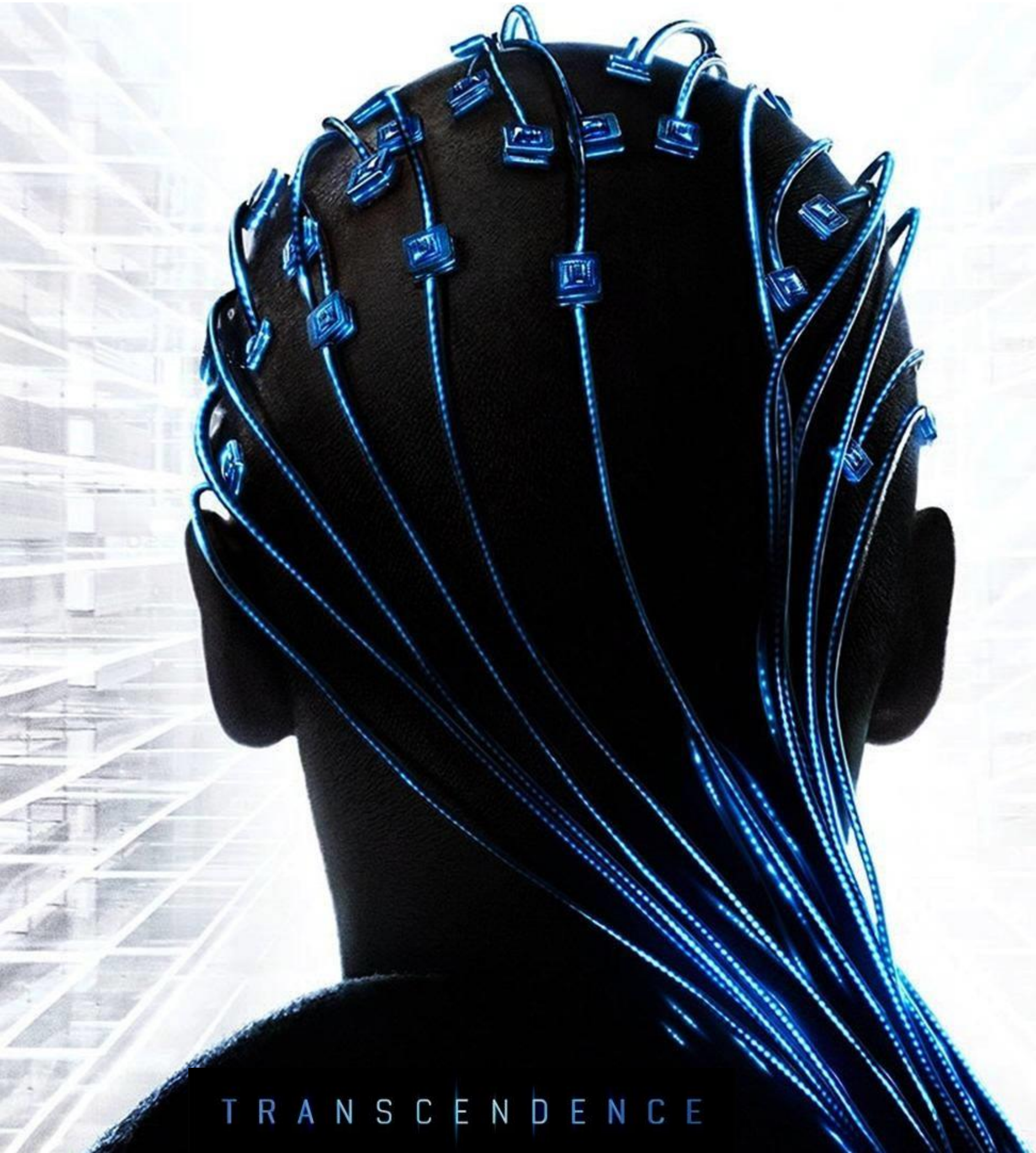


Solvers, Optimization, and Regularization

Alireza Fazeli
afazeli2@uwo.ca

Contents

We need to know how things work under the hood to:

- choose the appropriate model,
- choose the right training algorithm,
- use a good set of hyperparameters,
- help debug issues,
- perform error analysis more efficiently.

PS. Some of topics covered today will be essential in understanding neural networks.

Linear Regression

Model governing equation:

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

- $\hat{y}$ is the predicted value.
- n is the number of features.
- x_i is the i^{th} feature value.
- θ_j is the j^{th} model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$).

Linear Regression

In vectorized form:

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

Recap: Training this model translates to finding θ that minimizes the MSE (or RMSE):

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

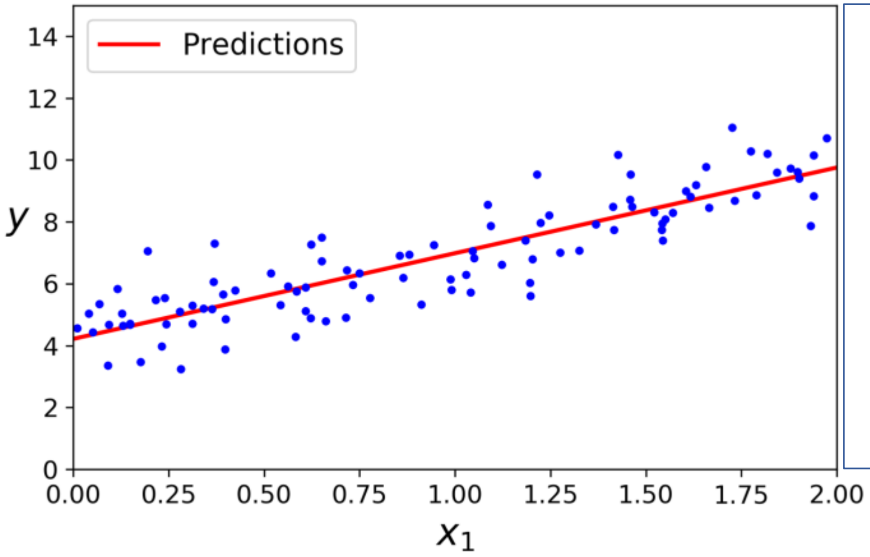
PS. We use h_θ to make it clear that the model is parametrized by the vector θ.

Linear Regression: Optimization

Normal equation:

$$\widehat{\boldsymbol{\theta}} = (\mathbf{X}^\top\mathbf{X})^{-1} \ \mathbf{X}^\top \ \mathbf{y}$$

- $\widehat{\boldsymbol{\theta}}$ is the value of $\boldsymbol{\theta}$ that minimizes the cost function.
- $\mathbf{y}$ is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.



Raw data generated by
$y = 4 + 3x_1$ + Gaussian noise

Using Scikit-Learn's *LinearRegression* class, the normal equation calculates
θ_0= 4.215 and **θ_1= 2.770**
(because of noise)

Linear Regression: Optimization

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^\top\mathbf{X})^{-1} \quad \mathbf{X}^\top \quad \mathbf{y} \qquad \Longrightarrow \qquad \hat{\boldsymbol{\theta}} = \mathbf{X}^{+}\mathbf{y}$$

Instead of computing the normal equation directly, Scikit-Learn Calculates the Moore-Penrose inverse (i.e., a pseudoinverse)

The pseudoinverse itself is computed using a matrix factorization technique called *Singular Value Decomposition* (SVD) that can decompose the training set (matrix **X**) into the matrix multiplication of three matrices **U, Σ,** and **V**$^\top$.

Linear Regression: Optimization

Why?

If $\mathbf{X}^T\mathbf{X}$ is singular, but the pseudoinverse is always defined.

Computational Complexity of Inverting

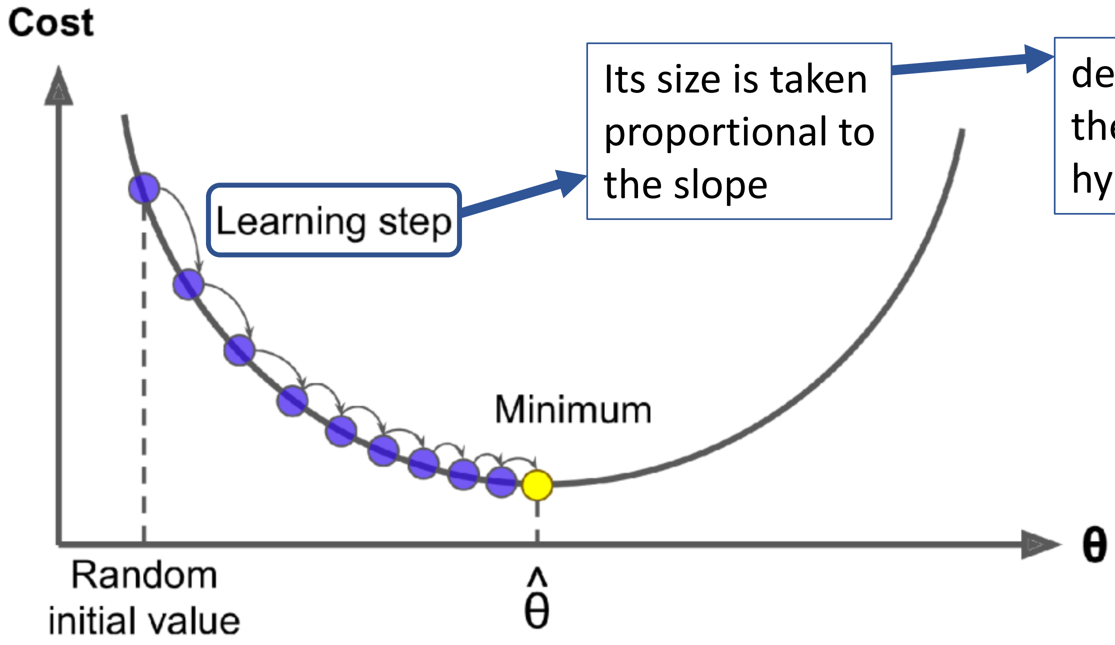
Normal Equation	Singular Value Decomposition (SVD)
$O(n^{2.4})$ - $O(n^3)$	$O(n^2)$

n is the number of features.

However, both get very slow when the number of features grows large. On the positive side, both are linear regarding the number of instances, *m*, in the training set: *O(m)*

Linear Regression: Optimization

Gradient Descent:
Tweak parameters iteratively to minimize a cost function. It measures the local gradient of the error function with respect to **θ**, and it goes in the direction of descending gradient. Once the gradient is zero, the minimum found.

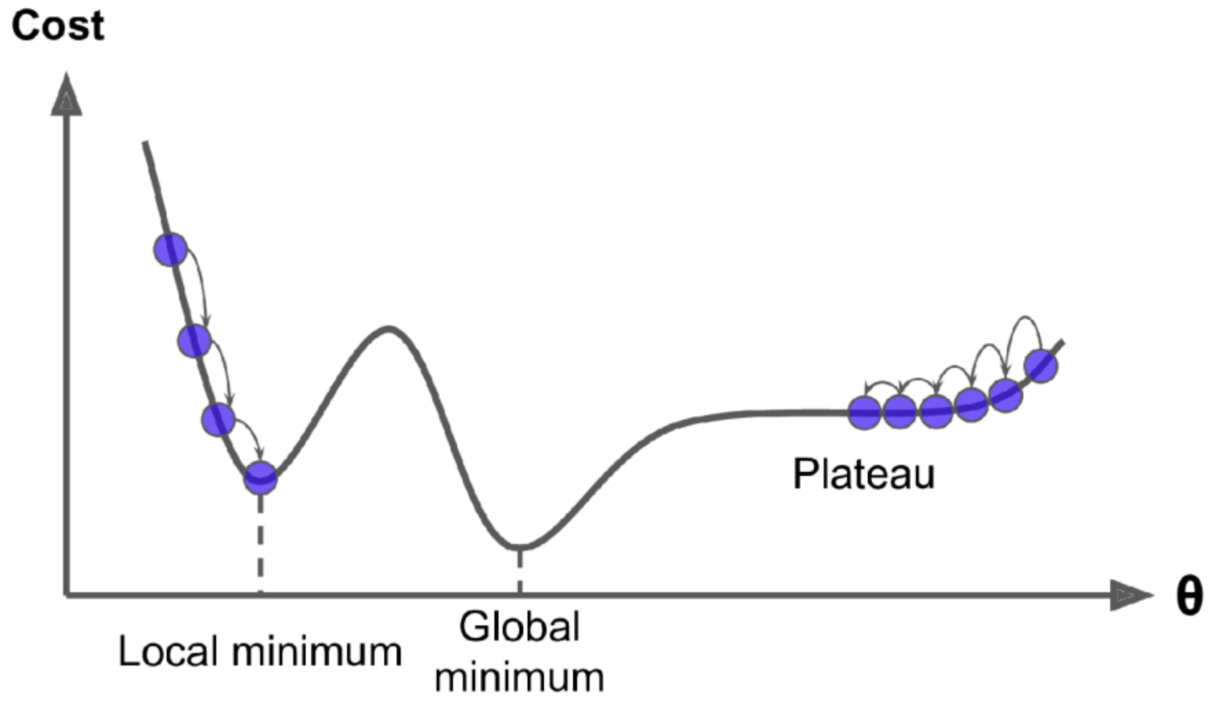


Its size is taken proportional to the slope

determined by the learning rate hyperparameter

Optimization

Not all cost functions look as nice as the MSE, which is a continuous convex function with just one global minimum.



Linear Regression: Optimization

Full (or Batch) Gradient Descent:

How much the cost function will change if you change θ_j just a little bit?

$$\frac{\partial}{\partial \theta_j} \mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^\top \mathbf{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

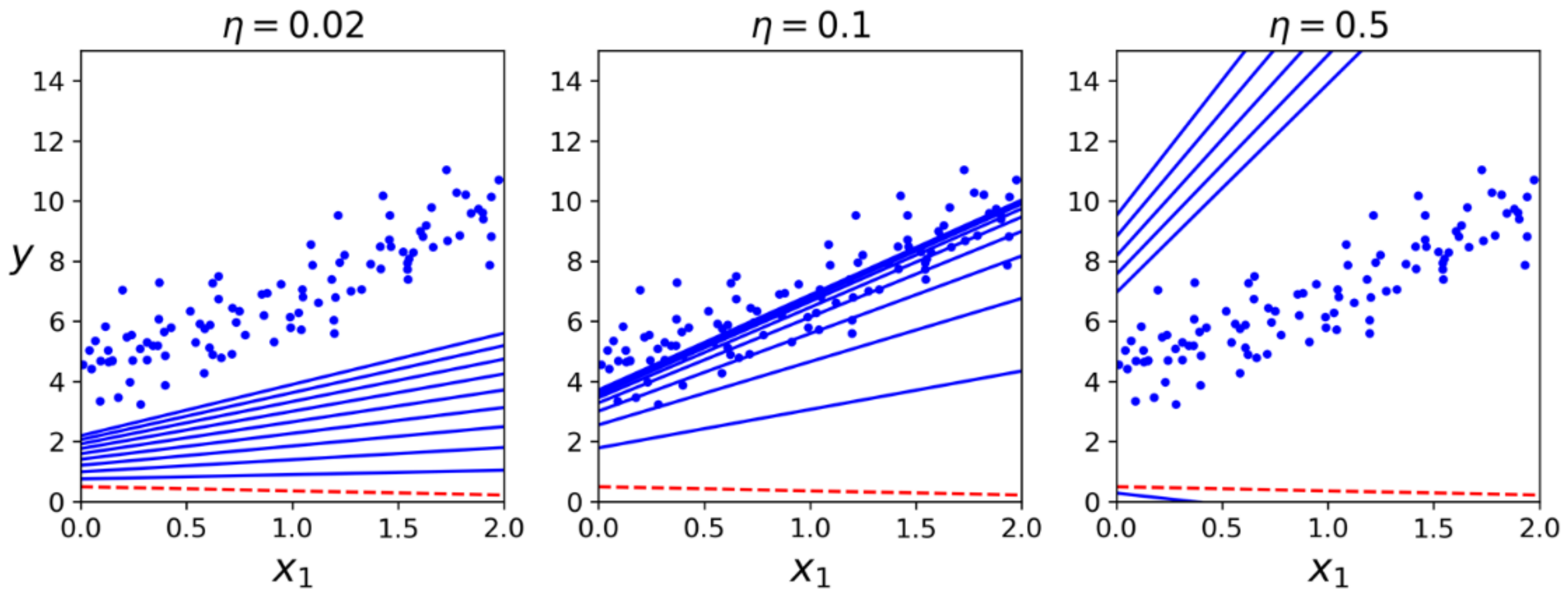
which, can be written as a gradient vector:

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

Linear Regression: Optimization

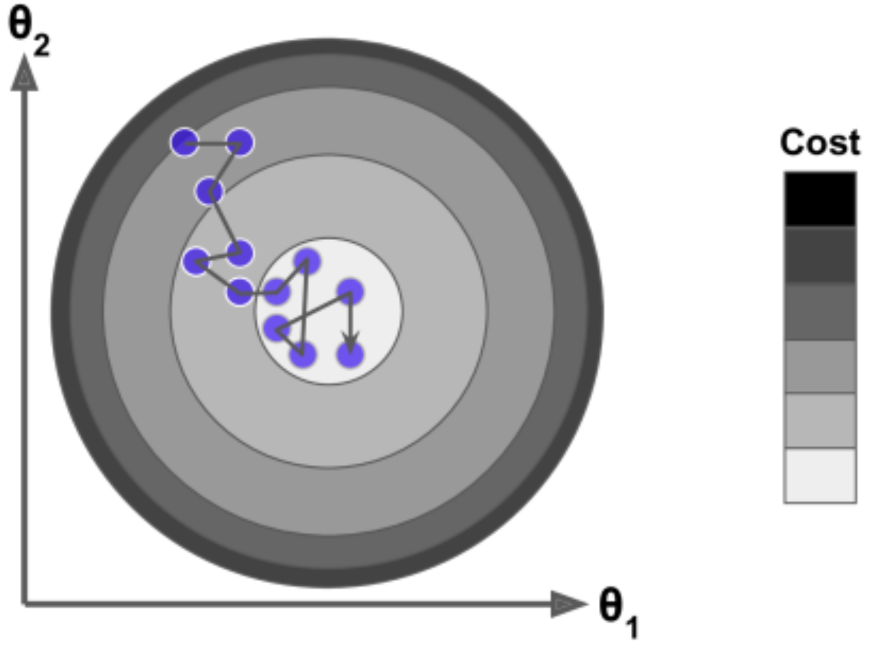
If the gradient points uphill, just go in the opposite direction to go downhill. This means subtracting the gradient from θ, and this is where the learning rate η comes into play:

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



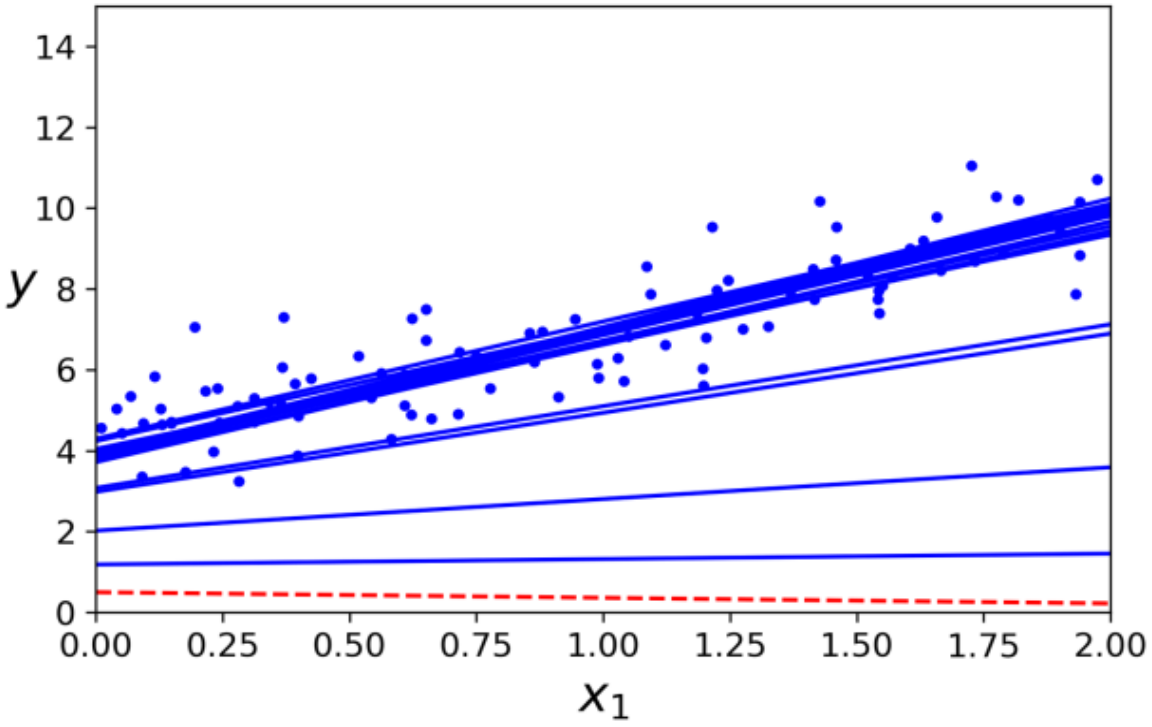
Linear Regression: Optimization

Stochastic Gradient Descent:
It picks a random instance in the training set at every step and computes the gradients based only on that single instance. Due to its stochastic nature, the cost function will bounce up and down, decreasing only on average.



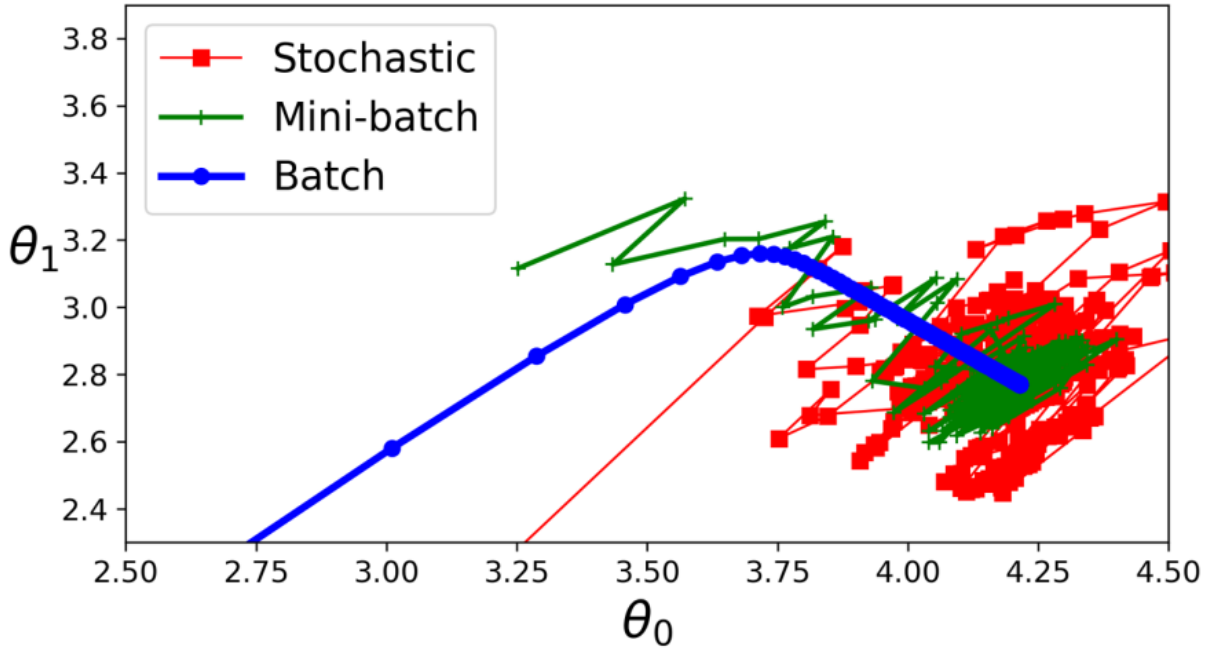
Linear Regression: Optimization

Batch GD iterated 1000 times through the whole training set, but Stochastic GD goes through the training set only 50 times and reaches a pretty good solution:



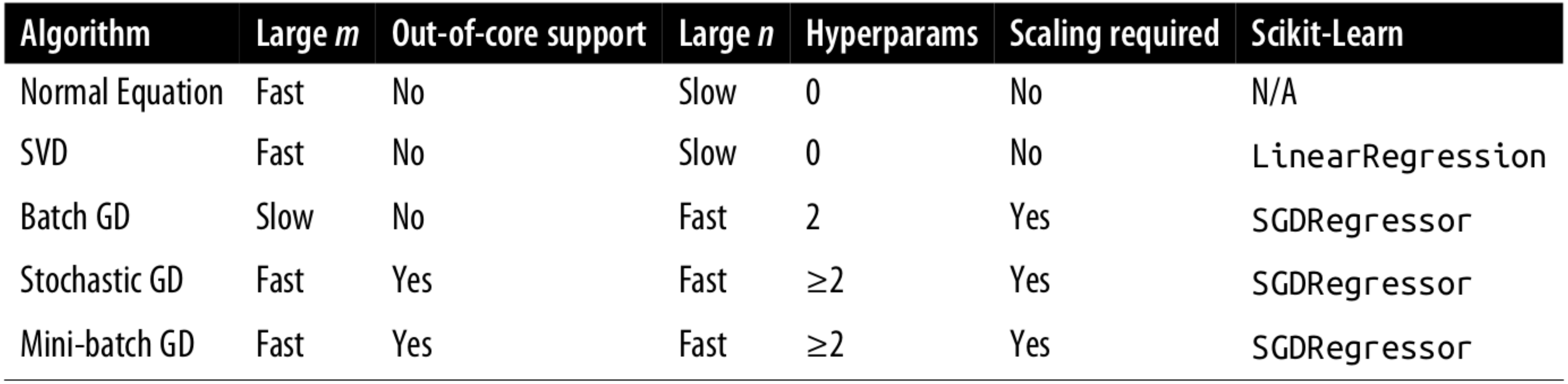
Linear Regression: Optimization

Mini-batch Gradient Descent:
Instead of computing the gradients based on the full training set (as in BGD) or based on just one instance (as in SGD), Mini-batch GD computes the gradients on small random sets of instances called mini-batches.



Linear Regression: Optimization

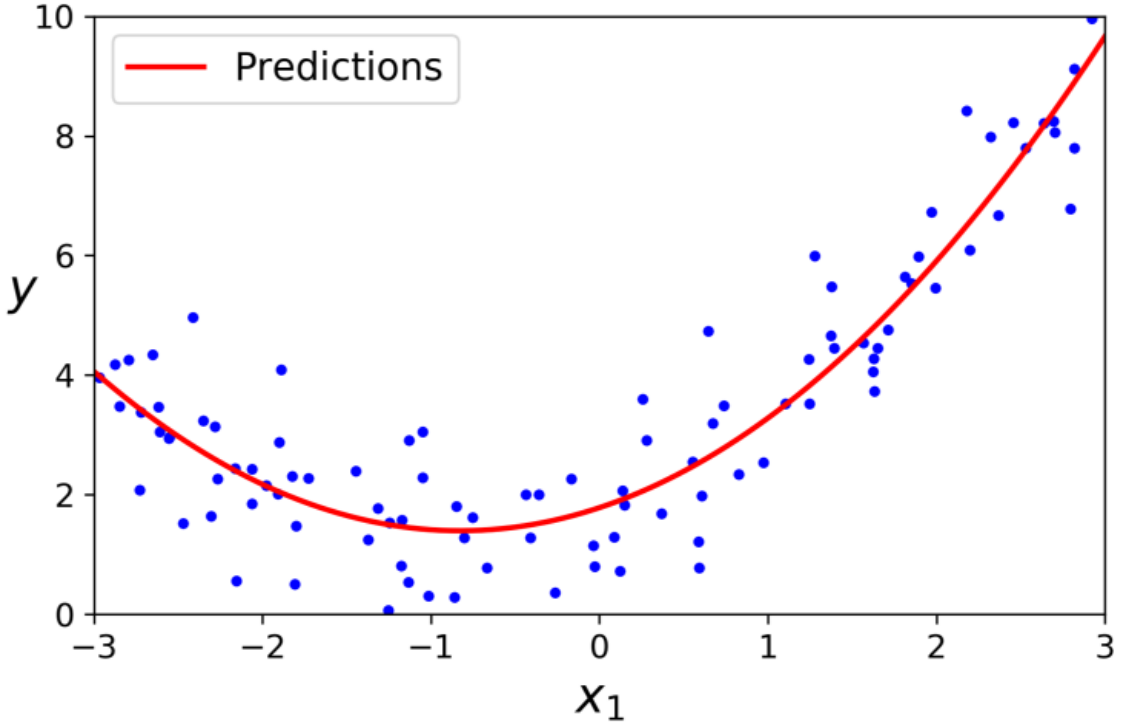
Comparison of the algorithms:



Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`

Polynomial Regression:

One way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features.



Original function:

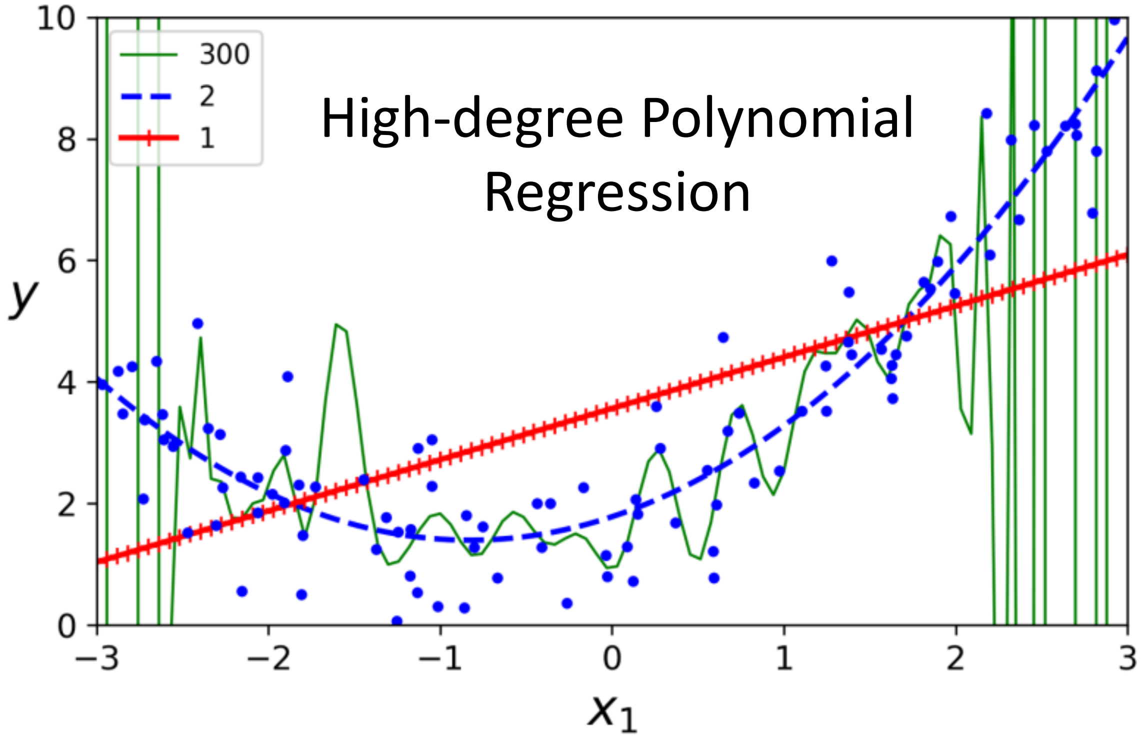
$$y = 0.5x_1^{\,2} + 1.0x_1 + 2.0 + \text{Gaussian noise}$$

Model estimation:

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

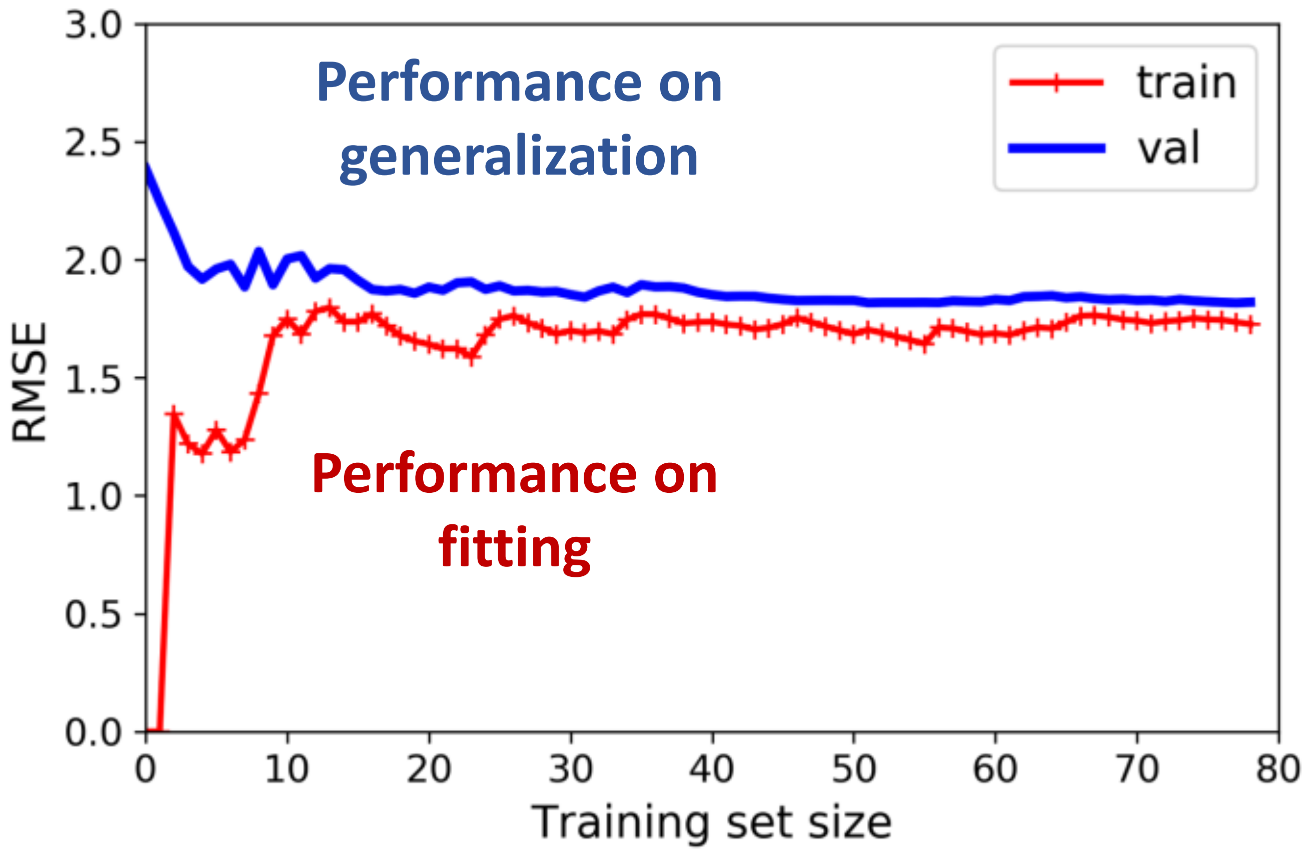
Learning Curves

They are plots of the model's performance on the training set and the validation set as a function of the training set size. To generate the plots, you need to rain the model several times on different sized subsets of the training set.



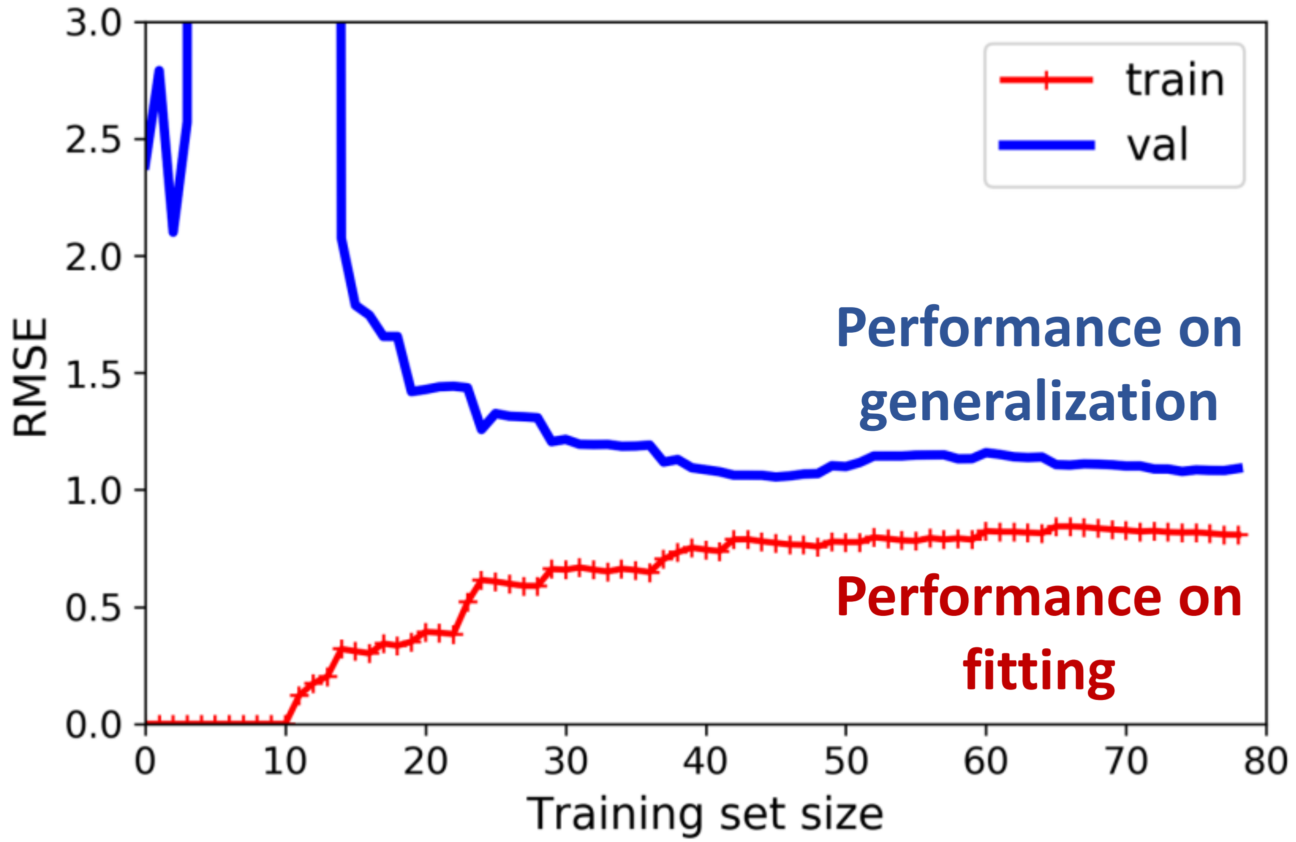
Learning Curves

The learning curves of the plain Linear Regression model:



Learning Curves

The learning curves for the 10th-degree polynomial model:



Performance on generalization

Performance on fitting

The Bias/Variance Trade-off

A model’s generalization error can be expressed as the sum of three very different errors:

- **Bias:** Wrong assumptions, such as assuming that the data is linear when it is actually quadratic.
- **Variance:** Model’s excessive sensitivity to small variations in the training data (e.g., the high-degree polynomial example).
- **Irreducible error:** Noisiness of the data (fix data sources and clean data)

Regularized Linear Models

A good way to reduce overfitting is to constrain the model. For example:

- **Reduce the number of polynomial degrees in a polynomial model.**
- **Constrain the weights of the model in a linear model:**
    1. **Ridge Regression**
    2. **Lasso Regression**
    3. **Elastic Net**

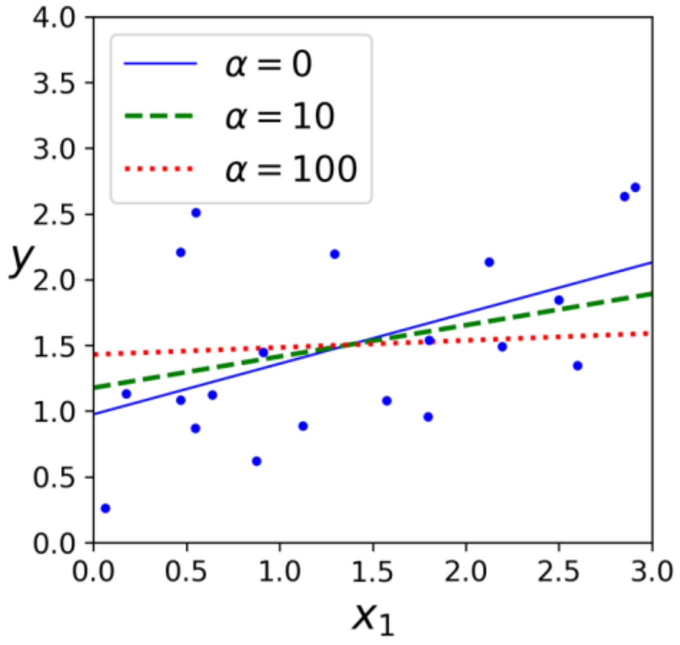
Ridge Regression

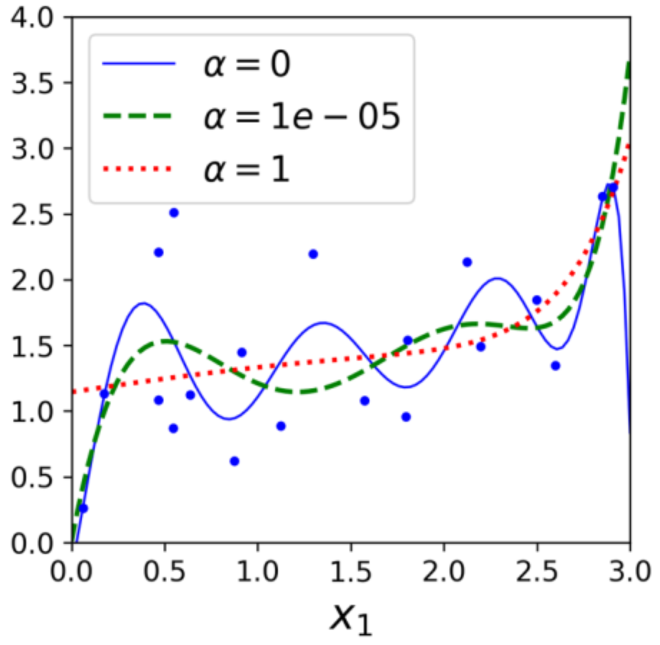
Ridge Regression Cost Function:

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha\frac{1}{2}\Sigma_{i=1}^{n}\theta_i^{\,2} = \mathrm{MSE}(\boldsymbol{\theta}) + ½(\|\ \mathbf{w}\ \|_2)^2$$

W being the vector of feature weights

Linear Model





Polynomial Model

Ridge Regression

It also has a closed-form equation:

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

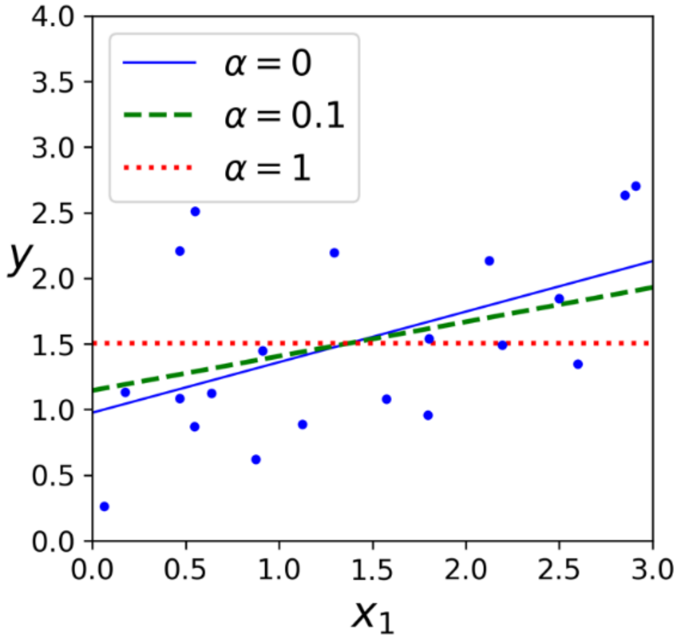
You can perform Ridge Regression either by computing the closed-form equation or by performing a Gradient Descent method.

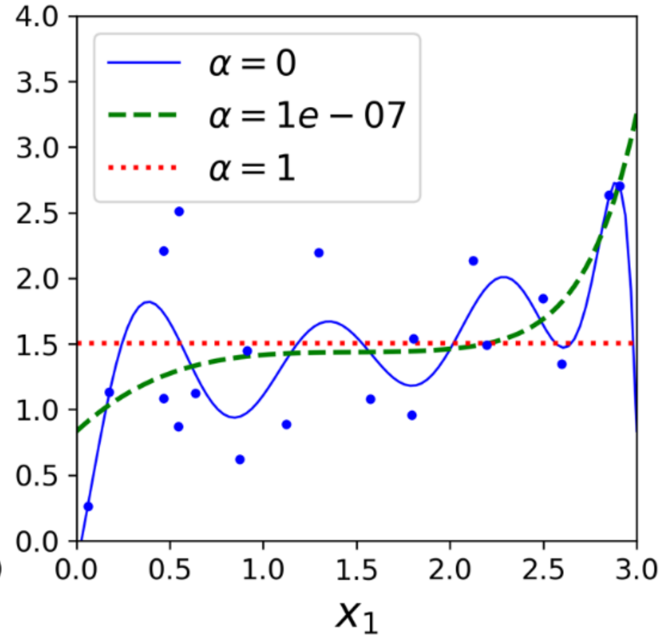
LASSO Regression

It is like Ridge except the fact that it uses the ℓ_1 norm of the weight vector instead of half the square of the ℓ_2 norm.

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

Linear Model





Polynomial Model

Elastic Net

The regularization term is a simple mix of both Ridge and Lasso's regularization terms, and you can control the mix ratio by tuning *r*.

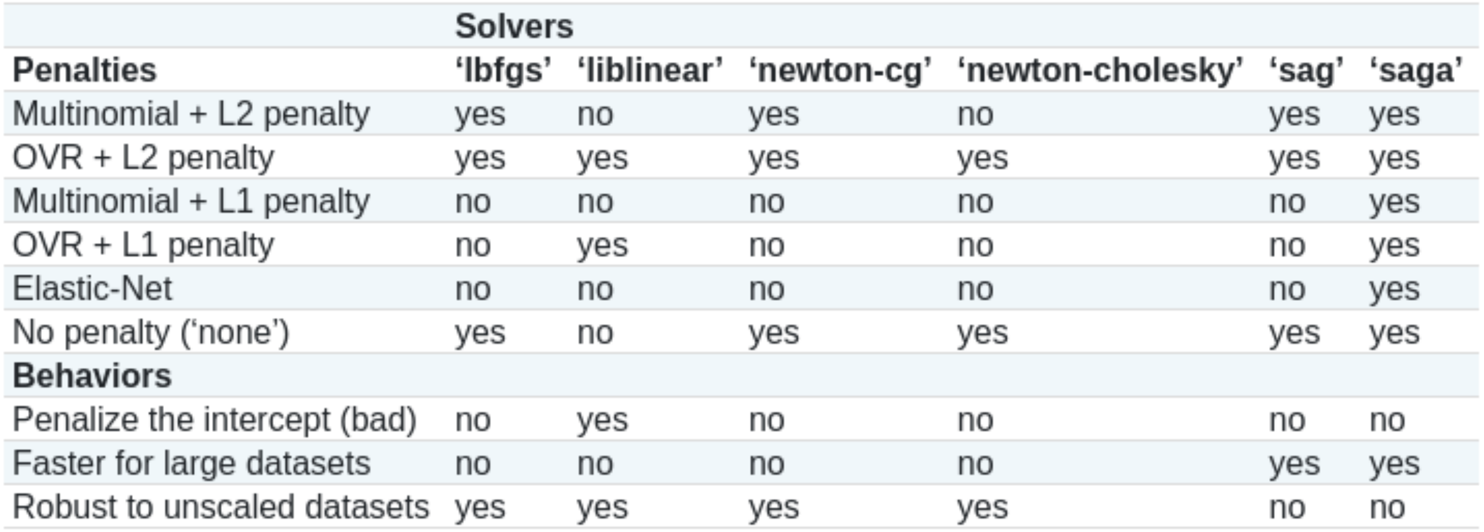
$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + r\alpha\Sigma_{i=1}^{n}\left|\theta_i\right| + \frac{1-r}{2}\alpha\Sigma_{i=1}^{n}\theta_i^{\,2}$$

Conclusion:

- **It is preferable to have always at least a little bit of regularization.**
- **Ridge is a good default, but if you suspect that only a few features are useful, you should prefer Lasso or Elastic Net, because they tend to reduce the useless features' weights down to zero.**

SciKit Learn Solvers

The following table summarizes the penalties supported by each solver:



	Solvers					
Penalties	**‘lbfgs’**	**‘liblinear’**	**‘newton-cg’**	**‘newton-cholesky’**	**‘sag’**	**‘saga’**
Multinomial + L2 penalty	yes	no	yes	no	yes	yes
OVR + L2 penalty	yes	yes	yes	yes	yes	yes
Multinomial + L1 penalty	no	no	no	no	no	yes
OVR + L1 penalty	no	yes	no	no	no	yes
Elastic-Net	no	no	no	no	no	yes
No penalty (‘none’)	yes	no	yes	yes	yes	yes
Behaviors						
Penalize the intercept (bad)	no	yes	no	no	no	no
Faster for large datasets	no	no	no	no	yes	yes
Robust to unscaled datasets	yes	yes	yes	yes	no	no

Please review this webpage.