Lecture Notes on Machine Learning

Rahul Shandilya

January 28, 2020

Chapter 1

Introduction to Machine Learning

Machine Learning

what is machine learning?

We define **Intelligence** as "the ability to apply knowledge in order to perform better in an environment." We define **artificial intelligence** as the study and construction of agent programs that perform well in a given environment, for a given agent architecture. An agent is **learning** if it improves its performance on future tasks after making observations about the world.

Machine learning, then, is about making computers modify or adapt their actions (whether these actions are making predictions, or controlling a robot) so that these actions get more accurate, where accuracy is measured by how well the chosen actions reflect the correct ones.

Today we have enteredin the era of big data where continuosly data is produced by millions of computers, smartphones, sensors etc. and are stored on huge data centers. This deluge of data calls for automated methods of data analysis, which is what machine learning provides. From this perspective, we define machine learning as a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty.

Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed.— Arthur L. Samuel, AI pioneer, 1959

Definition: A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks inT, as measured by P, improves with experience E.

Why we want machine to learn?

First, the designers cannot anticipate all possible situations that the agent might find itself in. For example, a robot designed to navigate mazes must learn the layout of each new maze it encounters. Second, the designers cannot anticipate all changes over time; a program designed to predict tomorrow's stock market prices must learn to adapt when conditions change from boom to bust. Third, sometimes human

programmers have no idea how to program a solution themselves. For example, most people are good at recognizing the faces of family members, but even the best programmers are unable to program a computer to accomplish that task, except by using learning algorithms

Examples of machine learning application?

A checkers learning problem:

- **Task T**: playing checkers
- **Performance measure P**: percent of games won against opponents
- **Training experience E**: playing practice games against itself

A handwriting recognition learning problem:

- **Task T**: recognizing and classifying handwritten words within images
- **Performance measure P**: percent of words correctly classified
- **Training experience E**: a database of handwritten words with given classifications

A robot driving learning problem:

- **Task T**: driving on public four-lane highways using vision sensors
- **Performance measure P**: average distance traveled before an error (as judgedby human overseer)
- **Training experience E**: a sequence of images and steering commandsrecorded while observing a human driver

Classifiication machine learning algorithm?

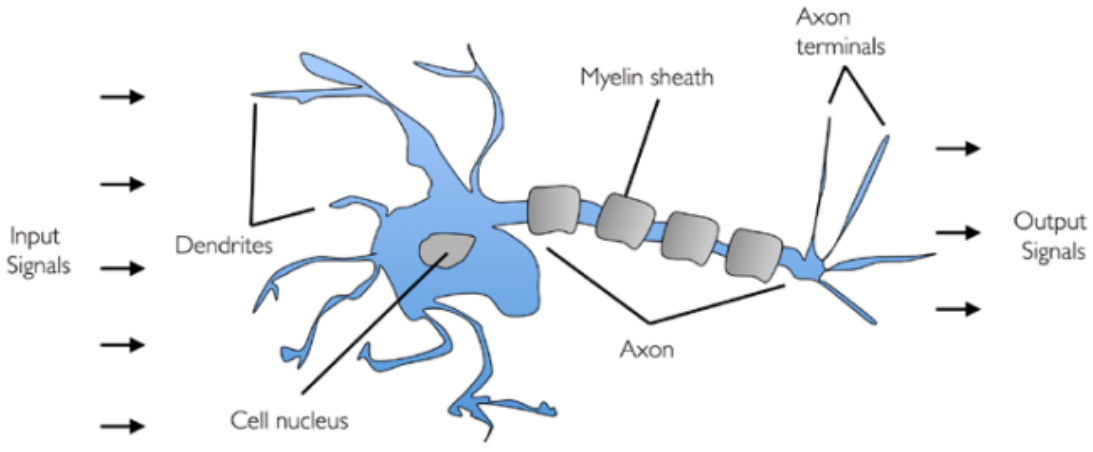
- **Supervised Learning:** In supervised learning the machine observes some example input–output pairs and learns a function that maps from input to output. A training set of examples with the correct responses (targets) is provided and, based on this training set, the algorithm generalises to respond correctly to all possible inputs. This is also called learning from **exemplars**. e.g driving under teacher.
- **Unsupervised Learning:** Correct responses are not provided, but instead the algorithm tries to identify similarities between the inputs so that inputs that have something in common are **categorised** together. The statistical approach to unsupervised learning is known as **density estimation**. e.g a taxi agent might gradually develop a concept of "good traffic days" and "bad traffic days"

- **reinforcement learning:** the agent learns from a series of reinforcements—rewards or punishments. The algorithm gets told when the answer is wrong, but does not get told how to correct it. It has to explore and try out different possibilities until it works out how to get the answer right. Reinforcement learning is sometime called learning with a **critic**. e.g lack of a tip means bad driving.

- **semi-supervised learning:** Here, the machine is given a large dataset, in which only a few data points are labelled. The algorithm will use clustering techniques (unsupervised learning) to identify groups within the given dataset and use the few labelled data points within each group to provide labels to other data points in the same cluster/group. Noise and lack of labels create a continuum between supervised and unsupervised learning. e.g identify cats and dogs with few labelled images.

Neural Network and Perceptron Learning

The processing unit of the brain are nerves cells called **neurons**, there are lots of them (100 billion = 10^{11} is the figure that is often given) and they come in lots of different types, depending upon their particular task. However, their general operation is similar in all cases: **transmitter chemicals** within the fluid of the brain raise or lower the electrical potential inside the body of the neuron. If this membrane potential reaches some threshold, the neuron **spikes or fires**, and a pulse of fixed strength and duration is sent down the **axon**. The axons divide (arborise) into connections to many other neurons, connecting to each of these neurons in a **synapse**. Each neuron is typically connected to thousands of other neurons, so that it is estimated that there are about 100 trillion (= 10^{14}) synapses within the brain. After firing, the neuron must wait for some time to recover its energy (the refractory period) before it can fire again.



Each neuron can be viewed as a separate processor, performing a very simple computation: deciding whether or not to fire. This makes the brain a massively parallel computer made up of 10^{11} processing elements

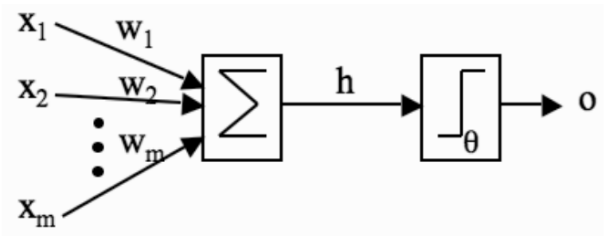
Hebb's Rule

So how does learning occur in the brain? The principal concept is **plasticity**- modifying the strength of synaptic connections between neurons, and creating new connections.

Hebb's rule says that the changes in the strength of synaptic connections are proportional to the correlation in the firing of the two connecting neurons. So if two neurons consistently fire simultaneously, then any connection between them will change in strength, becoming stronger. However, if the two neurons never fire simultaneously, the connection between them will die away.

McCulloch and Pitts Neuron's Model

Trying to understand how the biological brain works, in order to design artificial intelligence (AI), Warren McCulloch and Walter Pitts published the first concept of a simplified brain cell, the so-called **McCulloch-Pitts (MCP) neuron**, in 1943.



McCulloch and Pitts Model of neuron consists of :

1. **a set of weighted inputs** w_i that correspond to the synapses
2. **an adder** that sums the input signals (equivalent to the membrane of the cell that collects electrical charge)
3. **an activation function** (initially a threshold function) that decides whether the neuron fires ('spikes') for the current inputs

McCulloch and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, they are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

Only a few years later, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model (*The Perceptron: A Perceiving and Recognizing Automaton, F. Rosenblatt, Cornell Aeronautical Laboratory, 1957*). With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that would then be multiplied with the input features in order to make the decision of whether a neuron fires (transmits a signal) or not.

Rosenblatt's Perceptron Learning Rule

- **Initialisation**
  - set all of the weights w_{ij} to small (positive and negative) random numbers
- **Training**
  - for T iterations or until all the outputs are correct:
    - for each input vector:
      - compute the activation of each neuron j using activation function g:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } \sum_{i=0}^{m} w_{ij}x_i > 0 \\ 0 & \text{if } \sum_{i=0}^{m} w_{ij}x_i \leq 0 \end{cases}$$

      - update each of the weights individually using:

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i$$

- **Recall**
  - compute the activation of each neuron j using:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } w_{ij}x_i > 0 \\ 0 & \text{if } w_{ij}x_i \leq 0 \end{cases}$$

Machine Learning Process

1. Data Collection and Preparation
2. Feature Selection
3. Algorithm Choice
4. Parameter and Model Selection
5. Training
6. Evaluation

Supervised Learning

Introduction

The task of supervised learning is this:

Given a training set of N example input–output pairs

$$(x_1, y_1), (x_2, y_2), ...(x_N, y_N)$$

, where each y_j was generated by an unknown function $y = f(x)$, discover a function h that approximates the true function f.

Here x and y can be any value; they need not be numbers. The function h is a **hypothesis**. Learning is a search through the space of possible hypotheses for one that will perform well, even on new examples beyond the training set. To measure the accuracy of a hypothesis we give it a test set of examples that are distinct from the training set. We say a hypothesis generalizes well if it correctly predicts the value of y for novel examples. Sometimes the function f is stochastic—it is not strictly a function of x, and what we have to learn is a conditional probability distribution, $\mathbf{P}(Y|x)$.

When the output y is one of a finite set of values (such as sunny, cloudy or rainy),the learning problem is called classification, and is called Boolean or binary classification if there are only two values. When y is a number (such as tomorrow's temperature), the learning problem is called regression. (Technically, solving a regression problem is finding a conditional expectation or average value of y, because the probability that we have found exactly the right real-valued number for y is 0.)

Classification

Here the goal is to learn a mapping from inputs x to outputs y, where $y \epsilon \{1, ..., C\}$, with C being the number of classes. If $C = 2$, this is called **binary classification** (in which case we often assume $y \epsilon \{0, 1\}$); if $C > 2$, this is called **multiclass classification**. If the class labels are not mutually exclusive (e.g., somebody may be classified as tall and strong), we call it **multi-label classification**, but this is best viewed as predicting multiple related binary class labels (a so-called **multiple output model**).

One way to formalize the problem is as function approximation. We assume $y = f(x)$ for some unknown function f, and the goal of learning is to estimate the function f given a labeled training set, and then to make predictions using $\hat{y} = \hat{f}(x)$. Our main goal is to make predictions on novel inputs, meaning ones that we have

not seen before (this is called **generalization**), since predicting the response on the training set is easy (we can just look up the answer)

Real-world applications:-

Document classification and email spam filtering: In document classification, the goal is to classify a document, such as a web page or email message, into one of C classes, that is, to compute $p(y = c|x, D)$, where x is some representation of the text. A special case of this is email spam filtering, where the classes are spam $y = 1$ or ham $y = 0$.

Classifying flowers: The goal is to learn to distinguish three different kinds of iris flower, called setosa, versicolor and virginica. Fortunately, rather than working directly with images, a botanist has already extracted 4 useful features or characteristics: sepal length and width, and petal length and width. (Such feature extraction is an important, but difficult, task. Most machine learning methods use features chosen by some human.

Image classification and handwriting recognition We might want to classify the image as a whole, e.g., is it an indoors or outdoors scene? is it a horizontal or vertical photo? does it contain a dog or not? This is called **image classification**. In the special case that the images consist of isolated handwritten letters and digits, for example, in a postal or ZIP code on a letter, we can use classification to perform **handwriting recognition**.

Face detection and recognition A harder problem is to find objects within an image; this is called **object detection** or **object localization**. An important special case of this is face detection. One approach to this problem is to divide the image into many small overlapping patches at different locations, scales and orientations, and to classify each such patch based on whether it contains face-like texture or not. This is called a **sliding window detector**. The system then returns those locations where the probability of face is sufficiently high. Such face detection systems are built-in to most modern digital cameras; Having found the faces, one can then proceed to perform face recognition, which means estimating the identity of the person . In this case, the number of class labels might be very large. Also, the features one should use are likely to be different than in the face detection problem: for recognition, subtle differences between faces such as hairstyle may be important for determining identity, but for detection, it is important to be invariant to such details, and to just focus on the differences between faces and non-faces

Rigression

Rigression refers to the ability to predict values of continous variable. Regression is just like classification except the response variable is continuous.

Here are some examples of real-world regression problems.

- Predict tomorrow's stock market price given current market conditions and other possible side information.
- Predict the age of a viewer watching a given video on YouTube.
- Predict the location in 3d space of a robot arm end effector, given control signals (torques) sent to its various motors.
- Predict the amount of prostate specific antigen (PSA) in the body as a function of a number of different clinical measurements.
- Predict the temperature at any location inside a building using weather data, time, door sensors, etc.

Linear Regression Model

One of the most widely used models for regression is known as **linear regression**. This asserts that the response is a linear function of the inputs. This can be written as follows:

$$y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + \epsilon = \sum_{j=1}^{D} w_j x_j + \epsilon$$

where $\mathbf{w}^T$ represents the inner or **scalar product** between the input vector x and the model's **weight vector w**, and ϵ is the **residual error** between our linear predictions and the true response.

Univariate linear regression

A univariate linear function (a straight line) with input x and output y has the form $y = w_1 x + w_0$, where w_0 and w_1 are real-valued coefficients to be learned. We use the letter w because we think of the coefficients as **weights**; the value of y is changed by changing the relative weight of one term or another. We'll define $\mathbf{w}$ to be the vector $[w_0, w_1]$, and define

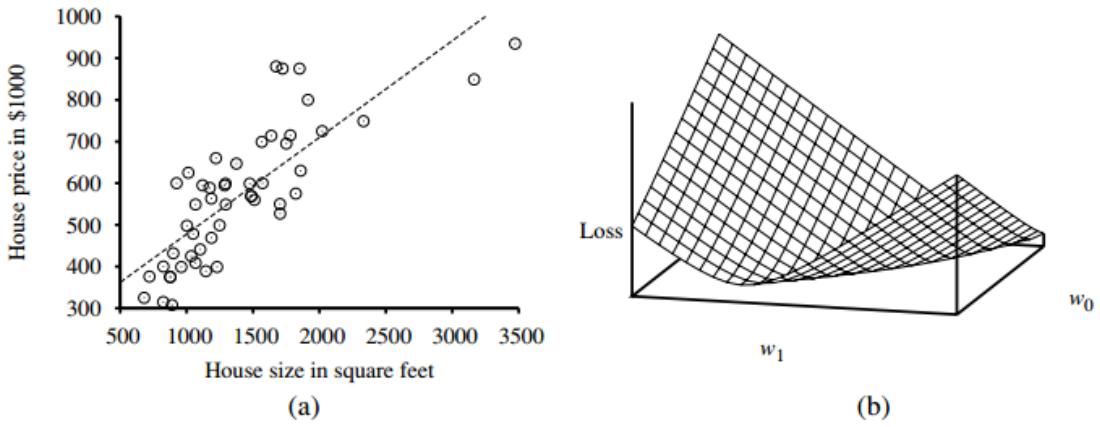
$$h_{\mathbf{w}}(x) = w_1 x + w_0$$

Figure shows an example of a training set of n points in the x, y plane, each point representing the size in square feet and the price of a house offered for sale. The task of finding the $h_{\mathbf{w}}$ that best fits these data is called **linear regression**. To fit a line to the data, all we have to do is find the values of the weights $[w_0, w_1]$ that minimize the empirical loss. It is traditional (going back to Gauss) to use the squared loss function, L_2, summed over all the training examples:

$$\text{Loss}\left(h_{\mathbf{w}}\right) = \sum_{j=1}^{N} L_2\left(y_j, h_{\mathbf{w}}\left(x_j\right)\right) = \sum_{j=1}^{N}\left(y_j - h_{\mathbf{w}}\left(x_j\right)\right)^2 = \sum_{j=1}^{N}\left(y_j - \left(w_1 x_j + w_0\right)\right)^2$$

We would like to find $\mathbf{w}^* = \text{argmin}_{\mathbf{w}} \text{Loss}(h_{\mathbf{w}})$ The sum $\sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2$ is minimized when its partial derivatives with respect to w_0 and w_1 are zero:

$$\frac{\partial}{\partial w_0} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0 \text{ and } \frac{\partial}{\partial w_1} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0$$



These equations have a unique solution:

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}; w_0 = \left(\sum y_j - w_1 \left(\sum x_j\right)\right)/N$$

Many forms of learning involve adjusting weights to minimize a loss, so it helps to have a mental picture of what's going on in weight space—the space defined by all possible settings of the weights. For univariate linear regression, the weight space defined by w_0 and w_1 is two-dimensional, so we can graph the loss as a function of w_0 and w_1 in a 3D plot (see Figure). We see that the loss function is **convex**, this is true for every linear regression problem with an $L2$ loss function, and implies that there are no local minima. In some sense that's the end of the story for linear models; if we need to fit lines to data, we apply Equation().

To go beyond linear models, we will need to face the fact that the equations defining minimum loss (as in Equation ()) will often have no closed-form solution. Instead, we will face a general optimization search problem in a continuous weight space. Such problems can be addressed by a hill-climbing algorithm that follows the gradient of the function to be optimized. In this case, because we are trying to minimize the loss, we will use **gradient descent**. We choose any starting point in weight space—here, a point in the (w_0, w_1) plane—and then move to a neighboring point that is downhill, repeating until we converge on the minimum possible loss:

$\mathbf{w} \leftarrow$ any point in the parameter space
loop until convergence **do**
for each w_i **in** w **do**

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w})$$

The parameter α, which we called the step size is usually called the **learning rate** when we are trying to minimize loss in a learning problem. It can be a fixed constant, or it can decay over time as the learning process proceeds.

For univariate regression, the loss function is a quadratic function, so the partial derivative will be a linear function. Let's first work out the partial derivatives—the slopes—in the simplified case of only one training example, (x, y):

$$\begin{aligned}\frac{\partial}{\partial w_i}\text{Loss}(\mathbf{w}) &= \frac{\partial}{\partial w_i}\left(y - h_w(x)\right)^2 \\ &= 2\left(y - h_{\mathbf{w}}(x)\right) \times \frac{\partial}{\partial w_i}\left(y - h_{\mathbf{w}}(x)\right) \\ &= 2\left(y - h_{\mathbf{w}}(x)\right) \times \frac{\partial}{\partial w_i}\left(y - (w_1 x + w_0)\right)\end{aligned}$$

applying this to both w_0 and w_1 we get:

$$\frac{\partial}{\partial w_0}\text{Loss}(\mathbf{w}) = -2\left(y - h_{\mathbf{w}}(x)\right); \quad \frac{\partial}{\partial w_1}\text{Loss}(\mathbf{w}) = -2\left(y - h_{\mathbf{w}}(x)\right) \times x$$

Then, plugging this back into Equation (), and folding the 2 into the unspecified learning rate α, we get the following learning rule for the weights:

$$w_0 \leftarrow w_0 + \alpha\left(y - h_{\mathbf{w}}(x)\right); \quad w_1 \leftarrow w_1 + \alpha\left(y - h_w(x)\right) \times x$$

These updates make intuitive sense: if $h_{\mathbf{w}}(x) > y$, i.e., the output of the hypothesis is too large, reduce w_0 a bit, and reduce w_1 if x was a positive input but increase w_1 if x was a negative input.

The preceding equations cover one training example. For N training examples, we want to minimize the sum of the individual losses for each example. The derivative of a sum is the sum of the derivatives, so we have:

$$w_0 \leftarrow w_0 + \alpha\sum_j\left(y_j - h_{\mathbf{w}}\left(x_j\right)\right); \quad w_1 \leftarrow w_1 + \alpha\sum_j\left(y_j - h_{\mathbf{w}}\left(x_j\right)\right) \times x_j$$

These updates constitute the **batch gradient descent** learning rule for univariate linear regression. Convergence to the unique global minimum is guaranteed (as long as we pick α small enough) but may be very slow: we have to cycle through all the training data for every step, and there may be many steps.

There is another possibility, called **stochastic gradient descent**, where we consider only a single training point at a time, taking a step after each one using Equation (). Stochastic gradient descent can be used in an online setting, where new data are coming in one at a time, or offline, where we cycle through the same data as many times as is necessary, taking a step after considering each single example. It is often faster than batch gradient descent. With a fixed learning rate α, however, it does not guarantee convergence; it can oscillate around the minimum without settling down. In some cases, as we see later, a schedule of decreasing learning rates (as in simulated annealing) does guarantee convergence.

Multivariate linear regression

We can easily extend to **multivariate linear regression** problems, in which each example x_j is an n-element vector. Our hypothesis space is the set of functions of the form

$$h_{sw}(\mathbf{x}_j) = w_0 + w_1 x_{j,1} + \cdots + w_n x_{j,n} = w_0 + \sum_i w_i x_{j,i}$$

The w_0 term, the intercept, stands out as different from the others. We can fix that by inventing a dummy input attribute, $x_{j,0}$, which is defined as always equal to 1. Then h is simply the dot product of the weights and the input vector (or equivalently, the matrix product of the transpose of the weights and the input vector):

$$h_{sw}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^\top \mathbf{x}_j = \sum_i w_i x_{j,i}$$

The best vector of weights, $\mathbf{w}^*$, minimizes squared-error loss over the examples:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j)$$

Multivariate linear regression is actually not much more complicated than the univariate case we just covered. Gradient descent will reach the (unique) minimum of the loss function; the update equation for each weight w_i is

$$w_i \leftarrow w_i + \alpha \sum_j x_{j,i} (y_j - h_{\mathbf{w}}(\mathbf{x}_j))$$

It is also possible to solve analytically for the w that minimizes loss. Let y be the vector of outputs for the training examples, and X be the data matrix, i.e., the matrix of inputs with one n-dimensional example per row. Then the solution

$$\mathbf{w}^* = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}$$

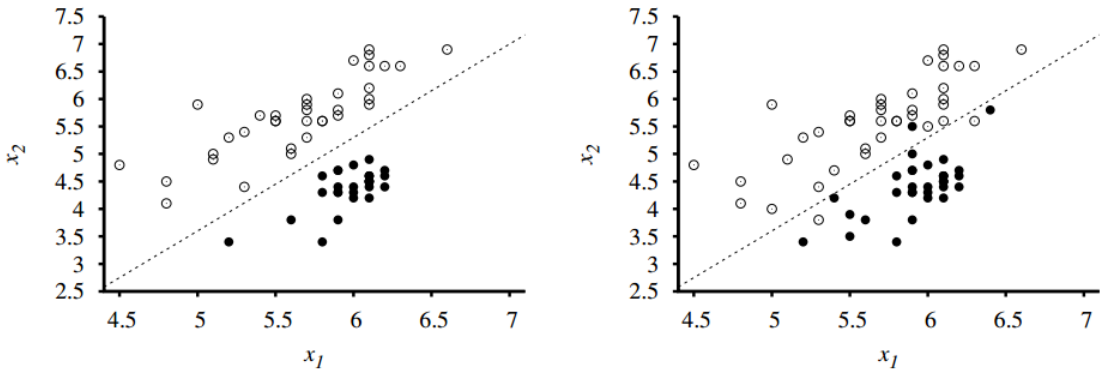
minimizes the squared error.

With univariate linear regression we didn't have to worry about **overfitting**. But with multivariate linear regression in high-dimensional spaces it is possible that some dimension that is actually irrelevant appears by chance to be useful, resulting in overfitting. Thus, it is common to use **regularization** on multivariate linear functions to avoid overfitting.

Logistic Regression

Linear classifiers with a hard threshold

Linear functions can be used to do classification as well as regression. For example, Figure a shows data points of two classes: earthquakes (which are of interest to seismologists) and underground explosions (which are of interest to arms control experts). Each point is defined by two input values, x_1 and x_2, that refer to body and surface wave magnitudes computed from the seismic signal. Given these training data, the task of classification is to learn a hypothesis h that will take new (x_1, x_2) points and return either 0 for earthquakes or 1 for explosions.



A **decision boundary** is a line (or a surface, in higher dimensions) that separates the two classes. In Figure 18.15(a), the decision boundary is a straight line. A linear decision boundary is called a **linear separator** and data that admit such a separator are called **linearly separable**. The linear separator in this case is defined by

$$x_2 = 1.7x_1 - 4.9 \quad \text{or} \quad -4.9 + 1.7x_1 - x_2 = 0$$

The explosions, which we want to classify with value 1, are to the right of this line with higher values of x_1 and lower values of x_2, so they are points for which $-4.9 + 1.7x_1 - x_2 > 0$, while earthquakes have $-4.9 + 1.7x_1 - x_2 < 0$. Using the convention of a dummy input $x_0 = 1$, we can write the classification hypothesis as

$$h_{\mathbf{w}}(\mathbf{x}) = 1 \text{ if } \mathbf{w} \cdot \mathbf{x} \geq 0 \text{ and } 0 \text{ otherwise.}$$

Alternatively, we can think of h as the result of passing the linear function $\mathbf{w} \cdot \mathbf{x}$ through a **threshold function**:

$$h_{\mathbf{w}}(\mathbf{x}) = \text{Threshold}(\mathbf{w} \cdot \mathbf{x}) \text{ where Threshold}(z) = 1 \text{ if } z \geq 0 \text{ and } 0 \text{ otherwise.}$$

The threshold function is shown in Figure(a).

Now that the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ has a well-defined mathematical form, we can think about choosing the weights w to minimize the loss. we did this both in closed form (by setting the gradient to zero and solving for the weights) and by gradient

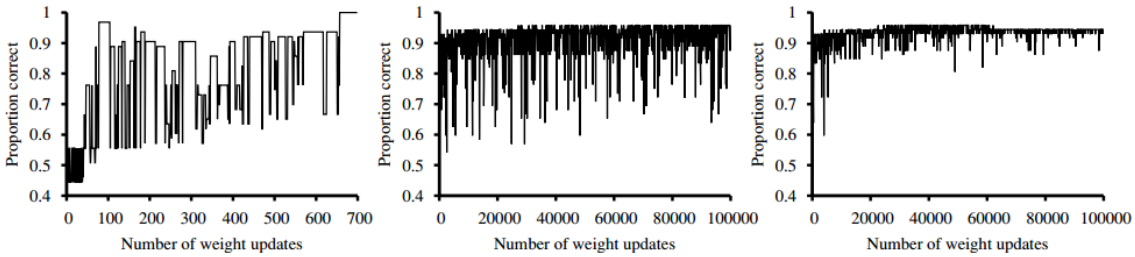
descent in weight space. Here, we cannot do either of those things because the gradient is zero almost everywhere in weight space except at those points where $w \cdot x = 0$, and at those points the gradient is undefined.

There is, however, a simple weight update rule that converges to a solution—that is, a linear separator that classifies the data perfectly–provided the data are linearly separable. For a single example $(\mathbf{x}, y)$, we have

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

which is essentially identical to the Equation (18.6), the update rule for linear regression! This rule is called the **perceptron learning rule**. Because we are considering a 0/1 classification problem, however, the behavior is somewhat different. Both the true value y and the hypothesis output $h_{\mathbf{w}}(\mathbf{x})$ are either 0 or 1, so there are three possibilities:

- If the output is correct, i.e., $y = h_{\mathbf{w}}(\mathbf{x})$, then the weights are not changed.
- If y is 1 but $h_{\mathbf{w}}(\mathbf{x})$ is 0, then w_i is increased when the corresponding input x_i is positive and decreased when xi is negative. This makes sense, because we want to make w · x bigger so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 1.
- If y is 0 but $h_{\mathbf{w}}(\mathbf{x})$ is 1, then w_i is decreased when the corresponding input x_i is positive and increased when xi is negative. This makes sense, because we want to make w · x smaller so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 0.



Typically the learning rule is applied one example at a time, choosing examples at random (as in stochastic gradient descent). Figure(a) shows a **training curve** for this learning rule applied to the earthquake/explosion data shown in (a). A training curve measures the classifier performance on a fixed training set as the learning process proceeds on that same training set. The curve shows the update rule converging to a zero-error linear separator. The "convergence" process isn't exactly pretty, but it always works.

Linear classification with logistic regression

We have seen that passing the output of a linear function through the threshold function creates a linear classifier; yet the hard nature of the threshold causes some problems: the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ is not differentiable and is in fact a discontinuous

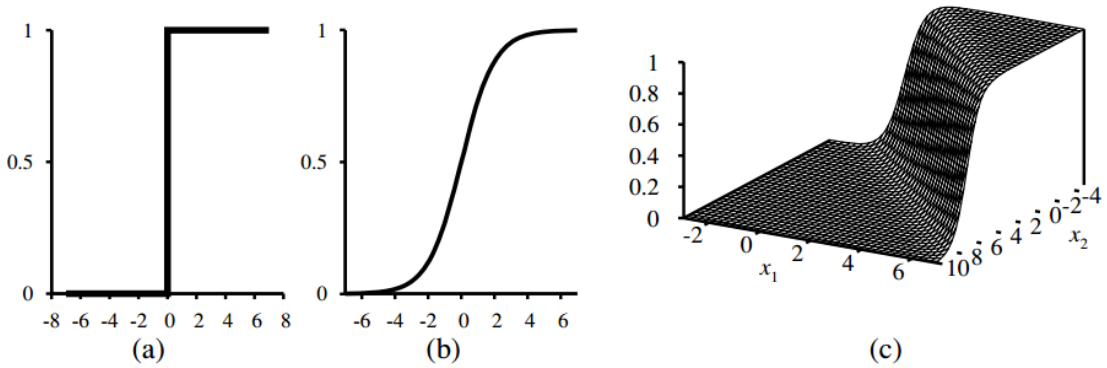
function of its inputs and its weights; this makes learning with the perceptron rule a very unpredictable adventure. Furthermore, the linear classifier always announces a completely confident prediction of 1 or 0, even for examples that are very close to the boundary; in many situations, we really need more gradated predictions.

All of these issues can be resolved to a large extent by softening the threshold function— approximating the hard threshold with a continuous, differentiable function.we saw two functions that look like soft thresholds: the integral of the standard normal distribution (used for the probit model) and the logistic function (used for the logit model). Although the two functions are very similar in shape, the logistic function

$$\text{Logistic}(z) = \frac{1}{1 + e^{-z}}$$

has more convenient mathematical properties. The function is shown in Figure(b). With the logistic function replacing the threshold function, we now have

$$h_{\mathbf{w}}(\mathbf{x}) = \text{Logistic}(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$



An example of such a hypothesis for the two-input earthquake/explosion problem is shown in Figure(c). Notice that the output, being a number between 0 and 1, can be interpreted as a probability of belonging to the class labeled 1. The hypothesis forms a soft boundary in the input space and gives a probability of 0.5 for any input at the center of the boundary region, and approaches 0 or 1 as we move away from the boundary.

The process of fitting the weights of this model to minimize loss on a data set is called **logistic regression**. There is no easy closed-form solution to find the optimal value of w with this model, but the gradient descent computation is straightforward. Because our hypotheses no longer output just 0 or 1, we will use the L2 loss function;

For a single example $(\mathbf{x}, y)$, the derivation of the gradient is the same as for linear regression up to the point where the actual form of h is inserted. (For this

derivation, we will need the chain rule:$\partial g(f(x))/\partial x = g'(f(x))\partial f(x)/\partial x$. We have

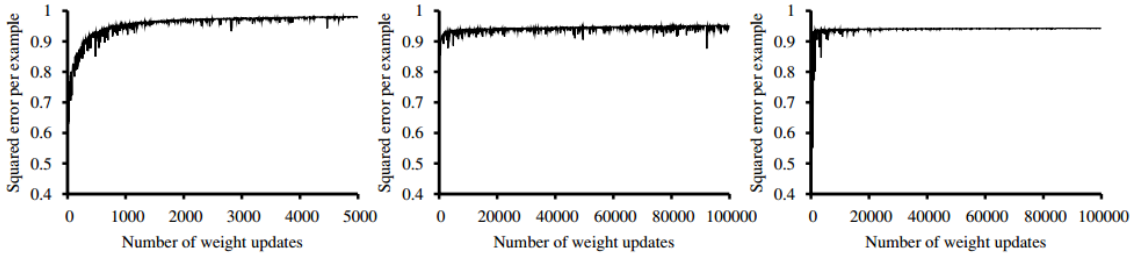
$$\begin{aligned}\frac{\partial}{\partial w_i}\text{loss}(\mathbf{w}) &= \frac{\partial}{\partial w_i}\left(y - h_{\mathbf{w}}(\mathbf{x})\right)^2 \\ &= 2\left(y - h_{\mathbf{w}}(\mathbf{x})\right) \times \frac{\partial}{\partial w_i}\left(y - h_{\mathbf{w}}(\mathbf{x})\right) \\ &= -2\left(y - h_{\mathbf{w}}(\mathbf{x})\right) \times g'(\mathbf{w}\cdot\mathbf{x}) \times \frac{\partial}{\partial w_i}\mathbf{w}\cdot\mathbf{x} \\ &= -2\left(y - h_{\mathbf{w}}(\mathbf{x})\right) \times g'(\mathbf{w}\cdot\mathbf{x}) \times x_i\end{aligned}$$

The derivative g' of the logistic function satisfies $g'(z) = g(z)(1 - g(z))$, so we have

$$g'(\mathbf{w}\cdot\mathbf{x}) = g(\mathbf{w}\cdot\mathbf{x})(1 - g(\mathbf{w}\cdot\mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})\left(1 - h_{\mathbf{w}}(\mathbf{x})\right)$$

so the weight update for minimizing the loss is

$$w_i \leftarrow w_i + \alpha\left(y - h_{\mathbf{w}}(\mathbf{x})\right) \times h_{\mathbf{w}}(\mathbf{x})\left(1 - h_{\mathbf{w}}(\mathbf{x})\right) \times x_i$$



Repeating the experiments with logistic regression instead of the linear threshold classifier, we obtain the results shown in Figure. In (a), the linearly separable case, logistic regression is somewhat slower to converge, but behaves much more predictably. In (b) and (c), where the data are noisy and nonseparable, logistic regression converges far more quickly and reliably. These advantages tend to carry over into real-world applications and logistic regression has become one of the most popular classification techniques for problems in medicine, marketing and survey analysis, credit scoring, public health, and other applications.For such learning tasks, the naive Bayes classifier is among the most effective algorithms known.

K nearest neighbor

Nonparametric Models

A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a **parametric model**. No matter how much data you throw at a parametric model, it won't change its mind about how many parameters it needs. When data sets are small, it makes sense to have a strong restriction on the allowable hypotheses, to avoid overfitting. But when there are thousands or millions or billions of examples to learn from, it seems like a better idea to let the data speak for themselves rather than forcing them to speak through a tiny vector of parameters.

A **nonparametric model** is one that cannot be characterized by a bounded set of parameters. For example, suppose that each hypothesis we generate simply retains within itself all of the training examples and uses all of them to predict the next example. Such a hypothesis family would be nonparametric because the effective number of parameters is unbounded — it grows with the number of examples. This approach is called instance-based learning or memory-based learning. The simplest instance-based learning method is **table lookup**. In contrast to learning methods that construct a general, explicit description of the target function when training examples are provided, instance-based learning methods simply store the training examples. Generalizing beyond these examples is postponed until a new instance must be classified. Each time a new query instance is encountered, its relationship to the previously stored examples is examined in order to assign a target function value for the new instance.

Nearest neighbor models

We can improve on table lookup with a slight variation: given a query $\mathbf{x}_q$, find the k examples that are *nearest* to $\mathbf{x}_q$. This is called k-**nearest neighbors** lookup. k-Nearest Neighbor algorithm is one of the most basic instance-based method. This algorithm assumes all instances correspond to points in the n-dimensional space. Let $NN(k, \mathbf{x}_q)$ denote the set of k nearest neighbours. To do classification, first find $NN(k, \mathbf{x}_q)$, then take the plurality vote of the neighbors (which is the majority vote in the case of binary classification). To avoid ties, k is always chosen to be an odd number. To do regression, we can take the mean or median of the k neighbors, or we can solve a linear regression problem on the neighbors.

The very word "nearest" implies a distance metric. How do we measure the distance from a query point $\mathbf{x}_q$ to an example point $\mathbf{x}_j$? Typically, distances are measured with a **Minkowski distance** or L^p, defined as

$$L^p(\mathbf{x}_j, \mathbf{x}_q) = \left(\sum_i |x_{j,i} - x_{q,i}|^p \right)^{1/p}$$

With $p = 2$ this is Euclidean distance and with $p = 1$ it is Manhattan distance. Often $p = 2$ is used if the dimensions are measuring similar properties, such as the width, height and depth of parts on a conveyor belt, and Manhattan distance is used

if they are dissimilar, such as age, weight, and gender of a patient. Note that if we use the raw numbers from each dimension then the total distance will be affected by a change in scale in any dimension.

To avoid this, it is common to apply **normalization** to the measurements in each dimension. One simple approach is to compute the mean μ_i and standard deviation σ_i of the values in each dimension, and rescale them so that $x_{j,i}$ becomes $(x_{j,i} - \mu_i)/\sigma_i$. A more complex metric known as the **Mahalanobis distance** takes into account the covariance between dimensions.

Let $a_r(x)$ denotes the value of the rth attribute of instance x.

$$\langle a_1(x), a_2(x), \ldots a_n(x) \rangle$$

Then the Euclidean distance between two instances x_i, and x_j is defined to be $d(x_i, X_j)$, where

$$d\left(x_i, x_j\right) \equiv \sqrt{\sum_{r=1}^{n} \left(a_r\left(x_i\right) - a_r\left(x_j\right)\right)^2}$$

In nearest-neighbor learning the target function may be either discrete-valued (classification) or real-valued (for regression). Let us discrete-valued target functions of the form $f : \mathbb{R}^n \to V$ where V is finite set $\{v_1, v_2, v_3 ...\}$

The k-Nearest Neighbor algorithm for approximating a discrete-valued target function is given as:

Training algorithm

- For each training example $< x, f(x) >$, add the example to the list *training examples*.

Classification algorithm

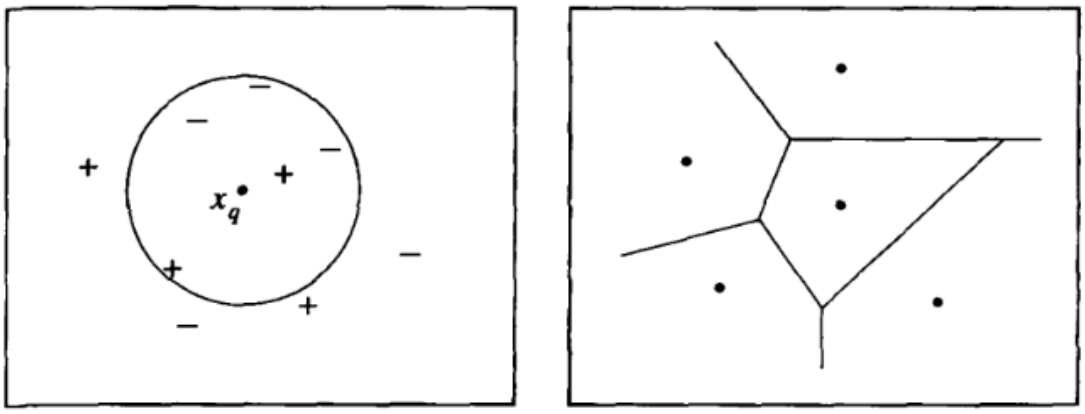
- Given a query instance xq to be classified,
  - Let $x_1, ..., x_k$ denote the k instances from *training examples* that are nearest to x_q
  - Return

$$\hat{f}\left(x_q\right) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta\left(v, f\left(x_i\right)\right)$$

  where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

As shown there, the value $\hat{f}(x_q)$ returned by this algorithm as its estimate of $f(x)$ is just the most common value of f among the k training examples nearest to xq. If we choose $k = 1$, then the 1-Nearest Neighbor algorithm assigns to $\hat{f}(x_q)$ the value $f(x_i)$ where x_i is the training instance nearest to x_q. For larger values of k, the algorithm assigns the most common value among the k nearest training examples.

Figure given below illustrates the operation of the k-Nearest Neighbor algorithm for the case where the instances are points in a two-dimensional space and where the target function is boolean valued. The positive and negative training examples are shown by "+" and "—" respectively. A query point x_q is shown as well. Note



the 1-Nearest Neighbor algorithm classifies xq as a positive example in this figure, whereas the 5-Nearest Neighbor algorithm classifies it as a negative example.

The Nearest Neighbor algorithm is easily adapted to approximating continuous-valued target functions. To accomplish this, we have die algorithm calculate the mean value of the k nearest training examples rather than calculate their most common value. More precisely, to approximate a real-valued target function.

$$\hat{f}\left(x_q\right) \leftarrow \frac{\sum_{i=1}^{k} f\left(x_i\right)}{k}$$

What is the nature of the hypothesis space H implicitly considered by the k-Nearest Neighbor algorithm? Note the k-Nearest Neighbor algorithm never forms an explicit general hypothesis regarding the target function. It simply computes the classification of each new query instance as needed. Nevertheless, we can still ask what the implicit general function is, or what classifications would be assigned if we were to hold the training examples constant and query the algorithm with every possible instance in X. The diagram on the right side of Figure shows the shape of this decision surface induced by 1-Nearest Neighbor over the entire instance space. The decision surface is a combination of convex polyhedra surrounding each of the training examples. For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the Voronoi diagram of the set of training examples.

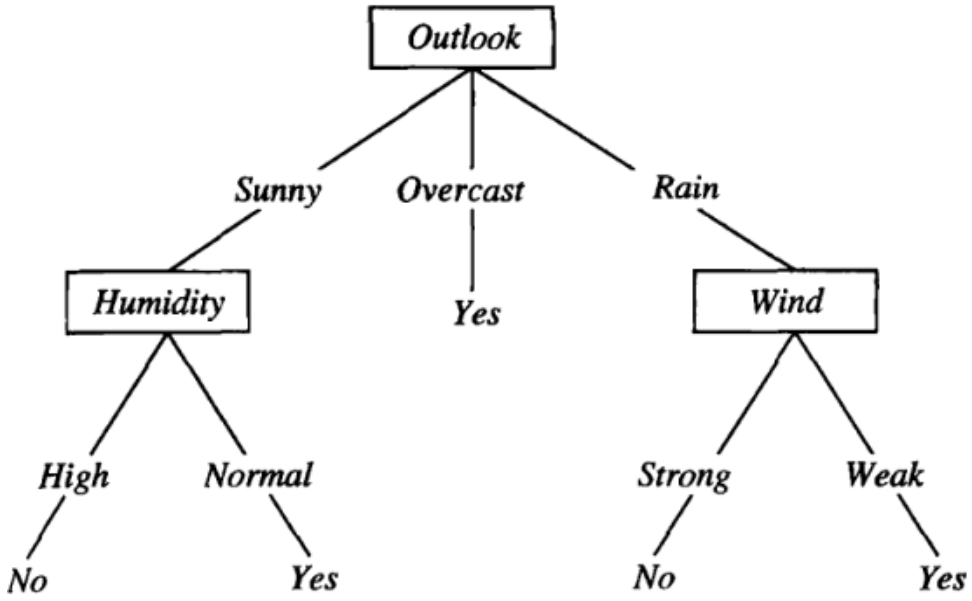
Decision Tree

All previously presented classifiers consider all p features at a time. This may be disadvantageous if not all features are necessary for a good classification, and if the assessment of the feature values causes a high effort. In medical diagnosis, for example, body temperature or blood pressure are measured for many patients because these features are assessed with low effort and yield significant information about the patient's health, whereas more expensive or time-consuming tests are only performed if needed to support a specific diagnosis. Depending on the already observed feature data, additional features may be ranked by importance, and only a subset of all possible features is used. This scheme leads to a *hierarchical structure* of the features, as opposed to the flat featurestructure of the classifiers presented so far. *Hierarchical classifiers* can be represented by decision trees.

A **decision tree** represents a function that takes as input a vector of attribute values and returns a "decision"—a single output value. The input and output values can be discrete or continuous. A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the input attributes, A_i, and the branches from the node are labeled with the possible values of the attribute, $A_i = v_{ik}$. Each leaf node in the tree specifies a value to be returned by the function.

Day	*Outlook*	*Temperature*	*Humidity*	*Wind*	*PlayTennis*
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

This decision tree classifies Saturday mornings according to whether they are suitable for playing tennis.



Expressiveness of decision trees

A Boolean decision tree is logically equivalent to the assertion that the goal attribute is true if and only if the input attributes satisfy one of the paths leading to a leaf with value true. Writing this out in propositional logic, we have

$$Goal \leftrightarrow (Path_1 \vee Path_2 \vee ...),$$

where each Path is a conjunction of attribute-value tests required to follow that path. Thus, the whole expression is equivalent to disjunctive normal form which means that any function in propositional logic can be expressed as a decision tree.

For a wide variety of problems, the decision tree format yields a nice, concise result. But some functions cannot be represented concisely. For example, the majority function, which returns true if and only if more than half of the inputs are true, requires an exponentially large decision tree.

Inducing decision trees from examples

We want a tree that is consistent with the examples and is as small as possible. Unfortunately, no matter how we measure size, it is an intractable problem to find the smallest consistent tree; there is no way to efficiently search through the 22n trees. With some simple heuristics, however, we can find a good approximate solution: a small (but not smallest) consistent tree. The DECISION-TREE-LEARNING algorithm adopts a greedy divide-and-conquer strategy: always test the most important attribute first. This test divides the problem up into smaller subproblems that can then be solved recursively. By "most important attribute," we mean the one that

makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be shallow.

In general, after the first attribute test splits up the examples, each outcome is a new decision tree learning problem in itself, with fewer examples and one less attribute. There are four cases to consider for these recursive problems:

1. If the remaining examples are all positive (or all negative), then we are done: we can answer Yes or No.

2. If there are some positive and some negative examples, then choose the best attribute to split them.

3. If there are no examples left, it means that no example has been observed for this combination of attribute values, and we return a default value calculated from the plurality classification of all the examples that were used in constructing the node's parent. These are passed along in the variable *parent examples*.

4. If there are no attributes left, but both positive and negative examples, it means that these examples have exactly the same description, but different classifications. This can happen because there is an error or **noise** in the data; because the domain is **nondeterministic**; or because we can't observe an attribute that would distinguish the examples. The best we can do is return the *plurality classification* of the remaining examples.

```
function DECISION-TREE-LEARNING(examples, attributes, parent_examples) returns
a tree

  if examples is empty then return PLURALITY-VALUE(parent_examples)
  else if all examples have the same classification then return the classification
  else if attributes is empty then return PLURALITY-VALUE(examples)
  else
      A ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
      tree ← a new decision tree with root test A
      for each value v_k of A do
          exs ← {e : e ∈ examples and e.A = v_k}
          subtree ← DECISION-TREE-LEARNING(exs, attributes − A, examples)
          add a branch to tree with label (A = v_k) and subtree subtree
      return tree
```

The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

Choosing attribute tests

The greedy search used in decision tree learning is designed to approximately minimize the depth of the final tree. The idea is to pick the attribute that goes as far as possible toward providing an exact classification of the examples. A perfect

attribute divides the examples into sets, each of which are all positive or all negative and thus will be leaves of the tree.

All we need, then, is a formal measure of "fairly good" and "really useless" and we can implement the IMPORTANCE function of algorithm. We will use the notion of **information gain**, which is defined in terms of **entropy**, the fundamental quantity in information theory. Entropy is a measure of the uncertainty of a random variable; acquisition of information corresponds to a reduction in entropy. A random variable with only one value—a coin that always comes up heads—has no uncertainty and thus its entropy is defined as zero; thus, we gain no information by observing its value. In general, the entropy of a random variable V with values v_k, each with probabilty $P(v_k)$, is defined as

$$H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k)$$

We define $B(q)$ as the entropy of a Boolean random variable that is true with probability q:

$$B(q) = -(q \log_2 q + (1-q) \log_2 (1-q))$$

If a training set contains p positive examples and n negative examples, then the entropy of the goal attribute on the whole set is:

$$H(Goal) = B(\frac{p}{p+n})$$

An attribute A with d distinct values divides the training set E into subsets $E_1, ..., E_d$. Each subset E_k has p_k positive examples and n_k negative examples, so if we go along that branch, we will need an additional $B(p_k/(p_k + n_k))$ bits of information to answer the question. A randomly chosen example from the training set has the kth value for the attribute with probability $(p_k + n_k)/(p + n)$, so the expected entropy remaining after testing attribute A is

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p+n} B(\frac{p_k}{P_k + n_k})$$

The **information gain** from the attribute test on A is the expected reduction in entropy:

$$Gain(A) = B(\frac{p}{p+n}) - Remainder(A)$$

Issues in Decision Tree Learning

Avoiding Overfitting the Data

When we fit highly flexible models, we need to be careful that we do not **overfit** the data, that is, we should avoid trying to model every minor variation in the input, since this is more likely to be noise than true signal.

Given a hypothesis space H, a hypothesis $h \in H$ is said to **overfit** the training data if there exists some alternative hypothesis $h' \in H$ such that h has smaller error

than h' over the training examples, but h' has a smaller error than hover the entire distribution of instances (i.e., including instances beyond the training set).

A general phenomenon, overfitting occurs with all types of learners, even when the target function is not at all random. Overfitting becomes more likely as the hypothesis space and the number of input attributes grows, and less likely as we increase the number of training examples.

There are several approaches to avoiding overfitting in decision tree learning.These can be grouped into two classes:

- approaches that stop growing the tree earlier, before it reaches the pointwhere it perfectly classifies the training data-**early stopping**.
- approaches that allow the tree to overfit the data, and then **post-prune** the tree.

Although the first of these approaches might seem more direct, the second approach of post-pruning overfit trees has been found to be more successful inpractice. This is due to the difficulty in the first approach of estimating precisely when to stop growing the tree. Also the problem with early stopping is that it stops us from recognizing situations where there is no one good attribute, but there are combinations of attributes that are informative.

decision tree pruning

Pruning works by eliminating nodes that are not clearly relevant. We start with a full tree, as generated by DECISION-TREE-LEARNING. We then look at a test node that has only leaf nodes as descendants. If the test appears to be irrelevant—detecting only noise in the data— then we eliminate the test, replacing it with a leaf node. We repeat this process, considering each test with only leaf descendants, until each one has either been pruned or accepted as is.

Suppose we are at a node consisting of p positive and n negative examples. If the attribute is irrelevant, we would expect that it would split the examples into subsets that each have roughly the same proportion of positive examples as the whole set, $p/(p+n)$, and so the information gain will be close to zero. Thus, the information gain is a good clue to irrelevance.

Now the question is, how large a gain should we require in order to split on a particular attribute? We can answer this question by using a **statistical significance** test. Such a test begins by assuming that there is no underlying pattern (the so-called **null hypothesis**). Then the actual data are analyzed to calculate the extent to which they deviate from a perfect absence of pattern. If the degree of deviation is statistically unlikely (usually taken to mean a 5% probability or less), then that is considered to be good evidence for the presence of a significant pattern in the data. The probabilities are calculated from standard distributions of the amount of deviation one would expect to see in random sampling.

In this case, the null hypothesis is that the attribute is irrelevant and, hence, that the information gain for an infinitely large sample would be zero. We need to calculate the probability that, under the null hypothesis, a sample of size $v = n + p$

would exhibit the observed deviation from the expected distribution of positive and negative examples. We can measure the deviation by comparing the actual numbers of positive and negative examples in each subset, p_k and n_k, with the expected numbers, $\hat{p}_k$ and $\hat{n}_k$, assuming true irrelevance:

$$\hat{p}_k = p \times \frac{p_k + n_k}{p + n} \; \hat{n}_k = n \times \frac{p_k + n_k}{p + n}$$

A convenient measure of the total deviation is given by

$$\Delta = \sum_{k=1}^{d} \frac{(p_k - \hat{p}_k)^2}{\hat{p}_k} + \frac{(n_k - \hat{n}_k)^2}{\hat{n}_k}$$

Under the null hypothesis, the value of Δ is distributed according to the χ^2 (chi-squared) distribution with $v - 1$ degrees of freedom. We can use a χ^2 table or a standard statistical library routine to see if a particular Δ value confirms or rejects the null hypothesis. This form of pruning is called χ^2**pruning**

1.0.1 Incorporating Continuous-Valued Attributes

Our initial definition of algorithm is restricted to attributes that take on a discrete setof values. First, the target attribute whose value is predicted by the learned treemust be discrete valued. Second, the attributes tested in the decision nodes ofthe tree must also be discrete valued. This second restriction can easily be removed so that continuous-valued decision attributes can be incorporated into thelearned tree. This can be accomplished by dynamically defining new discrete-valued attributes that partition the continuous attribute value into a discrete setof intervals. In particular, for an attribute A that is continuous-valued, the algorithm can dynamically create a new boolean attribute A_c that is true if $A < c$ and false otherwise. The only question is how to select the best value for the threshold c.

What threshold-based boolean attribute should be defined based on *Temperature*? Clearly, we would like to pick a threshold, c, that produces the greatest information gain. By sorting the examples according to the continuous attribute A, then identifying adjacent examples that differ in their target classification, wecan generate a set of candidate thresholds midway between the correspondingvalues of A. It can be shown that the value of c that maximizes information gainmust always lie at such a boundary. These candidate thresholds can then be evaluated by computing the information gain associated with each.In the current example, there are two candidate thresholds, corresponding to thevalues of Temperature at which the value of PlayTennis changes: (48 + 60)/2,and (80 + 90)/2. The information gain can then be computed for each of thecandidate attributes, $Temperature > 54$ and $Temperature > 85$, and the best can beselected ($Temperature > 54$). This dynamically created boolean attribute can thencompete with the other discrete-valued candidate attributes available for growingthe decision tree.