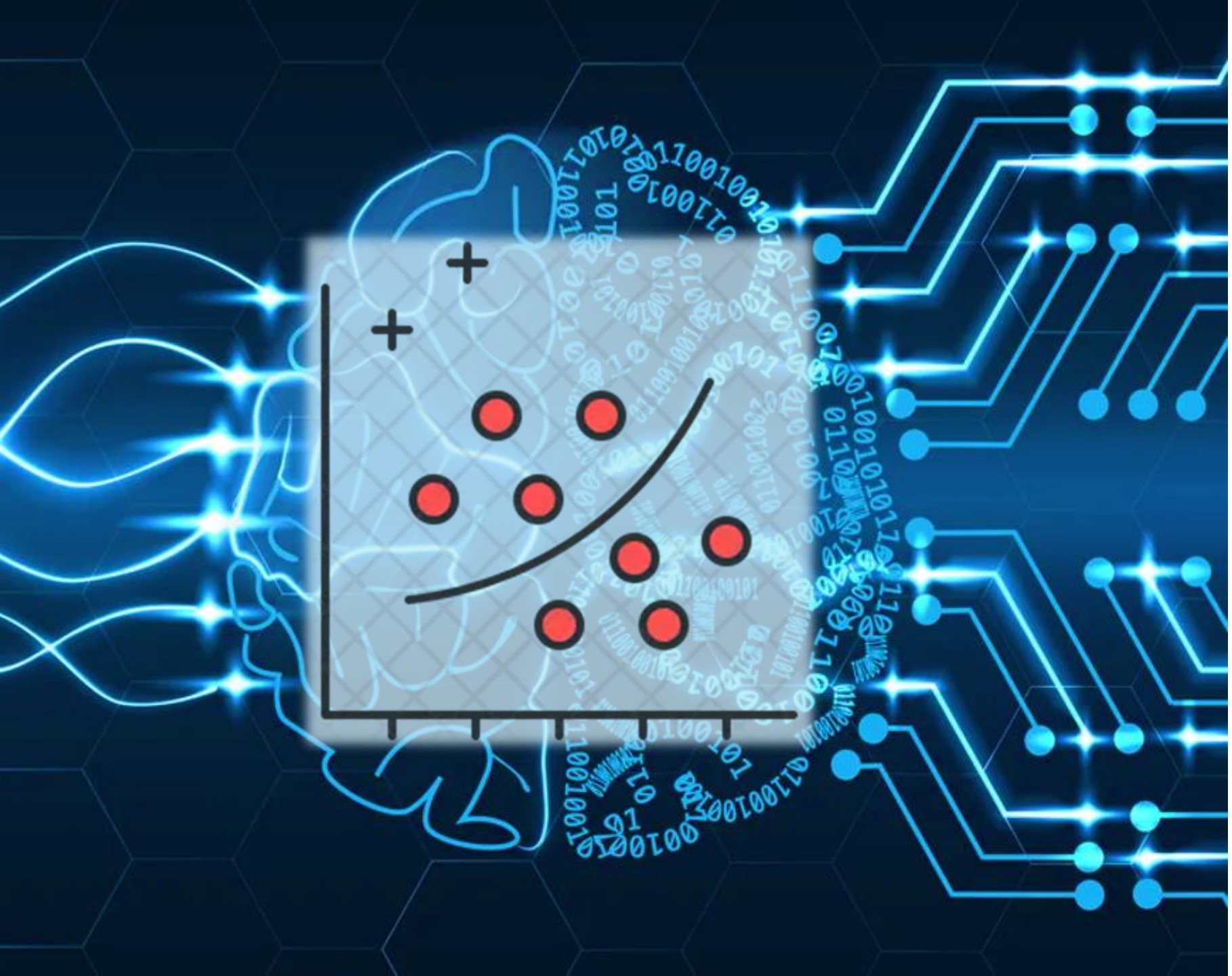


طبقه بندی

(Regression)

اسفند ۱۴۰۱

دانشکده مهندسی صنایع

دانشگاه صنعتی شریف

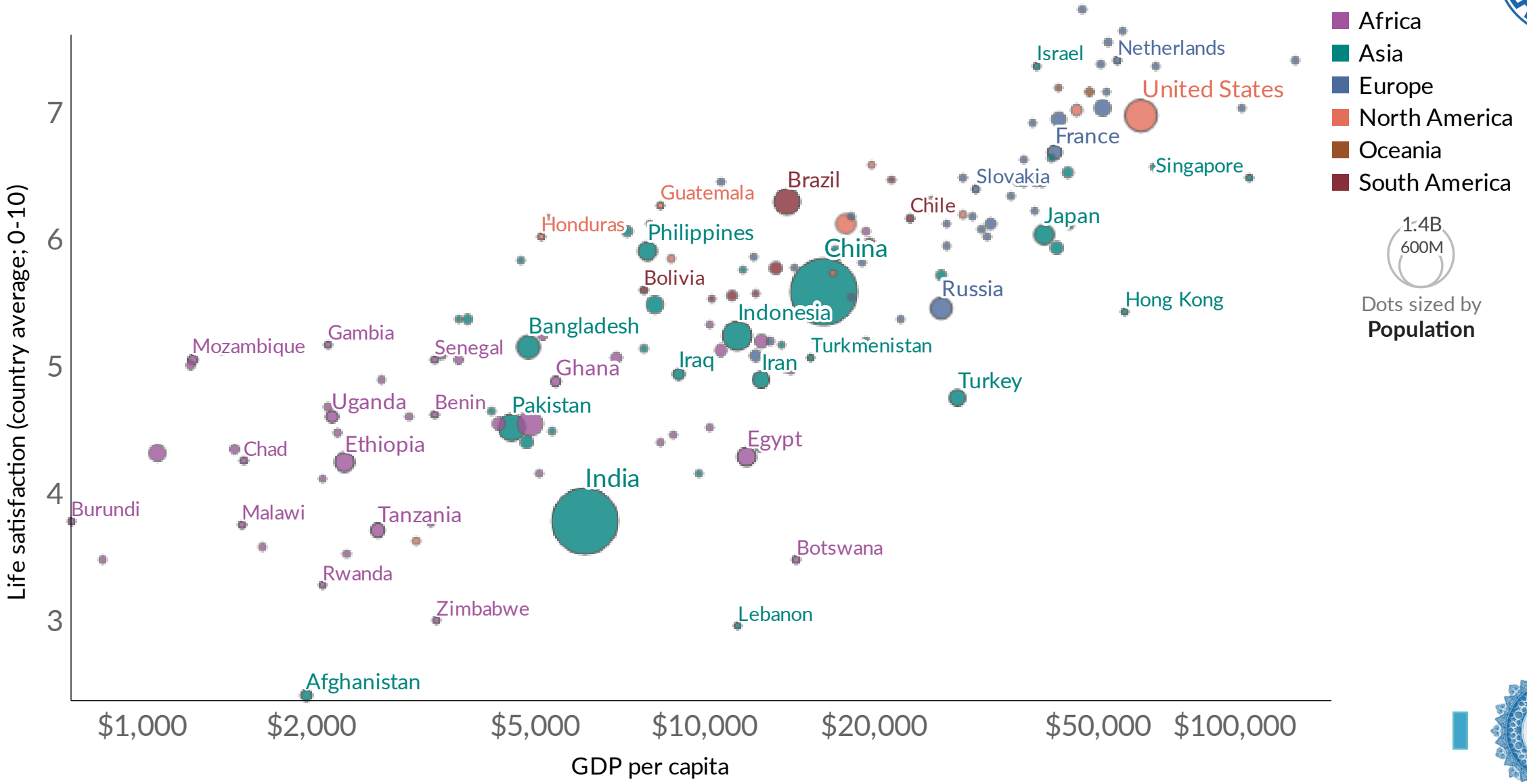
دکتر مهدی شریف‌زاده

آئیریا محمدی


بسم الله الرحمن الرحیم

فهرست مطالب

A Simple Linear Regression Model



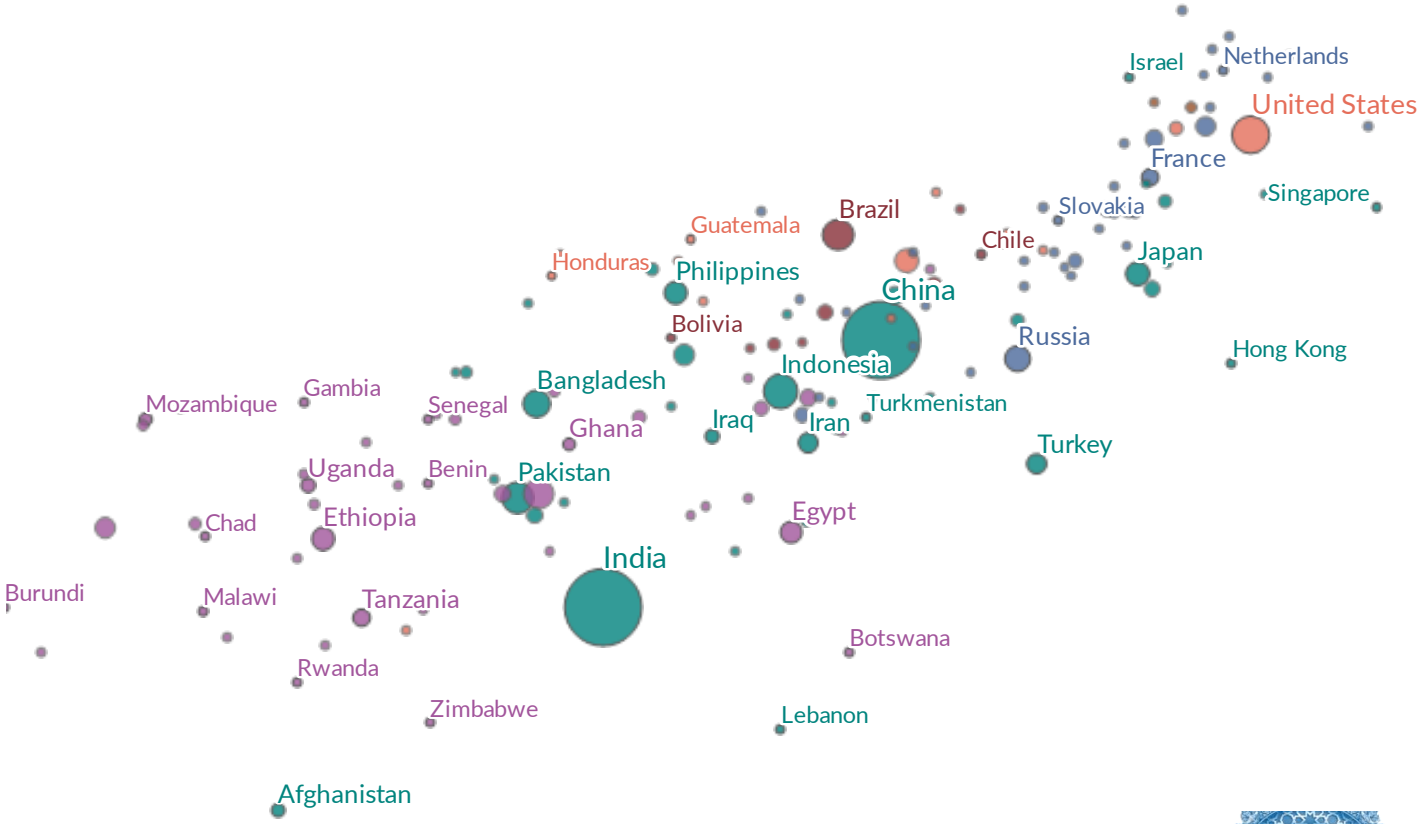
A Simple Linear Regression Model

$$\text{Life satisfaction} = \theta_0 + \theta_1 \times (\text{GDP per capita})$$

Model type: Linear function

Input feature: GDP per capita

Model's parameters: θ_0, θ_1



Linear Model

A linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the *bias term* (also called the *intercept term*), as shown here:

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

In this equation:

- $\hat{y}$ 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

Vectorized form of the linear regression model prediction equation:

$$\hat{y} = \boldsymbol{\theta}^T \boldsymbol{x}$$

$$\boldsymbol{x} = [1\ , x_1\ , \dots , xn]^T$$

$$\boldsymbol{\theta} = [\theta_0\ , \theta_1\ , \dots , \theta_n]^T$$

Where $\boldsymbol{\theta}^T$ is the transpose of $\boldsymbol{\theta}$.

Notice that $\boldsymbol{x}$ and $\boldsymbol{\theta}$ are column vectors and $\boldsymbol{\theta}^T \boldsymbol{x}$ *donotes matrix multiplication of* $\boldsymbol{\theta}^T$ and $\boldsymbol{x}$ which is also equal to the dot product of $\boldsymbol{\theta}$ and $\boldsymbol{x}$.

Matrix Form

Prediction on a dataset X:

$$\hat{\boldsymbol{y}} = \boldsymbol{\theta}^T \boldsymbol{X}$$

$$\boldsymbol{X} = \begin{pmatrix} x_0^1 & \cdots & x_n^1 \\ \vdots & \ddots & \vdots \\ x_0^m & \cdots & x_n^m \end{pmatrix} = \begin{pmatrix} (x^1)^T \\ \vdots \\ (x^m)^T \end{pmatrix}$$

$$\hat{\boldsymbol{y}} = \begin{pmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_n \end{pmatrix}$$

(x^i) is the i^{th} row of the dataset

m: size of dataset

n: number of feature

x_0^i is always equal to 1

The need for a performance measure

How do we **train** a linear regression model?

Training a model means setting its **parameters** so that the model best fits the training set.

For this purpose, we first need a measure of how well (or poorly) the model **fits** the training data.

How to train a regression model

The most common performance **measure** of a regression model is the root mean square **error**.

Therefore, to train a linear regression model, we need to find the value of **θ** that **minimizes** the RMSE.

In practice, it is simpler to minimize the mean squared error (**MSE**) than the RMSE, and it leads to the same result.

Root Mean Square Error (RMSE) & *Mean Absolute Error (MAE)*

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2} = \|\hat{y} - \mathrm{y}\|_2$$

$$MAE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}|\hat{y}_i - y_i|} = \|\hat{y} - \mathrm{y}\|_1$$

Although the RMSE is generally the preferred performance measure for regression tasks, in some contexts you may prefer to use another function.

For example, if there are many **outlier data**. In that case, you may consider using the MAE (also called the *average absolute deviation*).

The Normal Equation

To find the value of **θ** that minimizes the MSE, there exists a *closed-form solution*—in other words, a mathematical equation that gives the result directly. This is called the *Normal equation.*

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

In this equation:

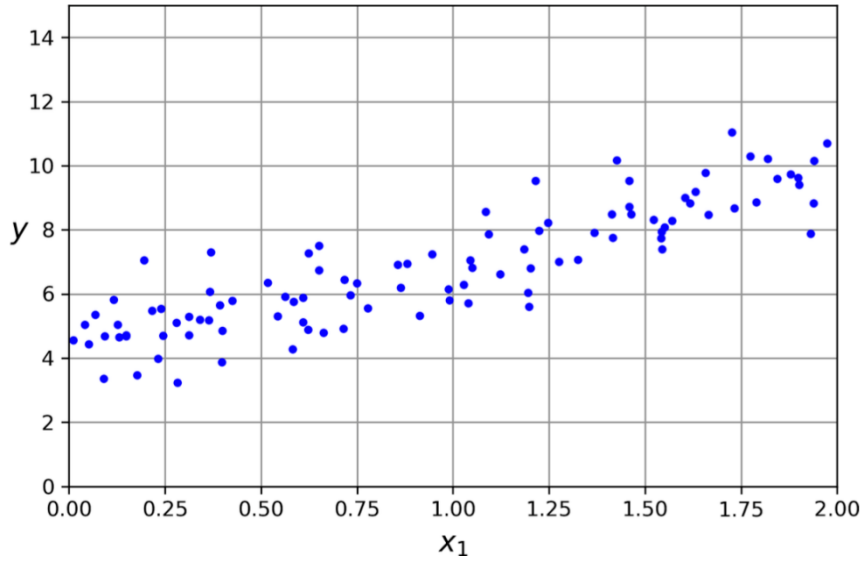
$\hat{\theta}$ is the value of **θ** that minimizes the cost function.

y is the vector of target values containing y_1 to y_m.

Code Demonstration

Let’s generate some linear-looking data to test this equation on:

```
import numpy as np

np.random.seed(42) # to make this code example reproducible

m = 100 # number of instances
X = 2 * np.random.rand(m, 1) # column vector
y = 4 + 3 * X + np.random.randn(m, 1) # column vector
```



A randomly generated linear dataset

Demonstration

```
from sklearn.preprocessing import add_dummy_feature

X_b = add_dummy_feature(X)  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```

The @ operator performs matrix multiplication. If A and B are NumPy arrays, then

A @ B is equivalent to np.matmul(A, B).

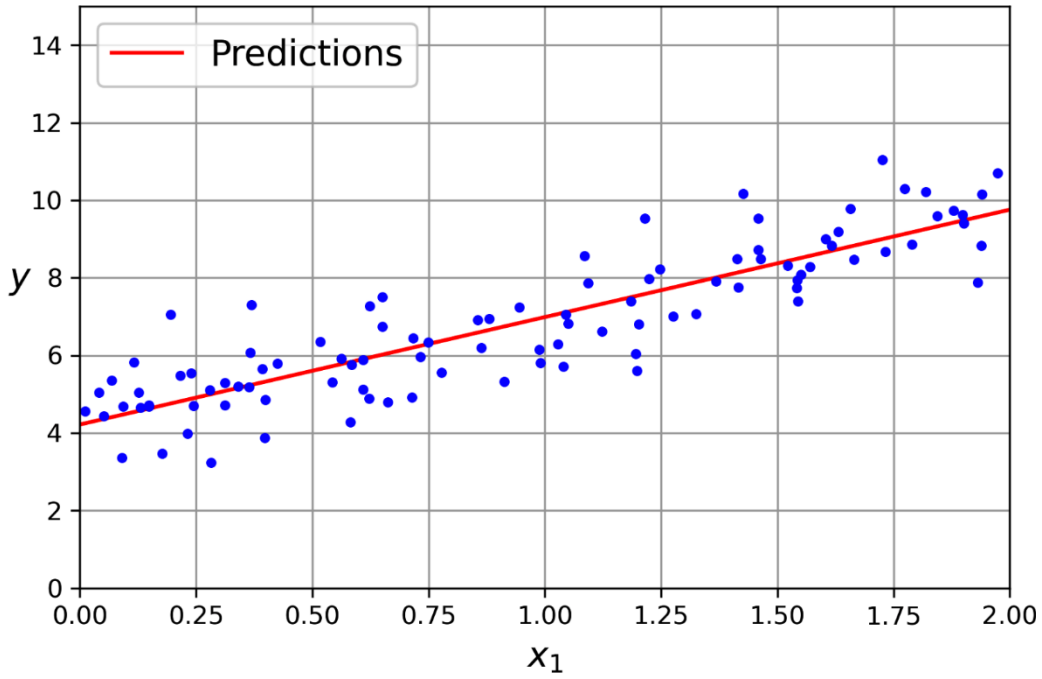
Now, we can make predictions using $\hat{\theta}$:

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = add_dummy_feature(X_new)  # add x0 = 1 to each instance
>>> y_predict = X_new_b @ theta_best
>>> y_predict
array([[4.21509616],
       [9.75532293]])
```

Demonstration

Let’s plot this model’s predictions:

```
import matplotlib.pyplot as plt

plt.plot(X_new, y_predict, "r-", label="Predictions")
plt.plot(X, y, "b.")
[...]  # beautify the figure: add labels, axis, grid, and legend
plt.show()
```



Using Scikit-Learn

Performing linear regression using Scikit-Learn is relatively straightforward:

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

Scikit-Learn Linear Regression Model.predict()

This function computes $\hat{\theta} = \mathbf{X}^{+}\mathbf{y}$, where $\mathbf{X}^{+}$ is the *pseudoinverse* of $\mathbf{X}$ (specifically, the Moore–Penrose inverse). You can use np.linalg.pinv() to compute the pseudoinverse directly.

The pseudoinverse itself is computed using a standard matrix factorization technique called *singular value decomposition* (SVD) that can decompose the training set matrix $\mathbf{X}$ into the matrix multiplication of three matrices $\mathbf{U}\,\mathbf{\Sigma}\,\boldsymbol{V}^{T}$ (see numpy.linalg.svd()).

Scikit-Learn Linear Regression Model.predict()

The pseudoinverse is computed as $\mathbf{X}^{+} = \mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^{T}$. To compute the matrix $\mathbf{\Sigma}^{+}$, the algorithm takes $\mathbf{\Sigma}$ and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their inverse, and finally it transposes the resulting matrix. This approach is more efficient than computing the Normal Equation, plus it handles edge cases nicely: indeed, the Normal Equation may not work if the matrix $\mathbf{X}^{T}\mathbf{X}$ is not invertible (i.e., singular), such as if $m < n$ or if some features are redundant, but the pseudoinverse is always defined.

Computational Complexity

The Normal equation computes the inverse of $\boldsymbol{X}^T\boldsymbol{X}$, which is an (n + 1) × (n+ 1) matrix (where n is the number of features).The *computational complexity* of inverting an $n \times n$ matrix is typically about $O(n^{2.4})$ to $O(n^3)$, depending on the implementation. In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$.

The SVD approach used by Scikit-Learn's LinearRegression class is about $O(n^2)$, which means if you double the number of features, you multiply the computation time by roughly 4.

Computational Complexity

Both the Normal Equation and the SVD approach get very slow when the number of features grows large (e.g., 100,000). On the positive side, both are linear with regard to the number of instances in the training set (they are O(m)), so they handle large training sets efficiently, provided they can fit in memory.

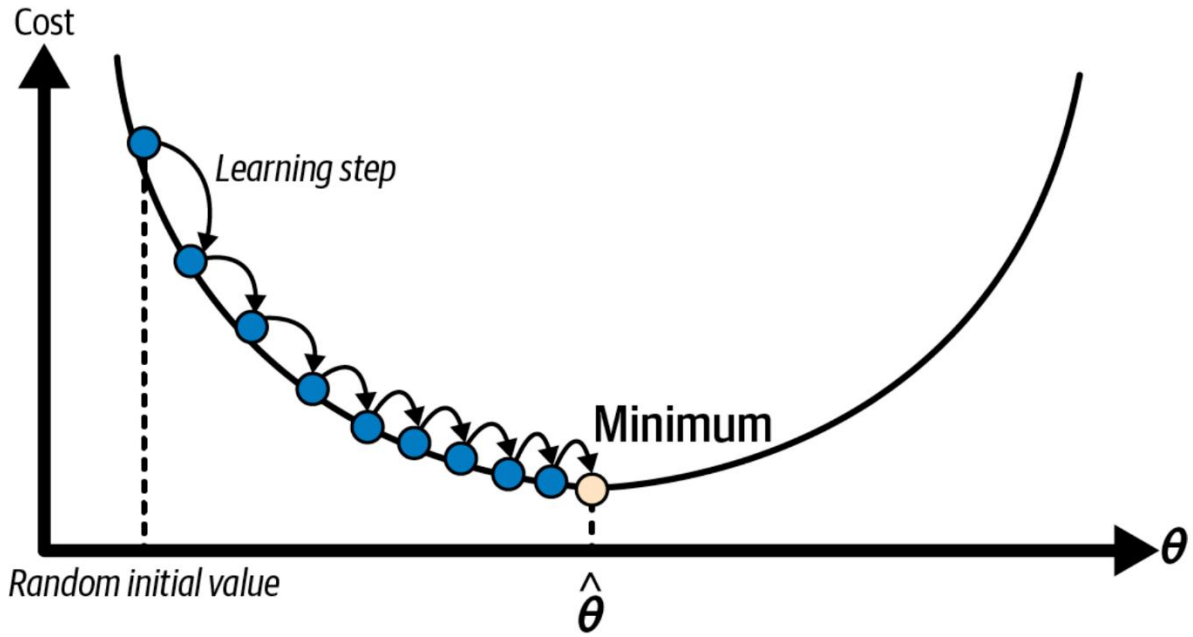
The need for another algorithm

- Regressen’s closed formula (the normal equation) is computationally expensive.
- Not necessarily all methods have closed form solutions.

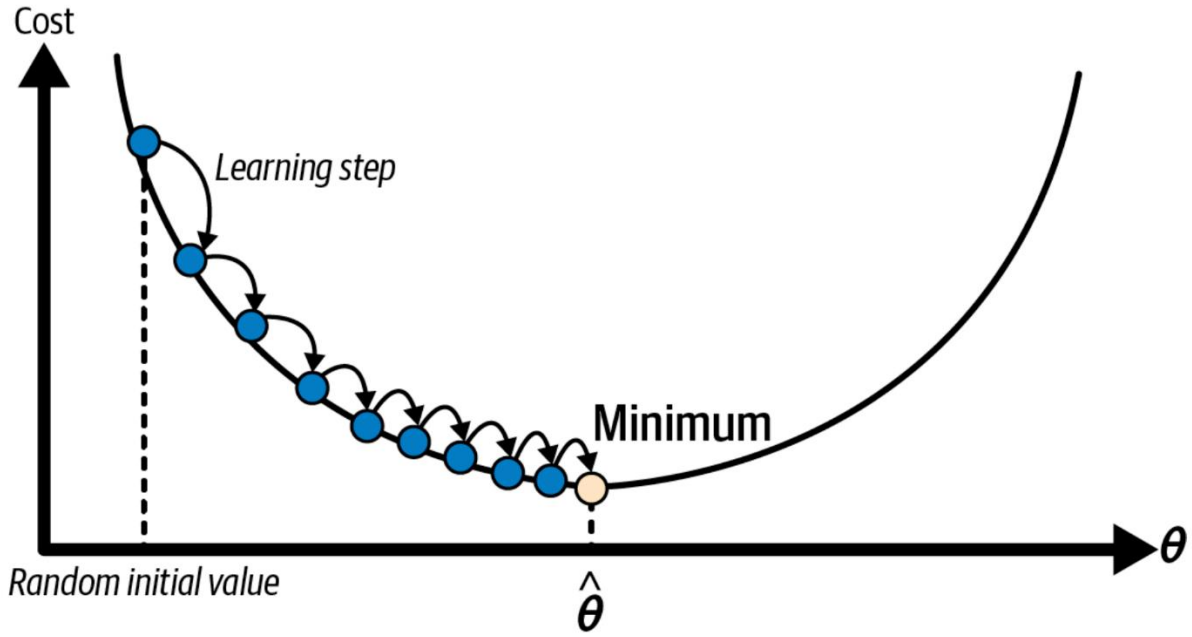
Hence, we need an iterative method to minimize loss function.

Gradient Descent

Gradient descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of gradient descent is to tweak parameters iteratively in order to minimize a cost function.



Gradient Descent



In this depiction of gradient descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the cost approaches the minimum.

Gradient Descent Step Size

An important parameter in gradient descent is the size of the steps, determined by the *learning rate* hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time.

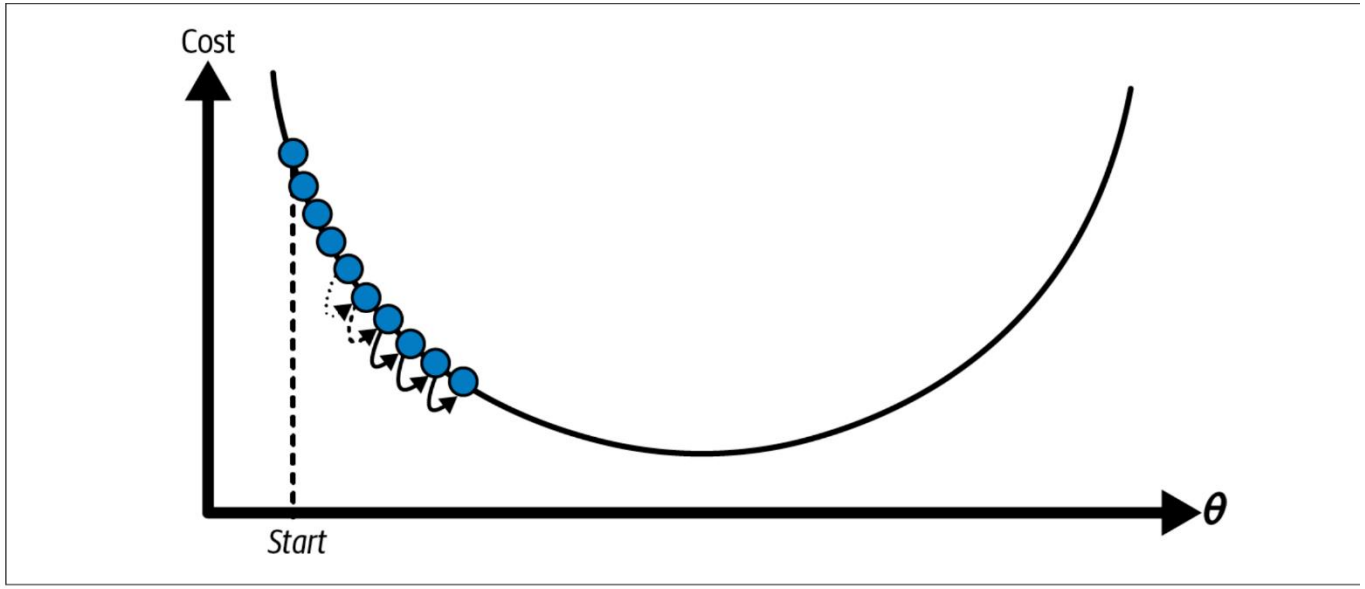


Figure 4-4. Learning rate too small

Gradient Descent Step Size

On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution.

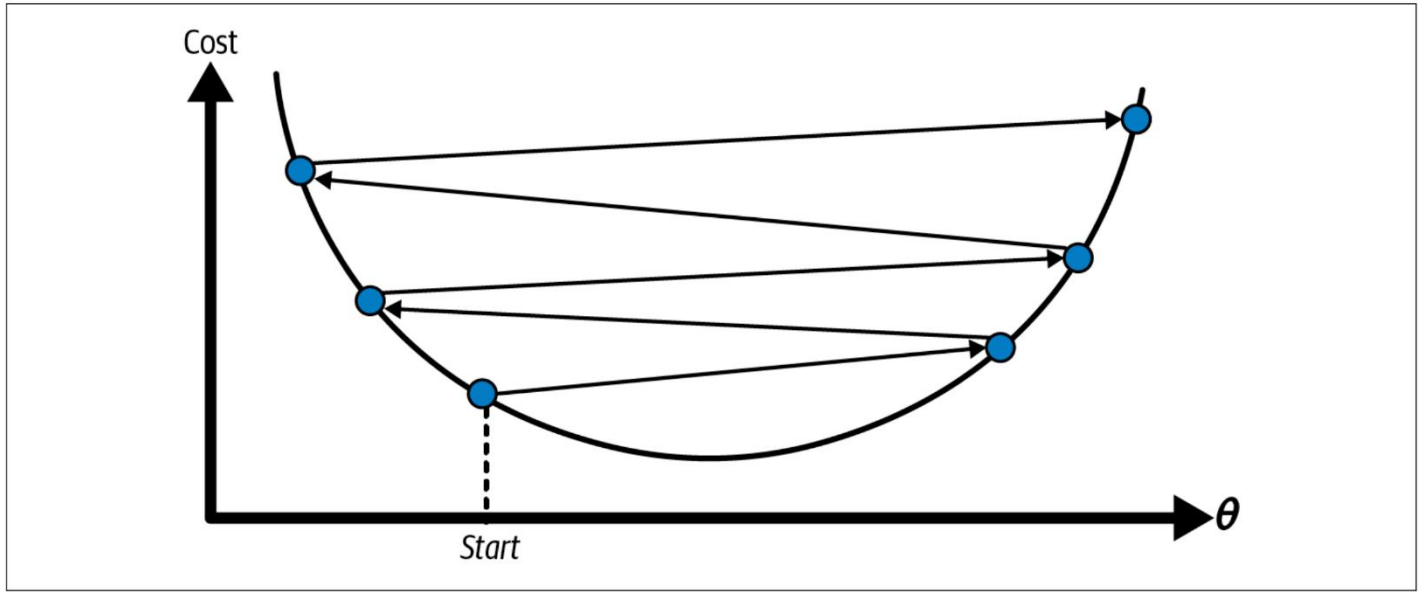


Figure 4-5. Learning rate too high

Gradient Descent Pitfalls

Not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrain, making convergence to the minimum difficult. Figure 4-6 shows the two main challenges with gradient descent.

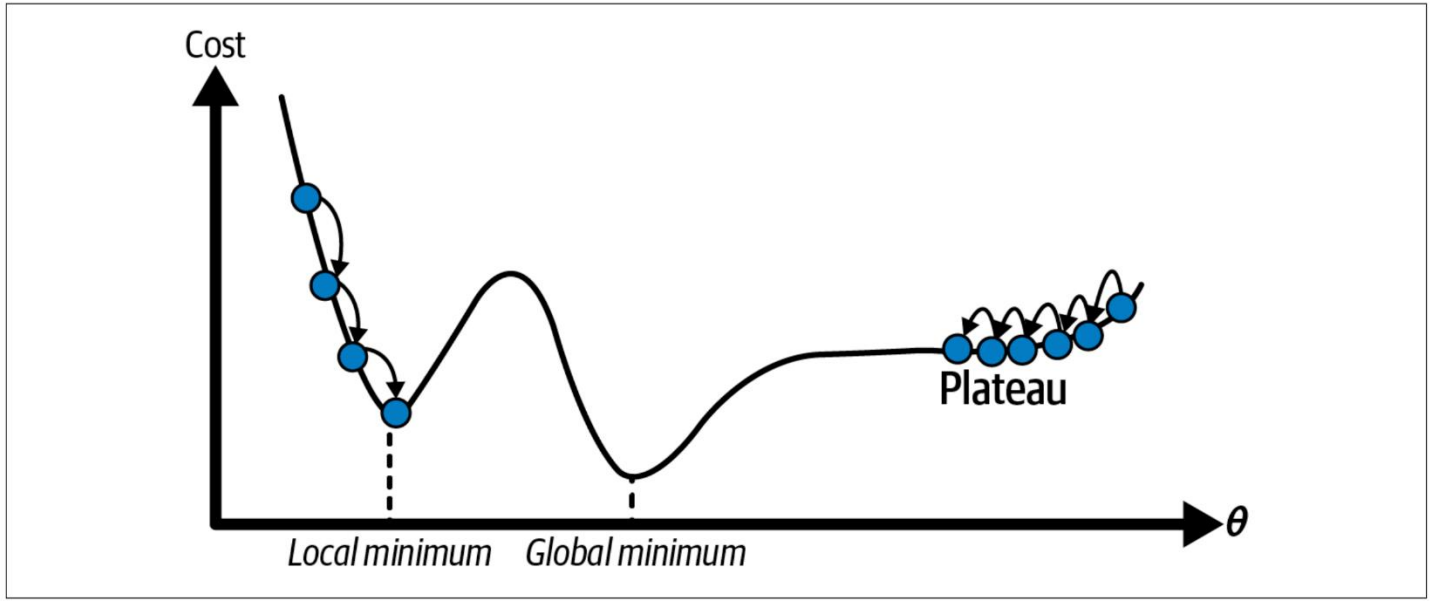
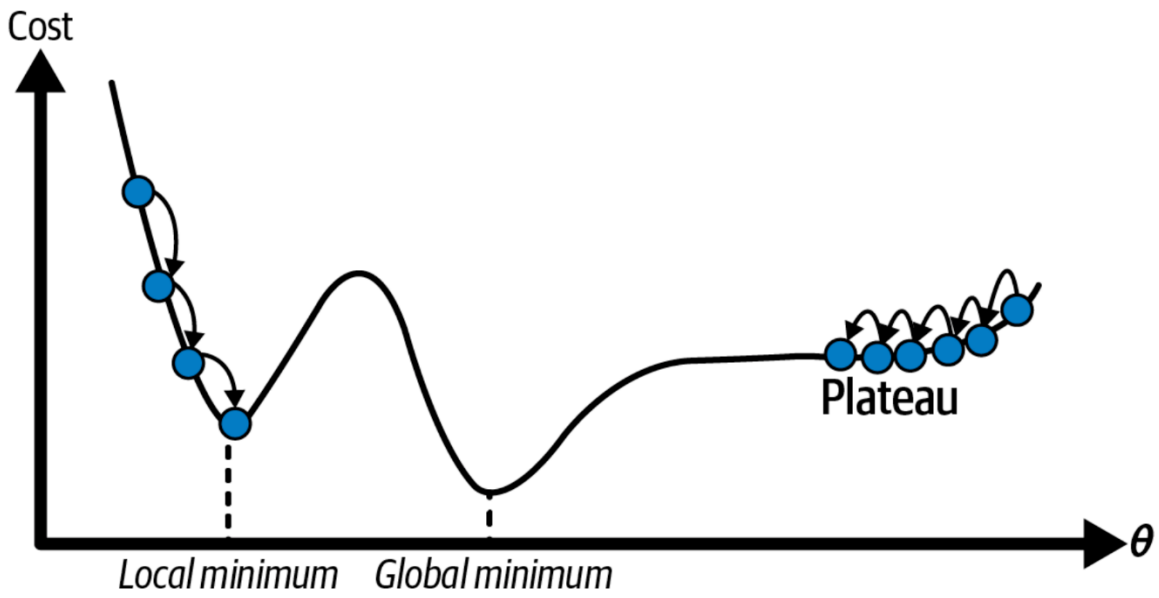


Figure 4-6. Gradient descent pitfalls

Gradient Descent Pitfalls

If the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*. If it starts on the right, then it will take a very long time to cross the plateau. And if you stop too early, you will never reach the global minimum.



Gradient Descent Pitfalls

But good news! The MSE cost function for a linear regression model is a *convex function with the shape of a bowl; It has just one local minimum, it is continuous, and its slope never changes abruptly.*

These two facts mean gradient descent is guaranteed to approach arbitrarily closely the global minimum (if you wait long enough and if the learning rate is not too high).

Gradient Descent Pitfalls

Tip: When using gradient descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's StandardScaler class), or else it will take much longer to converge.

Gradient Vector

To implement gradient descent, you need to compute the gradient of the cost function with regard to each model parameter θ_j.

The gradient vector, noted $\nabla_{\boldsymbol{\theta}} MSE(\boldsymbol{\theta})$, contains all the partial derivatives of the cost function.

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

Learning Rate

Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting $\nabla_{\boldsymbol{\theta}} MSE(\boldsymbol{\theta})$ from $\boldsymbol{\theta}$.

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

The learning rate $\boldsymbol{\eta}$ determines the size of the downhill step.

Learning Rate Selection

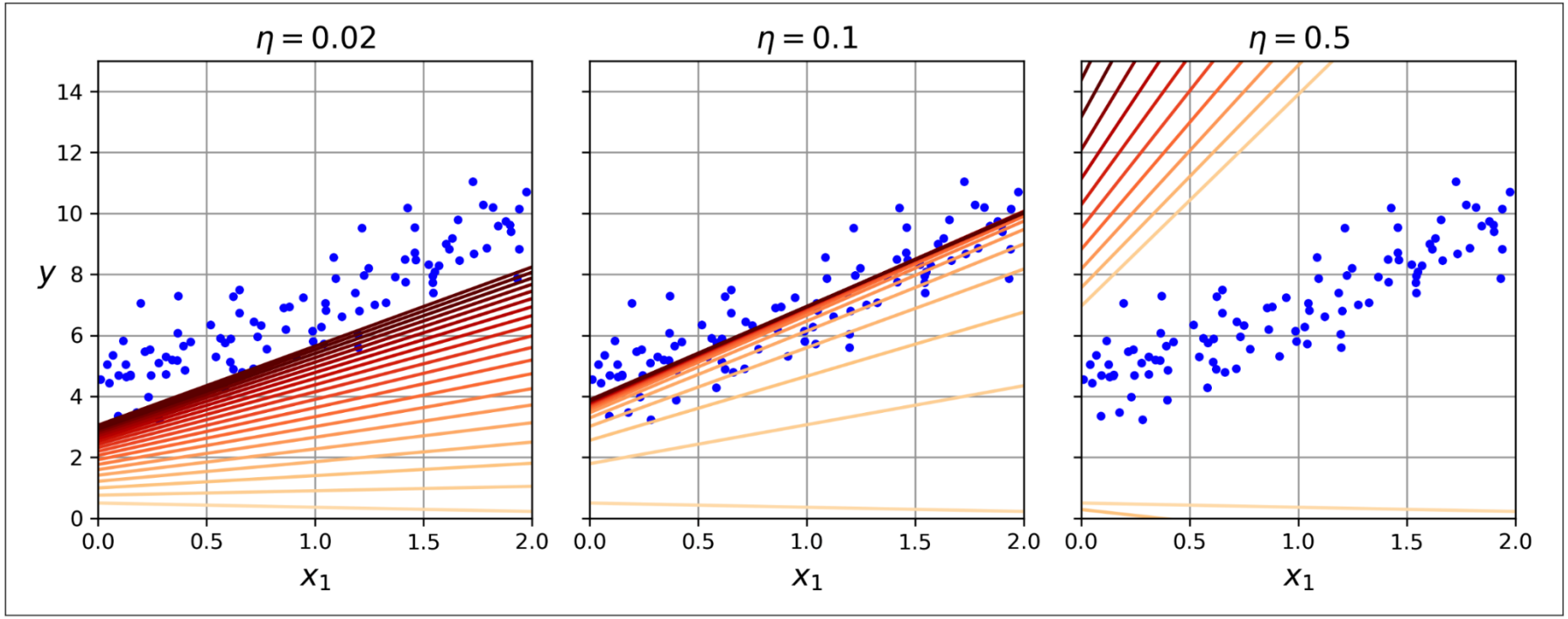


Figure 4-8. Gradient descent with various learning rates

To find a good learning rate, you can use grid search tool. Also don't forget to limit the number of epochs!

Epoch Number Selection

If too low: may not reach optimal solution

If too high: too much time will be wasted

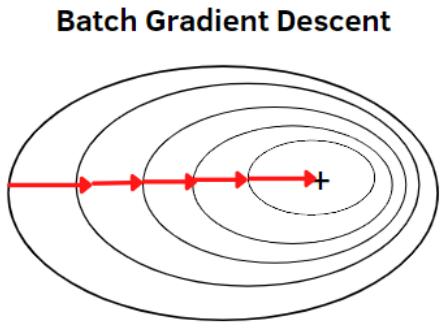
Solution: Interrupt algorithm when the gradient vector becomes tiny (i.e., when its norm becomes smaller than a tiny number $\boldsymbol{\varepsilon}$ called the *tolerance*).

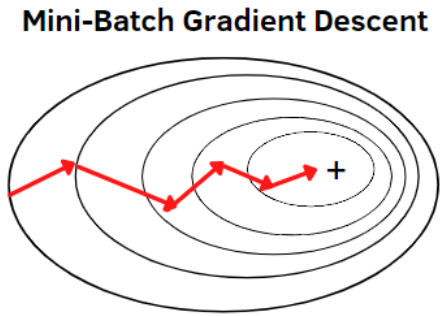
Gradient Descent Variants

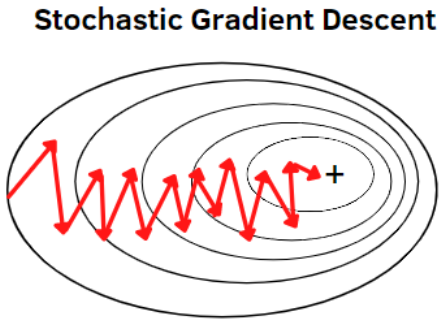
In this course we'll cover batch gradient descent, stochastic gradient descent and mini-batch gradient descent. Their biggest difference is in how they *evaluate the gradient vector*. Or which part of the dataset they use to compute gradients based on.

Gradient Descent Variants

- **Batch gradient descent**: computes gradients based on full training set
- **Stochastic gradient descent**: based on just one instance at each iteration
- **Mini-batch gradient descent**: based on small random sets of instances







Batch Gradient Descent

The formula of this algorithm involves calculations over the full training set **X** (the whole batch), at each gradient descent step. As a result, it is terribly *slow* on very large training sets. However, gradient descent *scales well* with the number of features; better than using the Normal equation or SVD.

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

Batch GD Implementation

Let’s look at a quick implementation of this algorithm:

```
eta = 0.1  # learning rate
n_epochs = 1000
m = len(X_b)  # number of instances

np.random.seed(42)
theta = np.random.randn(2, 1)  # randomly initialized model parameters

for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients
```

Each iteration over the training set is called an *epoch.*

Convergence Rate & Condition

When the cost function is **convex** and its slope does not change abruptly (as is the case for the MSE), batch gradient descent with a fixed learning rate will eventually **converge** to the optimal solution.

But you may have to wait a while: it can take $O(1/\varepsilon)$ iterations to reach the optimum within a range of ε, depending on the shape of the cost function. If you divide the tolerance by **10** to have a more precise solution, then the algorithm may have to run about **10 times** longer.

Stochastic Gradient Descent

At the opposite extreme of batch gradient descent, *stochastic gradient descent* picks a **random** instance in the training set at every step and computes the gradients based only on that single instance.

Obviously, working on a single instance at a time makes the algorithm much **faster** because it has very little data to manipulate at every iteration. It also makes it possible to train on huge training sets, since only **one** instance needs to be in memory at each iteration.

Stochastic Gradient Descent

On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than batch gradient descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average.

Scikit-Learn Demonstration

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-5, penalty=None, eta0=0.01,
                       n_iter_no_change=100, random_state=42)
sgd_reg.fit(X, y.ravel())  # y.ravel() because fit() expects 1D targets
```

Mini-Batch Gradient Descent

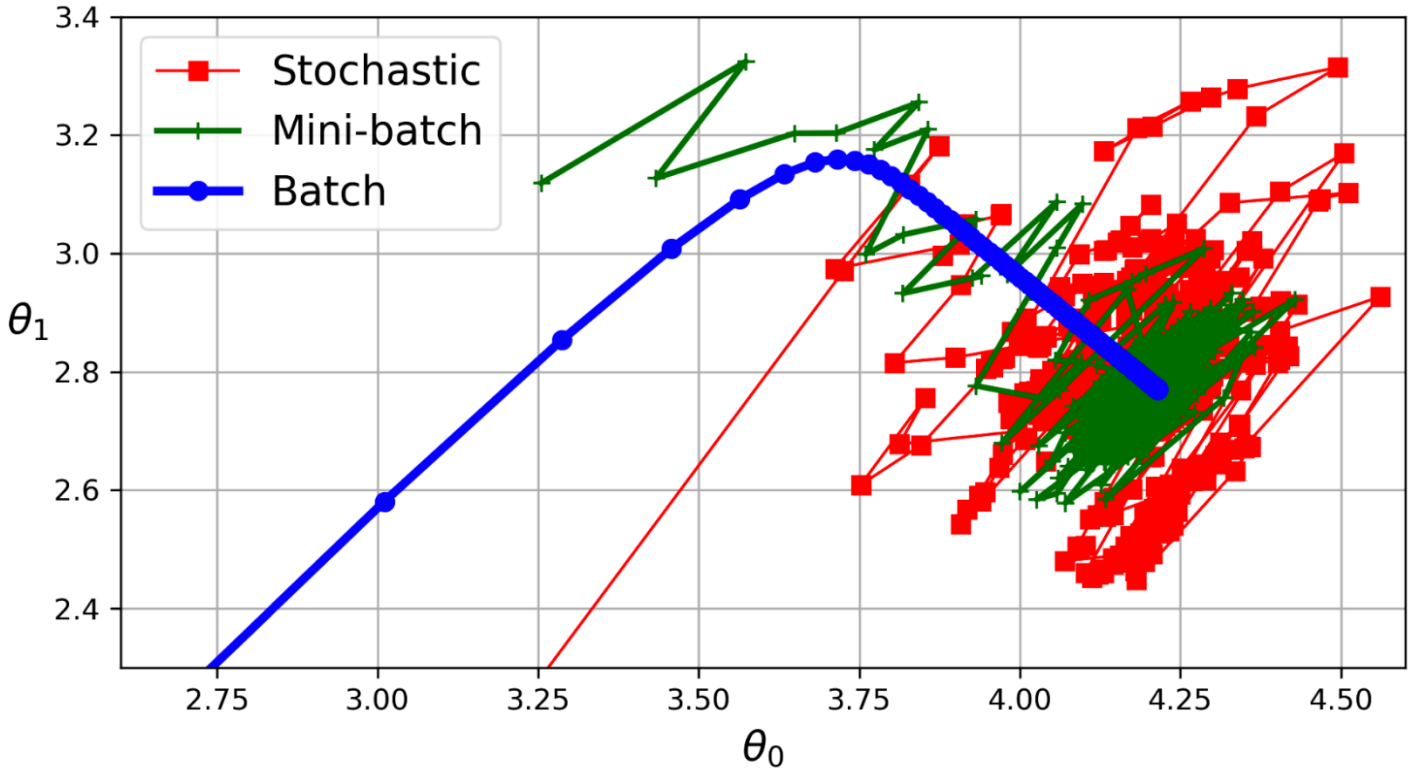
The last gradient descent algorithm we will look at is called *mini-batch gradient descent*. At each step, instead of computing the gradients based on the **full** training set (as in batch GD) or based on just **one instance** (as in stochastic GD), mini-batch GD computes the gradients on small **random sets** of instances called *mini-batches*.

Mini-Batch Gradient Descent Comparison

The main advantage of mini-batch GD over stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs. And its progress in parameter space is less erratic than with stochastic GD, so it will end up walking closer to the minimum than stochastic GD—but it may be harder for it to escape from local minima (in some case of problems).

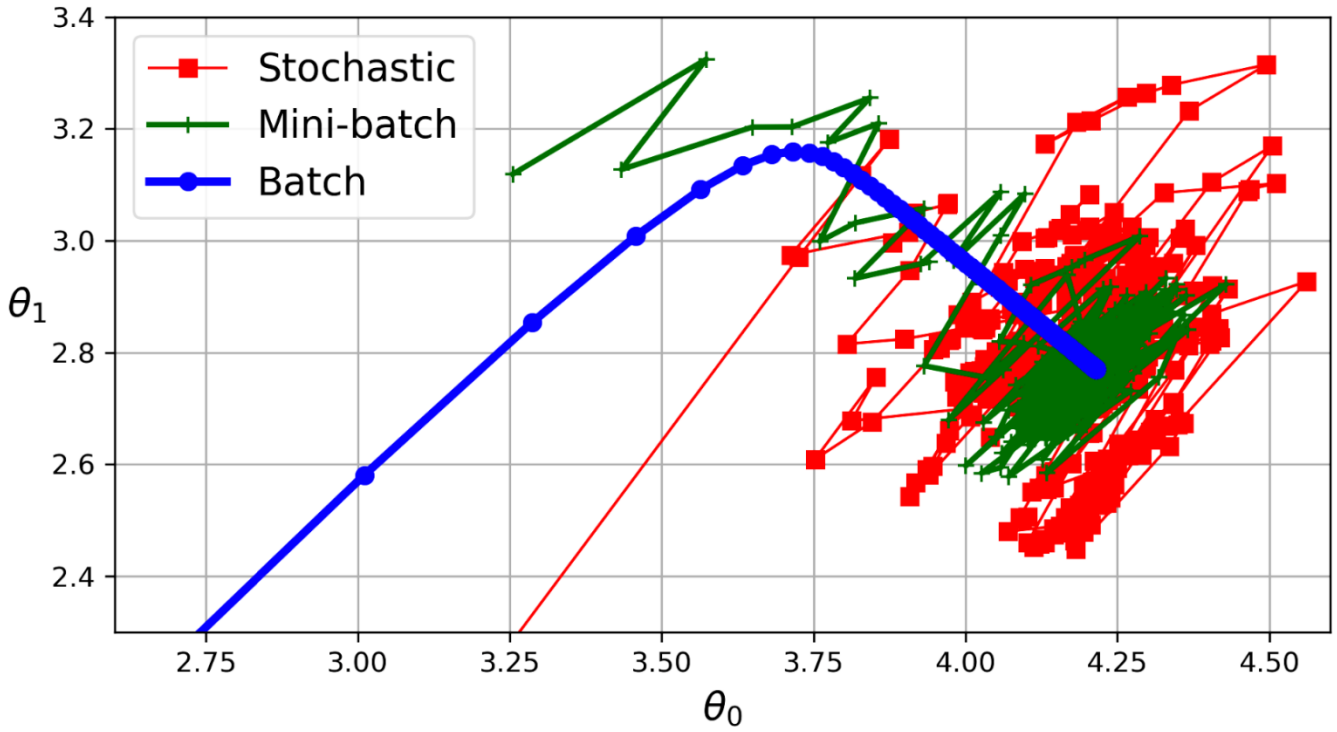
Gradient Descent Algorithm Paths

The following figure shows the paths taken by the three gradient descent algorithms in parameter space during training.



Gradient Descent Algorithm Paths

They all end up near the minimum, but batch GD's path actually stops at the minimum, while others continue to walk around. However, don't forget that batch GD takes a lot of time to take each step, and stochastic GD and mini-batch GD would also reach the minimum if you used a good learning schedule.



Algorithm Comparison

The following table compares the algorithms we've discussed so far for linear regression (*m* is the number of training instances and *n* is the number of features).

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	N/A
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	N/A

all these algorithms end up with very similar models and make predictions in exactly the same way after training.

Polynomial Regression

What if your data is more complex than a straight line? Surprisingly, you can use a linear model to fit nonlinear data. A simple 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. This technique is called *polynomial regression*.

Scikit-Learn Demonstration

We'll use Scikit-Learn's PolynomialFeatures class to transform our training data, adding the square (second-degree polynomial) of each feature in the training set as a new feature.

```
>>> 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])
```

Scikit-Learn Demonstration

X_poly now contains the original feature of X plus the square of this feature. Now we can 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]]))
```

Cross-Validation

One way to evaluate a model would be to split the data into a smaller training set and a validation set using *train_test_split(),* then train your models against the now smaller training set and evaluate them against the validation set. A great alternative is to use Scikit-Learn's *k-fold cross-validation* feature.

Scikit-Learn Cross-Validation

cross_val_score() randomly splits the training set into k (default: 10) nonoverlapping subsets called *folds*, then it trains and evaluates the model k times, picking a different fold for evaluation every time and using the other k-1 folds for training. The result is an array containing the k evaluation scores.

Scikit-Learn Demonstration

```
from sklearn.model_selection import cross_val_score

tree_rmses = -cross_val_score(tree_reg, housing, housing_labels,
                              scoring="neg_root_mean_squared_error", cv=10)
```

```
>>> pd.Series(tree_rmses).describe()
count       10.000000
mean     66868.027288
std       2060.966425
min      63649.536493
25%      65338.078316
50%      66801.953094
75%      68229.934454
max      70094.778246
dtype: float64
```

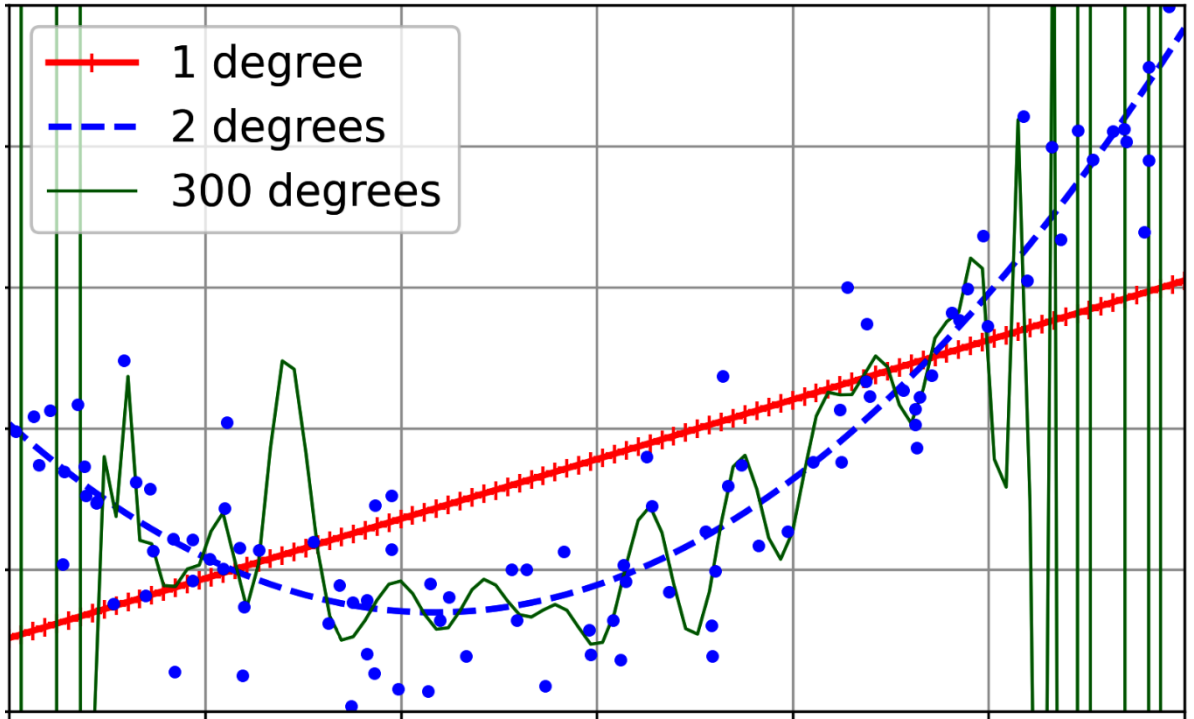
Note:

Sklearn's CV features expect a utility function (greater is better). So we're using *negative* of RMSE scoring function.

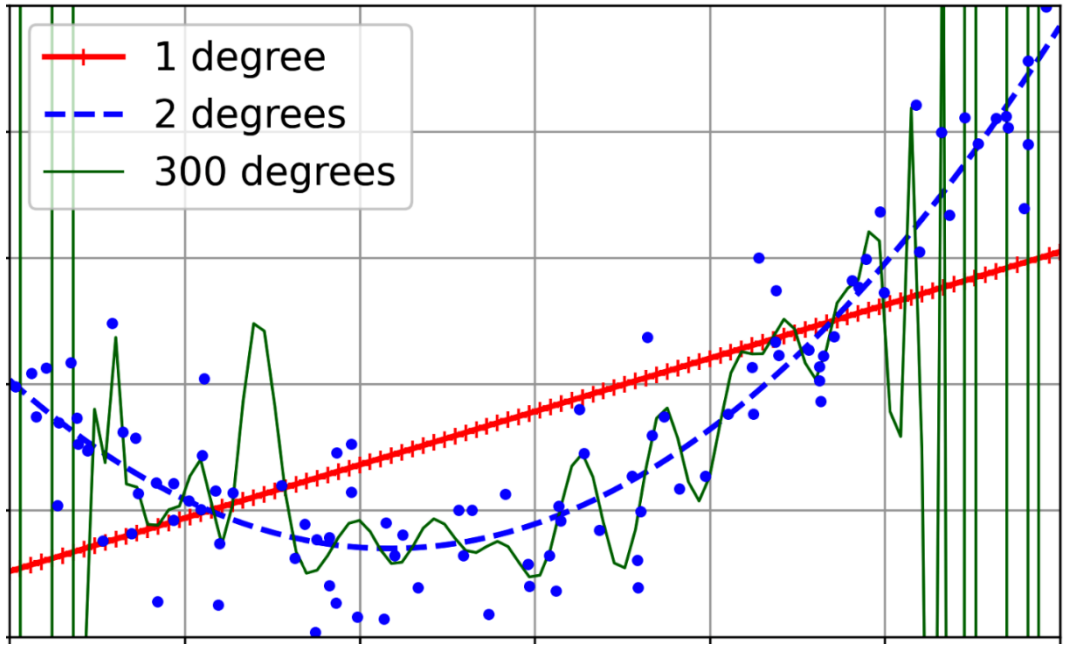
Overfitting & Underfitting

If you perform high-degree polynomial regression, you will likely fit the training data much better than with plain linear regression.

Notice how the 300-degree polynomial model wiggles around to get as close as possible to the training instances in the following figure:



Overfitting & Underfitting



But this high-degree polynomial regression model is severely overfitting the training data, while the linear model is underfitting it. The model that will generalize best in this case is the quadratic model, which makes sense because the data was generated using a quadratic model.

How to know if we have good fit

We can use **cross-validation** to get an estimate of a model's generalization performance. If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is **overfitting**. If it performs poorly on both, then it is **underfitting**.

This is one way to tell when a model is too simple or too complex.

Learning Curves

Another way to tell is to look at the ***learning curves***, which are plots of the model's training error and validation error on each training iteration.

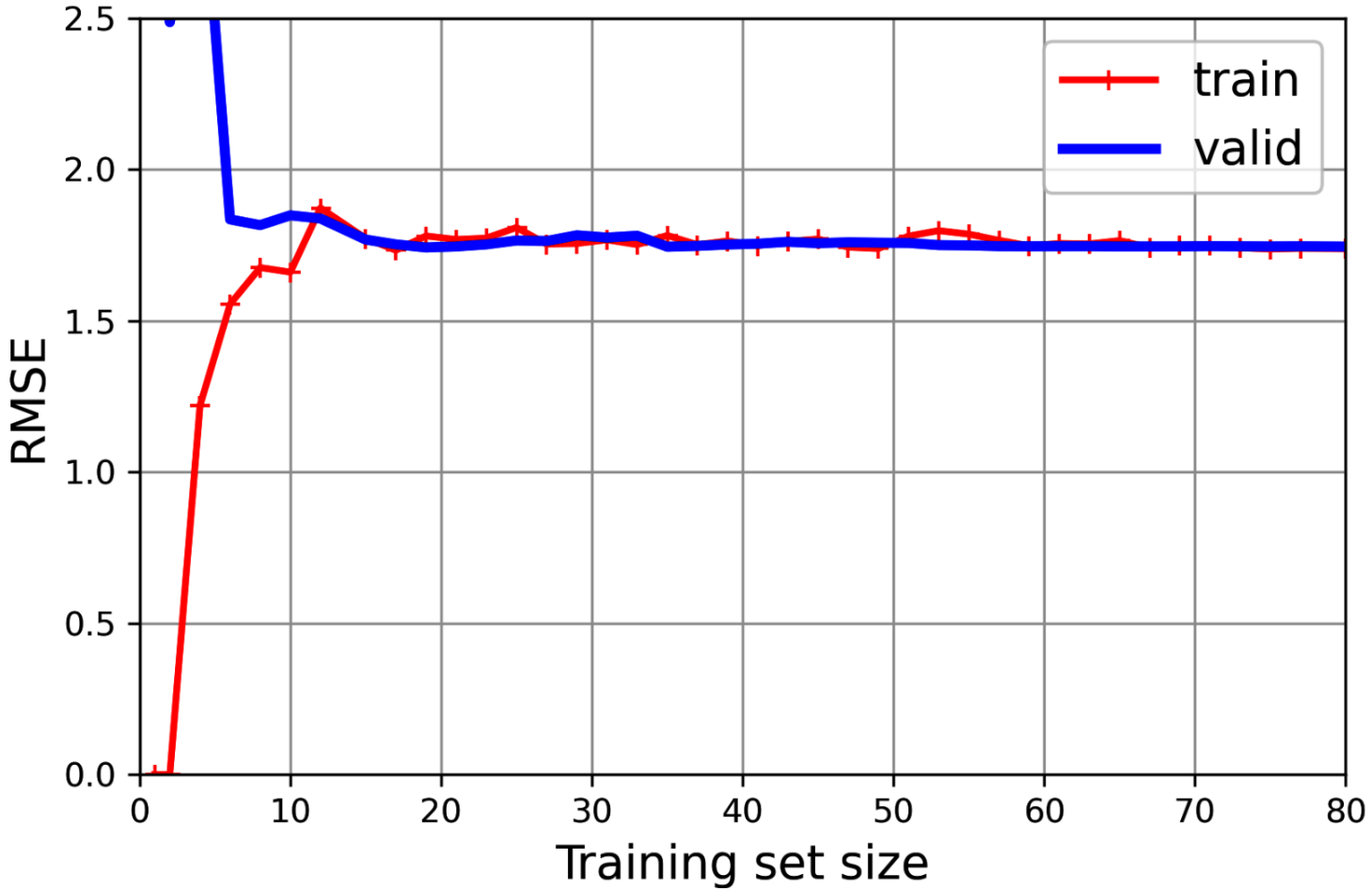
Scikit-Learn Example

```
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)

plt.plot(train_sizes, train_errors, "r-+", linewidth=2, label="train")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3, label="valid")
[...]  # beautify the figure: add labels, axis, grid, and legend
plt.show()
```

SKLearn’s implementation uses CV on different test sizes.

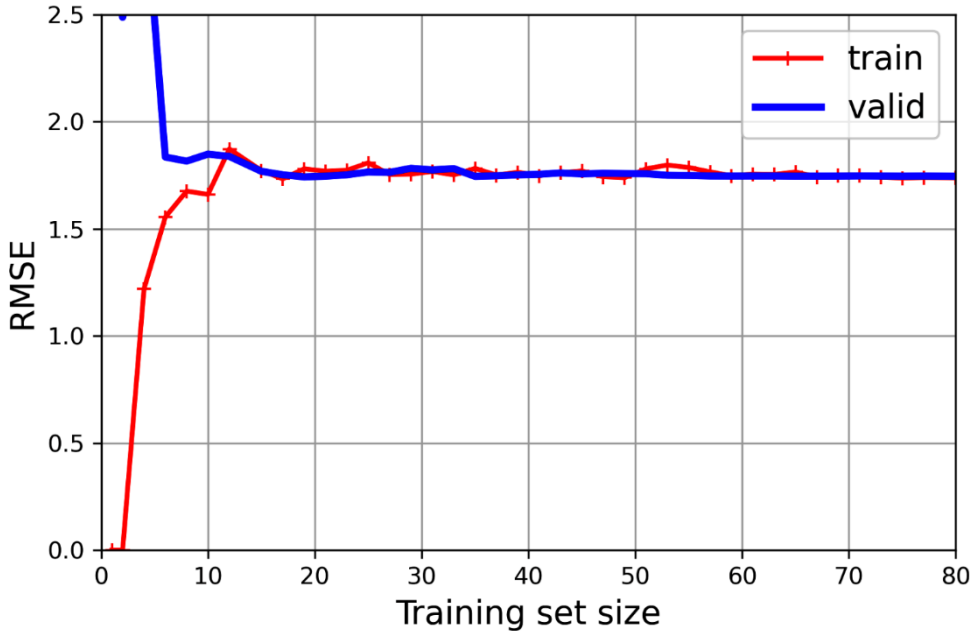
Scikit-Learn Example

Looking at the result, we could tell the model is underfitting.



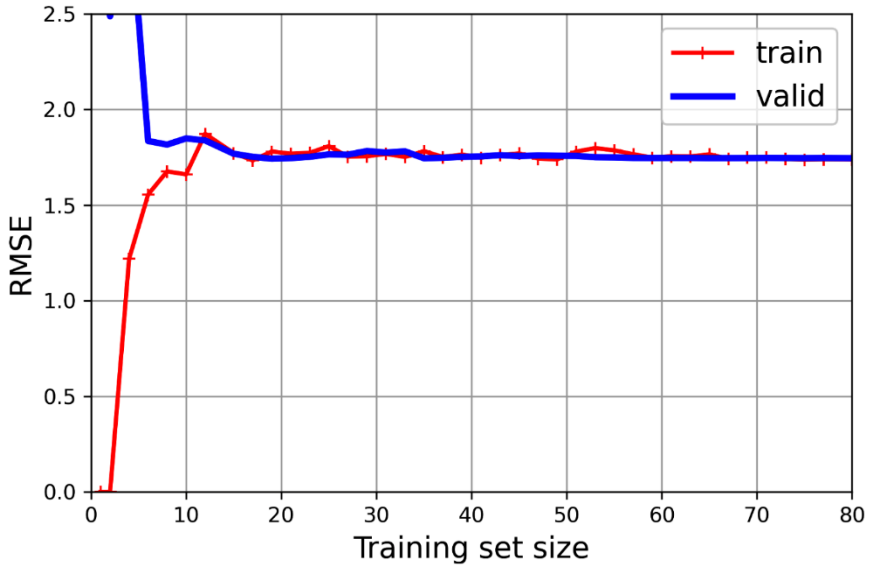
Underfitting Example

This model is underfitting. When there are just one or two instances in the training set, the model can fit them perfectly, which is why the curve starts at zero. But as new instances are added to the training set, it becomes impossible for the model to fit the training data.



Underfitting Example

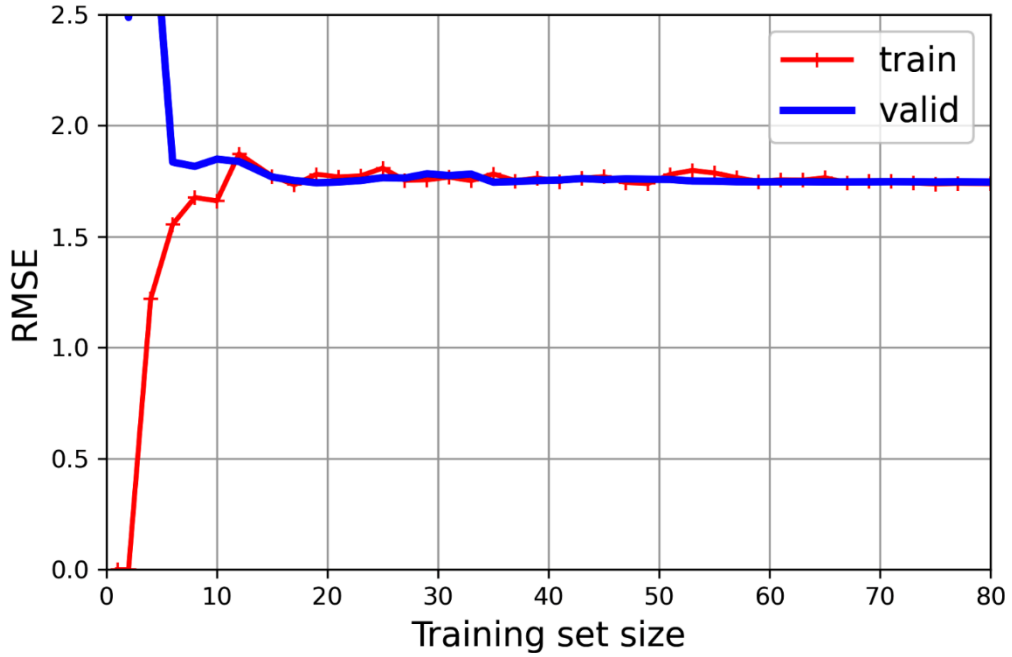
When the model is trained on very few training instances, it is incapable of generalizing properly, which is why the validation error is initially quite large. Then, as the model is shown more training examples, it learns, and thus the validation error slowly goes down, and very close to the other curve.



Underfitting Example

These learning curves are typical of a model that's underfitting. Both curves have reached a plateau; they are close and fairly high.

Tip: If your model is underfitting the training data, adding more training examples will not help. You need to use a better model or come up with better features.



Overfitting Example

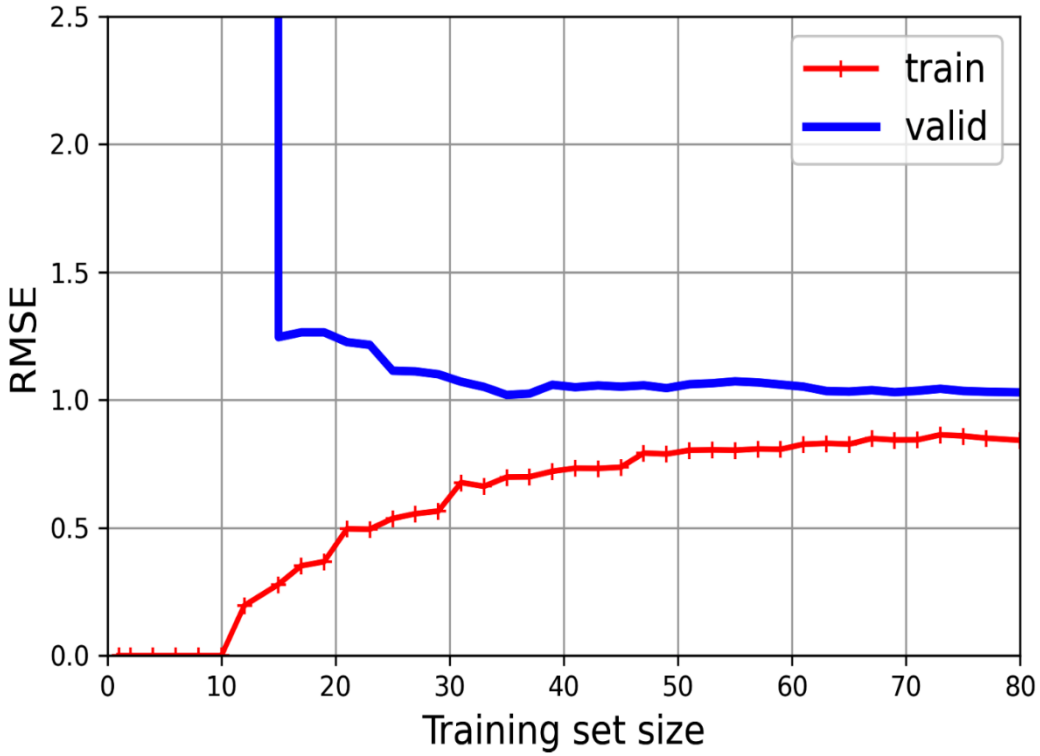
Now let's look at the learning curves of a 10th-degree polynomial model on the same data.

```
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")
[...]  # same as earlier
```

Overfitting Example

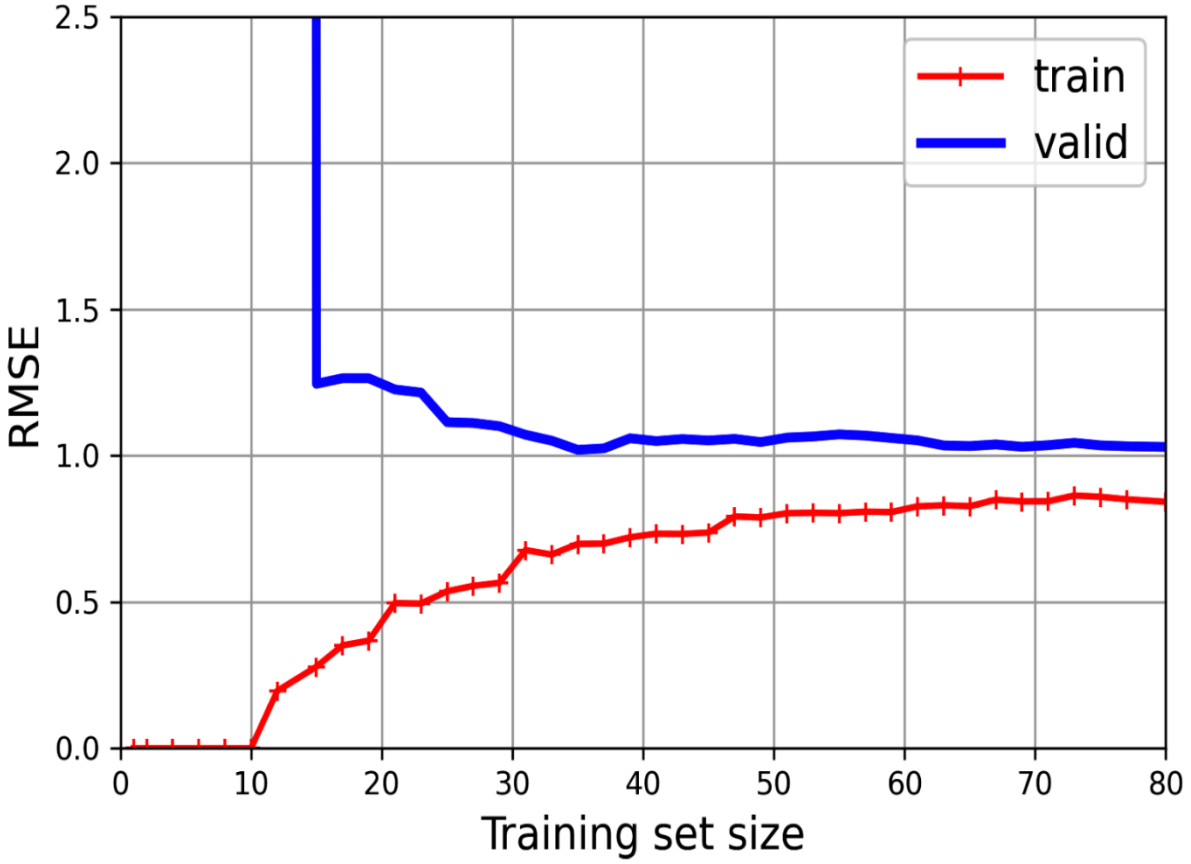
There are two very important differences here:

- The error on the training data is much lower than before.
- There is a **gap** between the curves. This means that the model performs significantly better on the training data than on the validation data, which is the hallmark of an **overfitting** model.



Overfitting Example

Tip: One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.



The Bias/Variance Trade-Off

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

- ***Bias:*** This part of the generalization error is 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:*** This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance and thus overfit the training data.

The Bias/Variance Trade-Off

- ***Irreducible error:*** This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases its bias and reduces its variance. This is why it is called a trade-off.

Regularized Linear Models

A simple way to regularize a polynomial model is to reduce the number of polynomial degrees.

For a linear model, regularization is typically achieved by constraining the weights of the model.

We will now look at **ridge regression**, **lasso regression** and **elastic net regression**, where regularization is achieved by simply augmenting the cost function during the training phase.

Ridge Regression

Ridge regression is a regularized version of linear regression: a regularization term equal to $\frac{\alpha}{m}\sum_{i=1}^{n}\theta_i^2$ is added to the MSE.

This forces the learning algorithm to not only fit the data but also keep the model weights *as small as possible*. Note that the regularization term should only be added to the cost function during training, and not during model evaluation.

Ridge Regression

Ridge regression cost function:

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

The hyperparameter α controls how much you want to regularize the model.

If $\alpha = 0$, then it's just linear regression. If α is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean.

Scikit-Learn 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])
```

Using a closed-form solution:

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=0.1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[1.55325833]])
```

Scikit-Learn Ridge Regression

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. Several other estimators (mostly linear) also have efficient CV variants, such as LassoCV and ElasticNetCV.

Lasso Regression

Least absolute shrinkage and selection operator regression (LASSO) is just like ridge regression, it adds a regularization term to the cost function, but it uses the l1 norm of the weight vector instead of the square of the l2 norm.

Lasso regression cost function:

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

Elastic Net Regression

Elastic net regression is a middle ground between ridge regression and lasso regression. The regularization term is a weighted sum of both ridge and lasso's regularization terms, and you can control the mix ratio *r*. When *r* = 0, elastic net is equivalent to ridge regression, and when *r* = 1, it is equivalent to lasso regression.

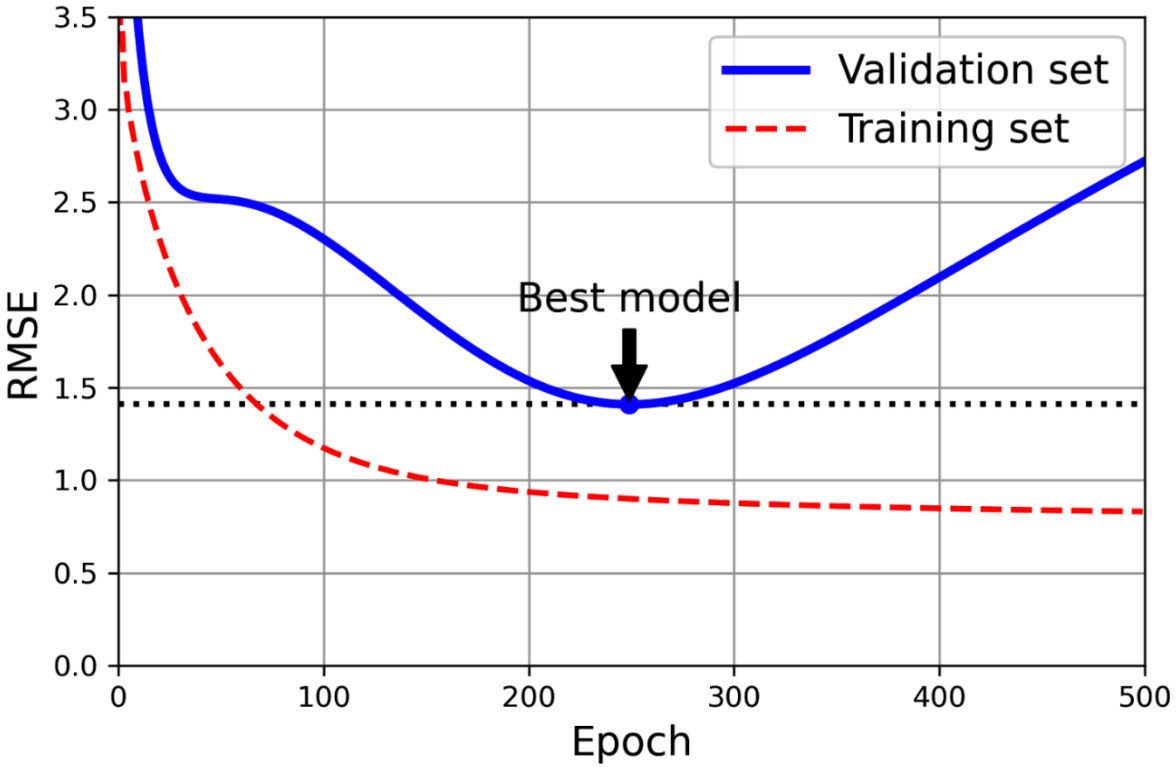
Elastic net cost function:

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

Early Stopping

A very different way to regularize iterative learning algorithms such as gradient descent is to stop training as soon as the validation error reaches a minimum.

As you could see with the high-degree polynomial regressor in the figure, after a while, the validation error stops decreasing and starts to go back up. This indicates that the model has started to overfit the training data.



Logistic Regression

Some regression algorithms can be used for classification (and vice versa) .

Logistic regression (also called *logit regression*) is a binary classifier commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?).

If the estimated probability is greater than a given threshold (typically 50%), then the model predicts that the instance belongs to that class (called the *positive class*, labeled “1”), and otherwise it predicts that it does not (i.e., it belongs to the *negative class*, labeled “0”).

Logistic Regression

Logistic regression model estimated probability equation (vectorized form):

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

Logistic function equation:

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

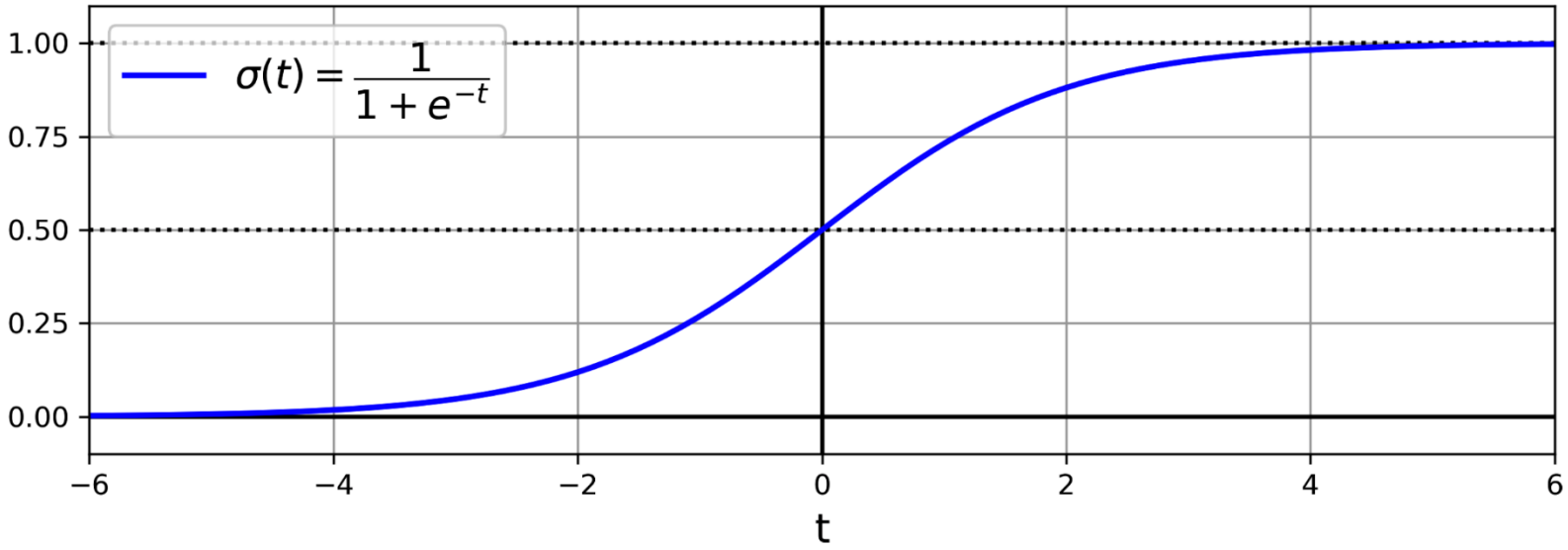
Just like a linear regression model, a logistic regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly, it outputs the *logistic* of this result.

Logistic Function

Logistic function equation:

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

The logistic—noted $\sigma(\cdot)$—is a *sigmoid function* (i.e., *S*-shaped) that outputs a number between 0 and 1.



The need for another cost fucntion

As discussed before, in order to train we need to use a proper cost function. One might ask why not just stick with RMSE? We need the cost function to encourage more correctly predicted labels. Remember how logistic regression prediction works:

$$y = \begin{cases} 1 \text{ if } \hat{p} > 0.5 \\ 0 \text{ if } \hat{p} < 0.5 \end{cases}$$

Also we need the $\hat{p}$ values to be close to the two ends of 0 and 1.

The problem with RMSE is it puts penalty on estimated vector distance and not necessarily on the correctness of labels. So we prepose another cost function:

How to train a logistic regression model

Cost function of a single training instance:

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

The objective of training is to set the parameter vector **θ** so that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$).

Log Loss

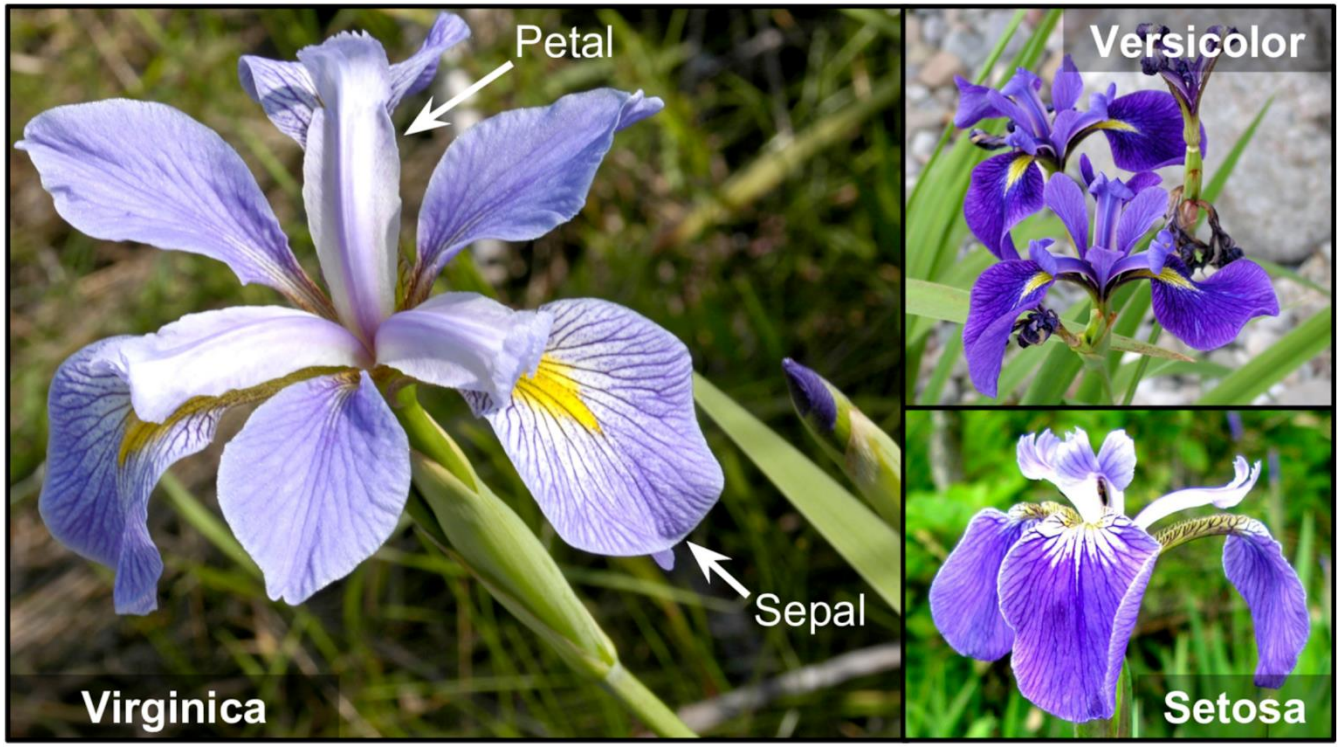
The cost function over the whole training set is the average cost over all training instances. It can be written in a single expression called the *log loss*, shown in

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

There is no known closed-form equation to compute the value of **θ** that minimizes this cost function but since cost function is convex, gradient descent is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough).

Showcase

We can use the iris dataset to illustrate logistic regression. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: Iris setosa, Iris versicolor, and Iris virginica (see Figure).



Load Data

Let’s try to build a classifier to detect the *Iris virginica* type based only on the petal width feature. The first step is to load the data and take a quick peek:

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris(as_frame=True)
>>> list(iris)
['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
 'filename', 'data_module']
>>> iris.data.head(3)
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
>>> iris.target.head(3)  # note that the instances are not shuffled
0    0
1    0
2    0
Name: target, dtype: int64
>>> iris.target_names
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

Split and Fit

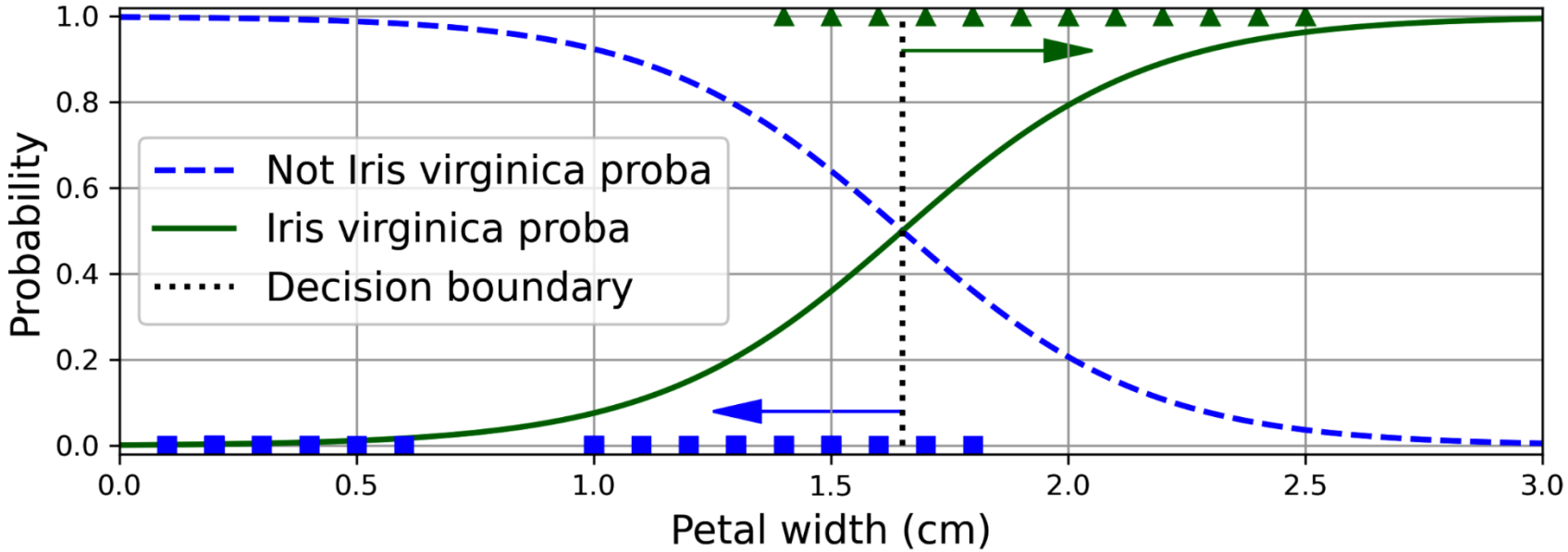
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X = iris.data[["petal width (cm)"]].values
y = iris.target_names[iris.target] == 'virginica'
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)
```

Estimate Probabilities

Let’s look at the model’s estimated probabilities for flowers with petal widths varying from 0 cm to 3 cm.

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)  # reshape to get a column vector
y_proba = log_reg.predict_proba(X_new)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0, 0]

plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2,
         label="Not Iris virginica proba")
plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris virginica proba")
plt.plot([decision_boundary, decision_boundary], [0, 1], "k:", linewidth=2,
         label="Decision boundary")
[...] # beautify the figure: add grid, labels, axis, legend, arrows, and samples
plt.show()
```

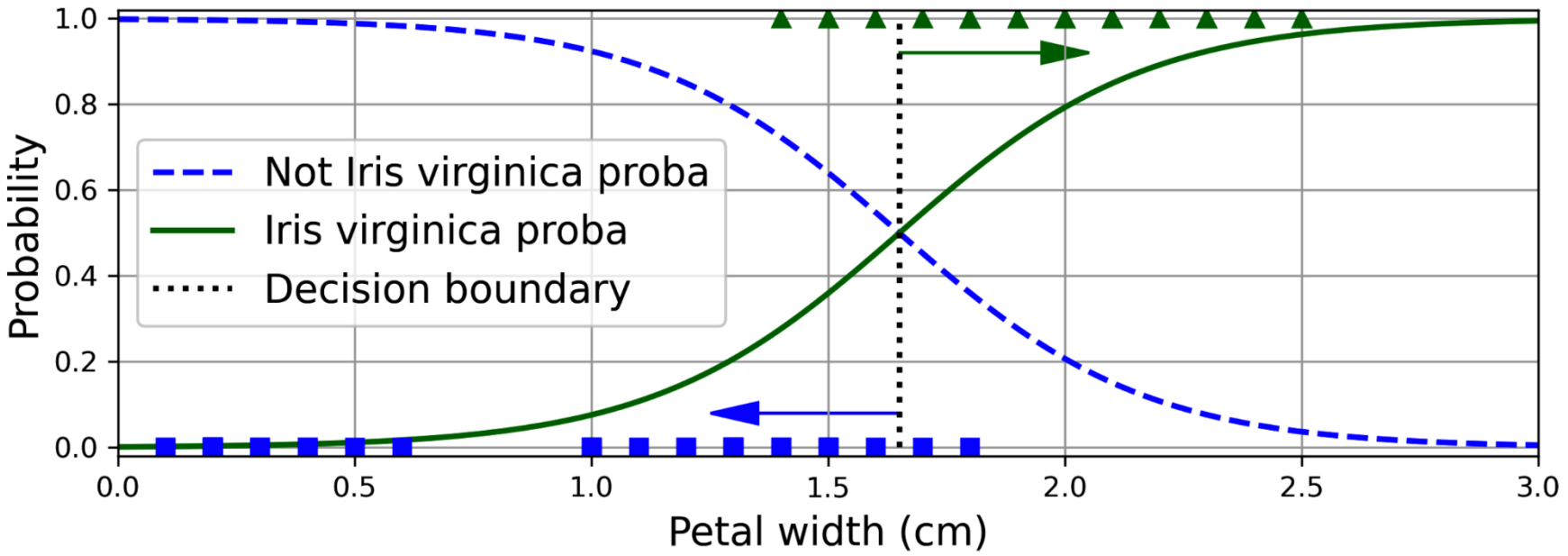
Decision Boundary



Notice that there is a bit of overlap between the petal width of *Iris virginica* flowers (represented as triangles) and others.

Above about 2 cm and below 1 cm, the classifier is highly confident with the result.

Decision Boundary

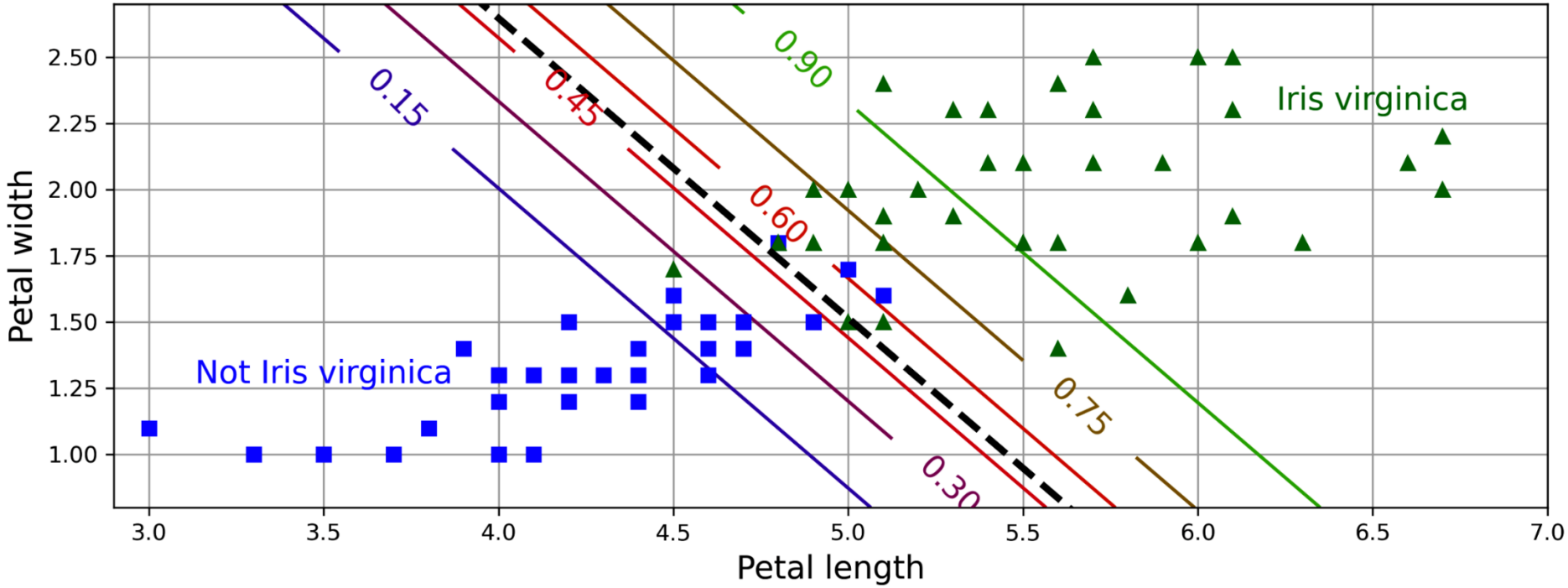


In between the extremes, the classifier is unsure. It will use the *decision boundary* where both probabilities are equal: if the petal width is greater than 1.6 cm the classifier will predict that the flower is an *Iris virginica*, and otherwise it is not (even if it is not very confident).

Show Case: Linear Descision Boundary

In this figure we’re desplaying two features: petal width and length.

The dashed line represents the points where the model estimates a 50% probability: this is the model’s decision boundary. Each parallel line represents a specific probability.



Softmax Regression

The logistic regression model can be generalized to support multiple classes, either by combining multiple binary classifiers, or directly, as it is done in **softmax regression** (also called **multinomial logistic regression**).

The idea is simple: given an instance **x**, the model first computes a score $s_k(\boldsymbol{x})$ for each class *k*, then estimates the probability of each by applying the *softmax function to the scores.*

Equation to compute softmax score for class k:

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

Note that each class has its dedicated parameter vector $\theta^{(\mathbf{k})}$.

Softmax Function

Softmax function:

$$\widehat{p_k} = \sigma\left(\mathbf{s}(\mathbf{x})\right)_k = \frac{exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} exp\left(s_j(\mathbf{x})\right)}$$

In this equation:

- K is the number of classes.
- $\mathbf{s}(\mathbf{x})$ is a vector containing the scores of each class for the instance $\mathbf{x}$.
- $\sigma\left(\mathbf{s}(\mathbf{x})\right)_k$ is the estimated probability that the instance $\mathbf{x}$ belongs to class k, given the scores of each class for that instance.

Softmax Regression Classifier Prediction

Softmax regression classifier prediction equation:

$$\hat{y} = \arg\max_k \sigma\big(\mathbf{s}(\mathbf{x})\big)_k = \arg\max_k s_k(\mathbf{x}) = \arg\max_k \big(\theta^{(\mathbf{k})}\big)^T \mathbf{x}$$

It simply returns the class with the most probability score.

Simple Example

Imagine $\boldsymbol{\theta}$ is a $k \times m$ parameter matrix, and x is a data instance with m features,

and the probability table of classes looks like this: (number of classes $k = 4$)

k	$s_k(\mathbf{x}) = \left(\theta^{(\mathbf{k})}\right)^T \mathbf{x}$	$e^{s_k(\mathbf{x})}$	$e^{s_k(\mathbf{x})} / \Sigma_j e^{s_j(\mathbf{x})}$
1	-1	0.368	0.002
2	0	1	0.006
3	3	20.09	0.118
4	5	148.41	0.874

The class with highest probability is class 4.

Cross Entropy Cost Function

The objective is to have a model that estimates a **high probability** for the target class (and consequently a low probability for the other classes). Minimizing the *cross entropy* cost function, should lead to this objective because it penalizes the model when it estimates a low probability for a target class. **Cross entropy** is frequently used to measure how well a set of estimated class **probabilities** matches the target classes.

Cross Entropy Cost Function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(\, p_k^{(i)})$$

In this equation, $y_k^{(i)}$ is the target probability that the i^{th} instance belongs to class *k*. In general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not.

When there are just two classes (*K* = 2), this cost function is equivalent to the log loss function.

Now What

Given the cost function, now you can compute the gradient vector for every class, then use gradient descent (or any other optimization algorithm) to find the parameter matrix **Θ** that minimizes the cost function.

Showcase

Let’s use softmax regression to classify the iris plants into all three classes. SKLearn’s LogisticRegression classifier uses softmax regression automatically when you train it on more than two classes.

It also applies l2 regularization by default, which you can control using the hyperparameter C (which is the inverse of alpha, used for the regularization of logistic regression model).

Showcase

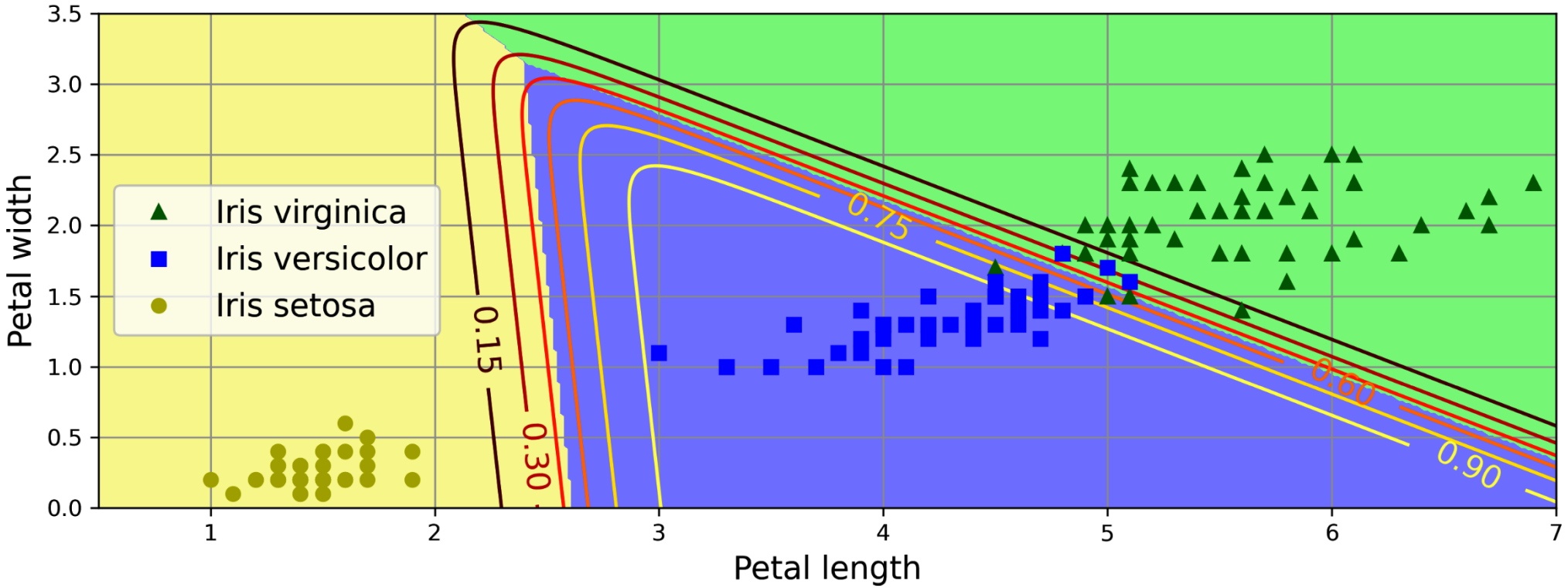
```
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = iris["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

softmax_reg = LogisticRegression(C=30, random_state=42)
softmax_reg.fit(X_train, y_train)
```

Let’s predict the class and probability of an iris with petals 5 cm long and 2 cm wide:

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]]).round(2)
array([[0.  , 0.04, 0.96]])
```

Decision Boundries

Notice that the model can predict a class that has an estimated probability below 50%. For example, at the point where all decision boundaries meet, all classes have an equal estimated probability of 33%.



Course Review

- **Linear Regression**: A model that predicts by computing a weighted sum of features and a bias.
- **Gradient Descent**: An iterative generic optimization algorithm.
- **Batch vs Stochastic vs Mini-batch GD**: Using whole, one, or some of data to compute the gradients.
- **Polynomial Regression**: Adds power of features as new features.

Course Review

- **Cross-Validation**: Split data into folds, select one fold for training and the rest for evaluation. Repeat for all folds.
- **Overfit/Underfit**: When there is (too much)/(no) generalization.
- Regularization: Constraining the model (e.g. limiting number of features)
- **Logistic/(Softmax) Regression**: Classifier that estimates the probability of an instance belonging to (a)/(each) class.

Thank you for your attention