Einführung in ML

M.Heinz

Kap 2 - 7

23.9.2024

Kap 2 : End-to-end project

Welches sind die Schritte in einem Projekt

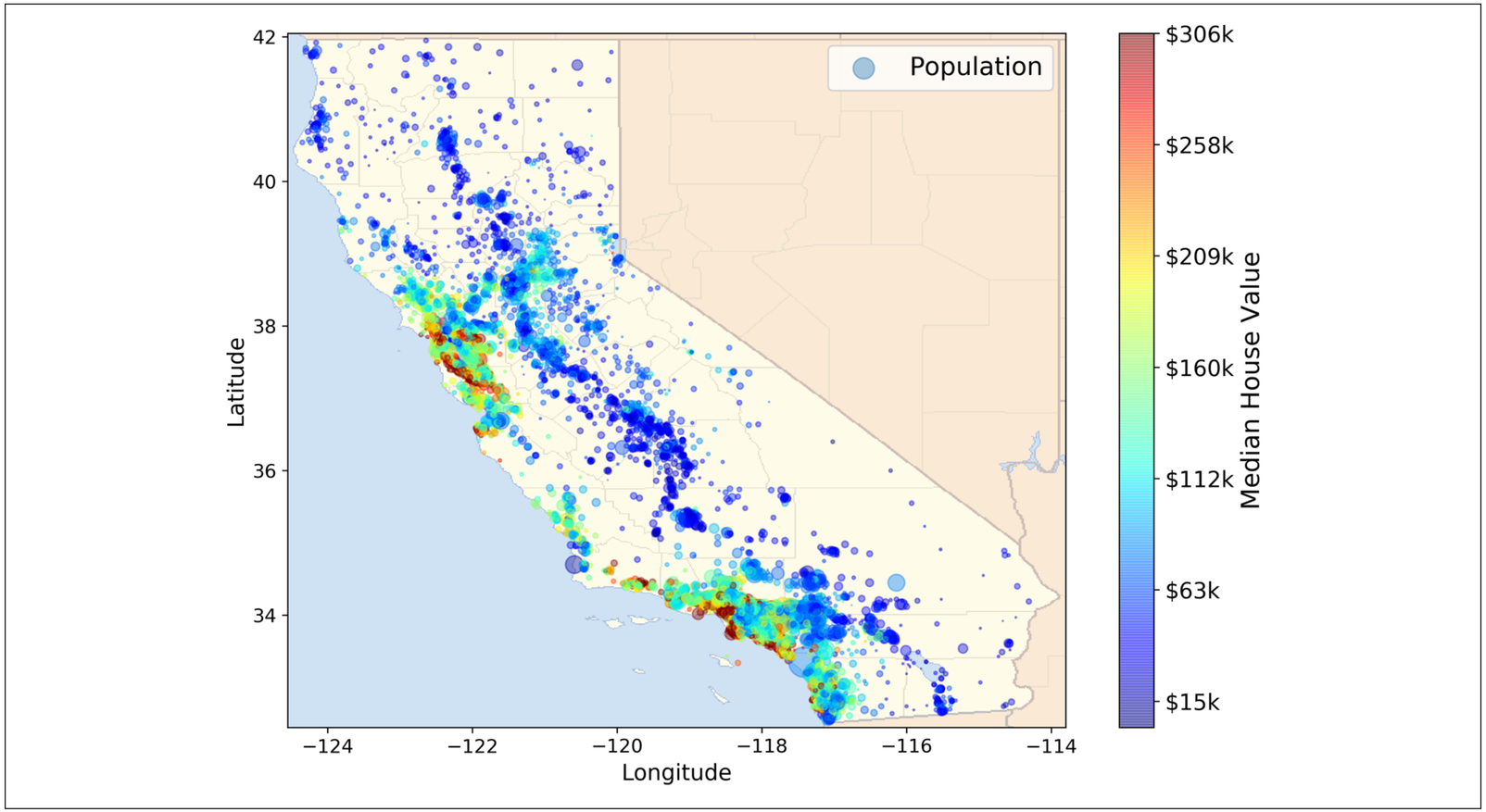


Figure 2-1. California housing prices

Alle Schritte auf einem Blick

- Look at the Big Picture
  - Frame the Problem
  - Select a Performance Measure
  - Check the Assumptions
- Get the Data
  - Create the Workspace
  - Download the Data
  - Take a Quick Look at the Data Structure
  - Create a Test Set
- Discover and Visualize the Data to Gain Insights
  - Visualizing Geographical Data
  - Looking for Correlations
  - Experimenting with Attribute Combinations
- Prepare the Data for Machine Learning Algorithms
  - Data Cleaning
  - Handling Text and Categorical Attributes
  - Custom Transformers
  - Feature Scaling
  - Transformation Pipelines
- Select and Train a Model
  - Training and Evaluating on the Training Set
  - Better Evaluation Using Cross-Validation
- Fine-Tune Your Model
  - Grid Search
  - Randomized Search
  - Ensemble Methods
  - Analyze the Best Models and Their Errors
  - Evaluate Your System on the Test Set
- Launch, Monitor, and Maintain Your System

Korrelationen

Kap 2 / S. 60

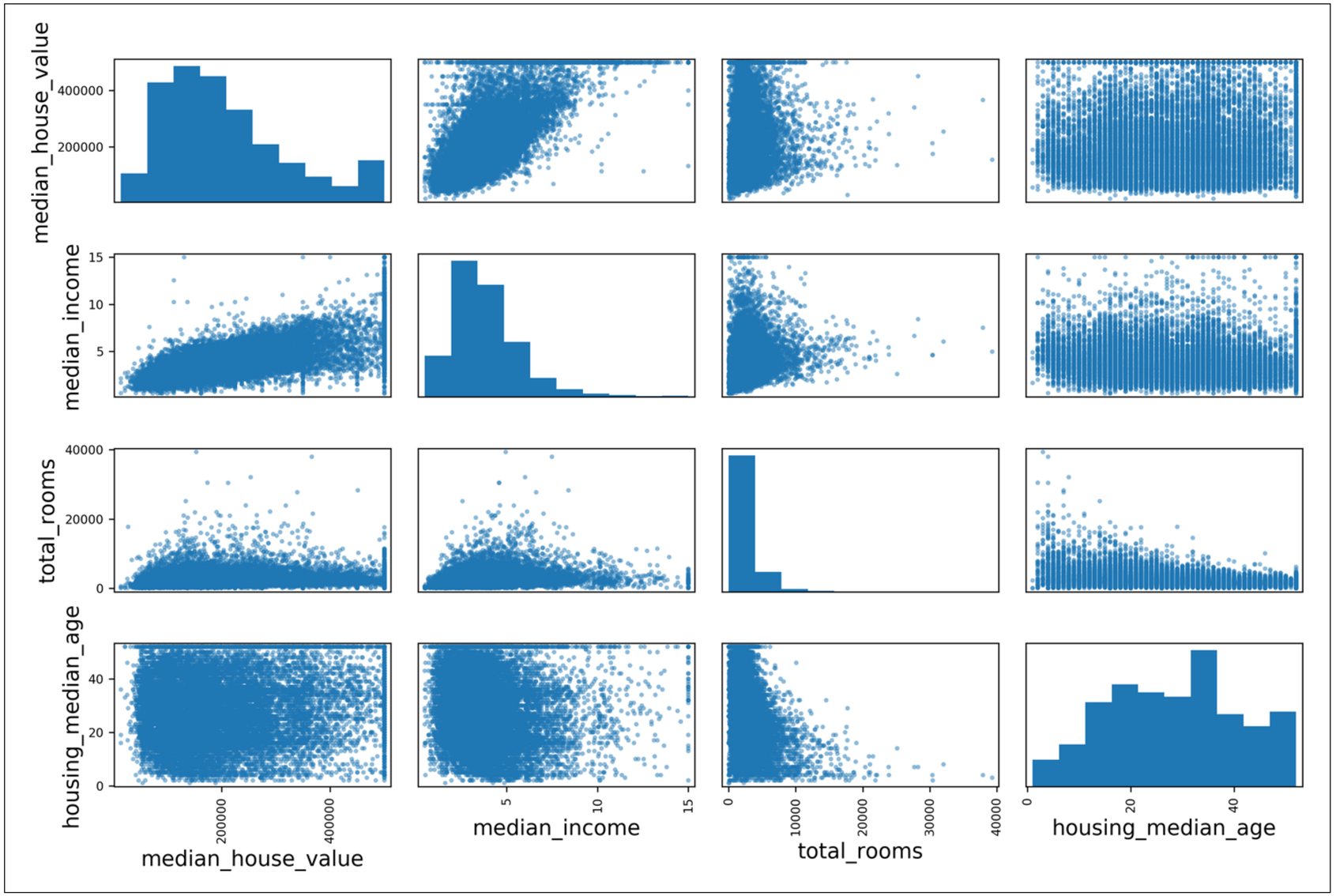


Figure 2-15. This scatter matrix plots every numerical attribute against every other numerical attribute, plus a histogram of each numerical attribute

Classification

Kap 3 / MNIST

Figure 3-1. Digits from the MNIST dataset

Classifiers

Die wichtigsten Definitionen

Training a Binary Classifier

Let's simplify the problem for now and only try to identify one digit—for example, the number 5. This "5-detector" will be an example of a *binary classifier*, capable of distinguishing between just two classes, 5 and not-5. Let's create the target vectors for this classification task:

Multiclass Classification

Whereas binary classifiers distinguish between two classes, *multiclass classifiers* (also called *multinomial classifiers*) can distinguish between more than two classes.

Some algorithms (such as SGD classifiers, Random Forest classifiers, and naive Bayes classifiers) are capable of handling multiple classes natively. Others (such as Logistic Regression or Support Vector Machine classifiers) are strictly binary classifiers. However, there are various strategies that you can use to perform multiclass classification with multiple binary classifiers.

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-the-rest* (OvR) strategy (also called *one-versus-all*).

Performance: Confusion Matrix

Wie werden Fehler klassifiziert…?

Equation 3-1. Precision

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

Equation 3-2. Recall

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

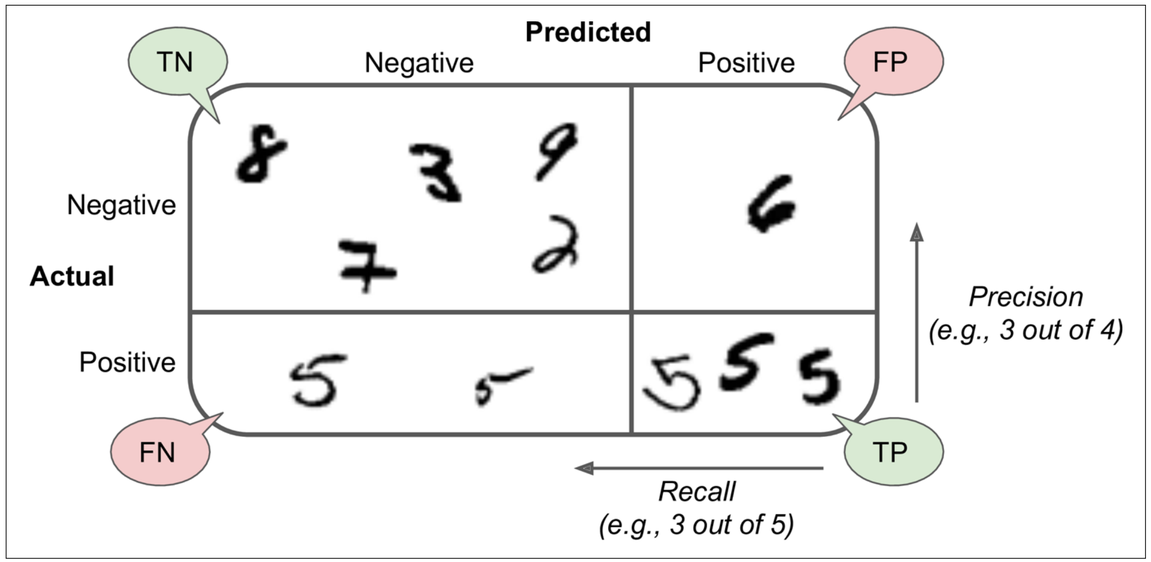


Figure 3-2. An illustrated confusion matrix shows examples of true negatives (top left), false positives (top right), false negatives (lower left), and true positives (lower right)

```
>>> 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([[5578,    0,   22,    7,    8,   45,   35,    5,  222,    1],
       [   0, 6410,   35,   26,    4,   44,    4,    8,  198,   13],
       [  28,   27, 5232,  100,   74,   27,   68,   37,  354,   11],
       [  23,   18,  115, 5254,    2,  209,   26,   38,  373,   73],
       [  11,   14,   45,   12, 5219,   11,   33,   26,  299,  172],
       [  26,   16,   31,  173,   54, 4484,   76,   14,  482,   65],
       [  31,   17,   45,    2,   42,   98, 5556,    3,  123,    1],
       [  20,   10,   53,   27,   50,   13,    3, 5696,  173,  220],
       [  17,   64,   47,   91,    3,  125,   24,   11, 5421,   48],
       [  24,   18,   29,   67,  116,   39,    1,  174,  329, 5152]])
```

Recall / Precision Tradeoff

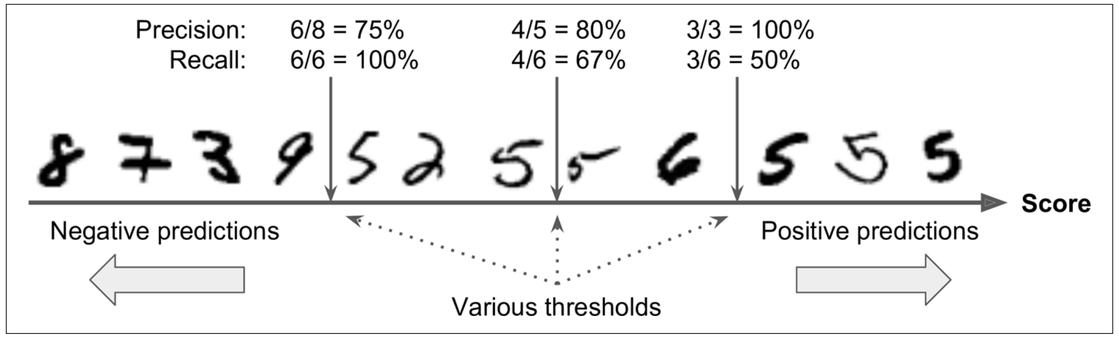


Figure 3-3. In this precision/recall trade-off, images are ranked by their classifier score, and those above the chosen decision threshold are considered positive; the higher the threshold, the lower the recall, but (in general) the higher the precision

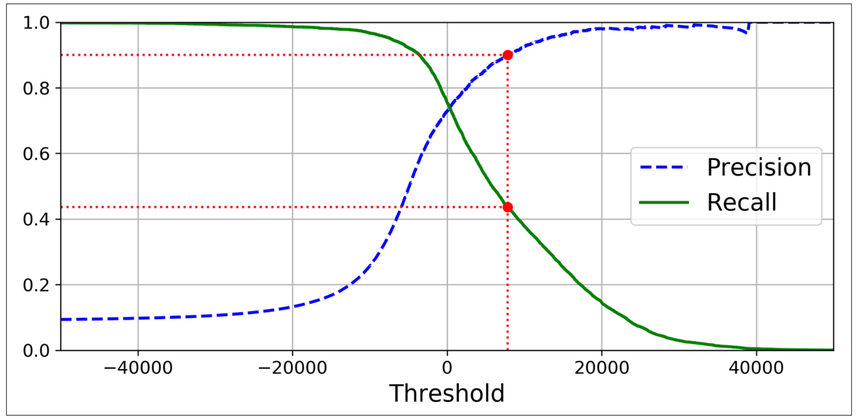


Figure 3-4. Precision and recall versus the decision threshold

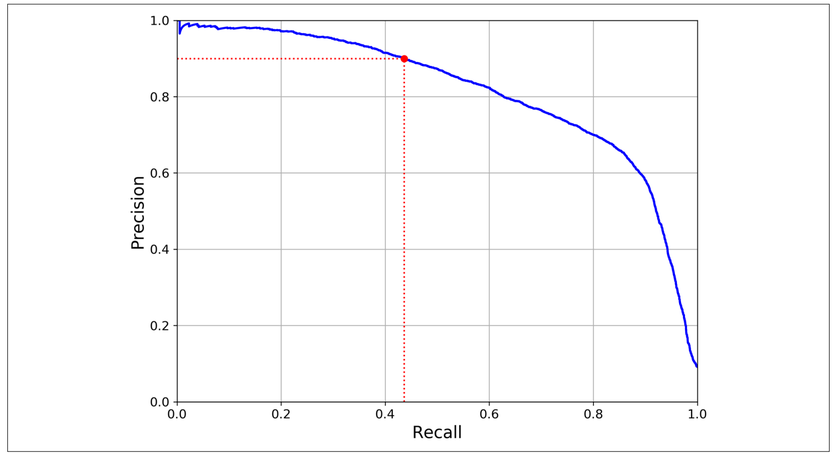


Figure 3-5. Precision versus recall

Linear Regression (Kap 4)

…lineare Regression

Equation 4-1. Linear Regression model prediction

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

In this equation:

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

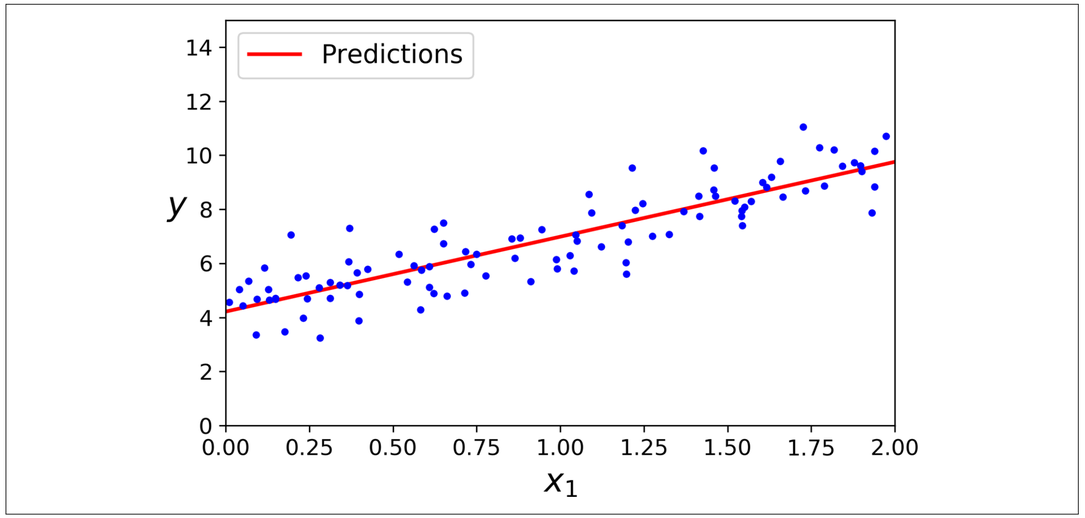


Figure 4-2. Linear Regression model predictions

ROC-Curve

Receiver Operating Characteristic

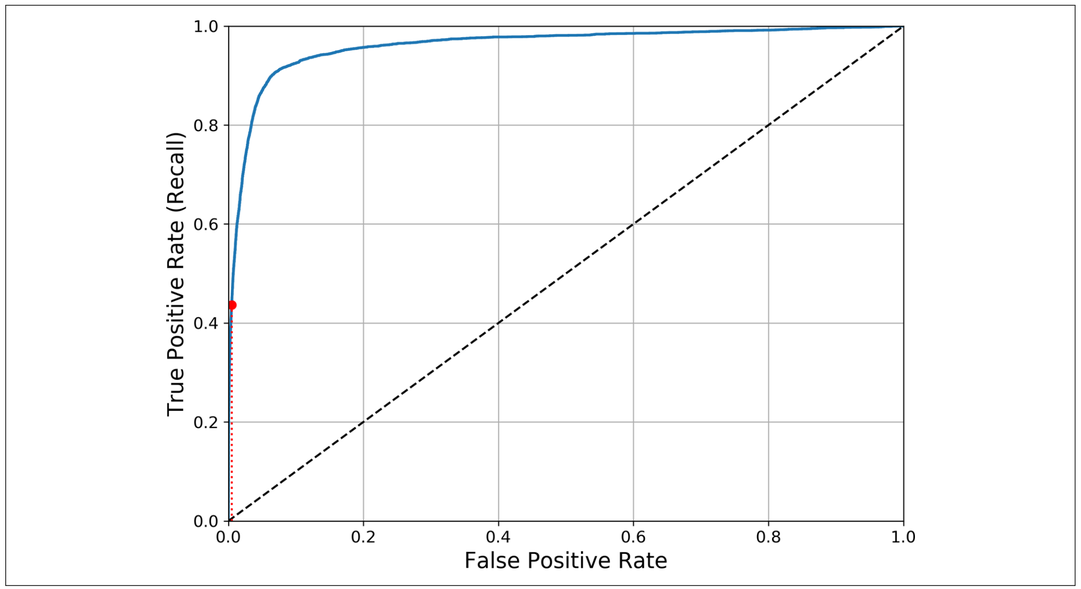


Figure 3-6. This ROC curve plots the false positive rate against the true positive rate for all possible thresholds; the red circle highlights the chosen ratio (at 43.68% recall)

```python
from sklearn.metrics import roc_curve

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


def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # Dashed diagonal
    [...] # Add axis labels and grid

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

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.9611778893101814
```

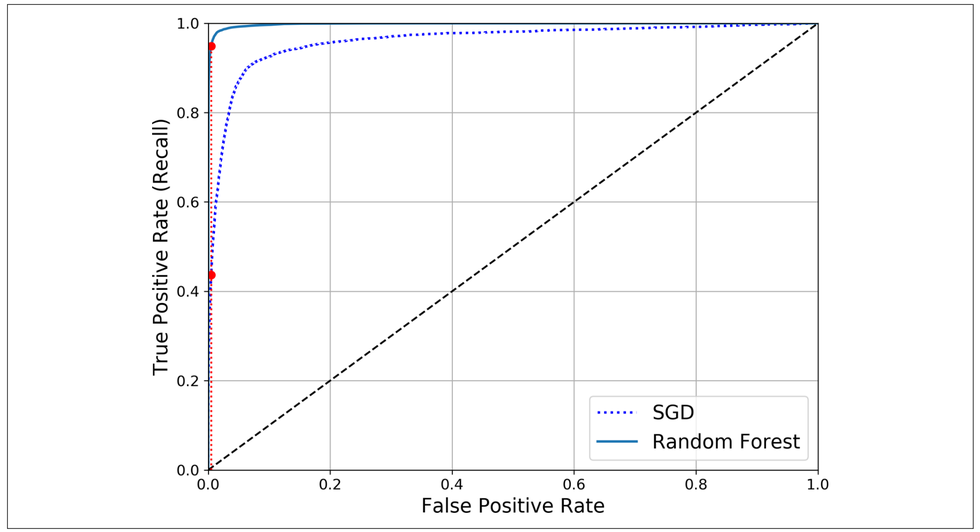


Figure 3-7. Comparing ROC curves: the Random Forest classifier is superior to the SGD classifier because its ROC curve is much closer to the top-left corner, and it has a greater AUC

Gradient Descent

Kap 4 : wie finden Modelle ihre besten Wert ?

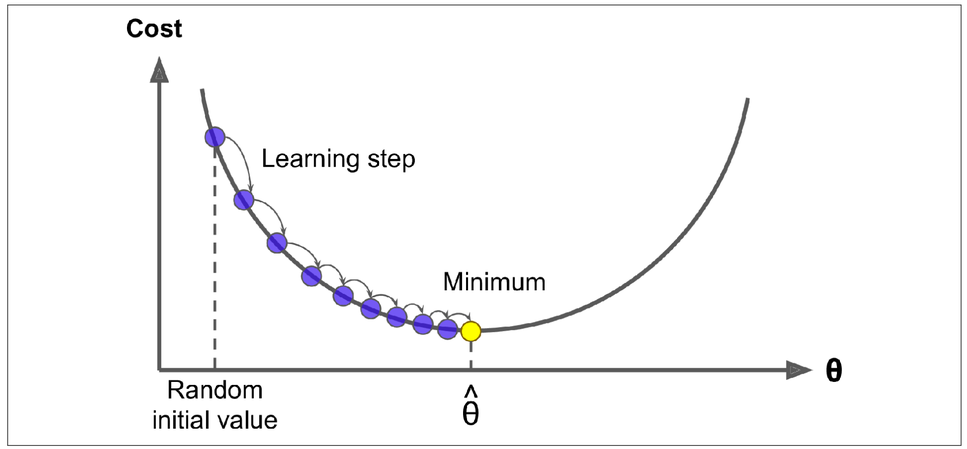


Figure 4-3. In this depiction of Gradient Descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the parameters approach the minimum

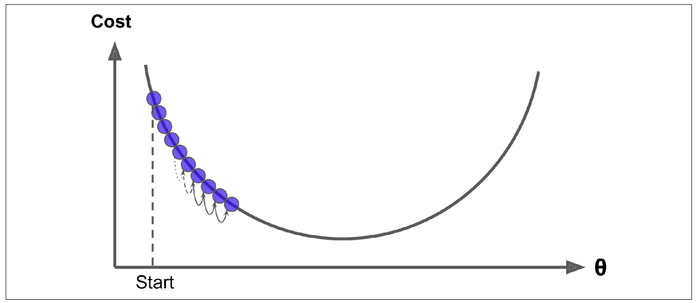


Figure 4-4. The learning rate is too small

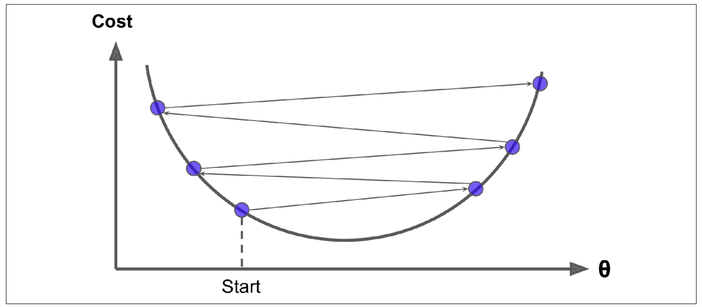


Figure 4-5. The learning rate is too large

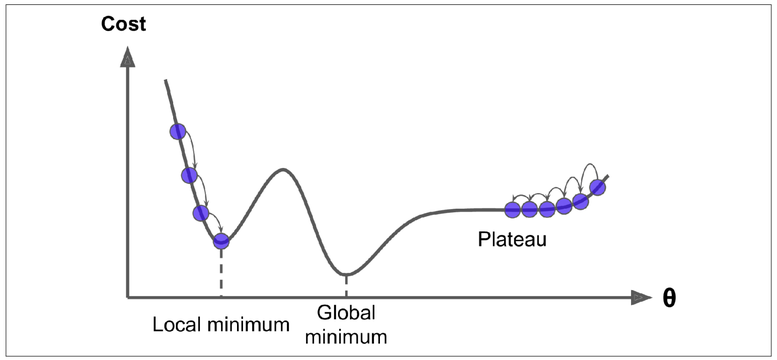


Figure 4-6. Gradient Descent pitfalls

Stochastic Gradient Descent

Kap 4

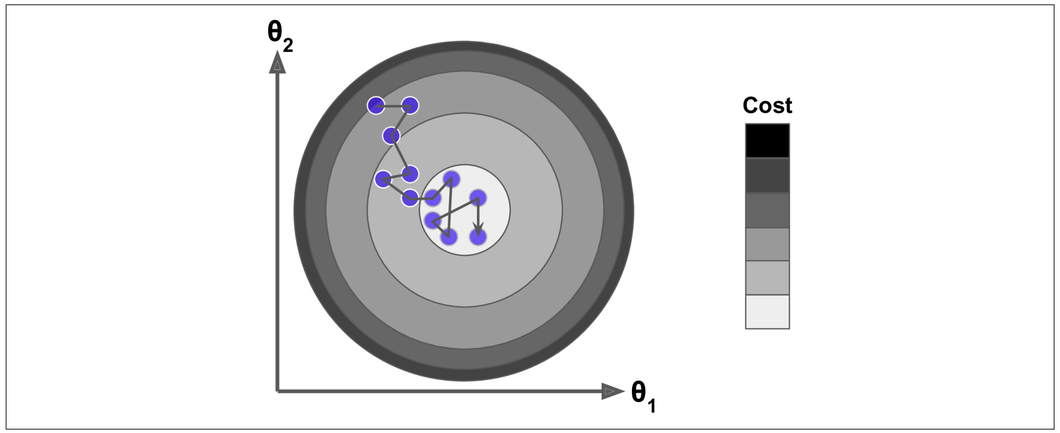


Figure 4-9. With Stochastic Gradient Descent, each training step is much faster but also much more stochastic than when using Batch Gradient Descent

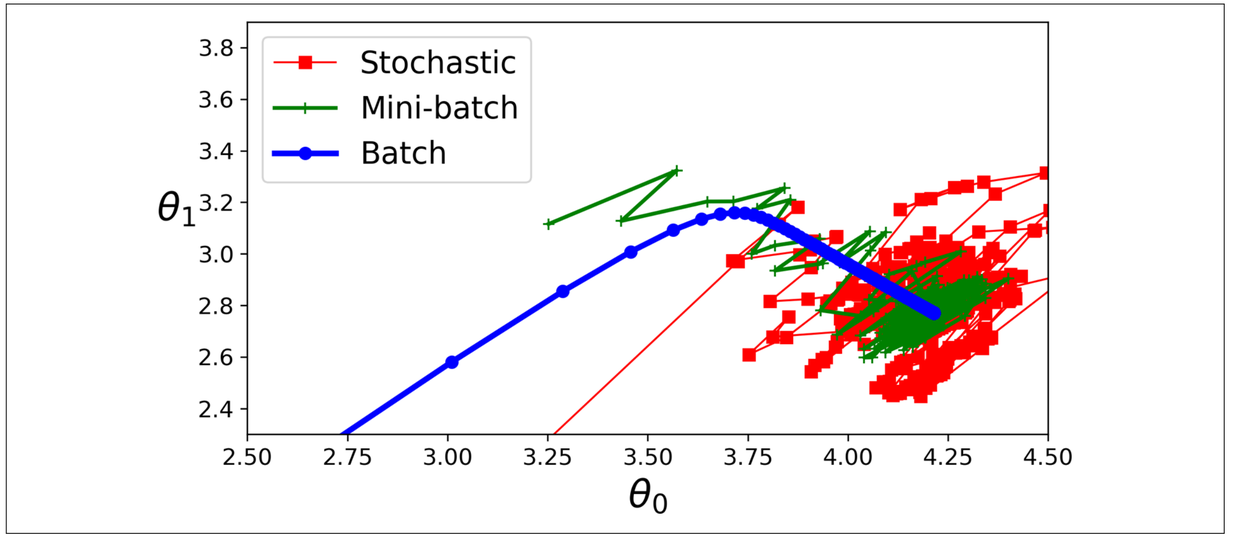


Figure 4-11. Gradient Descent paths in parameter space

Logistic Regression

What is it ?

As we discussed in Chapter 1, some regression algorithms can be used for classification (and vice versa). *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"), and otherwise it predicts that it does not (i.e., it belongs to the *negative class*, labeled "0"). This makes it a binary classifier.

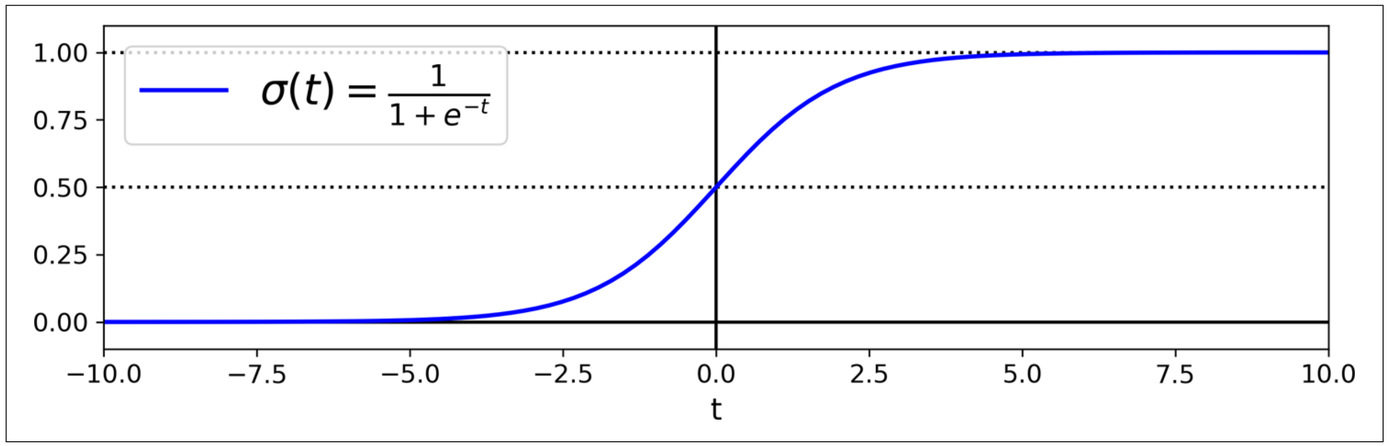


Figure 4-21. Logistic function

Equation 4-15. Logistic Regression model prediction

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

Logistic Regression

Cost function

Now you know how a Logistic Regression model estimates probabilities and makes predictions. But how is it trained? The objective of training is to set the parameter vector $\boldsymbol{\theta}$ so that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$). This idea is captured by the cost function shown in Equation 4-16 for a single training instance $\mathbf{x}$.

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

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

The cost function over the whole training set is the average cost over all training instances. It can be written in a single expression called the *log loss*, shown in Equation 4-17.

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

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

Logistic Regression

Kap 4

Decision Boundaries

Let's use the iris dataset to illustrate Logistic Regression. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: *Iris setosa*, *Iris versicolor*, and *Iris virginica* (see Figure 4-22).

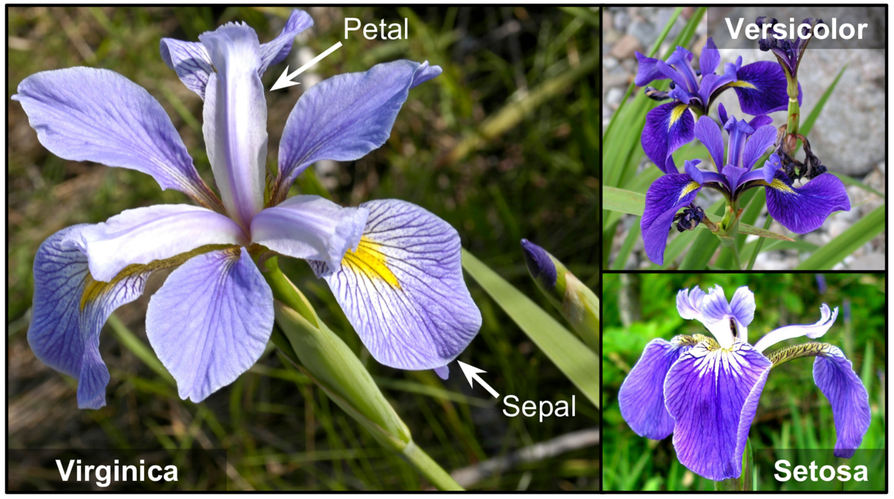


Figure 4-22. Flowers of three iris plant species[14]

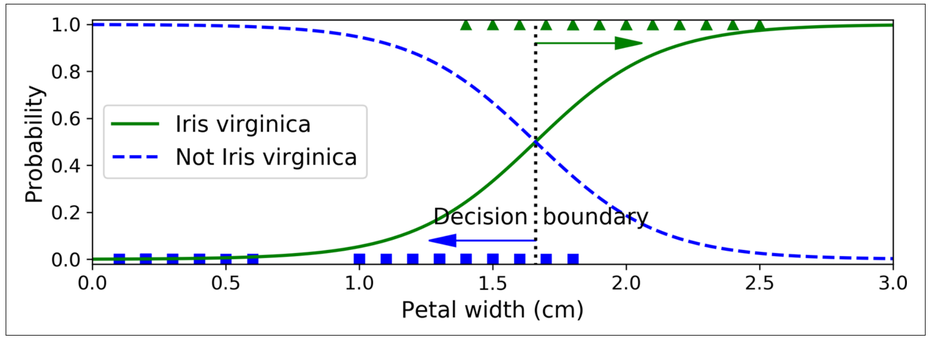


Figure 4-23. Estimated probabilities and decision boundary

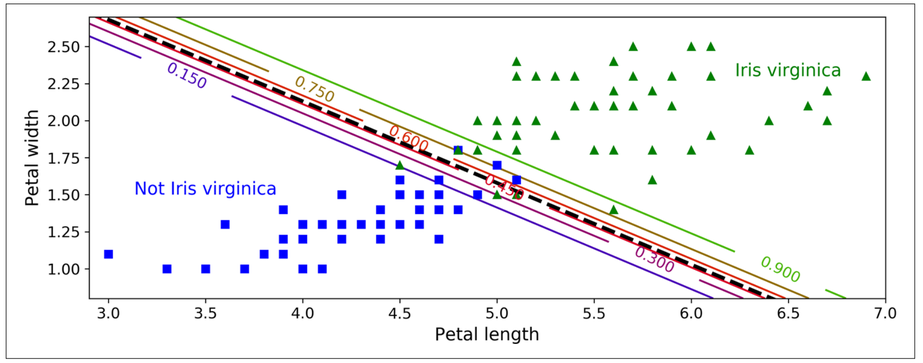


Figure 4-24. Linear decision boundary

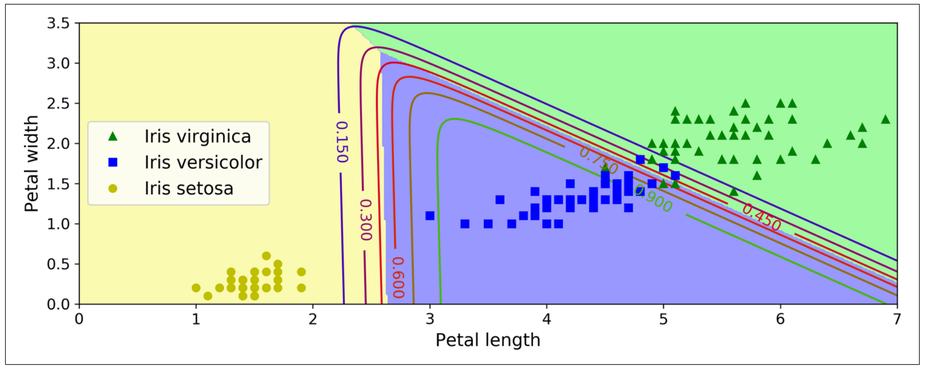


Figure 4-25. Softmax Regression decision boundaries

Decision Trees & Random Forest

Kap 5/6

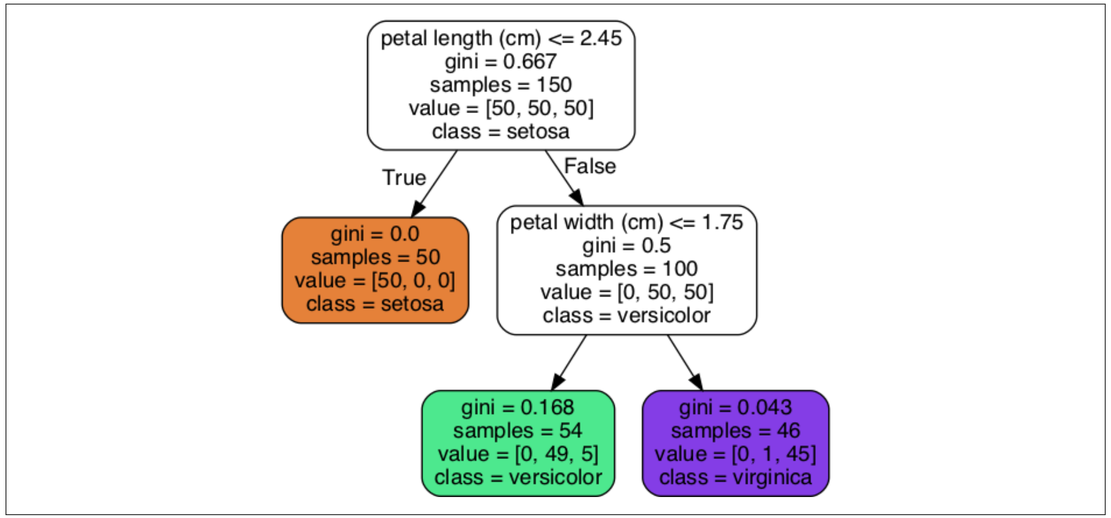


Figure 6-1. Iris Decision Tree

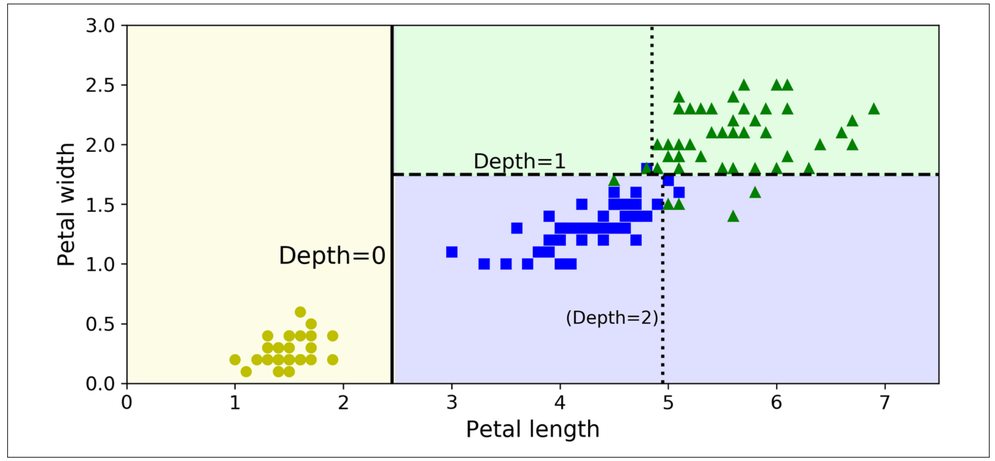


Figure 6-2. Decision Tree decision boundaries

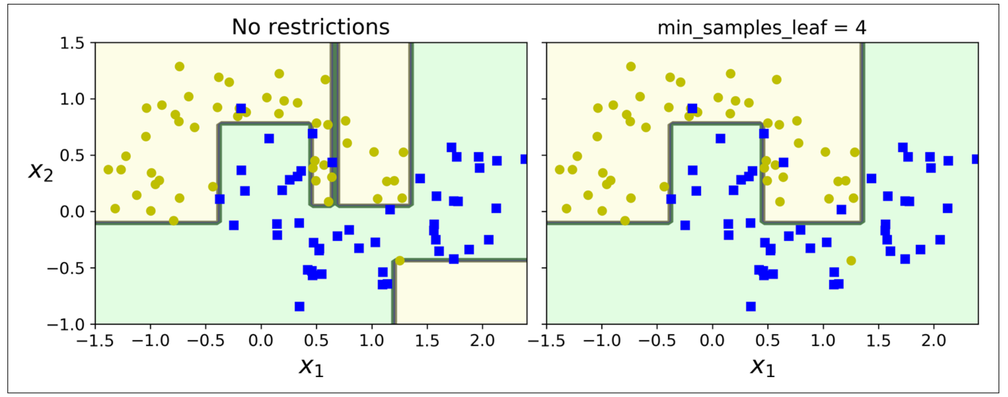


Figure 6-3. Regularization using `min_samples_leaf`

Performance of Decision Trees

Definition: GINI impurity and CART algorithm

A node's `samples` attribute counts how many training instances it applies to. For example, 100 training instances have a petal length greater than 2.45 cm (depth 1, right), and of those 100, 54 have a petal width smaller than 1.75 cm (depth 2, left). A node's `value` attribute tells you how many training instances of each class this node applies to: for example, the bottom-right node applies to 0 *Iris setosa*, 1 *Iris versicolor*, and 45 *Iris virginica*. Finally, a node's `gini` attribute measures its *impurity*: a node is "pure" (`gini=0`) if all training instances it applies to belong to the same class. For example, since the depth-1 left node applies only to *Iris setosa* training instances, it is pure and its `gini` score is 0. Equation 6-1 shows how the training algorithm computes the `gini` score G_i of the i^{th} node. The depth-2 left node has a `gini` score equal to $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$.

Equation 6-1. Gini impurity

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

Scikit-Learn uses the *Classification and Regression Tree* (CART) algorithm to train Decision Trees (also called "growing" trees). The algorithm works by first splitting the training set into two subsets using a single feature k and a threshold t_k (e.g., "petal length ≤ 2.45 cm"). How does it choose k and t_k? It searches for the pair (k, t_k) that produces the purest subsets (weighted by their size). Equation 6-2 gives the cost function that the algorithm tries to minimize.

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

More on Performance of Dec. trees

Gini vs 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"`. The concept of entropy originated in thermodynamics as a measure of molecular disorder: entropy approaches zero when molecules are still and well ordered. Entropy later spread to a wide variety of domains, including Shannon's *information theory*, where it measures the average information content of a message:[4] entropy is zero when all messages are identical. In Machine Learning, entropy is frequently used as an impurity measure: a set's entropy is zero when it contains instances of only one class. Equation 6-3 shows the definition of the entropy of the i^{th} node. For example, the depth-2 left node in Figure 6-1 has an entropy equal to $-(49/54)\log_2(49/54) - (5/54)\log_2(5/54) \approx 0.445$.

Equation 6-3. Entropy

$$H_i = -\sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log_2\left(p_{i,k}\right)$$

So, should you use Gini impurity or entropy? The truth is, most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.[5]

Random Forest Classifiers

Bagging and Pasting

Random Forests

As we have discussed, a Random Forest[9] is an ensemble of Decision Trees, generally trained via the bagging method (or sometimes pasting), typically with `max_samples` set to the size of the training set. 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[10] (similarly, there is

One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed. Another approach is to use the same training algorithm for every predictor and train them on different random subsets of the training set. When sampling is performed *with* replacement, this method is called *bagging*[1] (short for *bootstrap aggregating*[2]). When sampling is performed *without* replacement, it is called *pasting*.[3]

In other words, both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor. This sampling and training process is represented in Figure 7-4.

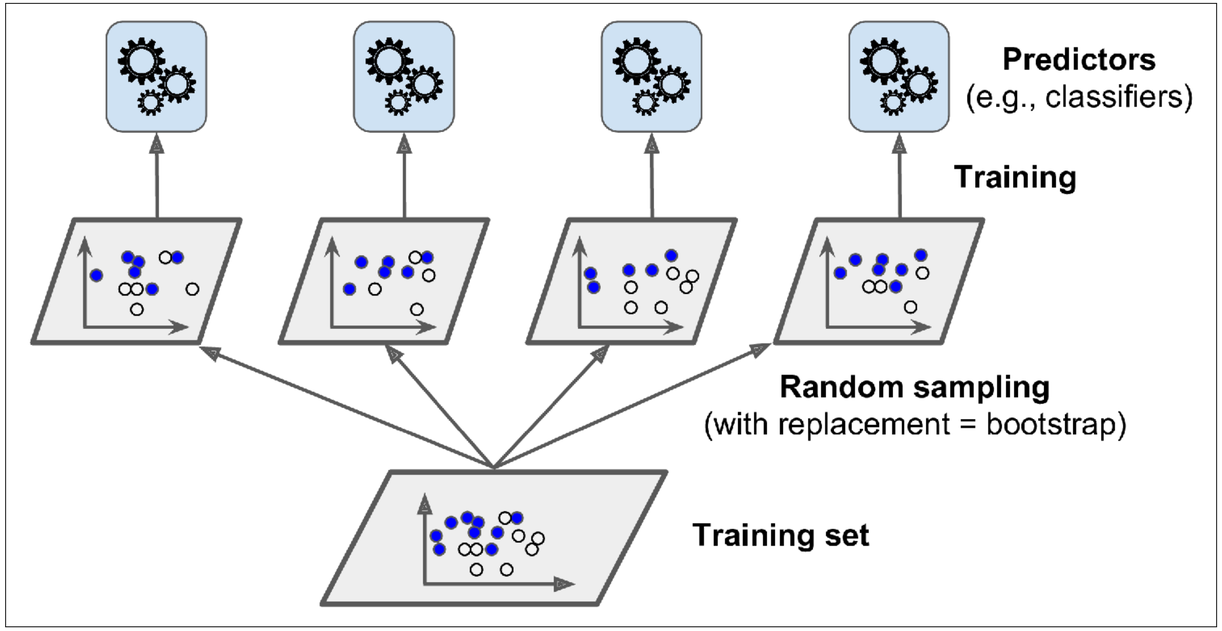


Figure 7-4. Bagging and pasting involves training several predictors on different random samples of the training set

Random Forest Classifier

Option: Bagging

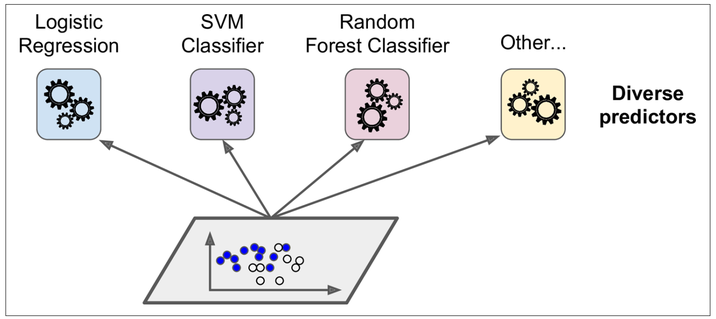


Figure 7-1. Training diverse classifiers

The Random Forest algorithm introduces extra randomness when growing trees; instead of searching for the very best feature when splitting a node (see Chapter 6), it searches for the best feature among a random subset of features. The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally yielding an overall better model. The following `BaggingClassifier` is roughly equivalent to the previous `RandomForestClassifier`:

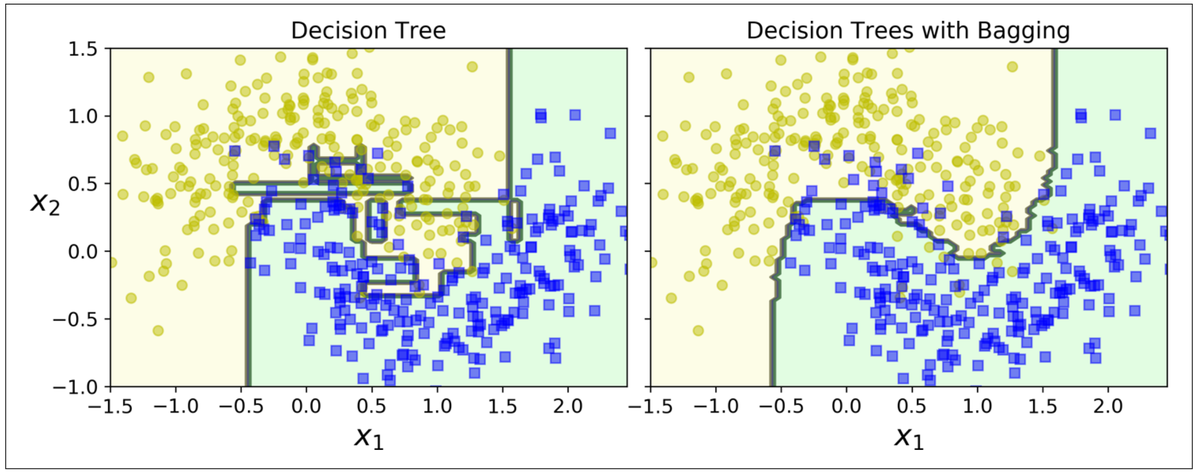


Figure 7-5. A single Decision Tree (left) versus a bagging ensemble of 500 trees (right)

Figure 7-5 compares the decision boundary of a single Decision Tree with the decision boundary of a bagging ensemble of 500 trees (from the preceding code), both trained on the moons dataset. As you can see, the ensemble's predictions will likely generalize much better than the single Decision Tree's predictions: the ensemble has a comparable bias but a smaller variance (it makes roughly the same number of errors on the training set, but the decision boundary is less irregular).

Random Forrest (cont)

Option; Boosting

Boosting (originally called *hypothesis boosting*) refers to any Ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. There are many boosting methods available, but by far the most popular are

AdaBoost[13] (short for *Adaptive Boosting*) and *Gradient Boosting*. Let's start with AdaBoost.

AdaBoost

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost.

For example, when training an AdaBoost classifier, the algorithm first trains a base classifier (such as a Decision Tree) and uses it to make predictions on the training set. The algorithm then increases the relative weight of misclassified training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on (see Figure 7-7).

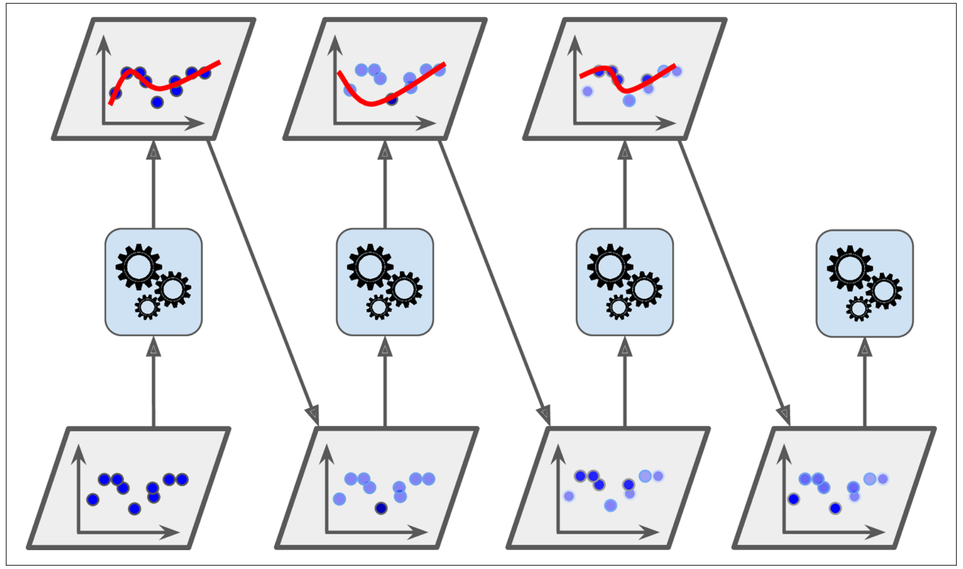


Figure 7-7. AdaBoost sequential training with instance weight updates

KNN example Chapter 3

3Blue 1 Brown

What is NN: https://www.youtube.com/watch?v=aircAruvnKk

Chapter 11 : Deep Neural Networks

M.Heinz

24.9.24

Probleme beim DNN

- You may be faced with the tricky *vanishing gradients* problem or the related *exploding gradients* problem. This is when the gradients grow smaller and smaller, or larger and larger, when flowing backward through the DNN during training. Both of these problems make lower layers very hard to train.
- You might not have enough training data for such a large network, or it might be too costly to label.
- Training may be extremely slow.
- A model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances or if they are too noisy.

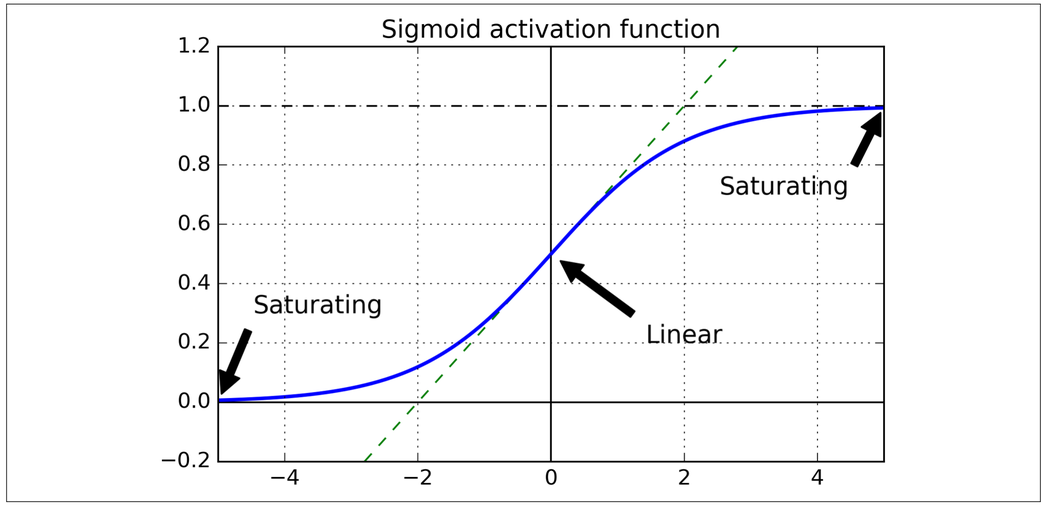


Figure 11-1. Logistic activation function saturation

Activation functions

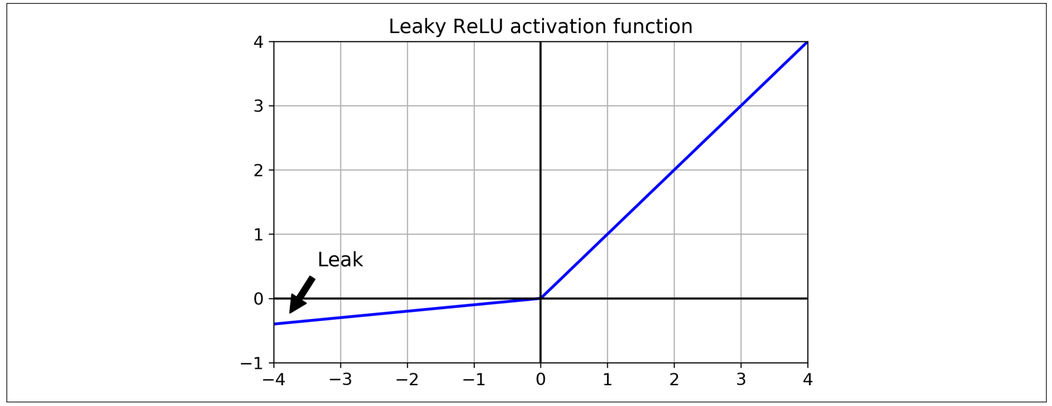


Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values

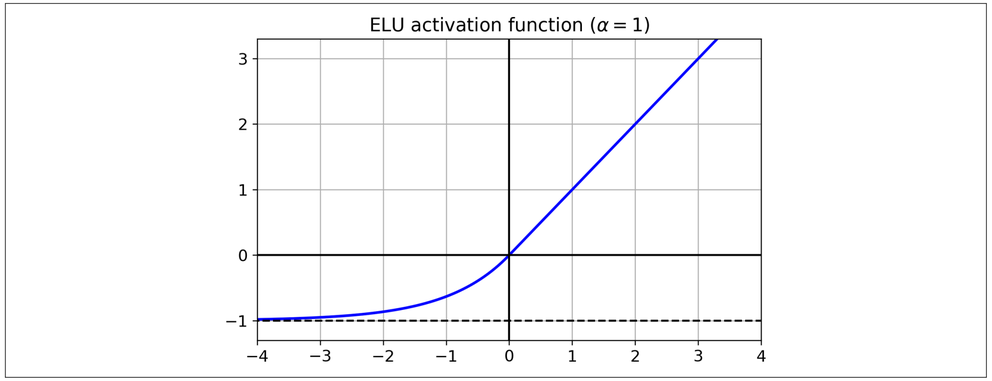


Figure 11-3. ELU activation function

Faster Optimizers

Faster Optimizers

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, using a good activation function, using Batch Normalization, and reusing parts of a pretrained network (possibly built on an auxiliary task or using unsupervised learning). Another huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer. In this section we will present the most popular algorithms: momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam and Nadam optimization.

Ada Grad Optimizer

AdaGrad

Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it very slowly goes down to the bottom of the valley. It would be nice if the algorithm could correct its direction earlier to point a bit more toward the global optimum. The *AdaGrad* algorithm[15] achieves this correction by scaling down the gradient vector along the steepest dimensions (see Equation 11-6).

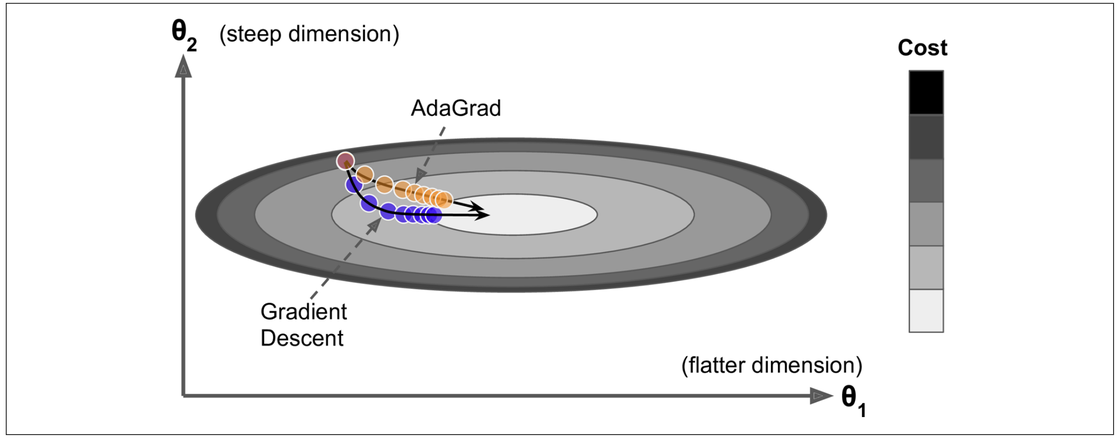


Figure 11-7. AdaGrad versus Gradient Descent: the former can correct its direction earlier to point to the optimum

Batch Normalisation

In a 2015 paper,[8] Sergey Ioffe and Christian Szegedy proposed a technique called *Batch Normalization* (BN) that addresses these problems. The technique consists of adding an operation in the model just before or after the activation function of each hidden layer. This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs. In many cases, if you add a BN layer as the very first layer of your neural network, you do not need to standardize your training set (e.g., using a `StandardScaler`); the BN layer will do it for you (well, approximately, since it only looks at one batch at a time, and it can also rescale and shift each input feature).

Reinforcement Learning

Kap 18

M.Heinz

Reinforcement Learning idea

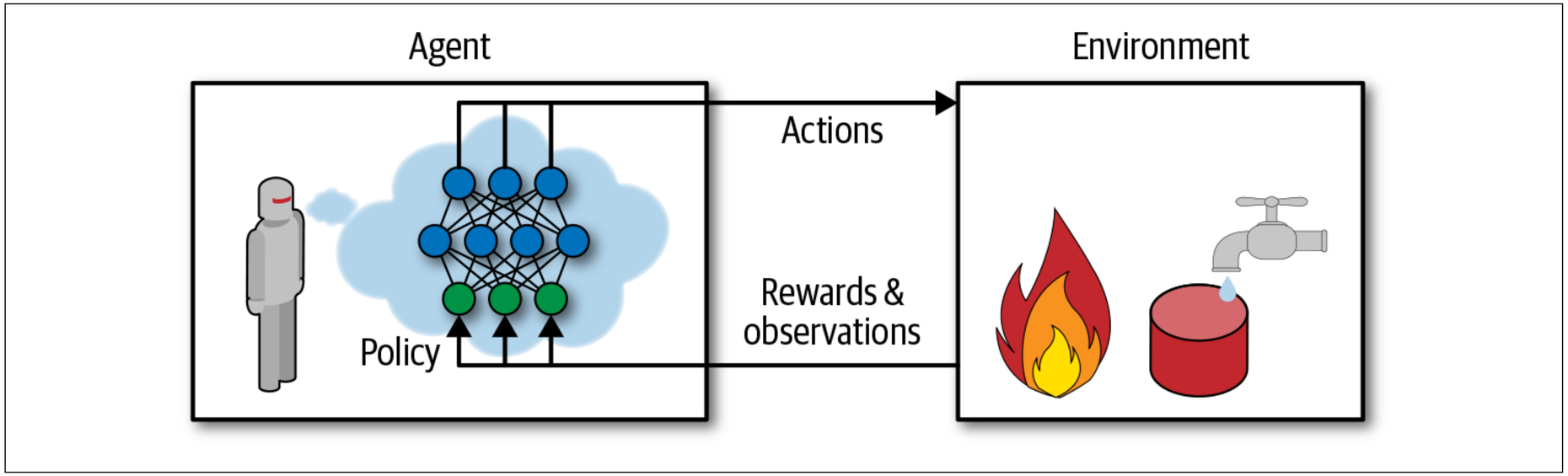


Figure 18-2. Reinforcement Learning using a neural network policy

How to implement a policy in a NN

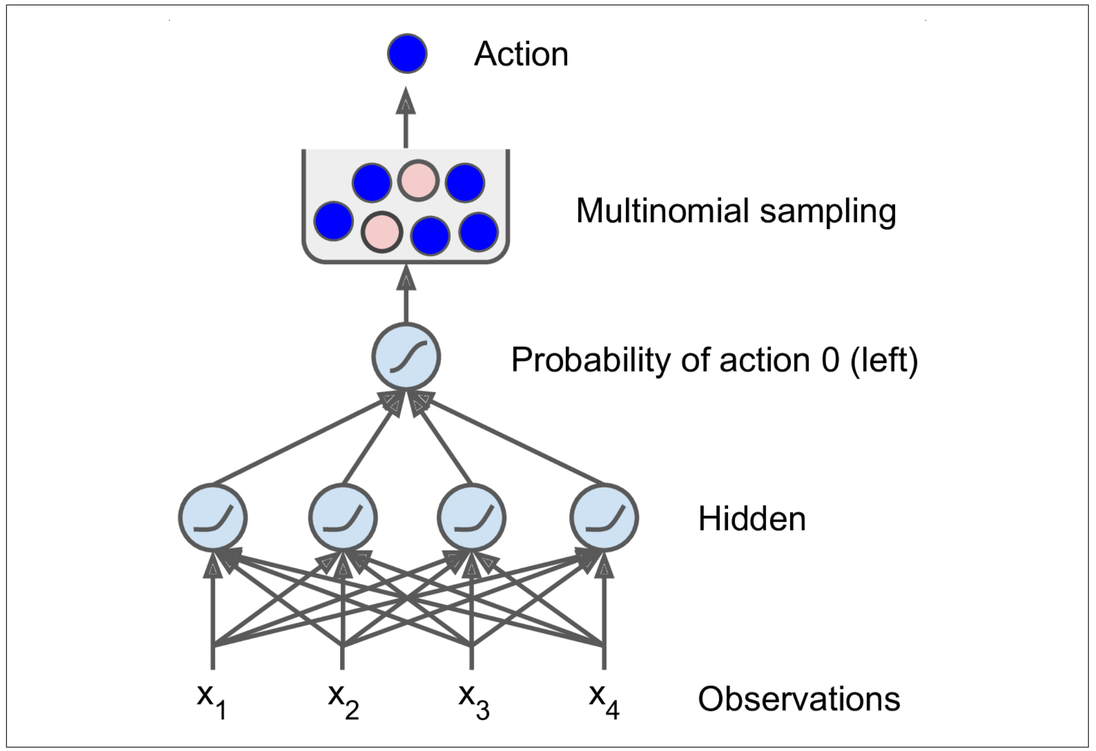


Figure 18-5. Neural network policy

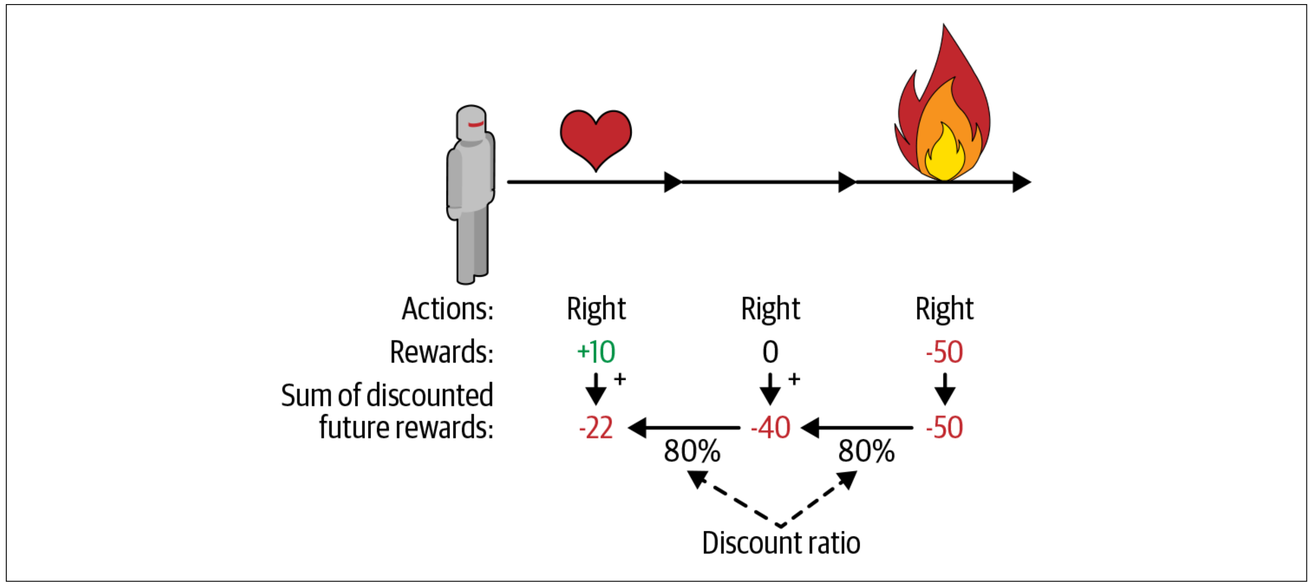


Figure 18-6. Computing an action's return: the sum of discounted future rewards

Markov chain and Markov process

Markov Decision Process (MDP- Richard Bellman)

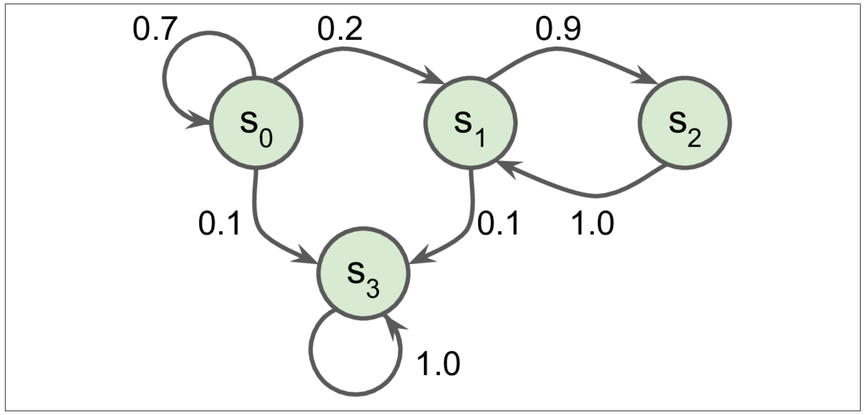


Figure 18-7. Example of a Markov chain

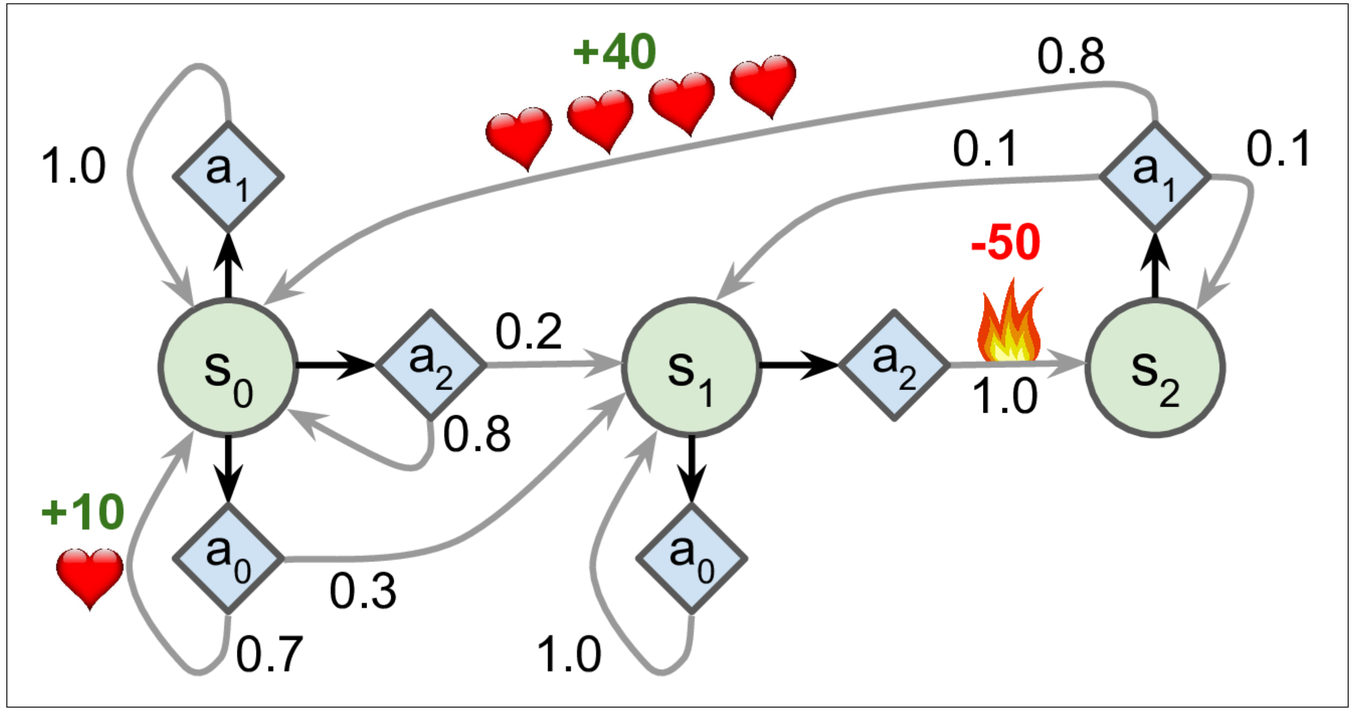


Figure 18-8. Example of a Markov decision process

Q-Value

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not give us the optimal policy for the agent. Luckily, Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-Values* (Quality Values). The optimal Q-Value of the state-action pair (s, a), noted $Q^*(s, a)$, is the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a, but before it sees the outcome of this action, assuming it acts optimally after that action.

Here is how it works: once again, you start by initializing all the Q-Value estimates to zero, then you update them using the *Q-Value Iteration* algorithm (see Equation 18-3).

Equation 18-3. Q-Value Iteration algorithm

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')\right] \quad \text{for all } (s'a)$$

First example

Cart Pole experiment

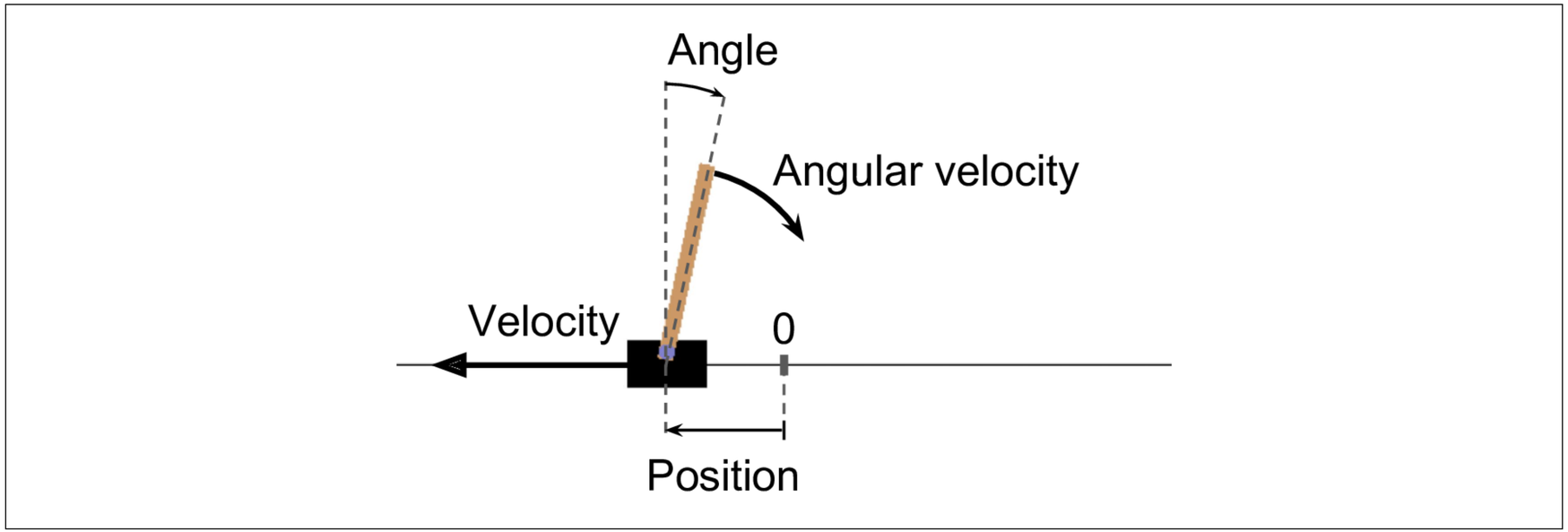


Figure 18-4. The CartPole environment

Second example

Frozen Lake

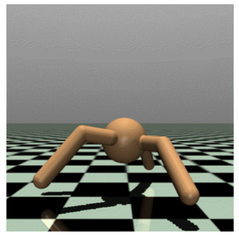
This environment is part of the Toy Text environments. Please read that page first for general information.

Action Space	Discrete(4)
Observation Space	Discrete(16)
Import	`gym.make("FrozenLake-v1")`

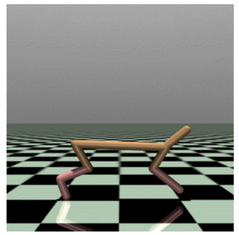
Frozen lake involves crossing a frozen lake from Start(S) to Goal(G) without falling into any Holes(H) by walking over the Frozen(F) lake. The agent may not always move in the intended direction due to the slippery nature of the frozen lake.

Third example

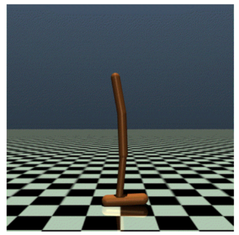
MuJoCo



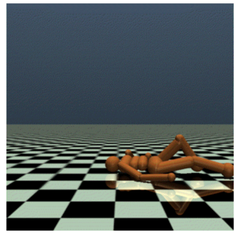
Ant



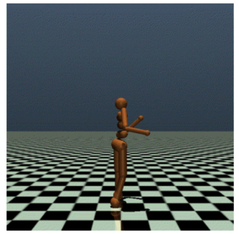
Half Cheetah



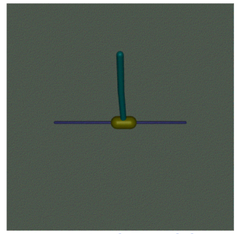
Hopper



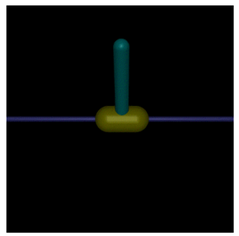
Humanoid Standup



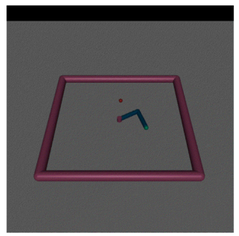
Humanoid



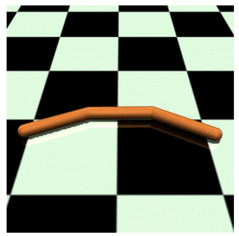
Inverted Double Pendulum



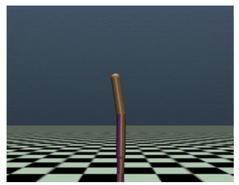
Inverted Pendulum



Reacher



Swimmer



Atari Games Performance

DQN Paper (Hasselt, Guez, Silver)