

Training Models [2] - Polynomial Regression & Learning Curves

Samkeun Kim <skim@hknu.ac.kr>

<http://cyber.hankyong.ac.kr>

Polynomial Regression

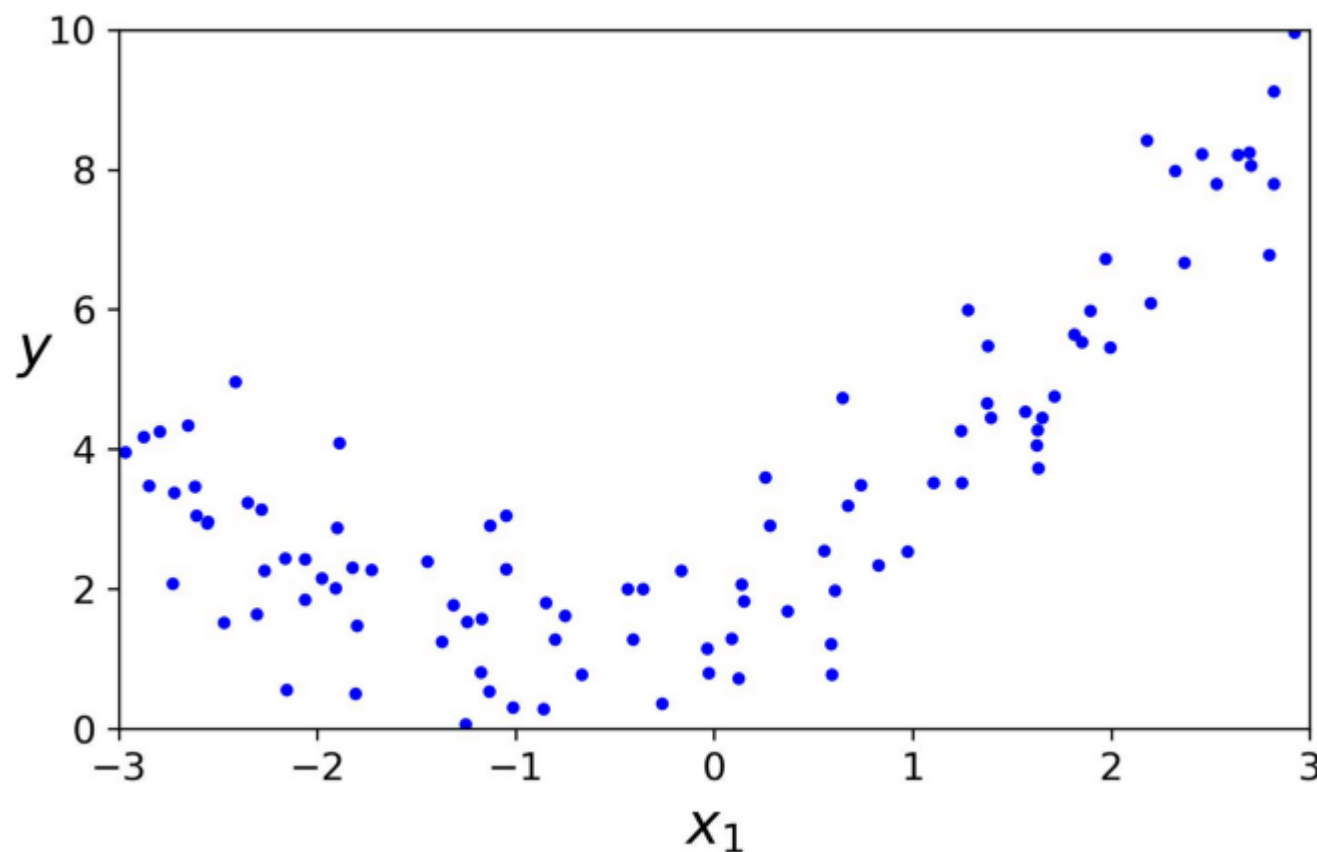
데이터가 직선보다 복잡한 경우 어떻게 해야 할까?

- 놀랍게도 선형 모델을 이용하여 비선형 데이터를 적합시킬 수 있다.
- 간단한 방법 => 각 특성의 거듭제곱(powers)을 새로운 특성으로 추가한 후
- => 새로 확장된 특성 세트에 대해 선형 모델을 학습시키면 된다.
- "Polynomial Regression"이라 함 (다항회귀)

간단한 2차 방정식에 의해 약간의 비선형 데이터 생성:

```
m = 100  
X = 6 * np.random.rand(m, 1) - 3  
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

← $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$



Scikit-Learn의 PolynomialFeatures 클래스를 이용하여 training data를 변환:

- Training set의 각 특성의 제곱(2차 다항식)을 새 특성으로 추가

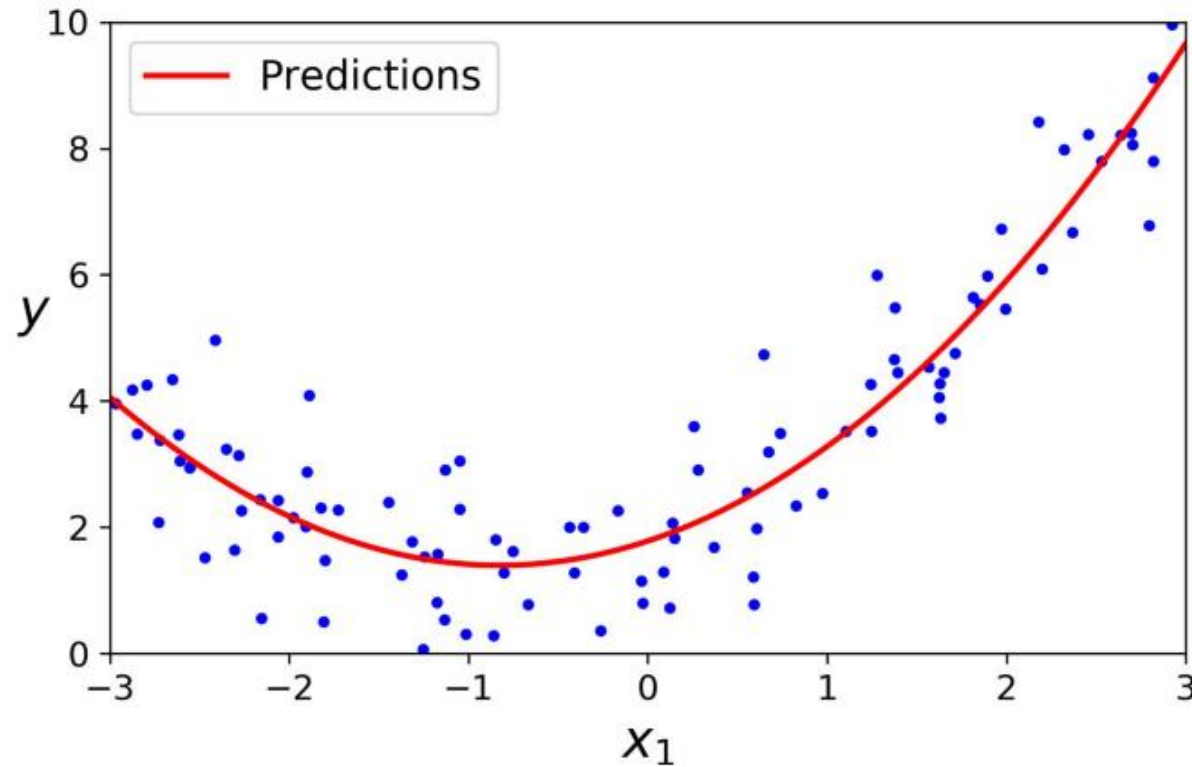
```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

- $X_{poly} \Rightarrow x$ 의 오리지널 특성 + x 의 제곱 특성
- 확장된 training data에 LinearRegression 모델 적용:

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

Polynomial Regression

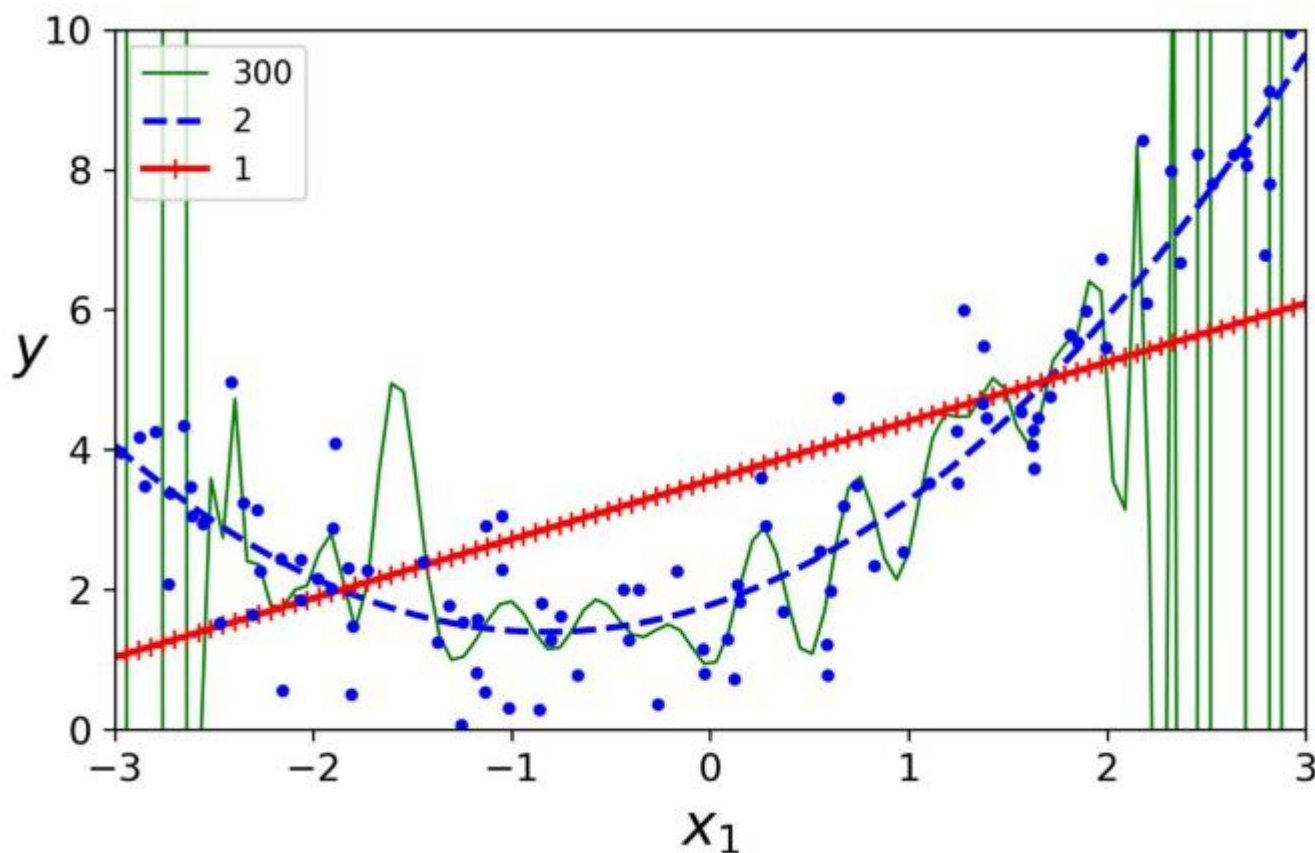
Original function: $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$



Model 추정치: $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$

Learning Curves

고차 Polynomial Regression 수행 => 순수 Linear Regression 보다 훨씬 더 잘 training data를 적합 시킴



300차 Polynomial 모델

⇒ 가급적 training 인스턴스 가까이로 움직인다.

고차 Polynomial Regression 모델 => training data를 심각하게 overfitting

Linear Regression 모델 => underfitting

2차 Polynomial 모델 => 가장 좋은 일반화 성능을 보임 (데이터가 2차 모델에 의해 생성되었기 때문에 당연)

일반적으로 어떤 함수에 의해 데이터가 생성되었는 지 모른다.

⇒ 즉 얼마나 복잡한 모델을 사용해야 할 지를 어떻게 결정해야 할까?

⇒ 모델이 데이터를 overfitting하는 지 아니면 underfitting하는 지를 어떻게 알 수 있나?

2장에서 Cross-validation을 이용하여 모델의 일반화 성능을 구할 수 있었다:

- 모델이 training data에 대해 잘 수행하지만 일반화 성능이 나쁘면 모델은 overfitting
- 둘 다 나쁘면 underfitting
 - ⇒ 모델이 너무 단순한 지 아니면 너무 복잡한 지를 알 수 있는 한 가지 방법!!

모델의 일반화 성능을 추정하기 위한 또 다른 방법: 학습 곡선을 살펴보는 것

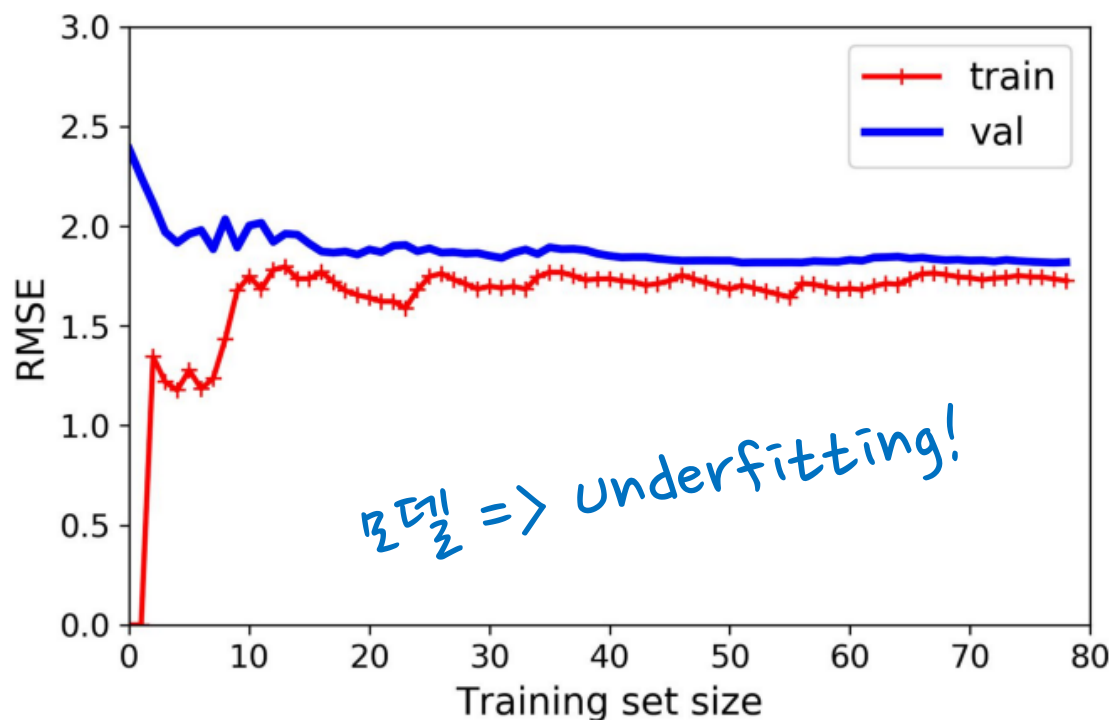
- 즉 training set과 validation set에 대한 모델 성능을 training set 사이즈 함수로서 그려 보는 것 (Plot)
- 그래프 생성을 위해 단순히 training set에서 크기가 다른 서브 세트들을 만들어 여러 번 학습시키면 된다.
- Training data가 주어졌을 때 모델의 학습 곡선을 그리는 코드:

```
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")
```


Linear Regression 모델의 학습 곡선:

```
lin_reg = LinearRegression()  
plot_learning_curves(lin_reg, X, y)
```



1. Training data에 대한 성능

데이터 세트에 한 두개의 인스턴스만 있을 때

⇒ 완벽하게 적합시킬 수 있음

⇒ 학습곡선이 0에서 시작하는 이유

데이터 세트에 새로운 인스턴스가 추가됨에 따라 노이즈도 있고 비선형이기 때문에 모델이 데이터를 완벽하게 학습하는 일이 불가능

⇒ 에러가 고원(plateau) 상태에 도달할 때까지 계속 상승

2. Validation data에 대한 성능

몇 개의 인스턴스에 대해서만 학습되었을 때

⇒ 일반화 할 수 없음 (에러가 초반에 매우 큰 이유)

모델에게 점점 더 많은 인스턴스를 제공함에 따라

⇒ 에러가 서서히 내려 감

하지만 데이터가 비선형이기 때문에 에러는 다시 한번 고원 상태에서 끝나고 training set의 그래프와 매우 가까워 짐

앞의 학습 곡선 => Underfitting의 전형적인 모습:

- 두 곡선이 고원 상태에 도달하고 매우 높은 에러로 가까이 근접해 있음

모델이 training data를 underfitting 한다면:


- => 더 많은 인스턴스를 추가하는 것은 무의미
- => 더 복잡한 모델을 사용하거나 또는 더 좋은 특성을 추가해야 함

앞 예제와 동일한 데이터에 대해 **10차 Polynomial 모델**의 학습 곡선을 살펴보자:

```
from sklearn.pipeline import Pipeline

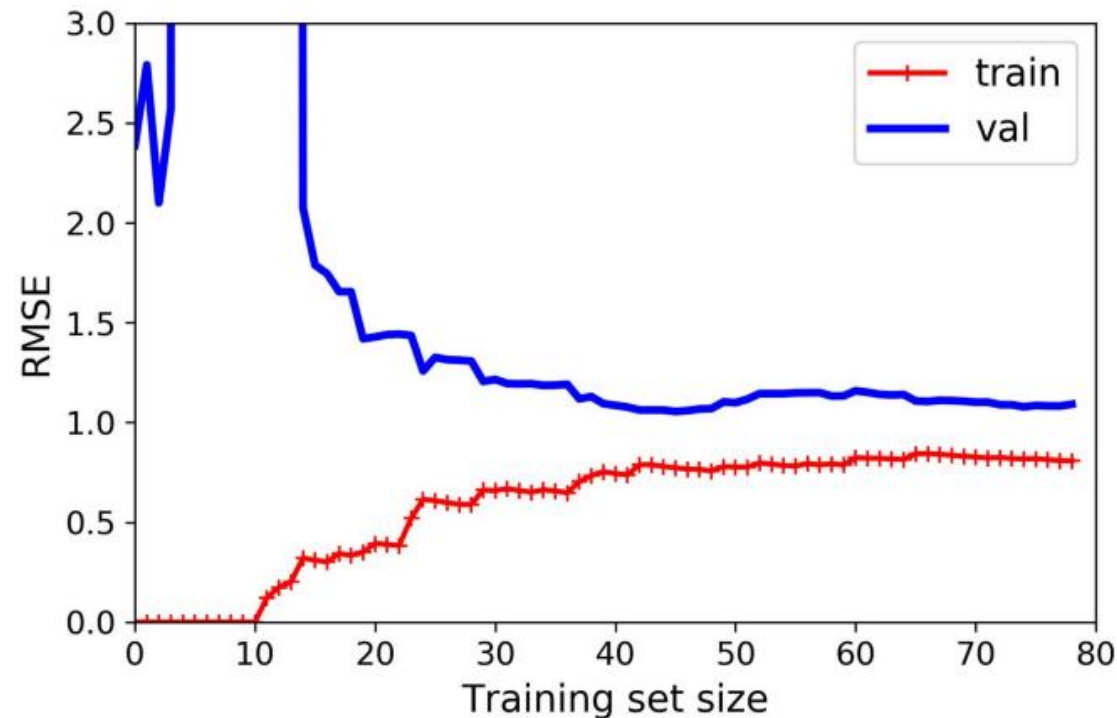
polynomial_regression = Pipeline([
    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
    ("lin_reg", LinearRegression()),
])

plot_learning_curves(polynomial_regression, X, y)
```



앞 그래프와 유사하지만 2가지 중요한 차이점이 있다:

- Training data에 대한 에러가 Linear Regression 모델의 경우보다 훨씬 더 낮다
- 곡선들 간에 간격이 있다 => Training data에 대한 모델 성능이 validation data에 대한 것보다 훨씬 좋다는 것을 의미 => 전형적인 Overfitting 모습
- => 훨씬 더 큰 training set을 이용하면 두 곡선 사이의 간격이 좁아질 것이다.



Regularized Linear Models

Overfitting을 막는 좋은 방법 => 모델을 정규화시키는 것:

- 자유도가 적을수록 데이터를 overfitting하는 가능성이 줄어든다.
- Polynomial 모델을 정규화하는 간단한 방법 => 다항식 차수를 줄이는 것

선형 모델에서 정규화:

- 전형적으로 모델의 가중치를 제한한다.
- 제한하는 방법에 따라 Ridge Regression, Lasso Regression, Elastic Net로 분류

Ridge Regression

Ridge Regression: Linear Regression의 정규화 버전

- 비용 함수에 $\alpha \sum_{i=1}^n \theta_i^2$ 항목을 추가한다.
- Ridge Regression cost function

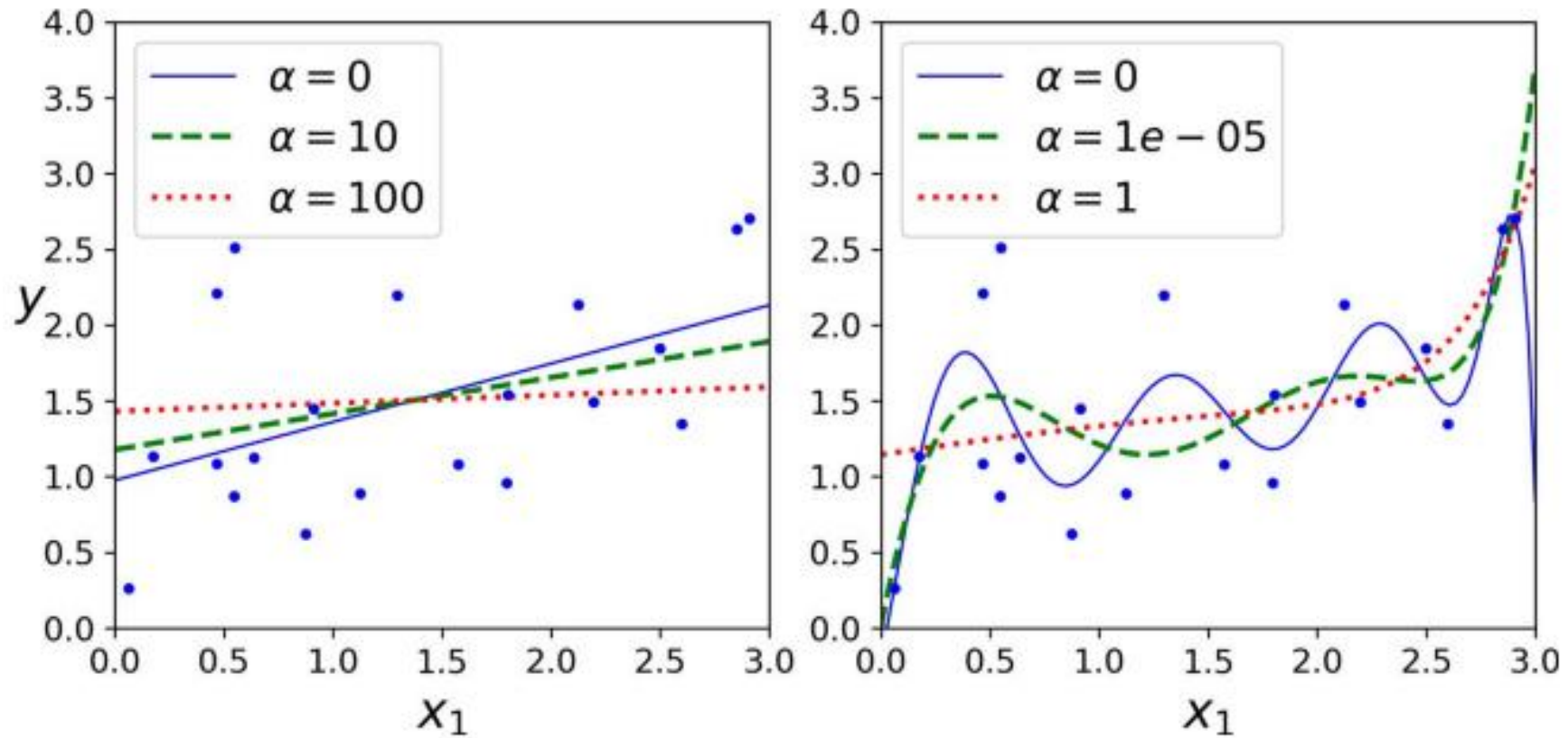
$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

Penalty



Ridge Regression

A linear model (left) and a polynomial model (right), both with various levels of Ridge regularization:



André-Louis Cholesky의 행렬 인수 분해 기법에 기반한 Scikit-Learn을 이용한 릿지 회귀 분석:

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[1.55071465]])
```

Stochastic Gradient Descent 이용:

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

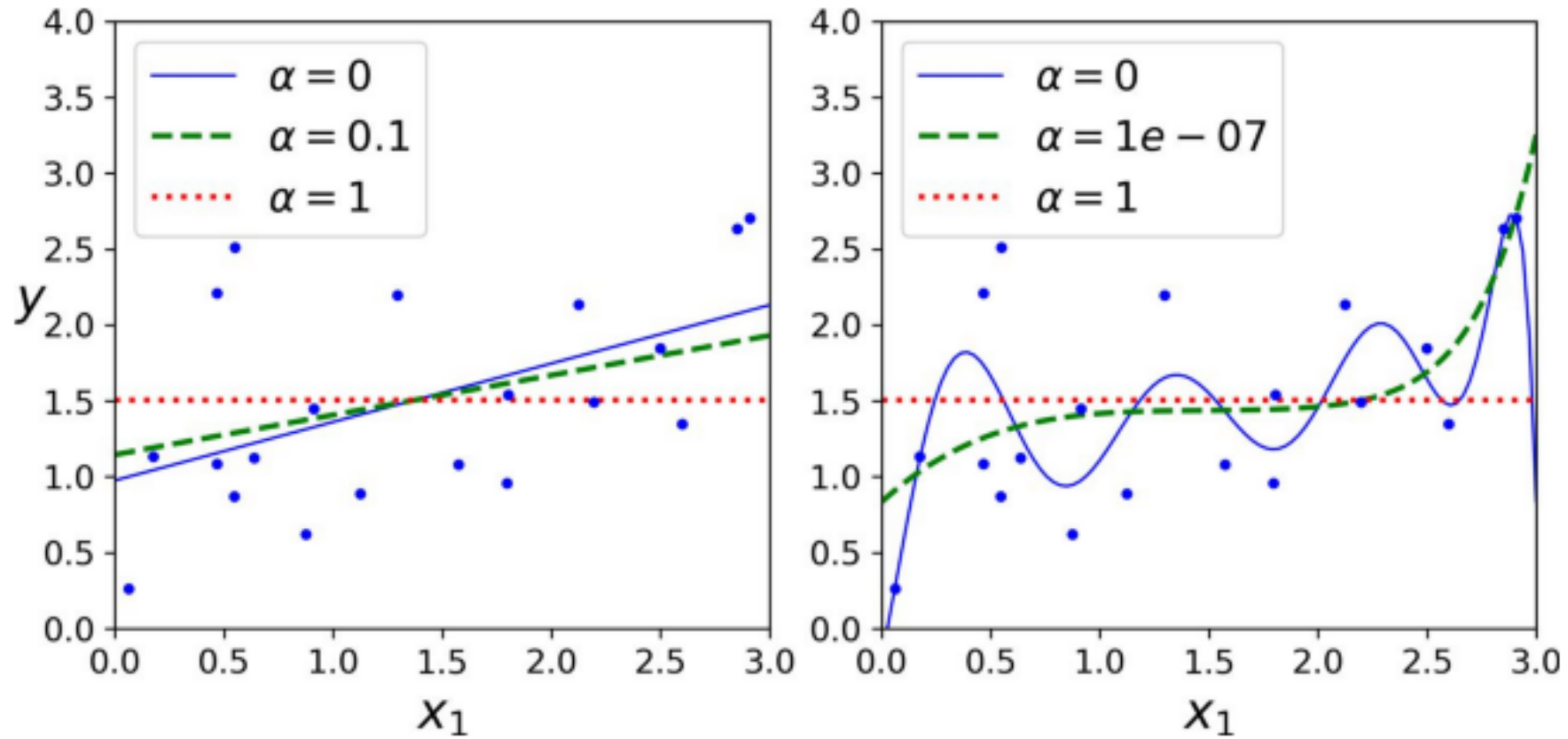

Lasso Regression

Ridge Regression과 유사

- 제곱 항 대신 가중치 벡터 항 사용
- Lasso Regression cost function

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

A linear model (left) and a polynomial model (right), both with various levels of Lasso regularization:



Lasso 클래스를 이용한 Scikit-Learn 예제:

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([1.53788174])
```

Elastic Net

Ridge Regression과 Lasso Regression의 중간 정도

- 두 정규화 항목들을 섞어 놓은 것
- Elastic Net cost function

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

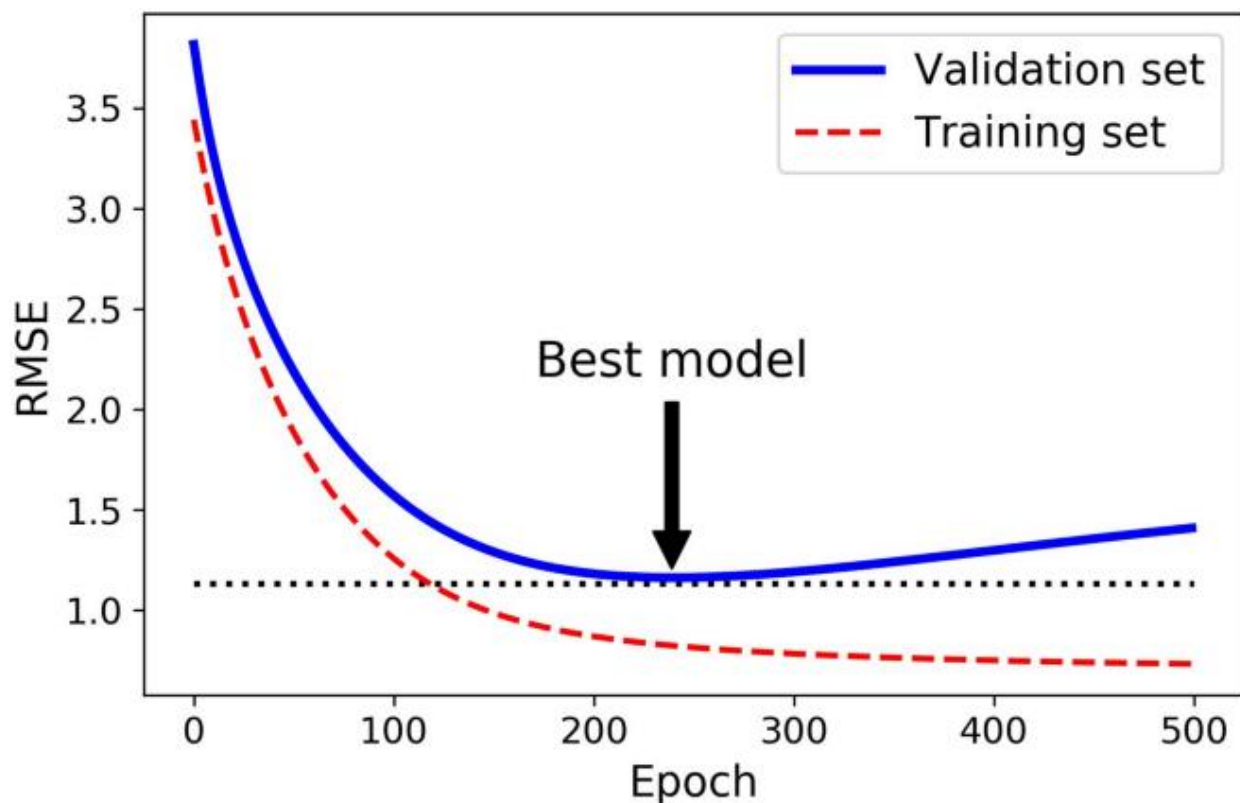
Scikit-Learn의 ElasticNet을 이용한 간단한 예제:

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.54333232])
```

Early Stopping

Gradient Descent와 같은 반복 학습 알고리즘을 정규화하기 위한 또 다른 방법

- Validation 에러가 최소 값에 도달하자 마자 학습을 멈춘다 ("early stopping"이라 함)



Stochastic GD와 Mini-batch GD는 학습 곡선이 부드럽지 않다:

⇒ 최소 값에 도달했는 지 판단하기 어려움

해결책:

⇒ Validation 에러가 얼마 동안 최소값 이상으로 머물러 있으면 학습 중단

⇒ Validation 에러가 최소값이었던 지점으로 롤백!

Early stopping의 간단한 구현:

```
from sklearn.base import clone

# prepare the data
poly_scaler = Pipeline([
    ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
    ("std_scaler", StandardScaler())
])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train) # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

실습과제 6-1

본문에 나오는 전체 내용 PyCharm에서 실행하기

★ 반드시 결과 값 및 결과로 나온 모든 그래프에 대해 자세한 분석을 하시오.

참고: [제 06강 실습과제 #6 Training Models \[2\] - Polynomial Regression & Learning Curves.pdf](#)

실습과제 6-2

Suppose you are using Polynomial Regression. You plot the learning curves and you notice that there is a large gap between the training error and the validation error. What is happening? What are three ways to solve this?

