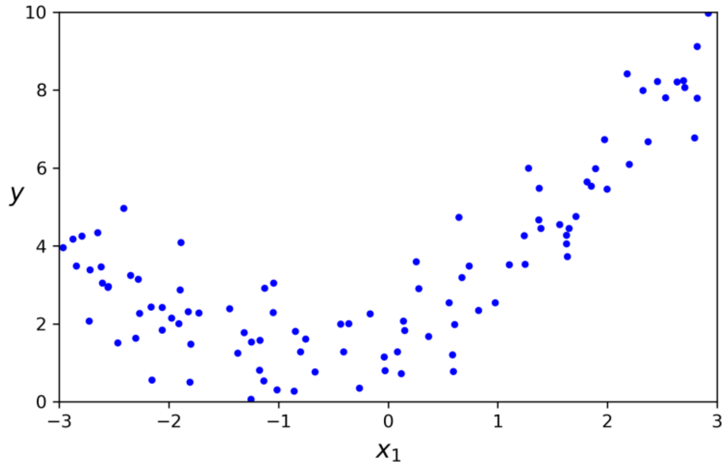
## 3.
## Polynomial Regression

# Polynomial Regression

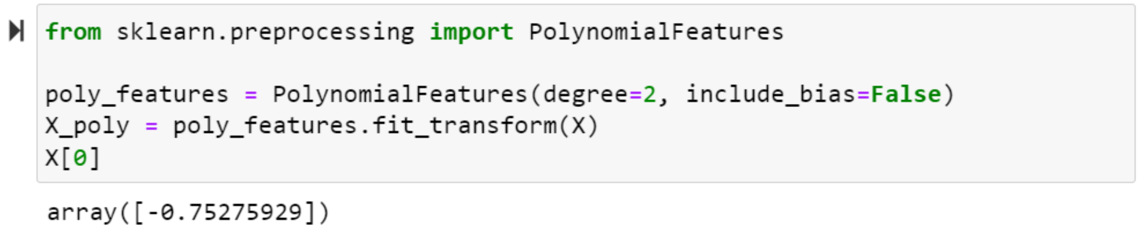
- *Polynomial regression*: add powers of each feature as new features, then train a linear model on this extended set of features.
- *Example:* generate nonlinear data, based on a quadratic equation:

```
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m, 1)
```



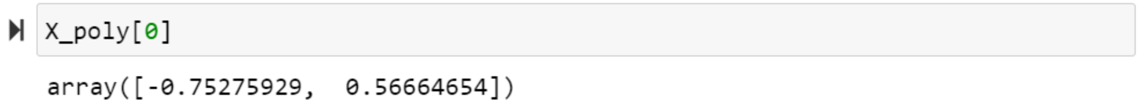
# Polynomial Regression

- Use ScikitLearn’s `PolynomialFeatures` class to transform our training data, adding the square of each feature in the training set:



```
from sklearn.preprocessing import PolynomialFeatures

poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0]
```

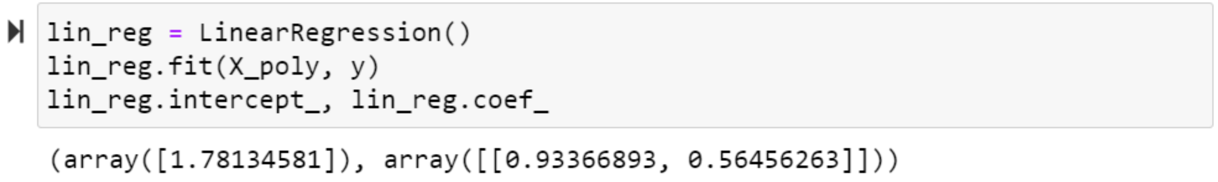
```
array([-0.75275929])
```



```
X_poly[0]
```

```
array([-0.75275929,  0.56664654])
```

- Fit a `LinearRegression` model to this extended training data:



```
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

- Fit a `LinearRegression` model to this extended training data:

```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

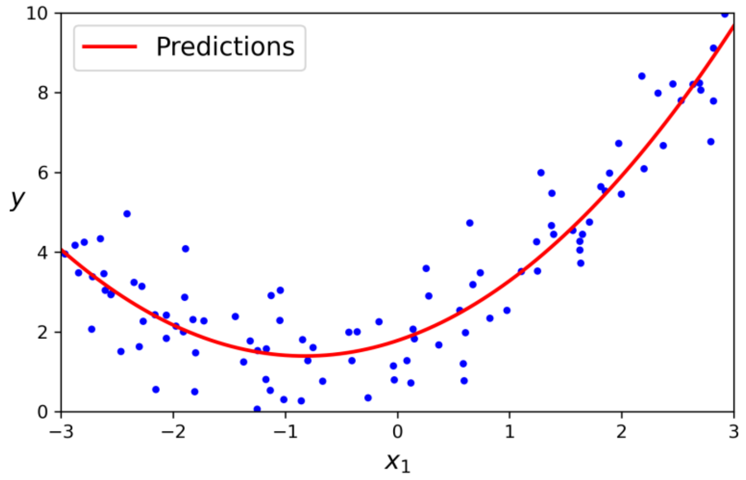
```
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

- The model estimates:

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

when the original function was:

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

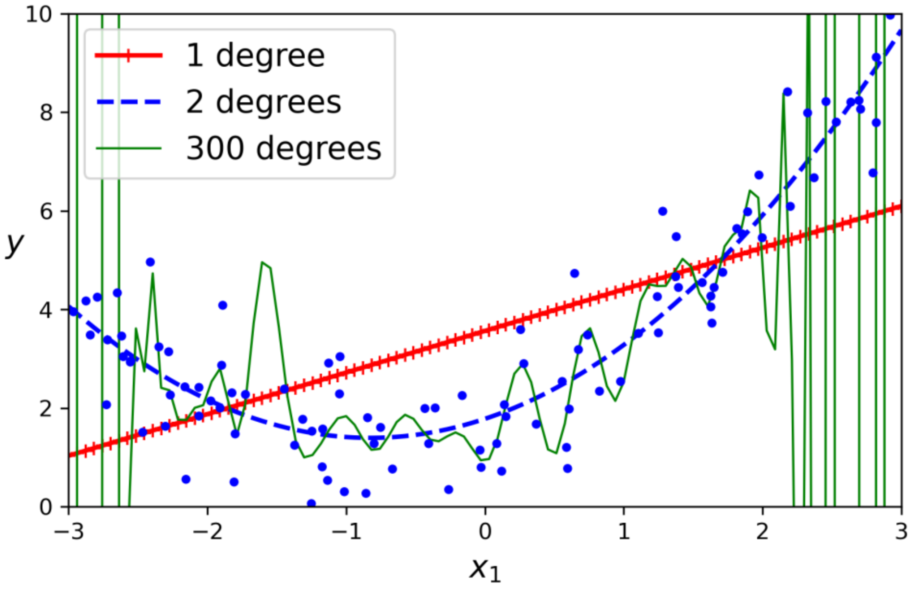


## Relationship Between Features

- When there are multiple features, polynomial regression is capable of finding relationships between features.
- The reason is `PolynomialFeatures` also adds all combinations of features up to the given degree.
- Example. if there are two features $a$ and $b$, `PolynomialFeatures` with degree = 3 would add features $a^2, b^2, a^3, b^3, ab\,, a^2b\,, ab^2$.
- `PolynomialFeatures(degree=d)` transforms an array containing $n$ features into an array containing $\binom{d+n}{d}$ features.

## High-degree Polynomial Regression

- High-degree polynomial regression model might severely overfit the training data:

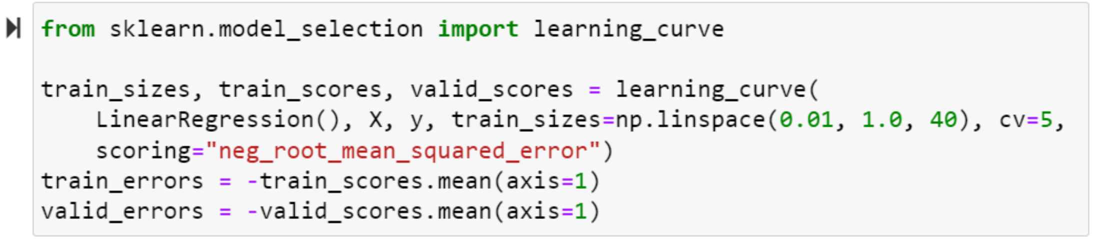


## Learning Curve

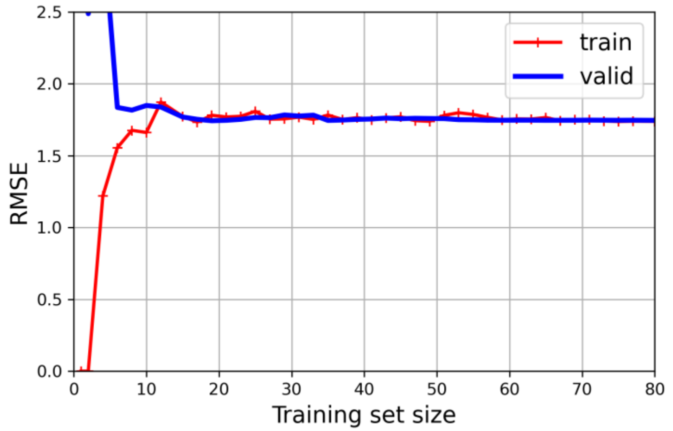
- *Learning curves*: plots of the model's performance on the training set and the validation set as a function of the training set size or the training iteration.
  - Evaluate the model at regular intervals during training on both the training set and the validation set, and plot the results.
- If the model cannot be trained incrementally (e.g., if it does not support `partial_fit`), train it several times on gradually larger subsets of the training set.
- Scikit-Learn has a `learning_curve`() function to help with this: it trains and evaluates the model using cross-validation.

# `learning_curve` function

- By default `learning_curve()` retrains the model on growing subsets of the training set, but if the model supports incremental learning, you can set `exploit_incremental_learning=True` and it will train the model incrementally instead.

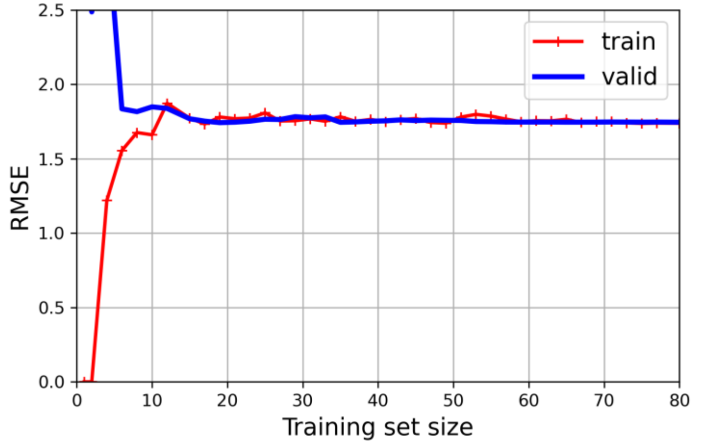


```
from sklearn.model_selection import learning_curve

train_sizes, train_scores, valid_scores = learning_curve(
    LinearRegression(), X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5,
    scoring="neg_root_mean_squared_error")
train_errors = -train_scores.mean(axis=1)
valid_errors = -valid_scores.mean(axis=1)
```



# Getting insight from the learning curve

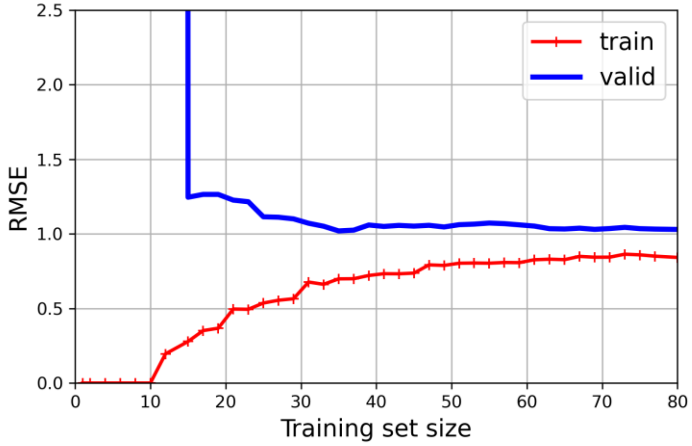
- Performance on the training data:
  - If just one or two instances in the training set, the model can fit them perfectly.
  - The error on the training data goes up until it reaches a plateau.
- Performance on the validation data:
  - A model that is trained on very few training instances is incapable of generalizing → big validation error.
  - As training set size grows, the model learns, and the validation error slowly goes down.
  - A straight line cannot model the data well, so the error ends up at a plateau.



**The model is underfitting the training data**

# Learning Curve

- The learning curves of a 10-degree polynomial model on the same data:
  - The error on the training data is much lower than with the Linear Regression.
  - There is a larger gap between the curves, in other words the model performs significantly better on the training data than on the validation data, which is the sign of an overfitting model.
- One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.



```
from sklearn.pipeline import make_pipeline

polynomial_regression = make_pipeline(
    PolynomialFeatures(degree=10, include_bias=False),
    LinearRegression())

train_sizes, train_scores, valid_scores = learning_curve(
    polynomial_regression, X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5,
    scoring="neg_root_mean_squared_error")
```

# The Bias-Variance Tradeoff

- A model's generalization error can be expressed as the sum of three different errors:
  - *Bias*: the error due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.
  - *Variance*: the error due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom is likely to have high variance and thus overfit the training data.
  - *Irreducible error:* the error due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data.
- Bias-Variance Tradeoff: Increasing a model's complexity will typically increase its variance and reduce its bias.

# Mathematical Intuition

- For any random variable $\boldsymbol{X}$ with a mean $\mu = \mathrm{E}(\boldsymbol{X})$, and any real number $a$:

$$\mathrm{E}\left((\boldsymbol{X}-a)^2\right) = \mathrm{E}\left((\boldsymbol{X}-\mu+\mu-a)^2\right)$$

$$= E\left((\boldsymbol{X}-\mu)^2\right) + 2\mathrm{E}(\boldsymbol{X}-\mu)(\mu-a) + (\mu-a)^2 = \mathrm{Var}(\boldsymbol{X}) + (E[\boldsymbol{X}]-a)^2$$

- The estimator $\hat{\boldsymbol{y}}$ is a random variable (gets a different value for different datasets) that tries to predict $y$, so the error of prediction is:

$$\mathrm{E}\left((\hat{\boldsymbol{y}}-y)^2\right) = \mathrm{Var}(\hat{\boldsymbol{y}}) + (\mathrm{E}(\hat{\boldsymbol{y}})-y)^2 = \mathrm{Variance} + \mathrm{Bias}^2$$

- **Variance** $=$ regardless of what the true $y$ is, how much our prediction change with the dataset? $\rightarrow$ high variance is a sign of overfitting

- **Bias** $=$ how far is the average prediction from the true $y$?
  - High bias means you have an insufficiently complex function class $\rightarrow$ high bias is a sign of underfitting

## 4.
## Regularized Linear Models

# Regularized Linear Models

- One way to reduce overfitting is to regularize the model.
- Generally you should avoid plain linear regression and have at least a little bit of regularization.
  - The fewer degrees of freedom means it’s harder for the model to overfit the data (i.e., smaller variance).
- For a linear model, regularization is typically achieved by constraining the weights ($\theta_j$) of the model.
  - Ridge Regression
  - Lasso Regression
  - Elastic Net

# Ridge Regression

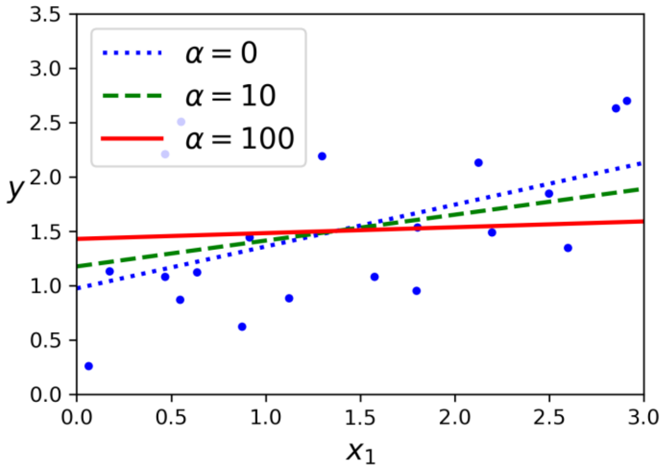
- *Ridge Regression* is a regularized version of Linear Regression: a *regularization term* ($\alpha \sum_{i=1}^{n} \theta_i^2$) is added to the cost function:

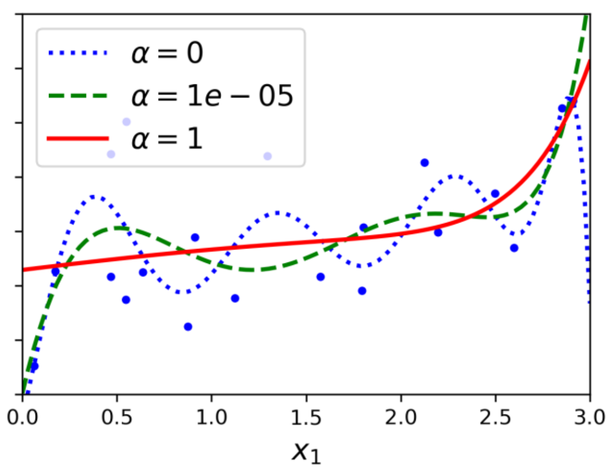
$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^{n} \theta_i^2$$

- This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible.
- Regularization term should only be added to the cost function during training.
  - You should use the unregularized performance measure to evaluate the model's performance.

# Regularization Strength

- *Regularization strength*: the hyperparameter $\alpha$ which controls how much you want to regularize the regression.
  - $\alpha = 0$: ridge regression is the same as ordinary linear regression.
  - Large $\alpha$: all weights end up very close to zero.



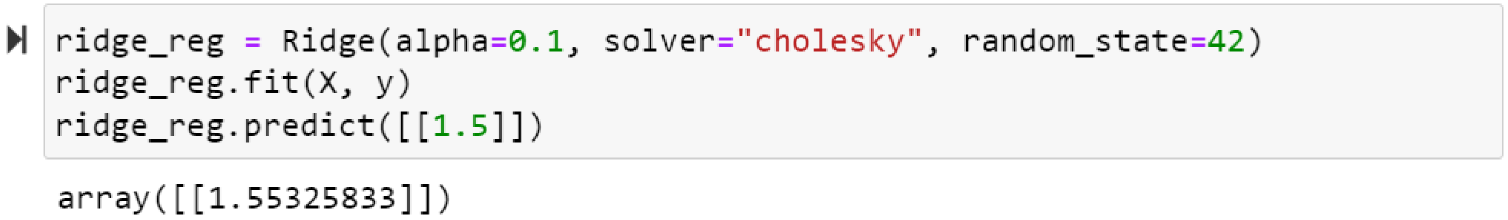


## Closed-form Equation for Ridge Regression

- Closed-form solution for ridge regression:

$$\hat{\boldsymbol{\theta}} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X} + \alpha\mathbf{A}\right)^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

$$\mathbf{A}_{(n+1)\times(n+1)} = \begin{bmatrix} 0 & \mathbf{0}_{1\times n} \\ \mathbf{0}_{n\times 1} & \mathbf{I}_{n\times n} \end{bmatrix}$$



```
ridge_reg = Ridge(alpha=0.1, solver="cholesky", random_state=42)
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])
```

```
array([[1.55325833]])
```

# SGD for Ridge Regression

- We can perform ridge regression using stochastic gradient descent:

```
sgd_reg = SGDRegressor(penalty="l2", alpha=0.1 / m, tol=None,
                       max_iter=1000, eta0=0.01, random_state=42)
sgd_reg.fit(X, y.ravel())  # y.ravel() because fit() expects 1D targets
sgd_reg.predict([[1.5]])
```

```
array([1.55302613])
```

- The penalty hyperparameter sets the type of regularization term to use (we used `penalty=None` for ordinary linear regression).
- Specifying "l2" indicates that we want SGD to add a regularization term to the cost function equal to the square of the $\ell_2$ norm of the weight vector.

# Ridge Regression

- It is important to scale the data (e.g., using a `StandardScaler`) before performing ridge regression, as it is sensitive to the scale of the input features.
  - This is true of most regularized models.

- The `RidgeCV` class also performs ridge regression, but it automatically tunes hyperparameters using cross-validation.
  - It's roughly equivalent to using `GridSearchCV`, but it's optimized for ridge regression and runs *much* faster.

## Lasso Regression

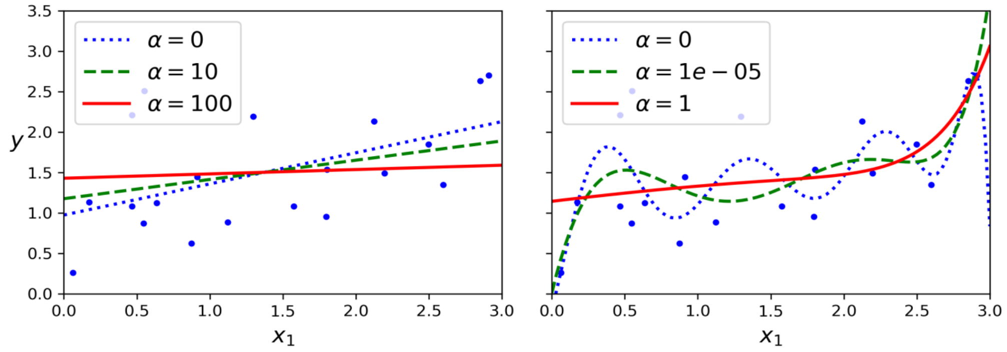
- *Least Absolute Shrinkage and Selection Operator (Lasso) Regression* adds a regularization term (the $\ell_1$ norm of the weight vector) to the cost function:

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

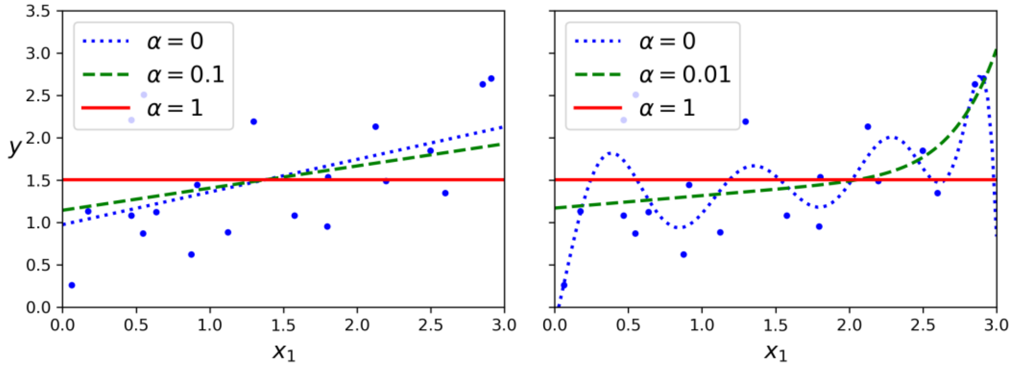
- Lasso tends to eliminate the weights of the least important features (i.e., set them to zero).
    - Lasso automatically performs feature selection and outputs a *sparse model* with few nonzero feature weights.

# Lasso vs. Ridge Regression

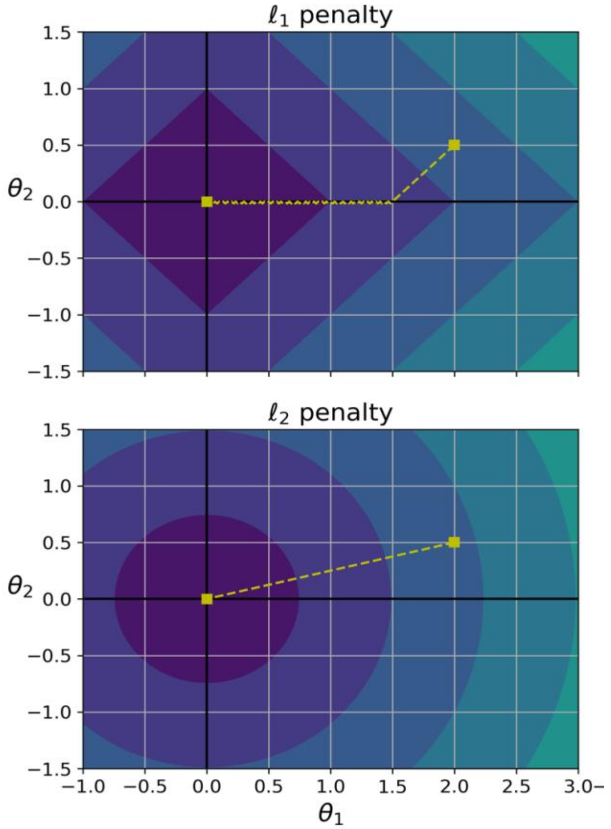
**Ridge Regression:**



**Lasso Regression:**



# Lasso vs. Ridge Regression



# SGD for Lasso Regression

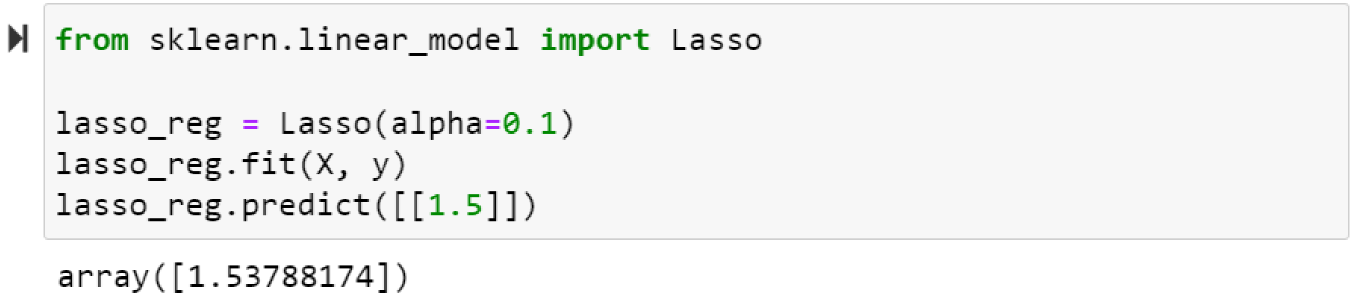
- The Lasso cost function is not differentiable at $\theta_i = 0$ but Gradient Descent works fine if you use a *subgradient vector* $\mathbf{g}$ instead when any $\theta_i = 0$:

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} \, \text{MSE}(\boldsymbol{\theta}) + \alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix} \quad \text{where } \text{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

- To avoid SGD from bouncing around the optimum at the end when using Lasso, you need to gradually reduce the learning rate during training.

# Lasso Regression in Scikit-Learn

- Using the `Lasso` class:



```
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])
```

- You could instead use:

```
SGDRegressor(penalty="l1", alpha=0.1)
```

# Lasso vs. Ridge vs. Ordinary Linear Regression

```python
np.random.seed(42)
m = 200
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m, 1)
poly_features = PolynomialFeatures(degree=5, include_bias=False)
X_poly = poly_features.fit_transform(X)
```

```python
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

```
(array([1.97265775]),
 array([[ 1.30386237,  0.54933791, -0.10904585,
-0.00332436,  0.00790312]]))
```

```python
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=0.1, solver="cholesky")
ridge_reg.fit(X_poly, y)
ridge_reg.intercept_, ridge_reg.coef_
```

```
(array([1.97320891]),
 array([[ 1.29943929,  0.54878793, -0.10723985,
-0.00325974,  0.0077417 ]]))
```

```python
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.05)
lasso_reg.fit(X_poly, y)
lasso_reg.intercept_, lasso_reg.coef_
```

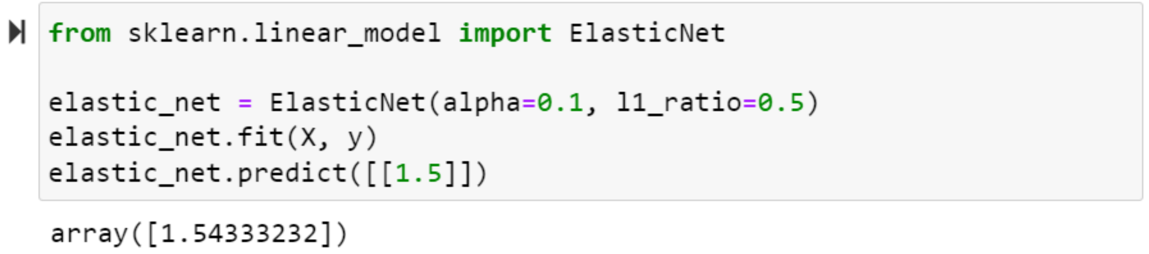
```
(array([2.06444745]),
 array([ 1.02199338,  0.46431901,  0.        ,
0.00677193, -0.00149875]))
```

# Elastic Net

- In *Elastic Net* the regularization term is a mix of both Ridge and Lasso’s regularization terms. You can control the mix ratio $r$:

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

- Ridge regression is a good default, but if you suspect that only a few features are useful, you may prefer Lasso or Elastic Net.



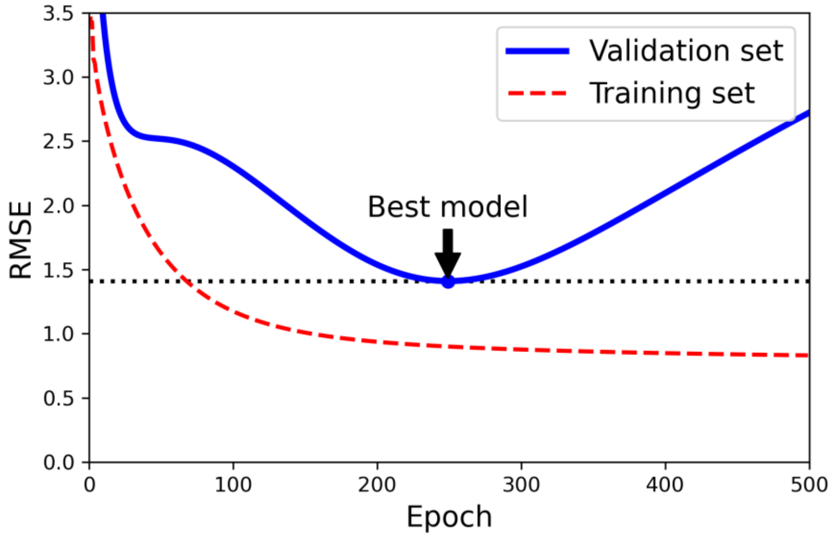
```
from sklearn.linear_model import ElasticNet

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

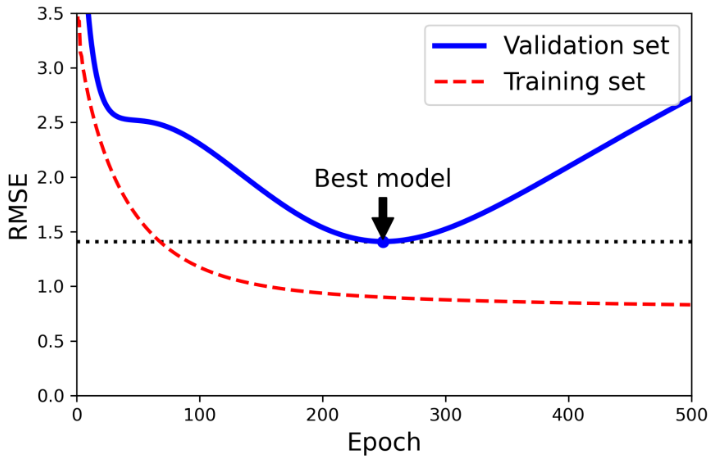
- *Early Stopping:* a way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum.

- When the validation error stops decreasing and starts to go back up, the model has started to overfit the training data.



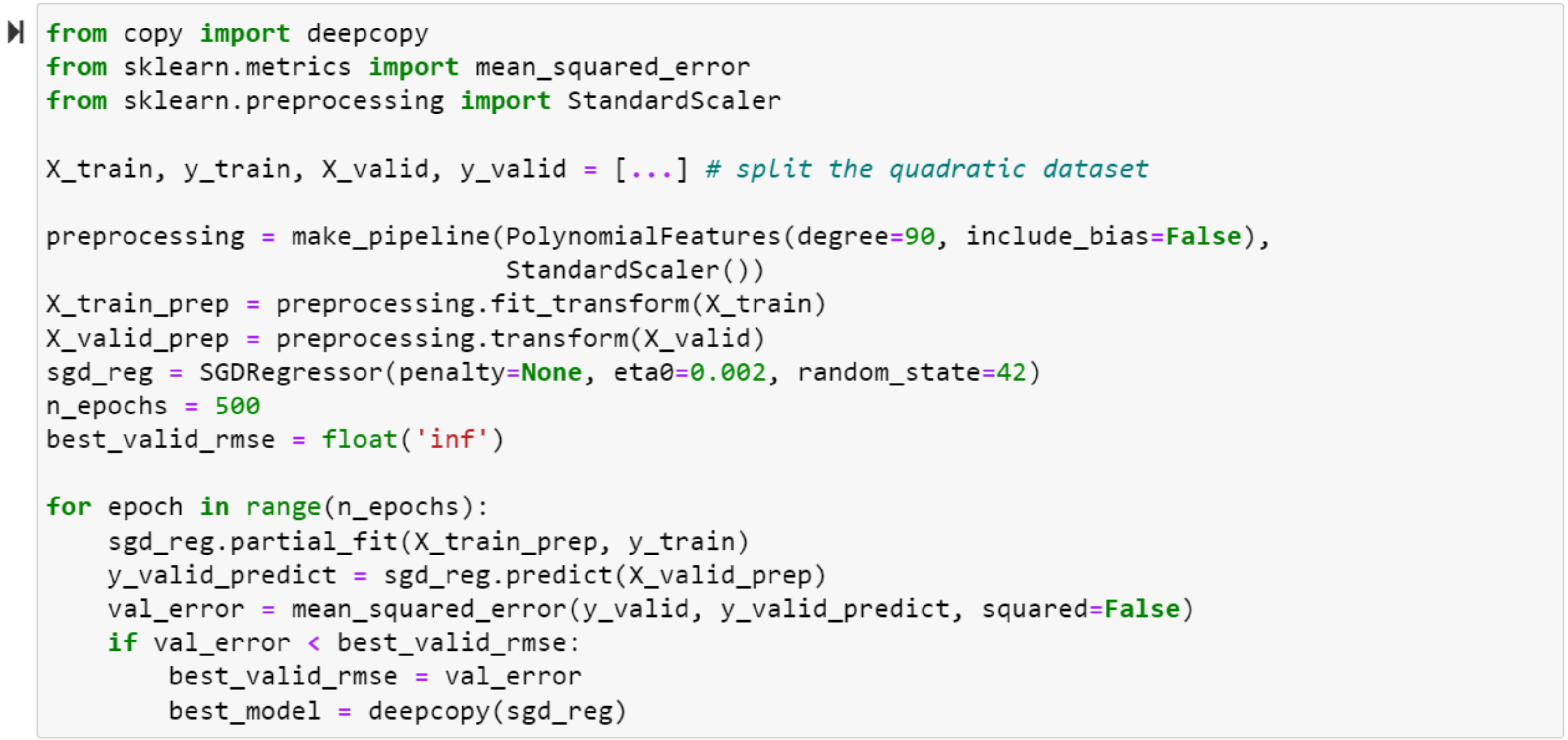
# Early Stopping

- With stochastic and mini-batch gradient descent, the curves are not so smooth, and it may be hard to know whether you have reached the minimum or not.



- Solution: stop only after the validation error has been above the minimum for some time, then roll back the model parameters to the point where the validation error was at a minimum.

# Implementation of Early Stopping



```python
from copy import deepcopy
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler

X_train, y_train, X_valid, y_valid = [...] # split the quadratic dataset

preprocessing = make_pipeline(PolynomialFeatures(degree=90, include_bias=False),
                              StandardScaler())
X_train_prep = preprocessing.fit_transform(X_train)
X_valid_prep = preprocessing.transform(X_valid)
sgd_reg = SGDRegressor(penalty=None, eta0=0.002, random_state=42)
n_epochs = 500
best_valid_rmse = float('inf')

for epoch in range(n_epochs):
    sgd_reg.partial_fit(X_train_prep, y_train)
    y_valid_predict = sgd_reg.predict(X_valid_prep)
    val_error = mean_squared_error(y_valid, y_valid_predict, squared=False)
    if val_error < best_valid_rmse:
        best_valid_rmse = val_error
        best_model = deepcopy(sgd_reg)
```