CS50's Introduction to Artificial Intelligence with Python
CS50 的人工智能入门与 Python

OpenCourseWare 开放课程

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 4 第 4 讲

Machine Learning 机器学习

Machine learning provides a computer with data, rather than explicit instructions. Using these data, the computer learns to recognize patterns and becomes able to execute tasks on its own.

机器学习为计算机提供数据，而不是明确的指令。利用这些数据，计算机学会识别模式，并能够自行执行任务。

Supervised Learning 监督学习

Supervised learning is a task where a computer learns a function that maps inputs to outputs based on a dataset of input-output pairs.

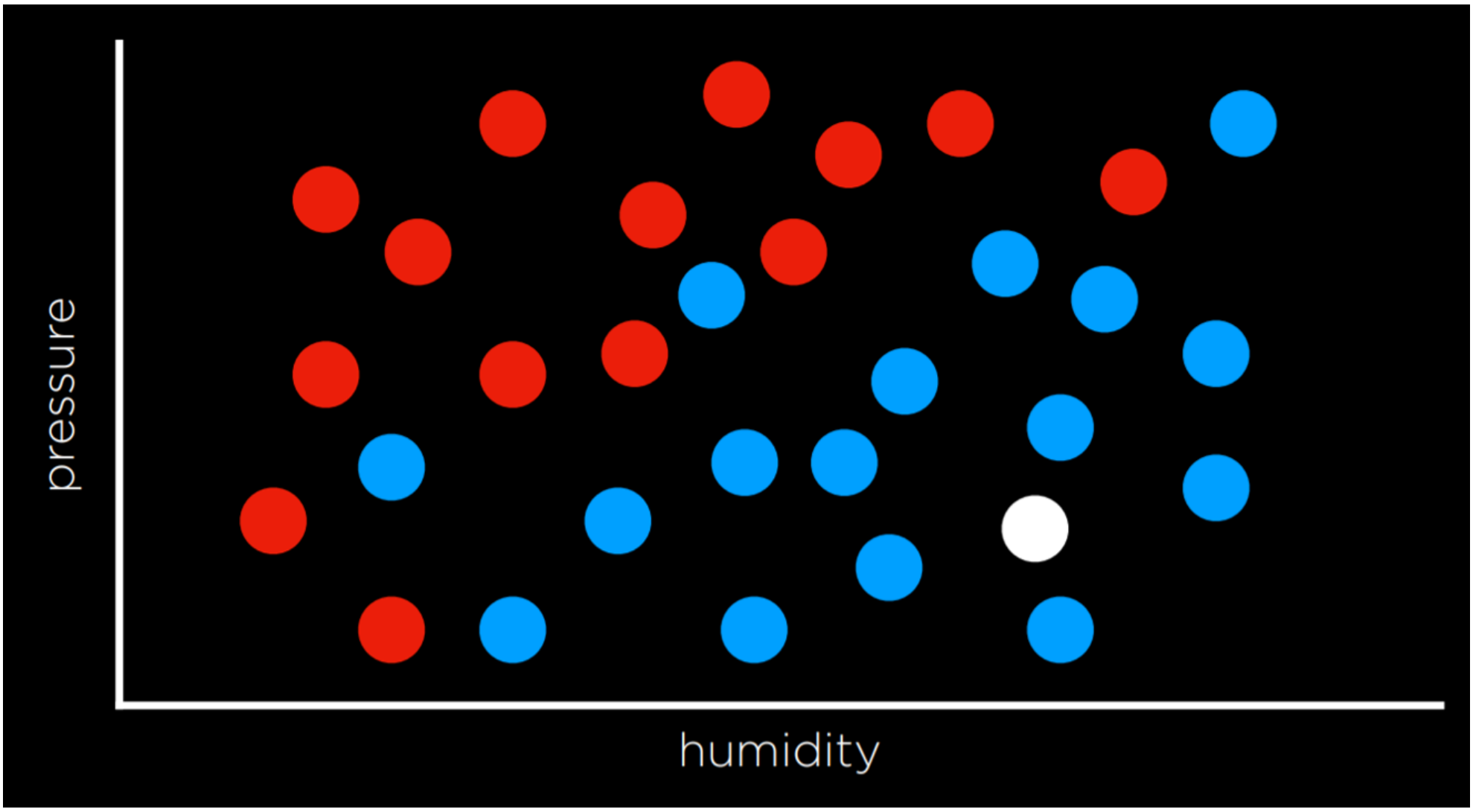
监督学习是一种任务，其中计算机根据输入-输出对的数据集学习将输入映射到输出的函数。

There are multiple tasks under supervised learning, and one of those is **Classification**. This is a task where the function maps an input to a discrete output. For example, given some information on humidity and air pressure for a particular day (input), the computer decides whether it will rain that day or not (output). The computer does this after training on a dataset with multiple days where humidity and air pressure are already mapped to whether it rained or not.

监督学习下的任务有多个，其中之一是分类。这是一个函数将输入映射到离散输出的任务。例如，给定某一天的湿度和气压信息（输入），计算机决定这一天是否会下雨（输出）。在训练集中有多个天数的数据，其中湿度和气压已经映射到是否下雨，计算机在训练后会进行这个决定。

This task can be formalized as follows. We observe nature, where a function *f(humidity, pressure)* maps the input to a discrete value, either Rain or No Rain. This function is hidden from us, and it is probably affected by many other variables that we don't have access to. Our goal is to create function *h(humidity, pressure)* that can approximate the behavior of function *f*. Such a task can be visualized by plotting days on the dimensions of humidity and rain (the input), coloring each data point in blue if it rained that day and in red if it didn't rain that day (the output). The white data point has only the input, and the computer needs to figure out the output.

此任务可以如下形式化。我们观察自然，其中函数 f(humidity, pressure) 将输入映射到离散值，要么是雨，要么是无雨。该函数对我们隐藏，它可能受到我们无法访问的许多其他变量的影响。我们的目标是创建函数 h(humidity, pressure)，它可以近似函数 f 的行为。这样的任务可以通过在湿度和降雨的维度上绘制天数，并将降雨天的数据点着色为蓝色，不降雨天的数据点着色为红色来可视化。只有输入的数据点是白色的，计算机需要找出输出。

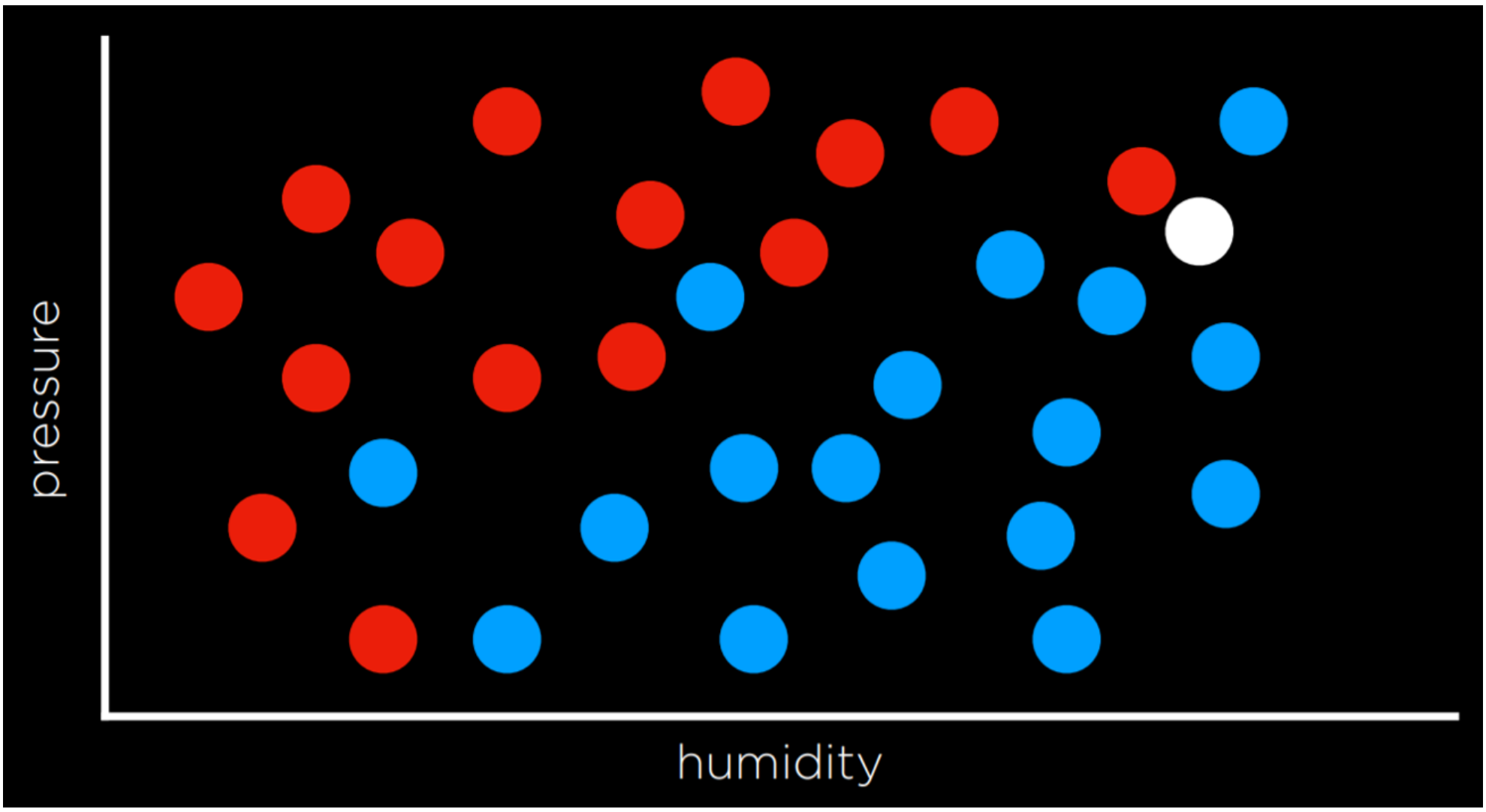


Nearest-Neighbor Classification

最近邻分类

One way of solving a task like the one described above is by assigning the variable in question the value of the closest observation. So, for example, the white dot on the graph above would be colored blue, because the nearest observed dot is blue as well. This might work well some times, but consider the graph below.

解决上述类型任务的一种方法是，将所关注的变量赋值为最近观测值的值。例如，上述图表中的白色点会被着色为蓝色，因为最近观测到的点也是蓝色的。这种方法在某些情况下可能效果很好，但请考虑下面的图表。



Following the same strategy, the white dot should be colored red, because the nearest observation to it is red as well. However, looking at the bigger picture, it looks like most of the other observations around it are blue, which might give us the intuition that blue is a better prediction

in this case, even though the closest observation is red.

遵循相同的策略，白点应该被涂成红色，因为离它最近的观察结果也是红色的。然而，从大局来看，似乎它周围大部分其他观察结果都是蓝色的，这可能让我们产生直觉，认为在这种情况下蓝色可能是更好的预测，尽管离它最近的观察结果是红色。

One way to get around the limitations of nearest-neighbor classification is by using **k-nearest-neighbors classification**, where the dot is colored based on the most frequent color of the *k* nearest neighbors. It is up to the programmer to decide what *k* is. Using a 3-nearest neighbors classification, for example, the white dot above will be colored blue, which intuitively seems like a better decision.

绕过最近邻分类限制的一种方法是使用 k-最近邻分类，在这种分类中，点的颜色根据 k 个最近邻中最常见的颜色决定。k 的值由程序员决定。例如，使用 3-最近邻分类，上图中的白色点将被着色为蓝色，这直观上似乎是更好的决策。

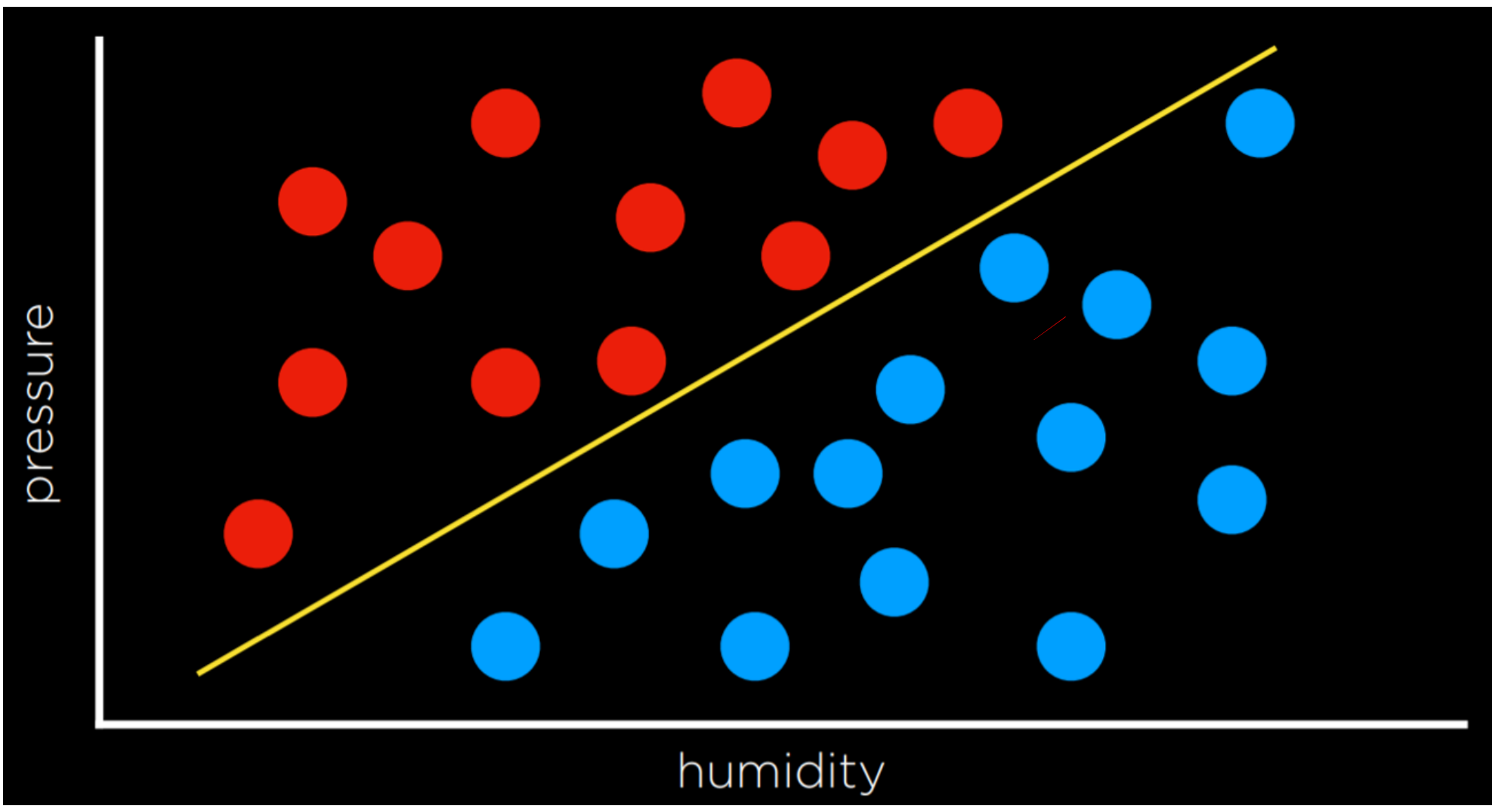
A drawback of the k-nearest-neighbors classification is that, using a naive approach, the algorithm will have to measure the distance of every single point to the point in question, which is computationally expensive. This can be sped up by using data structures that enable finding neighbors more quickly or by pruning irrelevant observations.

K-最近邻分类的一个缺点是，使用朴素的方法，算法将不得不测量每个点与问题点之间的距离，这在计算上是昂贵的。通过使用允许更快找到邻居的数据结构或修剪无关观察，可以加快速度。

Perceptron Learning 感知器学习

Another way of going about a classification problem, as opposed to the nearest-neighbor strategy, is looking at the data as a whole and trying to create a decision boundary. In two-dimensional data, we can draw a line between the two types of observations. Every additional data point will be classified based on the side of the line on which it is plotted.

处理分类问题的另一种方法，与最近邻策略不同，是将数据作为一个整体考虑，并尝试创建一个决策边界。在二维数据中，我们可以画一条线来区分两种类型的观察。每个额外的数据点将根据其在直线上的位置进行分类。



The drawback to this approach is that data are messy, and it is rare that one can draw a line and neatly divide the classes into two observations without any mistakes. Often, we will compromise, drawing a boundary that separates the observations correctly more often than not, but still occasionally misclassifies them.

这种方法的缺点是数据杂乱，很少有人能够画一条线并整洁地将类别分为两组观察而没有任何错误。通常，我们会妥协，画出一个边界，这个边界在大多数情况下能正确地区分观察结果，但仍然偶尔会错误分类。

In this case, the input of

在这种情况下，输入为

- x_1 = Humidity x_1 = 湿度
- x_2 = Pressure x_2 = 压力

will be given to a hypothesis function $h(x_1, x_2)$, which will output its prediction of whether it is going to rain that day or not. It will do so by checking on which side of the decision boundary the observation falls. Formally, the function will weight each of the inputs with an addition of a constant, ending in a linear equation of the following form:

将被提供给假设函数 \(h(x_1, x_2) \)，该函数将输出其对当天是否会下雨的预测。它将通过检查观察值位于决策边界哪一侧来这样做。正式地，该函数将每个输入项加权，再加上一个常数，最终形成以下形式的线性方程：

- Rain $w_0 + w_1x_1 + w_2x_2 \geqslant 0$

  降雨量 $w_0 + w_1x_1 + w_2x_2 \geqslant 0$

- No Rain otherwise 无雨否则

Often, the output variable will be coded as 1 and 0, where if the equation yields more than 0, the output is 1 (Rain), and 0 otherwise (No Rain).

通常，输出变量会被编码为 1 和 0，其中如果方程式的结果大于 0，则输出为 1（降雨），否则为 0（无降雨）。

The weights and values are represented by vectors, which are sequences of numbers (which can be stored in lists or tuples in Python). We produce a Weight Vector **w**: (w_0, w_1, w_2), and getting to the best weight vector is the goal of the machine learning algorithm. We also produce an Input Vector **x**: $(1, x_1, x_2)$.

权重和值由向量表示，向量是数字的序列（在 Python 中可以存储在列表或元组中）。我们生成一个权重向量 w：(w_0, w_1, w_2)，找到最佳权重向量是机器学习算法的目标。我们还生成一个输入向量 x：$(1, x_1, x_2)$。

We take the dot product of the two vectors. That is, we multiply each value in one vector by the corresponding value in the second vector, arriving at the expression above: $w_0 + w_1x_1 + w_2x_2$. The first value in the input vector is 1 because, when multiplied by the weight vector w_0, we want to keep it a constant.

我们将两个向量进行点积运算。也就是说，我们将一个向量中的每个值乘以第二个向量中对应的值，得到上面的表达式：$w_0 + w_1x_1 + w_2x_2$。输入向量中的第一个值为 1，因为当我们将其与权重向量 w_0相乘时，我们希望保持其为常数。

Thus, we can represent our hypothesis function the following way:

因此，我们可以将假设函数表示为以下方式：

$$h_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 \text{ if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

Since the goal of the algorithm is to find the best weight vector, when the algorithm encounters new data it updates the current weights. It does so using the **perceptron learning rule**:

由于算法的目标是找到最佳权重向量，因此当算法遇到新数据时，会更新当前的权重。它使用感知器学习规则进行这一操作：

$$w_i = w_i + \alpha(y - h_{\mathbf{w}}(\mathbf{x})) \times x_i$$

The important takeaway from this rule is that for each data point, we adjust the weights to make our function more accurate. The details, which are not as critical to our point, are that each weight is set to be equal to itself plus some value in parentheses. Here, y stands for the observed value while the hypothesis function stands for the estimate. If they are identical, this whole term is equal to zero, and thus the weight is not changed. If we underestimated (calling No Rain while Rain was observed), then the value in the parentheses will be 1 and the weight will increase by the value of x_i scaled by α the learning coefficient. If we overestimated (calling Rain while No Rain was observed), then the value in the parentheses will be -1 and the weight will decrease by the value of x scaled by α. The higher α, the stronger the influence each new event

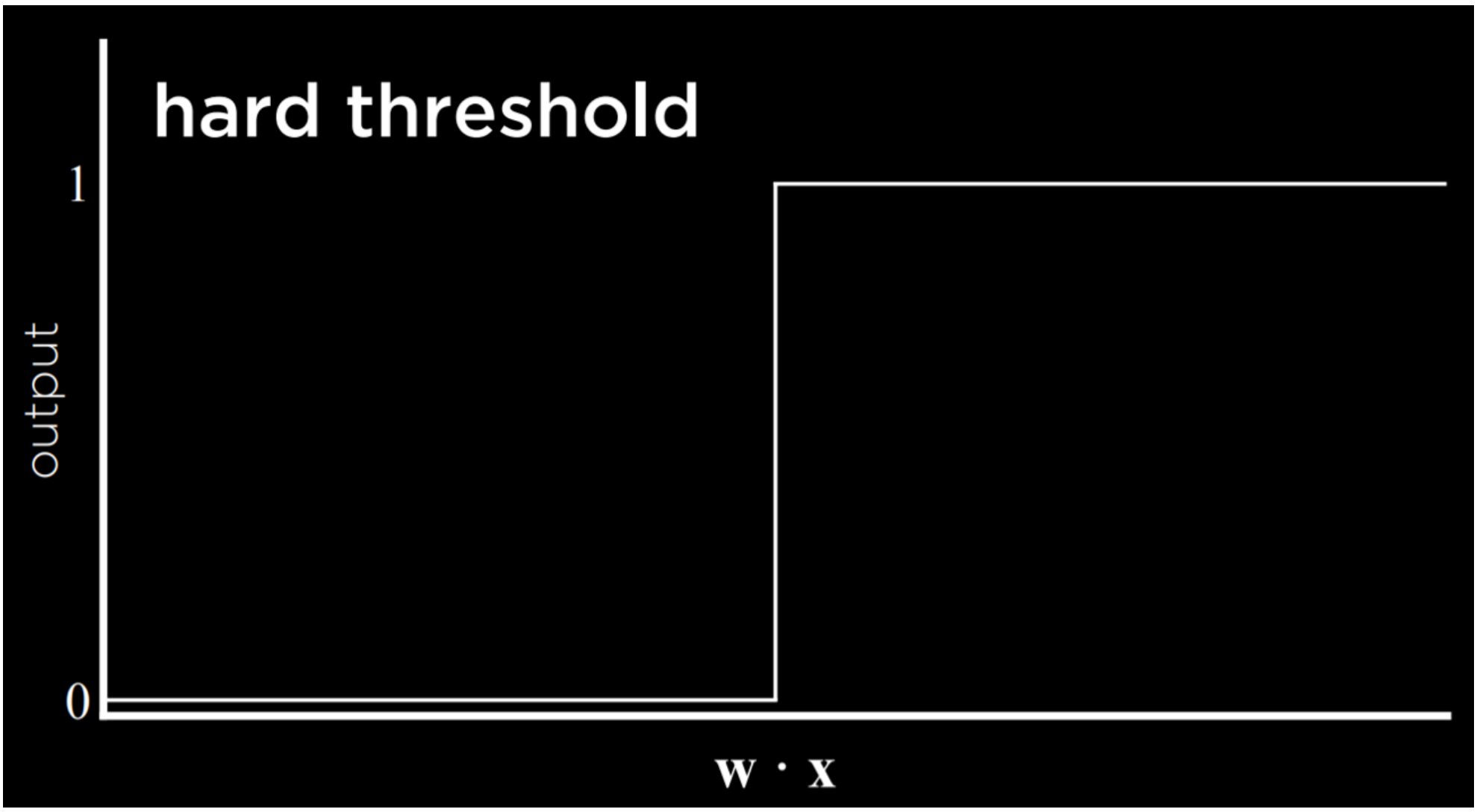
has on the weight.

即，有雨还是无雨

这条规则的关键点在于，对于每一点数据，我们调整权重以使我们的函数更准确。细节上，每一点权重被设置为它自身加上括号中的某个值。在这里，y 代表观察到的值，而假设函数代表估计。①如果它们相等，整个项等于零，因此权重不会改变。②如果我们的估计过低（在观察到雨时说无雨），则括号中的值为 1，权重将增加 x_i 的值乘以学习系数α。③如果我们的估计过高（在观察到无雨时说有雨），则括号中的值为-1，权重将减少 x 的值乘以α。α越高，每个新事件对权重的影响就越大。

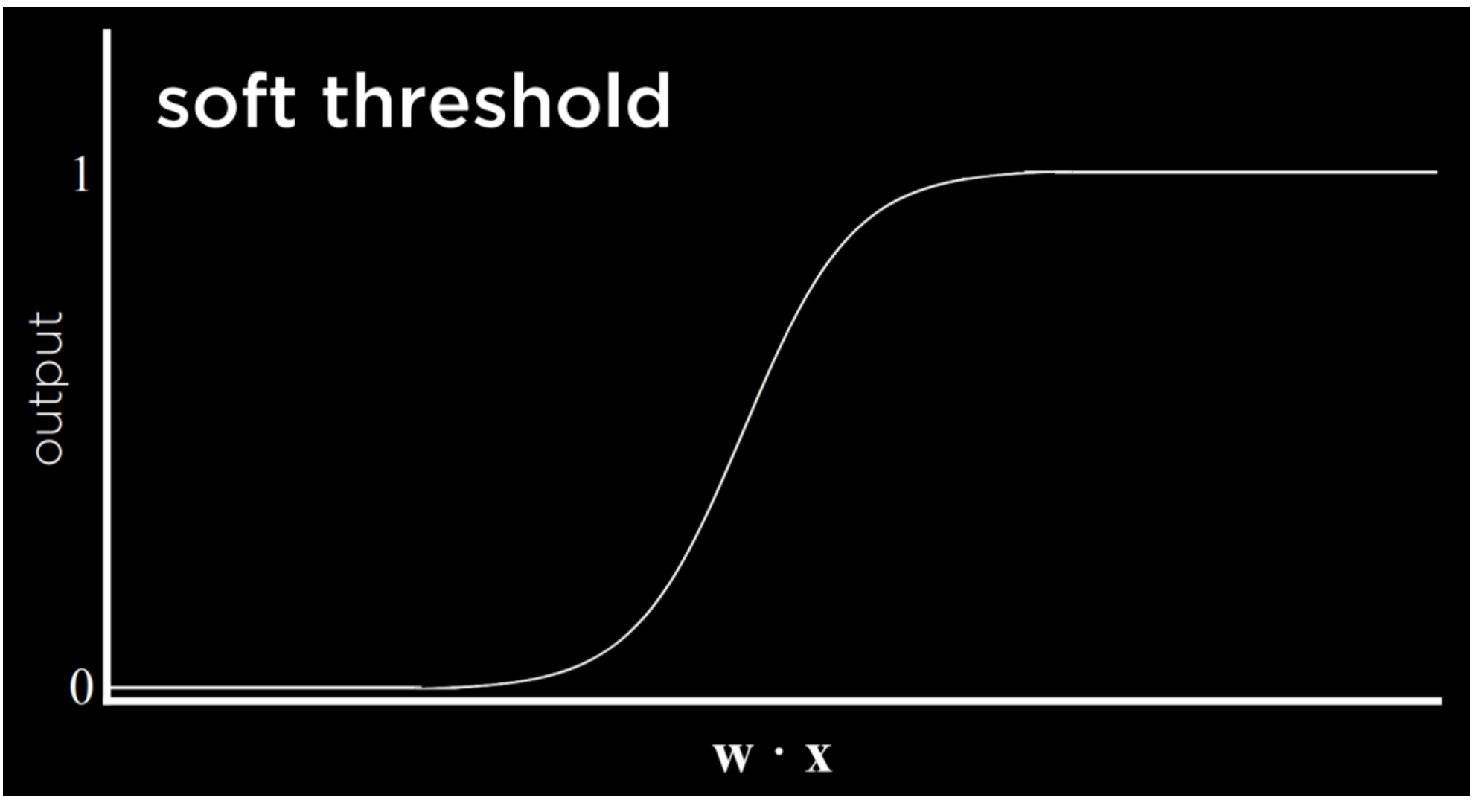
The result of this process is a threshold function that switches from 0 to 1 once the estimated value crosses some threshold.

这个过程的结果是一个阈值函数，当估计值越过某个阈值时，该函数从 0 切换到 1。



The problem with this type of function is that it is unable to express uncertainty, since it can only be equal to 0 or to 1. It employs a **hard threshold**. A way to go around this is by using a logistic function, which employs a **soft threshold**. A logistic function can yield a real number between 0 and 1, which will express confidence in the estimate. The closer the value to 1, the more likely it is to rain.

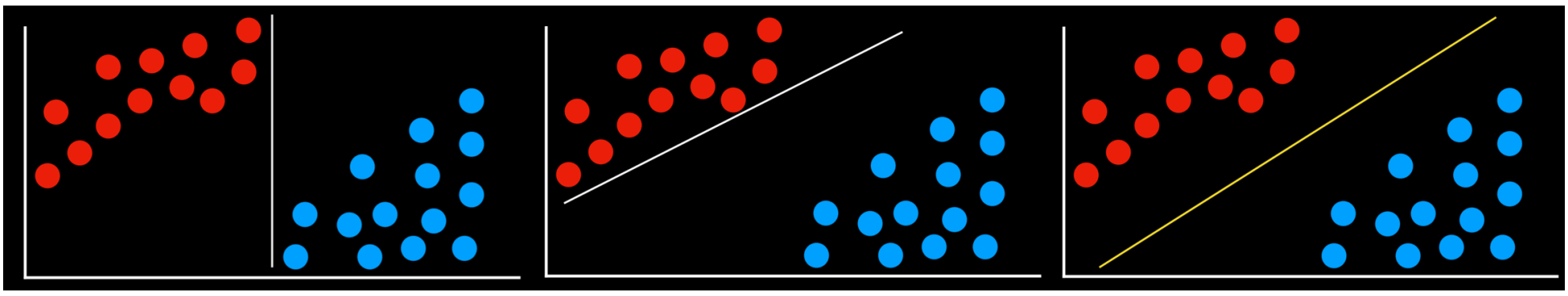
这种类型的函数的问题在于，它无法表达不确定性，因为它只能等于 0 或 1。它采用了一个硬阈值。绕过这个问题的一种方法是使用逻辑函数，它采用了一个软阈值。逻辑函数可以提供一个在 0 和 1 之间的实际数字，以表达对估计的信心。值越接近 1，下雨的可能性就越大。



Support Vector Machines 支持向量机

In addition to nearest-neighbor and linear regression, another approach to classification is the Support Vector Machine. This approach uses an additional vector (support vector) near the decision boundary to make the best decision when separating the data. Consider the example below.

除了最近邻和线性回归之外，分类的另一种方法是支持向量机。这种方法在决策边界附近使用额外的向量（支持向量）来在分离数据时做出最佳决策。考虑以下示例。

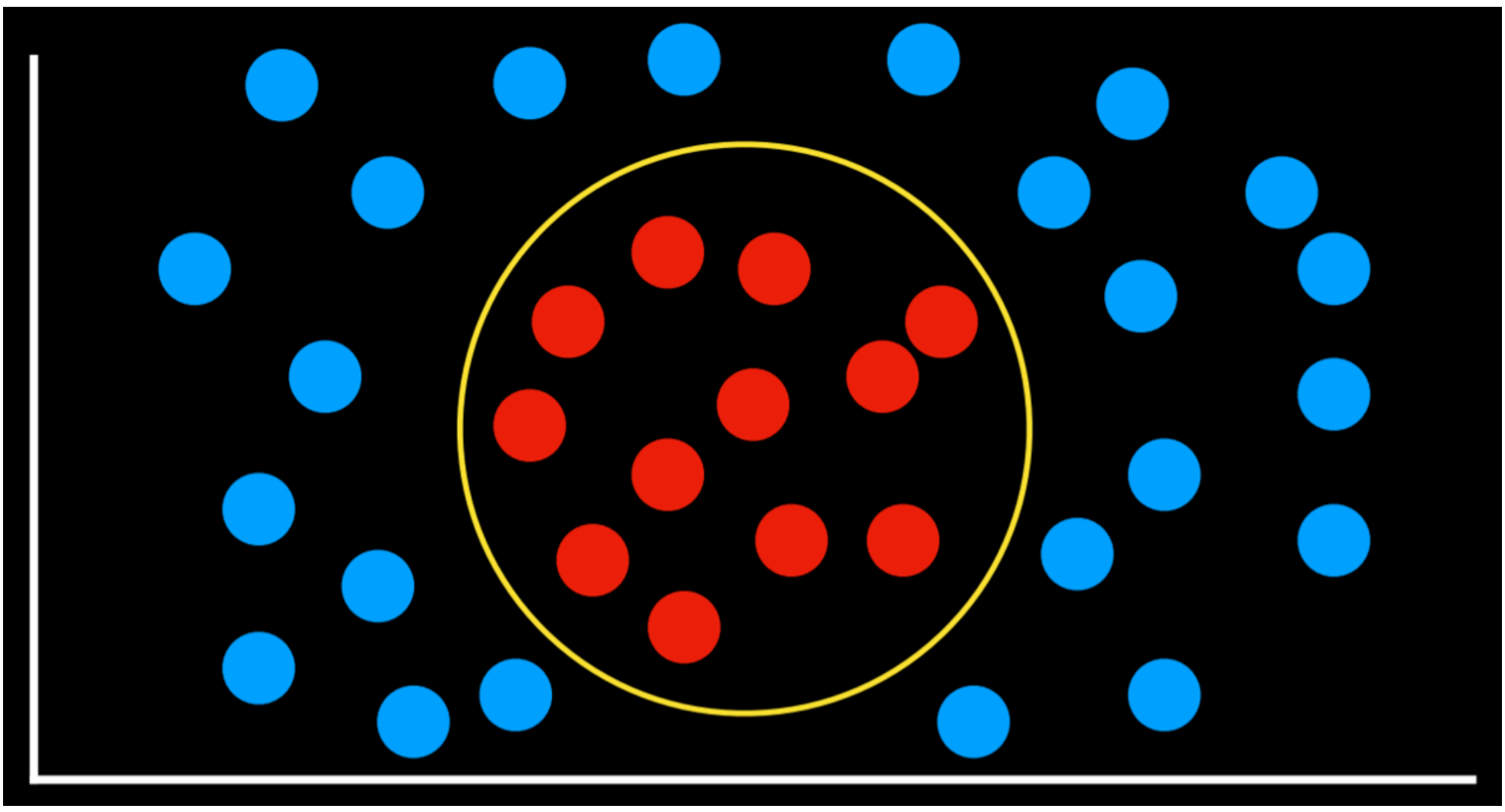


All the decision boundaries work in that they separate the data without any mistakes. However, are they equally as good? The two leftmost decision boundaries are very close to some of the observations. This means that a new data point that differs only slightly from one group can be wrongly classified as the other. As opposed to that, the rightmost decision boundary keeps the most distance from each of the groups, thus giving the most leeway for variation within it. This type of boundary, which is as far as possible from the two groups it separates, is called the **Maximum Margin Separator**.

所有决策边界的工作原理是它们能够准确地将数据分开。然而，它们是否同样优秀呢？最左边的两个决策边界非常接近一些观察结果。这意味着，与某个组仅略有不同的新数据点可能会被错误地分类为另一个组。相比之下，最右边的决策边界与每个组保持最远的距离，因此为内部变化提供了最大的灵活性。这种边界，它尽可能远离它所分离的两个组，被称为最大间隔分离器。

Another benefit of support vector machines is that they can represent decision boundaries with more than two dimensions, as well as non-linear decision boundaries, such as below.

支持向量机的另一个好处是它们可以表示超过两个维度的决策边界，以及非线性决策边界，如下所示。



To summarize, there are multiple ways to go about classification problems, with no one being always better than the other. Each has their drawbacks and might prove more useful than others in specific situations.

总之，处理分类问题的方法有很多种，并没有哪一种方法总是优于其他方法。每种方法都有其缺点，在特定情况下可能比其他方法更有用。

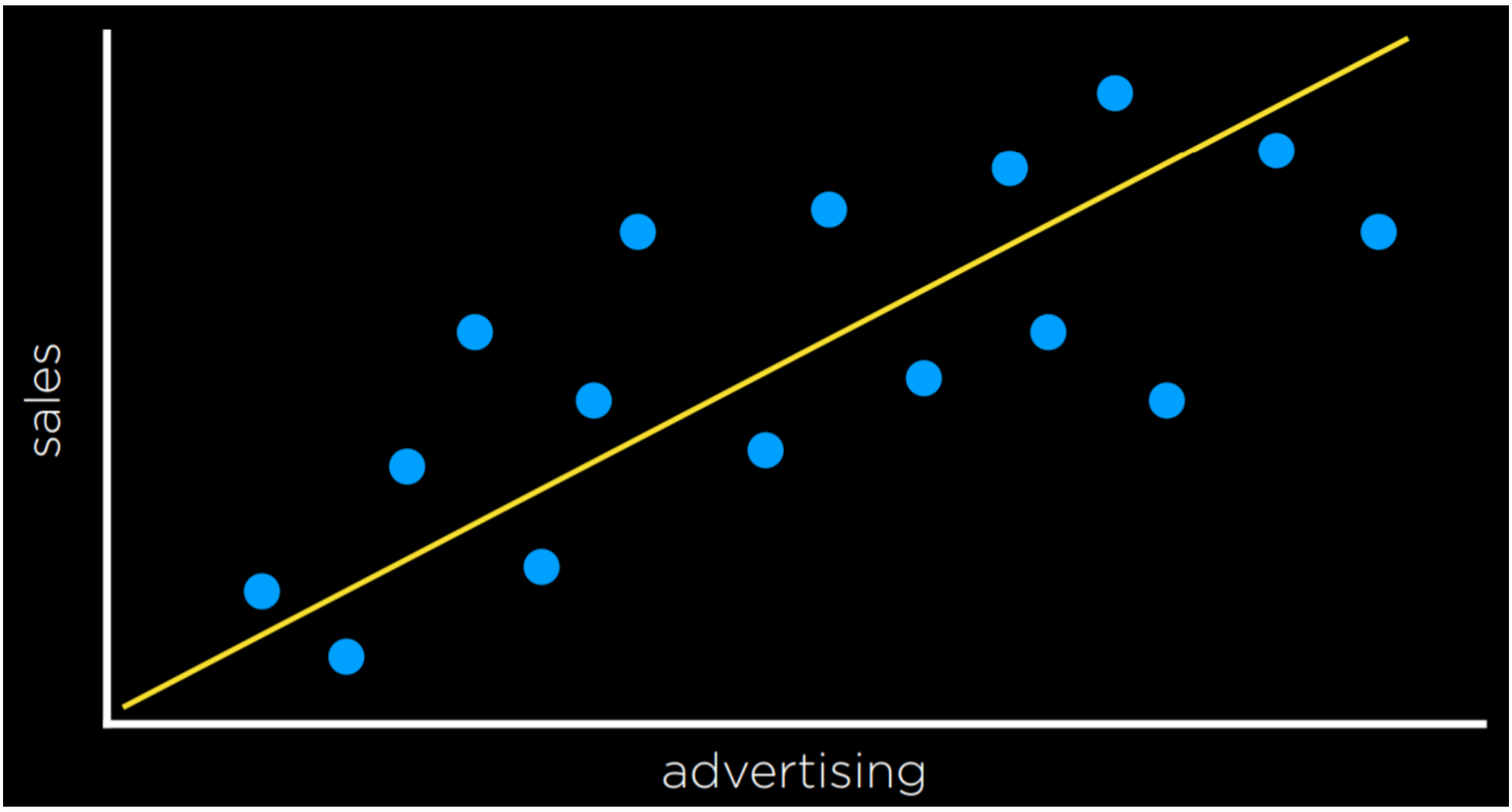
Regression 回归

Regression is a supervised learning task of a function that maps an input point to a continuous value, some real number. This differs from classification in that classification problems map an input to discrete values (Rain or No Rain).

回归是一种监督学习任务，其功能是映射输入点到连续值，一些实数。这与分类问题不同，分类问题将输入映射到离散值（如雨或无雨）。

For example, a company might use regression to answer the question of how money spent advertising predicts money earned in sales. In this case, an observed function *f(advertising)* represents the observed income following some money that was spent in advertising (note that the function can take more than one input variable). These are the data that we start with. With this data, we want to come up with a hypothesis function *h(advertising)* that will try to approximate the behavior of function *f*. *h* will generate a line whose goal is not to separate between types of observations, but to predict, based on the input, what will be the value of the output.

例如，一家公司可能会使用回归来回答这样一个问题：在广告上花费的钱如何预测销售收入。在这种情况下，观察到的函数 f(advertising)表示在进行了某些广告支出后观察到的收入（请注意，函数可以接受多个输入变量）。这些都是我们开始的数据。有了这些数据，我们想要提出一个假设函数 h(advertising)，试图近似函数 f 的行为。h 将生成一条线，其目标不是在不同类型的观察之间进行划分，而是根据输入预测输出的值。



Loss Functions 损失函数

Loss functions are a way to quantify the utility lost by any of the decision rules above. The less accurate the prediction, the larger the loss.

损失函数是一种量化上述任何决策规则所损失的效用的方法。预测越不准确，损失就越大。

For classification problems, we can use a **0-1 Loss Function**.

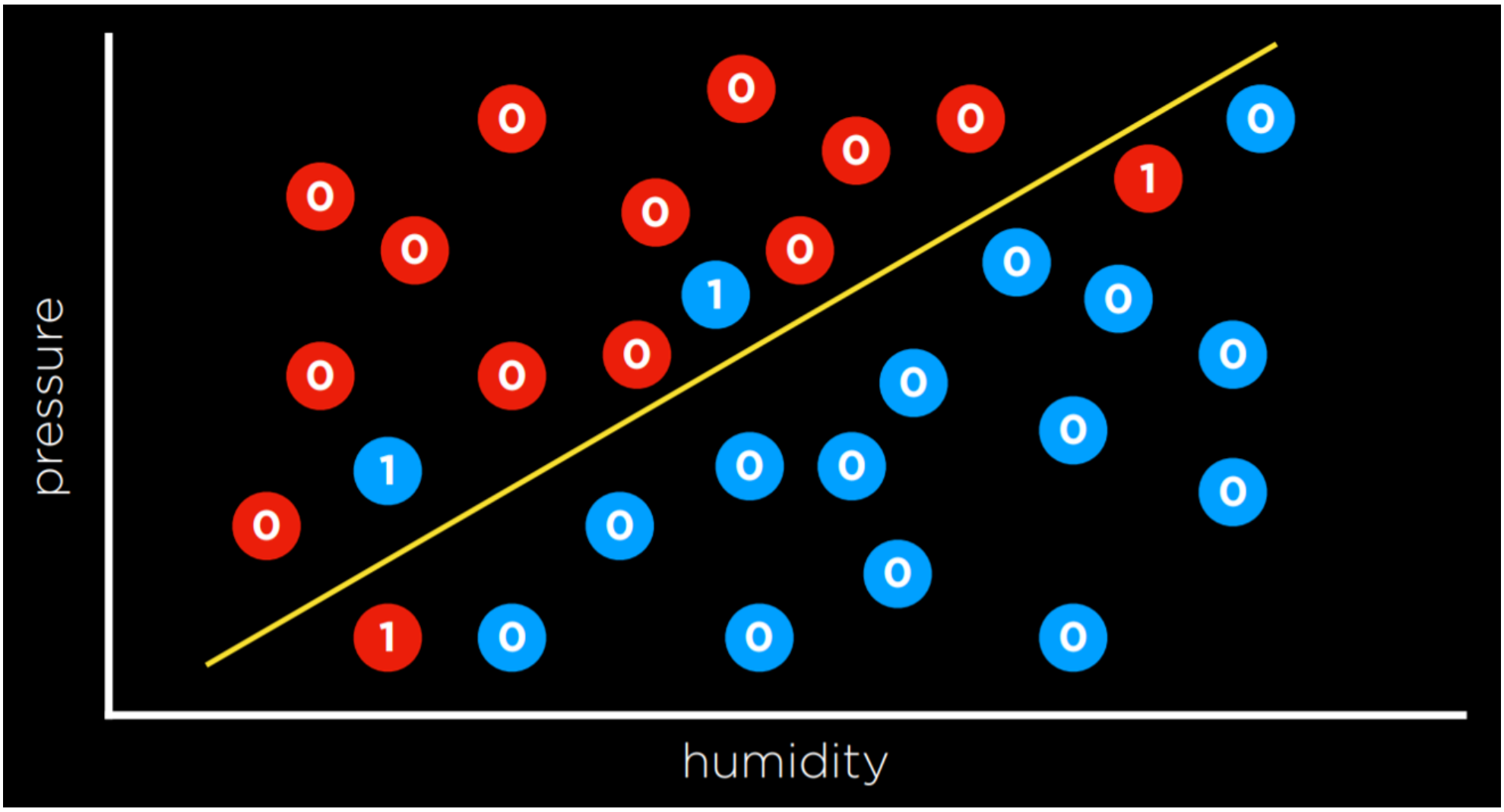
对于分类问题，我们可以使用 0-1 损失函数。

- *L*(actual, predicted)：实际值，预测值：L(actual, predicted)
  - 0 if actual = predicted

    如果实际值等于预测值，则输出 0
  - 1 otherwise 1 否则

In words, this function gains value when the prediction isn't correct and doesn't gain value when it is correct (i.e. when the observed and predicted values match).

在语言中，这个函数在预测不正确时获得价值，在正确时（即观察值和预测值匹配时）不获得价值。



In the example above, the days that are valued at 0 are the ones where we predicted the weather correctly (rainy days are below the line and not rainy days are above the line). However, days when it didn't rain below the line and days when it did rain above it are the ones that we failed to predict. We give each one the value of 1 and sum them up to get an empirical estimate of how lossy our decision boundary is.

在上述示例中，值为 0 的天数是我们正确预测了天气的天数（下雨的天数在下部，未下雨的天数在上部）。然而，下部未下雨和上部下雨的天数是我们未能预测的天数。我们将每个这样的天数赋值为 1，并将它们相加，以得到我们决策边界损失程度的实证估计。

L_1 and L_2 loss functions can be used when predicting a continuous value. In this case, we are interested in quantifying for each prediction *how much* it differed from the observed value. We do this by taking either the absolute value or the squared value of the observed value minus the predicted value (i.e. how far the prediction was from the observed value).

在预测连续值时，可以使用 L_1 和 L_2 损失函数。在这种情况下，我们感兴趣的是量化每个预测与观察值之间的差异程度。我们通过取观察值减去预测值的绝对值或平方值（即预测值与观察值之间的距离）来实现这一点。

- L_1: L(actual, predicted) = |actual - predicted|

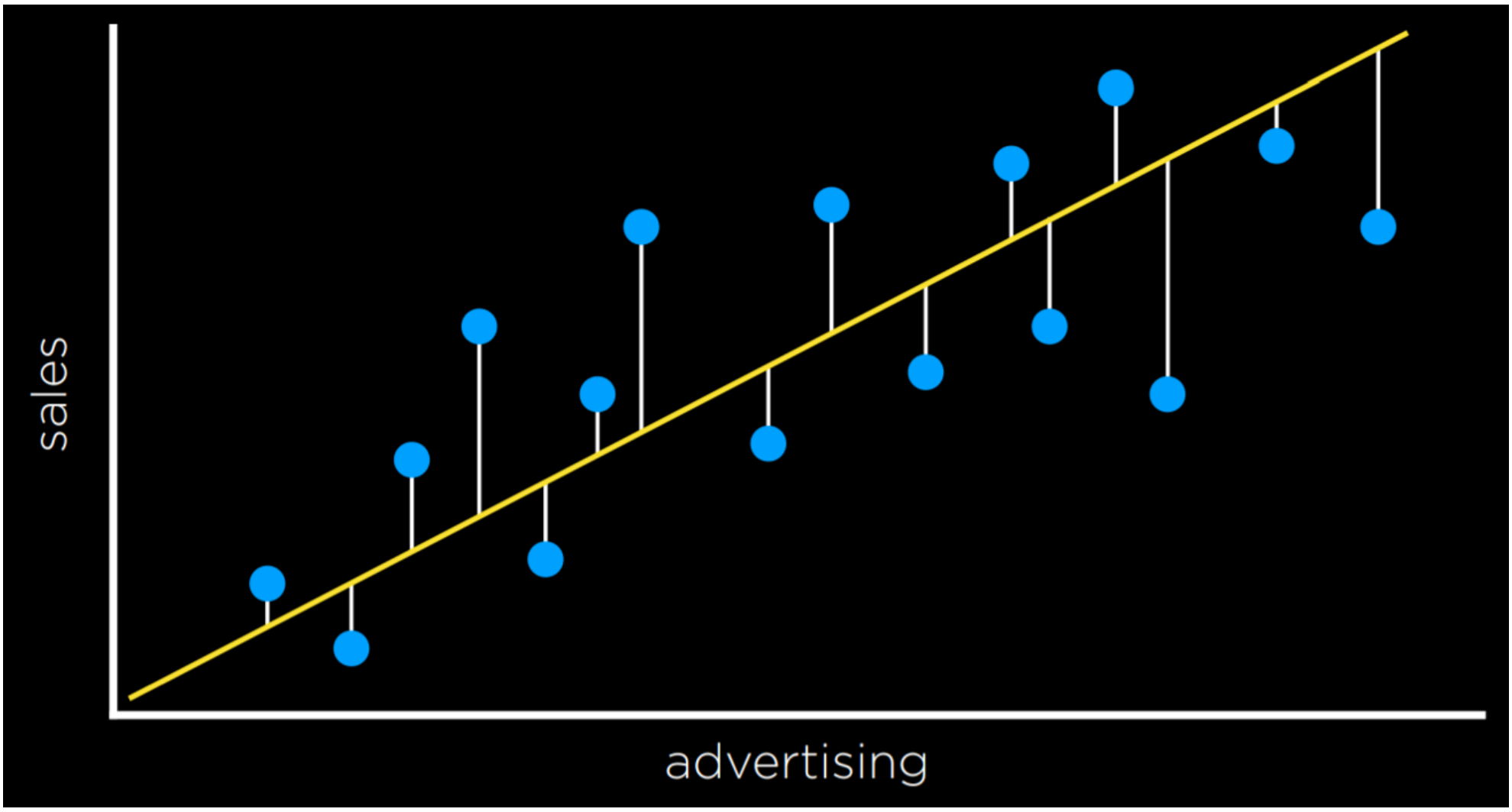
  L_1: L(实际, 预测) = |实际 - 预测|

- L_2: L(actual, predicted) = $(\text{actual} - \text{predicted})^2$

  L_2: L(实际, 预测) = $(\text{实际} - \text{预测})^2$

One can choose the loss function that serves their goals best. L_2 penalizes outliers more harshly than L_1 because it squares the the difference. L_1 can be visualized by summing the distances from each observed point to the predicted point on the regression line:

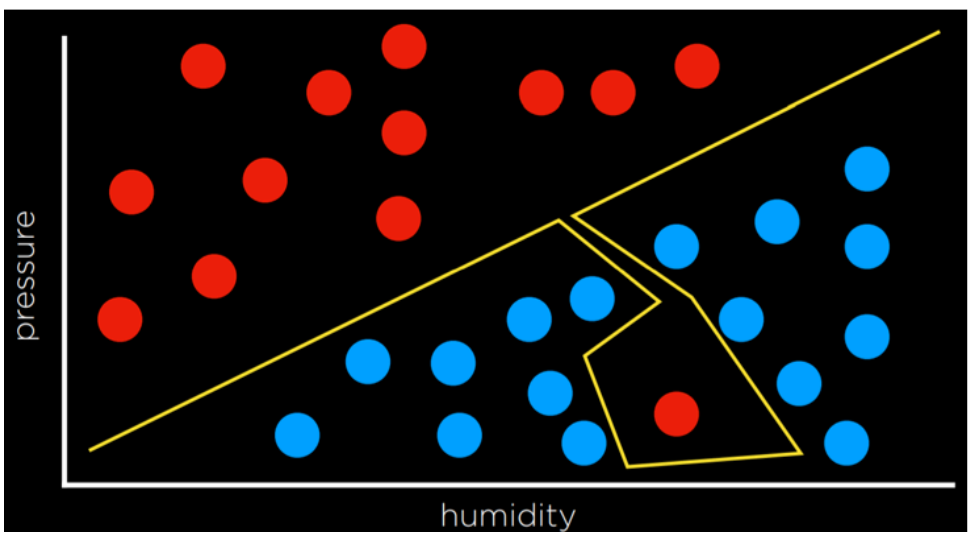
人们可以选择最符合其目标的损失函数。L_2 比 L_1 更严厉地惩罚异常值，因为它将差异平方。L_1 可以通过计算回归线上每个观测点到预测点的距离之和来可视化：

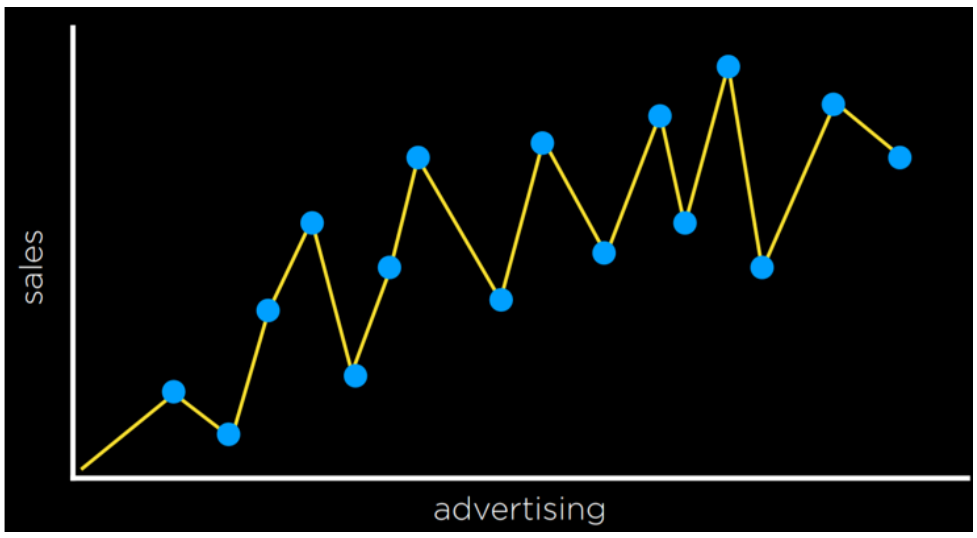


Overfitting 过拟合

Overfitting is when a model fits the training data so well that it fails to generalize to other data sets. In this sense, loss functions are a double edged sword. In the two examples below, the loss function is minimized such that the loss is equal to 0. However, it is unlikely that it will fit new data well.

过拟合是指模型对训练数据拟合得如此之好，以至于无法泛化到其他数据集。从这个意义上说，损失函数是一把双刃剑。在下面的两个例子中，损失函数被最小化到损失等于 0。然而，它可能无法很好地适应新数据。





For example, in the left graph, a dot next to the red one at the bottom of the screen is likely to be Rain (blue). However, with the overfitted model, it will be classified as No Rain (red).

例如，在左侧的图表中，屏幕底部红色点旁边的一个点很可能是雨（蓝色）。然而，使用过拟合的模型，它会被分类为无雨（红色）。

Regularization 正则化

Regularization is the process of penalizing hypotheses that are more complex to favor simpler, more general hypotheses. We use regularization to avoid overfitting.

正则化是惩罚更复杂的假设以偏好更简单、更普遍假设的过程。我们使用正则化来避免过拟合。

In regularization, we estimate the cost of the hypothesis function h by adding up its loss and a measure of its complexity.

在正则化中，我们通过将假设函数 h 的损失与其复杂性的度量相加，来估算其成本。

$cost(h) = loss(h) + \lambda complexity(h)$

成本(h) = 损失(h) + λ复杂性(h)

Lambda (λ) is a constant that we can use to modulate how strongly to penalize for complexity in our cost function. The higher λ is, the more costly complexity is.

拉姆达（λ）是一个常数，我们可以使用它来调整在成本函数中对复杂性的惩罚力度。λ的值越高，复杂性就越昂贵。

One way to test whether we overfitted the model is with **Holdout Cross Validation**. In this technique, we split all the data in two: a **training set** and a **test set**. We run the learning algorithm on the training set, and then see how well it predicts the data in the test set. The idea here is that by testing on data that were not used in training, we can a measure how well the learning generalizes.

测试我们是否过拟合模型的一种方法是使用留出交叉验证。在这个技术中，我们将所有数据分为两部分：训练集和测试集。我们在训练集上运行学习算法，然后查看它在测试集上的预测能力如何。这里的理念是，通过在未用于训练的数据上进行测试，我们可以衡量学习算法的泛化能力如何。

The downside of holdout cross validation is that we don't get to train the model on half the data, since it is used for evaluation purposes. A way to deal with this is using ***k*-Fold Cross-Validation**. In this process, we divide the data into k sets. We run the training k times, each time leaving out one dataset and using it as a test set. We end up with k different evaluations of our model, which we can average and get an estimate of how our model generalizes without losing any data.

保留交叉验证的缺点是我们不能在模型上训练一半的数据，因为它用于评估目的。处理这个问题的一种方法是使用 k-折交叉验证。在这个过程中，我们将数据分为 k 组。我们运行 k 次训练，每次排除一组数据并将其用作测试集。我们最终得到 k 个模型的不同评估，我们可以对这些评估求平均，得到模型泛化能力的估计，而不会丢失任何数据。

scikit-learn

As often is the case with Python, there are multiple libraries that allow us to conveniently use machine learning algorithms. One of such libraries is scikit-learn.

正如通常情况下，Python 中有多个库允许我们方便地使用机器学习算法。其中一个这样的库是 scikit-learn。

As an example, we are going to use a CSV (https://en.wikipedia.org/wiki/Comma-separated_values) dataset of counterfeit banknotes.

例如，我们将使用一个假冒钞票的 CSV 数据集作为示例。

```
variance,skewness,curtosis,entropy,class
-0.89569,3.0025,-3.6067,-3.4457,1
3.4769,-0.15314,2.53,2.4495,0
3.9102,6.065,-2.4534,-0.68234,0
0.60731,3.9544,-4.772,-4.4853,1
2.3718,7.4908,0.015989,-1.7414,0
-2.2153,11.9625,0.078538,-7.7853,0
3.9433,2.5017,1.5215,0.903,0
3.931,1.8541,-0.023425,1.2314,0
3.9719,1.0367,0.75973,1.0013,0
0.55298,-3.4619,1.7048,1.1008,1
0.26877,4.987,-5.1508,-6.3913,1
```

The four left columns are data that we can use to predict whether a note is genuine or counterfeit, which is external data provided by a human, coded as 0 and 1. Now we can train our model on this data set and see if we can predict whether new banknotes are genuine or not.

四列左侧的数据是我们可以用来预测笔记是真品还是伪造品的资料，这是人类提供的外部数据，编码为 0 和 1。现在我们可以使用这些数据集训练我们的模型，并看看是否可以预测新的钞票是真品还是假的。

```
import csv
import random

from sklearn import svm
from sklearn.linear_model import Perceptron
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier

# model = KNeighborsClassifier(n_neighbors=1)
# model = svm.SVC()
model = Perceptron()
```

Note that after importing the libraries, we can choose which model to use. The rest of the code will stay the same. SVC stands for Support Vector Classifier (which we know as support vector machine). The KNeighborsClassifier uses the k-neighbors strategy, and requires as input the number

of neighbors it should consider.

注意，在导入库后，我们可以选择要使用的模型。代码的其余部分将保持不变。SVC 代表支持向量分类器（我们称之为支持向量机）。KNeighborsClassifier 使用 k 邻近策略，并需要输入它应该考虑的邻居数量。

```
# Read data in from file
with open("banknotes.csv") as f:
    reader = csv.reader(f)
    next(reader)

    data = []
    for row in reader:
        data.append({
            "evidence": [float(cell) for cell in row[:4]],
            "label": "Authentic" if row[4] == "0" else "Counterfeit"
        })

# Separate data into training and testing groups
holdout = int(0.40 * len(data))
random.shuffle(data)
testing = data[:holdout]
training = data[holdout:]

# Train model on training set
X_training = [row["evidence"] for row in training]
y_training = [row["label"] for row in training]
model.fit(X_training, y_training)

# Make predictions on the testing set
X_testing = [row["evidence"] for row in testing]
y_testing = [row["label"] for row in testing]
predictions = model.predict(X_testing)

# Compute how well we performed
correct = 0
incorrect = 0
total = 0
for actual, predicted in zip(y_testing, predictions):
    total += 1
    if actual == predicted:
        correct += 1
    else:
        incorrect += 1

# Print results
print(f"Results for model {type(model).__name__}")
print(f"Correct: {correct}")
print(f"Incorrect: {incorrect}")
print(f"Accuracy: {100 * correct / total:.2f}%")
```

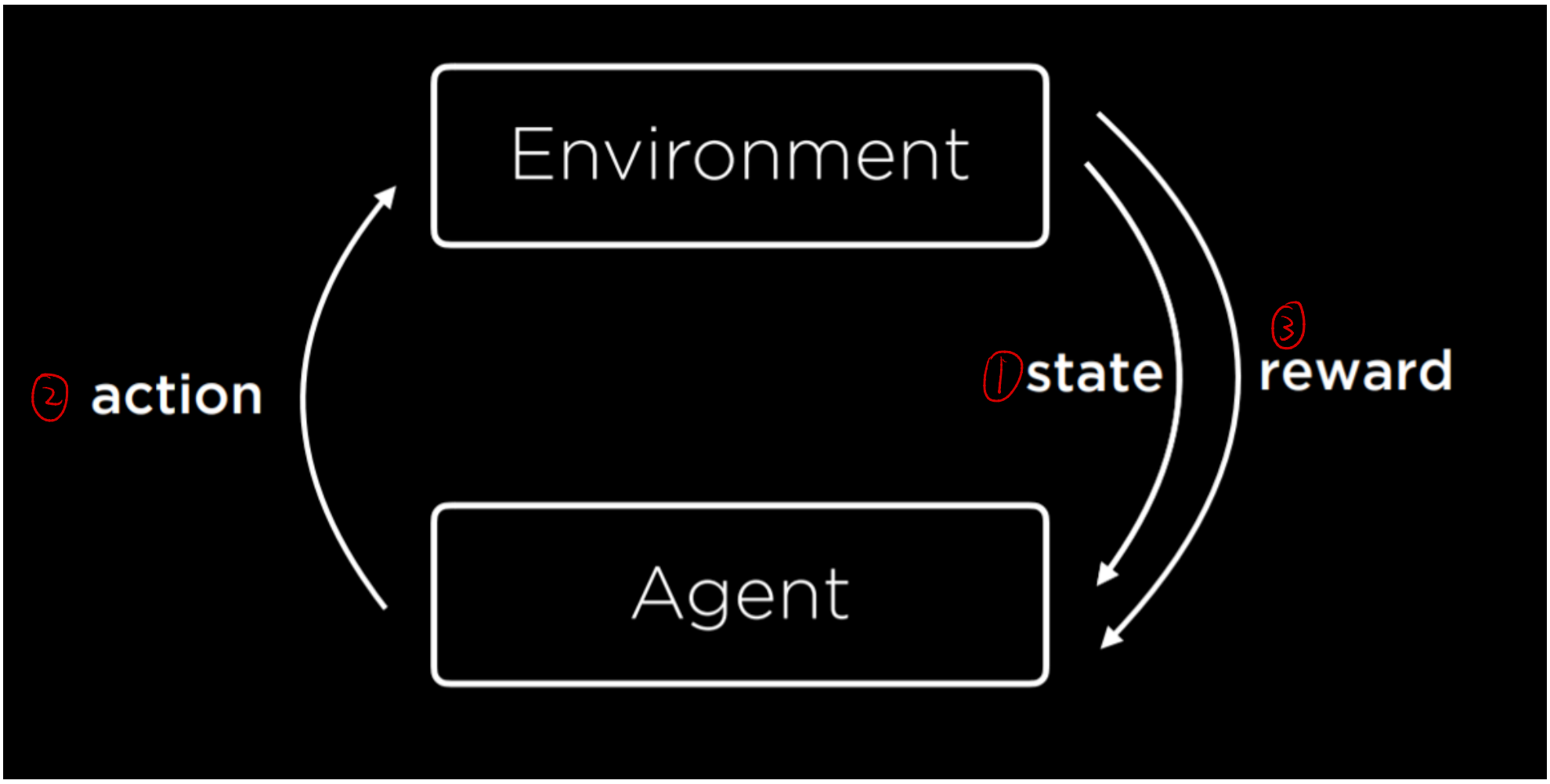
This manual version of running the algorithm can be found in the source code for this lecture under banknotes0.py. Since the algorithm is used often in a similar way, scikit-learn contains additional functions that make the code even more succinct and easy to use, and this version can be found under banknotes1.py.

此讲座的源代码中，banknotes0.py 文件中可以找到此讲义的手动算法运行版本。由于该算法经常以类似的方式使用，scikit-learn 包含了一些额外的函数，使代码更加简洁且易于使用，此版本可以在 banknotes1.py 文件中找到。

Reinforcement Learning 强化学习

Reinforcement learning is another approach to machine learning, where after each action, the agent gets feedback in the form of reward or punishment (a positive or a negative numerical value).

增强学习是机器学习的另一种方法，其中在每个动作之后，代理会收到奖励或惩罚的反馈（正数或负数的数值）。



The learning process starts by the environment providing a state to the agent. Then, the agent performs an action on the state. Based on this action, the environment will return a state and a reward to the agent, where the reward can be positive, making the behavior more likely in the future, or negative (i.e. punishment), making the behavior less likely in the future.

学习过程从环境向代理提供状态开始。然后，代理对状态执行操作。根据此操作，环境将向代理返回一个状态和奖励，其中奖励可以是正数，使未来的行为更有可能，或者负数（即惩罚），使未来的行为不太可能。

This type of algorithm can be used to train walking robots, for example, where each step returns a positive number (reward) and each fall a negative number (punishment).

这种算法可以用于训练行走机器人，例如，每一步返回正数（奖励），每次摔倒返回负数（惩罚）。

Markov Decision Processes

马尔可夫决策过程

Reinforcement learning can be viewed as a Markov decision process, having the following properties:

增强学习可以被视为马尔可夫决策过程，具有以下特性： - 它基于概率，每个状态都有可能转换到其他状态。 - 学习者通过观察当前状态和采取行动来获得奖励或惩罚。 - 目标是通过选择最优行动来最大化累积奖励。 - 状态转移概率是确定的，依赖于当前状态和采取的行动。 - 马尔可夫性意味着未来状态只依赖于当前状态，而不依赖于历史状态序列

- Set of states ***S*** 状态集 S
- Set of actions ***Actions(S)***

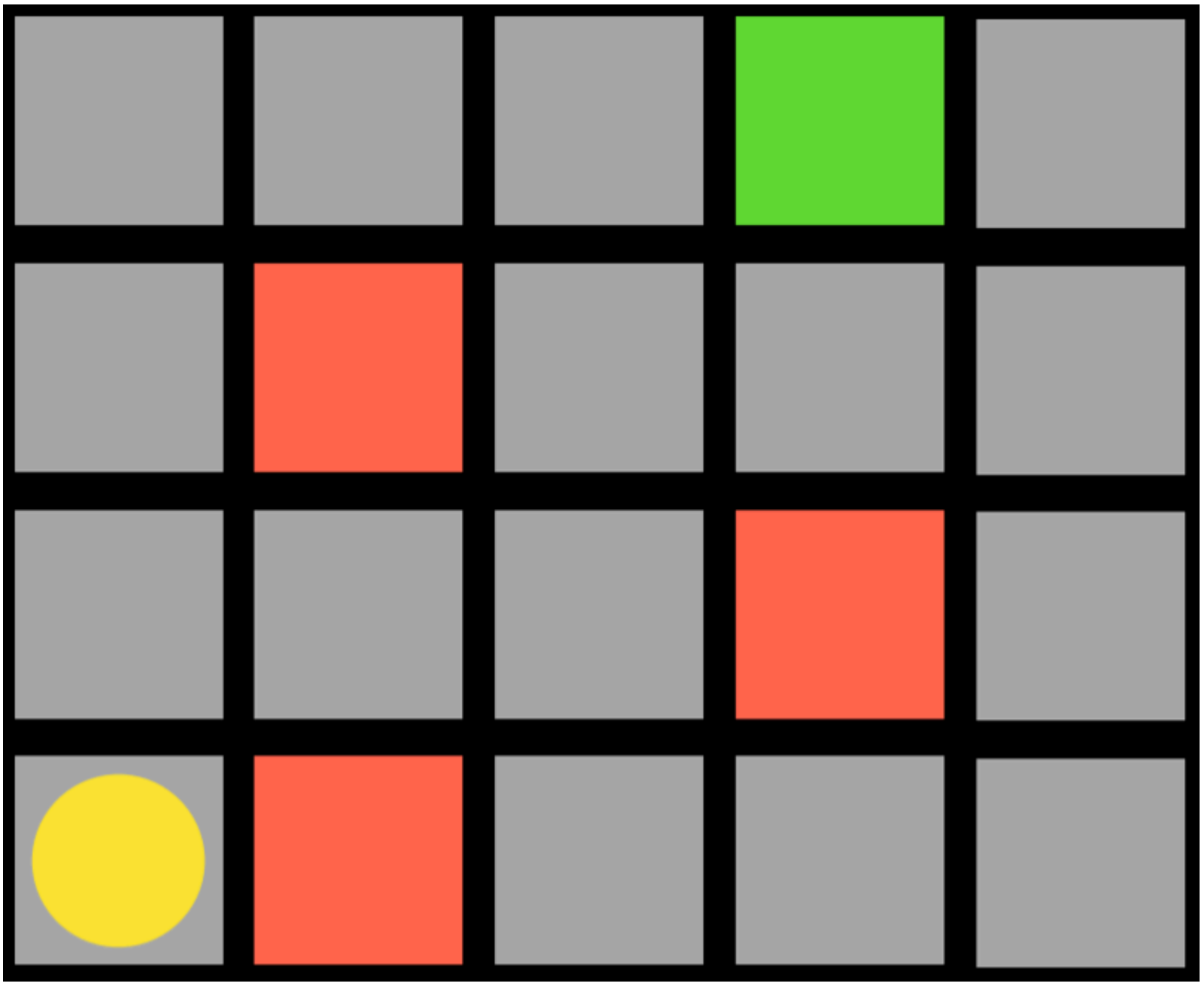
  动作集 Actions(S)
- Transition model ***P(s' | s, a)***

  转换模型 P(s' | s, a)
- Reward function ***R(s, a, s')***

  奖励函数 R(s, a, s')

For example, consider the following task:

例如，考虑以下任务：



The agent is the yellow circle, and it needs to get to the green square while avoiding the red squares. Every single square in the task is a state. Moving up, down, or to the sides is an action. The transition model gives us the new state after performing an action, and the reward function is what kind of feedback the agent gets. For example, if the agent chooses to go right, it will step on a red square and get negative feedback. This means that the agent will learn that, when in the state of being in the bottom-left square, it should avoid going right. This way, the agent will start exploring the space, learning which state-action pairs it should avoid. The algorithm can be probabilistic, choosing to take different actions in different states based on some probability that's being increased or decreased based on reward. When the agent reaches the green square, it will get a positive reward, learning that it is favorable to take the action it took in the previous state.

代理是黄色的圆圈，它需要到达绿色的正方形，同时避开红色的正方形。任务中的每一个正方形都是一种状态。向上、向下或向侧面移动是一种行动。转换模型告诉我们执行行动后的新状态，奖励函数是代理得到的反馈类型。例如，如果代理选择向右移动，它会踩到一个红色的正方形并得到负面反馈。这意味着，当处于左下角的状态时，代理应该避免向右移动。这样，代理将开始探索空间，学习哪些状态-行动对应该避免。算法可以是概率性的，根据某些概率在不同状态下选择不同的行动，该概率根据奖励增加或减少。当代理到达绿色的正方形时，它会得到正向奖励，学习在前一状态采取的行动是有利的。

Q-Learning Q-学习

Q-Learning is one model of reinforcement learning, where a function ***Q(s, a)*** outputs an estimate of the value of taking action *a* in state *s*.

Q 学习是强化学习的一种模型，其中函数 Q(s, a)输出在状态 s 下采取行动 a 的价值的估计。

The model starts with all estimated values equal to 0 (***Q(s,a)* = 0** for all *s*, *a*). When an action is taken and a reward is received, the function does two things: 1) it estimates the value of ***Q(s, a)*** based on current reward and expected future rewards, and 2) updates ***Q(s, a)*** to take into account both the old estimate and the new estimate. This gives us an algorithm that is capable of improving upon its past knowledge without starting from scratch.

模型开始时，所有估计值都等于 0（Q(s,a) = 0 对所有 s, a）。当采取行动并收到奖励时，函数做两件事：1）根据当前奖励和预期的未来奖励来估计 Q(s, a)的价值，2）更新 Q(s, a)，考虑到旧估计和新估计。这为我们提供了一个能够不从头开始就改进其过去知识的算法。

$$Q(s, a) \leftarrow Q(s, a) + \alpha(\text{new value estimate} - Q(s, a))$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha(\text{新值估计} - Q(s, a))$$

The updated value of ***Q(s, a)*** is equal to the previous value of ***Q(s, a)*** in addition to some updating value. This value is determined as the difference between the new value and the old value, multiplied by α, a learning coefficient. When α = 1 the new estimate simply overwrites the old one. When α = 0, the estimated value is never updated. By raising and lowering α, we can determine how fast previous knowledge is being

updated by new estimates.

更新后的 Q(s, a)值等于 Q(s, a)的先前值加上一些更新值。这个值是新值与旧值之差乘以α（学习系数）确定的。当α=1 时，新估计简单地覆盖了旧估计。当α=0 时，估计值永远不会更新。通过调整α的大小，我们可以决定旧知识被新估计快速更新的程度。

The new value estimate can be expressed as a sum of the reward (r) and the future reward estimate. To get the future reward estimate, we consider the new state that we got after taking the last action, and add the estimate of the action in this new state that will bring to the highest reward. This way, we estimate the utility of making action *a* in state *s* not only by the reward it received, but also by the expected utility of the next step. The value of the future reward estimate can sometimes appear with a coefficient gamma that controls how much future rewards are valued. We end up with the following equation:

新价值估计可以表示为奖励（r）和未来奖励估计的总和。为了获得未来奖励估计，我们考虑在执行最后的动作后获得的新状态，并添加在这个新状态下将带来最高奖励的动作估计。这样，我们不仅通过它收到的奖励，而且通过下一步的预期效用来估计执行动作 a 在状态 s 中的效用。价值的未来奖励估计有时会与控制未来奖励价值的系数γ一起出现。我们最终得到以下方程：

$$Q(s, a) \leftarrow Q(s, a) + \alpha((r + \gamma \max_{a'} Q(s', a')) - Q(s, a))$$

A **Greedy Decision-Making** algorithm completely discounts the future estimated rewards, instead always choosing the action ***a*** in current state ***s*** that has the highest ***Q(s, a)***.

贪婪决策算法完全忽略了未来的估计奖励，总是选择当前状态下 s 具有最高 Q(s, a)的动作 a。

This brings us to discuss the **Explore vs. Exploit** tradeoff. A greedy algorithm always exploits, taking the actions that are already established to bring to good outcomes. However, it will always follow the same path to the solution, never finding a better path. Exploration, on the other hand, means that the algorithm may use a previously unexplored route on its way to the target, allowing it to discover more efficient solutions along the way. For example, if you listen to the same songs every single time, you know you will enjoy them, but you will never get to know new songs that you might like even more!

这使我们讨论探索与利用之间的权衡。贪婪算法总是利用，采取已知能带来良好结果的行动。然而，它总是遵循同一条路径解决问题，从不找到更好的路径。另一方面，探索意味着算法在前往目标的途中可能使用之前未探索的路径，允许它在途中发现更有效的解决方案。例如，如果你每次都听同样的歌曲，你知道你会喜欢它们，但你永远无法了解你可能甚至更喜欢的新歌曲！

To implement the concept of exploration and exploitation, we can use the **ε (epsilon) greedy** algorithm. In this type of algorithm, we set ε equal to how often we want to move randomly. With probability 1-ε, the algorithm chooses the best move (exploitation). With probability ε, the algorithm chooses a random move (exploration).

为了实现探索与开发的概念，我们可以使用ε（epsilon）贪婪算法。在这种类型的算法中，我们将ε设置为我们希望随机移动的频率。以概率 1-ε，算法选择最佳行动（开发）。以概率ε，算法选择随机行动（探索）。

Another way to train a reinforcement learning model is to give feedback not upon every move, but upon the end of the whole process. For example, consider a game of Nim. In this game, different numbers of objects are distributed between piles. Each player takes any number of objects from any one single pile, and the player who takes the last object looses. In such a game, an untrained AI will play randomly, and it will be easy to win against it. To train the AI, it will start from playing a game randomly, and in the end get a reward of 1 for winning and -1 for losing. When it is trained on 10,000 games, for example, it is already smart enough to be hard to win against it.

训练强化学习模型的另一种方法是在整个过程结束后给出反馈，而不是在每一步之后。例如，考虑一个“纳姆”游戏。在这个游戏中，不同数量的物体分布在多个堆中。每个玩家可以从任意一个单堆中取任意数量的物体，取到最后一个物体的玩家输掉。在这种游戏中，未经训练的 AI 会随机操作，很容易就能战胜它。为了训练 AI，它会从随机玩游戏开始，在赢的时候获得 1 的奖励，在输的时候获得-1 的奖励。例如，当它在 10,000 场比赛中被训练后，就已经聪明到很难战胜它了。

This approach becomes more computationally demanding when a game has multiple states and possible actions, such as chess. It is infeasible to generate an estimated value for every possible move in every possible state. In this case, we can use a **function approximation**, which allows us to approximate ***Q(s, a)*** using various other features, rather than storing one value for each state-action pair. Thus, the algorithm becomes able to recognize which moves are similar enough so that their estimated value should be similar as well, and use this heuristic in its decision making.

当游戏有多个状态和可能的动作，如国际象棋时，这种方法的计算需求会增加。不可能为每个可能的状态和每种可能的移动生成一个估计值。在这种情况下，我们可以使用函数近似，这允许我们使用其他各种特征来近似 Q(s, a)，而不是为每种状态-动作对存储一个值。因此，算法能够识别哪些移动足够相似，以至于它们的估计值也应该相似，并在决策中使用这个启发式方法。

Unsupervised Learning 无监督学习

In all the cases we saw before, as in supervised learning, we had data with labels that the algorithm could learn from. For example, when we trained an algorithm to recognize counterfeit notes, each banknote had four variables with different values (the input data) and whether it is

counterfeit or not (the label). In unsupervised learning, only the input data is present and the AI learns patterns in these data.

在我们之前看到的所有案例中，就像在监督学习中一样，我们有带标签的数据，算法可以从这些数据中学习。例如，当我们训练一个算法来识别假钞时，每张钞票有四个变量，每个变量都有不同的值（输入数据），以及它是否是假钞（标签）。在无监督学习中，只有输入数据存在，AI 从这些数据中学习模式。

Clustering 聚类

Clustering is an unsupervised learning task that takes the input data and organizes it into groups such that similar objects end up in the same group. This can be used, for example, in genetics research, when trying to find similar genes, or in image segmentation, when defining different parts of the image based on similarity between pixels.

聚类是一种无监督学习任务，它将输入数据组织成组，使得相似的对象最终位于同一组中。例如，在遗传学研究中，可以用来寻找相似的基因，或者在图像分割中，根据像素之间的相似性定义图像的不同部分。

k-means Clustering k 均值聚类

k-means Clustering is an algorithm to perform a clustering task. It maps all data points in a space, and then randomly places k cluster centers in the space (it is up to the programmer to decide how many; this is the starting state we see on the left). Each cluster center is simply a point in the space. Then, each cluster gets assigned all the points that are closest to its center than to any other center (this is the middle picture). Then, in an iterative process, the cluster center moves to the middle of all these points (the state on the right), and then points are reassigned again to the clusters whose centers are now closest to them. When, after repeating the process, each point remains in the same cluster it was before, we have reached an equilibrium and the algorithm is over, leaving us with points divided between clusters.

K 均值聚类是一种执行聚类任务的算法。它将空间中的所有数据点映射出来，并在空间中随机放置 k 个聚类中心（程序员可以决定数量；这是我们看到的左侧状态）。每个聚类中心仅仅是空间中的一个点。然后，每个聚类被分配所有距离其中心最近的点，而不是任何其他中心（这是中间图片）。然后，在一个迭代过程中，聚类中心移动到所有这些点的中心，然后将点重新分配给中心现在离它们最近的聚类。当在重复过程后，每个点仍然留在它之前所在的聚类中时，我们达到了平衡，算法结束，留下我们将点分为聚类。