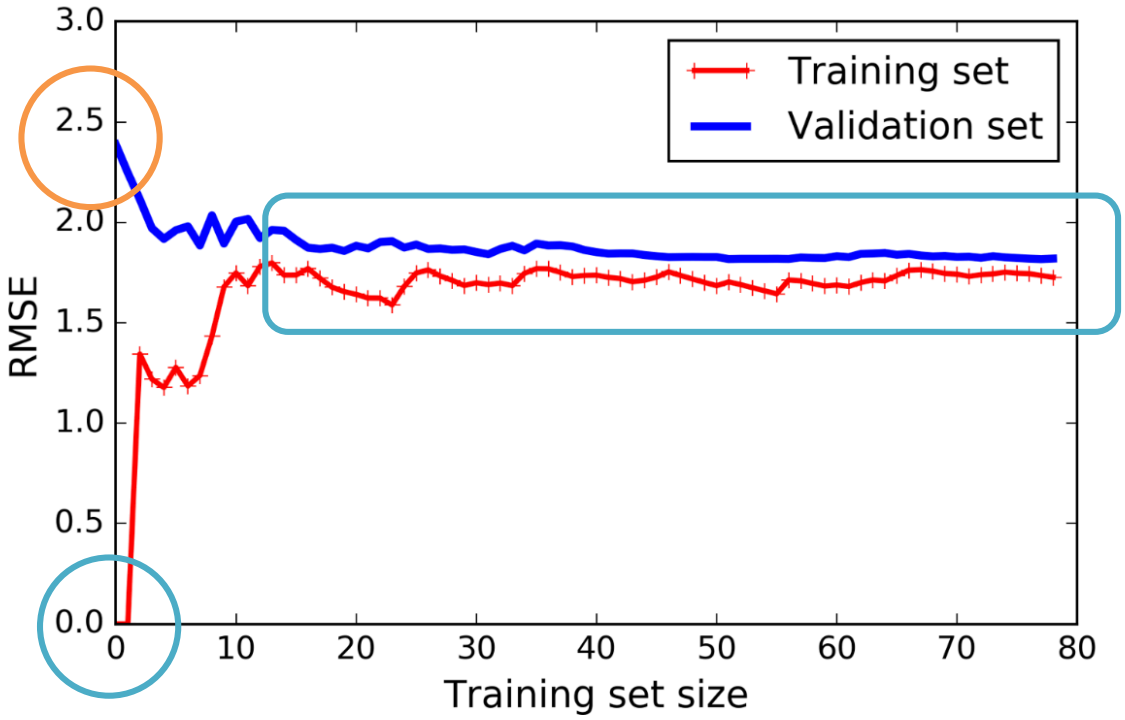
- 모델의 과대적합 / 과소적합 판단 : 교차검증 이용
- 훈련데이터에서 성능 좋음 / 교차검증 점수 나쁨 → 과대적합
- 둘 다 좋지 않음 → 과소적합

- 또 다른 방법 : 학습 곡선(그래프)
- 훈련 세트/검증 세트 모델 성능 → 훈련 세트 크기의 함수로 표현
- 훈련 세트에서 크기가 다른 서브 세트를 만들어 모델을 여러번 훈련

단순 선형 회귀 모델의 학습 곡선

- **underfitting model의 전형적인 모습**(simple linear regression)

- 두 곡선이 높은 오차에서 가까이 근접해 수평한 구간을 만든다.
- 훈련 샘플을 더 추가해도 효과 없음 → 더 복잡한 모델을 사용하거나 더 나은 특성 선택 필요



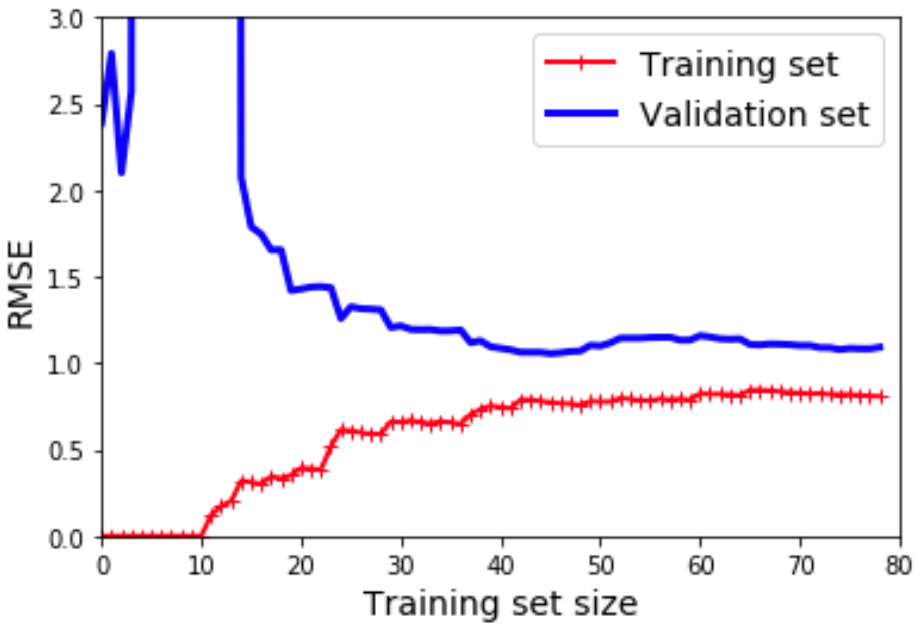
선형회귀 학습곡선 구현

In [90]:
```
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, random_state=10)
    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="Training set")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="Validation set")
    plt.legend(loc="upper right", fontsize=14)
    plt.xlabel("Training set size", fontsize=14)
    plt.ylabel("RMSE", fontsize=14)
```

In [91]:
```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
plt.axis([0, 80, 0, 3])

plt.show()
```

다항(10차)회귀 학습곡선 구현

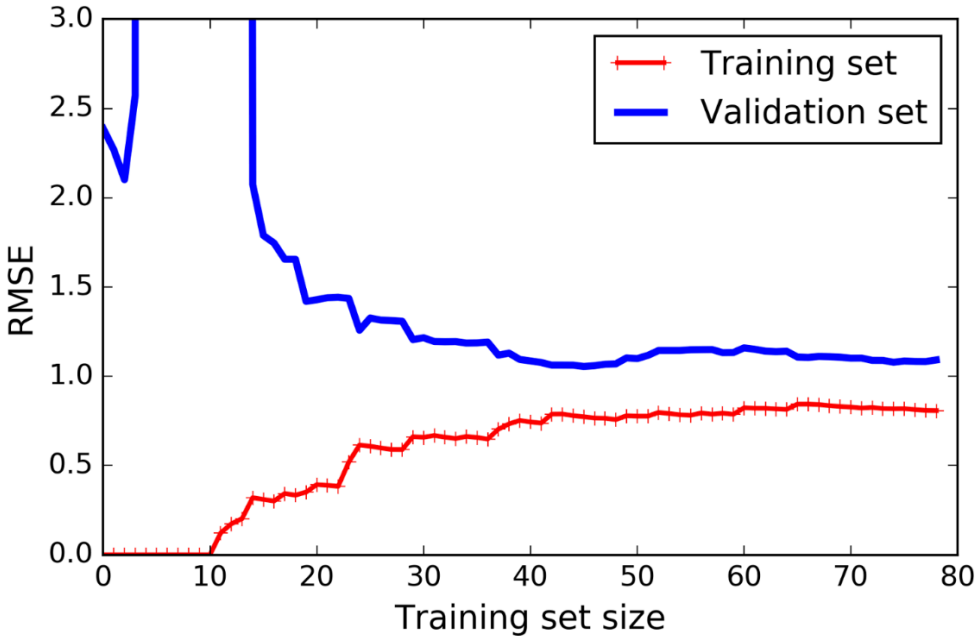
In [92]:
```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])

plt.show()
```



다항(10차) 회귀 모델의 학습 곡선

- training data의 오차가 앞선 일반 선형 회귀 모델에 비해 훨씬 낮음
- Training set size가 커져도 두 곡선 사이에 공간이 존재 (overfitting model의 특징)
- 더 큰 training set을 사용하면 두 곡선이 점점 가까워 짐

- **overfitting model을 개선하는 방법**
  - 검증 오차가 훈련 오차에 근접할 때 까지 더 많은 training data를 추가



편향/분산 트레이드오프

- **Bias (편향)**
  - 일반화 오차 중에서 편향은 잘못된 가정에 의해 발생
  - 예) 2차원 데이터를 선형으로 가정
  - bias가 큰 모델은 training data에 underfitting되기 쉬움

- **Variance (분산)**
  - training data에 있는 작은 변동에 모델이 과도하게 민감하기 때문에 나타남
  - 자유도가 높은 회귀 모델(ex. 고차 다항 회귀 모델)이 높은 분산을 가지기 쉬워 training data에 overfitting되는 경향

- **모델의 복잡도가 커지면 통상적으로 분산이 늘어나고 편향은 줄어든다, 반대로 모델의 복잡도가 줄어들면 편향이 커지고 분산이 작아진다.**

4.5 규제가 있는 선형 모델

- **Regularized Linear Models**
  - 과대 적합을 감소시키는 방법 : 모델 규제
  - 다항회귀 모델 규제 : 다항식의 차수를 감소 시키는 방법 사용
  - 선형회귀 모델 규제 : 모델의 가중치를 제한하는 방법 사용

  - 가중치 제한 방법 3가지 :
  - Ridge Regression, Lasso Regression, Elastic Net

4.5.1 릿지 회귀 (Ridge Regression)

- 티호노프 (러시아 수학자) 규제 (Tikhonov Regularization)

- 규제항 $\alpha \sum_{i=1}^{n} \theta_1^2$ 비용함수에 추가
- 모델의 가중치가 가능한 작게 유지되도록 함

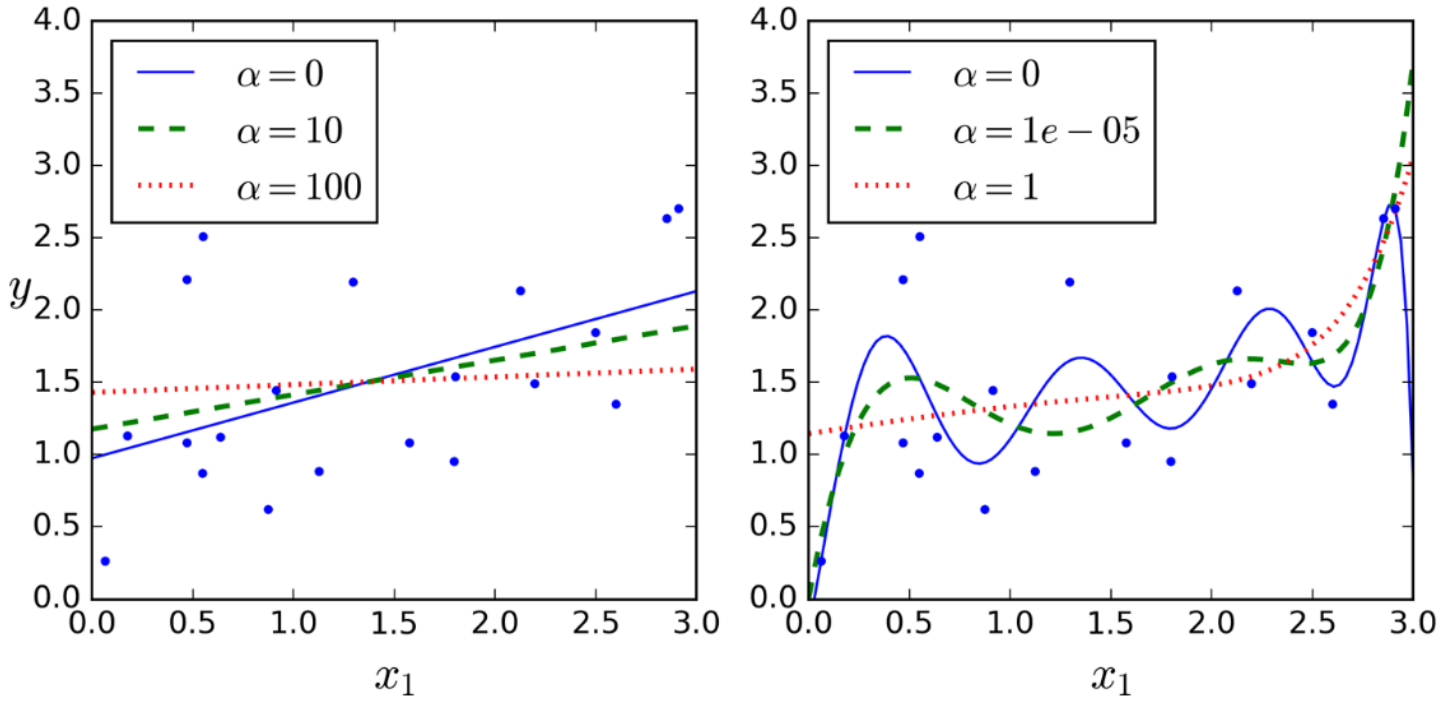
- 규제항은 훈련 기간에만 비용함수에 추가됨. (훈련이 끝나면 규제가 없는 성능 지표로 평가)
- 하이퍼파라미터인 α는 어느 정도로 모델을 규제할 지 결정
  - α=0 : 릿지회귀 = 선형회귀
  - α=매우큰값 : 모든 가중치가 거의 0에 가까워짐. 수평선

- Ridge Regression의 비용 함수

$$J(\theta) = MSE(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

- **몇 가지 α를 사용해 릿지 모델을 훈련시킨 결과**
  - (왼쪽) 평범한 릿지 모델 (선형예측)
  - (오른쪽) 데이터 확장 → 스케일 조정 → 릿지 모델 적용 (다항회귀)

  - α값이 증가할수록 직선에 가까워짐



릿지 회귀 구현

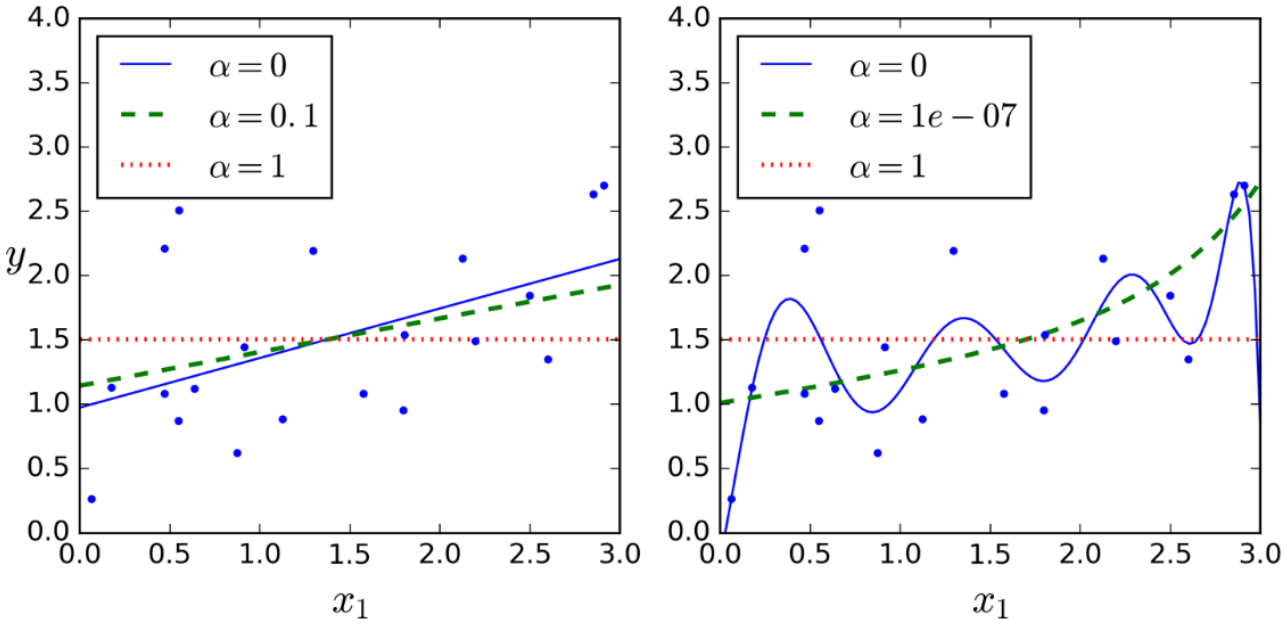
In [93]:
```
from sklearn.linear_model import Ridge

np.random.seed(42)
m = 20
X = 3 * np.random.rand(m, 1)
y = 1 + 0.5 * X + np.random.randn(m, 1) / 1.5
X_new = np.linspace(0, 3, 100).reshape(100, 1)

def plot_model(model_class, polynomial, alphas, **model_kargs):
    for alpha, style in zip(alphas, ("b-", "g--", "r:")):
        model = model_class(alpha, **model_kargs) if alpha > 0 else LinearRegression()
        if polynomial:
            model = Pipeline([
                    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
                    ("std_scaler", StandardScaler()),
                    ("regul_reg", model),
                ])
        model.fit(X, y)
        y_new_regul = model.predict(X_new)
        lw = 2 if alpha > 0 else 1
        plt.plot(X_new, y_new_regul, style, linewidth=lw, label=r"$\alpha = {}$".format(alpha))
    plt.plot(X, y, "b.", linewidth=3)
    plt.legend(loc="upper left", fontsize=15)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 3, 0, 4])

plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Ridge, polynomial=False, alphas=(0, 10, 100), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Ridge, polynomial=True, alphas=(0, 10**-5, 1), random_state=42)

plt.show()
```

4.5.2 라쏘 회귀

- **Lasso(Least Absolute Shrinkage and Selection Operator) Regression**
  - 릿지 회귀와 비슷하지만 조금 다른 비용함수 사용
  - Lasso Regression 비용 함수

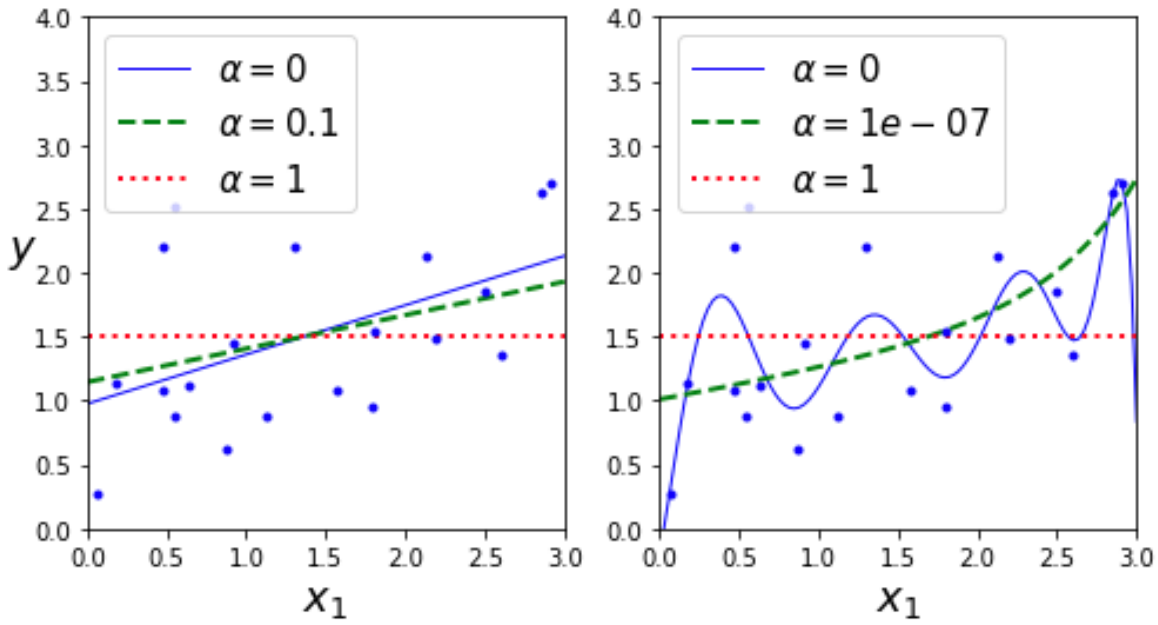
$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

  - 덜 중요한 가중치를 완전히 제거하려고 함



랏쏘 회귀 구현

In [94]:
```
from sklearn.linear_model import Lasso

plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Lasso, polynomial=False, alphas=(0, 0.1, 1), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Lasso, polynomial=True, alphas=(0, 10**-7, 1), tol=1, random_state=42)

plt.show()
```



Ridge vs. Lasso

- **예) 10,000 개의 변수를 가진 큰 data set이 존재**
- **그리고 이 변수들 중에는 서로 상관된 변수들이 존재**

  - 1) Ridge regression을 사용하면 모든 변수를 가지고 오면서 계수 값을 줄일 것이다. 하지만 문제는 1만개의 변수를 그대로 유지하므로 여전히 model이 복잡한 상태이다. 이는 모델 성능 저하에 영향을 미칠 수 있다.

  - 2) Lasso regression을 적용하면, 서로 correlate된 변수들 중에서 Lasso는 단 한개의 변수만 채택하고 다른 변수들의 계수를 0으로 바꿈. 이는 정보가 손실됨에 따라 정확성이 떨어지는 결과를 가져올 수 있다.

4.5.3 엘라스틱넷

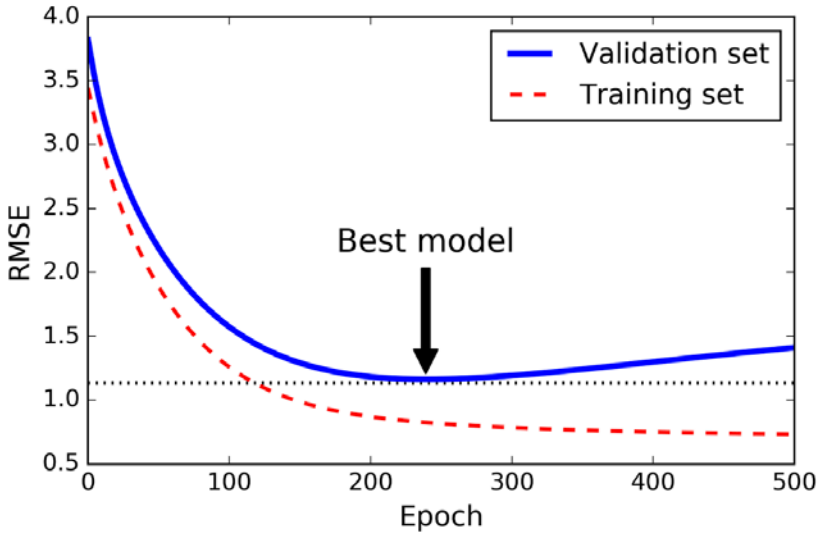
- **Elastic Net**
  - Ridge와 Lasso model을 절충한 모델
  - 규제항은 Ridge와 Lasso의 규제항을 더해서 사용
  - 혼합 비율(r)을 사용해 조절
  - r=0이면 Ridge, r=1이면 Lasso regression과 같아진다.

$$J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2$$

  - 대부분의 경우 **약간의 규제가 있는 model을 사용하는 것이 좋으므로** Ridge model 사용을 기본으로 하고, 실제로 쓰이는 특성이 몇 개 뿐이라고 의심되면 Lasso나 Elastic Net을 사용하는 것이 좋다.

4.5.4 조기 종료

- **Early Stopping**
  - 반복적인 학습 알고리즘을 규제하는 다른 방법
  - 검증 에러가 최솟값에 도달할 때 훈련을 중지시킴
  - 경사하강법으로 훈련시킨 고차원 다항 회귀 모델



  - Epoch 약 220정도에서 error가 가장 적게 나타나지만,
  - epoch가 증가하며 다시 error가 증가하는 overfitting 현상
  - Early stopping을 적용하면 epoch 약 220일 때, 훈련이 종료되고 최적의 파라미터를 반환

4.6 로지스틱 회귀

Logistic Regression

- sample이 특정 class에 속할 확률을 추정하는데 널리 사용
- 추정 확률이 50%가 넘으면 모델은 sample이 해당 class에 속한다고 예측

4.6.1 확률 추정

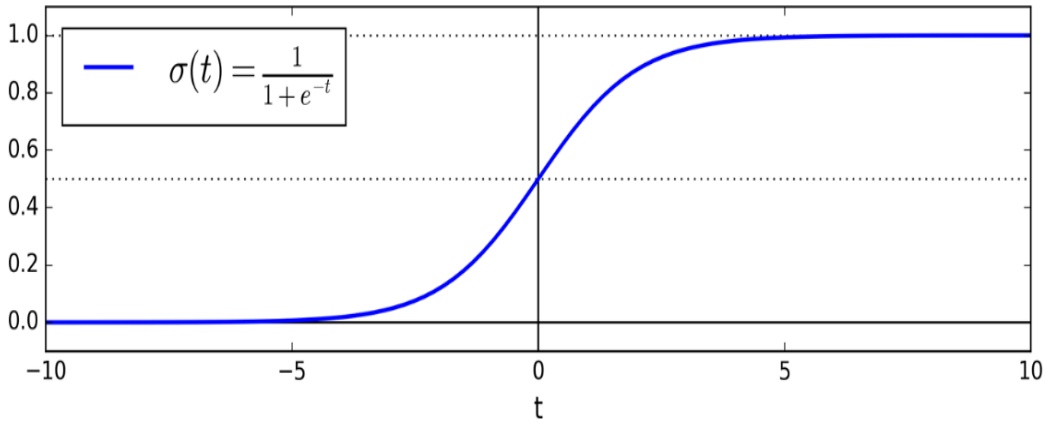
- (선형 회귀 모델처럼) 입력 특성의 가중치 합을 계산하고 bias를 더함
- 대신 선형 회귀처럼 결과를 바로 출력하지 않고 결과값의 logistic을 출력
- logistic regression model의 확률 추정 벡터 표현식

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

  - σ : logistic/logic이라 부르며 0과 1사이 값을 출력하는 sigmoid function

Logistic function

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$



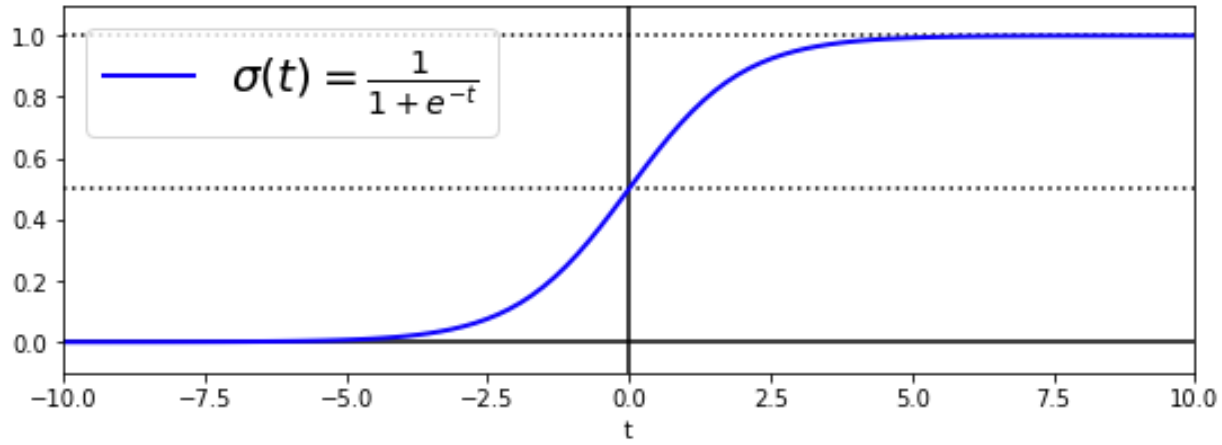
- sample x가 양성 클래스에 속할 확률 $\hat{p} = h_\theta(\boldsymbol{x})$ 를 추정하면 이에 대한 예측 $\hat{y}$ 를 쉽게 구할 수 있다.

$$\hat{y} = \begin{cases} 0 & \hat{p} < 0.5\text{일 경우} \\ 1 & \hat{p} \geq 0.5\text{일 경우} \end{cases}$$

  - t<0이면 $\sigma(t)$<0.5이고 t≥ 0이면 $\sigma(t)$ ≥0.5 이므로 logistic regression model은 $\theta^T \cdot \boldsymbol{x}$가 양수일 때 1(positive), 음수일 때 0(negative)라고 예측

로지스틱 회귀 구현

In [95]:
```
t = np.linspace(-10, 10, 100)
sig = 1 / (1 + np.exp(-t))
plt.figure(figsize=(9, 3))
plt.plot([-10, 10], [0, 0], "k-")
plt.plot([-10, 10], [0.5, 0.5], "k:")
plt.plot([-10, 10], [1, 1], "k:")
plt.plot([0, 0], [-1.1, 1.1], "k-")
plt.plot(t, sig, "b-", linewidth=2, label=r"$\sigma(t) = \frac{1}{1 + e^{-t}}$")
plt.xlabel("t")
plt.legend(loc="upper left", fontsize=20)
plt.axis([-10, 10, -0.1, 1.1])

plt.show()
```



4.6.2 훈련과 비용 함수

- **Logistic model의 훈련 목적**
  - 양성 샘플(y=1)에 대해서는 높은 확률을 추정,
  - 음성 샘플(y=0)에 대해서는 낮은 확률을 추정하는
  - 파라미터 벡터 θ를 찾는 것

  - 하나의 샘플에 대한 cost function

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

    - log function에 의해 양성 샘플을 0에 가까운 값으로 추정하면 cost가 매우 커지고, 음성 샘플을 1에 가까운 값으로 추정해도 cost가 매우 커짐

- **전체 training set에 대한 cost function**
  - 모든 훈련 샘플의 비용의 평균 : log loss

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

  - 위 cost function의 최솟값을 계산하는 해는 없다. 다만 convex function이므로 gradient descent를 이용하면 전역 최솟값을 찾을 수 있다.

4.6.3 결정 경계

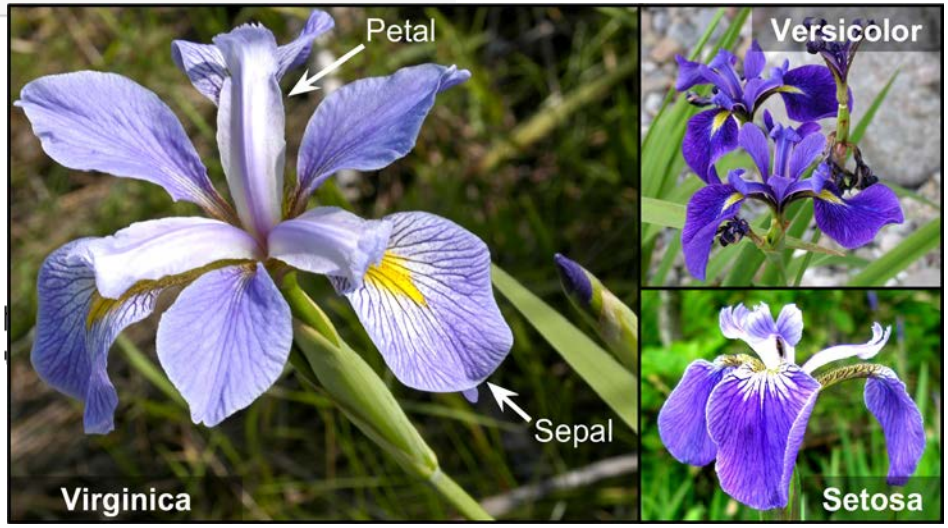
- 붓꽃 data set 분류 예
- 3개의 품종(Iris-Setosa, Iris-Versicolor, Iris-Virginica)에 속하는 붓꽃 150개의 꽃잎과 꽃받침의 너비와 길이를 포함

In [96]:
```
from sklearn import datasets
iris = datasets.load_iris()
list(iris.keys())
```

Out[96]:
```
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
```

In [97]:
```
print(iris.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of t
    :Number of Attributes: 4 numeric, predicti
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica
```



- logistic regression model을 이용해서 꽃잎의 너비가 0~3cm인 꽃에 대해 추정 확률 계산

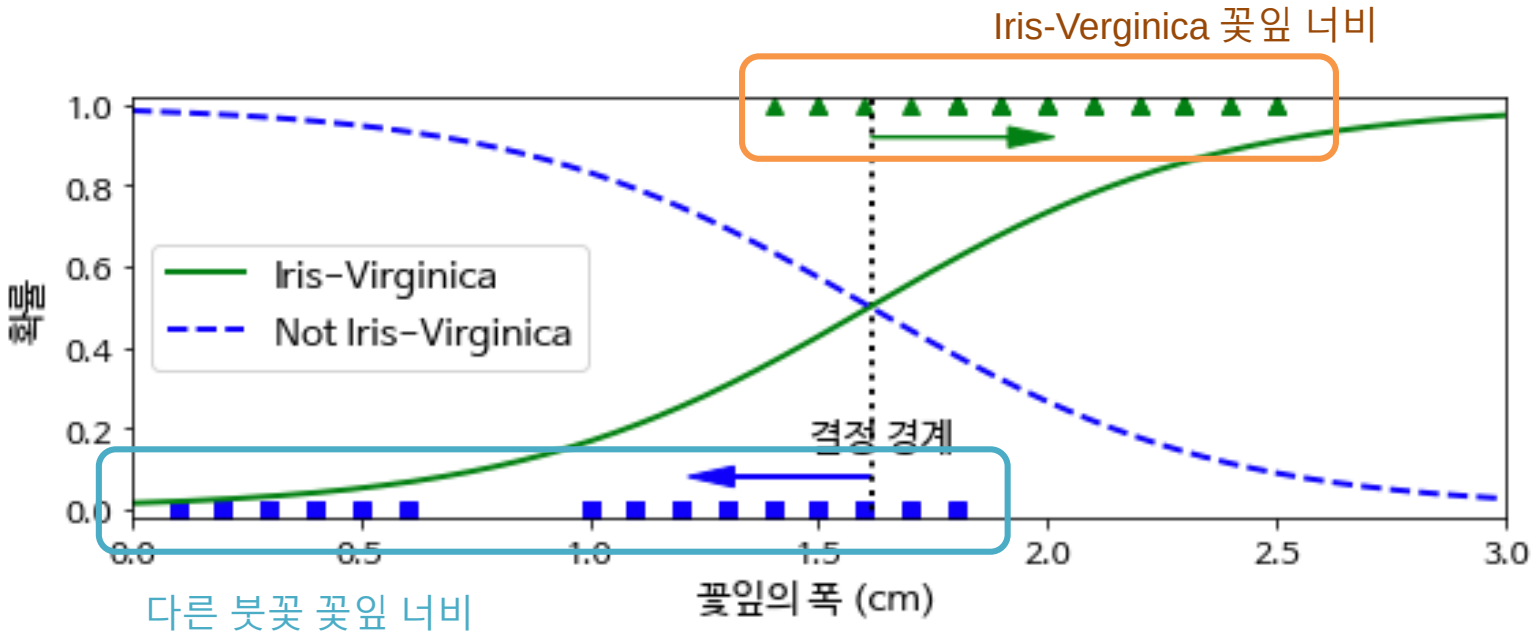
In [98]:
```
X = iris["data"][:, 3:]  # 꽃잎 넓이
y = (iris["target"] == 2).astype(np.int)  # Iris-Virginica이면 1 아니면 0
```

In [99]:
```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(solver='liblinear', random_state=42)
log_reg.fit(X, y)
```

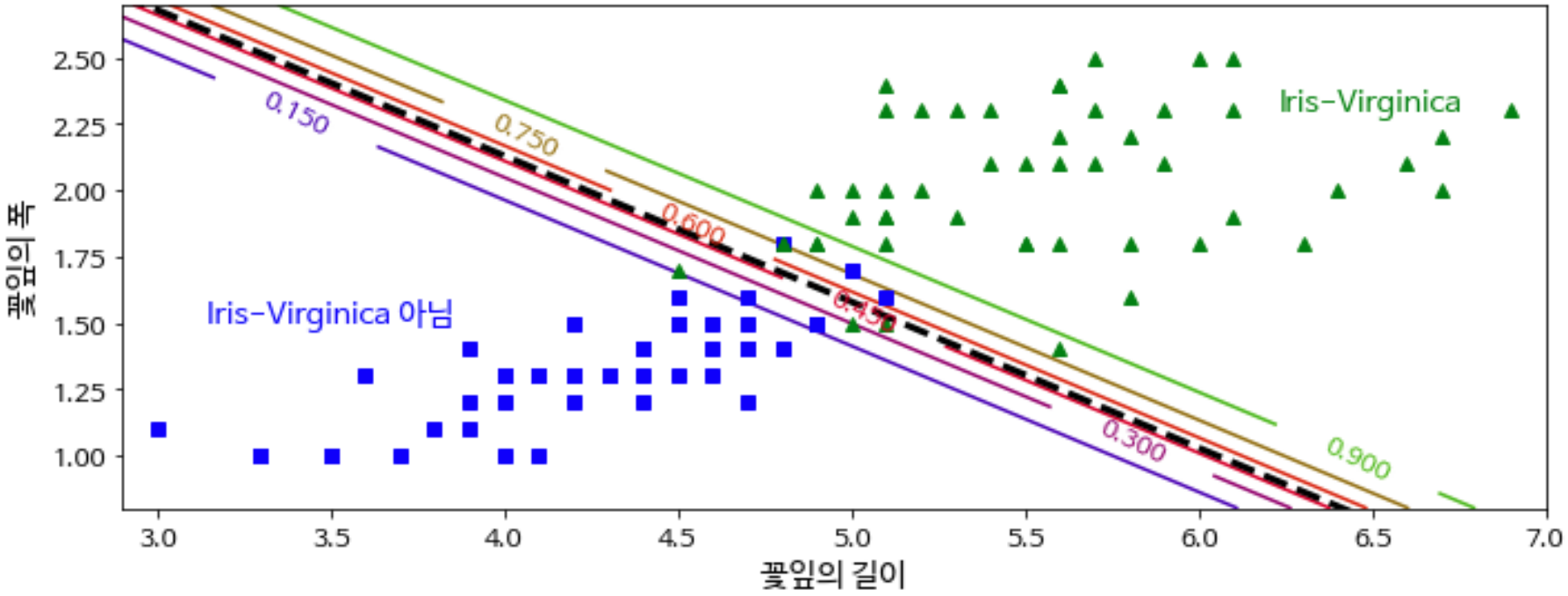
Out[99]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=42, solver='liblinear', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [100]:
```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0]

plt.figure(figsize=(8, 3))
plt.plot(X[y==0], y[y==0], "bs")
plt.plot(X[y==1], y[y==1], "g^")
plt.plot([decision_boundary, decision_boundary], [-1, 2], "k:", linewidth=2)
plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2, label="Not Iris-Virginica")
plt.text(decision_boundary+0.02, 0.15, "Decision boundary", fontsize=14, color="k", ha="center")
plt.arrow(decision_boundary, 0.08, -0.3, 0, head_width=0.05, head_length=0.1, fc='b', ec='b')
plt.arrow(decision_boundary, 0.92, 0.3, 0, head_width=0.05, head_length=0.1, fc='g', ec='g')
plt.xlabel("Petal width (cm)", fontsize=14)
plt.ylabel("Probability", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 3, -0.02, 1.02])

plt.show()
```

◆ 추정 확률과 결정 경계


Iris-Verginica 꽃잎 너비
1.0
0.8
0.6
0.4
0.2
0.0
확률
Iris-Virginica
Not Iris-Virginica
결정 경계
0.0
0.5
1.0
1.5
2.0
2.5
3.0
꽃잎의 폭 (cm)
다른 붓꽃 꽃잎 너비

- 꽃잎 너비와 길이, 두 개의 특성 그래프



- 점선은 모델이 50% 확률을 추정하는 지점으로 이 모델의 decision boundary

선형 결정 경계 구현

In [102]:
```
from sklearn.linear_model import LogisticRegression

X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.int)

log_reg = LogisticRegression(solver='liblinear', C=10**10, random_state=42)
log_reg.fit(X, y)

x0, x1 = np.meshgrid(
        np.linspace(2.9, 7, 500).reshape(-1, 1),
        np.linspace(0.8, 2.7, 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]

y_proba = log_reg.predict_proba(X_new)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs")
plt.plot(X[y==1, 0], X[y==1, 1], "g^")

zz = y_proba[:, 1].reshape(x0.shape)
contour = plt.contour(x0, x1, zz, cmap=plt.cm.brg)


left_right = np.array([2.9, 7])
boundary = -(log_reg.coef_[0][0] * left_right + log_reg.intercept_[0]) / log_reg.coef_[0][1]

plt.clabel(contour, inline=1, fontsize=12)
plt.plot(left_right, boundary, "k--", linewidth=3)
plt.text(3.5, 1.5, "Not Iris-Virginica", fontsize=14, color="b", ha="center")
plt.text(6.5, 2.3, "Iris-Virginica", fontsize=14, color="g", ha="center")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.axis([2.9, 7, 0.8, 2.7])

plt.show()
```

4.6.4 소프트맥스 회귀

- 직접 다중 클래스 지원하도록 일반화 : 다항 로지스틱 회귀

- 샘플 x를 각 클래스 k에 대한 점수 계산
- 점수에 softmax function을 적용하여 각 클래스의 확률을 추정
  - 한 번에 하나의 클래스만 예측
  - 다중 클래스, 다중 출력 안됨
    (예: 하나의 사진에서 여러 사람 얼굴 인식)

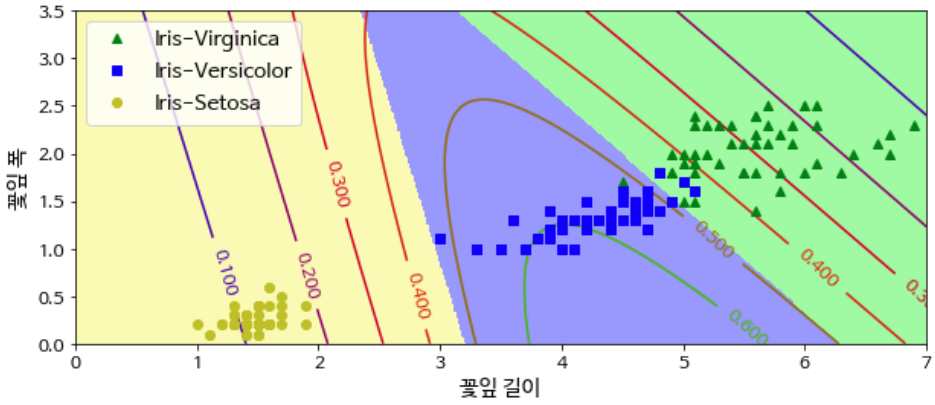
- LogisticRegression에서 'multi_class=multinomial' 설정

```
In [103]: X = iris["data"][:, (2, 3)]  # 꽃잎 길이, 꽃잎 넓이
          y = iris["target"]

          softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10, random_state=42)
          softmax_reg.fit(X, y)

Out[103]: LogisticRegression(C=10, class_weight=None, dual=False, fit_intercept=True,
                             intercept_scaling=1, l1_ratio=None, max_iter=100,
                             multi_class='multinomial', n_jobs=None, penalty='l2',
                             random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
                             warm_start=False)
```

소프트맥스 회귀 구현

In [104]:

```
x0, x1 = np.meshgrid(
        np.linspace(0, 8, 500).reshape(-1, 1),
        np.linspace(0, 3.5, 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]


y_proba = softmax_reg.predict_proba(X_new)
y_predict = softmax_reg.predict(X_new)

zz1 = y_proba[:, 1].reshape(x0.shape)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==2, 0], X[y==2, 1], "g^", label="Iris-Virginica")
plt.plot(X[y==1, 0], X[y==1, 1], "bs", label="Iris-Versicolor")
plt.plot(X[y==0, 0], X[y==0, 1], "yo", label="Iris-Setosa")

from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
contour = plt.contour(x0, x1, zz1, cmap=plt.cm.brg)
plt.clabel(contour, inline=1, fontsize=12)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 7, 0, 3.5])

plt.show()
```



Any Questions...
Just Ask!