



Machine Learning

Reference:

Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems
by
Geron Aurelien

Dr.B.Santhosh Kumar,
Associate Professor,
CSE Department,
GPREC, Kurnool

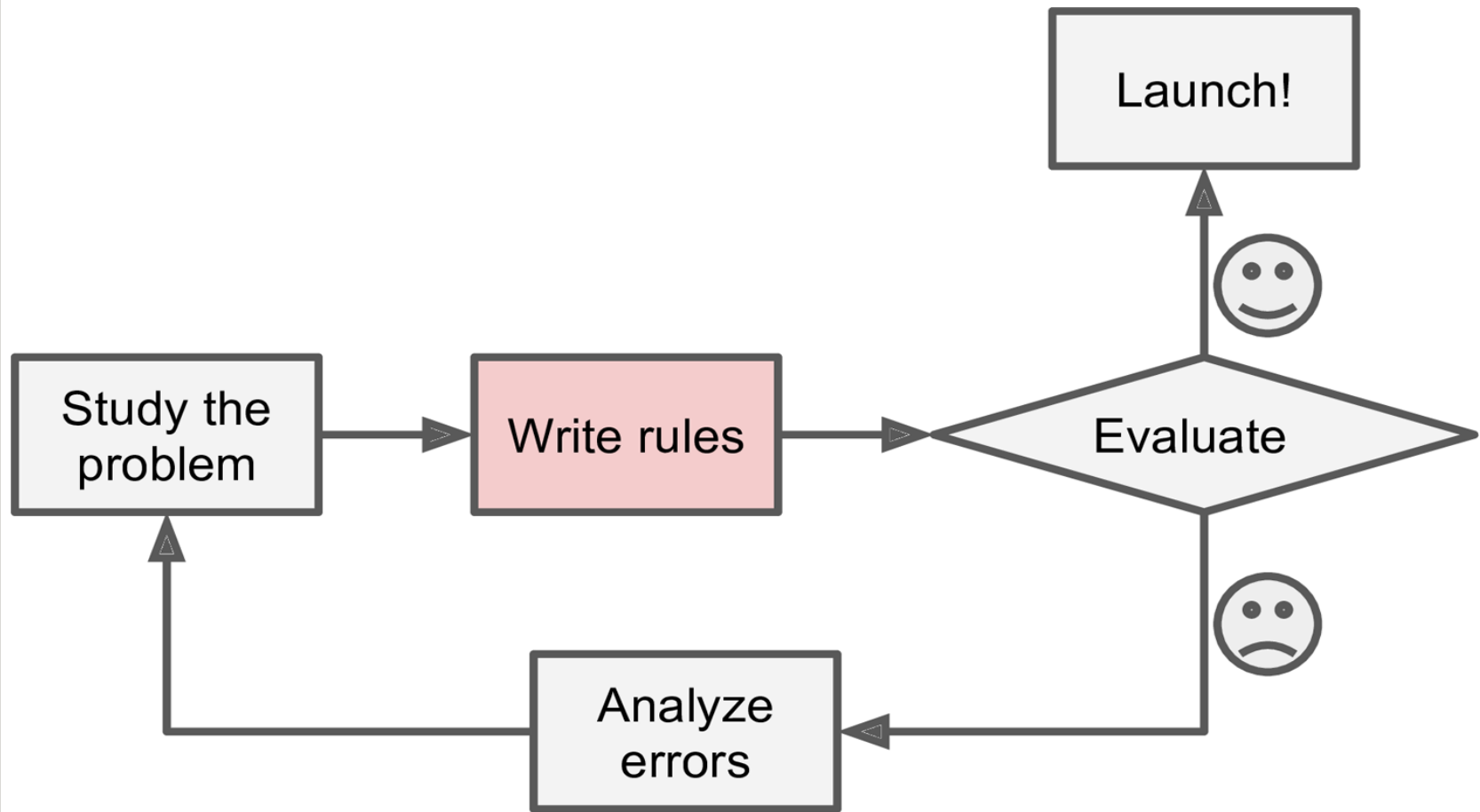
Fundamentals of Machine Learning

- Machine Learning is the science of programming computers so they can *learn from data*.
- Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed.
- A computer program is said to learn from experience E with respect to some task T and some performance measure P , if its performance on T , as measured by P , improves with experience E .

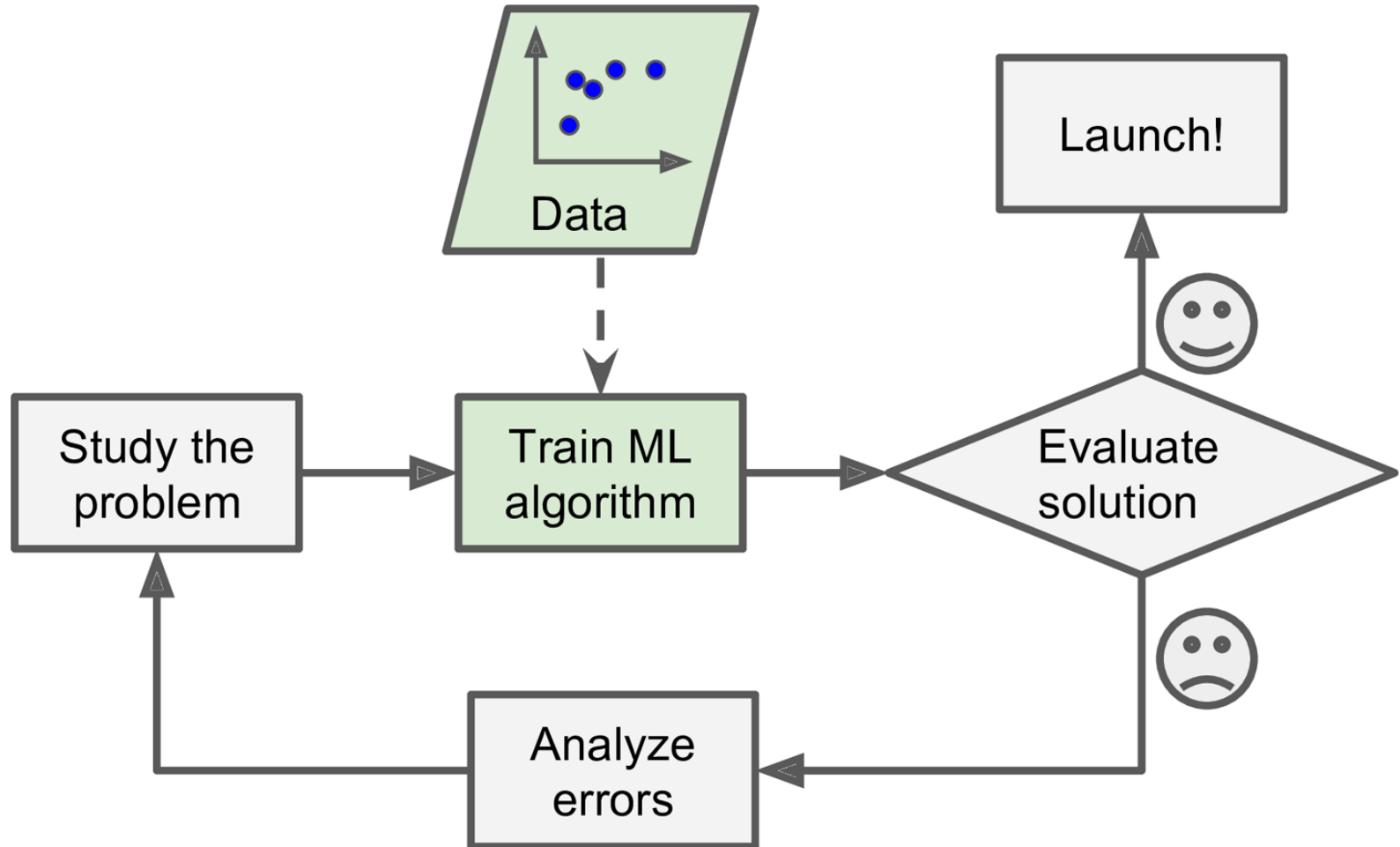
Example

- Spam filter is a Machine Learning program that can learn to flag spam given examples of spam emails (e.g., flagged by users) and examples of regular (nospam, also called “ham”) emails.
- The examples that the system uses to learn are called the *training set*. *Each training example is called a training instance (or sample)*.
- In this case, the task T is to flag spam for new emails, the experience E is the *training data*, and the *performance measure P needs to be defined*; for example, you can use the ratio of correctly classified emails.
- This particular performance measure is called *accuracy and it is often used in classification tasks*.
- If you just download a copy of Wikipedia, your computer has a lot more data, but it is not suddenly better at any task. Thus, it is not Machine Learning.

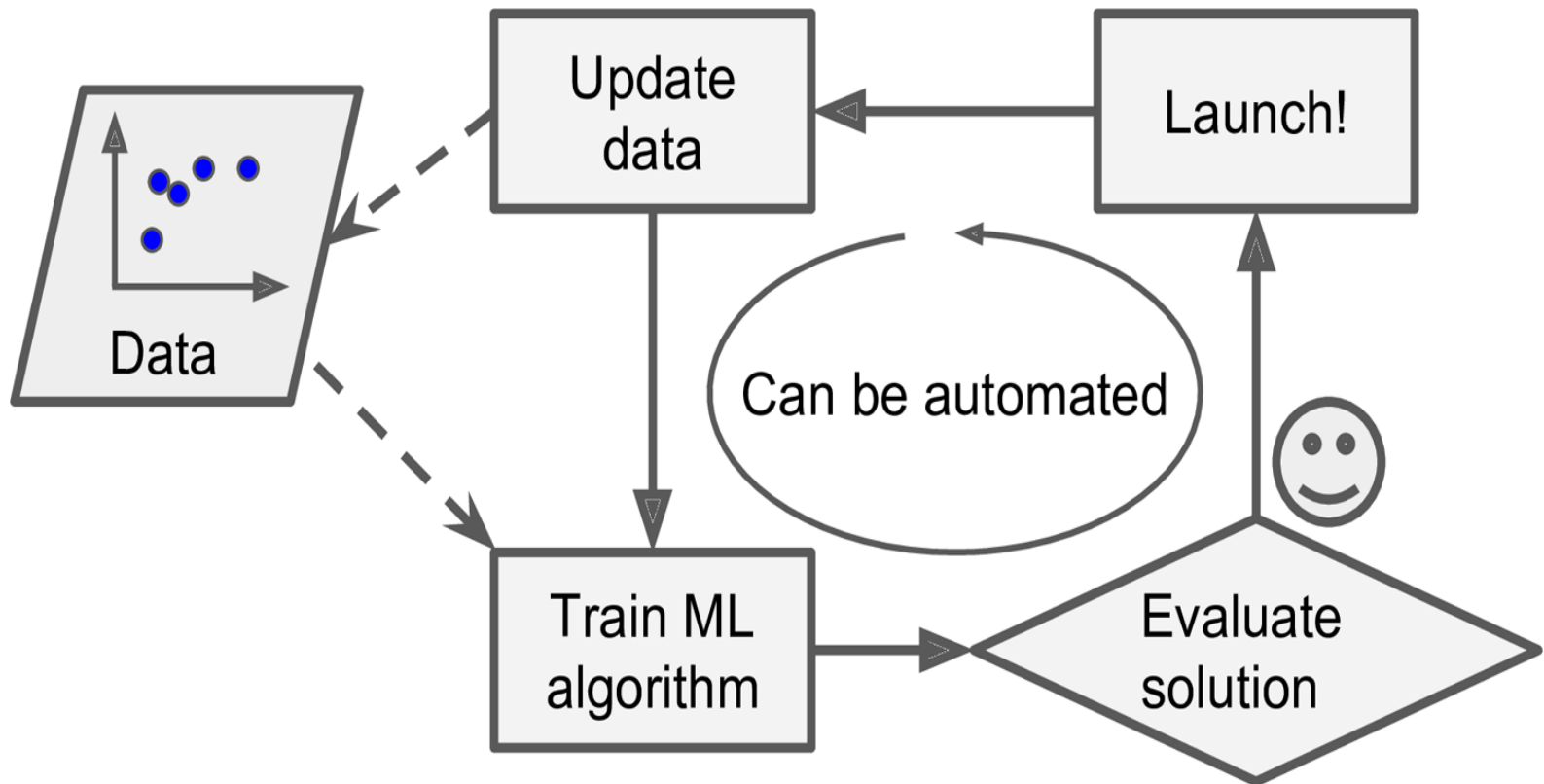
The traditional *approach*



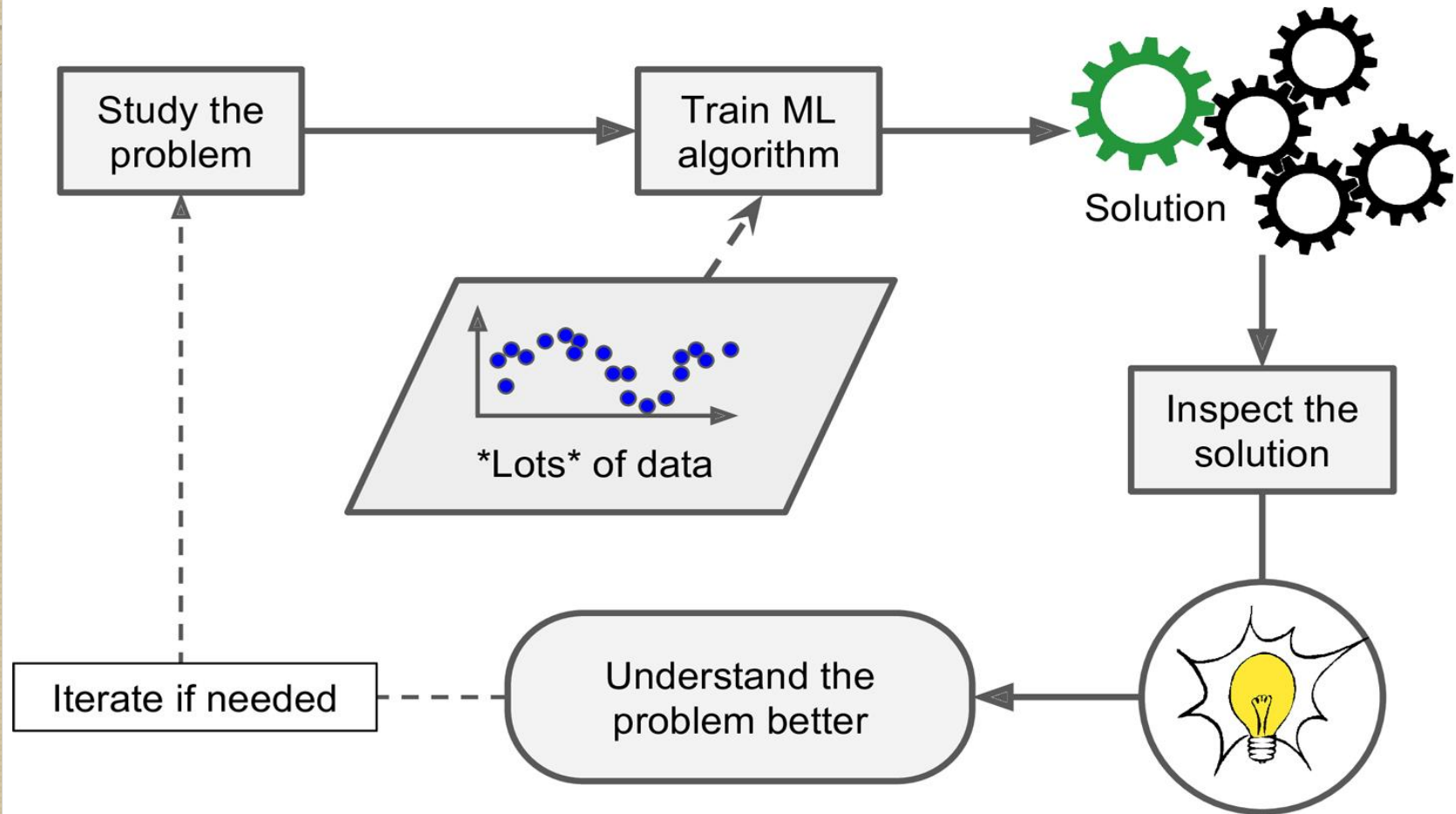
Use of Machine Learning



Automatic Adaptation



Machine Learning helps Humans Learn

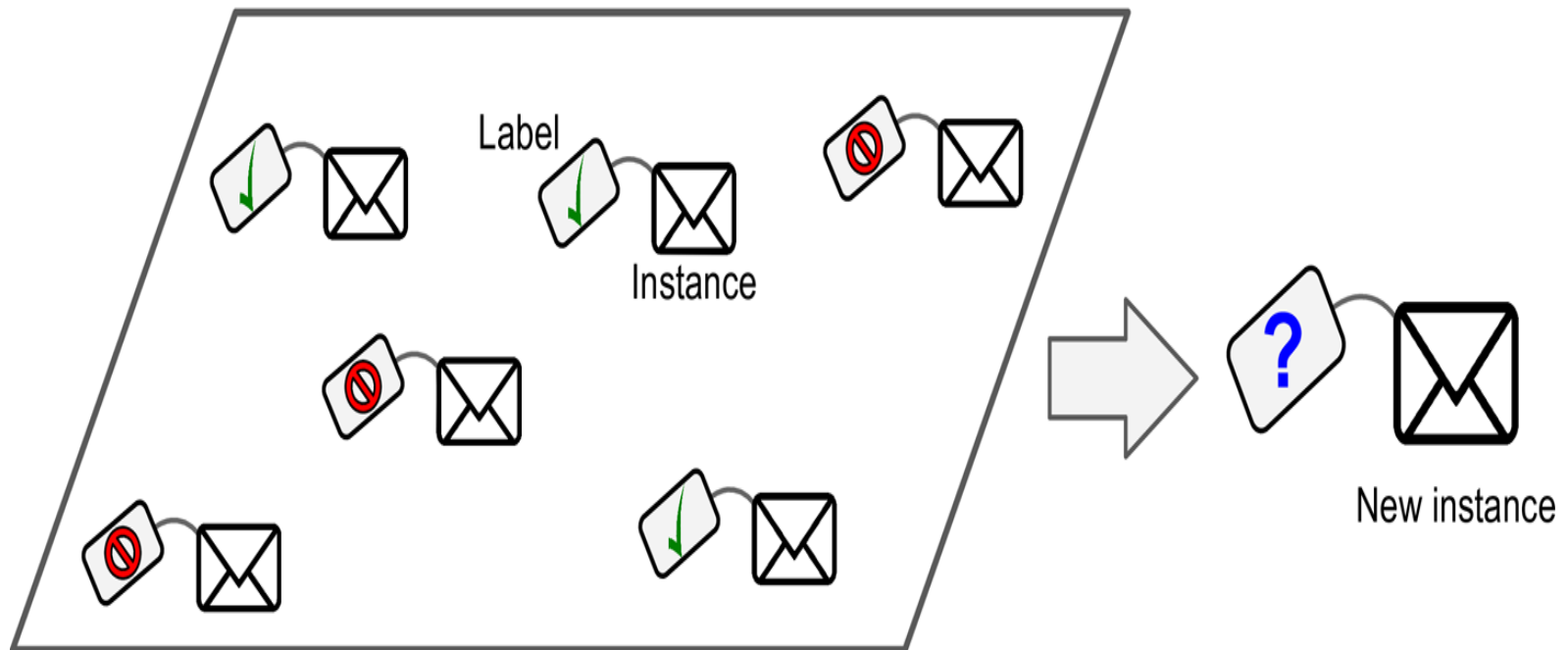


Types of Machine Learning Systems

- Machine Learning systems can be classified according to the amount and type of supervision they get during training(supervised, unsupervised, semi supervised, and Reinforcement Learning)
- Whether or not they can learn incrementally on the fly (online versus batch learning)
- Whether they work by simply comparing new data points to known data points, or instead detect patterns in the training data and build a predictive model, much like scientists do (instance-based versus model-based learning)

Supervised learning

Training set

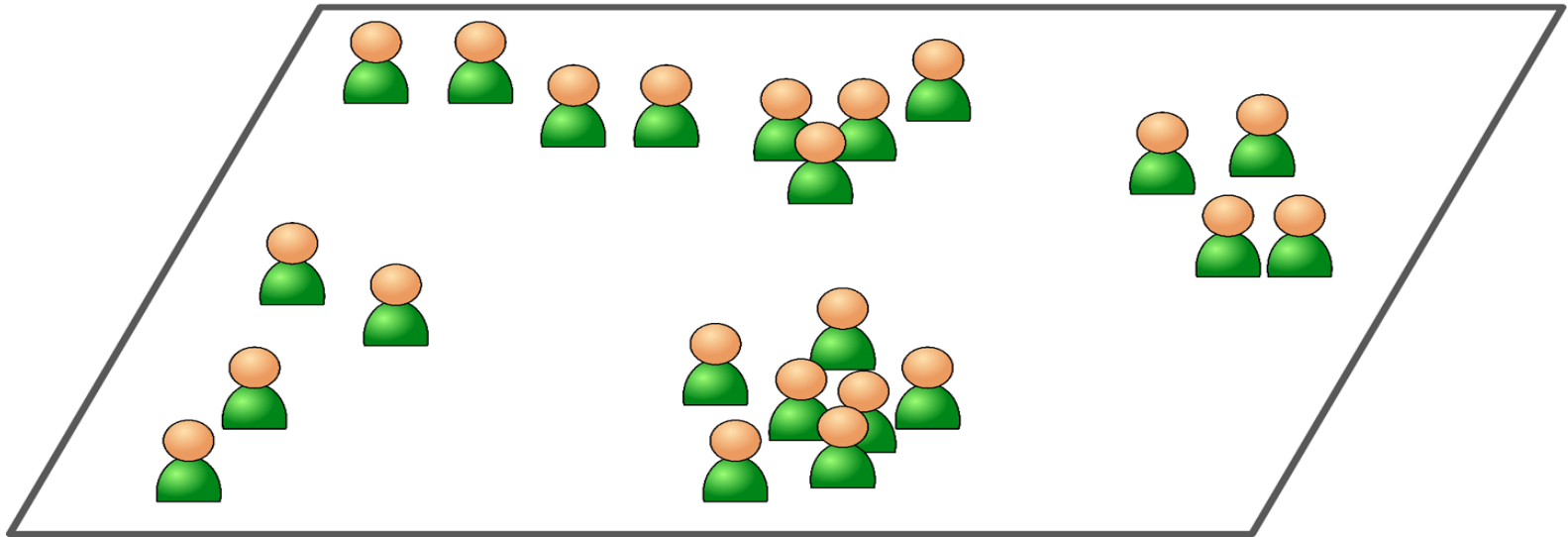


Examples: k-Nearest Neighbors, Linear Regression, Logistic Regression, Support Vector Machines (SVMs), Decision Trees and Random Forests, Neural networks2

Unsupervised learning

Data is unlabeled

Training set

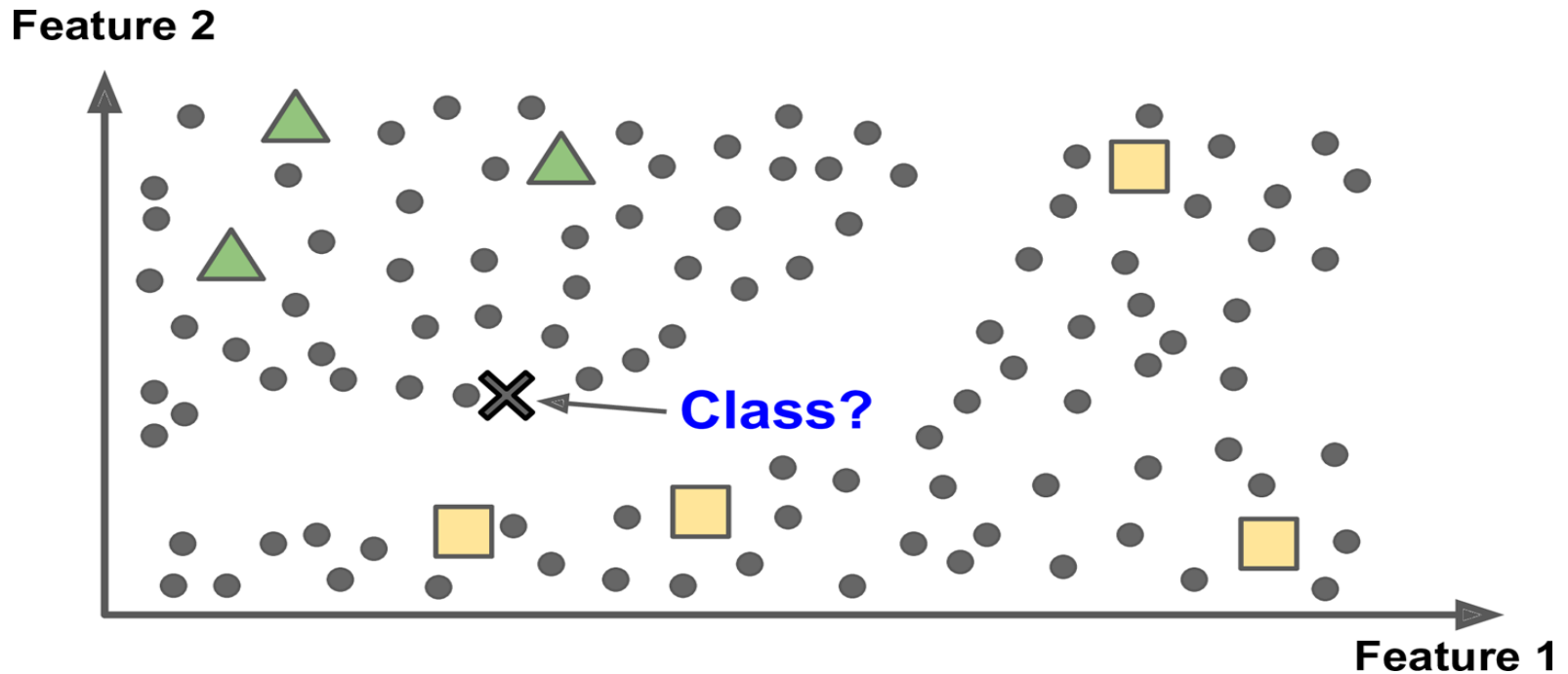


Examples: Clustering -- k-Means, Hierarchical Cluster Analysis (HCA), Visualization and dimensionality reduction -- Principal Component Analysis (PCA), Anomaly detection, Association rule learning -- Apriori

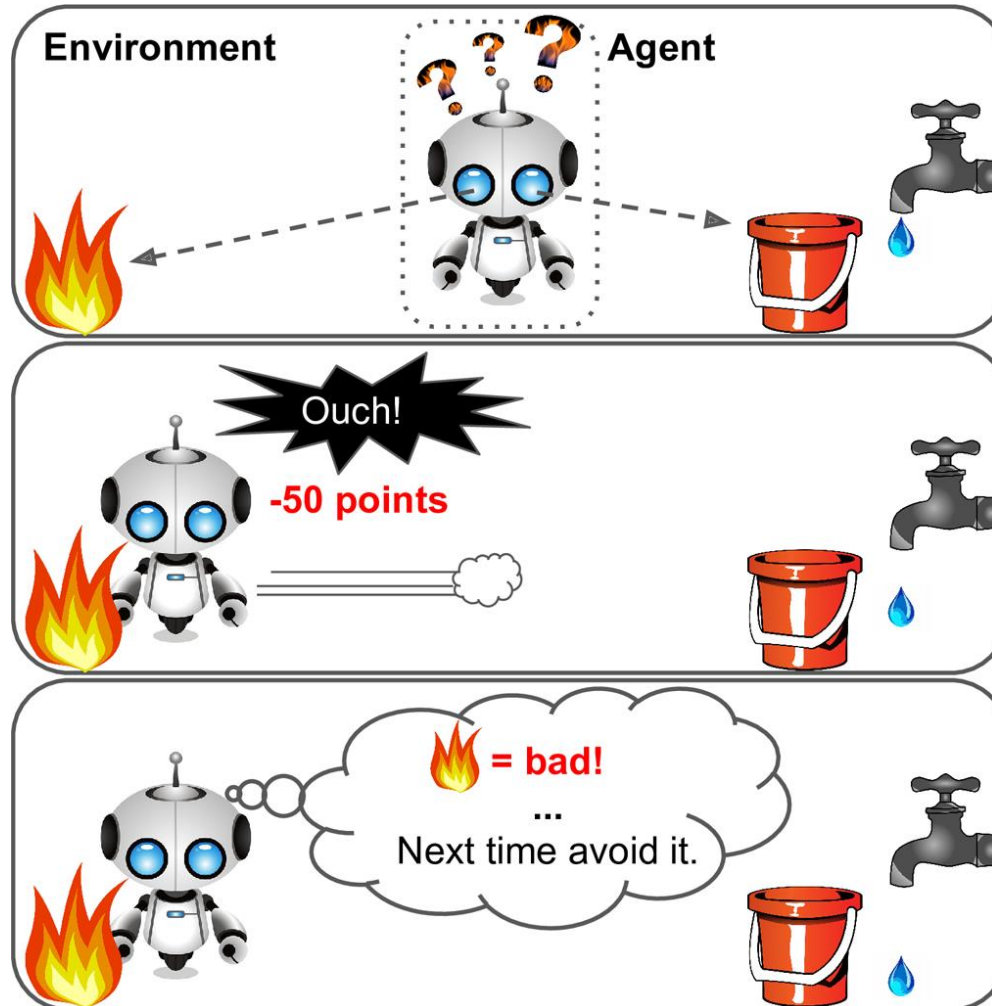
- A related task is *dimensionality reduction*, in which the goal is to simplify the data without losing too much information.
- One way to do this is to merge several correlated features into one. For example, a car's mileage may be very correlated with its age.
- So the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear. This is called *feature extraction*.

Semi supervised learning

Partially labeled training data i.e. usually a lot of unlabeled data and a little bit of labeled data.



Reinforcement Learning

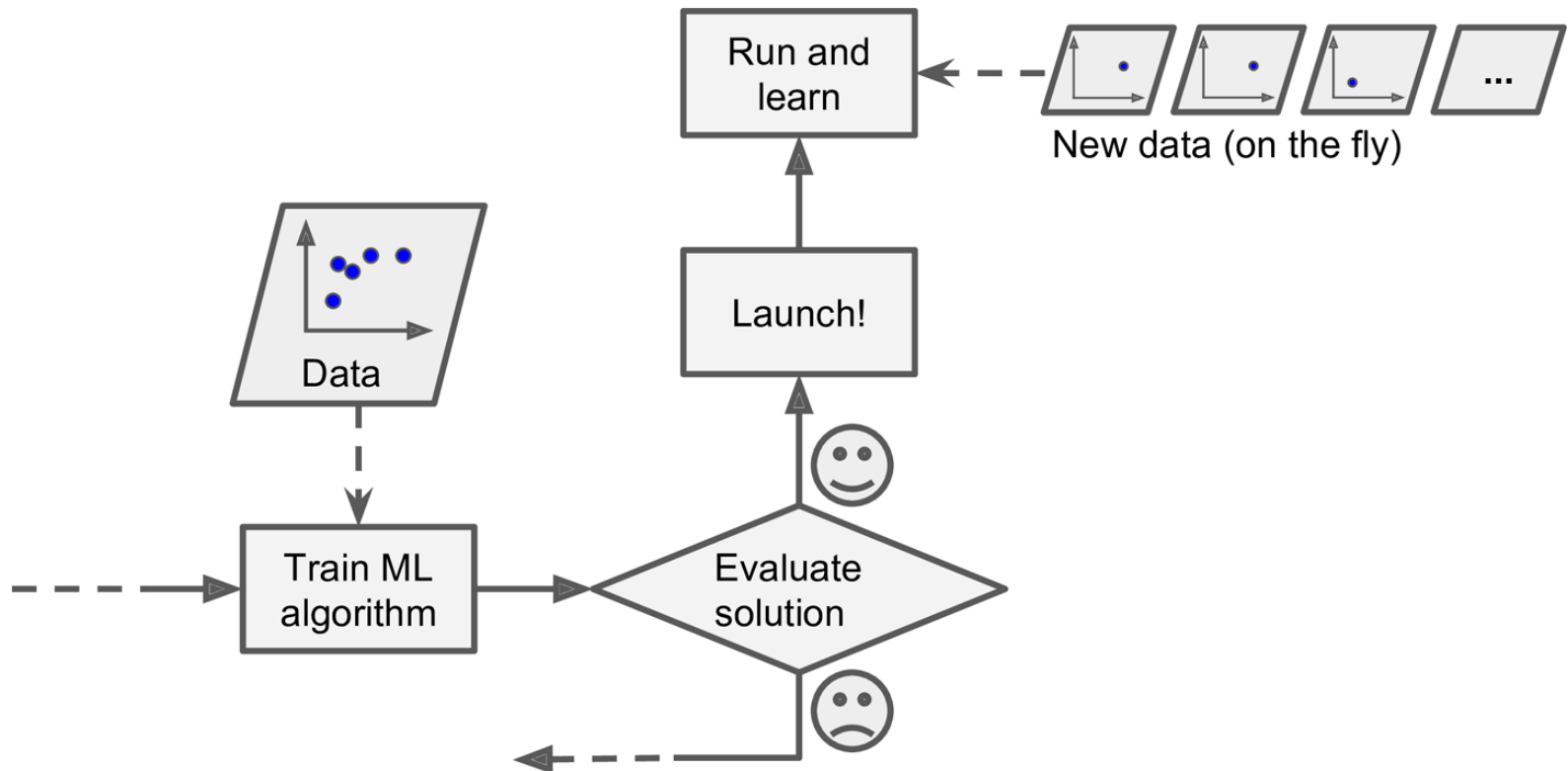


- 1 Observe
- 2 Select action using policy
- 3 Action!
- 4 Get reward or penalty
- 5 Update policy (learning step)
- 6 Iterate until an optimal policy is found

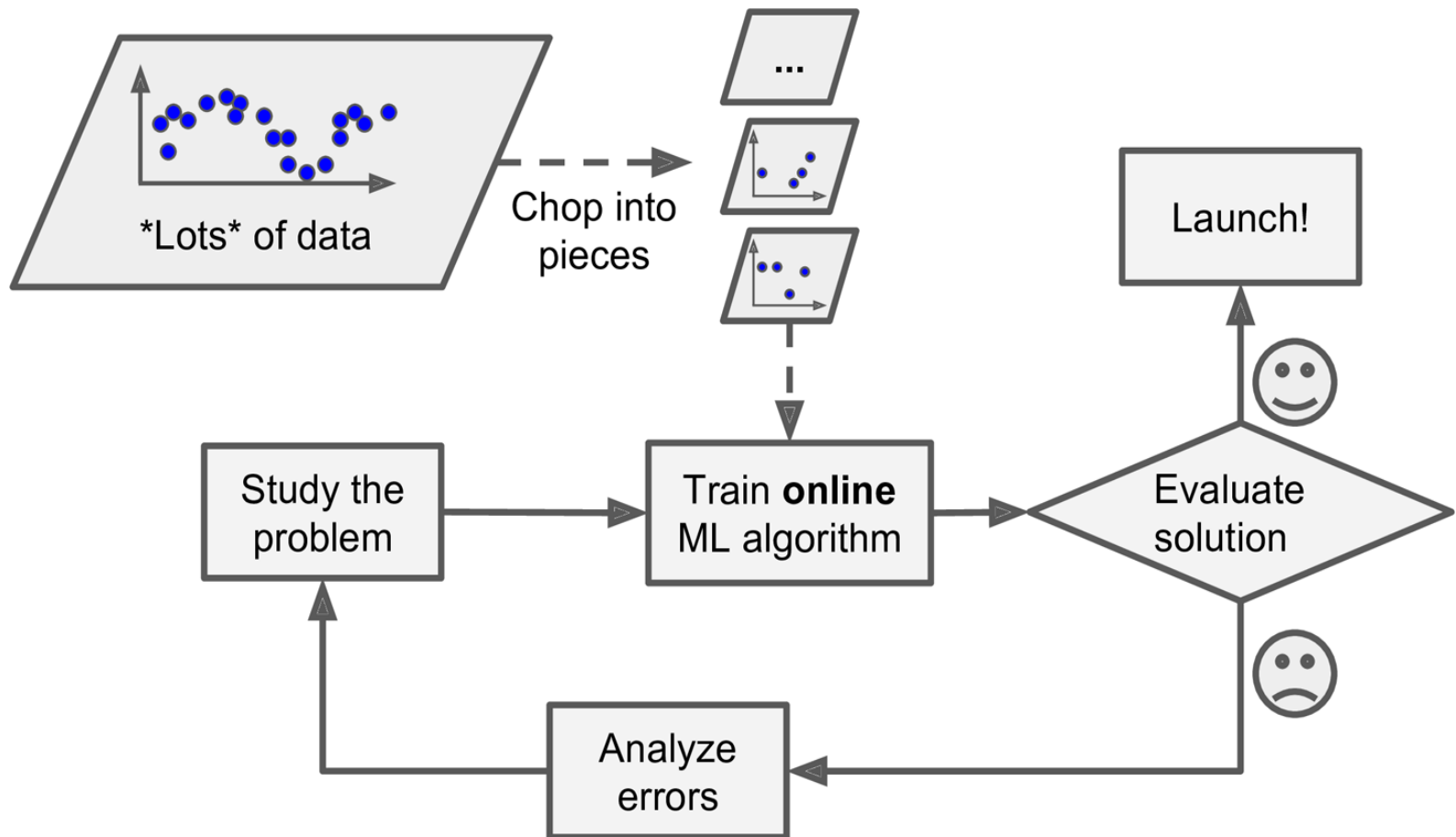
Batch Learning

- In *batch learning*, the system is incapable of learning incrementally: it must be trained using all the available data
- This will generally take a lot of time and computing resources, so it is typically done offline.
- First the system is trained, and then it is launched into production and runs without learning anymore; it just applies what it has learned. This is called *offline learning*.
- If you want a batch learning system to know about new data you need to train a new version of the system from scratch on the full dataset (not just the new data, but also the old data), then stop the old system and replace it with the new one.

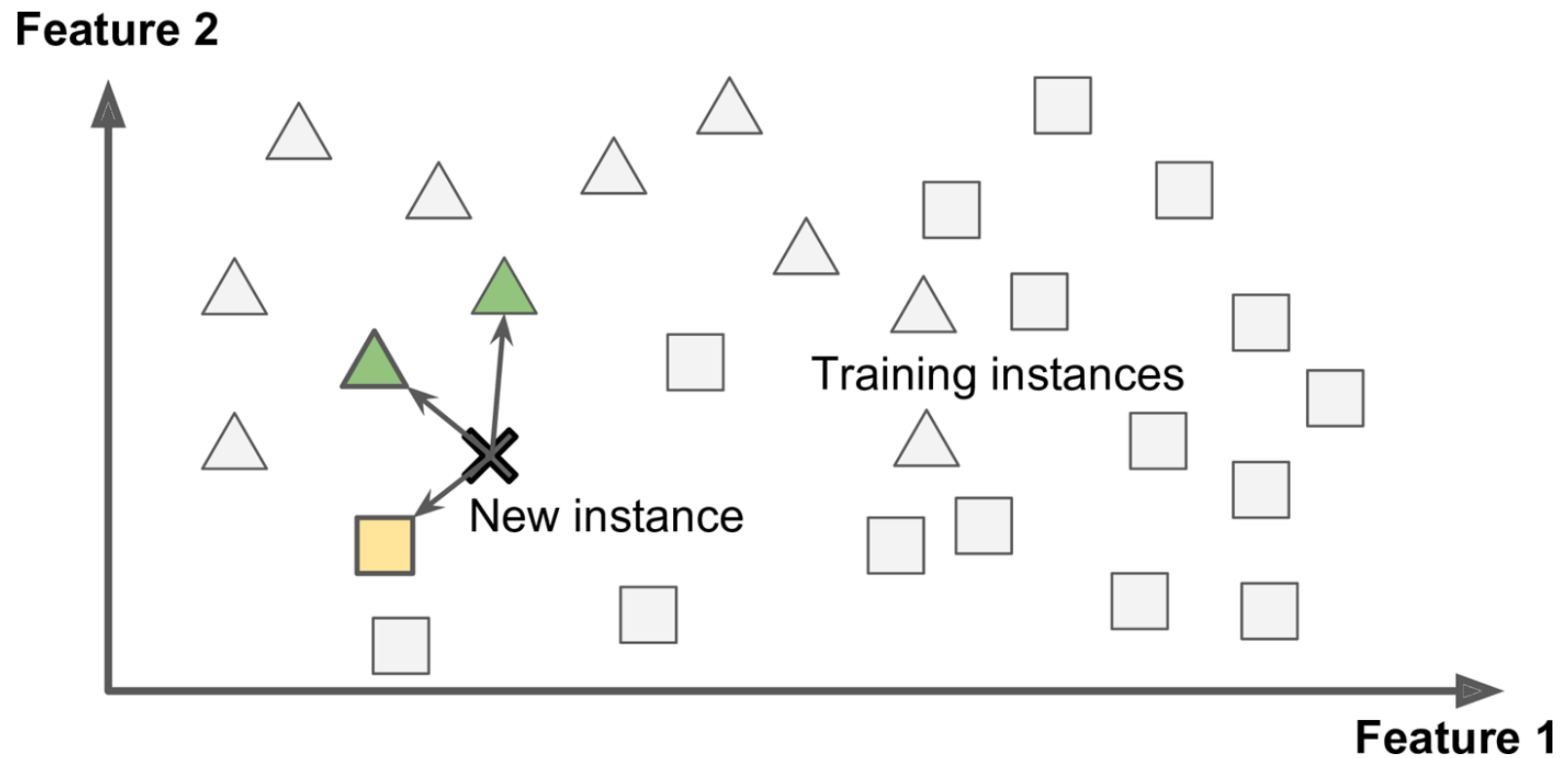
online learning



Online Learning with lots of data



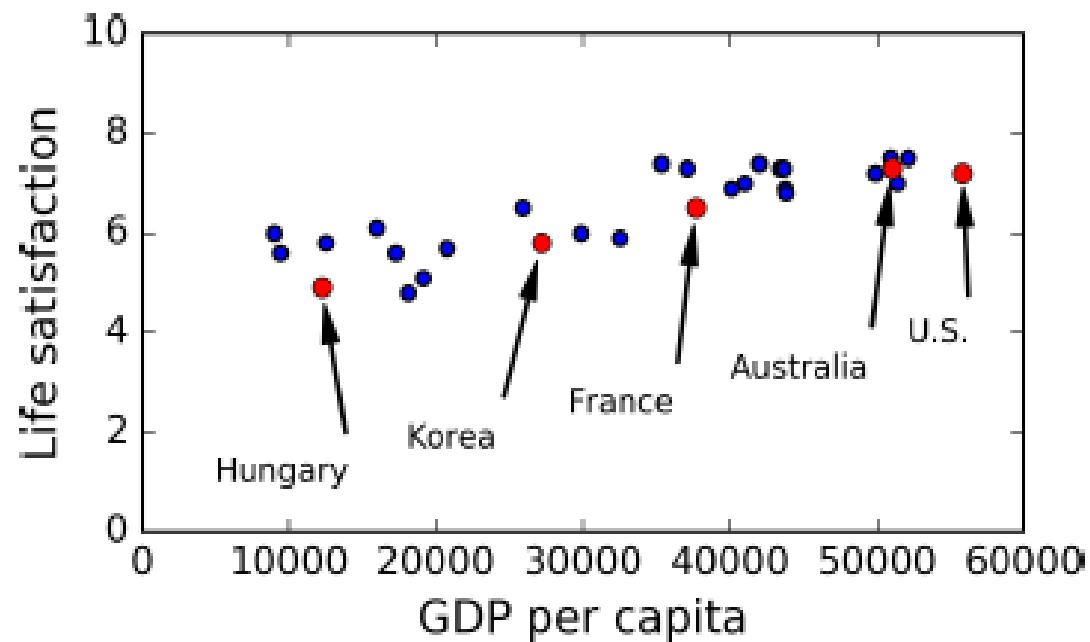
Instance Based Learning



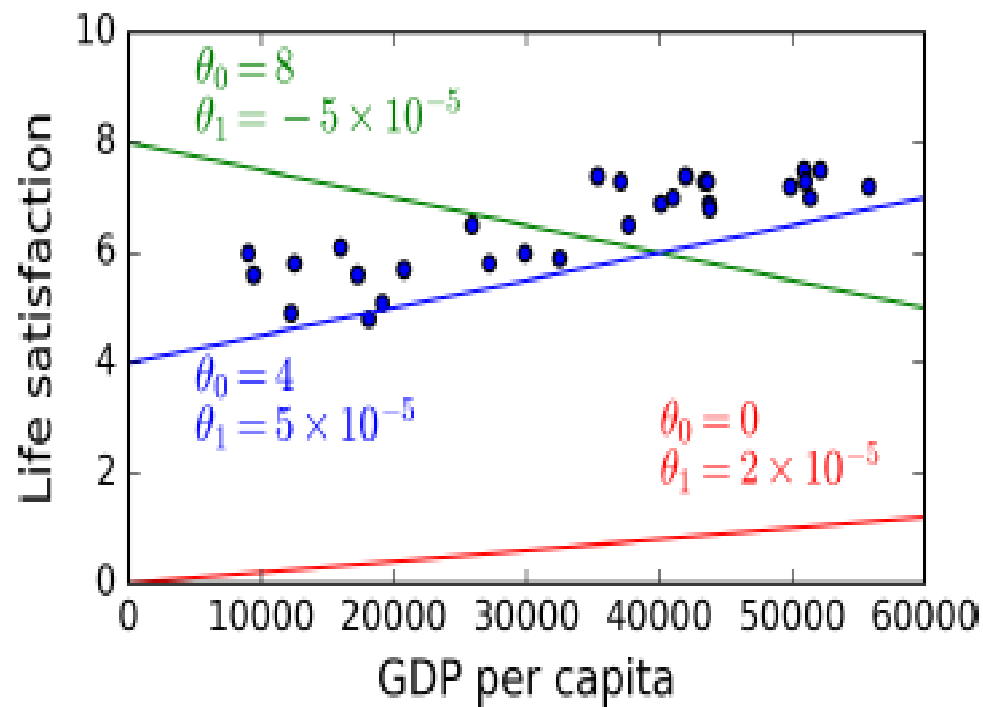
The diagram shows a 2D coordinate system with 'Feature 1' on the horizontal axis and 'Feature 2' on the vertical axis. Data points are represented by green triangles and yellow squares. A dashed line, labeled 'Model', represents the decision boundary. A 'New instance' (marked with a black 'X') is located near the boundary, and a dashed arc indicates its distance from the boundary.

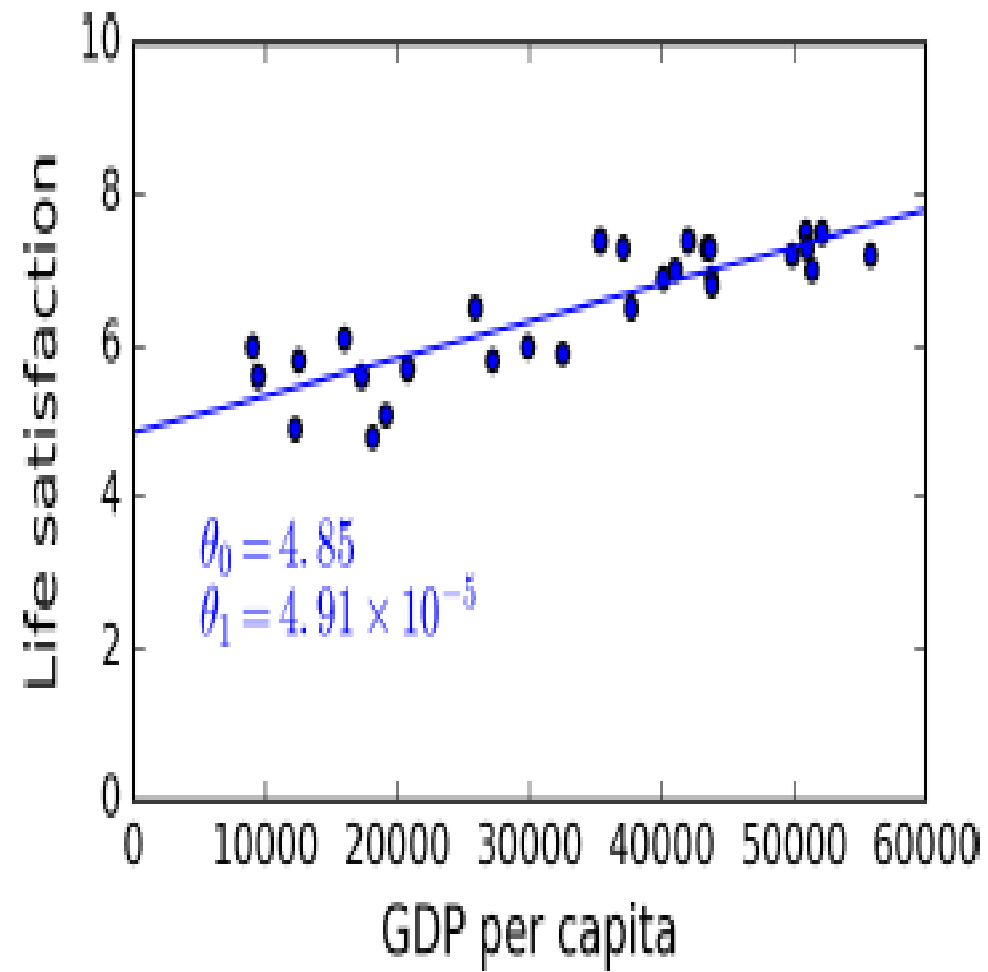
Table 1-1. Does money make people happier?

Country	GDP per capita (USD)	Life satisfaction
Hungary	12,240	4.9
Korea	27,195	5.8
France	37,675	6.5
Australia	50,962	7.3
United States	55,805	7.2



$$life_satisfaction = \theta_0 + \theta_1 \times GDP_per_capita$$





```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn

# Load the data
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv", thousands=',', delimiter='\t',
                             encoding='latin1', na_values="n/a")

# Prepare the data
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# Visualize the data
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

# Select a linear model
lin_reg_model = sklearn.linear_model.LinearRegression()

# Train the model
lin_reg_model.fit(X, y)

# Make a prediction for Cyprus
X_new = [[22587]] # Cyprus' GDP per capita
print(lin_reg_model.predict(X_new)) # outputs [[ 5.96242338]]
```

- Instance-based learning algorithm : Slovenia has the closest GDP per capita to that of Cyprus (\$20,732), and since the OECD' life satisfaction is 5.7, you would have predicted a life satisfaction of 5.7 for Cyprus.
- If you look at the two next closest countries, you will find Portugal and Spain with life satisfactions of 5.1 and 6.5, respectively. Averaging these three values, you get 5.77, which is pretty close to your model-based prediction. This simple algorithm is called *k-Nearest Neighbors regression* (in this example, $k = 3$).
- Replacing the Linear Regression model with k-Nearest Neighbors regression in the previous code is as simple as replacing this line:
 `clf = sklearn.linear_model.LinearRegression()`
with this one:
 `clf= sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)`

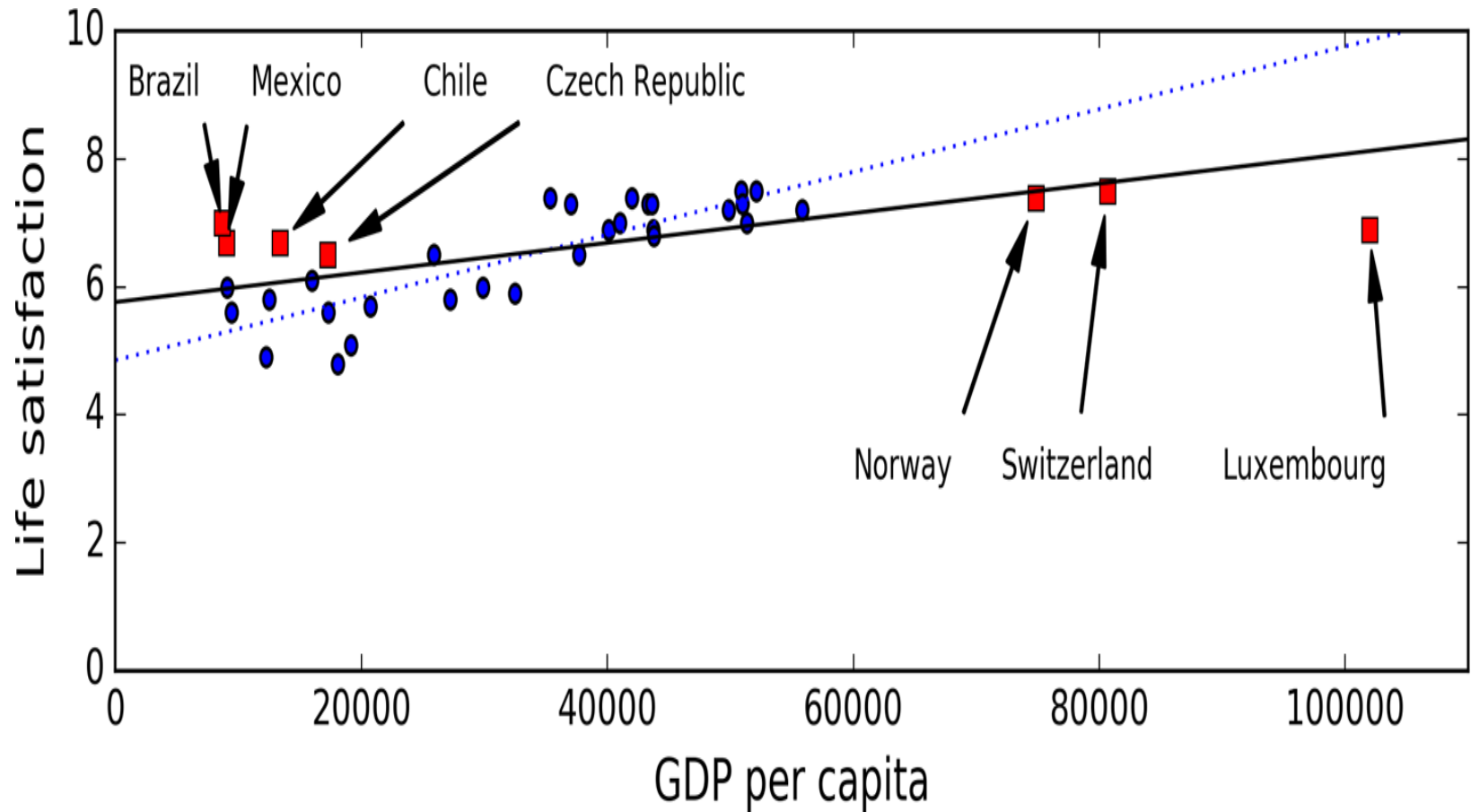
Challenges


- Insufficient Quantity of Training Data
- Non representative Training Data
- Poor-Quality Data
- Irrelevant Features
- Overfitting the Training Data
- Underfitting the Training Data

Insufficient Quantity of Training Data

- It takes a lot of data for most Machine Learning algorithms to work properly.
- Even for very simple problems you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples (unless you can reuse parts of an existing model).

Non Representative Training Data



- 
- It is crucial to use a training set that is representative of the cases you want to generalize to.
 - If the sample is too small, you will have *sampling noise* i.e., *nonrepresentative data*.
 - *Even very large* samples can be non representative if the sampling method is flawed. This is called *sampling bias*.

Poor Quality Data

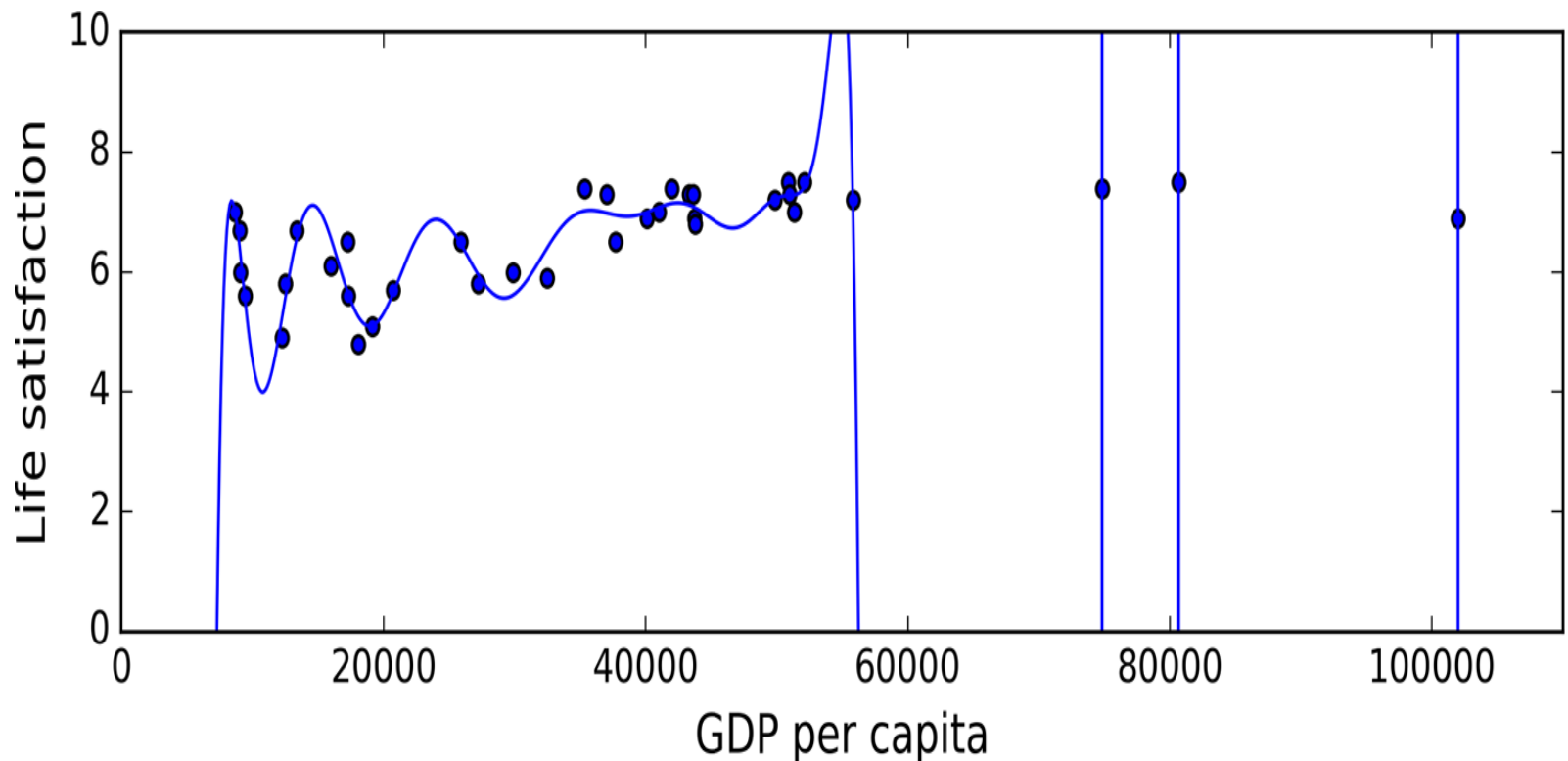
- If some instances are clearly outliers, it may help to simply discard them or try to fix the errors manually.
- If some instances are missing a few features (e.g., 5% of your customers did not specify their age)
- you must decide whether you want to ignore this attribute altogether, ignore these instances, fill in the missing values (e.g., with the median age), or train one model with the feature and one model without it, and so on.

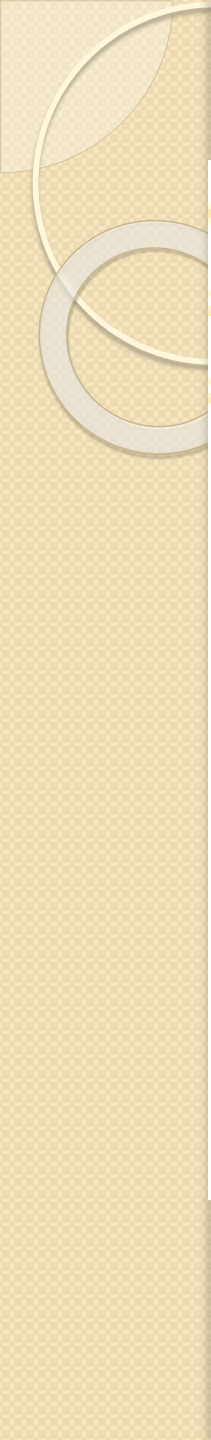
Irrelevant Features

- A critical part of the success of a Machine Learning project is coming up with a good set of features to train on. This process is called *feature engineering*.
- *Feature selection*: selecting the most useful features to train on among existing features.
- *Feature extraction*: combining existing features to produce a more useful one.
- Creating new features by gathering new data.

Overfitting the Training Data

- *Overfitting:* The model performs well on the training data, but it does not generalize well.

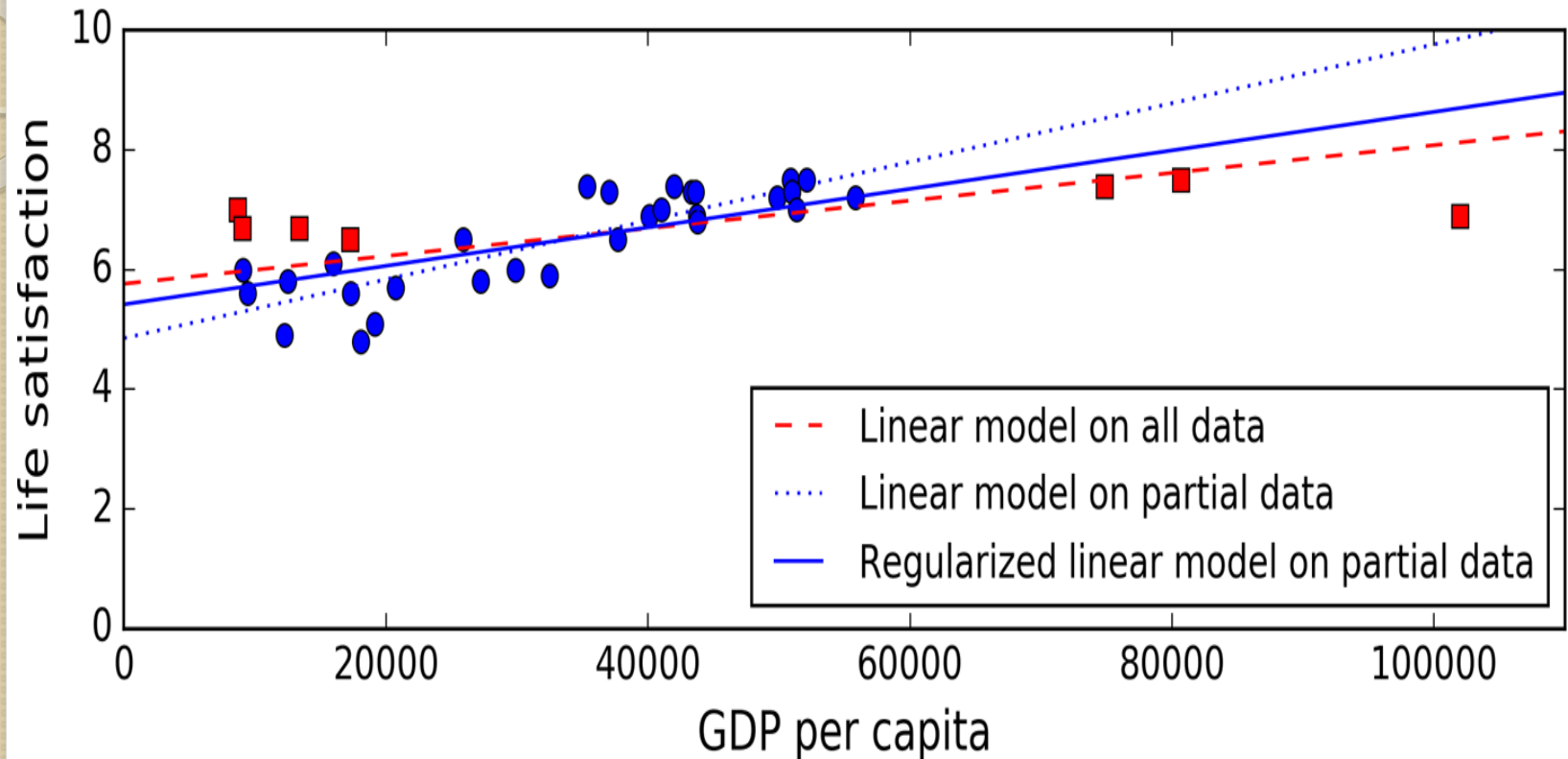




Overfitting happens when the model is too complex relative to the amount and noisiness of the training data. The possible solutions are:

- To simplify the model by selecting one with fewer parameters (e.g., a linear model rather than a high-degree polynomial model), by reducing the number of attributes in the training data or by constraining the model
- To gather more training data
- To reduce the noise in the training data (e.g., fix data errors and remove outliers)

- Constraining a model to make it simpler and reduce the risk of overfitting is called *regularization*.



The amount of regularization to apply during learning can be controlled by a *hyperparameter*. A hyperparameter is a parameter of a learning algorithm (not of the model)

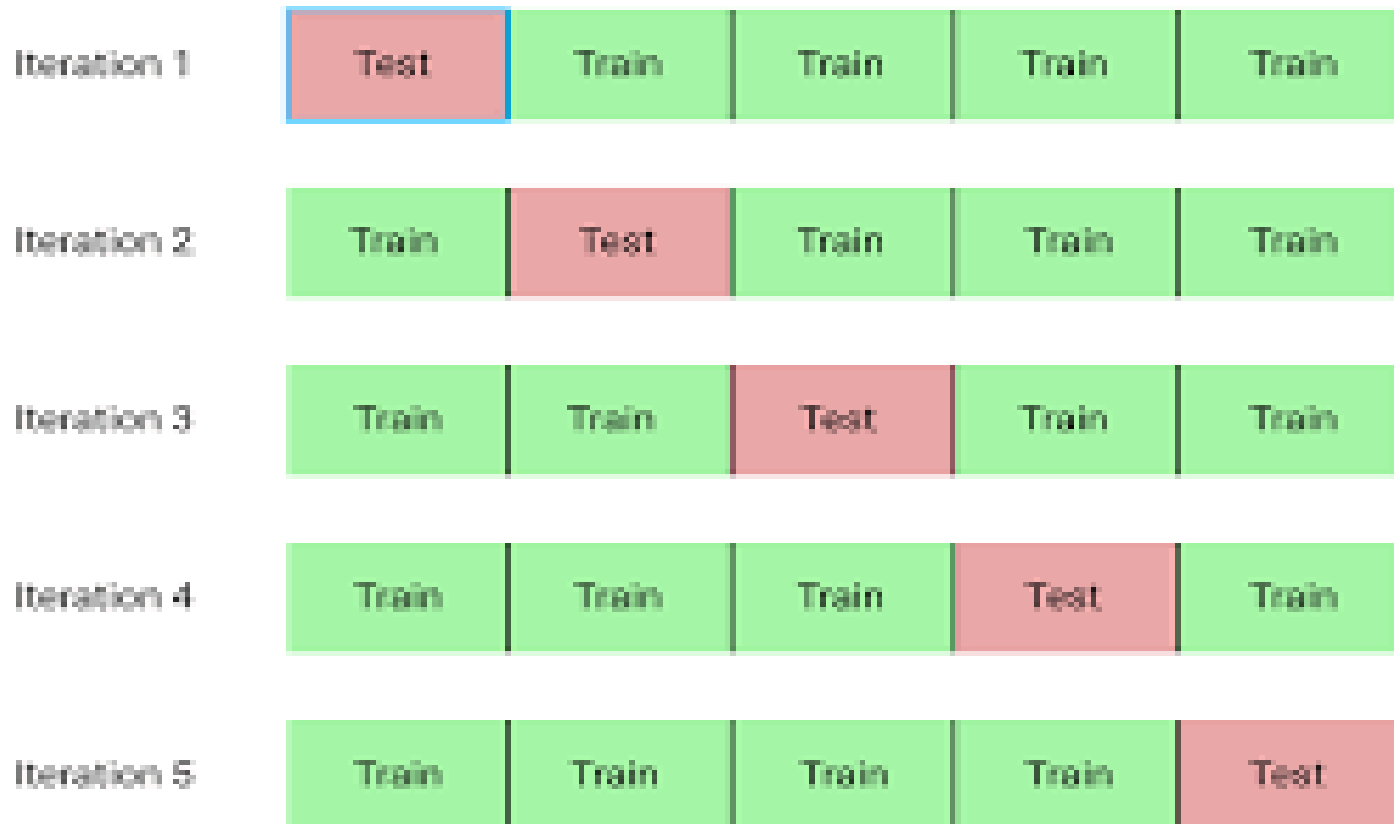
Underfitting the Training Data

- *Underfitting is the opposite of overfitting: it occurs when your model is too simple to learn the underlying structure of the data.*
- The main options to fix this problem are:
 - Selecting a more powerful model, with more parameters
 - Feeding better features to the learning algorithm (feature engineering)

Testing and Validating

- Split data in to two sets: Training set and test set.
- Error rate on new cases is called generalization error.
- If the training error is low but the generalization error is high, it means that your model is overfitting the training data.
- Cross validation — Split training set in to complementary subsets
- model is trained against a different combination of these subsets and validated against the remaining parts.

Cross Validation



End-to-End Machine Learning Project

- 1. Look at the big picture.
 Frame the problem
 Select the Performance Measure
- 2. Get the data.
- 3. Discover and visualize the data to gain insights.
- 4. Prepare the data for Machine Learning algorithms.
- 5. Select a model and train it.
- 6. Fine-tune your model.
- 7. Present your solution.
- 8. Launch, monitor, and maintain your system.

Look at the Big Picture

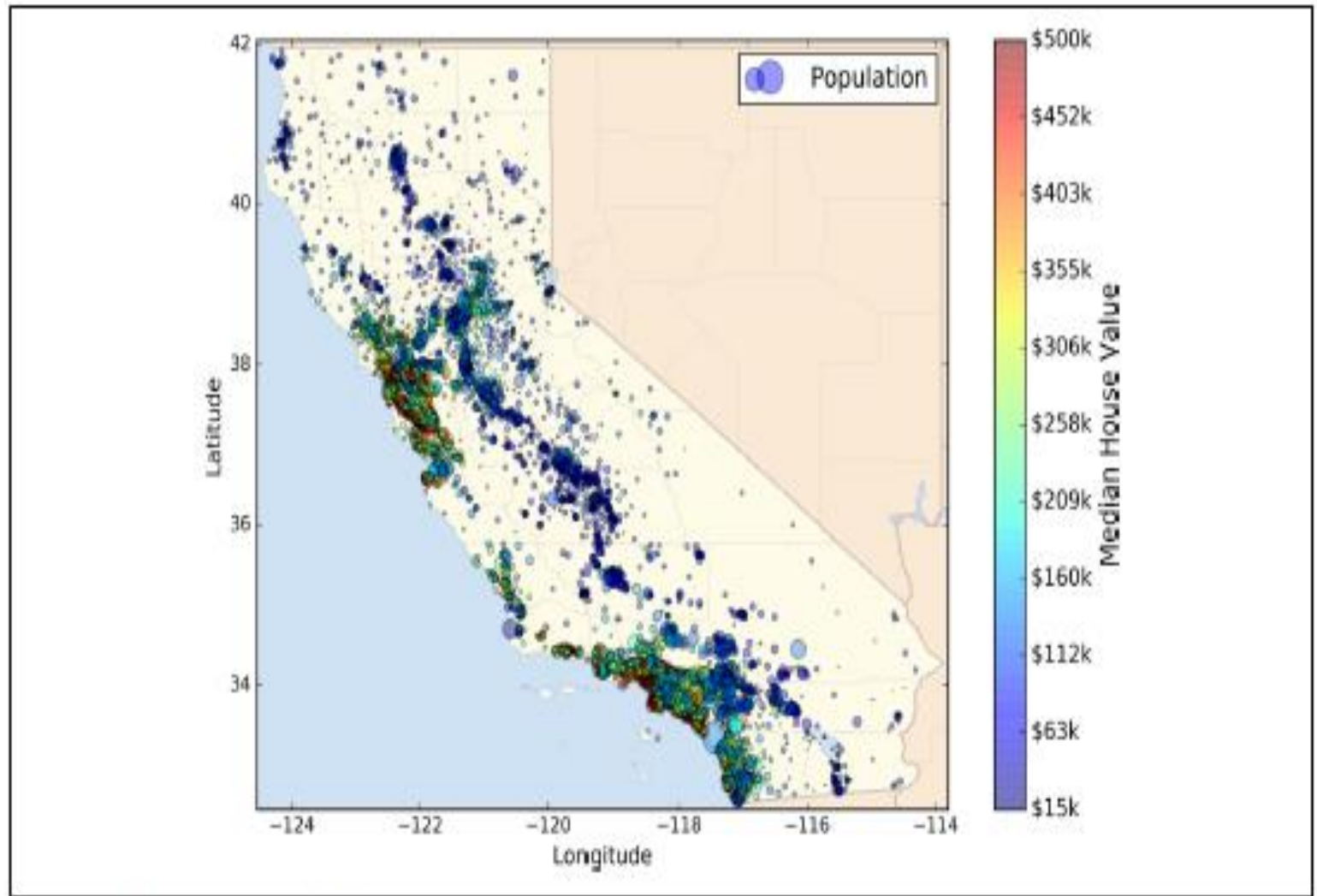
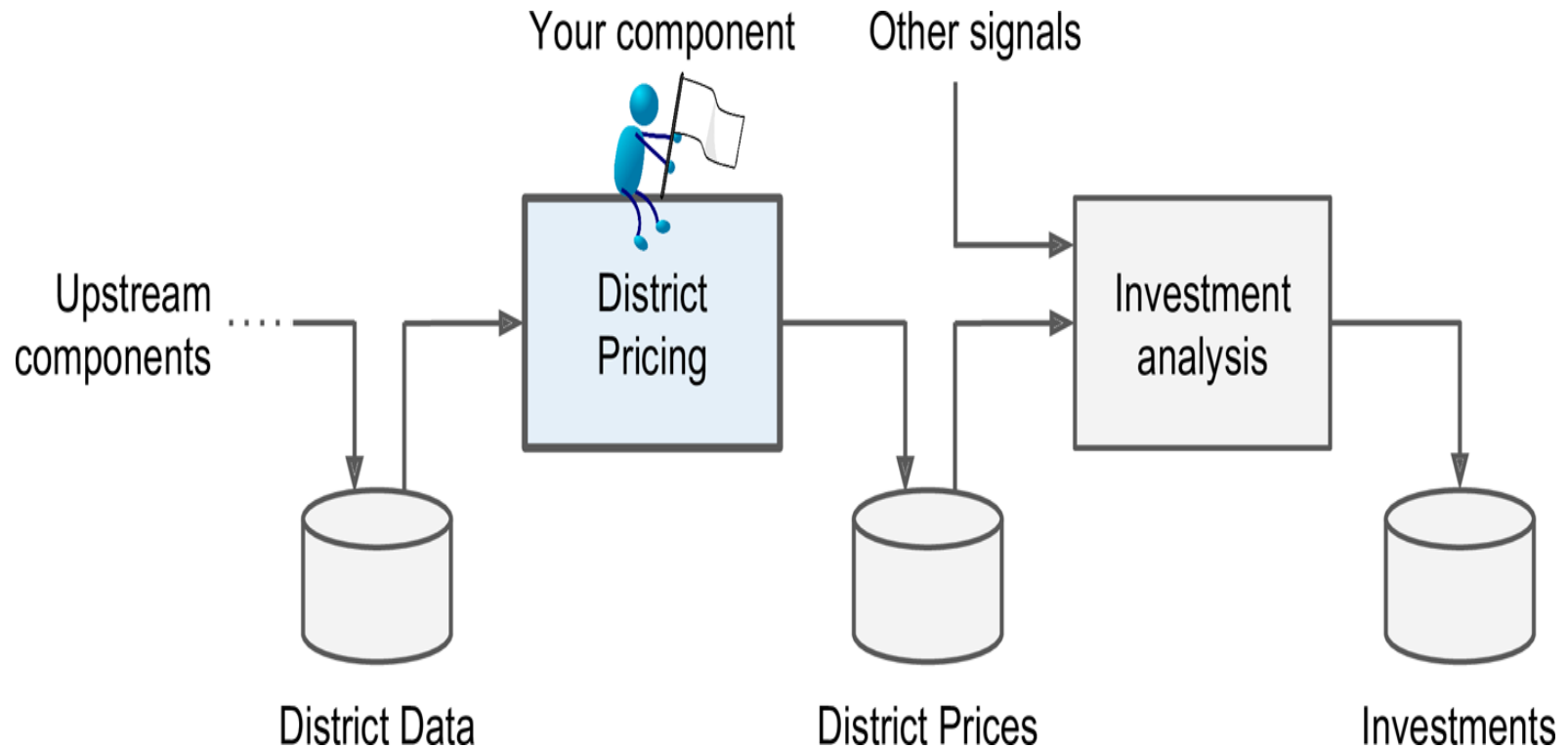


Figure 2-1. California housing prices

Working with Real data

- Popular data repositories:
 - UC Irvine Machine Learning Repository
 - Kaggle datasets
 - Amazon's AWS datasets
- Meta portals (they list open data repositories):
 - [*http://dataportals.org/*](http://dataportals.org/)
 - [*http://opendatamonitor.eu/*](http://opendatamonitor.eu/)
 - [*http://quandl.com/*](http://quandl.com/)
- Other pages listing many popular open data repositories:
 - Wikipedia's list of Machine Learning datasets
 - Quora.com question
 - Datasets subreddit

Machine Learning Pipeline for Real Estate Investments



Frame the Problem

- A sequence of data processing components is called a **data pipeline**
- First, you need to frame the problem: is it supervised, unsupervised, or Reinforcement Learning?
- Is it a classification task, a regression task, or something else?
- Should you use batch learning or online learning techniques?

Select a Performance Measure

- A typical performance measure for regression problems is the Root Mean Square Error (RMSE).

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

$\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n$ are predicted values

y_1, y_2, \dots, y_n are observed values

n is the number of observations

MNIST

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1, cache=False)
print mnist
```

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

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9	9

Datasets loaded by Scikit-Learn generally have a similar dictionary structure including:

- A DESCR key describing the dataset
- A data key containing an array with one row per instance and one column per feature
- A target key containing an array with the labels

Let's look at these arrays:

```
>>> X, y = mnist["data"], mnist["target"]
>>> X.shape
(70000, 784)
>>> y.shape
(70000,)
```


There are 70,000 images, and each image has 784 features. This is because each image is 28×28 pixels, and each feature simply represents one pixel's intensity, from 0 (white) to 255 (black). Let's take a peek at one digit from the dataset. All you need to do is grab an instance's feature vector, reshape it to a 28×28 array, and display it using Matplotlib's `imshow()` function:

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

some_digit = X[36000]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(some_digit_image, cmap = matplotlib.cm.binary,
           interpolation="nearest")
plt.axis("off")
plt.show()
```



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

```
import numpy as np
```

```
shuffle_index = np.random.permutation(60000)
```

```
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

Training a Binary Classifier

```
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
```

```
from sklearn.linear_model import SGDClassifier
```

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

```
>>> sgd_clf.predict([some_digit])
array([ True], dtype=bool)
```

Performance Measures

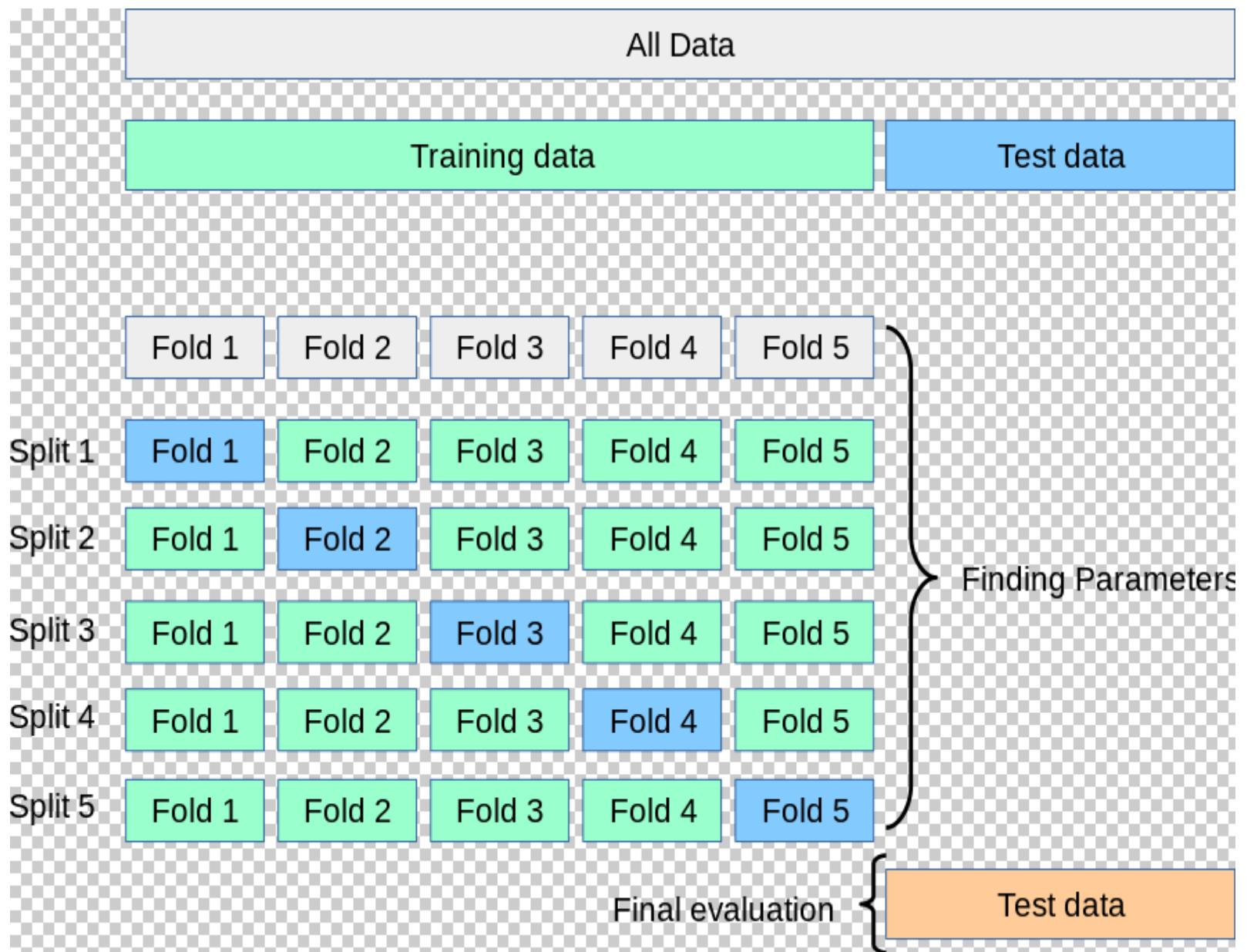
Measuring Accuracy Using Cross-Validation

Confusion Matrix

Precision and Recall

Precision/Recall Tradeoff

The ROC Curve



Accuracy - Accuracy is the ratio of correctly predicted observation to the total observations.

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.9502 ,  0.96565,  0.96495])
```

```
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)
```

Can you guess this model's accuracy? Let's find out:

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.909 ,  0.90715,  0.9128 ])
```

Confusion Matrix

```
from sklearn.model_selection import cross_val_predict
```

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

Now Get The Confusion Matrix

```
>>> from sklearn.metrics import confusion_matrix
```

```
>>> confusion_matrix(y_train_5, y_train_pred)
```

```
array([[53272,  1307],  
       [ 1077,  4344]])
```

	Predicted 0	Predicted 1
Actual 0	TN	FP
Actual 1	FN	TP

- Each row in a confusion matrix represents an *actual class*, while each column represents a *predicted class*.
- The first row of this matrix considers non-5 images (the *negative class*): 53,272 of them were correctly classified as non-5s (they are called *true negatives*), while the remaining 1,307 were wrongly classified as 5s (*false positives*).
- The second row considers the images of 5s (the *positive class*): 1,077 were wrongly classified as non-5s (*false negatives*), while the remaining 4,344 were correctly classified as 5s (*true positives*).

```
array([[53272, 1307],  
       [ 1077, 4344]])
```

- A perfect classifier would have only true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal(top left to bottom right):

```
array([[54579, 0],  
       [ 0, 5421]])
```

PRECISION and RECALL

- An interesting one to look at is the accuracy of the positive predictions, this is called the *precision of the classifier*.

$$\text{precision} = \frac{TP}{TP + FP}$$

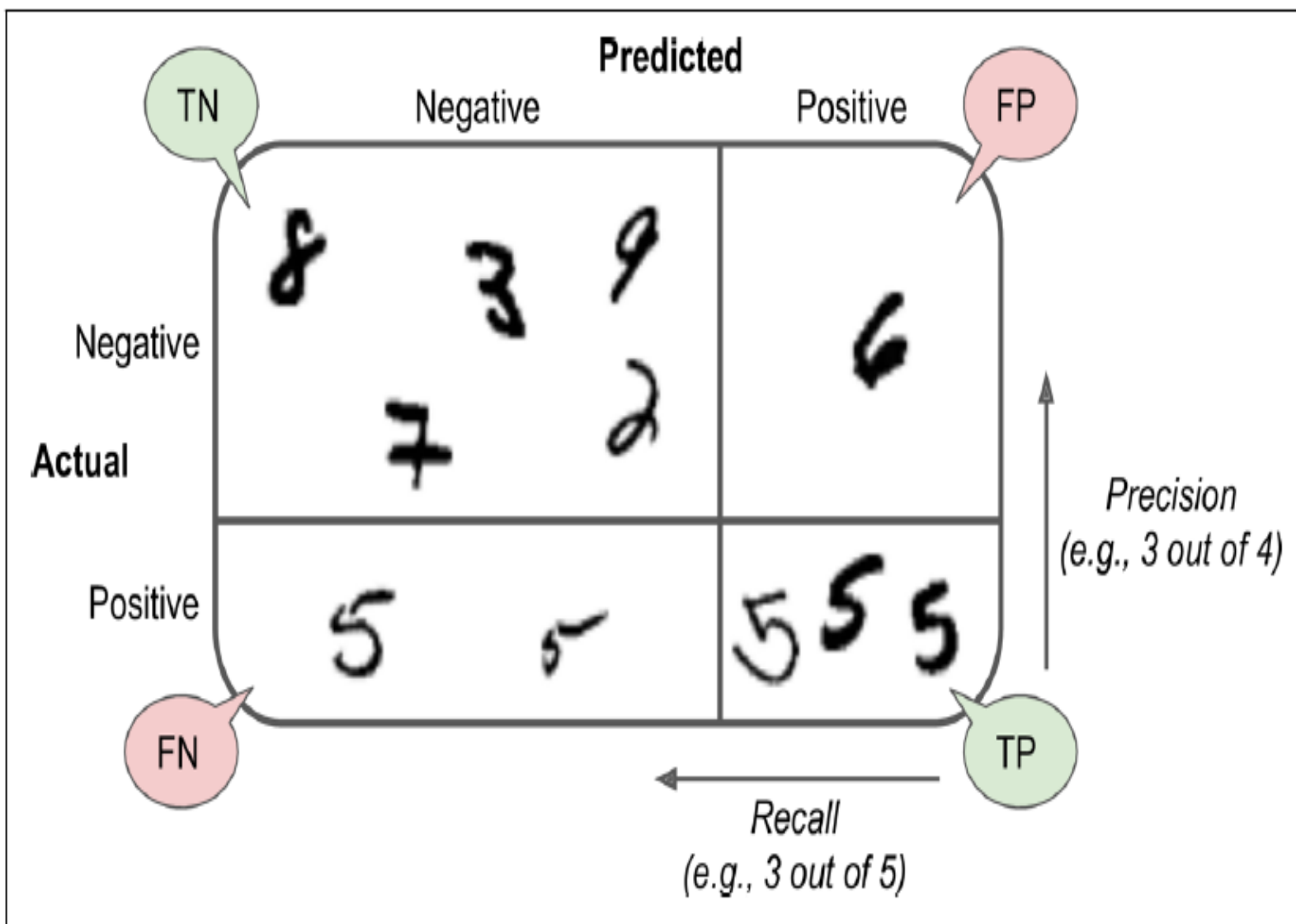
TP is the number of true positives, and FP is the number of false positives.

PRECISION and RECALL

- precision is typically used along with another metric named *recall*, also called *sensitivity* or *true positive rate (TPR)*:
- *This is the ratio of positive instances that are correctly detected by the classifier.*

$$\text{recall} = \frac{TP}{TP + FN}$$

FN is of course the number of false negatives.



Precision and Recall


Scikit-Learn provides several functions to compute classifier metrics, including precision and recall:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_pred)      # == 4344 / (4344 + 1307)
0.76871350203503808
>>> recall_score(y_train_5, y_train_pred)  # == 4344 / (4344 + 1077)
0.79136690647482011
```

```
array([[53272, 1307],
       [ 1077, 4344]])
```

- It is often convenient to combine precision and recall into a single metric called the F_1 score.
- *The F_1 score is the harmonic mean of precision and recall*
Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values.
- As a result, the classifier will only get a high F_1 score if both recall and precision are high.

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$



To compute the F_1 score, simply call the `f1_score()` function:

```
>>> from sklearn.metrics import f1_score  
>>> f1_score(y_train_5, y_pred)  
0.78468208092485547
```


- The F_1 score favors classifiers that have similar precision and recall. This is not always what you want: in some contexts you mostly care about precision, and in other contexts you really care about recall.
- Suppose you train a classifier to detect shoplifters on surveillance images: it is probably fine if your classifier has only 30% precision as long as it has 99% recall.
- Unfortunately, you can't have it both ways: increasing precision reduces recall, and vice versa. This is called the *precision/recall tradeoff*.

- **Accuracy** - Accuracy is the ratio of correctly predicted observation to the total observations.
- **Precision** - Precision is the ratio of correctly predicted positive observations to the total predicted positive observations.
- **Recall** - Recall is the ratio of correctly predicted positive observations to the all observations in actual class.
- **F₁ score** - F₁ Score is the weighted average of Precision and Recall.

Precision/Recall Tradeoff

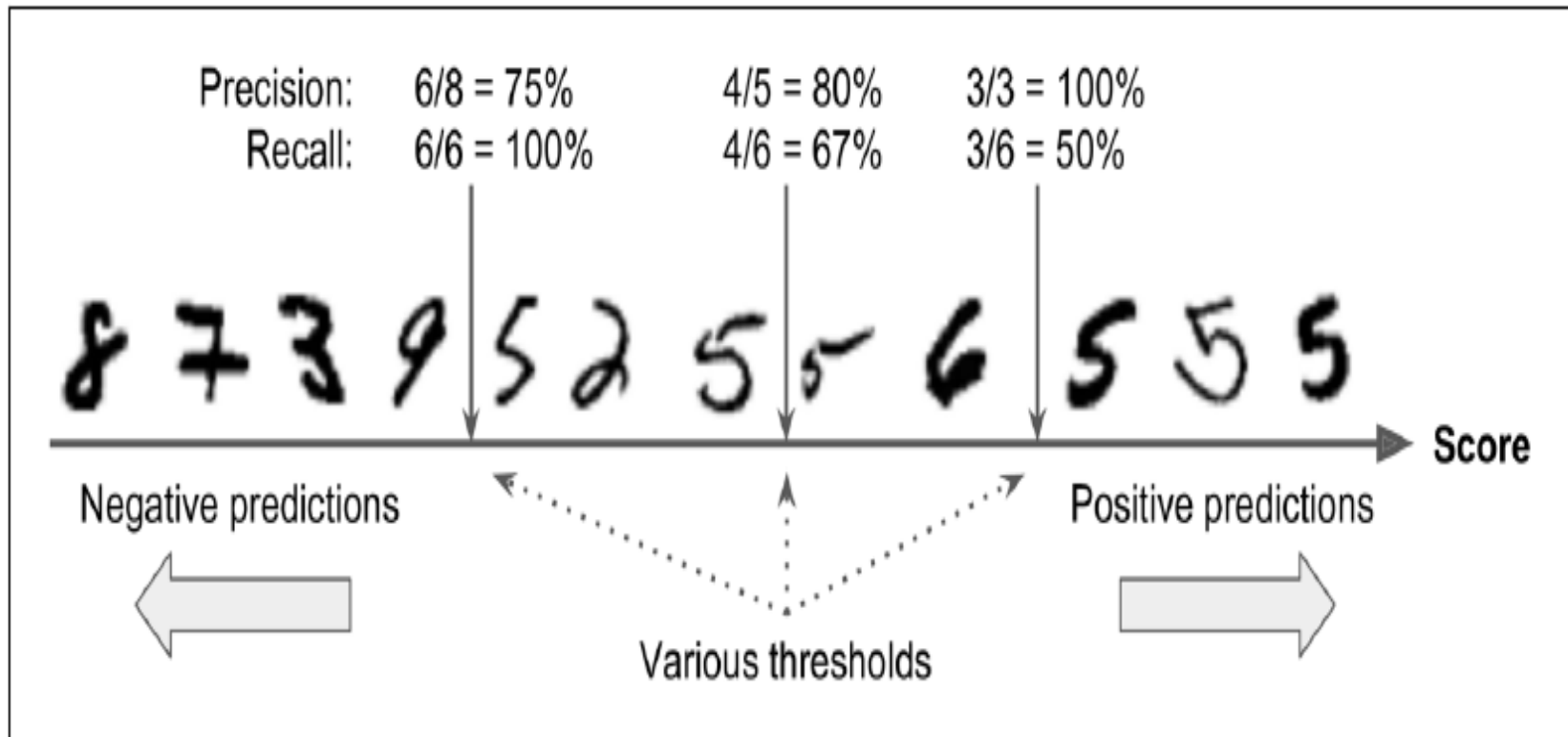


Figure 3-3. Decision threshold and precision/recall tradeoff

- Scikit-Learn does not let you set the threshold directly, but it does give you access to the decision scores that it uses to make predictions.
- Instead of calling the classifier's `predict()` method, you can call its `decision_function()` method, which returns a score for each instance, and then make predictions based on those scores using any threshold you want:
- **`y_scores= sgd_clf.decision_function([some_digit])`**
`y_scores`
`array([161855.74572176])`
`threshold = 0`
`y_some_digit_pred = (y_scores > threshold)`
`array([True], dtype=bool)`

- The SGDClassifier uses a threshold equal to 0, so the previous code returns the same result as the predict() method (i.e., True). Let's raise the threshold:
- **threshold = 200000**
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
array([False], dtype=bool)
- This confirms that raising the threshold decreases recall. The image actually represents a 5, and the classifier detects it when the threshold is 0, but it misses it when the threshold is increased to 200,000.

- So how can you decide which threshold to use?
- For this you will first need to get the scores of all instances in the training set using the `cross_val_predict()` function again.
- But this time specifying that you want it to return decision scores instead of predictions:

```
y_scores = cross_val_predict(sgd_clf, X_train,  
y_train_5, cv=3, method="decision_function")
```

- Now with these scores you can compute precision and recall for all possible thresholds
- using the `precision_recall_curve()` function:

```
from sklearn.metrics import precision_recall_curve
```

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

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):  
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")  
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")  
    plt.xlabel("Threshold")  
    plt.legend(loc="upper left")  
    plt.ylim([0, 1])
```

```
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)  
plt.show()
```

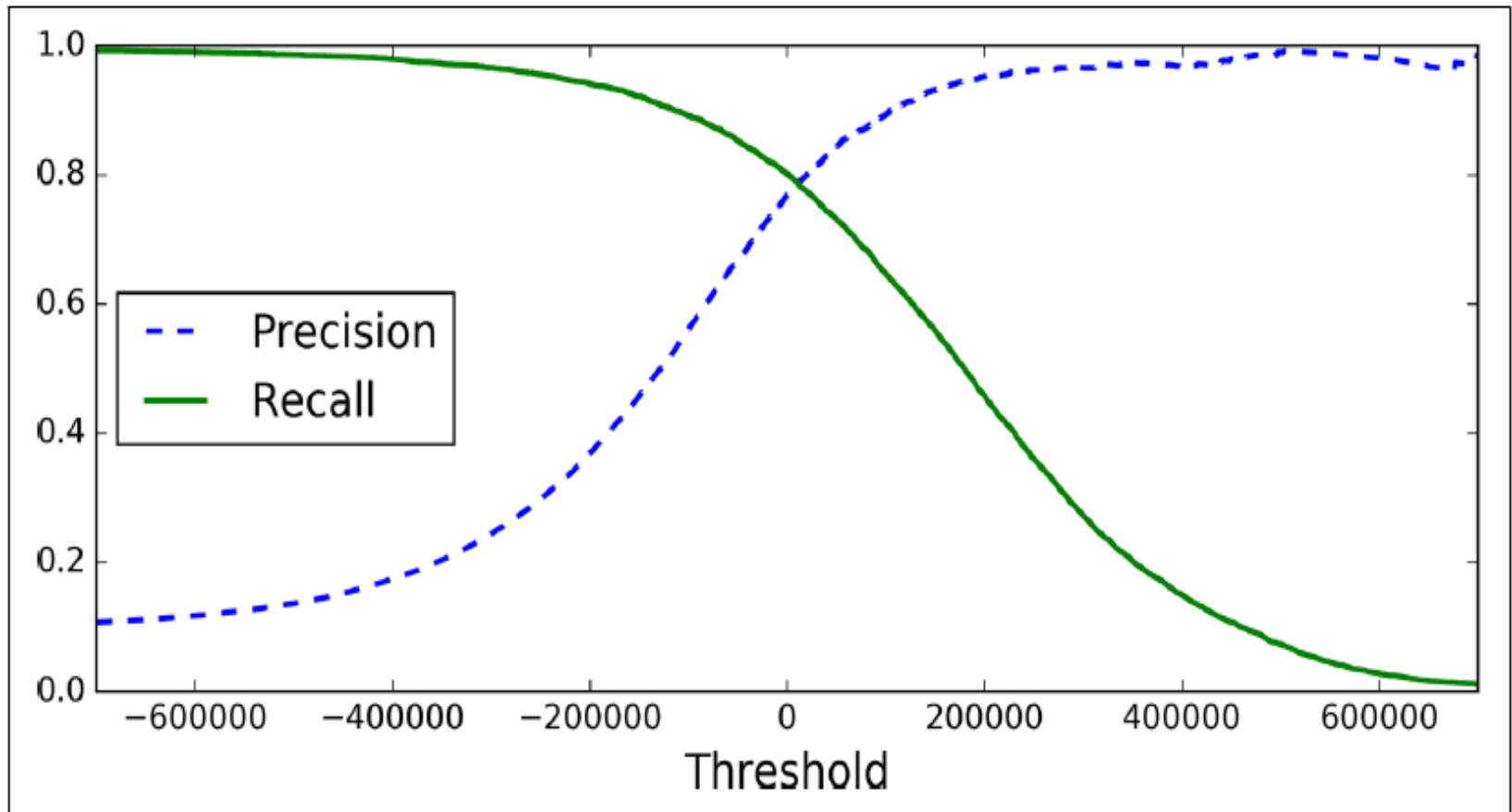



Figure 3-4. Precision and recall versus the decision threshold

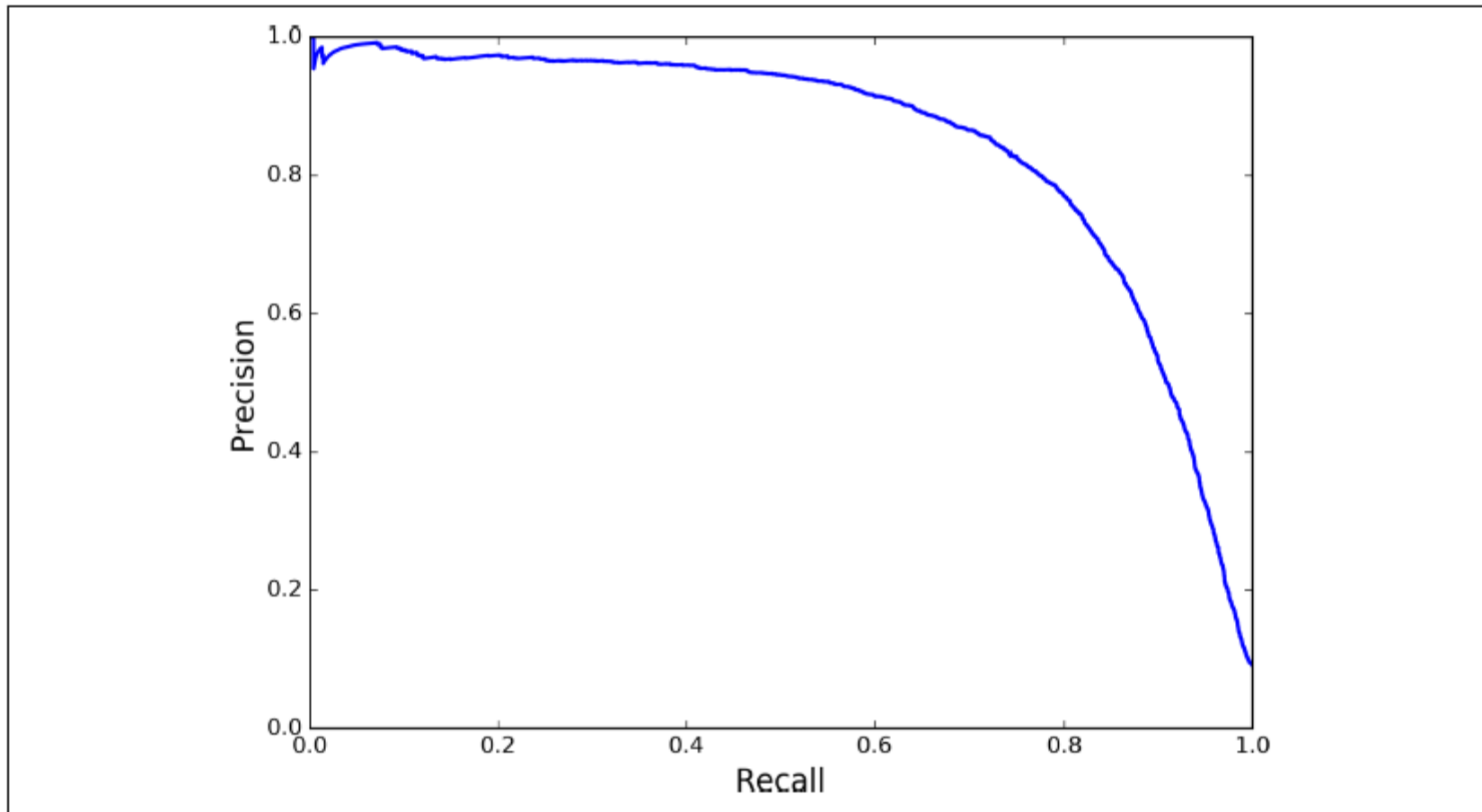


Figure 3-5. Precision versus recall



```
y_train_pred_90 = (y_scores > 70000)
```

Let's check these predictions' precision and recall:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.8998702983138781
```

```
>>> recall_score(y_train_5, y_train_pred_90)
0.63991883416343853
```

The ROC Curve

- The *receiver operating characteristic (ROC) curve* is another common tool used with binary classifiers.
- It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the *true positive rate* (another name for recall) against the *false positive rate*.
- The *FPR* is the ratio of negative instances that are incorrectly classified as positive. It is equal to one minus the *true negative rate*, which is the ratio of negative instances that are correctly classified as negative.

$$\text{FPR} = \text{FP} / \text{FP} + \text{TN}$$

- The *TNR* is also called *specificity*. Hence the ROC curve plots sensitivity (recall) versus $1 - \text{specificity}$.

$$\text{TNR} = \text{TN} / \text{TN} + \text{FP}$$

To plot the ROC curve, you first need to compute the TPR and FPR for various threshold values, using the `roc_curve()` function:

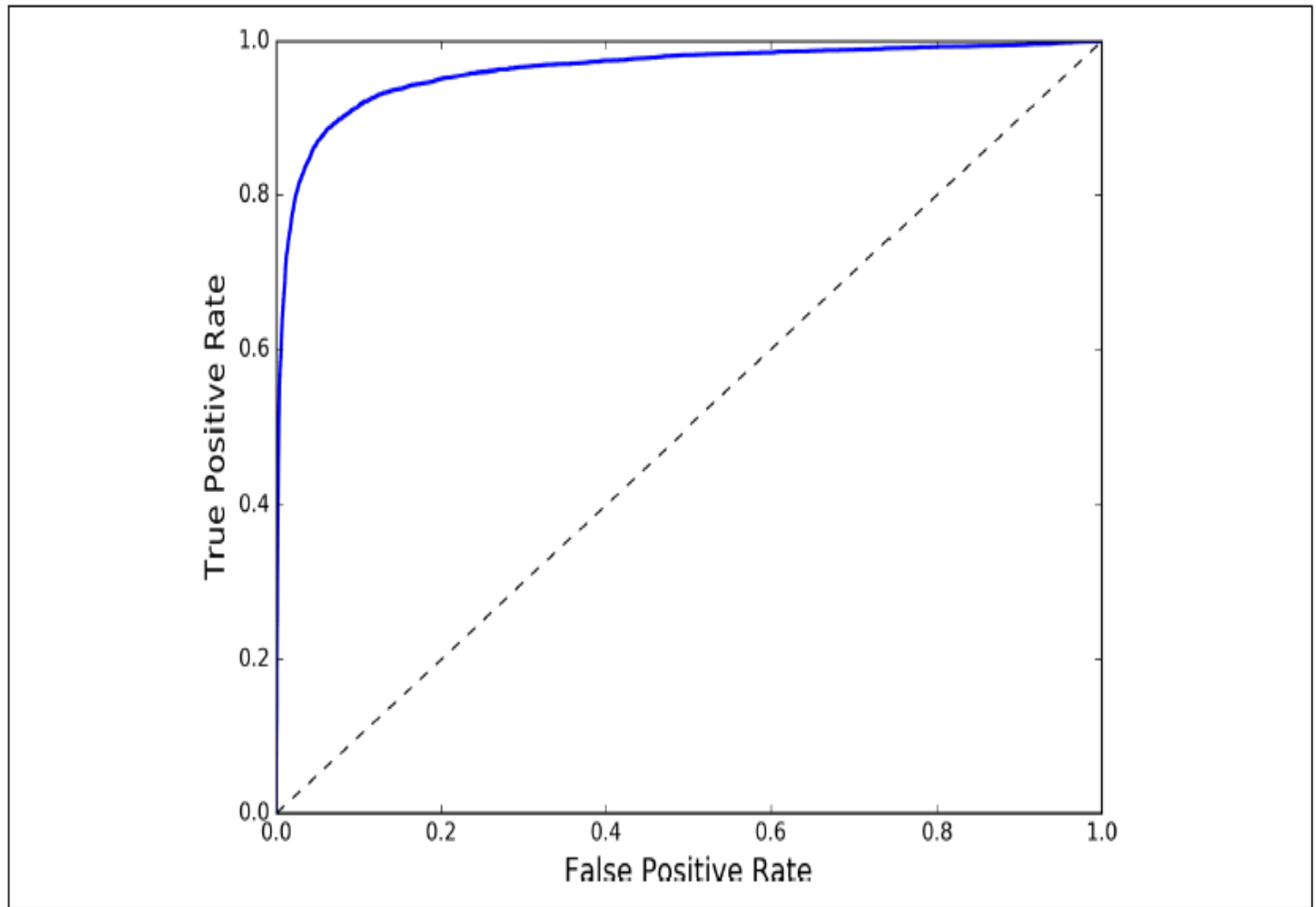
```
from sklearn.metrics import roc_curve
```


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

Then you can plot the FPR against the TPR using Matplotlib. This code produces the plot in [Figure 3-6](#):

```
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('False Positive Rate')  
    plt.ylabel('True Positive Rate')
```

```
plot_roc_curve(fpr, tpr)  
plt.show()
```





One way to compare classifiers is to measure the *area under the curve* (AUC). A perfect classifier will have a *ROC AUC* equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5. Scikit-Learn provides a function to compute the ROC AUC:

```
>>> from sklearn.metrics import roc_auc_score  
>>> roc_auc_score(y_train_5, y_scores)  
0.97061072797174941
```

Multiclass Classification

- Binary classifiers distinguish between two classes, *multiclass classifiers* (also called *multinomial classifiers*) can distinguish between more than two classes.
- One way to create a system that can classify the digit images into 10 classes (from 0 to 9) is to train 10 binary classifiers, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on).
- Then when you want to classify an image, you get the decision score from each classifier for that image and you select the class whose classifier outputs the highest score.
- This is called the *one-versus-all (OvA) strategy* (also called *one-versus-the-rest*).

- Another strategy is to train a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on.
- This is called the *one-versus-one* (*OvO*) strategy. If there are N classes, you need to train $N \times (N - 1) / 2$ classifiers. For the MNIST problem, this means training 45 binary classifiers.
- When you want to classify an image, you have to run the image through all 45 classifiers and see which class wins the most duels.
- The main advantage of OvO is that each classifier only needs to be trained on the part of the training set for the two classes that it must distinguish.

- Scikit-Learn detects when you try to use a binary classification algorithm for a multiclass classification task, and it automatically runs OvA (except for SVM classifiers for which it uses OvO).
- Let's try this with the SGDClassifier:
sgd_clf.fit(X_train, y_train)
sgd_clf.predict([some_digit])
array([5.])
- This code trains the SGDClassifier on the training set using the original target classes from 0 to 9 (y_train).
- Then it makes a prediction (a correct one in this case). Under the hood, Scikit-Learn actually trained 10 binary classifiers, got their decision scores for the image, and selected the class with the highest score.

- `some_digit_scores =`
`sgd_clf.decision_function([some_digit])`
`print(some_digit_scores)`

```
array([[ -311402.62954431,  -363517.28355739,  -446449.5306454 ,  
        -183226.61023518,  -414337.15339485,   161855.74572176,  
        -452576.39616343,  -471957.14962573,  -518542.33997148,  
        -536774.63961222]])
```

The highest score is indeed the one corresponding to class 5:

```
>>> np.argmax(some_digit_scores)  
5  
>>> sgd_clf.classes_  
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])  
>>> sgd_clf.classes[5]  
5.0
```

```
>>> from sklearn.multiclass import OneVsOneClassifier
>>> ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
>>> ovo_clf.fit(X_train, y_train)
>>> ovo_clf.predict([some_digit])
array([ 5.])
>>> len(ovo_clf.estimators_)
45
```

Training a RandomForestClassifier is just as easy:

```
>>> forest_clf.fit(X_train, y_train)
>>> forest_clf.predict([some_digit])
array([ 5.])

>>> forest_clf.predict_proba([some_digit])
array([[ 0.1,  0. ,  0. ,  0.1,  0. ,  0.8,  0. ,  0. ,  0. ,  0.]])
```

Error Analysis

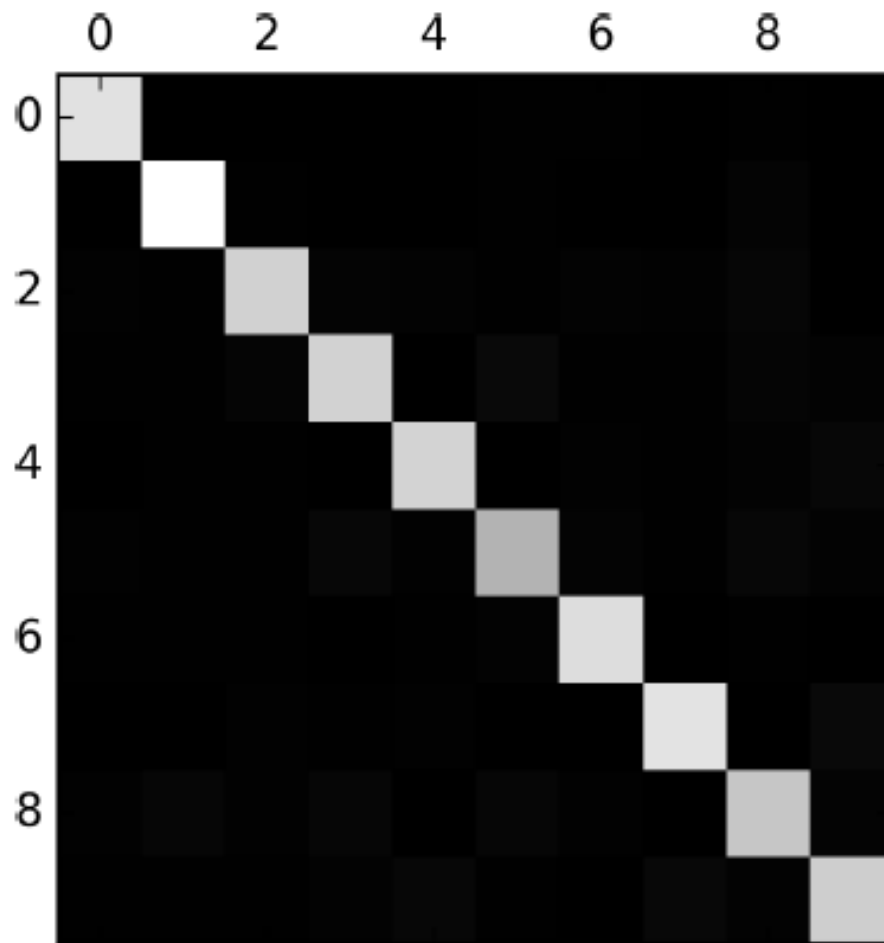
```
>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
```

```
>>> conf_mx = confusion_matrix(y_train, y_train_pred)
```

```
>>> conf_mx
```

```
array([[5725,   3,  24,   9,  10,  49,  50,  10,  39,   4],
       [   2, 6493,  43,  25,   7,  40,   5,  10, 109,   8],
       [  51,  41, 5321, 104,  89,  26,  87,  60, 166,  13],
       [  47,  46, 141, 5342,   1, 231,  40,  50, 141,  92],
       [  19,  29,  41,  10, 5366,   9,  56,  37,  86, 189],
       [  73,  45,  36, 193,  64, 4582, 111,  30, 193,  94],
       [  29,  34,  44,   2,  42,  85, 5627,  10,  45,   0],
       [  25,  24,  74,  32,  54,  12,   6, 5787,  15, 236],
       [  52, 161,  73, 156,  10, 163,  61,  25, 5027, 123],
       [  43,  35,  26,  92, 178,  28,   2, 223,  82, 5240]])
```

```
plt.matshow(conf_mx, cmap=plt.cm.gray)  
plt.show()
```

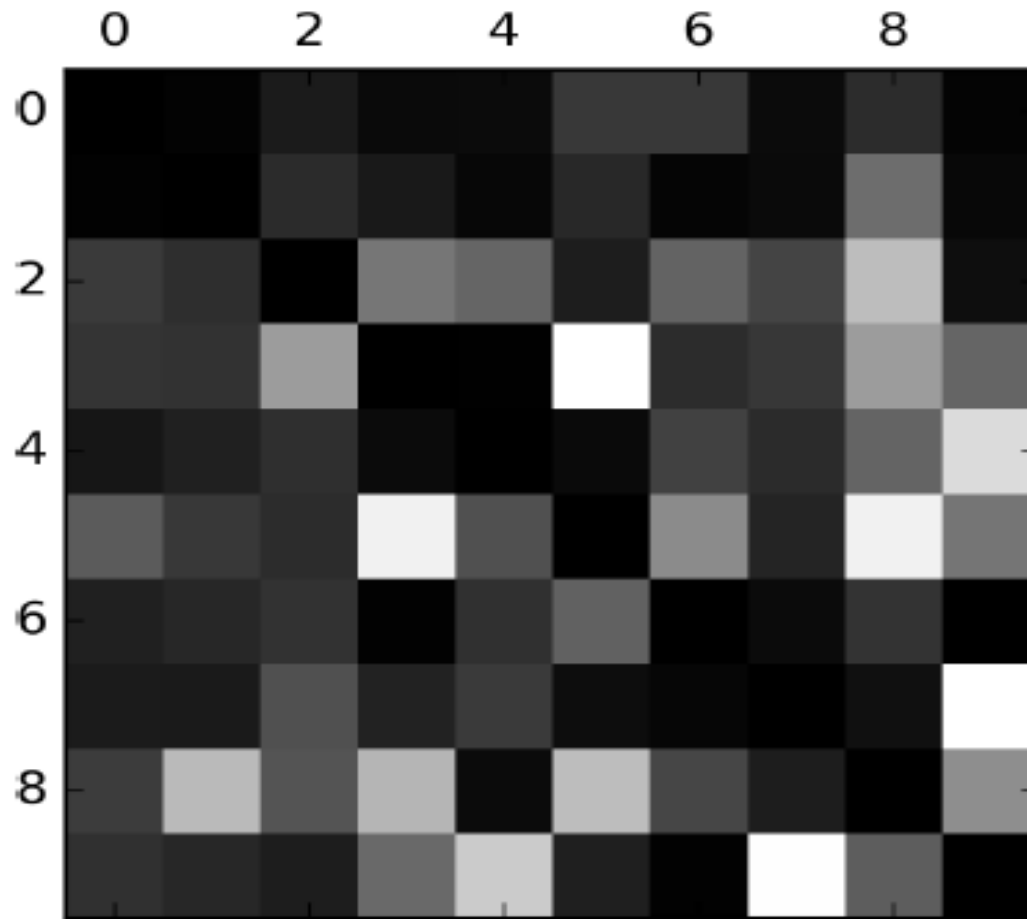


- Focus the plot on errors
- First, you need to divide each value in the confusion matrix by the number of images in the corresponding class

```
row_sums = conf_mx.sum(axis=1, keepdims=True)  
norm_conf_mx = conf_mx / row_sums
```

Now let's fill the diagonal with zeros to keep only the errors, and let's plot the result:

```
np.fill_diagonal(norm_conf_mx, 0)  
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)  
plt.show()
```




```
cl_a, cl_b = 3, 5
```

```
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
```

```
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
```

```
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
```

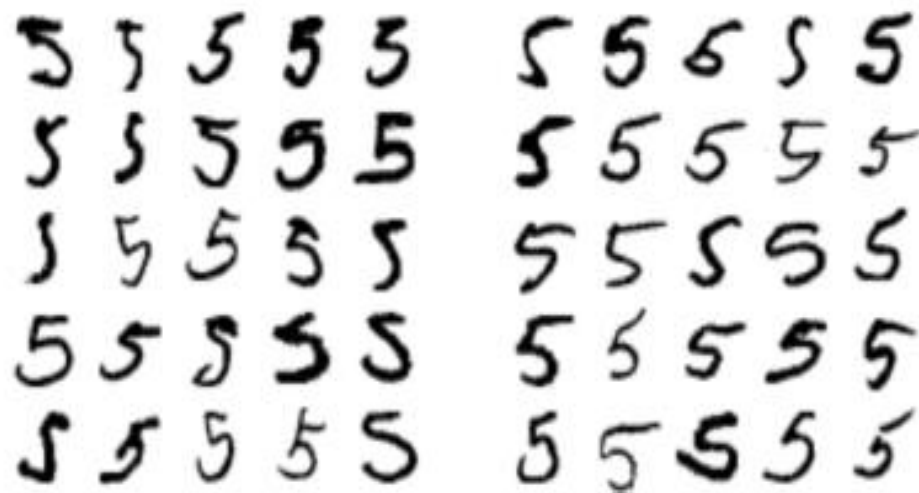
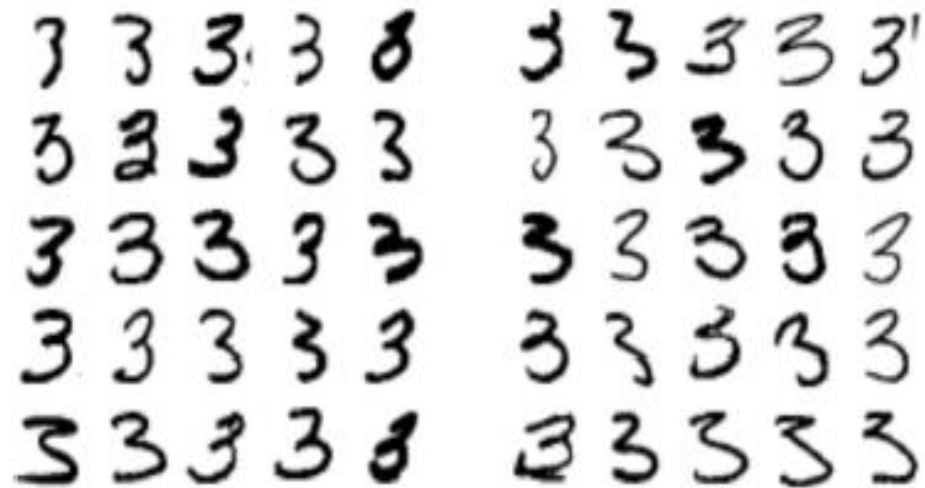
```
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]
```

```
plt.figure(figsize=(8,8))
```

```
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
```

```
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
```

```
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
plt.show()
```



Multilabel Classification

- Until now each instance has always been assigned to just one class. In some cases you may want your classifier to output multiple classes for each instance.
- For example, consider a face-recognition classifier: what should it do if it recognizes several people on the same picture? Of course it should attach one label per person it recognizes.
- Say the classifier has been trained to recognize three faces, Alice, Bob, and Charlie; then when it is shown a picture of Alice and Charlie, it should output $[1, 0, 1]$ (meaning “Alice yes, Bob no, Charlie yes”).
- Such a classification system that outputs multiple binary labels is called a *multilabel classification system*.

```
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)
```

```
>>> knn_clf.predict([some_digit])
```

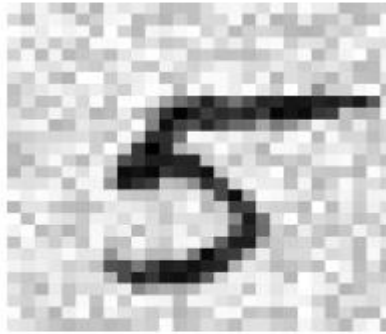
```
array([[False,  True]], dtype=bool)
```

Multiclass Classification

- The last type of classification task we are going to discuss here is called *Multiclass-multiclass classification* (or simply *multiclass classification*).
- It is simply a generalization of multilabel classification where each label can be multiclass (i.e., it can have more than two possible values).
- To illustrate this, let's build a system that removes noise from images. It will take as input a noisy digit image, and it will (hopefully) output a clean digit image, represented as an array of pixel intensities, just like the MNIST images.
- Notice that the classifier's output is multilabel (one label per pixel) and each label can have multiple values (pixel intensity ranges from 0 to 255). It is thus an example of a Multiclass classification system.

Let's start by creating the training and test sets by taking the MNIST images and adding noise to their pixel intensities using NumPy's `randint()` function. The target images will be the original images:

```
noise = rnd.randint(0, 100, (len(X_train), 784))  
noise = rnd.randint(0, 100, (len(X_test), 784))  
X_train_mod = X_train + noise  
X_test_mod = X_test + noise  
y_train_mod = X_train  
y_test_mod = X_test
```



```
knn_clf.fit(X_train_mod, y_train_mod)  
clean_digit = knn_clf.predict([X_test_mod[some_index]])  
plot_digit(clean_digit)
```



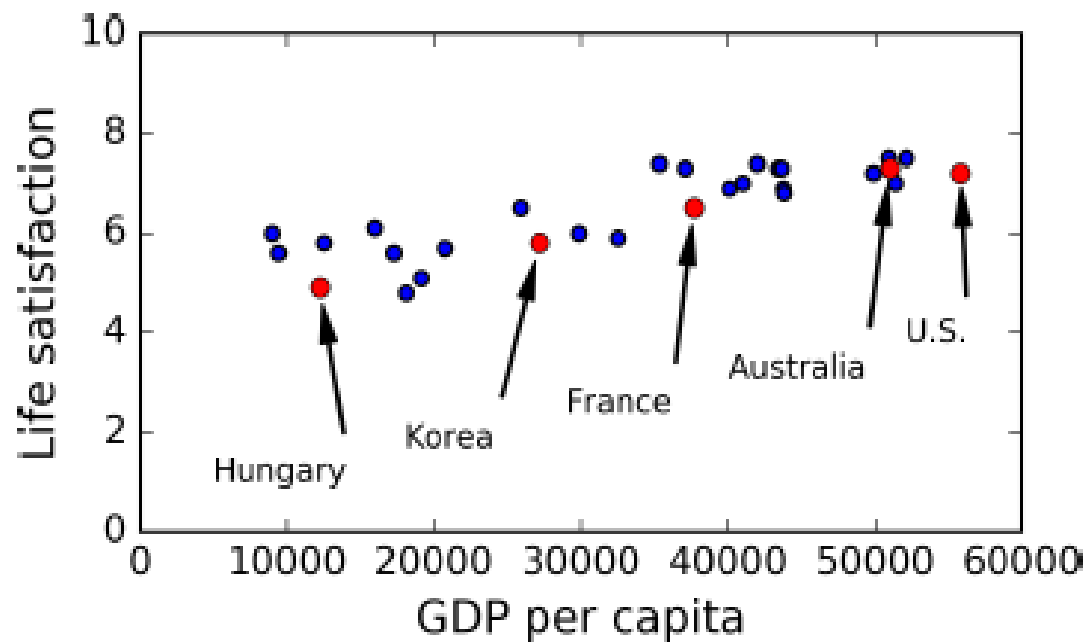
Training Models

Ways of training a Linear Regression Model:

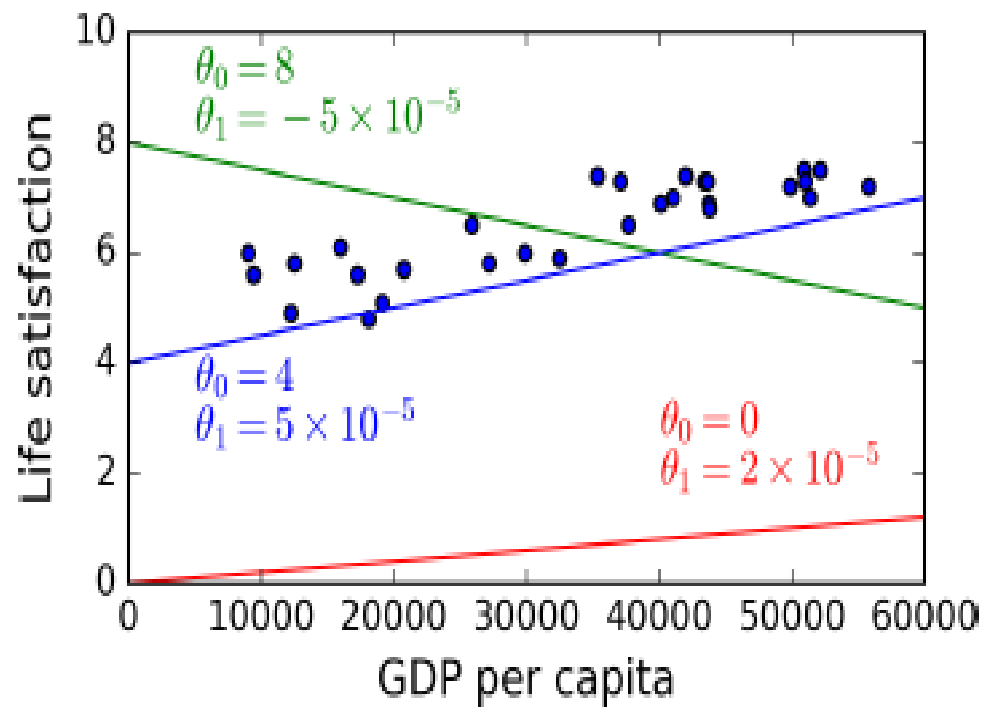
- Using a direct “closed-form” equation that directly computes the model parameters that best fit the model to the training set
- Using an iterative optimization approach, called Gradient Descent (GD), that gradually tweaks the model parameters to minimize the cost function over the training set, eventually converging to the same set of parameters as the first method.

Table 1-1. Does money make people happier?

Country	GDP per capita (USD)	Life satisfaction
Hungary	12,240	4.9
Korea	27,195	5.8
France	37,675	6.5
Australia	50,962	7.3
United States	55,805	7.2



$$life_satisfaction = \theta_0 + \theta_1 \times GDP_per_capita$$



Performance Measure

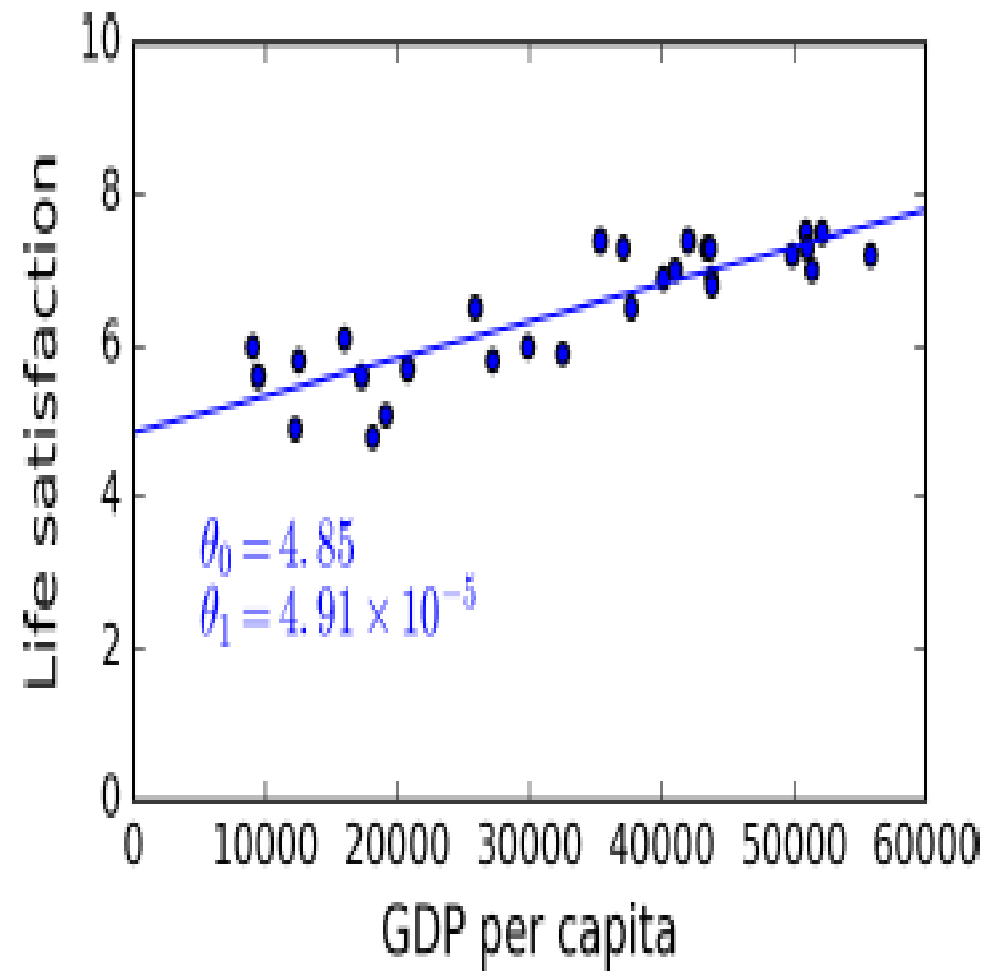
- A typical performance measure for regression problems is the Root Mean Square Error (RMSE).

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

$\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n$ are predicted values

y_1, y_2, \dots, y_n are observed values

n is the number of observations



Linear Regression

In **Chapter 1**, we looked at a simple regression model of life satisfaction: *life_satisfaction* = $\theta_0 + \theta_1 \times \text{GDP_per_capita}$.

Equation 4-1. Linear Regression model prediction

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

- \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 $\theta_1, \theta_2, \cdots, \theta_n$).

This can be written much more concisely using a vectorized form, as shown in Equation 4-2.

Equation 4-2. Linear Regression model prediction (vectorized form)

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

- θ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n .
- θ^T is the transpose of θ (a row vector instead of a column vector).
- \mathbf{x} is the instance's *feature vector*, containing x_0 to x_n , with x_0 always equal to 1.
- $\theta^T \cdot \mathbf{x}$ is the dot product of θ^T and \mathbf{x} .
- h_{θ} is the hypothesis function, using the model parameters θ .

The MSE of a Linear Regression hypothesis h_{θ} on a training set \mathbf{X} is calculated using Equation 4-3.

Equation 4-3. MSE cost function for a Linear Regression model

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

The Normal Equation

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

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

```
import numpy as np
```

```
X = 2 * np.random.rand(100, 1)
```

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

```
X_b = np.c_[np.ones((100, 1)), X] # add  $x_0 = 1$  to each instance
```

```
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

```
>>> theta_best
```

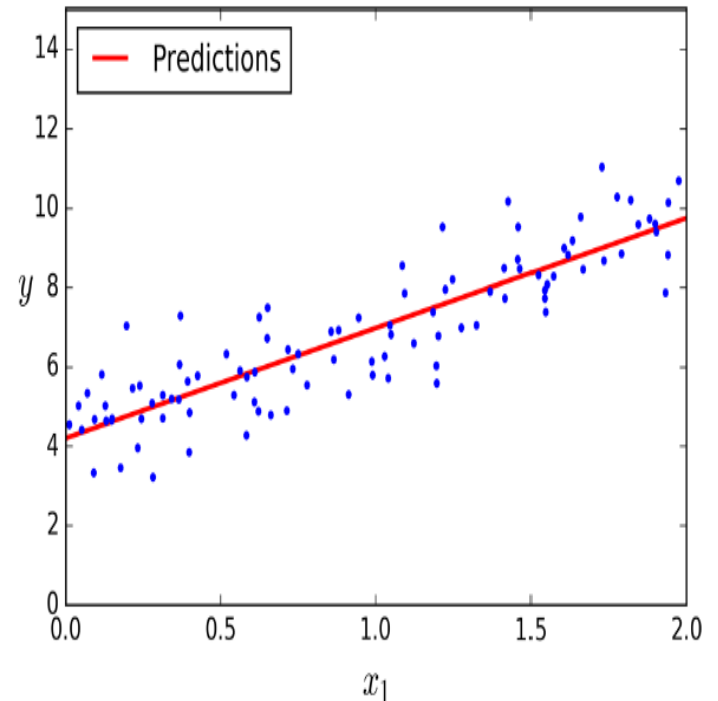
```
array([[ 4.21509616],  
       [ 2.77011339]])
```


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

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

Let's plot this model's predictions (Figure 4-2):

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```



The equivalent code using Scikit-Learn looks like this:³

```
>>> 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]])
```

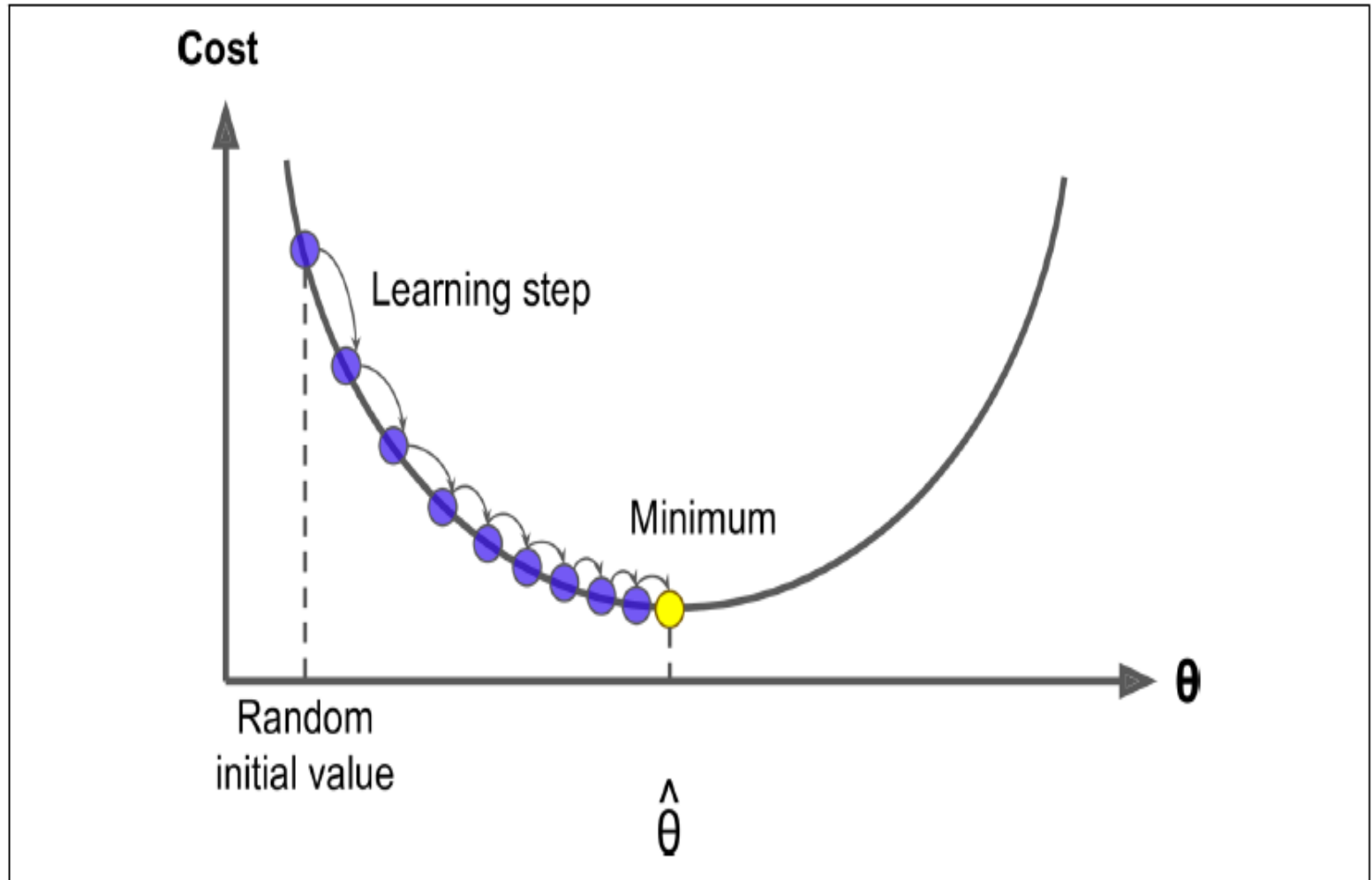
Computational Complexity

- The Normal Equation computes the inverse of $\mathbf{X}^T \cdot \mathbf{X}$, **which is an $n \times n$ matrix** (where n is the number of features). The computational complexity of inverting such a matrix is typically about $O(n^3)$.
- In other words, if you double the number of features, you multiply the computation time by roughly to $2^3 = 8$.

Gradient Descent

- Gradient Descent is a very 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
- Concretely, you start by filling ϑ with random values (*this is called random initialization*), and then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm *converges* to a minimum

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad (\text{for } j = 0 \text{ and } j = 1)$$



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 (see Figure 4-4).

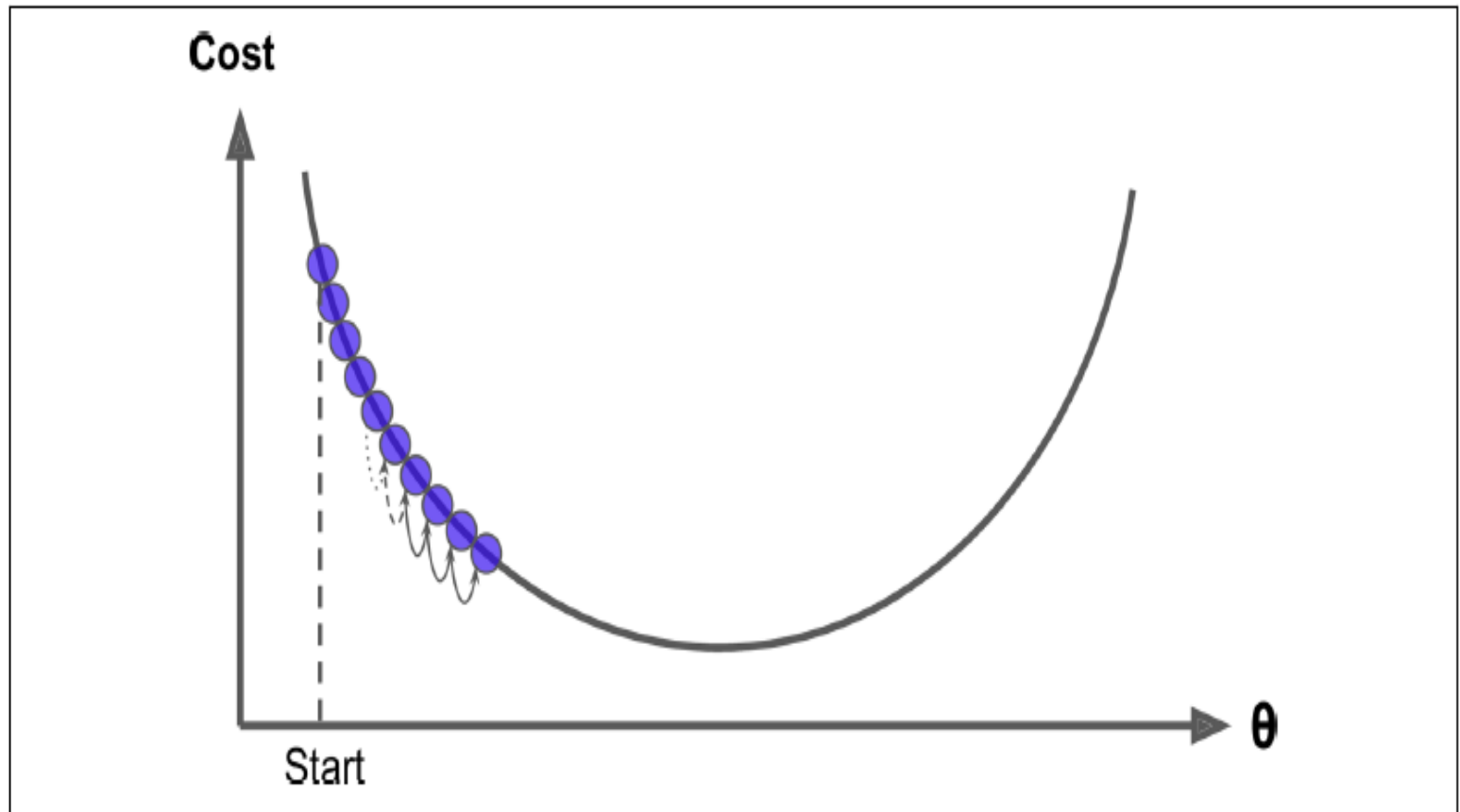


Figure 4-4. Learning rate too small

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 (see [Figure 4-5](#)).

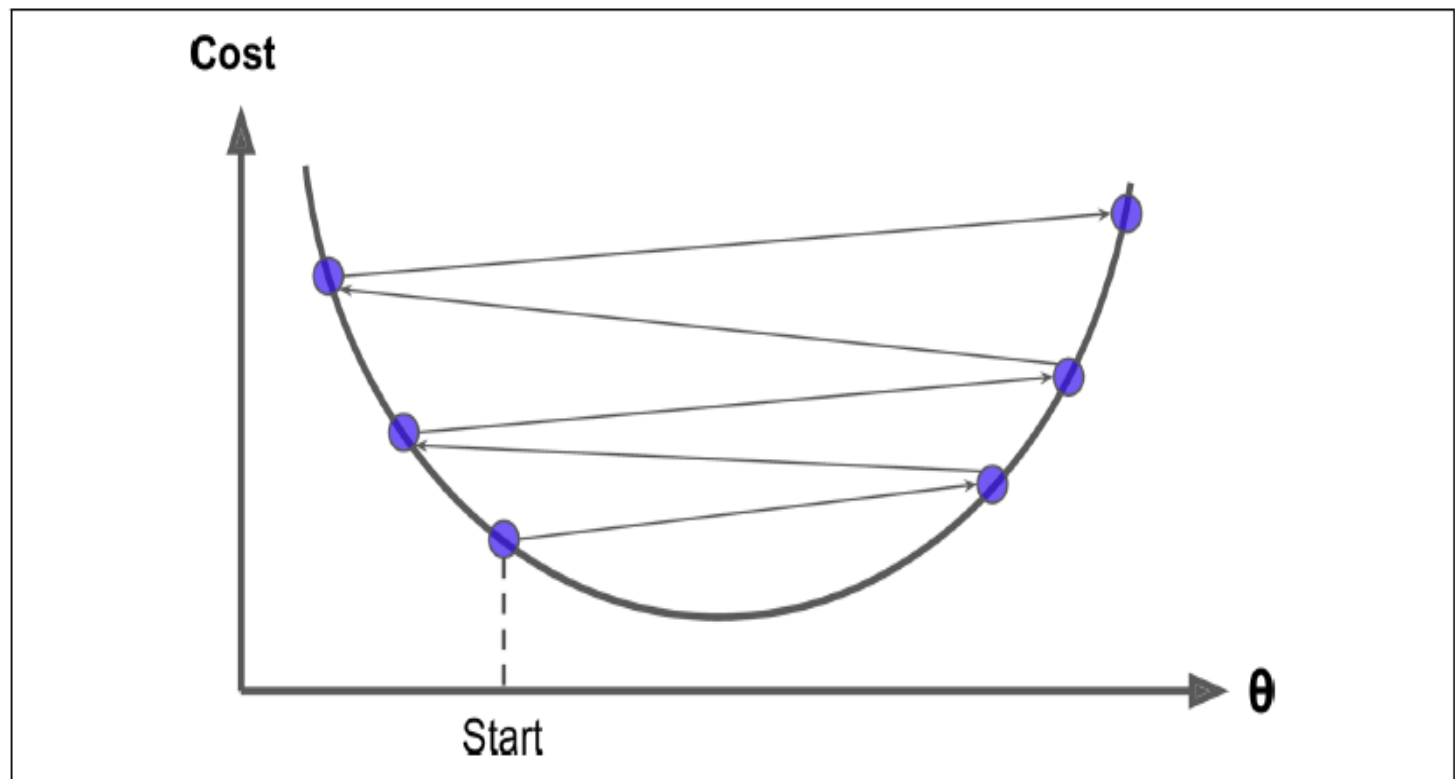


Figure 4-5. Learning rate too large

difficult. Figure 4-6 shows the two main challenges with Gradient Descent: 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.

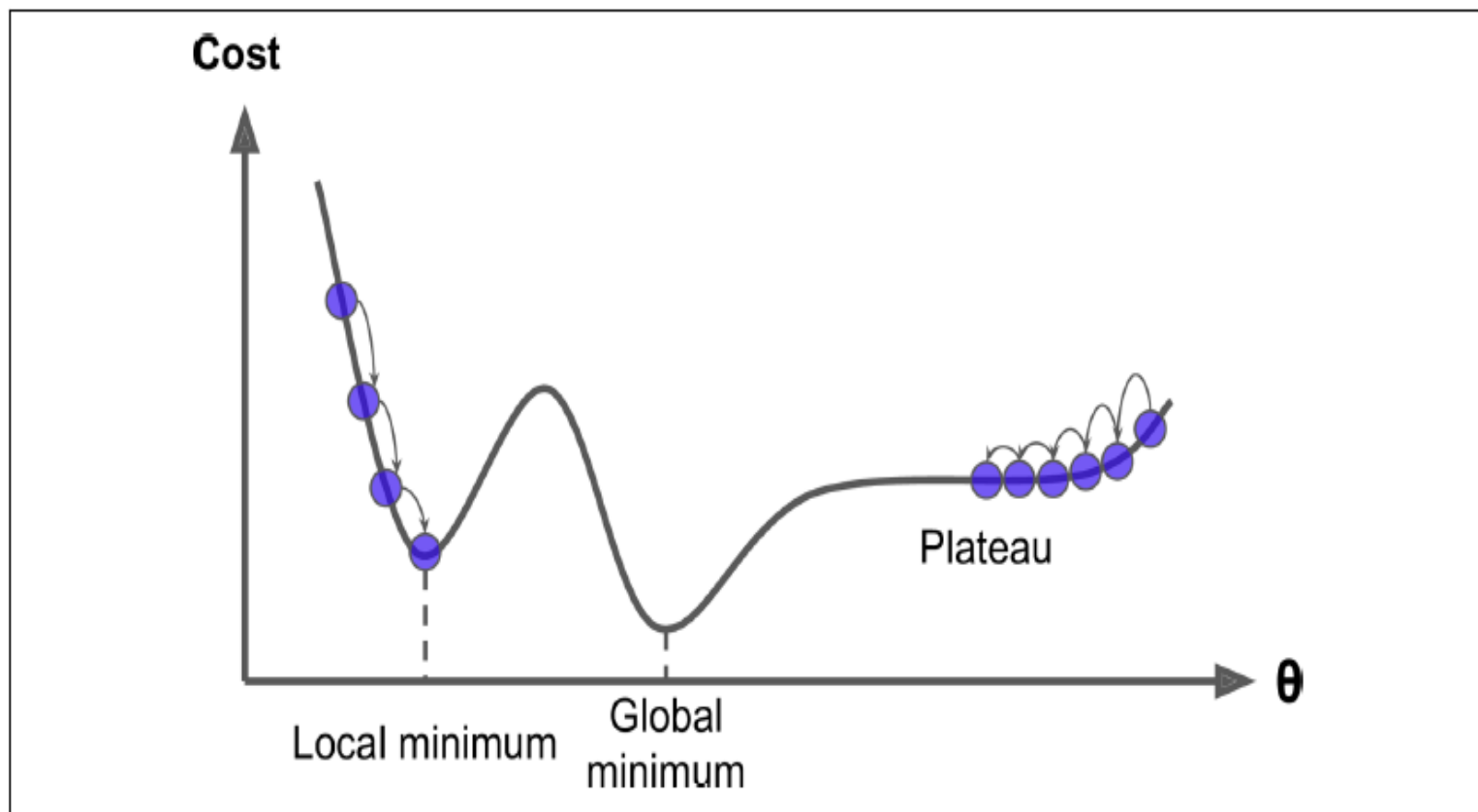


Figure 4-6. Gradient Descent pitfalls

Example

HOUSING DATA	
House Size (X)	House Price (Y)
1,100	1,99,000
1,400	2,45,000
1,425	3,19,000
1,550	2,40,000
1,600	3,12,000
1,700	2,79,000
1,700	3,10,000
1,875	3,08,000
2,350	4,05,000
2,450	3,24,000

Min-Max Standardization	
X (X-Min/Max-min)	Y (Y-Min/Max-Min)
0.00	0.00
0.22	0.22
0.24	0.58
0.33	0.20
0.37	0.55
0.44	0.39
0.44	0.54
0.57	0.53
0.93	1.00
1.00	0.61

Step 1: To fit a line $Y_{pred} = a + bX$, start off with random values of a and b and calculate prediction error (SSE)

a	b	X	Y	YP=a+bX	SSE=1/2(Y-YP)^2
0.45	0.75	0.00	0.00	0.45	0.101
		0.22	0.22	0.62	0.077
		0.24	0.58	0.63	0.001
		0.33	0.20	0.70	0.125
		0.37	0.55	0.73	0.016
		0.44	0.39	0.78	0.078
		0.44	0.54	0.78	0.030
		0.57	0.53	0.88	0.062
		0.93	1.00	1.14	0.010
		1.00	0.61	1.20	0.176
Total					SSE
					0.677

Step 2: Calculate the error gradient w.r.t the weights

$$\partial SSE / \partial a = -(Y - YP)$$

$$\partial SSE / \partial b = -(Y - YP)X$$

$$\text{Here, } SSE = \frac{1}{2} (Y - YP)^2 = \frac{1}{2} (Y - (a + bX))^2$$

You need to know a bit of calculus, but that's about it!!

$\partial SSE / \partial a$ and $\partial SSE / \partial b$ are the **gradients** and they give the direction of the movement of a, b w.r.t to SSE.

a	b	X	Y	YP=a+bX	SSE	$\partial SSE / \partial a$ = -(Y-YP)	$\partial SSE / \partial b$ = -(Y-YP)X
0.45	0.75	0.00	0.00	0.45	0.101	0.45	0.00
		0.22	0.22	0.62	0.077	0.39	0.09
		0.24	0.58	0.63	0.001	0.05	0.01
		0.33	0.20	0.70	0.125	0.50	0.17
		0.37	0.55	0.73	0.016	0.18	0.07
		0.44	0.39	0.78	0.078	0.39	0.18
		0.44	0.54	0.78	0.030	0.24	0.11
		0.57	0.53	0.88	0.062	0.35	0.20
		0.93	1.00	1.14	0.010	0.14	0.13
		1.00	0.61	1.20	0.176	0.59	0.59
Total SSE					0.677	Sum	3.300
							1.545

We need to update the random values of a, b so that we move in the direction of optimal a, b .

Update rules:

- $a - \partial \text{SSE} / \partial a$
- $b - \partial \text{SSE} / \partial b$

So, update rules:

1. New $a = a - r * \partial \text{SSE} / \partial a = 0.45 - 0.01 * 3.300 = 0.42$
2. New $b = b - r * \partial \text{SSE} / \partial b = 0.75 - 0.01 * 1.545 = 0.73$

here, r is the learning rate = 0.01, which is the pace of adjustment to the weights.

Step 4: Use new a and b for prediction and to calculate new Total SSE

a	b	X	Y	YP=a+bX	SSE	∂SSE/∂a	∂SSE/∂b	
0.42	0.73	0.00	0.00	0.42	0.087	0.42	0.00	
		0.22	0.22	0.58	0.064	0.36	0.08	
		0.24	0.58	0.59	0.000	0.01	0.00	
		0.33	0.20	0.66	0.107	0.46	0.15	
		0.37	0.55	0.69	0.010	0.14	0.05	
		0.44	0.39	0.74	0.063	0.36	0.16	
		0.44	0.54	0.74	0.021	0.20	0.09	
		0.57	0.53	0.84	0.048	0.31	0.18	
		0.93	1.00	1.10	0.005	0.10	0.09	
		1.00	0.61	1.15	0.148	0.54	0.54	
Total SSE					0.553	Sum	2.900	1.350

Batch Gradient Descent

Equation 4-5. Partial derivatives of the cost function

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

Equation 4-6. Gradient vector of the cost function

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

Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting $\nabla_{\theta} \text{MSE}(\theta)$ from θ . This is where the learning rate η comes into play:⁶ multiply the gradient vector by η to determine the size of the downhill step (Equation 4-7).

Equation 4-7. Gradient Descent step

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

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

```
eta = 0.1 # learning rate
n_iterations = 1000
m = 100

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

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

```
>>> theta  
array([[ 4.21509616],  
       [ 2.77011339]])
```

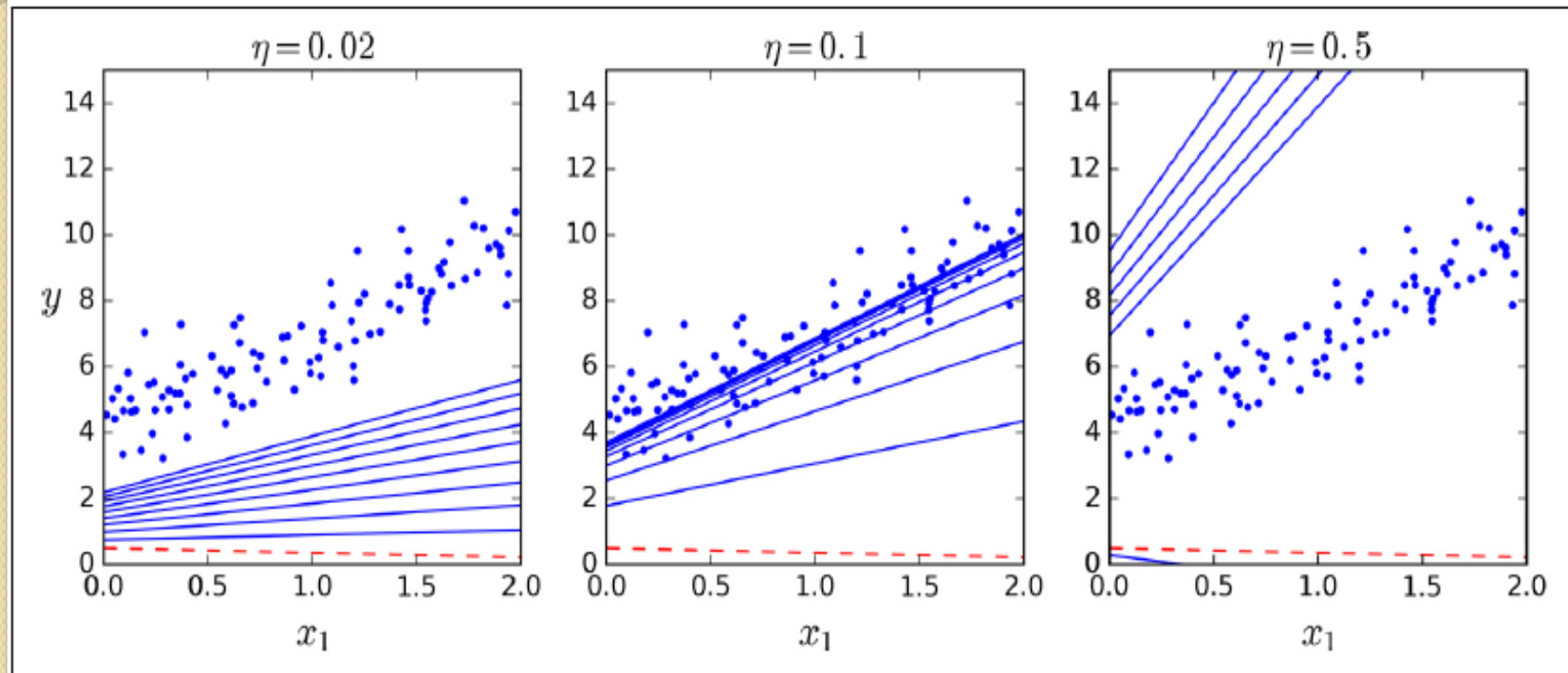


Figure 4-8. Gradient Descent with various learning rates

Stochastic Gradient Descent

- The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large.
- At the opposite extreme, *Stochastic Gradient Descent* just picks a random instance in the training set at every step and computes the gradients based only on that single instance.
- Obviously this makes the algorithm much faster since it has very little data to manipulate at every iteration.

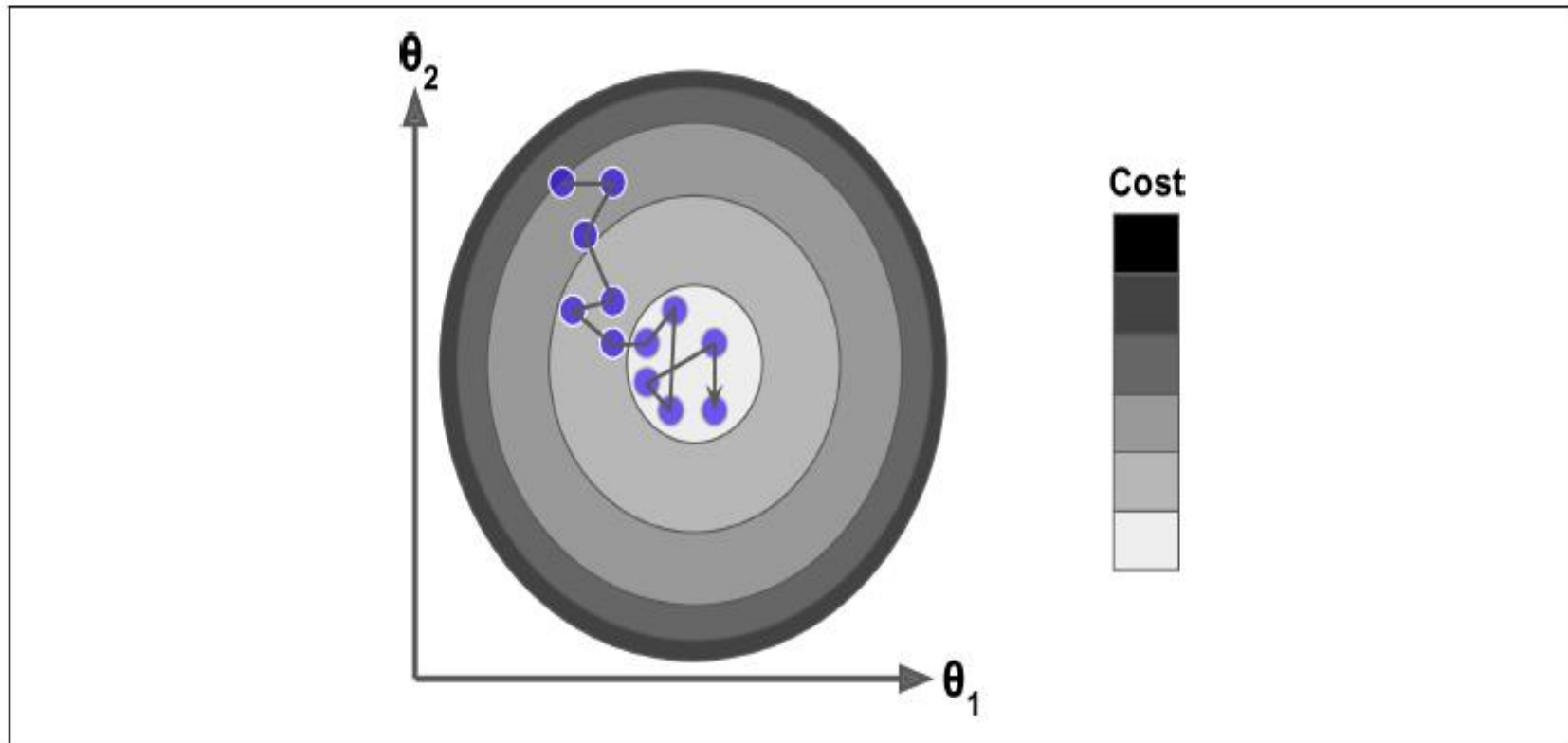


Figure 4-9. Stochastic Gradient Descent

- When the cost function is very irregular this can actually help the algorithm jump out of local minima, so Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.
- Therefore randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum. One solution to this dilemma is to gradually reduce the learning rate. The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum.
- This process is called *simulated annealing*, because it resembles the process of annealing in metallurgy where molten metal is slowly cooled down.
- The function that determines the learning rate at each iteration is called the *learning schedule*. If the learning rate is reduced too quickly, you may get stuck in a local minimum, or even end up frozen halfway to the minimum. If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution if you halt training too early.

This code implements Stochastic Gradient Descent using a simple learning schedule:

```
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

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

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

for epoch in range(n_epochs):
    for i in range(m):
        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  
array([[ 4.21076011],  
       [ 2.74856079]])
```

```
from sklearn.linear_model import SGDRegressor  
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)  
sgd_reg.fit(X, y.ravel())
```

Once again, you find a solution very close to the one returned by the Normal Equation:

```
>>> sgd_reg.intercept_, sgd_reg.coef_  
(array([ 4.18380366]), array([ 2.74205299]))
```

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 *minibatches*.

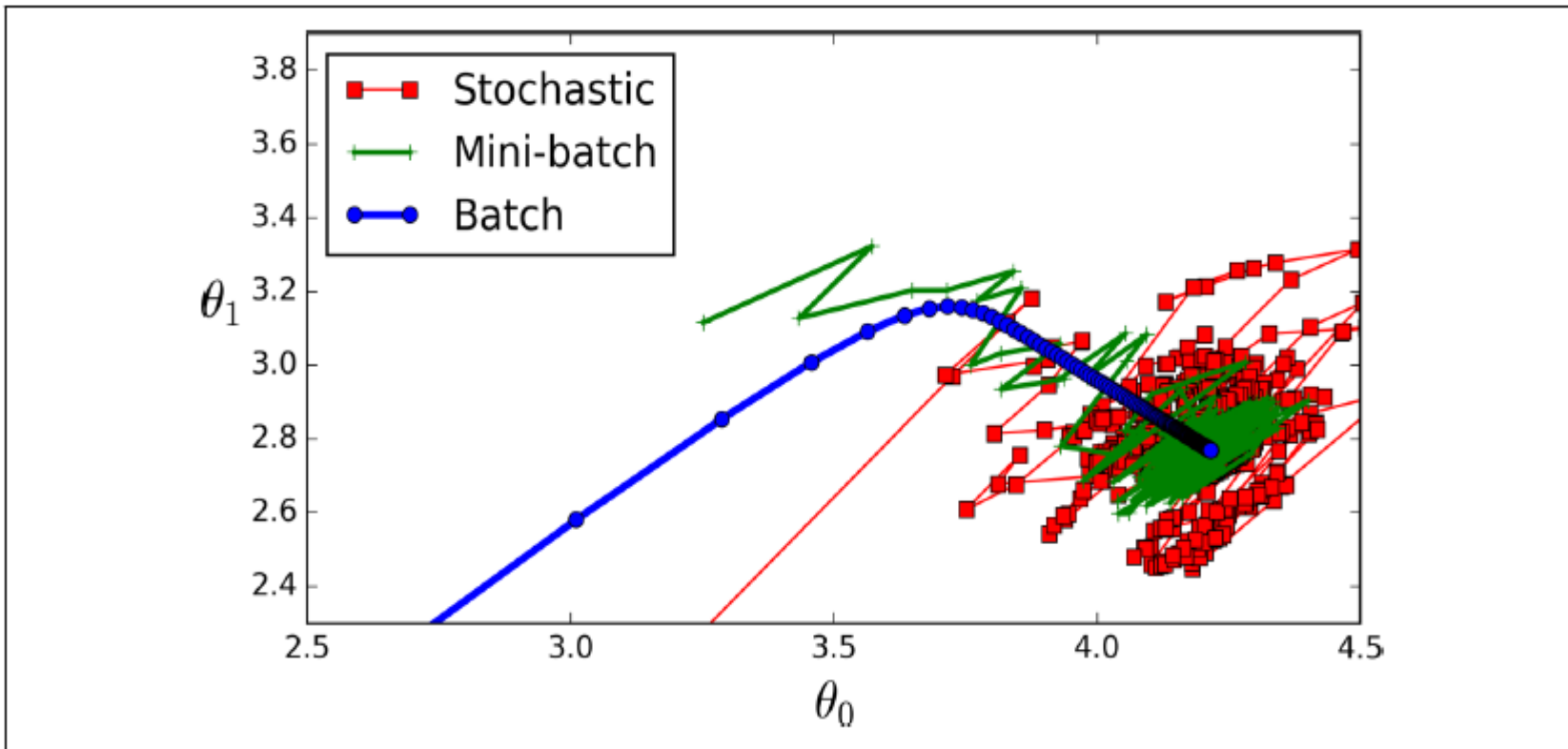


Figure 4-11. Gradient Descent paths in parameter space

Comparison

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large m	Out-of-core support	Large n	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	LinearRegression
Batch GD	Slow	No	Fast	2	Yes	n/a

Algorithm	Large m	Out-of-core support	Large n	Hyperparams	Scaling required	Scikit-Learn
Stochastic GD	Fast	Yes	Fast	≥ 2	Yes	SGDRegressor
Mini-batch GD	Fast	Yes	Fast	≥ 2	Yes	n/a

There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

Polynomial Regression

- 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*.

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

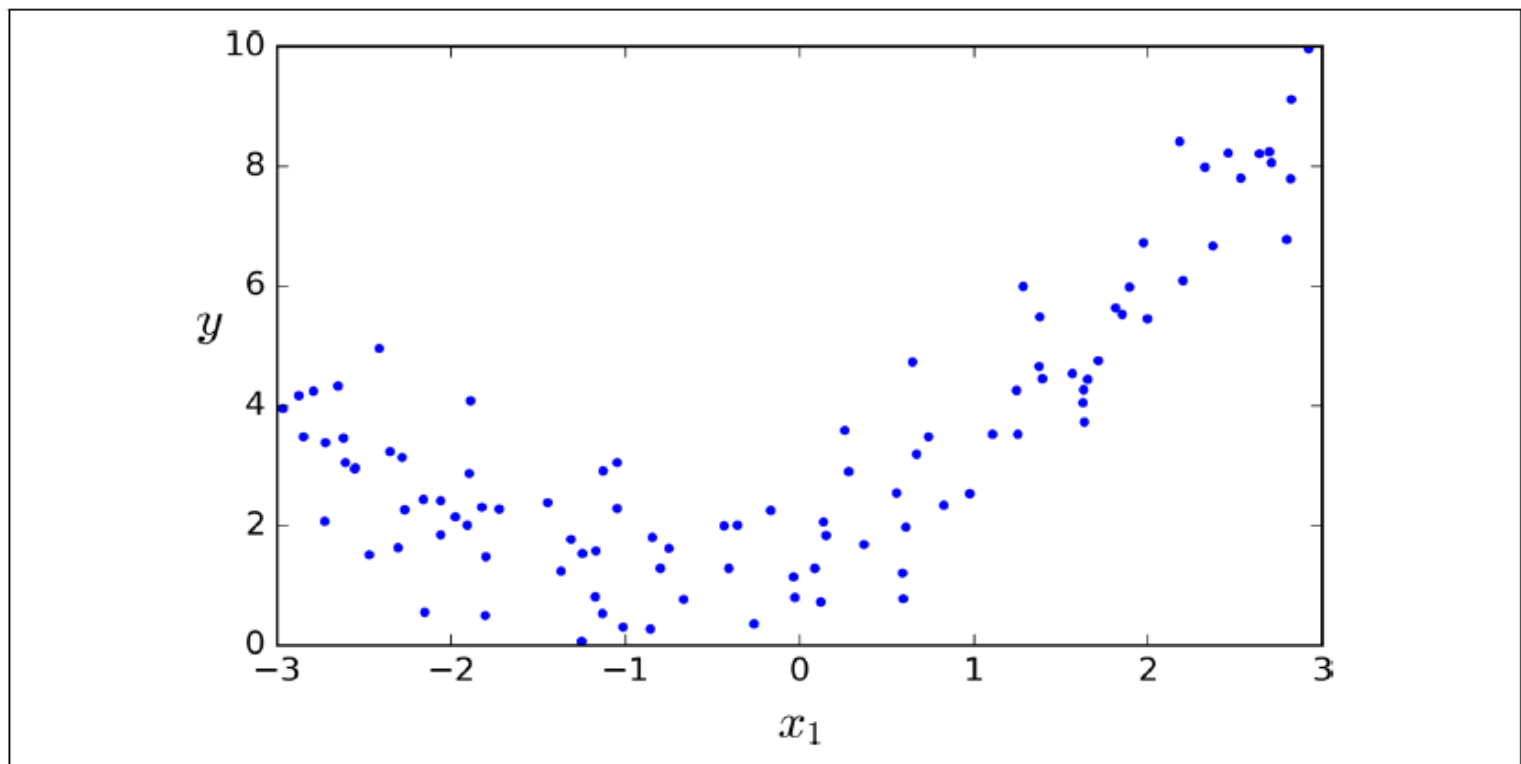


Figure 4-12. Generated nonlinear and noisy dataset


```
>>> 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])

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

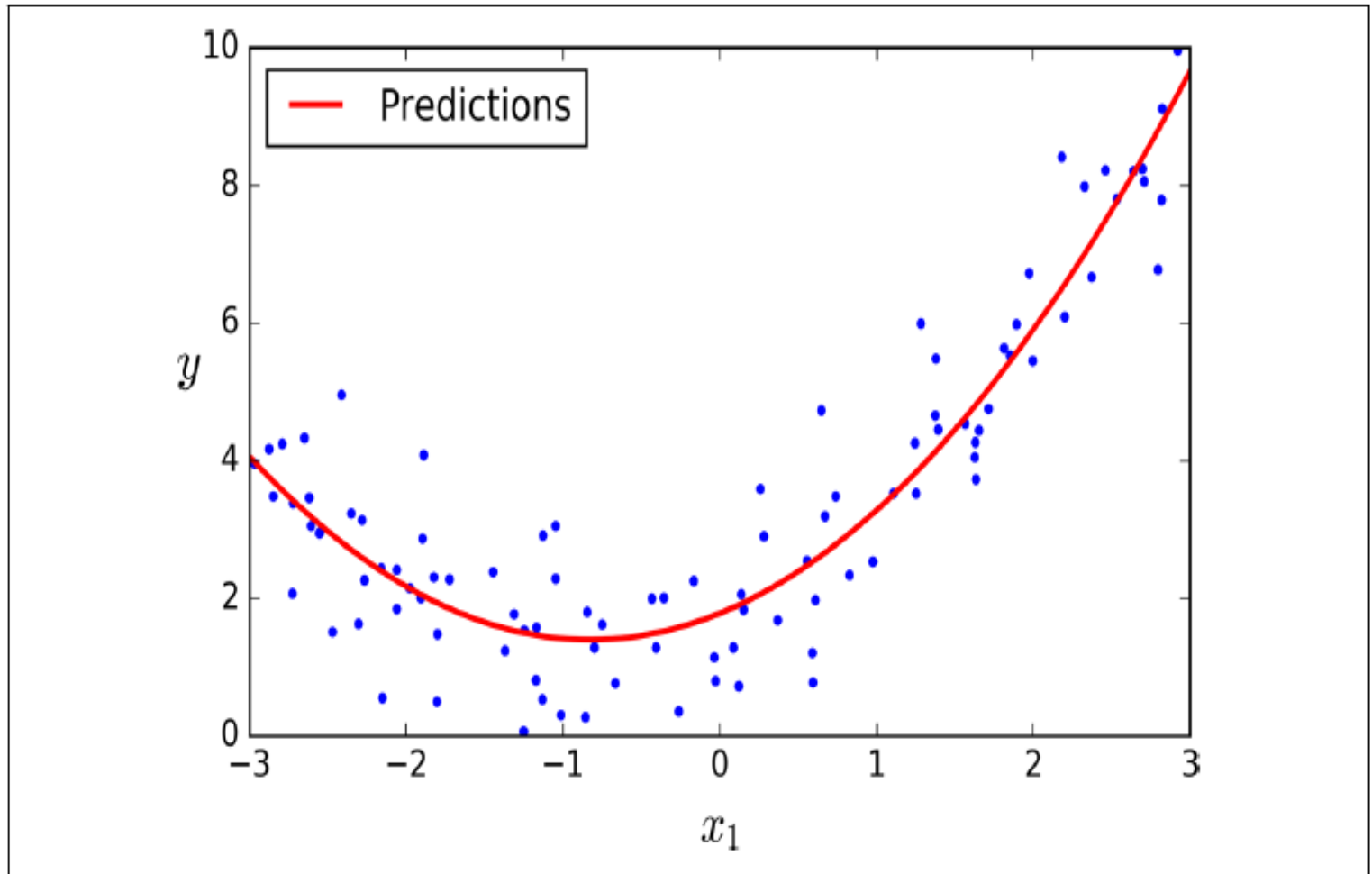
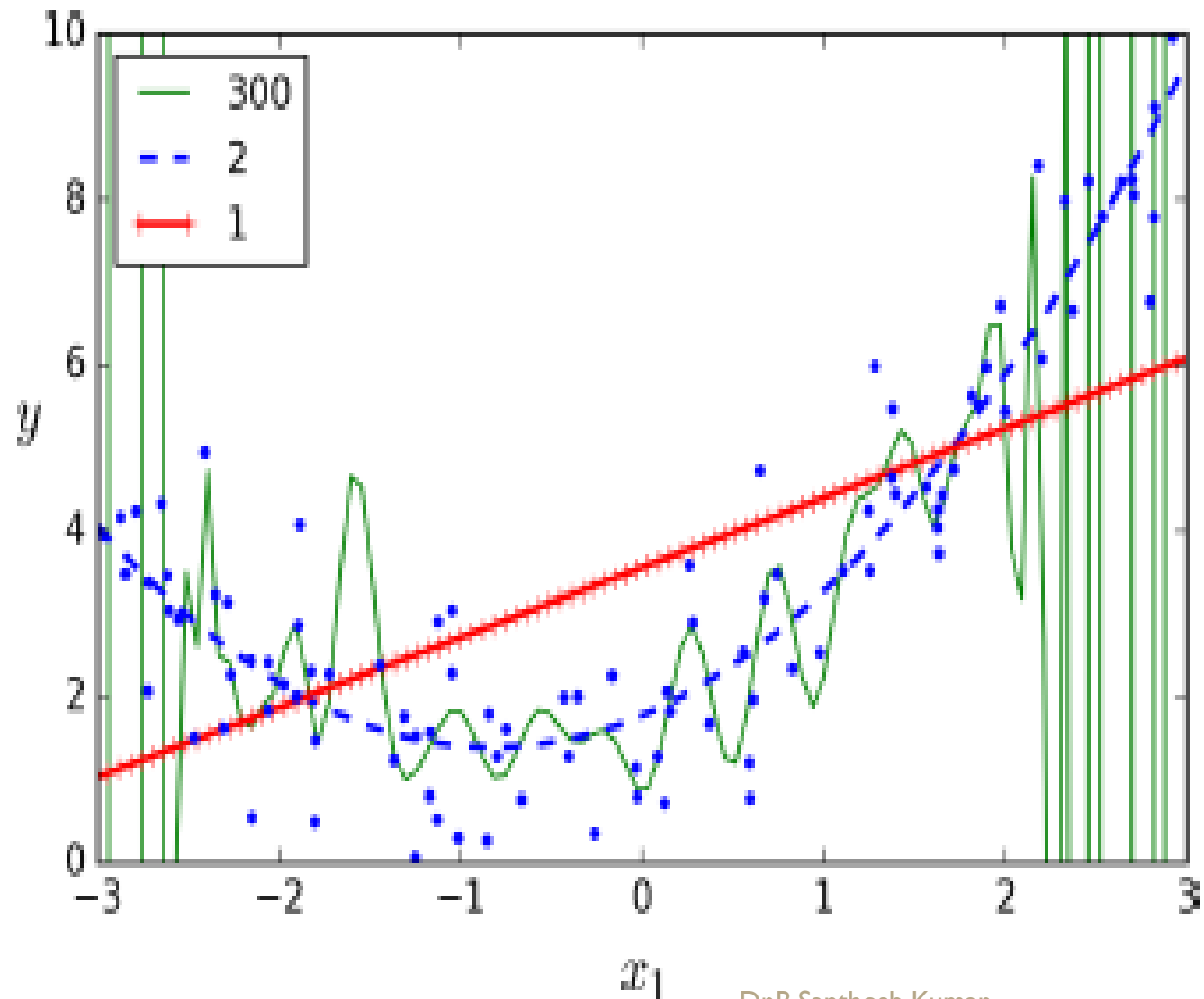


Figure 4-13. Polynomial Regression model predictions

Not bad: the model estimates $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$ when in fact the original function was $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$.

Learning Curves



Learning Curves

```
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_predict, y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_val))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

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

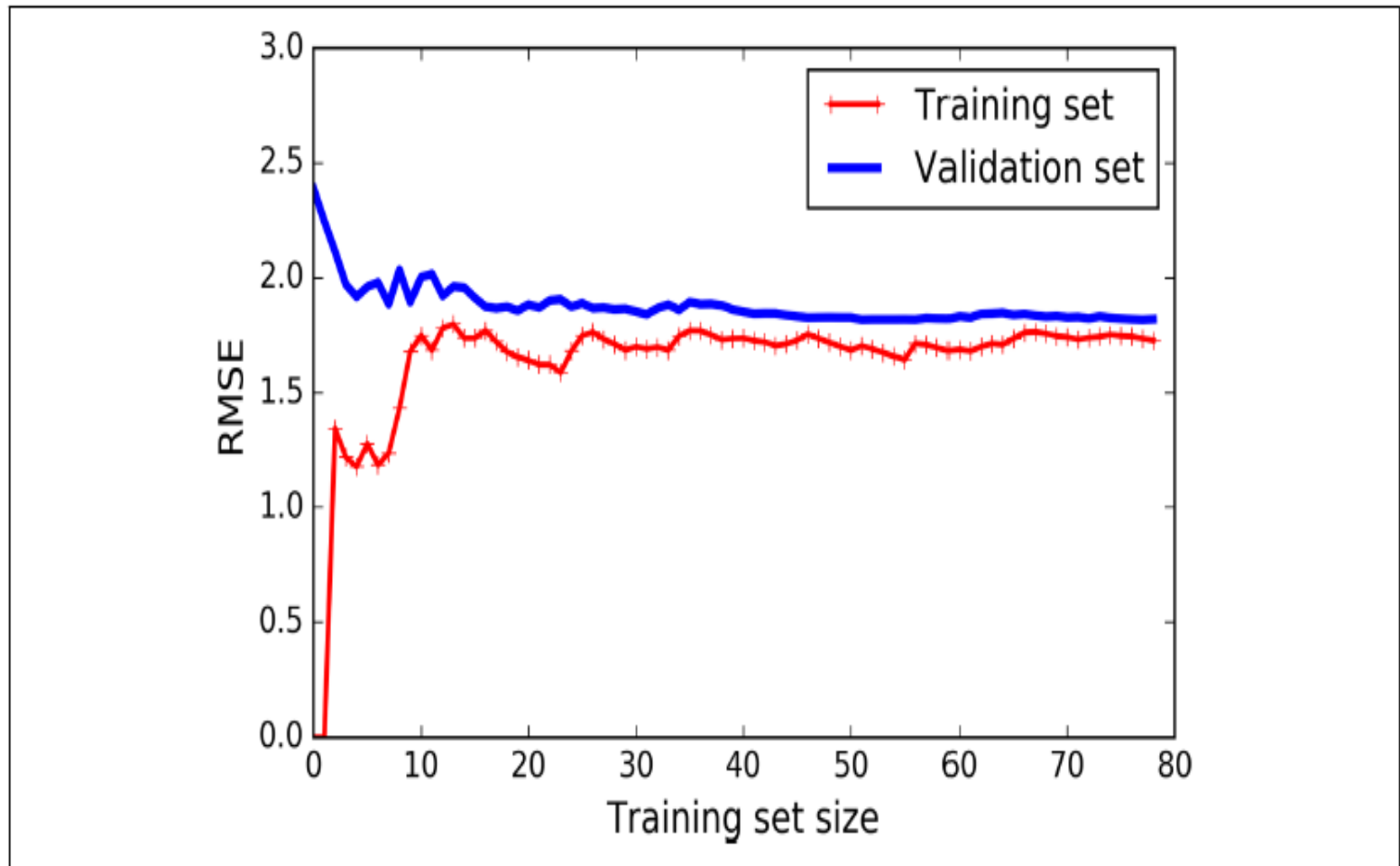


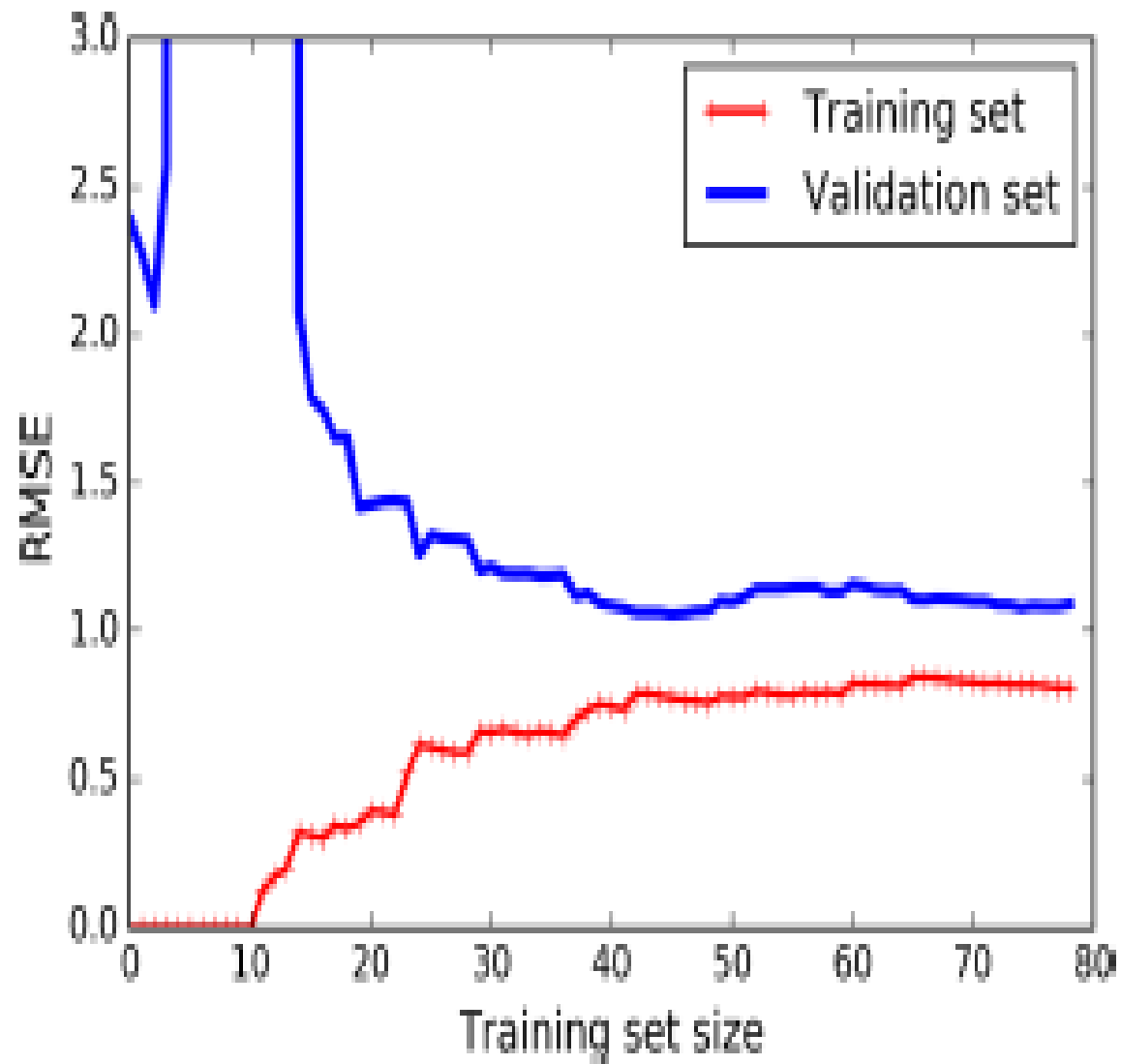
Figure 4-15. Learning curves

Now let's look at the learning curves of a 10th-degree polynomial model on the same data (Figure 4-16):

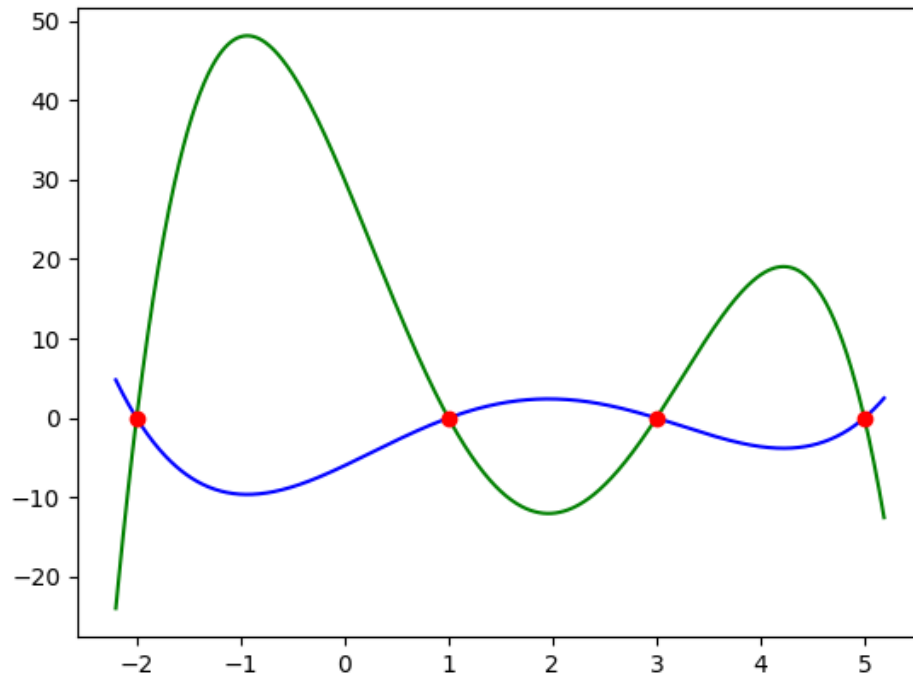
```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline((
    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
    ("sgd_reg", LinearRegression()),
))

plot_learning_curves(polynomial_regression, X, y)
```



Overfitting example



The green curve:

$$h_1(x) = -x^4 + 7x^3 - 5x^2 - 31x + 30$$

The red curve:

$$h_2(x) = \frac{x^4}{5} - \frac{7x^3}{5} + x^2 + \frac{31x}{5} - 6$$

Regularized Linear Models

Ridge Regression

Lasso Regression

Elastic Net

Early Stopping

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad (\text{for } j = 0 \text{ and } j = 1)$$

Ridge Regression

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

Equation 4-8. Ridge Regression cost function

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

Note that the bias term θ_0 is not regularized (the sum starts at $i = 1$, not 0). If we define \mathbf{w} as the vector of feature weights (θ_1 to θ_n), then the regularization term is simply equal to $\frac{1}{2}(\|\mathbf{w}\|_2)^2$, where $\|\cdot\|_2$ represents the ℓ_2 norm of the weight vector.¹² For Gradient Descent, just add $\alpha \mathbf{w}$ to the MSE gradient vector ([Equation 4-6](#)).

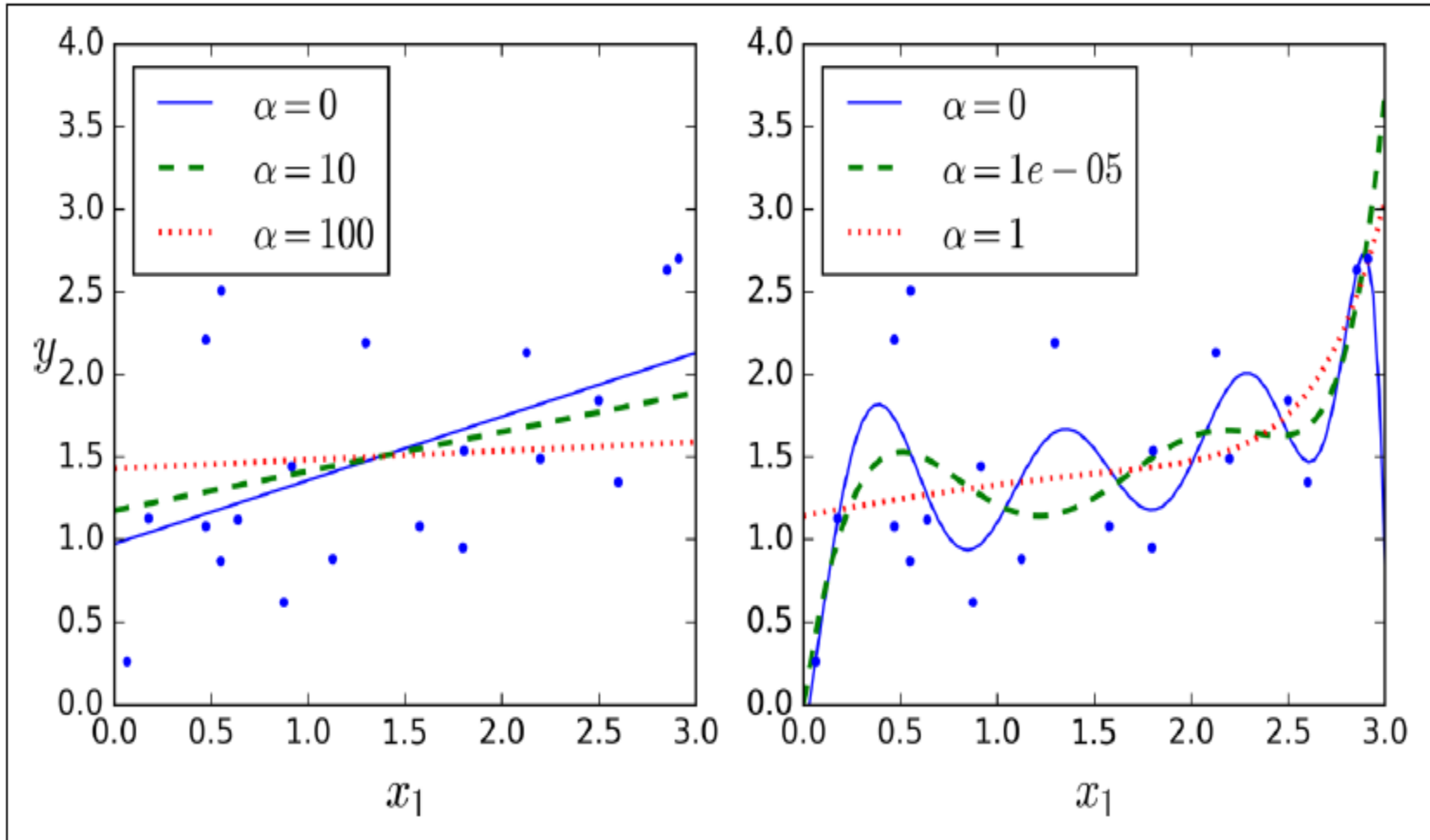


Figure 4-17. Ridge Regression

Equation 4-9. Ridge Regression closed-form solution

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

```
m = 20
X = 3 * np.random.rand(m, 1)
y = 1 + 0.5 * X + np.random.randn(m, 1) / 1.5
```

```
>>> 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]])
```

And using Stochastic Gradient Descent:¹⁴


```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([[ 1.13500145]])
```

Lasso Regression

- *Least Absolute Shrinkage and Selection Operator Regression (simply called Lasso Regression) is another regularized version of Linear Regression: just like Ridge Regression, it adds a regularization term to the cost function*
- It uses the ℓ_1 norm of the weight vector instead of half the square of the ℓ_2 norm

Equation 4-10. Lasso Regression cost function

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



Here is a small Scikit-Learn example using the Lasso class. Note that you could instead use an SGDRegressor(penalty="l1").

```
>>> 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

- Elastic Net is a middle ground between Ridge Regression and Lasso Regression. The regularization term is a simple mix 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

Equation 4-12. Elastic Net cost function

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

Here is a short example using Scikit-Learn's ElasticNet ($l1_ratio$ corresponds to the mix ratio r):

```
>>> 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

- 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. This is called *early stopping*.

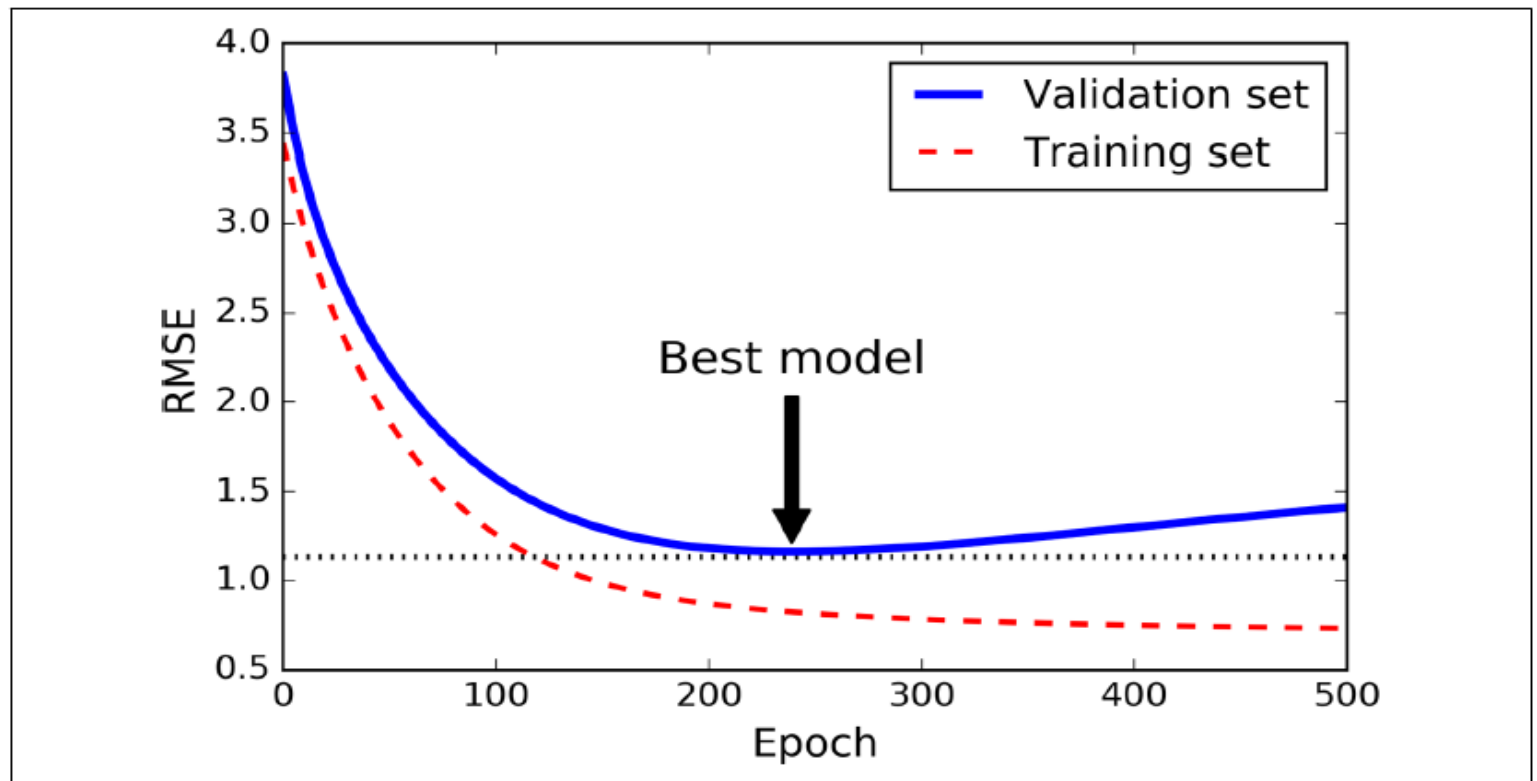


Figure 4-20. Early stopping regularization

Logistic Regression

- *Logistic Regression (also called Logit Regression)* is 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 50%, then the model predicts that the instance belongs to that class (called the positive class, labeled “1”)
- Else it predicts that it does not (i.e., it belongs to the negative class, labeled “0”). This makes it a binary classifier.

Equation 4-13. Logistic Regression model estimated probability (vectorized form)

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

The logistic—also called the *logit*, noted $\sigma(\cdot)$ —is a *sigmoid function* (i.e., S-shaped) that outputs a number between 0 and 1. It is defined as shown in Equation 4-14 and Figure 4-21.

Equation 4-14. Logistic function

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

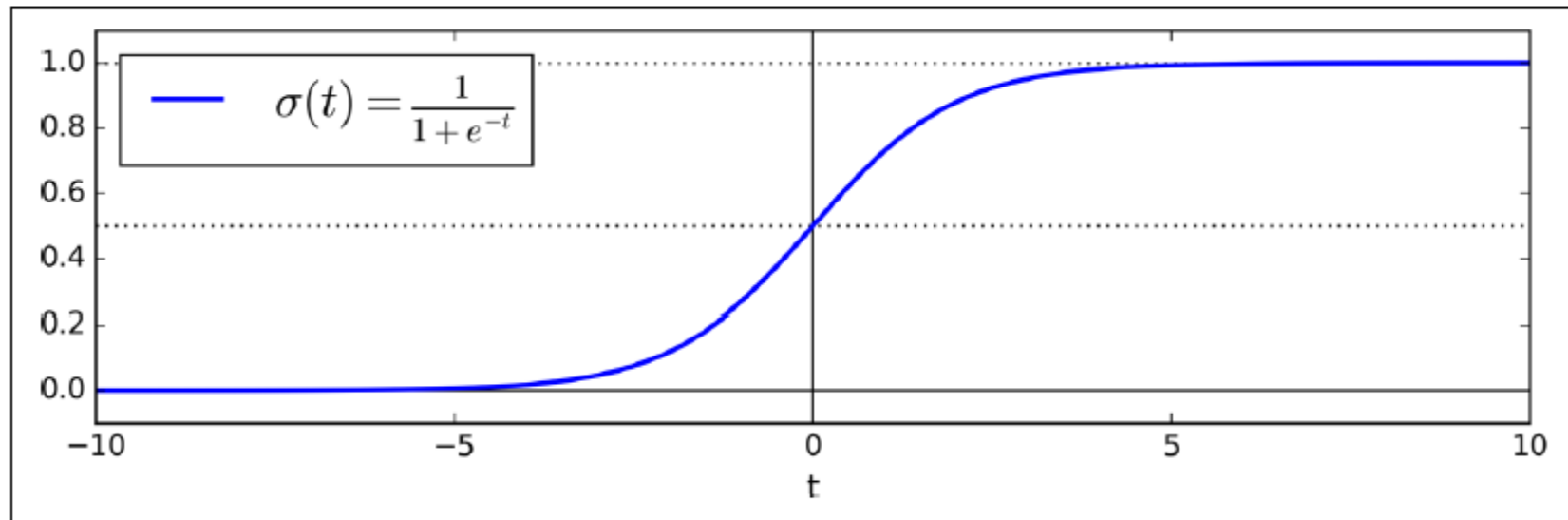


Figure 4-21. Logistic function

Once the Logistic Regression model has estimated the probability $\hat{p} = h_{\theta}(\mathbf{x})$ that an instance \mathbf{x} belongs to the positive class, it can make its prediction \hat{y} easily (see Equation 4-15).

Equation 4-15. Logistic Regression model prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5, \\ 1 & \text{if } \hat{p} \geq 0.5. \end{cases}$$

Notice that $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, so a Logistic Regression model predicts 1 if $\theta^T \cdot \mathbf{x}$ is positive, and 0 if it is negative.

Training and Cost Function

Equation 4-16. Cost function of a single training instance

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

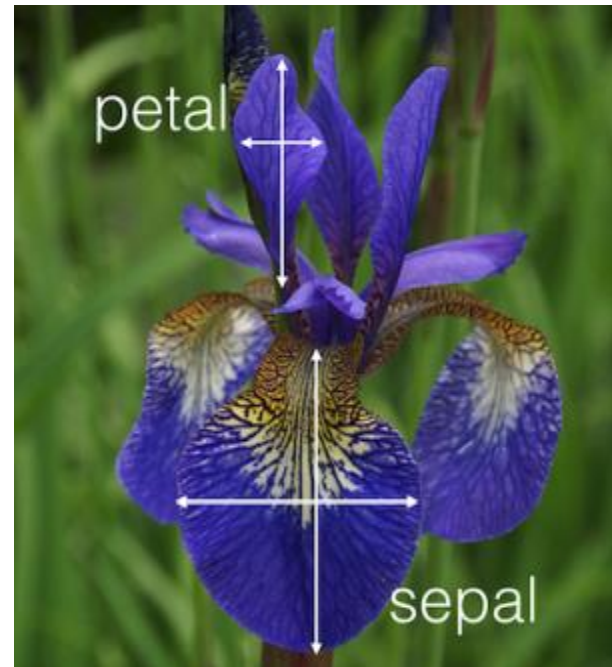
Equation 4-17. Logistic Regression cost function (log loss)

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

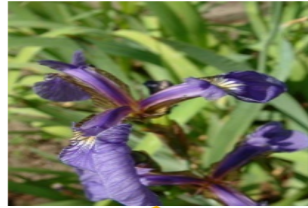
Iris Dataset

■ Attribute Information:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:
 - Iris Setosa
 - Iris Versicolour
 - Iris Virginica



Iris Dataset



	A	B	C	D	E
1	Sepal Length	Sepal Width	Petal Length	Petal Width	Class
2	5.1	3.5	1.4	0.2	Iris-setosa
3	4.9	3	1.4	0.2	Iris-setosa
4	4.7	3.2	1.3	0.2	Iris-setosa
5	4.6	3.1	1.5	0.2	Iris-setosa
6	5	3.6	1.4	0.2	Iris-setosa
7	5.4	3.9	1.7	0.4	Iris-setosa
8	4.6	3.4	1.4	0.3	Iris-setosa
9	5	3.4	1.5	0.2	Iris-setosa
10	4.4	2.9	1.4	0.2	Iris-setosa
11	4.9	3.1	1.5	0.1	Iris-setosa
12	5.4	3.7	1.5	0.2	Iris-setosa
13	4.8	3.4	1.6	0.2	Iris-setosa
14	4.8	3	1.4	0.1	Iris-setosa
15	4.3	3	1.1	0.1	Iris-setosa
16	5.8	4	1.2	0.2	Iris-setosa
17	5.7	4.4	1.5	0.4	Iris-setosa
18	5.4	3.9	1.3	0.4	Iris-setosa
19	5.1	3.5	1.4	0.3	Iris-setosa
20	5.7	3.8	1.7	0.3	Iris-setosa
21	5.1	3.8	1.5	0.3	Iris-setosa
22	5.4	3.4	1.7	0.2	Iris-setosa
23	5.1	3.7	1.5	0.4	Iris-setosa
24	4.6	3.6	1	0.2	Iris-setosa
25	5.1	3.3	1.7	0.5	Iris-setosa

Decision Boundaries

Let's try to build a classifier to detect the Iris-Virginica type based only on the petal width feature. First let's load the data:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
>>> X = iris["data"][:, 3:] # petal width
>>> y = (iris["target"] == 2).astype(np.int) # 1 if Iris-Virginica, else 0
```

Now let's train a Logistic Regression model:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```


Let's look at the model's estimated probabilities for flowers with petal widths varying from 0 to 3 cm (Figure 4-23):

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica")
```

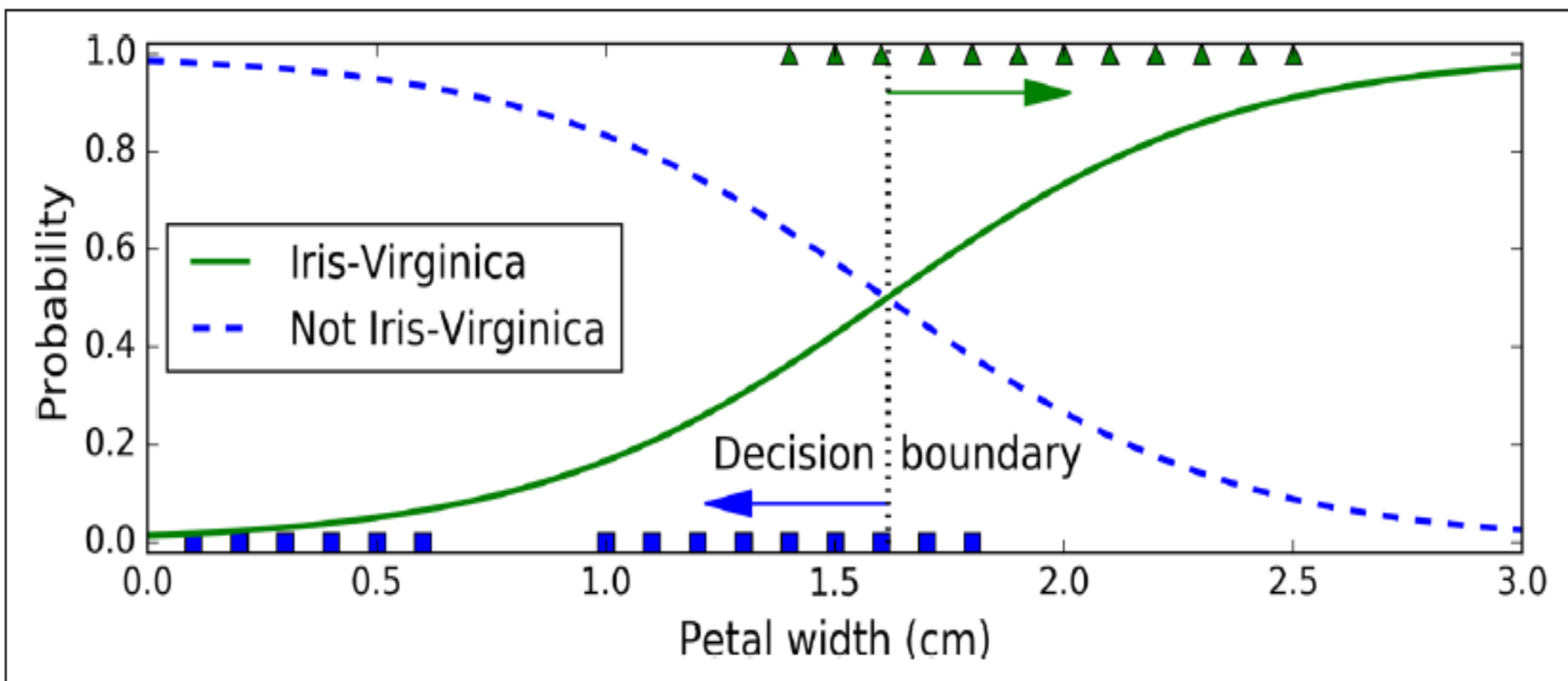


Figure 4-23. Estimated probabilities and decision boundary

```
>>> log_reg.predict([[1.7], [1.5]])  
array([1, 0])
```

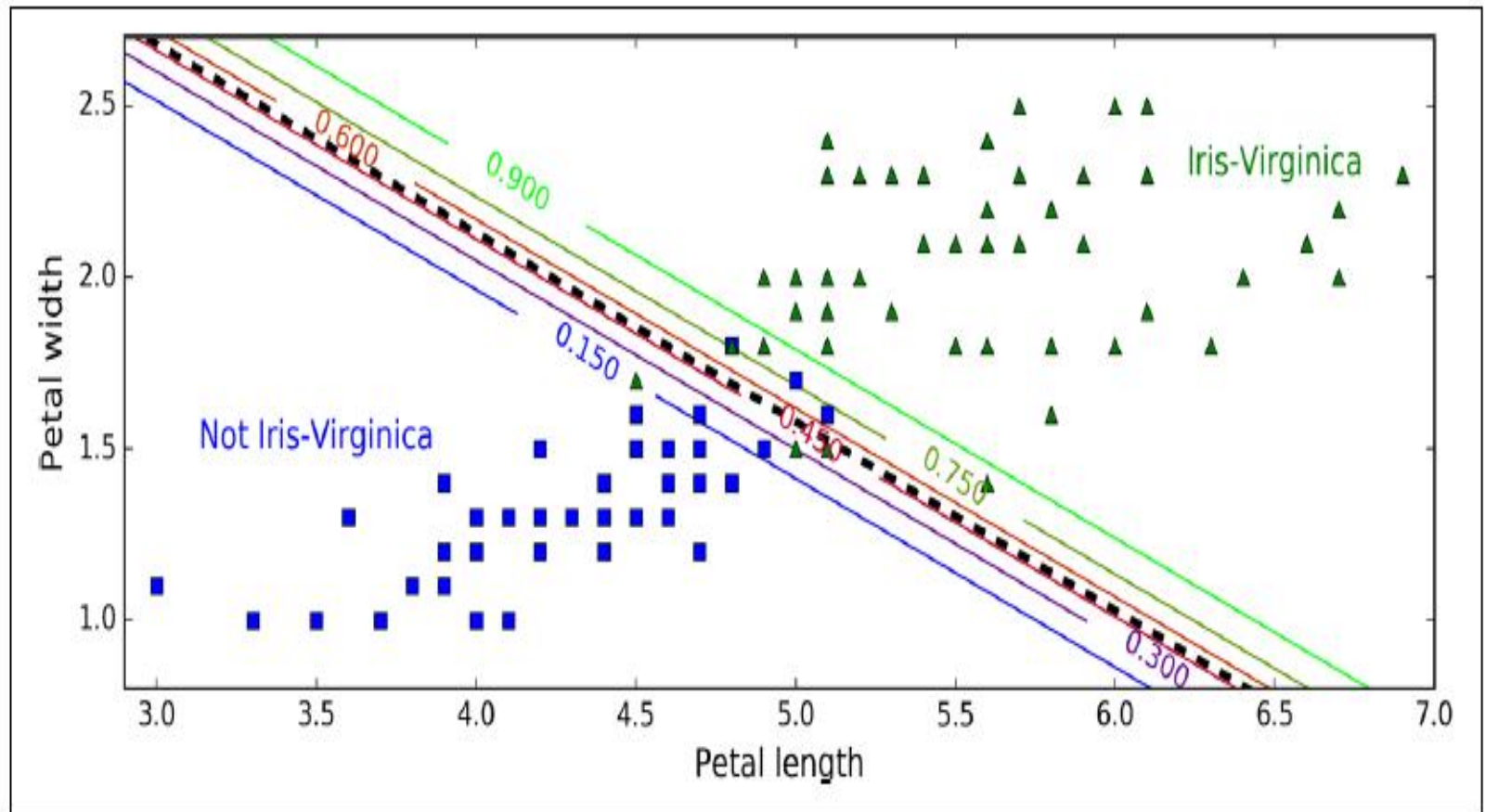


Figure 4-24. Linear decision boundary

Softmax Regression

- The Logistic Regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers.
- This is called *Softmax Regression*, or *Multinomial Logistic Regression*.
- The idea is quite simple: when given an instance x , the **Softmax Regression model** first computes a score $s_k(x)$ for each class k , then estimates the probability of each class by applying the softmax function (also called the normalized exponential) to the scores.
- The equation to compute $s_k(x)$ **should look familiar, as it is just like the equation** for Linear Regression prediction .

Equation 4-19. Softmax score for class k

$$s_k(\mathbf{x}) = \theta_k^T \cdot \mathbf{x}$$

Note that each class has its own dedicated parameter vector ϑ_k . All these vectors are typically stored as rows in a *parameter matrix* Θ .

Once you have computed the score of every class for the instance \mathbf{x} , you can estimate the probability p_k that the instance belongs to class k by running the scores through the softmax function. It computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials).

Equation 4-20. Softmax function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

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

Just like the Logistic Regression classifier, the Softmax Regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score), as shown in Equation 4-21.

Equation 4-21. Softmax Regression classifier prediction

$$\hat{y} = \operatorname{argmax}_k \sigma(\mathbf{s}(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x}) = \operatorname{argmax}_k (\theta_k^T \cdot \mathbf{x})$$

- The *argmax* operator returns the value of a variable that maximizes a function. In this equation, it returns the value of k that maximizes the estimated probability $\sigma(\mathbf{s}(\mathbf{x}))_k$.

```
X = iris["data"][:, (2, 3)] # petal length, petal width
y = iris["target"]
```

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

So the next time you find an iris with 5 cm long and 2 cm wide petals, you can ask your model to tell you what type of iris it is, and it will answer Iris-Virginica (class 2) with 94.2% probability (or Iris-Versicolor with 5.8% probability):

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[ 6.33134078e-07,  5.75276067e-02,  9.42471760e-01]])
```

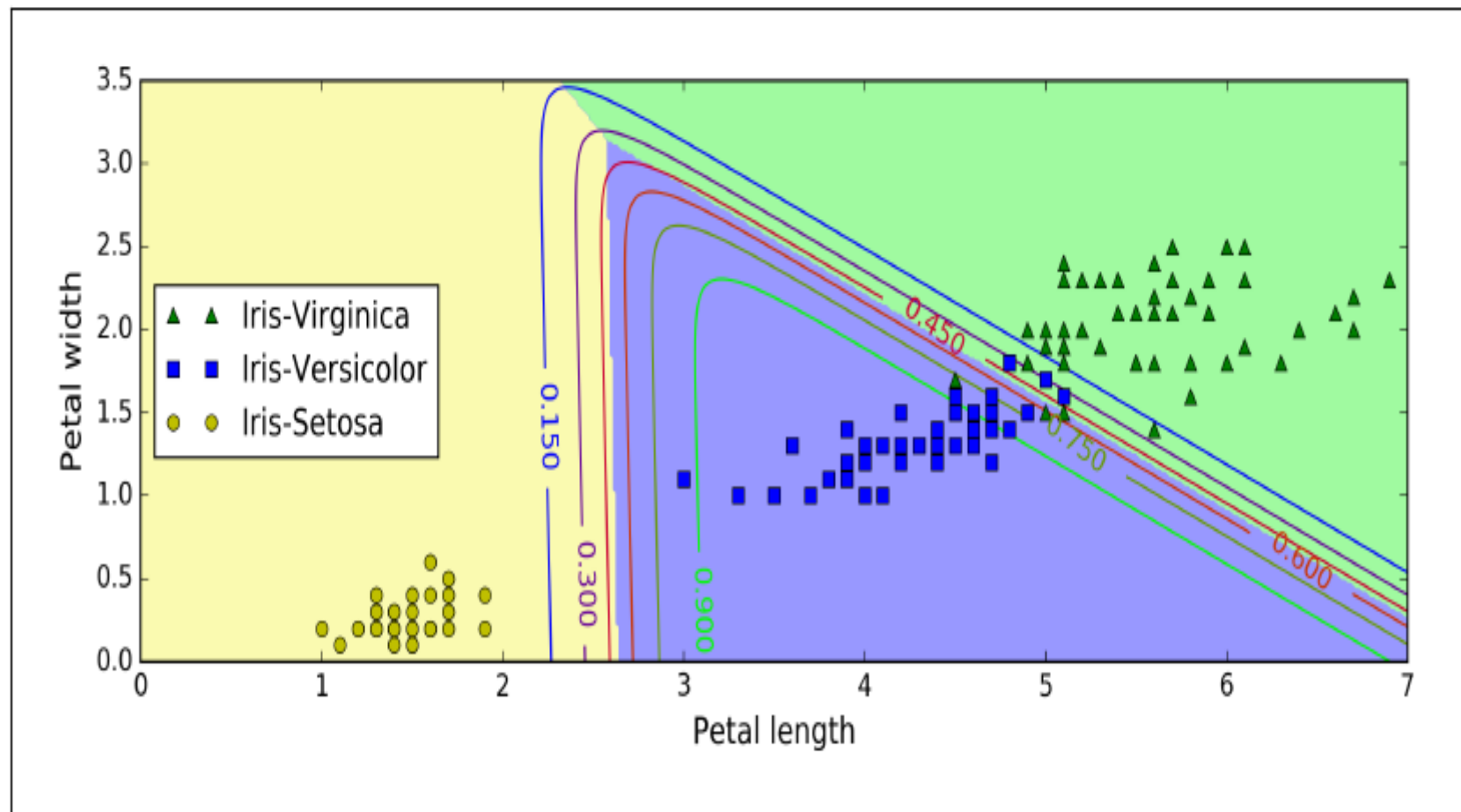



Figure 4-25. Softmax Regression decision boundaries

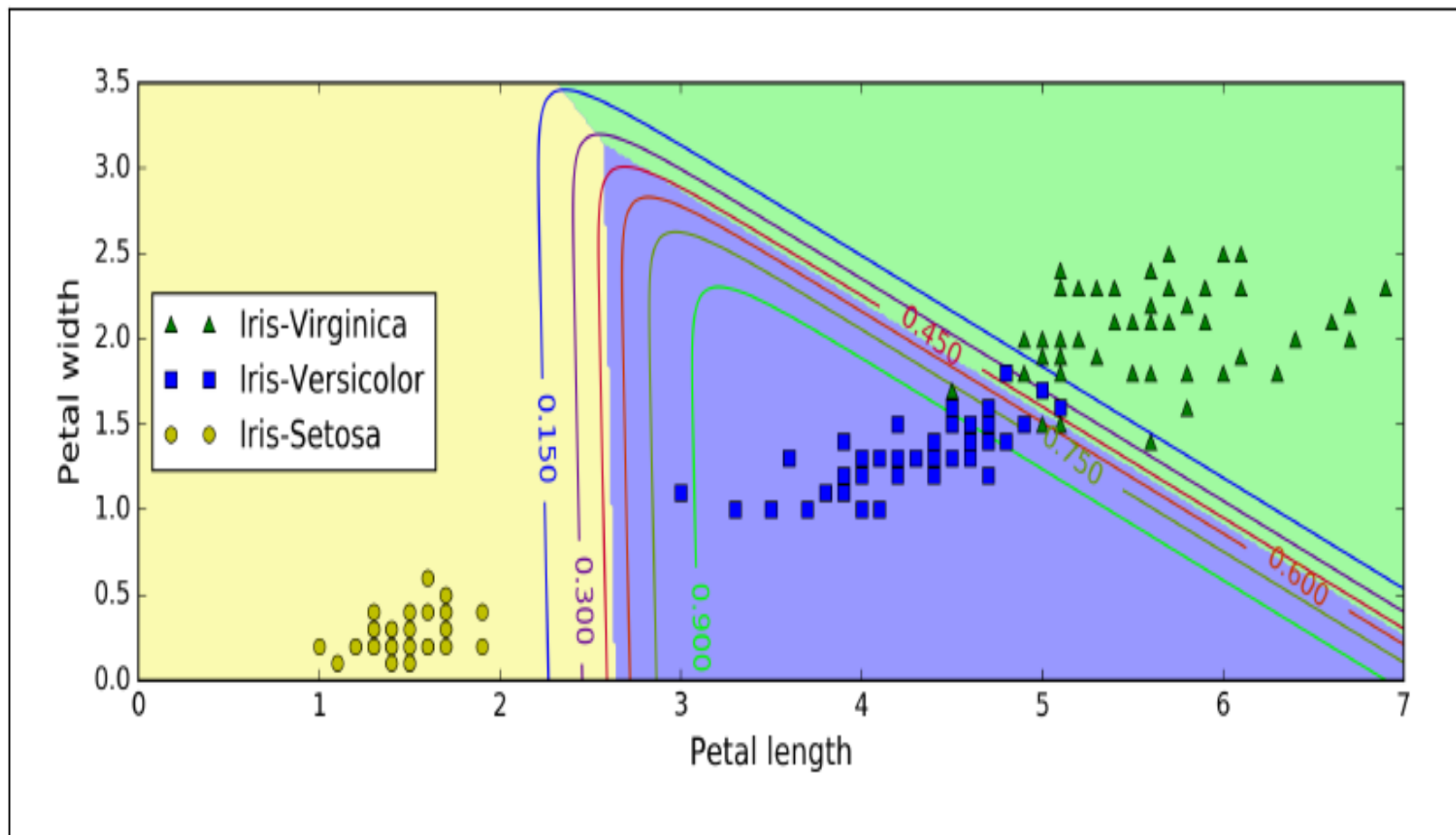


Figure 4-25. Softmax Regression decision boundaries

Support Vector Machines

Linear SVM Classification:

You can think of an SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes. This is called *large margin classification*.

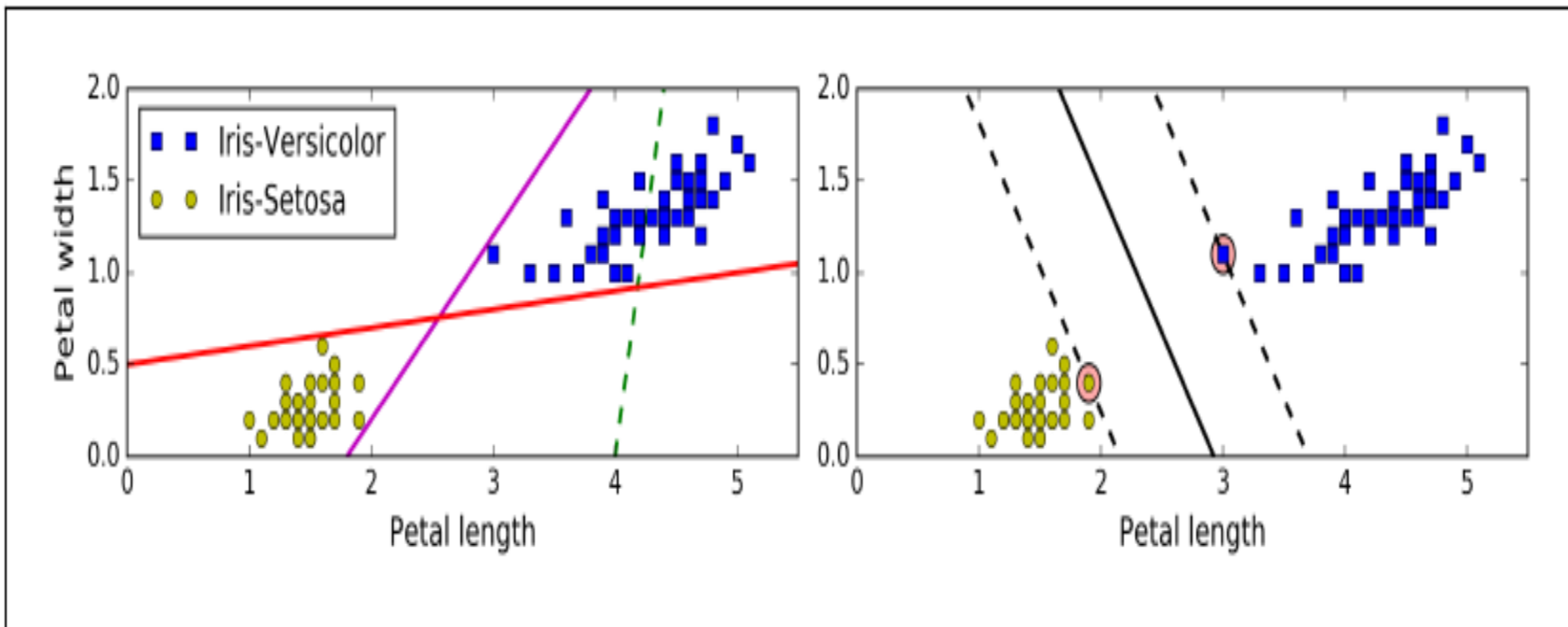


Figure 5-1. Large margin classification

Soft Margin Classification

- If we strictly impose that all instances be off the street and on the right side, this is called *hard margin classification*.
- There are two main issues with hard margin classification. First, it only works if the data is linearly separable, and second it is quite sensitive to outliers.
- Figure 5-3 shows the iris dataset with just one additional outlier: on the left, it is impossible to find a hard margin, and on the right the decision boundary ends up very different from the one we saw in Figure 5-1 without the outlier, and it will probably not generalize as well.

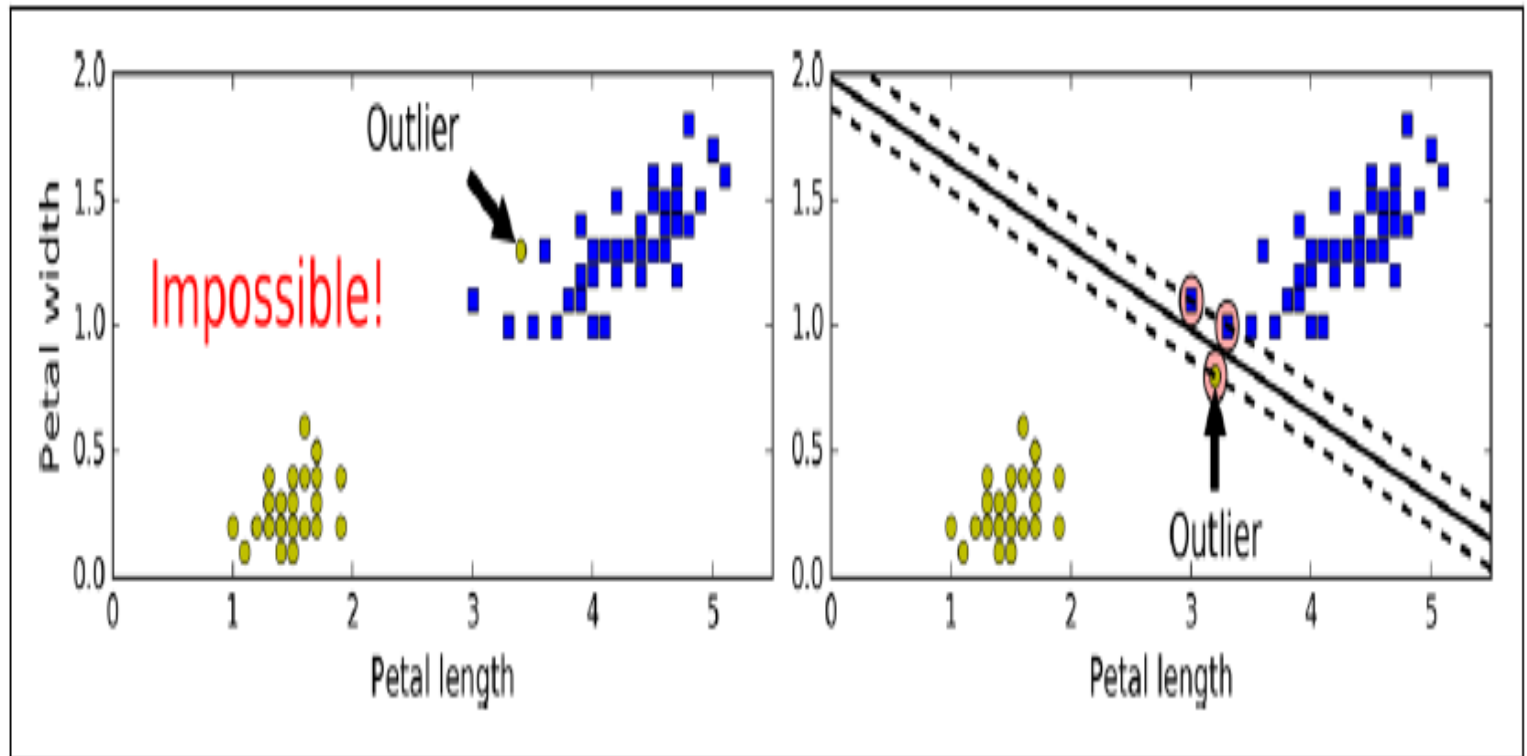


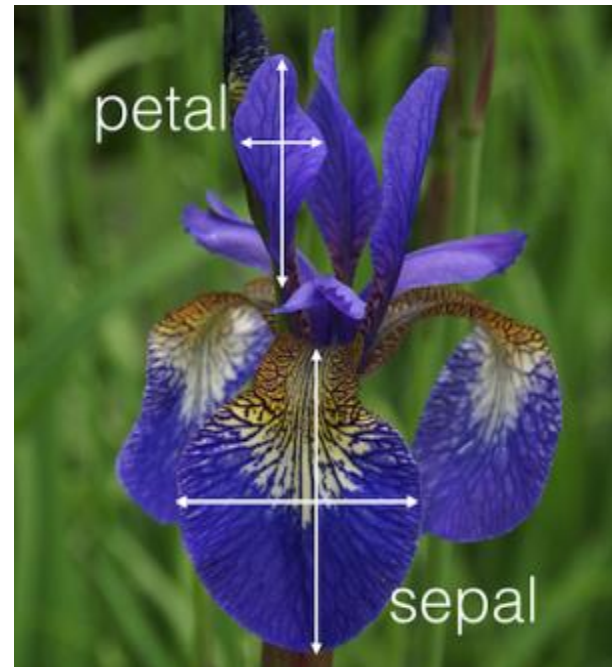
Figure 5-3. Hard margin sensitivity to outliers

- To avoid these issues it is preferable to use a more flexible model. The objective is to find a good balance between keeping the street as large as possible and limiting the *margin violations* (i.e., *instances that end up in the middle of the street or even on the wrong side*). This is called *soft margin classification*.
- In Scikit-Learn's SVM classes, you can control this balance using the C hyperparameter: a smaller C value leads to a wider street but more margin violations. Figure 5-4 shows the decision boundaries and margins of two soft margin SVM classifiers on a nonlinearly separable dataset.
- On the left, using a high C value the classifier makes fewer margin violations but ends up with a smaller margin. On the right, using a low C value the margin is much larger, but many instances end up on the street.
- However, it seems likely that the second classifier will generalize better: in fact even on this training set it makes fewer prediction errors, since most of the margin violations are actually on the correct side of the decision boundary.

Iris Dataset

■ Attribute Information:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:
 - Iris Setosa
 - Iris Versicolour
 - Iris Virginica



Iris Dataset



	A	B	C	D	E
1	Sepal Length	Sepal Width	Petal Length	Petal Width	Class
2	5.1	3.5	1.4	0.2	Iris-setosa
3	4.9	3	1.4	0.2	Iris-setosa
4	4.7	3.2	1.3	0.2	Iris-setosa
5	4.6	3.1	1.5	0.2	Iris-setosa
6	5	3.6	1.4	0.2	Iris-setosa
7	5.4	3.9	1.7	0.4	Iris-setosa
8	4.6	3.4	1.4	0.3	Iris-setosa
9	5	3.4	1.5	0.2	Iris-setosa
10	4.4	2.9	1.4	0.2	Iris-setosa
11	4.9	3.1	1.5	0.1	Iris-setosa
12	5.4	3.7	1.5	0.2	Iris-setosa
13	4.8	3.4	1.6	0.2	Iris-setosa
14	4.8	3	1.4	0.1	Iris-setosa
15	4.3	3	1.1	0.1	Iris-setosa
16	5.8	4	1.2	0.2	Iris-setosa
17	5.7	4.4	1.5	0.4	Iris-setosa
18	5.4	3.9	1.3	0.4	Iris-setosa
19	5.1	3.5	1.4	0.3	Iris-setosa
20	5.7	3.8	1.7	0.3	Iris-setosa
21	5.1	3.8	1.5	0.3	Iris-setosa
22	5.4	3.4	1.7	0.2	Iris-setosa
23	5.1	3.7	1.5	0.4	Iris-setosa
24	4.6	3.6	1	0.2	Iris-setosa
25	5.1	3.3	1.7	0.5	Iris-setosa

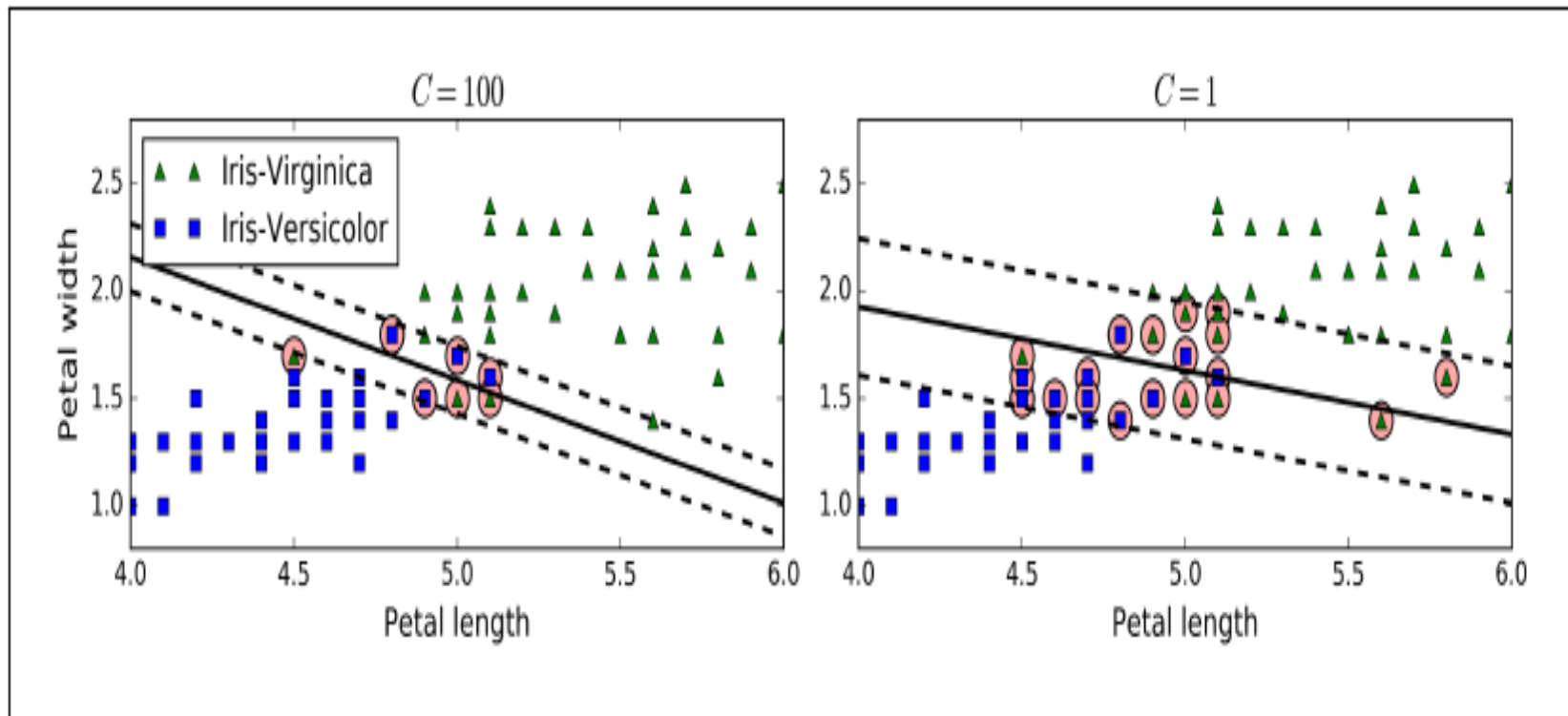


Figure 5-4. Fewer margin violations versus large margin

If your SVM model is overfitting, you can try regularizing it by reducing C .

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris-Virginica

svm_clf = Pipeline((
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1, loss="hinge")),
))

svm_clf.fit(X_scaled, y)
```

Then, as usual, you can use the model to make predictions:

```
>>> svm_clf.predict([[5.5, 1.7]])
array([ 1.])
```


Nonlinear SVM Classification

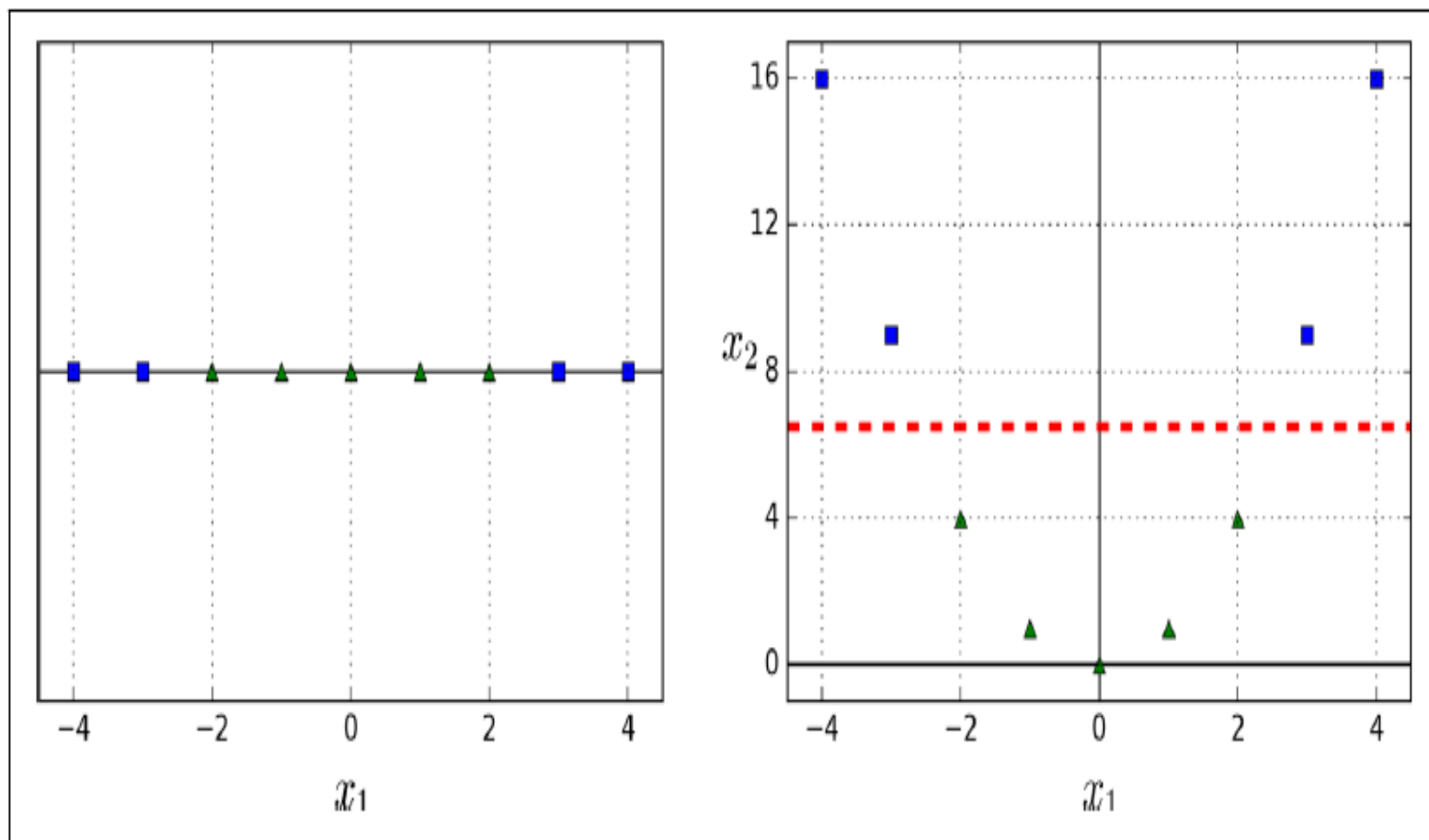


Figure 5-5. Adding features to make a dataset linearly separable

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline((
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
))

polynomial_svm_clf.fit(X, y)
```

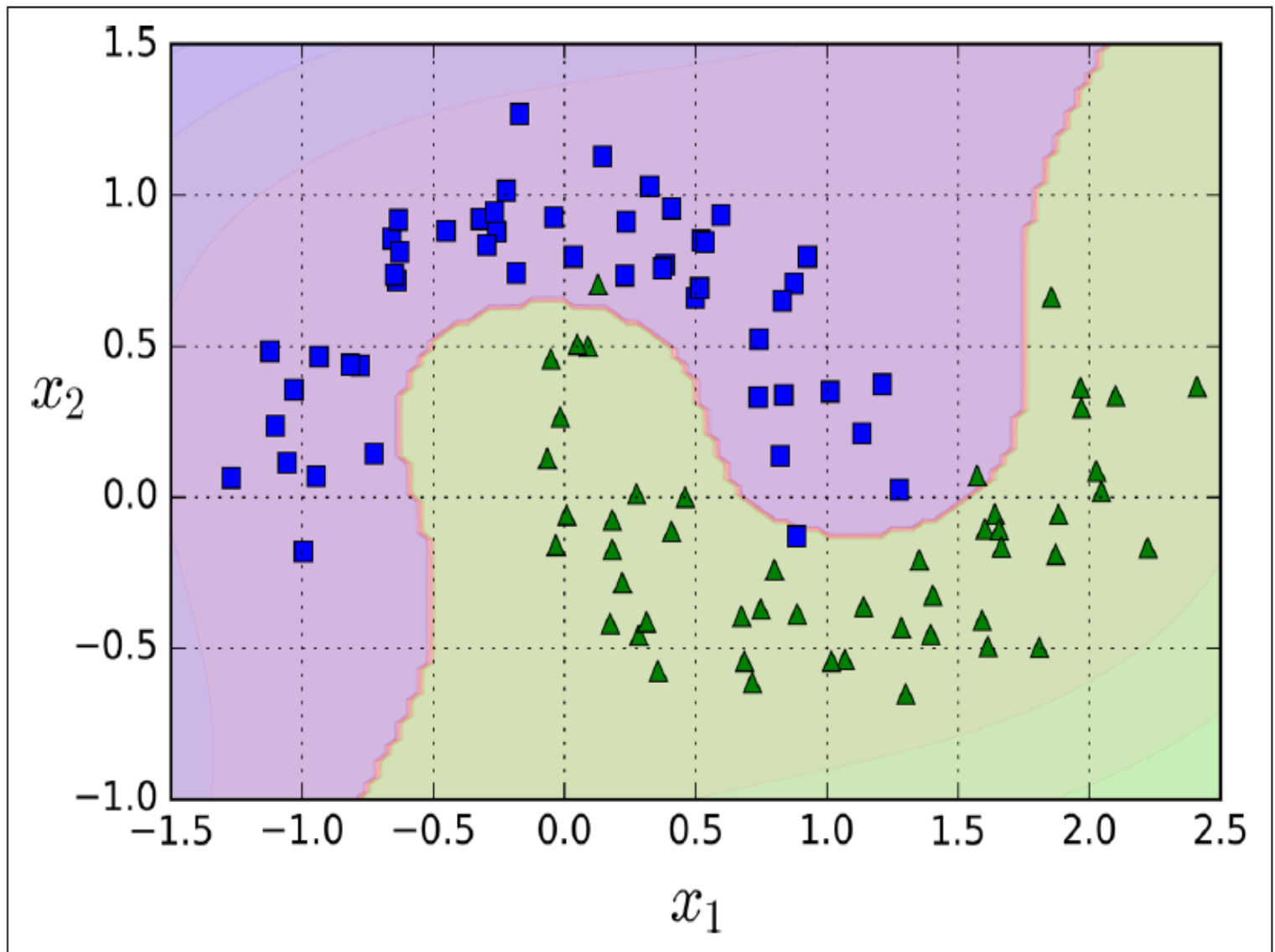


Figure 5-6. Linear SVM classifier using polynomial features

Polynomial Kernel

- Adding polynomial features is simple to implement and can work great with all sorts of Machine Learning algorithms, but at a low polynomial degree it cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow.
- Fortunately, when using SVMs you can apply an almost miraculous mathematical technique called the *kernel trick* .It makes it possible to get the same result as if you added many polynomial features, even with very highdegree polynomials, without actually having to add them.
- So there is no combinatorial explosion of the number of features since you don't actually add any features. This trick is implemented by the SVC class.

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline((
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
))
poly_kernel_svm_clf.fit(X, y)
```

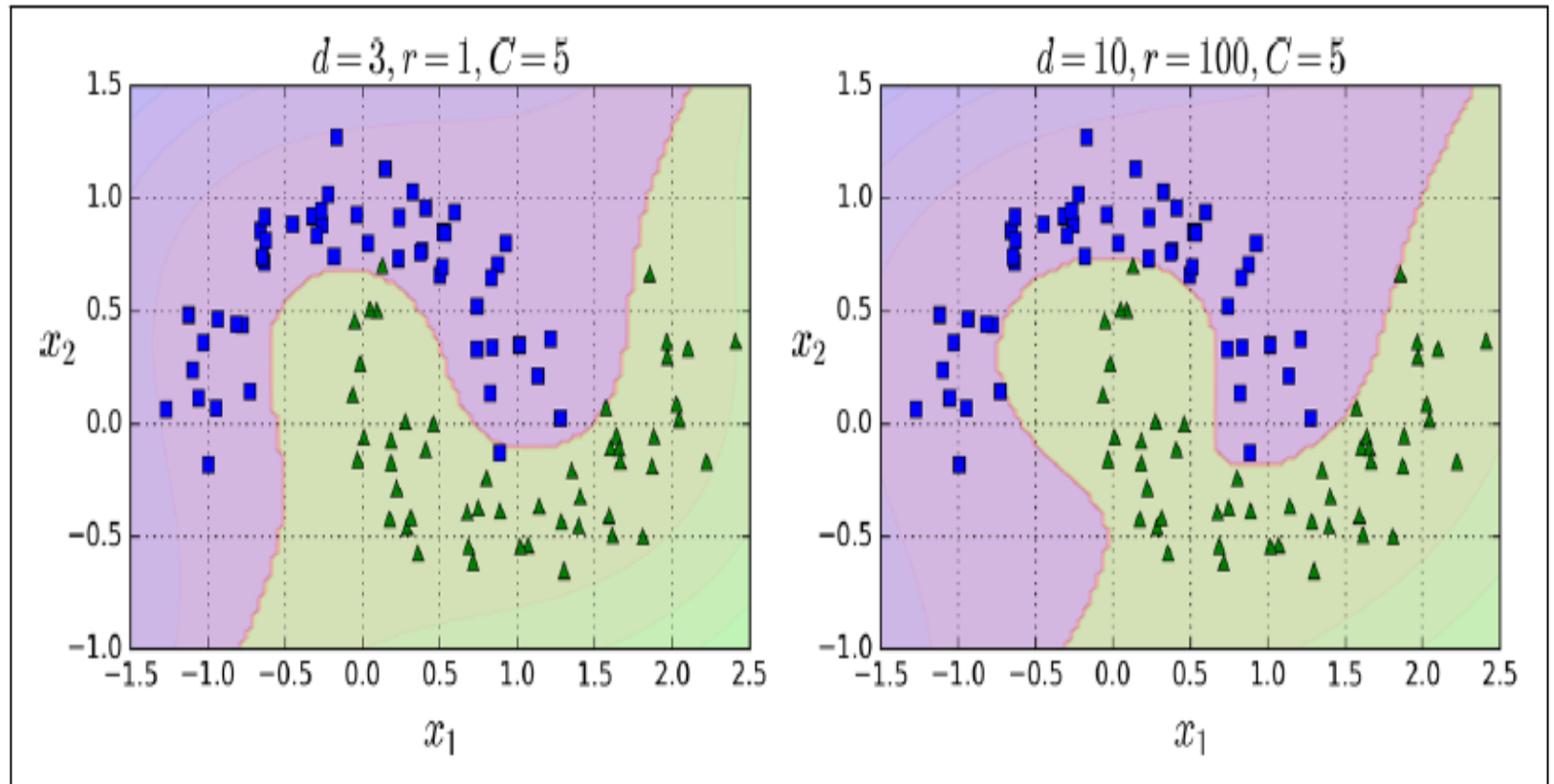


Figure 5-7. SVM classifiers with a polynomial kernel

Adding Similarity Features

- Another technique to tackle nonlinear problems is to add features computed using a *similarity function that measures how much each instance resembles a particular landmark*.
- For example, let's take the one-dimensional dataset discussed earlier and add two landmarks to it at $x_l = -2$ and $x_l = 1$
- Next, let's define the similarity function to be the Gaussian Radial Basis Function (RBF) with $\gamma = 0.3$

Equation 5-1. Gaussian RBF

$$\phi_{\gamma}(\mathbf{x}, \ell) = \exp \left(-\gamma \| \mathbf{x} - \ell \|^2 \right)$$

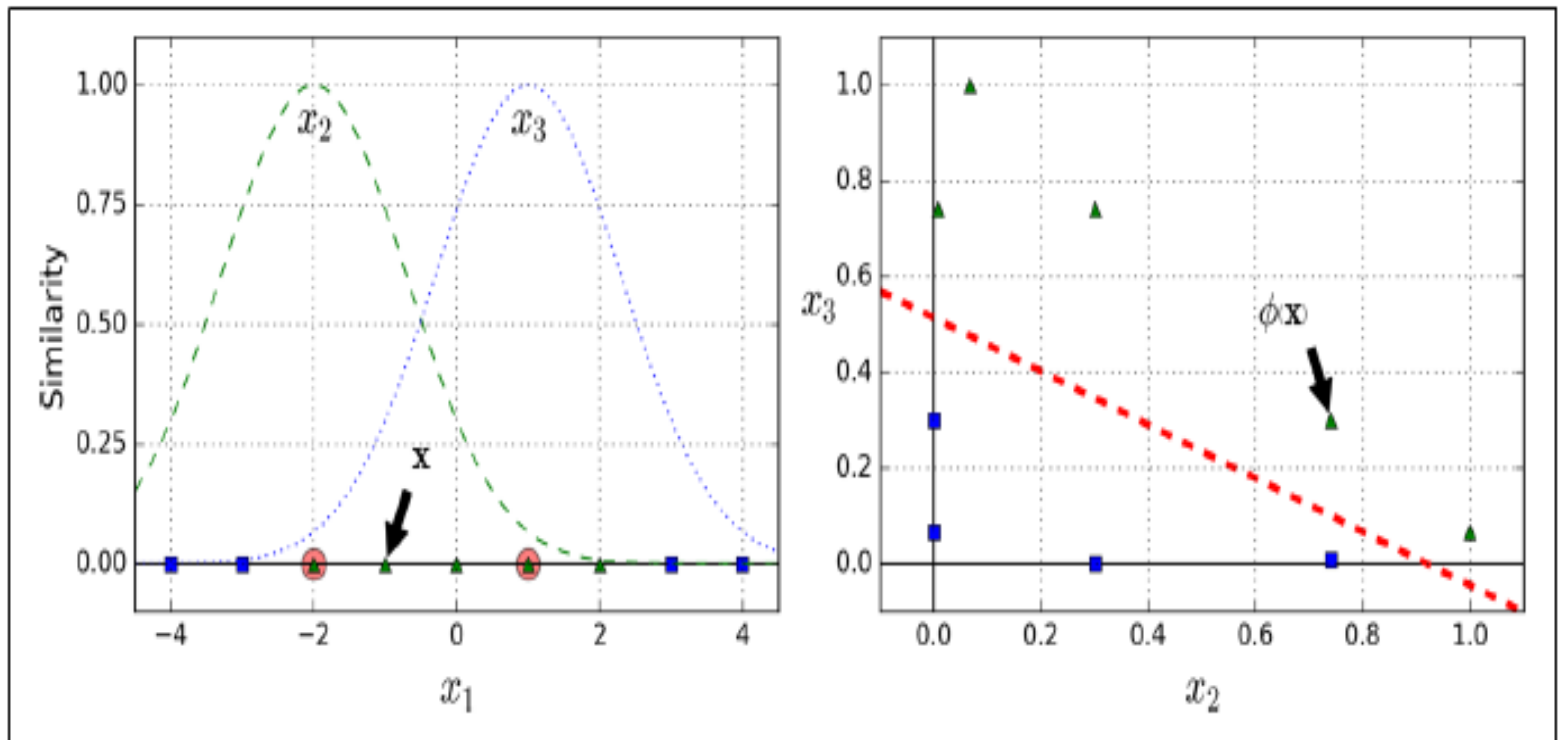


Figure 5-8. Similarity features using the Gaussian RBF

Gaussian RBF Kernel

```
rbf_kernel_svm_clf = Pipeline(  
    ("scaler", StandardScaler()),  
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))  
))  
rbf_kernel_svm_clf.fit(X, y)
```

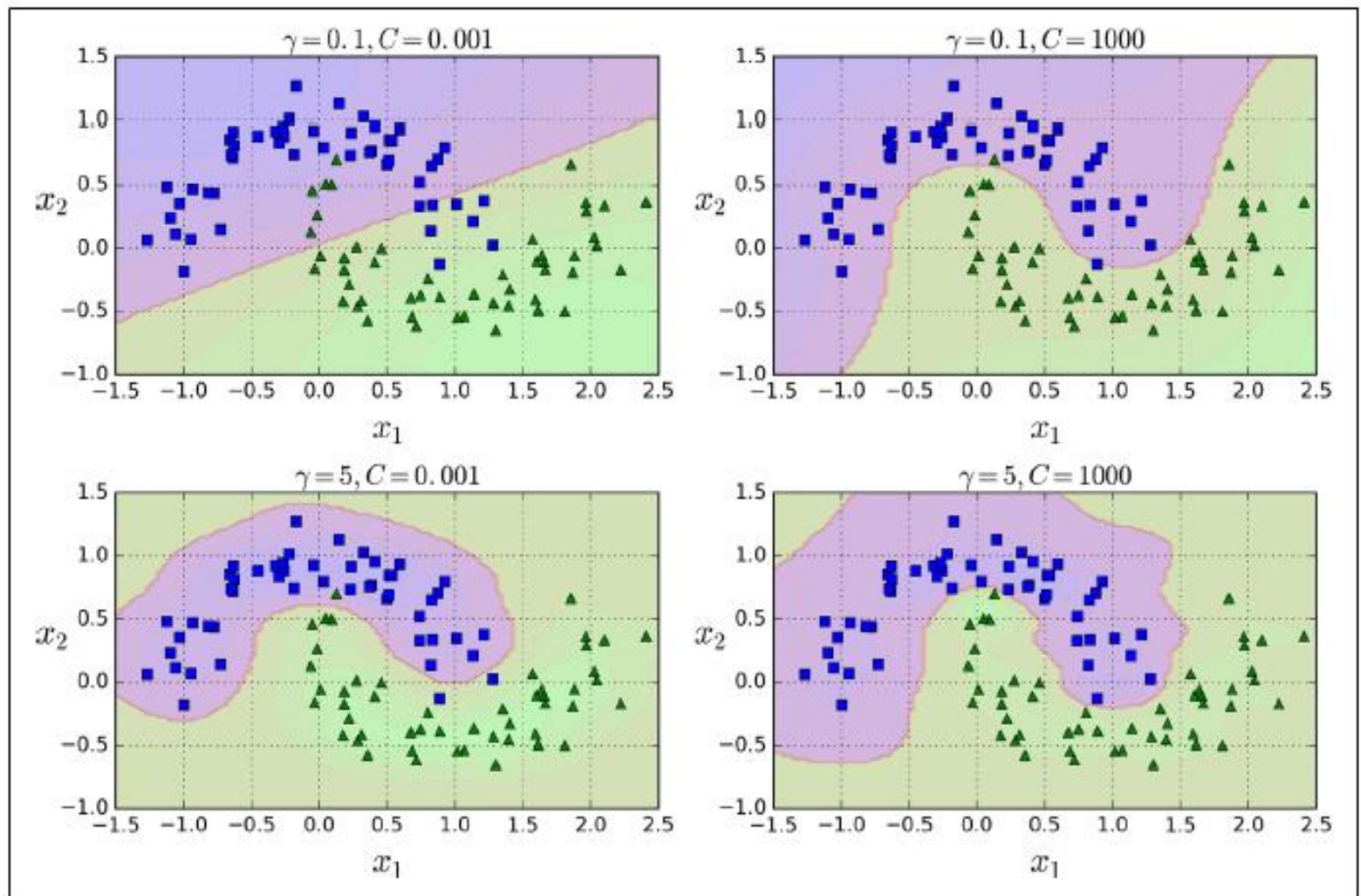


Figure 5-9. SVM classifiers using an RBF kernel

Computational Complexity

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
LinearSVC	$O(m \times n)$	No	Yes	No
SGDClassifier	$O(m \times n)$	Yes	Yes	No
SVC	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

SVM Regression

- The SVM algorithm is quite versatile: not only does it support linear and nonlinear classification, but it also supports linear and nonlinear regression.
- The trick is to reverse the objective: instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM Regression tries to fit as many instances as possible on the street while limiting margin violations (i.e., instances off the street). The width of the street is controlled by a hyperparameter ϵ

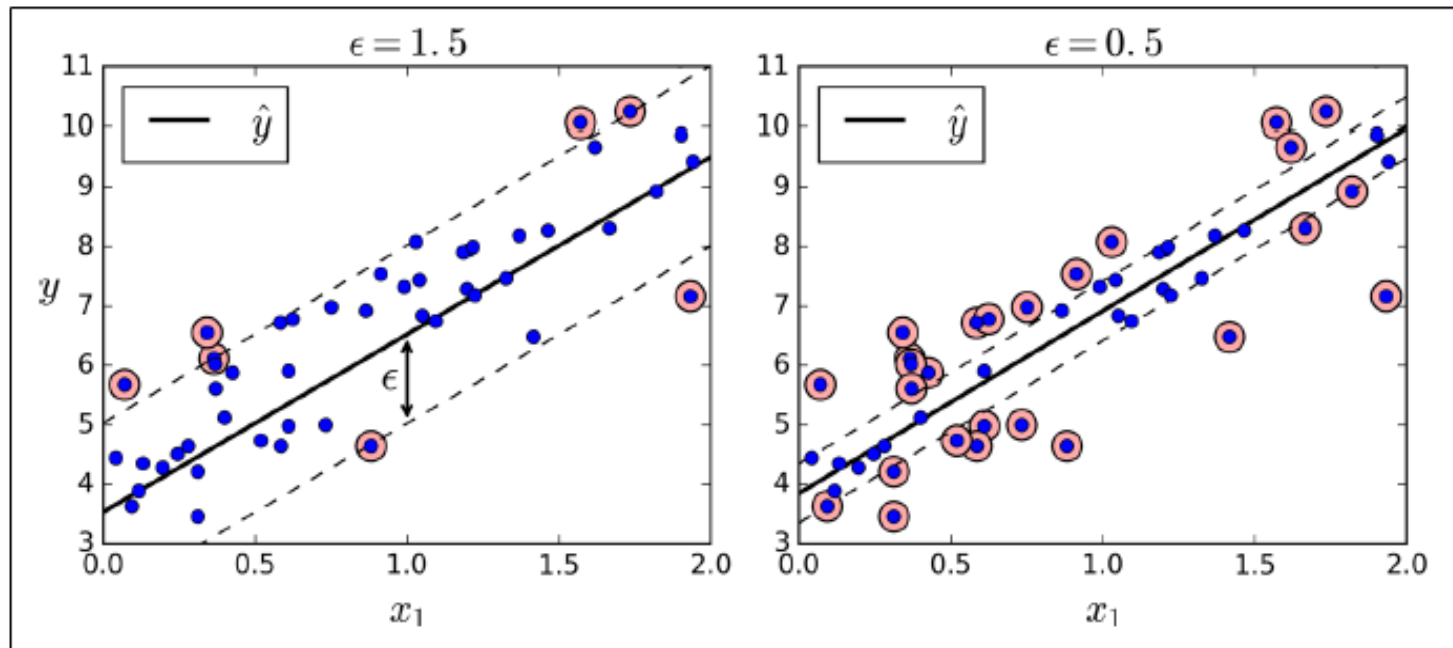


Figure 5-10. SVM Regression

```
from sklearn.svm import LinearSVR
```

```
svm_reg = LinearSVR(epsilon=1.5)
```

```
svm_reg.fit(X, y)
```

To tackle nonlinear regression tasks, you can use a kernelized SVM model. For example, **Figure 5-11** shows SVM Regression on a random quadratic training set, using a 2nd-degree polynomial kernel. There is little regularization on the left plot (i.e., a large C value), and much more regularization on the right plot (i.e., a small C value).

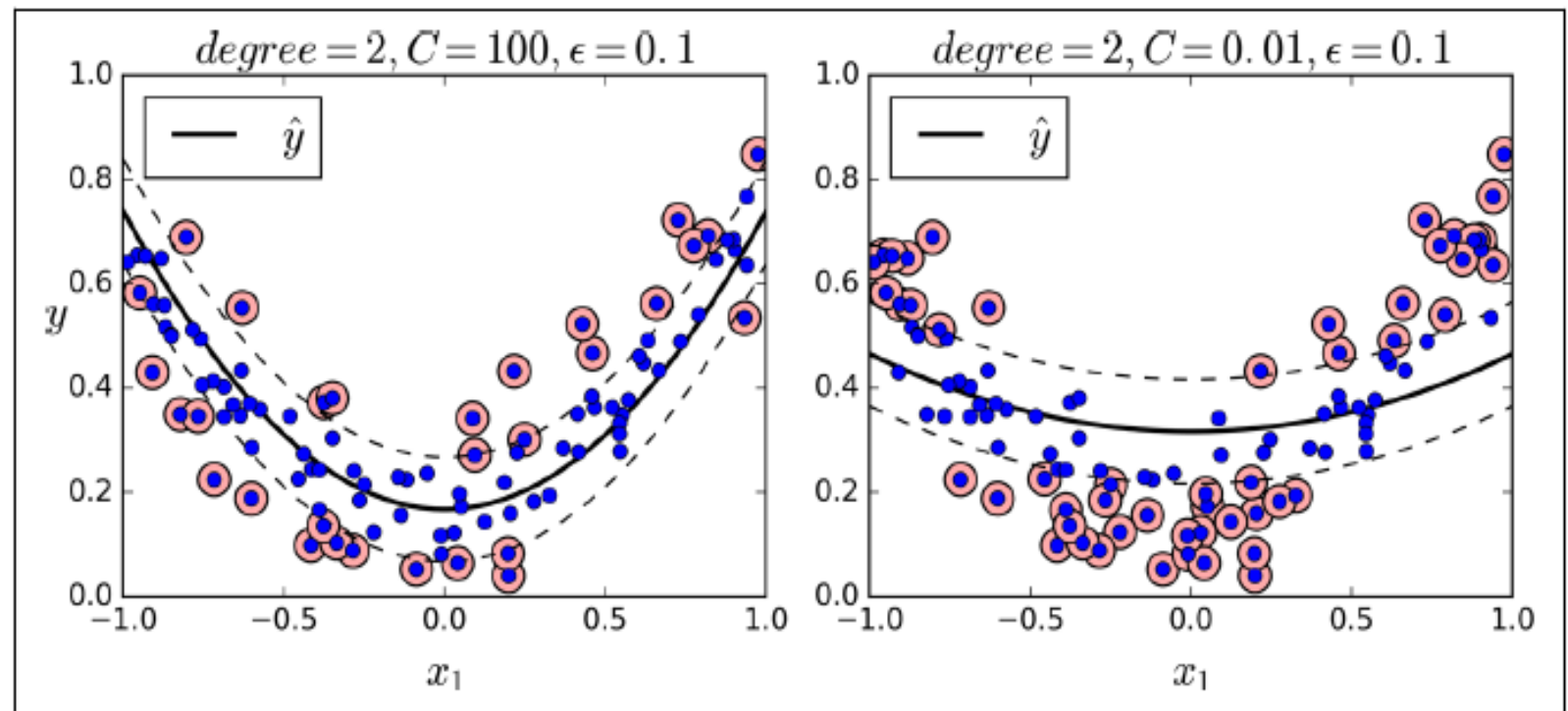


Figure 5-11. SVM regression using a 2nd-degree polynomial kernel

- The SVR class is the regression equivalent of the SVC class, and the LinearSVR class is the regression equivalent of the LinearSVC class.

```
from sklearn.svm import SVR
```

```
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)  
svm_poly_reg.fit(X, y)
```

Training and Visualizing a Decision Tree

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
```

```
iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target
```

```
tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

```
from sklearn.tree import export_graphviz
```

```
export_graphviz(
    tree_clf,
    out_file=image_path("iris_tree.dot"),
    feature_names=iris.feature_names[2:],
    class_names=iris.target_names,
    rounded=True,
    filled=True
)
```



```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

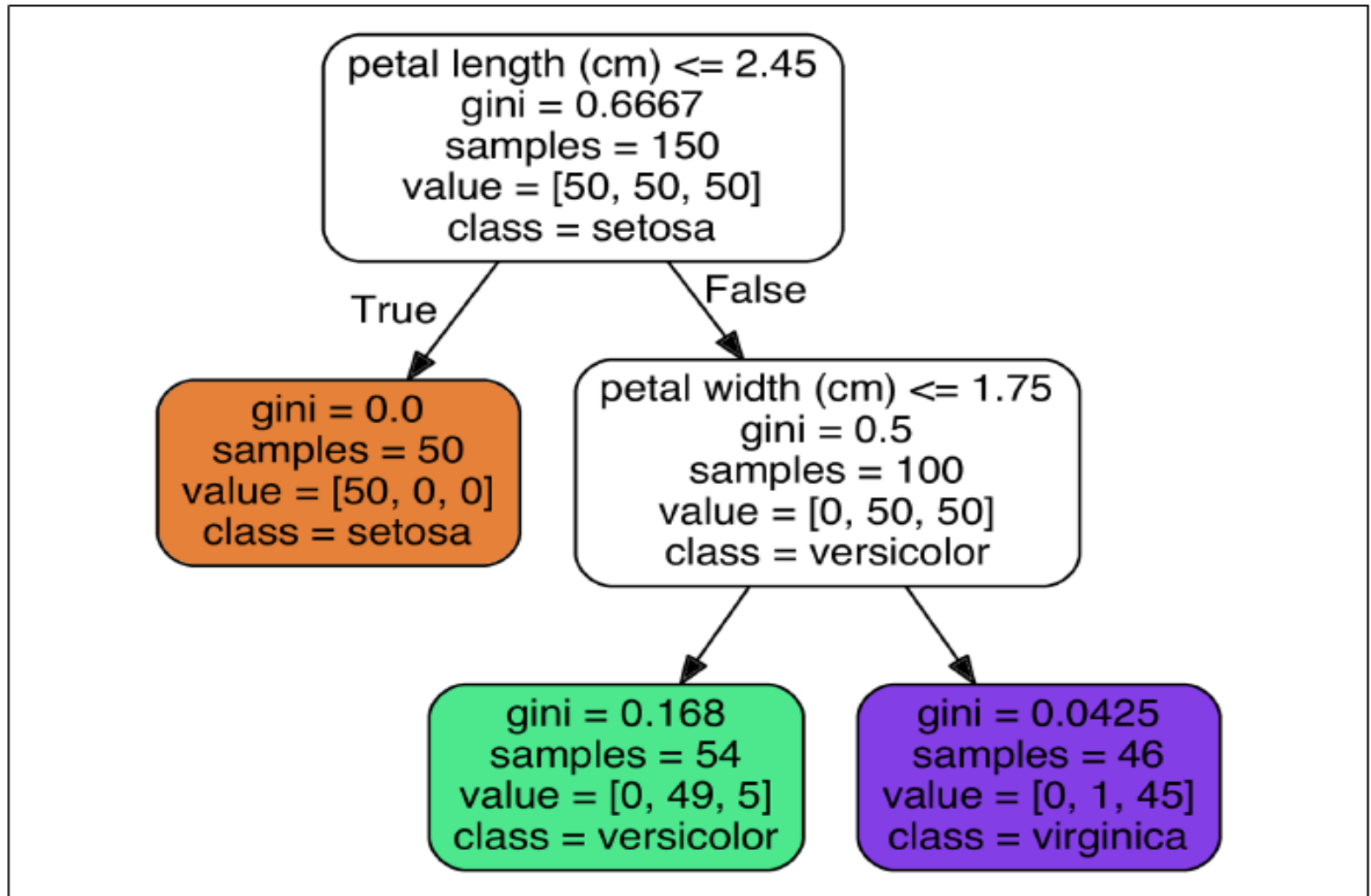


Figure 6-1. Iris Decision Tree

Estimating Class Probabilities

```
>>> tree_clf.predict_proba([[5, 1.5]])  
array([[ 0. ,  0.90740741,  0.09259259]])  
>>> tree_clf.predict([[5, 1.5]])  
array([1])
```

Making Predictions

Equation 6-1. Gini impurity

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

- $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node.

gini score equal to $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$.

0 49 5
54 54 54

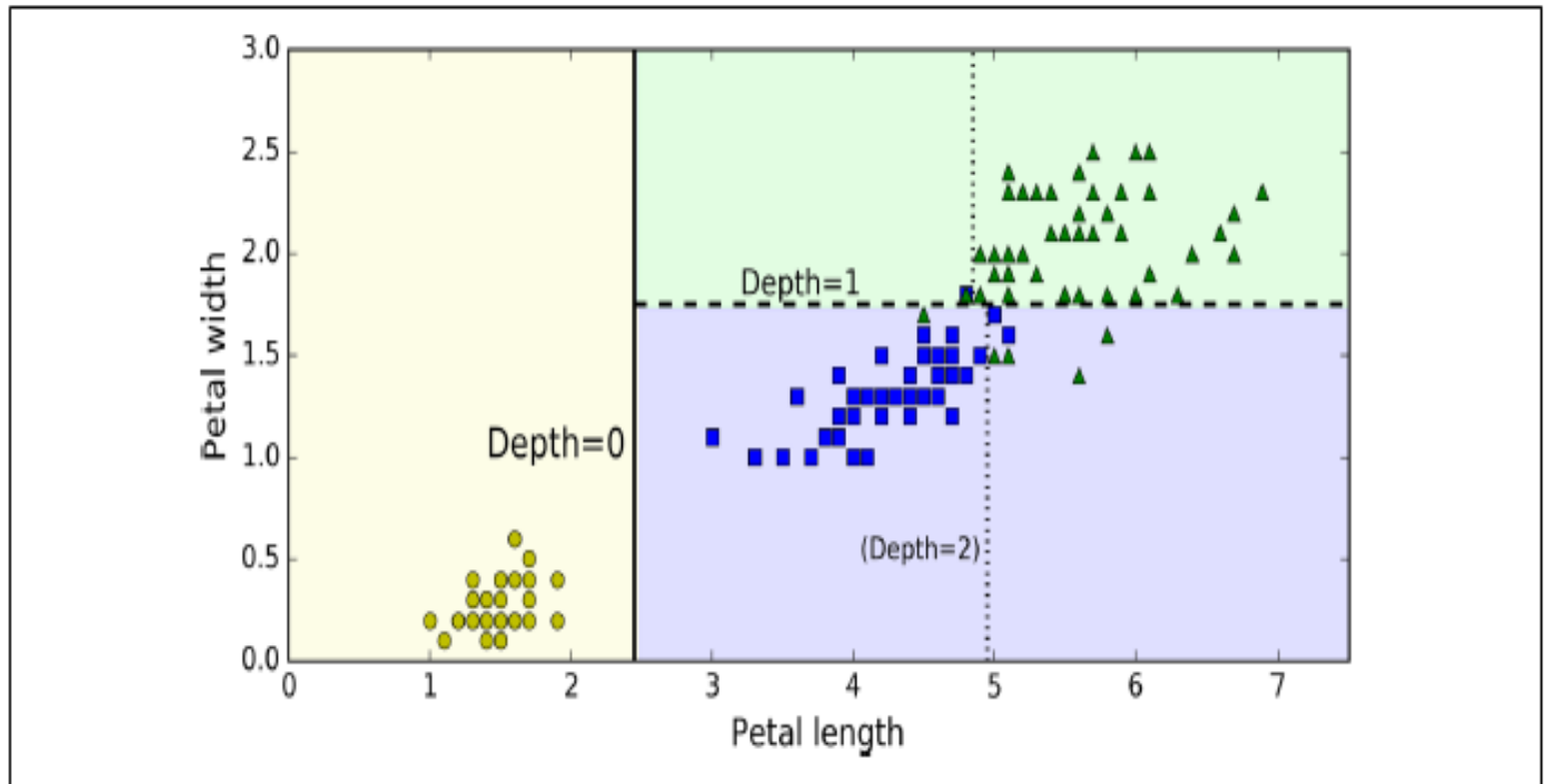


Figure 6-2. Decision Tree decision boundaries

The CART Training Algorithm

Scikit-Learn uses the *Classification And Regression Tree (CART) algorithm* to train Decision Trees (also called “growing” trees).


The algorithm first splits the training set in two subsets using a single feature k and a threshold t_k (e.g., “petal length ≤ 2.45 cm”).

It searches for the pair (k, t_k) that produces the purest subsets (weighted by their size). The cost function that the algorithm tries to minimize is given by Equation 6-2.

Equation 6-2. CART cost function for classification

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where $\begin{cases} G_{\text{left/right}} & \text{measures the impurity of the left/right subset,} \\ m_{\text{left/right}} & \text{is the number of instances in the left/right subset.} \end{cases}$

- 
- Once it has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets and so on, recursively.
 - It stops recursion once it reaches the maximum depth (defined by the `max_depth` hyperparameter), or if it cannot find a split that will reduce impurity.
 - A few other hyperparameters control additional stopping conditions(`min_samples_split`,`min_samples_leaf`,`min_weight_fraction_leaf`, and `max_leaf_nodes`).

Computational Complexity

- Making predictions requires traversing the Decision Tree from the root to a leaf. Decision Trees are generally approximately balanced, so traversing the Decision Tree requires going through roughly $O(\log_2(m))$ nodes.
- However, the training algorithm compares all features (or less if `max_features` is set) on all samples at each node. This results in a training complexity of $O(n \times m \log(m))$.

Gini Impurity or Entropy?

- By default, the Gini impurity measure is used, but you can select the *entropy impurity* measure instead by setting the criterion hyperparameter to "entropy".

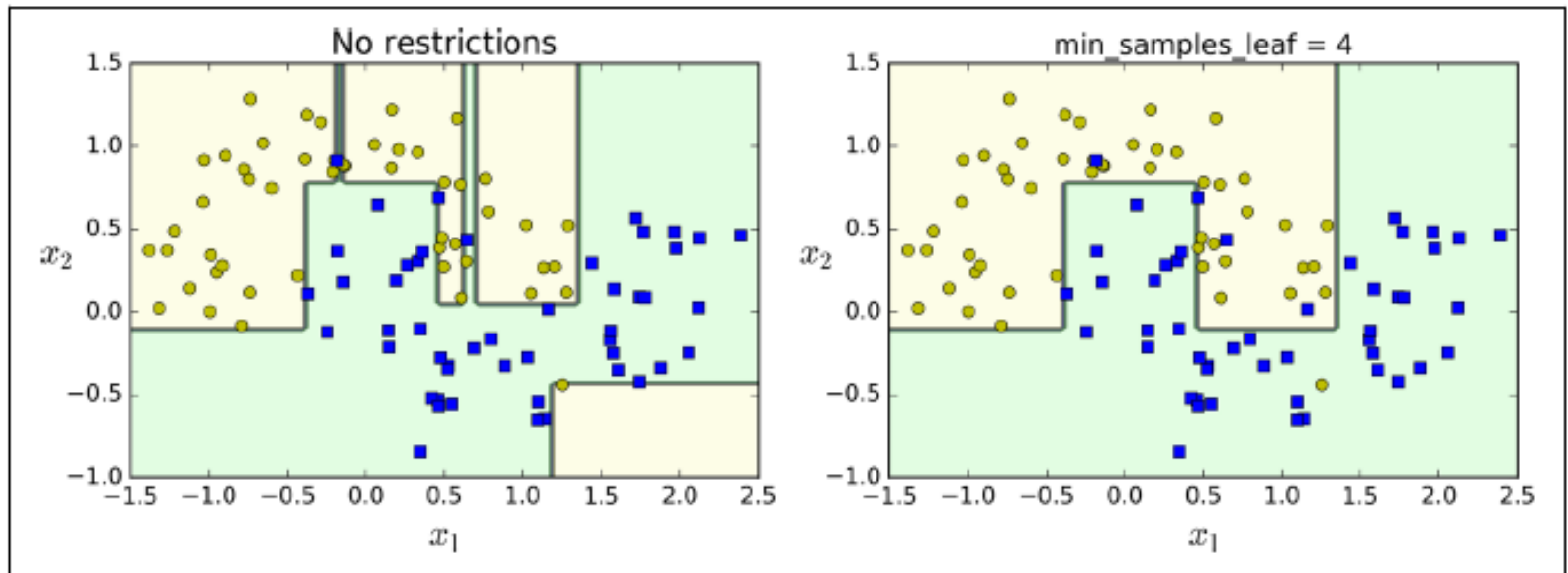
$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^n p_{i,k} \log(p_{i,k})$$

$$-\frac{49}{54} \log\left(\frac{49}{54}\right) - \frac{5}{54} \log\left(\frac{5}{54}\right) \approx 0.31.$$

Regularization Hyperparameters

- Generally you can restrict the maximum depth of the Decision Tree using `max_depth` hyperparameter. The default value is `None`, which means unlimited. Reducing `max_depth` will regularize the model and thus reduce the risk of overfitting.
- `min_samples_split` : the minimum number of samples a node must have before it can be split.
- `min_samples_leaf` : the minimum number of samples a leaf node must have.
- `min_weight_fraction_leaf` : same as `min_samples_leaf` but expressed as a fraction of the total number of weighted instances

- `max_leaf_nodes` : maximum number of leaf nodes.
- `max_features` : maximum number of features that are evaluated for splitting at each node.
- Increasing `min_*` hyperparameters or reducing `max_*` hyperparameters will regularize the model.



Regression

```
# Quadratic training set + noise
```

```
np.random.seed(42)
```

```
m = 200
```

```
X = np.random.rand(m, 1)
```

```
y = 4 * (X - 0.5) ** 2
```

```
y = y + np.random.randn(m, 1) / 10
```

```
from sklearn.tree import DecisionTreeRegressor
```

```
tree_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
```

```
tree_reg.fit(X, y)
```

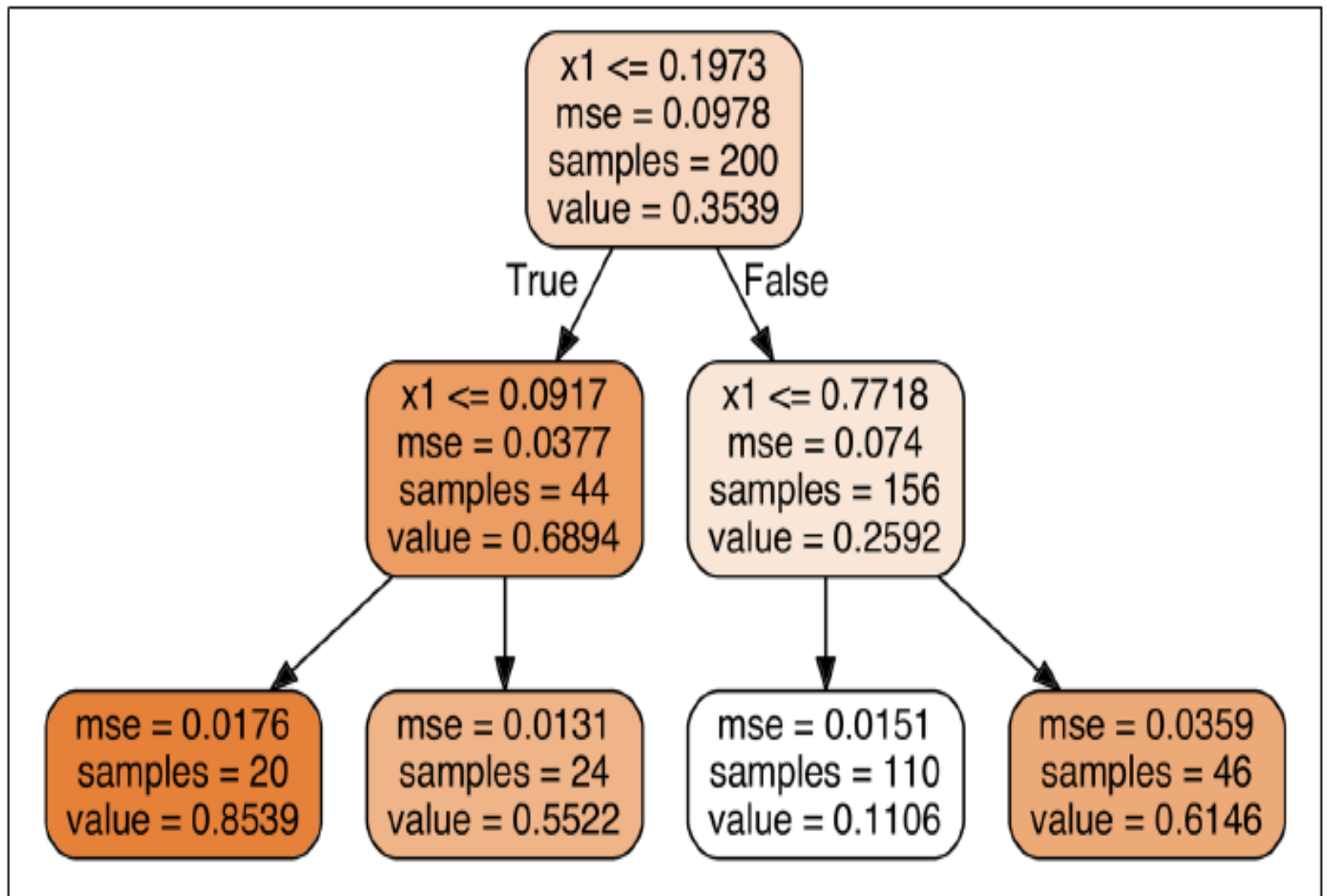


Figure 6-4. A Decision Tree for regression

Equation 6-4. CART cost function for regression

$$J(k, t_k) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m} \text{MSE}_{\text{right}} \quad \text{where} \quad \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} (\hat{y}_{\text{node}} - y^{(i)})^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

Ensemble Learning and Random Forests

- If you aggregate the predictions of a group of predictors such as classifiers or regressors, you will often get better predictions than with the best individual predictor.
- A group of predictors is called an *ensemble*, thus, this technique is called *Ensemble Learning*, and an Ensemble Learning algorithm is called an *Ensemble method*.
- For example, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you just obtain the predictions of all individual trees, then predict the class that gets the most votes. Such an ensemble of Decision Trees is called a *Random Forest*

Voting Classifiers

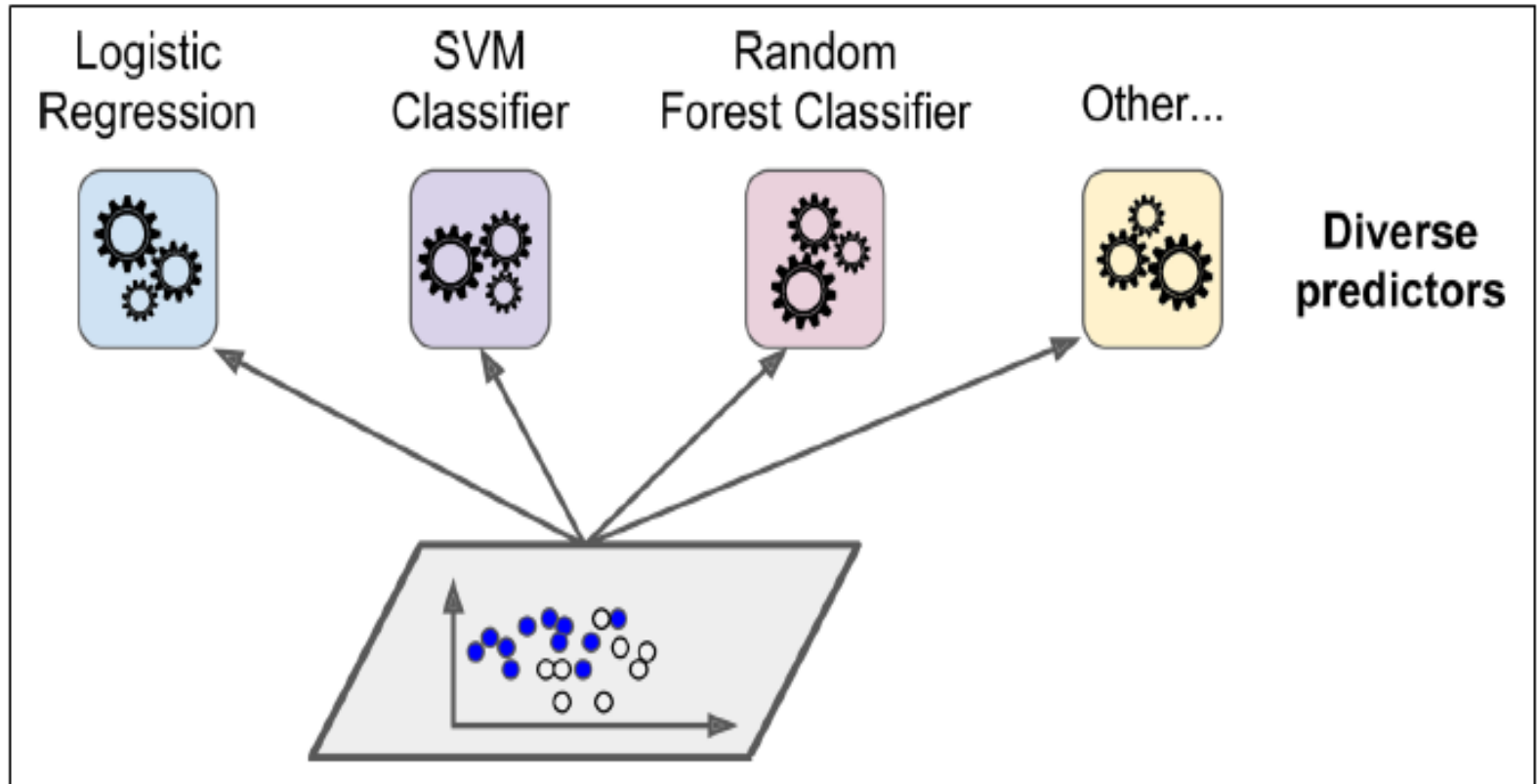


Figure 7-1. Training diverse classifiers

- A very simple way to create an even better classifier is to aggregate the predictions of each classifier and predict the class that gets the most votes. This majority-vote classifier is called a *hard voting classifier*.

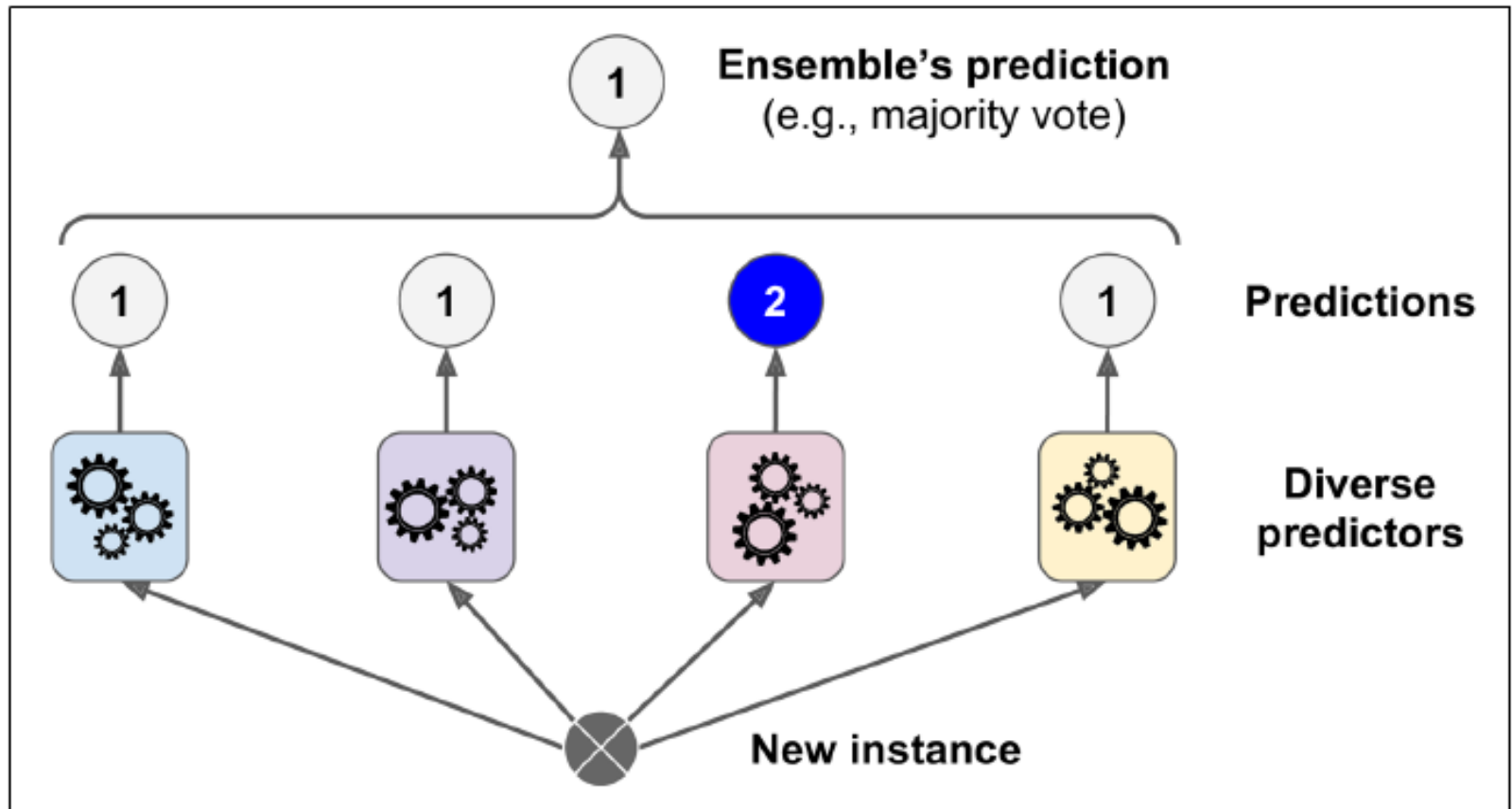
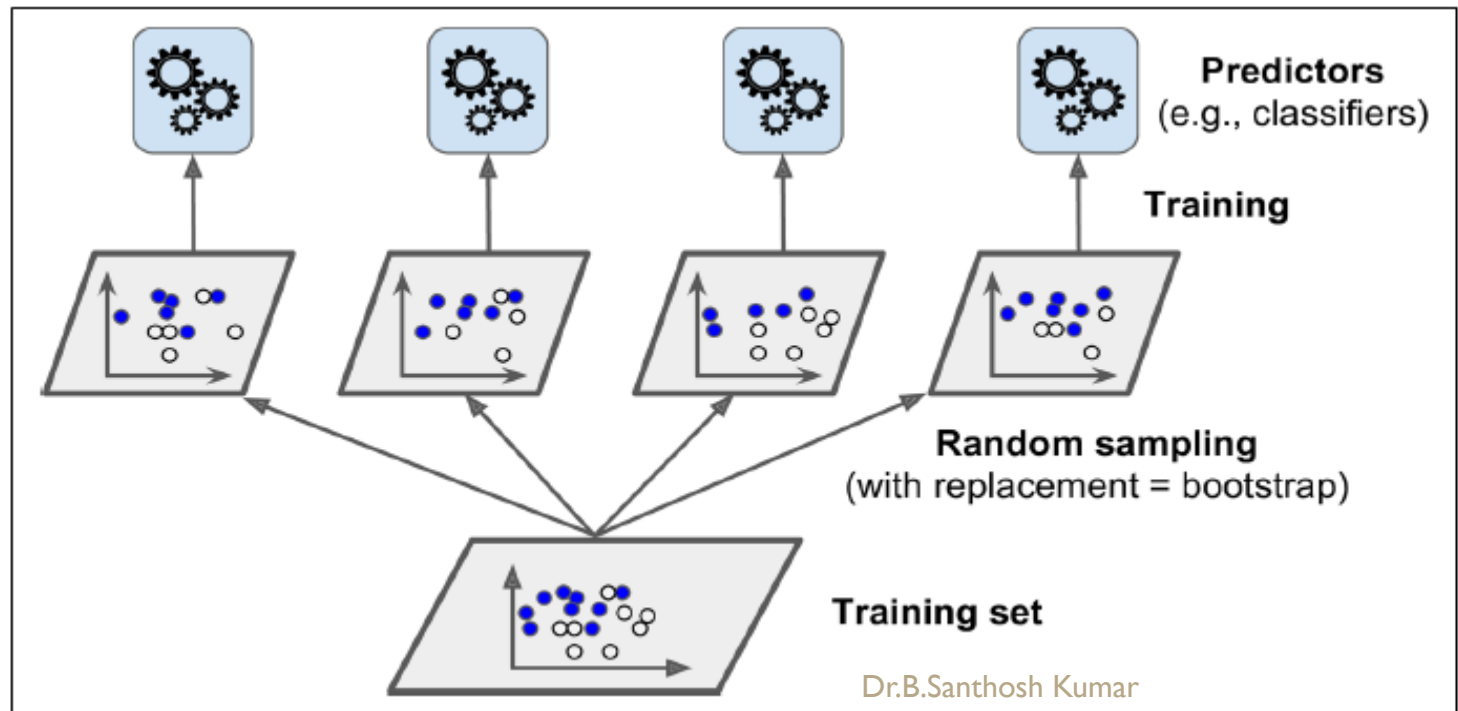


Figure 7-2. Hard voting classifier predictions

- If all classifiers are able to estimate class probabilities (i.e., they have a `predict_proba()` method), then you can tell Scikit-Learn to predict the class with the highest class probability, averaged over all the individual classifiers.
- This is called *soft voting*. *It often achieves higher performance than hard voting because it gives more weight to highly confident votes.*
- All you need to do is replace `voting="hard"` with `voting="soft"` and ensure that all classifiers can estimate class probabilities.


Bagging and Pasting

- The approach of using the same training algorithm for every predictor, but to train them on different random subsets of the training set where the sampling is performed with replacement is called Bagging.
- If the sampling is performed without replacement it is called Pasting.



Bagging and Pasting in Scikit-Learn

- Scikit-Learn offers a simple API for both bagging and pasting with the `BaggingClassifier` class (or `BaggingRegressor` for regression).
- The following code trains an ensemble of 500 Decision Tree classifiers each trained on 100 training instances randomly sampled from the training set with replacement (this is an example of bagging, but if you want to use pasting instead, just set `bootstrap=False`).
- The `n_jobs` parameter tells Scikit-Learn the number of CPU cores to use for training and predictions (–1 tells Scikit-Learn to use all available cores)



```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1
)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

Out-of-Bag Evaluation

- With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all.
- By default a `BaggingClassifier` samples m training instances with replacement (`bootstrap=True`), where m is the size of the training set.
- This means that only about 63% of the training instances are sampled on average for each predictor. The remaining 37% of the training instances that are not sampled are called *out-of-bag (oob) instances*.

In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic oob evaluation after training. The following code demonstrates this. The resulting evaluation score is available through the `oob_score_` variable:

```
>>> bag_clf = BaggingClassifier(  
>>>     DecisionTreeClassifier(), n_estimators=500,  
>>>     bootstrap=True, n_jobs=-1, oob_score=True)  
  
>>> bag_clf.fit(X_train, y_train)  
>>> bag_clf.oob_score_  
0.93066666666666664
```

According to this oob evaluation, this `BaggingClassifier` is likely to achieve about 93.1% accuracy on the test set. Let's verify this:

```
>>> from sklearn.metrics import accuracy_score  
>>> y_pred = bag_clf.predict(X_test)  
>>> accuracy_score(y_test, y_pred)  
0.93600000000000005
```

- The oob decision function for each training instance is also available through the `oob_decision_function_` variable. In this case the decision function returns the class probabilities for each training instance.
- For example, the oob evaluation estimates that the second training instance has a 60.6% probability of belonging to the positive class (and 39.4% of belonging to the negative class)

```
>>> bag_clf.oob_decision_function_  
array([[ 0.          ,  1.          ],  
       [ 0.60588235,  0.39411765],  
       [ 1.          ,  0.          ],  
       ...,  
       [ 1.          ,  0.          ],  
       [ 0.          ,  1.          ],  
       [ 0.48958333,  0.51041667]])
```

Random Patches and Random Subspaces

- The BaggingClassifier class supports sampling the features as well. This is controlled by two hyperparameters: max_features and bootstrap_features.
- They work the same way as max_samples and bootstrap, but for feature sampling instead of instance sampling.
- Thus, each predictor will be trained on a random subset of the input features.
- This is particularly useful when you are dealing with high-dimensional inputs (such as images).
- Sampling both training instances and features is called the *Random Patches method*. Keeping all training instances (i.e., bootstrap=False and max_samples=1.0) but sampling features (i.e., bootstrap_features=True and/or max_features smaller than 1.0) is called the *Random Subspaces method*.

Random Forests

- A Random Forest is an ensemble of Decision Trees, generally trained via the bagging method (or sometimes pasting)
- Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can instead use the `RandomForestClassifier` class, which is more convenient and optimized for Decision Trees.
- The following code trains a Random Forest classifier with 500 trees (each limited to maximum 16 nodes), using all available CPU cores



```
from sklearn.ensemble import RandomForestClassifier
```

```
rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)  
rnd_clf.fit(X_train, y_train)
```

```
y_pred_rf = rnd_clf.predict(X_test)
```

Dimensionality Reduction

- Many Machine Learning problems involve thousands or even millions of features for each training instance which makes training extremely slow and can also make it much harder to find a good solution.
- This problem is often referred to as the *curse of dimensionality*.
- For example, consider the MNIST images the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information.

Main Approaches for Dimensionality Reduction

- Projection :

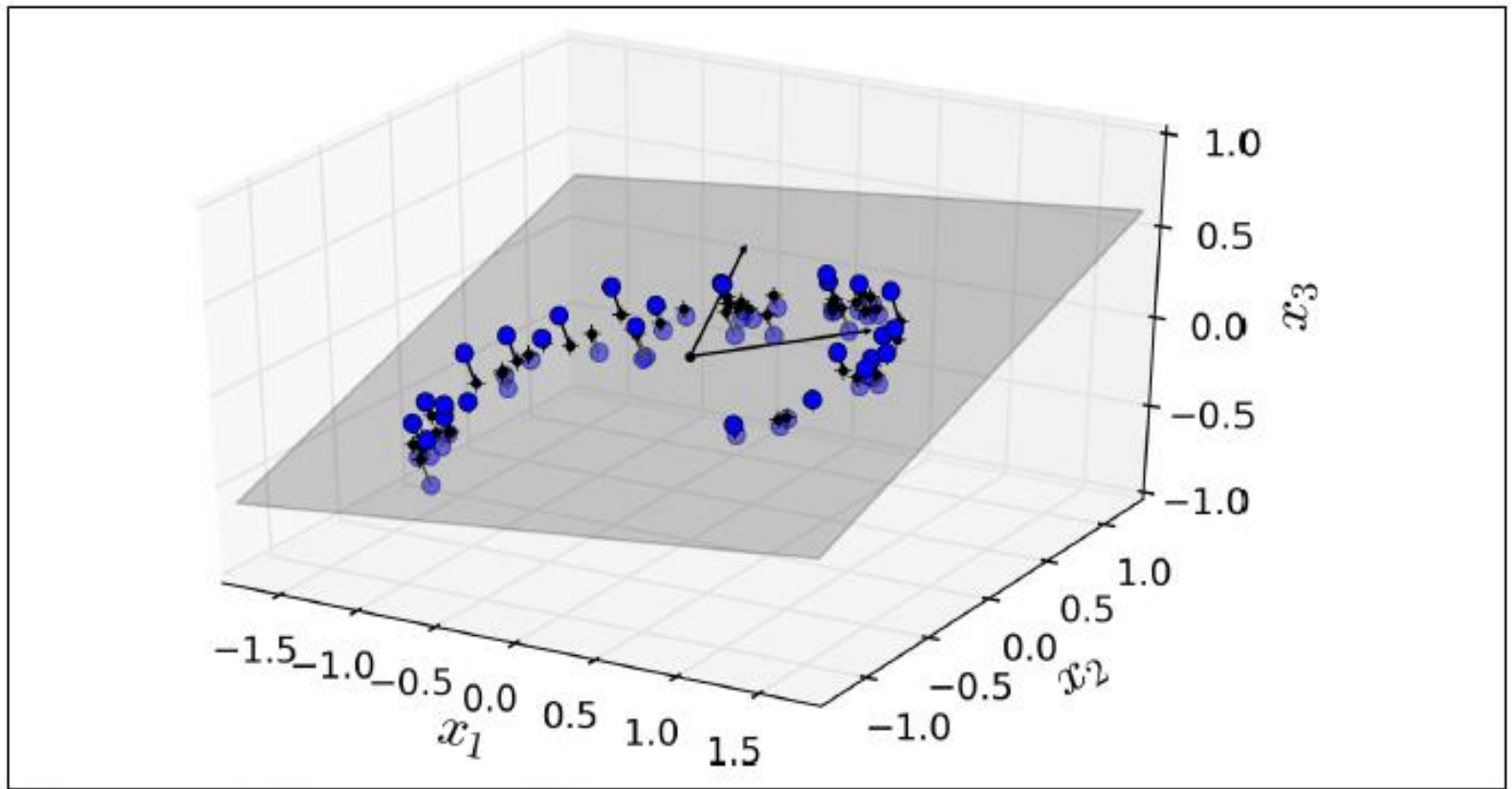


Figure 8-2. A 3D dataset lying close to a 2D subspace

- Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the high-dimensional (3D) space.
- Now if we project every training instance perpendicularly onto this subspace (as represented by the short lines connecting the instances to the plane), we get the new 2D dataset shown in Figure 8-3.
- We have just reduced the dataset's dimensionality from 3D to 2D. Note that the axes correspond to new features z_1 and z_2 (*the coordinates of the projections on the plane*).

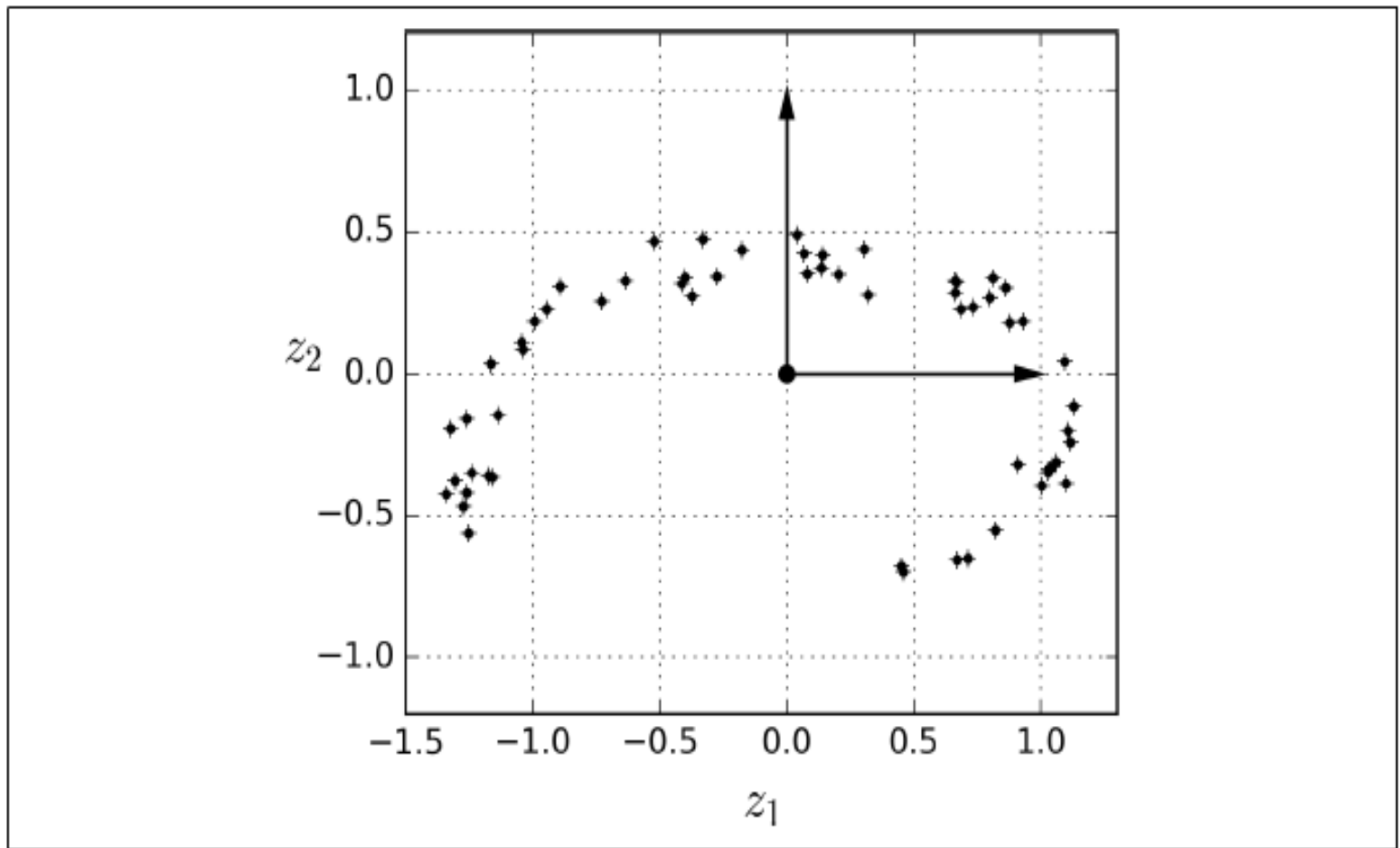


Figure 8-3. The new 2D dataset after projection

- However, projection is not always the best approach to dimensionality reduction. In many cases the subspace may twist and turn, such as in the famous *Swiss roll toy dataset* represented in Figure 8-4.

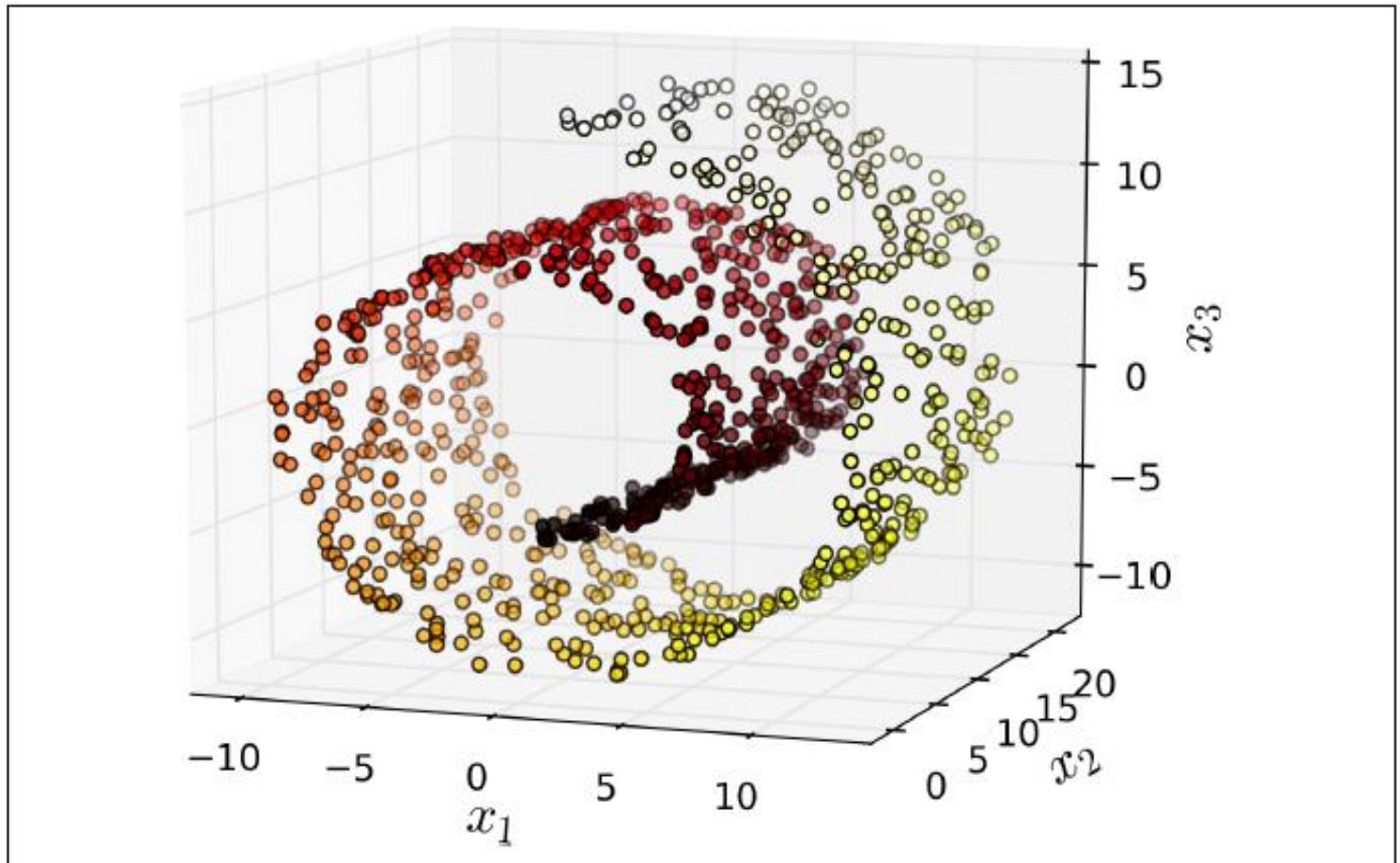


Figure 8-4. Swiss roll dataset

Manifold Learning

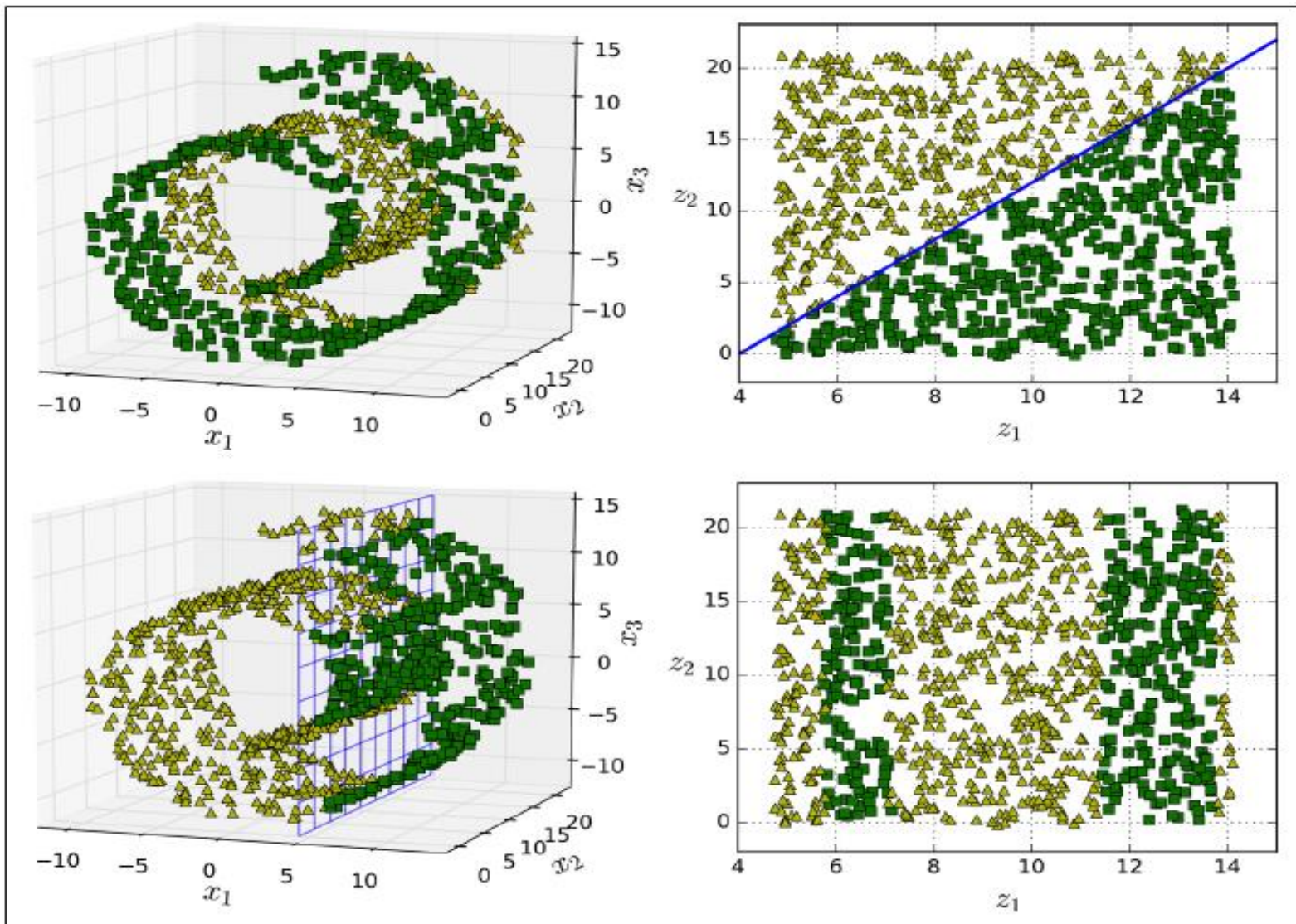


Figure 8-6. The decision boundary may not always be simpler with lower dimensions

PCA

- *Principal Component Analysis (PCA)* is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it.
- Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane.
- For example, a simple 2D dataset is represented on the left of Figure 8-7, along with three different axes (i.e., one-dimensional hyperplanes). On the right is the result of the projection of the dataset onto each of these axes.
- As you can see, the projection onto the solid line preserves the maximum variance, while the projection onto the dotted line preserves very little variance, and the projection onto the dashed line preserves an intermediate amount of variance

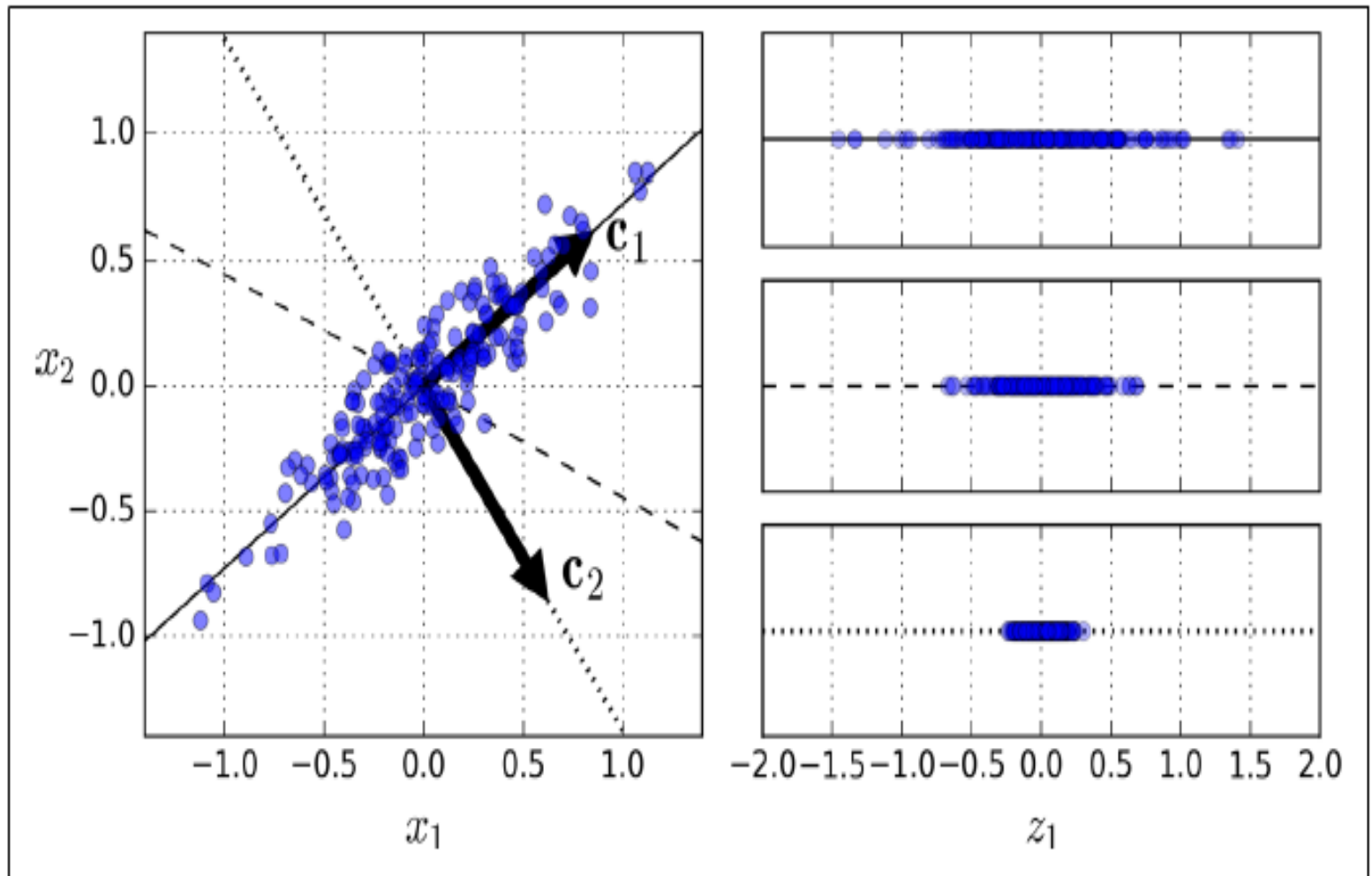


Figure 8-7. Selecting the subspace onto which to project

Principal Components

- PCA identifies the axis that accounts for the largest amount of variance in the training set.
- In Figure 8-7, it is the solid line. It also finds a second axis, orthogonal to the first one, that accounts for the largest amount of remaining variance.
- In this 2D example there is no choice: it is the dotted line. If it were a higher-dimensional dataset, PCA would also find a third axis, orthogonal to both previous axes, and a fourth, a fifth, and so on—as many axes as the number of dimensions in the dataset.

- The unit vector that defines the i th axis is called the i th *principal component (PC)*. In Figure the 1st PC is c_1 and the 2nd PC is c_2 .
- There is a standard matrix factorization technique called *Singular Value Decomposition (SVD)* that can decompose the training set matrix X into the dot product of three matrices $U \cdot \Sigma \cdot V^T$, where V^T contains all the *principal components* that we are looking for, as shown in Equation 8-1.

Equation 8-1. Principal components matrix

$$V^T = \begin{pmatrix} | & | & & | \\ c_1 & c_2 & \cdots & c_n \\ | & | & & | \end{pmatrix}$$

The following Python code uses NumPy's `svd()` function to obtain all the principal components of the training set, then extracts the first two PCs:

```
X_centered = X - X.mean(axis=0)
U, s, V = np.linalg.svd(X_centered)
c1 = V.T[:, 0]
c2 = V.T[:, 1]
```

Projecting Down to d Dimensions

- Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components.
- Selecting this hyperplane ensures that the projection will preserve as much variance as possible.
- To project the training set onto the hyperplane, you can simply compute the dot product of the training set matrix X by the matrix W_d , defined as the matrix containing the first d principal components.

Equation 8-2. Projecting the training set down to d dimensions

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X} \cdot \mathbf{W}_d$$

The following Python code projects the training set onto the plane defined by the first two principal components:

```
W2 = V.T[:, :2]  
X2D = X_centered.dot(W2)
```

Using Scikit-Learn

Scikit-Learn's PCA class implements PCA using SVD decomposition just like we did before. The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
from sklearn.decomposition import PCA
```

```
pca = PCA(n_components = 2)
```

```
X2D = pca.fit_transform(X)
```

After fitting the PCA transformer to the dataset, you can access the principal components using the `components_` variable (note that it contains the PCs as horizontal vectors, so, for example, the first principal component is equal to `pca.components_.T[:, 0]`).

Explained Variance Ratio

- Another very useful piece of information is the *explained variance ratio of each principal component*, available via the `explained_variance_ratio_` variable.
- It indicates the proportion of the dataset's variance that lies along the axis of each principal component.

```
>>> print(pca.explained_variance_ratio_)  
array([ 0.84248607,  0.14631839])
```

This tells you that 84.2% of the dataset's variance lies along the first axis, and 14.6% lies along the second axis. This leaves less than 1.2% for the third axis, so it is reasonable to assume that it probably carries little information.

Choosing the Right Number of Dimensions

- Instead of arbitrarily choosing the number of dimensions to reduce down to, it is generally preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%)
- The following code computes PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the training set's variance:

```
pca = PCA()  
pca.fit(X)  
cumsum = np.cumsum(pca.explained_variance_ratio_)  
d = np.argmax(cumsum >= 0.95) + 1
```

- However, there is a much better option: instead of specifying the number of principal components you want to preserve, you can set `n_components` to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

```
pca = PCA(n_components=0.95)  
X_reduced = pca.fit_transform(X)
```

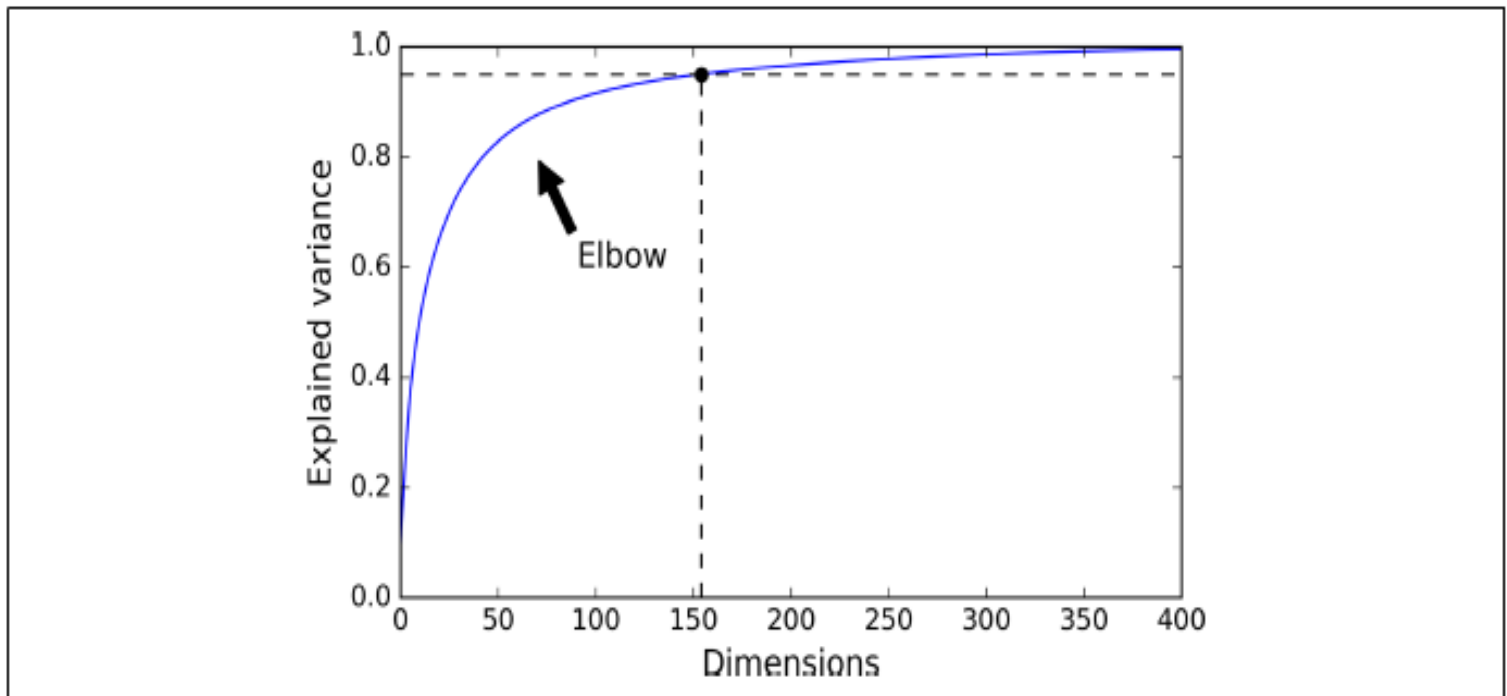


Figure 8-8. Explained variance as a function of the number of dimensions

PCA for Compression

- Obviously after dimensionality reduction, the training set takes up much less space. For example, try applying PCA to the MNIST dataset while preserving 95% of its variance.
- You should find that each instance will have just over 150 features, instead of the original 784 features. So while most of the variance is preserved, the dataset is now less than 20% of its original size.
- It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. Of course this won't give you back the original data but it will likely be quite close to the original data.
- The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the *reconstruction error*.

```
pca = PCA(n_components = 154)
X_mnist_reduced = pca.fit_transform(X_mnist)
X_mnist_recovered = pca.inverse_transform(X_mnist_reduced)
```

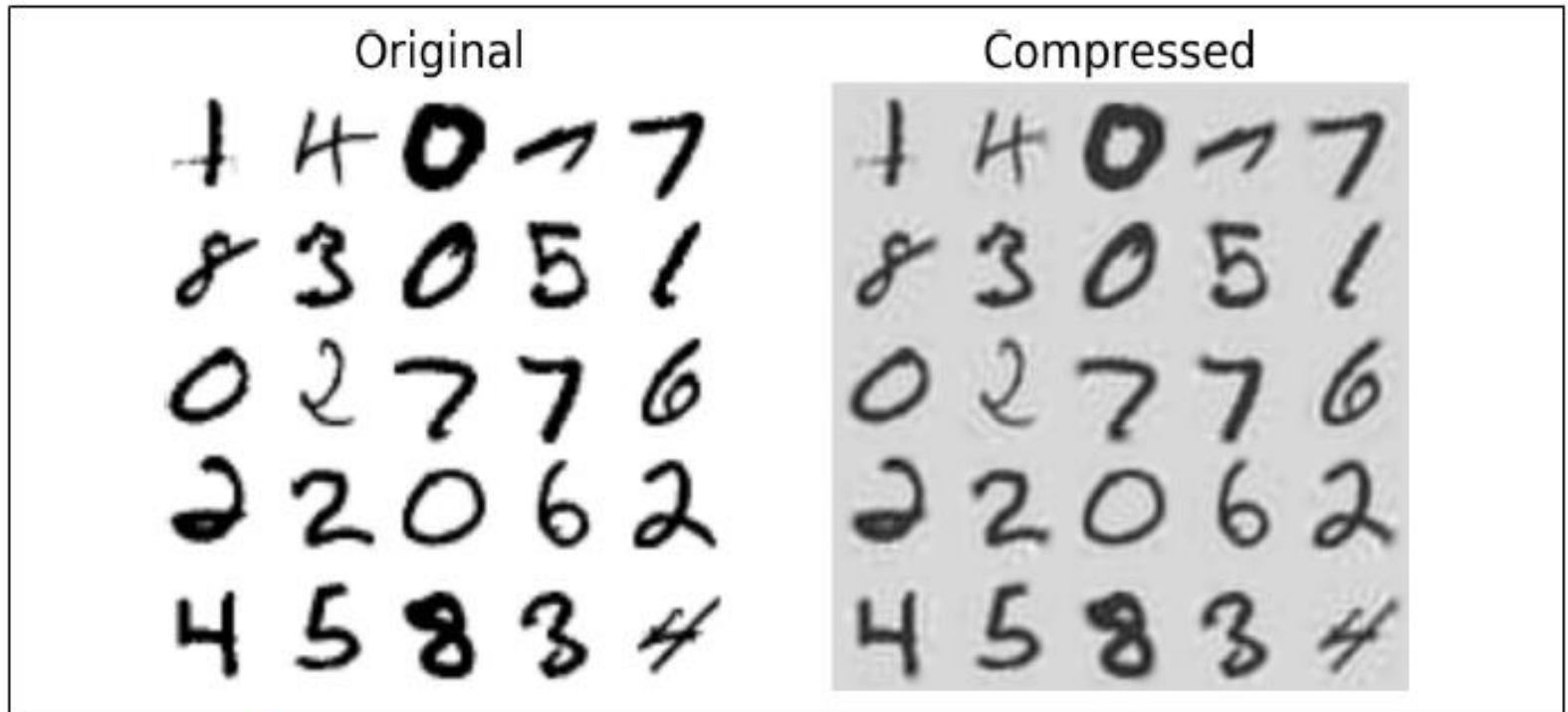


Figure 8-9. MNIST compression preserving 95% of the variance


The equation of the inverse transformation is shown in **Equation 8-3**.

Equation 8-3. PCA inverse transformation, back to the original number of dimensions

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}} \cdot \mathbf{W}_d^T$$

Incremental PCA

- One problem with the implementation of PCA is that it requires the whole training set to fit in memory in order for the SVD algorithm to run.
- *Incremental PCA (IPCA) algorithms have been developed to split the training set into mini-batches and feed an IPCA algorithm one mini-batch at a time.*
- This is useful for large training sets, and also to apply PCA online.
- The following code splits the MNIST dataset into 100 mini-batches (using NumPy's `array_split()` function) and feeds them to Scikit-Learn's `IncrementalPCA` class to reduce the dimensionality of the MNIST dataset down to 154 dimensions.
- Note that you must call the `partial_fit()` method with each mini-batch rather than the `fit()` method with the whole training set



```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_mnist, n_batches):
    inc_pca.partial_fit(X_batch)

X_mnist_reduced = inc_pca.transform(X_mnist)
```

- Alternatively, you can use NumPy's memmap class, which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory .The class loads only the data it needs in memory, when it needs it.
- Since the IncrementalPCA class uses only a small part of the array at any given time, the memory usage remains under control.

```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))  
  
batch_size = m // n_batches  
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)  
inc_pca.fit(X_mm)
```


Randomized PCA

- Scikit-Learn offers yet another option to perform PCA, called *Randomized PCA*
- It is dramatically faster than the previous algorithms when d is *much* smaller.

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")  
X_reduced = rnd_pca.fit_transform(X_mnist)
```

Kernel PCA

- The kernel trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called *Kernel PCA (kPCA)*

```
from sklearn.decomposition import KernelPCA
```

```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)  
X_reduced = rbf_pca.fit_transform(X)
```