國立成功大學
National Cheng Kung University

Introduction to Artificial Intelligence

Chapter 19
Learning from Examples

Wei-Ta Chu (朱威達)

Introduction

- An agent is **learning** if it improves its performance on future tasks after making observations about the world.
- From a collection of input–output pairs, learn a function that predicts the output for new inputs.
- Why would we want an agent to learn?
  - (1) the designers cannot anticipate all possible situations
  - (2) the designers cannot anticipate all changes over time
  - (3) sometimes human programmers have no idea how to program a solution themselves

Supervised Learning

Given a **training set** of N example input–output pairs

$$(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N) \ ,$$

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

- 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.

Supervised Learning

- When the output y is one of a finite set of values, the learning problem is called **classification**. When y is a number, the learning problem is called **regression**.
- Fitting a function of a single variable to some data points.

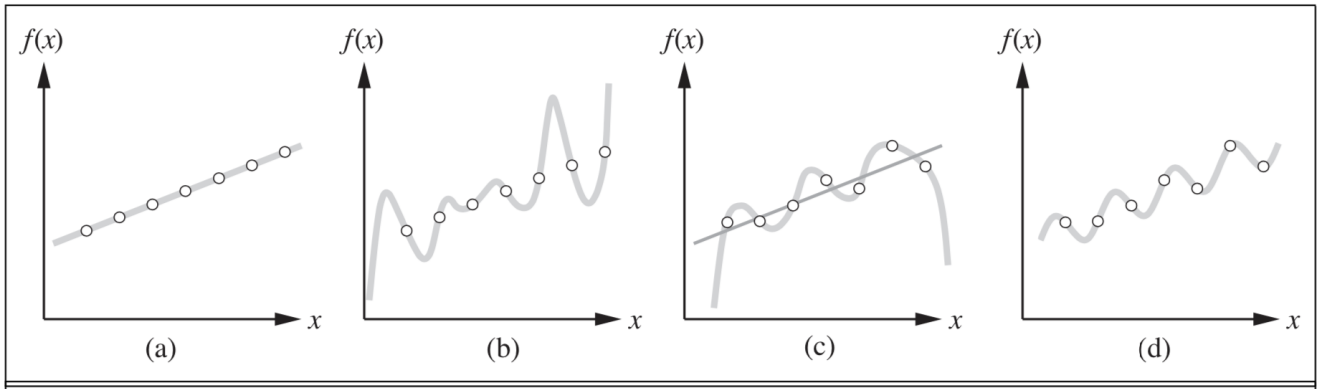


Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set, which admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data set.

Supervised Learning

- Figure 18.1(b) shows a high-degree polynomial that is also consistent with the same data. This illustrates a fundamental problem in inductive learning: how do we choose from among multiple consistent hypotheses? One answer is to prefer the simplest hypothesis consistent with the data. This principle is called **Ockham's razor**.
- Figure 18.1(c) shows a second data set. There is no consistent straight line for this data set; in fact, it requires a degree-6 polynomial for an exact fit. A straight line that is not consistent with any of the data points, but might generalize fairly well for unseen values of x, is also shown in (c).

Supervised Learning

- In general, there is a tradeoff between complex hypotheses that fit the training data well and simpler hypotheses that may generalize better. In Figure 18.1(d) we expand the hypothesis space H to allow polynomials over both x and $\sin(x)$, and find that the data in (c) can be fitted exactly by a simple function of the form $ax + b + c \sin(x)$. This shows the importance of the choice of hypothesis space.

Supervised Learning

- Supervised learning can be done by choosing the hypothesis h^* that is most probable given the data:

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmax}} P(h|data)$$

By Bayes' rule this is equivalent to

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmax}} P(data|h)\, P(h)$$

Then we can say that the prior probability $P(h)$ is high for a degree-1 or -2 polynomial, lower for a degree-7 polynomial.

Supervised Learning

- *There is a tradeoff between the expressiveness of a hypothesis space and the complexity of finding a good hypothesis within that space.*
- Fitting a straight line to data is an easy computation; fitting high-degree polynomials is somewhat harder.
- Most work on learning has focused on simple representations.

Learning Decision Trees

- We will build a decision tree to decide whether to wait for a table at a restaurant. The aim here is to learn a definition for the goal predicate *WillWait*.

1. *Alternate*: whether there is a suitable alternative restaurant nearby.
2. *Bar*: whether the restaurant has a comfortable bar area to wait in.
3. *Fri/Sat*: true on Fridays and Saturdays.
4. *Hungry*: whether we are hungry.
5. *Patrons*: how many people are in the restaurant (values are *None*, *Some*, and *Full*).
6. *Price*: the restaurant's price range ($, $$, $$$).
7. *Raining*: whether it is raining outside.
8. *Reservation*: whether we made a reservation.
9. *Type*: the kind of restaurant (French, Italian, Thai, or burger).
10. *WaitEstimate*: the wait estimated by the host (0–10 minutes, 10–30, 30–60, or >60).

Learning Decision Trees

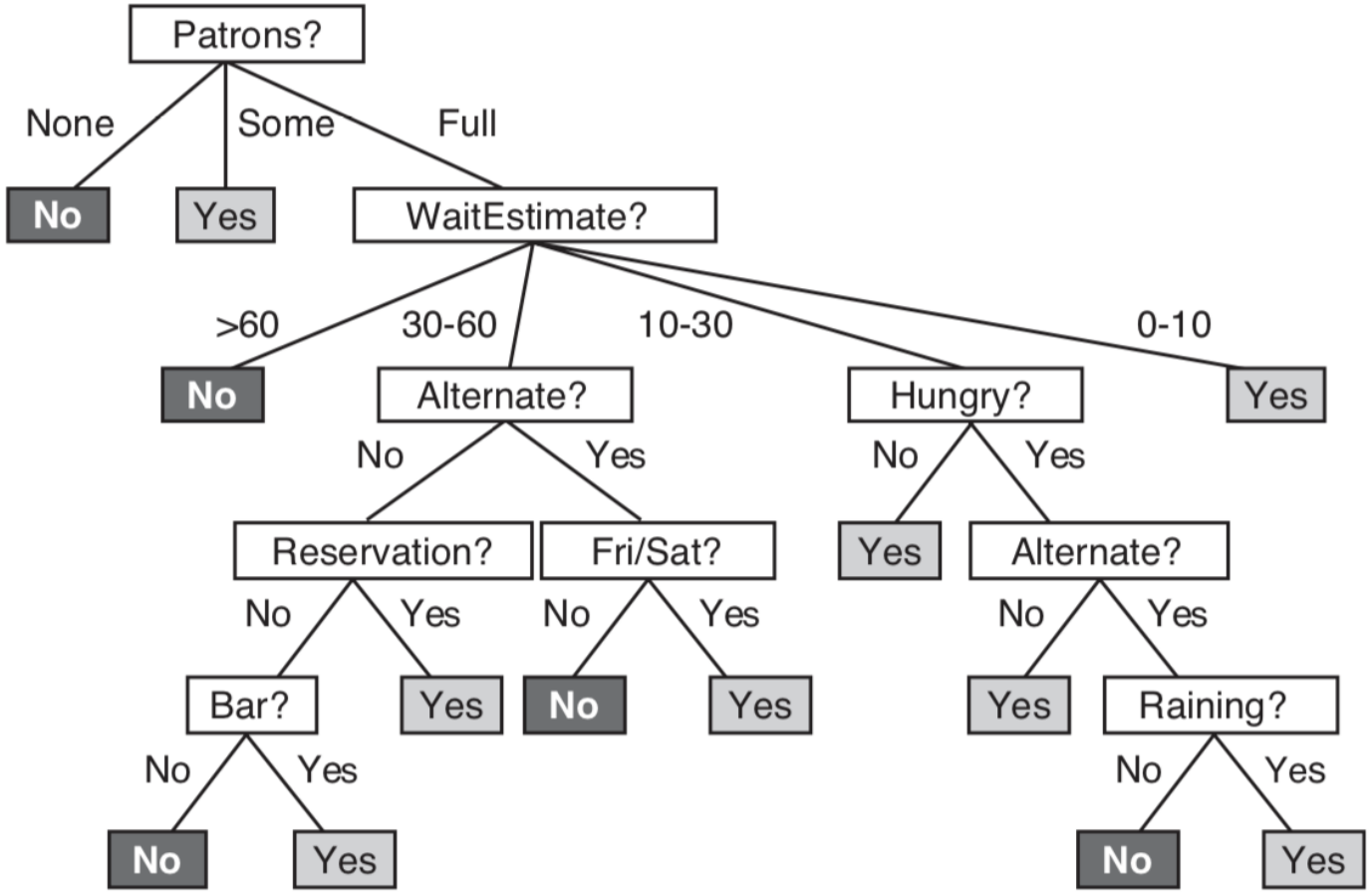


Figure 18.2 A decision tree for deciding whether to wait for a table.

Learning Decision Trees

- An example for a Boolean decision tree consists of an ($\mathbf{x}$, y) pair, where $\mathbf{x}$ is a vector of values for the input attributes, and y is a single Boolean output value. A training set of 12 examples is shown in Figure 18.3. The positive examples are the ones in which the goal *WillWait* is true ($\mathbf{x}_1$, $\mathbf{x}_3$, . . .); the negative examples are the ones in which it is false ($\mathbf{x}_2$, $\mathbf{x}_5$, . . .).

Example	Input Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y_1 = Yes$
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y_2 = No$
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y_3 = Yes$
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 = Yes$
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y_5 = No$
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 = Yes$
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 = No$
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 = Yes$
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y_9 = No$
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} = No$
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y_{11} = No$
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y_{12} = Yes$

Figure 18.3 Examples for the restaurant domain.

Learning Decision Trees

- We want a tree that is consistent with the examples and is as small as possible.
- **Always test the most important attribute first**. By "most important attribute," we mean the one that makes *the most difference to the classification of an example*.

- Figure 18.4(a) shows that *Type* is a poor attribute, because it leaves us with four possible outcomes, each of which has the same number of positive as negative examples.

Learning Decision Trees

- In (b), we see that *Patrons* is a fairly important attribute, because if the value is None or Some, then we are left with example sets for which we can answer definitively (No and Yes, respectively). If the value is Full, we are left with a mixed set of examples.
- 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.

Learning Decision Trees

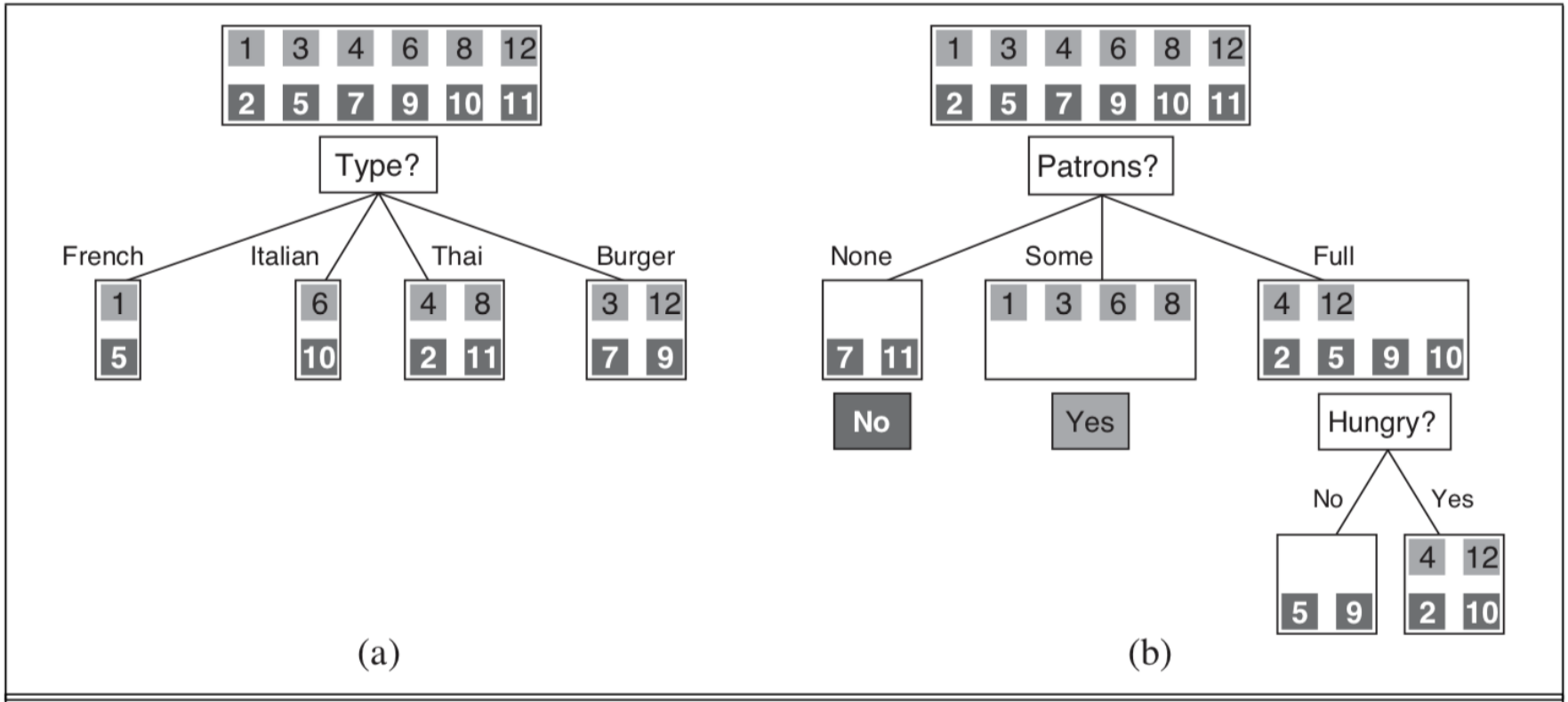


Figure 18.4 Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

Learning Decision Trees

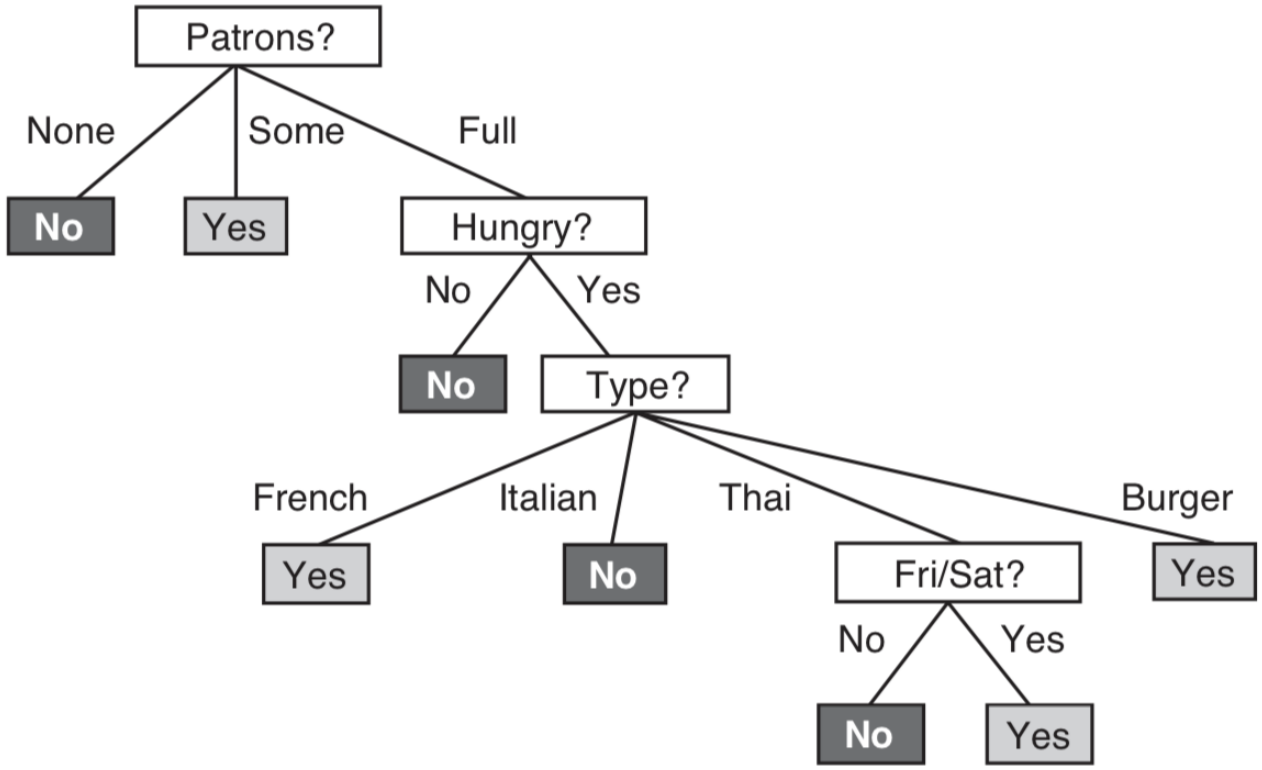


Figure 18.6 The decision tree induced from the 12-example training set.

Learning Decision Trees

- Choosing attribute tests
  - **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.
  - A flip of a fair coin is equally likely to come up heads or tails, 0 or 1, and we will soon show that this counts as "1 bit" of entropy. The roll of a fair four-sided die has 2 bits of entropy, because it takes two bits to describe one of four equally probable choices.

Learning Decision Trees

- Choosing attribute tests
  - The entropy of a random variable V with values v_k, each with probability $P(v_k)$, is defined as

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

We can check that the entropy of a fair coin flip is indeed 1 bit:

$$H(Fair) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1 .$$

If the coin is loaded to give 99% heads, we get

$$H(Loaded) = -(0.99 \log_2 0.99 + 0.01 \log_2 0.01) \approx 0.08 \text{ bits.}$$

Learning Decision Trees

- Choosing attribute tests
  - 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

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

  - The restaurant training set in Figure 18.3 has $p = n = 6$, so the corresponding entropy is $B(0.5)$ or exactly 1 bit. A test on a single attribute A might give us only part of this 1 bit. We can measure exactly how much by looking at the entropy remaining after the attribute test.

Learning Decision Trees

- Choosing attribute tests
  - An attribute A with d distinct values divides the training set E into subsets $E_1, \ldots, E_d$. 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

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

Learning Decision Trees

- Choosing attribute tests
  - The **information gain** from the attribute test on A is the *expected reduction in entropy*: $Gain(A) = B(\frac{p}{p+n}) - Remainder(A)$

In fact $Gain(A)$ is just what we need to implement the IMPORTANCE function. Returning to the attributes considered in Figure 18.4, we have

$$Gain(Patrons) = 1 - \left[\frac{2}{12}B(\frac{0}{2}) + \frac{4}{12}B(\frac{4}{4}) + \frac{6}{12}B(\frac{2}{6})\right] \approx 0.541 \text{ bits},$$

$$Gain(Type) = 1 - \left[\frac{2}{12}B(\frac{1}{2}) + \frac{2}{12}B(\frac{1}{2}) + \frac{4}{12}B(\frac{2}{4}) + \frac{4}{12}B(\frac{2}{4})\right] = 0 \text{ bits},$$

confirming our intuition that $Patrons$ is a better attribute to split on. In fact, $Patrons$ has the maximum gain of any of the attributes and would be chosen by the decision-tree learning algorithm as the root.

Learning Decision Trees

- In many areas of industry and commerce, decision trees are usually the first method tried when a classification method is to be extracted from a data set. One important property of decision trees is that it is possible for a human to understand the reason for the output of the learning algorithm. This is a property not shared by some other representations, such as neural networks.

Evaluating and Choosing the Best Hypothesis

- We want to learn a hypothesis that fits the future data best. To make that precise we need to define "future data" and "best." We make the stationarity assumption: that there is a probability distribution over examples that remains stationary over time.
- Each example data point (before we see it) is a random variable E_j whose observed value $e_j = (x_j, y_j)$ is sampled from that distribution, and is independent of the previous examples:

$$\mathbf{P}(E_j | E_{j-1}, E_{j-2}, \ldots) = \mathbf{P}(E_j)$$

and each example has an identical prior probability distribution:

$$\mathbf{P}(E_j) = \mathbf{P}(E_{j-1}) = \mathbf{P}(E_{j-2}) = \cdots$$

Evaluating and Choosing the Best Hypothesis

- Examples that satisfy these assumptions are called ***independent and identically distributed*** or **i.i.d**..
- The next step is to define "best fit." We define the ***error rate*** of a hypothesis as the proportion of mistakes it makes—the proportion of times that $h(x) \neq y$ for an (x, y) example. A hypothesis h has a low error rate on the training set does not mean that it will generalize well.
  - Randomly split the available data into a training set from which the learning algorithm produces h and a test set on which the accuracy of h is evaluated.

Evaluating and Choosing the Best Hypothesis

- ***k-fold cross-validation***: First we split the data into k equal subsets. We then perform k rounds of learning; on each round $1/k$ of the data is held out as a test set and the remaining examples are used as training data. The average test set score of the k rounds should then be a better estimate than a single score.
  - The extreme is $k = n$, also known as **leave-one-out cross-validation** or **LOOCV**.

Evaluating and Choosing the Best Hypothesis

- Users frequently invalidate their results by inadvertently **peeking** at the test data. A learning algorithm has various "knobs" (旋鈕) that can be twiddled to tune its behavior. The researcher generates hypotheses for various different settings of the knobs, measures their error rates on the test set, and reports the error rate of the best hypothesis. Alas, peeking has occurred!
- The reason is that the hypothesis was selected on the basis of its test set error rate, so information about the test set has leaked into the learning algorithm.

Evaluating and Choosing the Best Hypothesis

- Peeking is a consequence of using test-set performance to both choose a hypothesis and evaluate it. The way to avoid this is to really hold the test set out—lock it away until you are completely done with learning.
- If the test set is locked away, but you still want to measure performance on unseen data as a way of selecting a good hypothesis, then divide the available data (without the test set) into a training set and a **validation set**.

Evaluating and Choosing the Best Hypothesis

- Model selection: Complexity versus goodness of fit
  - In the polynomial fitting problem, choosing the degree of the polynomial is an instance of the problem of **model selection**.
  - Think of the task of finding the best hypothesis as two tasks: **model selection** defines the hypothesis space and then **optimization** finds the best hypothesis within that space.
  - Usually we start with the smallest, simplest models (which probably underfit the data), and iterate, considering more complex models at each step, until the models start to overfit.

Evaluating and Choosing the Best Hypothesis

- Model selection: Complexity versus goodness of fit

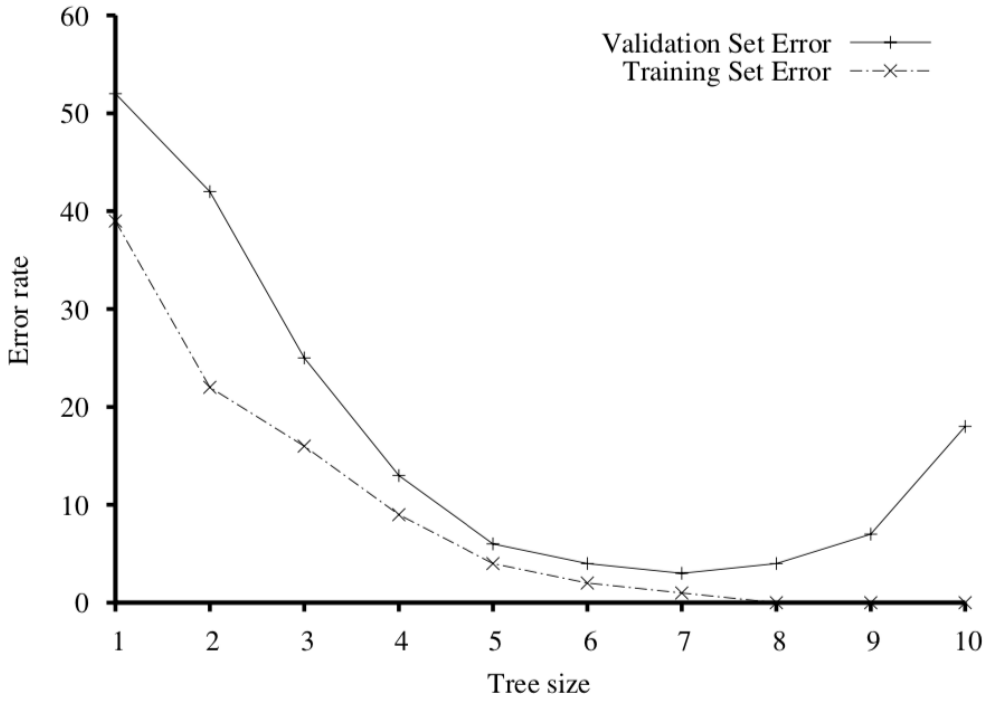


Figure 18.9 Error rates on training data (lower, dashed line) and validation data (upper, solid line) for different size decision trees. We stop when the training set error rate asymptotes, and then choose the tree with minimal error on the validation set; in this case the tree of size 7 nodes.

Evaluating and Choosing the Best Hypothesis

- From error rates to loss
  - In machine learning it is traditional to express utilities by means of a loss function.
  - The loss function $L(x, y, \hat{y})$ is defined as the amount of utility lost by predicting $h(x) = \hat{y}$ when the correct answer is $f(x) = y$:

$$\begin{aligned} L(x, y, \hat{y}) &= Utility(\text{result of using } y \text{ given an input } x) \\ &- Utility(\text{result of using } \hat{y} \text{ given an input } x) \end{aligned}$$

  This is the most general formulation of the loss function. Often a simplified version is used, $L(y, \hat{y})$.

Evaluating and Choosing the Best Hypothesis

- From error rates to loss
  - Two functions that implement that idea are the absolute value of the difference (called the L_1 loss), and the square of the difference (called the L_2 loss).
  - We can also use the $L_{0/1}$ loss function, which has a loss of 1 for an incorrect answer and is appropriate for discrete-valued outputs:

$$\begin{aligned}
&\text{Absolute value loss:} && L_1(y, \hat{y}) = |y - \hat{y}| \\
&\text{Squared error loss:} && L_2(y, \hat{y}) = (y - \hat{y})^2 \\
&\text{0/1 loss:} && L_{0/1}(y, \hat{y}) = 0 \text{ if } y = \hat{y}, \text{ else } 1
\end{aligned}$$

Evaluating and Choosing the Best Hypothesis

- From error rates to loss
  - The learning agent can theoretically maximize its expected utility by choosing the hypothesis that minimizes expected loss over all input–output pairs it will see.
  - It is meaningless to talk about this expectation without defining a prior probability distribution, $\mathbf{P}(X, Y)$ over examples. Let $\mathscr{E}$ be the set of all possible input–output examples. Then the expected generalization loss for a hypothesis h (with respect to loss function L) is

$$GenLoss_L(h) = \sum_{(x,y)\in\mathcal{E}} L(y, h(x))\, P(x, y)$$

Evaluating and Choosing the Best Hypothesis

- From error rates to loss
  - The best hypothesis, *h**, is the one with the minimum expected generalization loss: $h^* = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \, GenLoss_L(h)$
  - Because $P(x, y)$ is not known, the learning agent can only *estimate* generalization loss with **empirical loss** on a set of examples, E:

$$EmpLoss_{L,E}(h) = \frac{1}{N} \sum_{(x,y) \in E} L(y, h(x))$$

The estimated best hypothesis $\hat{h}^*$ is then the one with minimum empirical loss:

$$\hat{h}^* = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \, EmpLoss_{L,E}(h) \, .$$

Evaluating and Choosing the Best Hypothesis

- Regularization
  - Search for a hypothesis that directly minimizes the weighted sum of empirical loss and the complexity of the hypothesis, which we will call the total cost:

$$\begin{aligned} Cost(h) &= EmpLoss(h) + \lambda\, Complexity(h) \\ \hat{h}^* &= \operatorname*{argmin}_{h \in \mathcal{H}} Cost(h)\ . \end{aligned}$$

    We select the value of λ that gives us the best validation set score.
  - This process of explicitly penalizing complex hypotheses is called **regularization**.

Regression and Classification with Linear Models

- **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 (**weights**) to be learned.
  - We'll define **w** to be the vector $[w_0, w_1]$, and define

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

  - The task of finding the $h_{\mathbf{w}}$ that best fits these data is called **linear regression**.

Regression and Classification with Linear Models

- **Univariate linear regression**

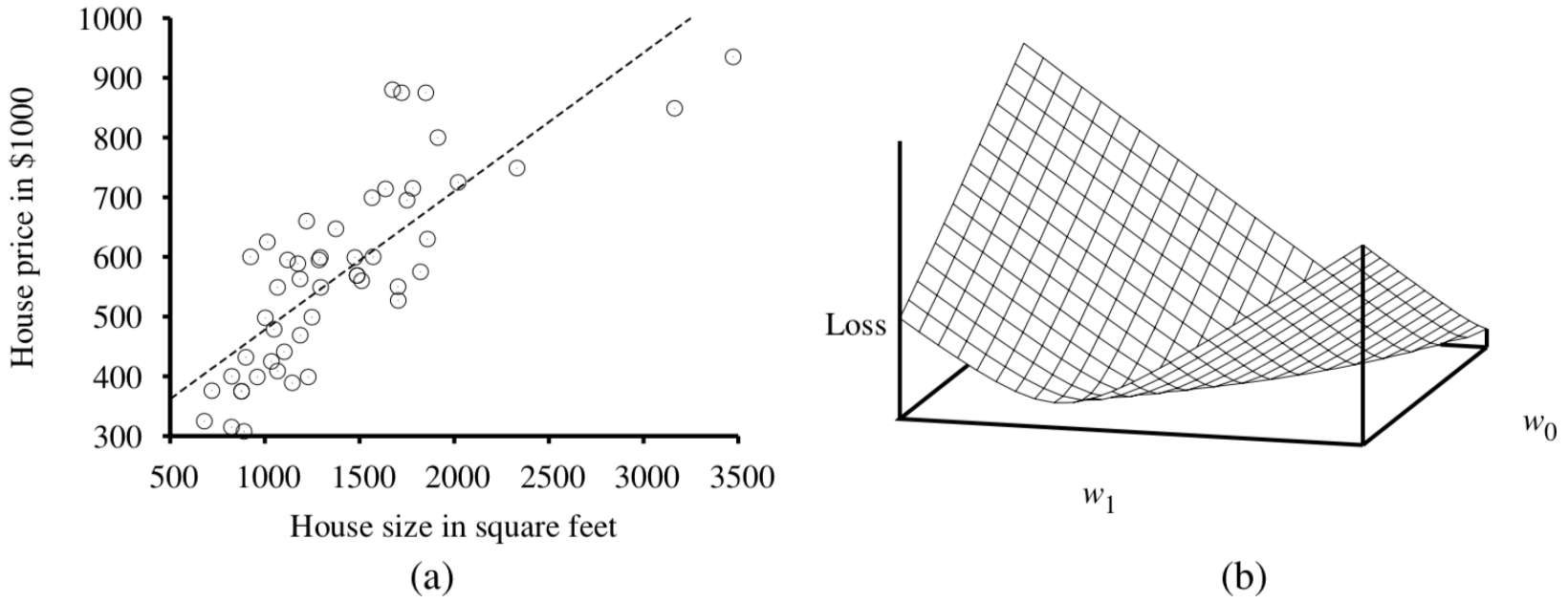


Figure 18.13 (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (w_1 x_j + w_0 - y_j)^2$ for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum.

Regression and Classification with Linear Models

- **Univariate linear regression**
  - To fit a line to the data, it is traditional to use the squared loss function, L_2, summed over all the training examples:

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

We would like to find $\mathbf{w}^* = \text{argmin}_{\mathbf{w}} Loss(h_{\mathbf{w}})$. 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:

$$\frac{\partial}{\partial w_0} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0 \text{ and } \frac{\partial}{\partial w_1} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0 \,. \quad (18.2)$$

These equations have a unique solution:

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}; \quad w_0 = (\sum y_j - w_1(\sum x_j))/N \,. \quad (18.3)$$

Regression and Classification with Linear Models

- **Univariate linear regression**
  - Many forms of learning involve adjusting weights to minimize a loss. Consider the **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.

Regression and Classification with Linear Models

- **Univariate linear regression**
  - To go beyond linear models, we will need to face the fact that the equations defining minimum loss will often have no closed-form solution.
  - Such problems can be addressed by a hill-climbing algorithm that follows the **gradient** of the function to be optimized.

$\mathbf{w} \leftarrow$ any point in the parameter space

loop until convergence **do**

  for each w_i **in w do**

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

Regression and Classification with Linear Models

- **Univariate linear regression**

$\mathbf{w} \leftarrow$ any point in the parameter space

loop until convergence **do**

  for each w_i **in** w **do**

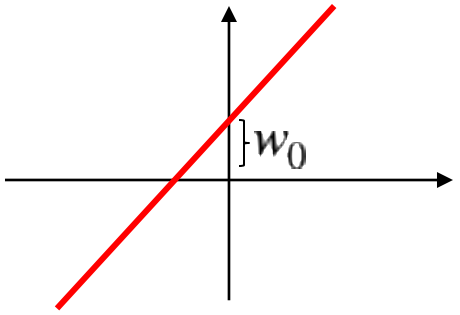
$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$$

  - The parameter α, which we called the **step size** in Section 4.2, 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.

Regression and Classification with Linear Models

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

- **Univariate linear regression**



$$\begin{aligned}\frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(x))^2 \\ &= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(x)) \\ &= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i}(y - (w_1 x + w_0)) \,, \end{aligned} \tag{18.5}$$

applying this to both w_0 and w_1 we get:

$$\frac{\partial}{\partial w_0} Loss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)) \,; \qquad \frac{\partial}{\partial w_1} Loss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)) \times x$$

Then, plugging this back into Equation (18.4), and folding the 2 into the unspecified learning rate α, we get the following learning rule for the weights:

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

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.

Regression and Classification with Linear Models

- **Univariate linear regression**
  - 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:

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

    These updates constitute the **batch gradient descent** learning rule for univariate linear regression.

Regression and Classification with Linear Models

- **Univariate linear regression**
  - **Stochastic gradient descent (SGD)**: It randomly selects a small number of training examples at each time step, and updates according to Equation (18.5).
  - SGD is widely applied to models other than linear regression, in particular neural networks.

Regression and Classification with Linear Models

- **Multivariate linear regression**
  - We can easily extend to **multivariate linear regression** problems, in which each example $\mathbf{x}_j$ is an n-element vector.

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

  - Then h is simply the dot product of the weights and the input vector

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

Regression and Classification with Linear Models

- **Multivariate linear regression**
  - The best vector of weights, **w***, minimizes squared-error loss over the examples: $$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j)$$
  - Gradient descent will reach the (unique) minimum of the loss function; the update equation for each weight w_i is $$w_i \leftarrow w_i + \alpha \sum_j x_{j,i}(y_j - h_{\mathbf{w}}(\mathbf{x}_j))$$

Regression and Classification with Linear Models

- **Multivariate linear regression**

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

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

minimizes the squared error.

- It is common to use **regularization** on multivariate linear functions to avoid overfitting. Recall that with regularization we minimize the total cost of a hypothesis, counting both the empirical loss and the complexity of the hypothesis: $Cost(h) = EmpLoss(h) + \lambda\, Complexity(h)$

Regression and Classification with Linear Models

- **Multivariate linear regression**
  - For linear functions the complexity can be specified as a function of the weights.

$$Complexity(h_{\mathbf{w}}) = L_q(\mathbf{w}) = \sum_i |w_i|^q$$

  - With $q = 1$ we have L_1 regularization, which minimizes the sum of the absolute values; with $q = 2$, L_2 regularization minimizes the sum of squares. Which regularization function should you pick? That depends on the specific problem, but L_1 regularization has an important advantage: it tends to produce a **sparse model**.

Regression and Classification with Linear Models

- **Linear classifiers with a hard threshold**
  - Linear functions can be used to do classification as well as regression.
  - 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.

Regression and Classification with Linear Models

- **Linear classifiers with a hard threshold**

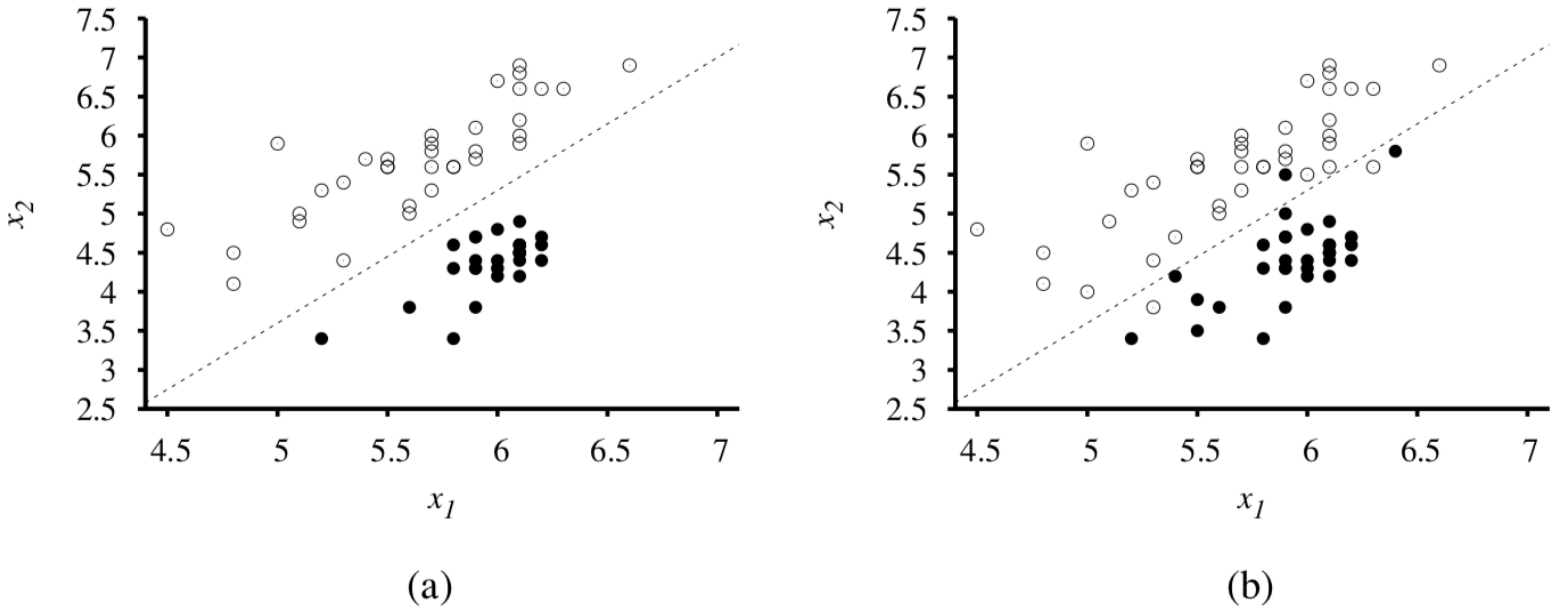


Figure 18.15 (a) Plot of two seismic data parameters, body wave magnitude x_1 and surface wave magnitude x_2, for earthquakes (white circles) and nuclear explosions (black circles) occurring between 1982 and 1990 in Asia and the Middle East (Kebeasy *et al.*, 1998). Also shown is a decision boundary between the classes. (b) The same domain with more data points. The earthquakes and explosions are no longer linearly separable.

Regression and Classification with Linear Models

- **Linear classifiers with a hard threshold**
  - 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**.

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

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

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

Regression and Classification with Linear Models

- **Linear classifiers with a hard threshold**
  - Alternatively, we can think of *h* as the result of passing the linear function $\boldsymbol{w} \cdot \boldsymbol{x}$ through a threshold function:

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

  - There is a simple weight update rule that converges to a solution provided the data are linearly separable. For a single example (x, y), we have

$$w_i \leftarrow w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x})) \times x_i \tag{18.7}$$

    which is essentially identical to the update rule for linear regression. This rule is called the **perceptron learning rule**.

Regression and Classification with Linear Models

- **Linear classifiers with a hard threshold**

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 x_i is negative. This makes sense, because we want to make $\mathbf{w} \cdot \mathbf{x}$ 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 x_i is negative. This makes sense, because we want to make $\mathbf{w} \cdot \mathbf{x}$ smaller so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 0.

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

Regression and Classification with Linear Models

- The perceptron rule may not converge to a stable solution for fixed learning rate.

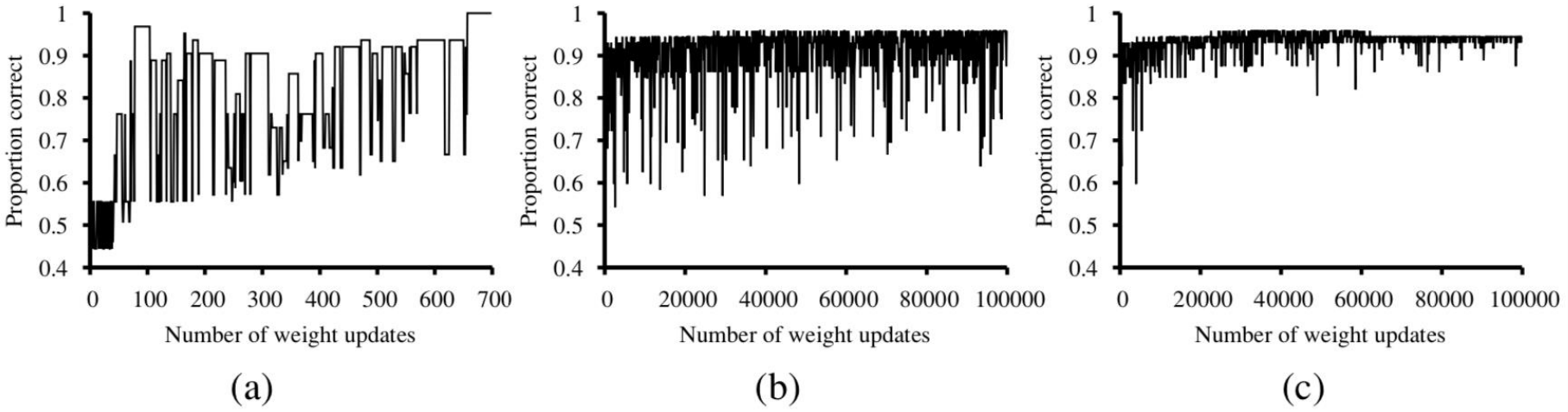


Figure 18.16 (a) Plot of total training-set accuracy vs. number of iterations through the training set for the perceptron learning rule, given the earthquake/explosion data in Figure 18.15(a). (b) The same plot for the noisy, non-separable data in Figure 18.15(b); note the change in scale of the x-axis. (c) The same plot as in (b), with a learning rate schedule $\alpha(t) = 1000/(1000 + t)$.

Regression and Classification with Linear Models

- **Linear classification with logistic regression**
  - 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.
  - The logistic function $Logistic(z) = \frac{1}{1+e^{-z}}$

Regression and Classification with Linear Models

- **Linear classification with logistic regression**

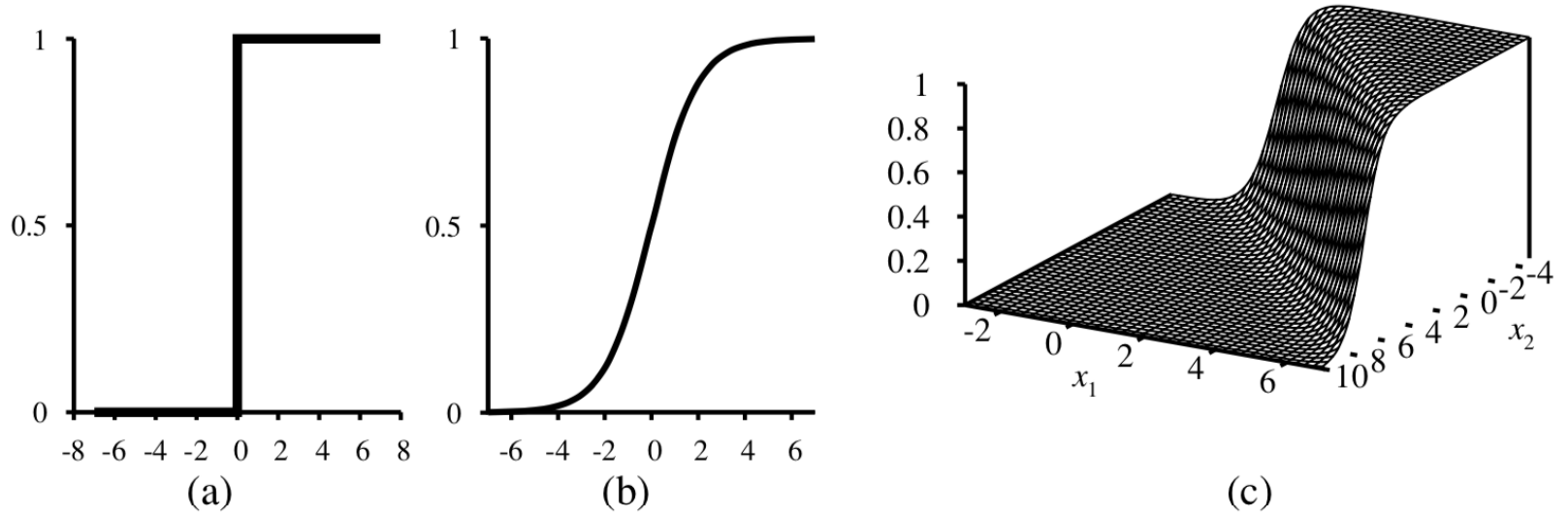


Figure 18.17 (a) The hard threshold function $Threshold(z)$ with 0/1 output. Note that the function is nondifferentiable at $z = 0$. (b) The logistic function, $Logistic(z) = \frac{1}{1+e^{-z}}$, also known as the sigmoid function. (c) Plot of a logistic regression hypothesis $h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x})$ for the data shown in Figure 18.15(b).

Regression and Classification with Linear Models

- **Linear classification with logistic regression**
  - With the logistic function replacing the threshold function, we now have

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

    An example of such a hypothesis for the two-input earthquake/explosion problem is shown in Figure 18.17(c).
  - The process of fitting the weights of this model to minimize loss on a data set is called **logistic regression**.

Regression and Classification with Linear Models

- **Linear classification with logistic regression**

For a single example $(\mathbf{x}, y)$, the derivation of the gradient is the same as for linear regression (Equation (18.5)) 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

$$\begin{aligned}
\frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x}))^2 \\
&= 2(y - h_{\mathbf{w}}(\mathbf{x})) \times \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x})) \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times \frac{\partial}{\partial w_i}\mathbf{w} \cdot \mathbf{x} \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times x_i \,.
\end{aligned}$$

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

Regression and Classification with Linear Models

- **Linear classification with logistic regression**

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

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

so the weight update for minimizing the loss is

$$w_i \leftarrow w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i\ . \tag{18.8}$$

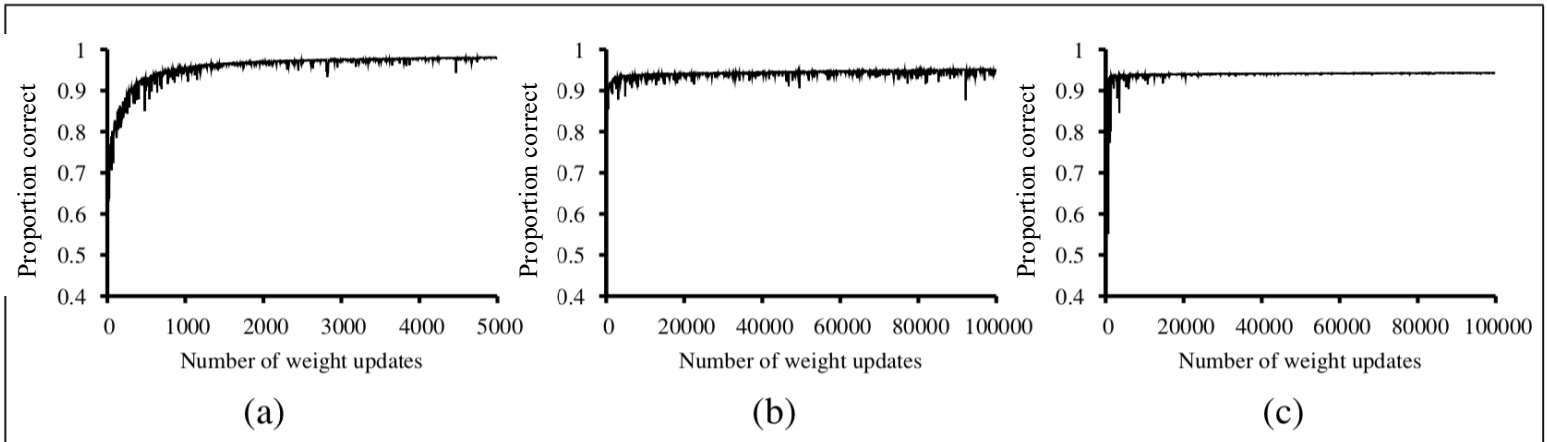


Figure 18.18 Repeat of the experiments in Figure 18.16 using logistic regression and squared error. The plot in (a) covers 5000 iterations rather than 1000, while (b) and (c) use the same scale.

Nonparametric Models

- Linear regression use the training data to estimate a fixed set of parameters **w**. That defines our hypothesis $h_{\mathbf{w}}(\mathbf{x})$, and at that point we can throw away the training data, because they are all summarized by **w**. A learning model that summarizes data with a set of parameters of fixed size is called a **parametric model**.
- 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.

Nonparametric Models

- **Nearest neighbor models**
  - Given a query $\mathbf{x}_q$, find the k examples that are *nearest* to $\mathbf{x}_q$. This is called *k-nearest neighbors* lookup. We'll use the notation $NN(k, \mathbf{x}_q)$ to denote the set of k nearest neighbors.
  - 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.

Nonparametric Models

- **Nearest neighbor models**

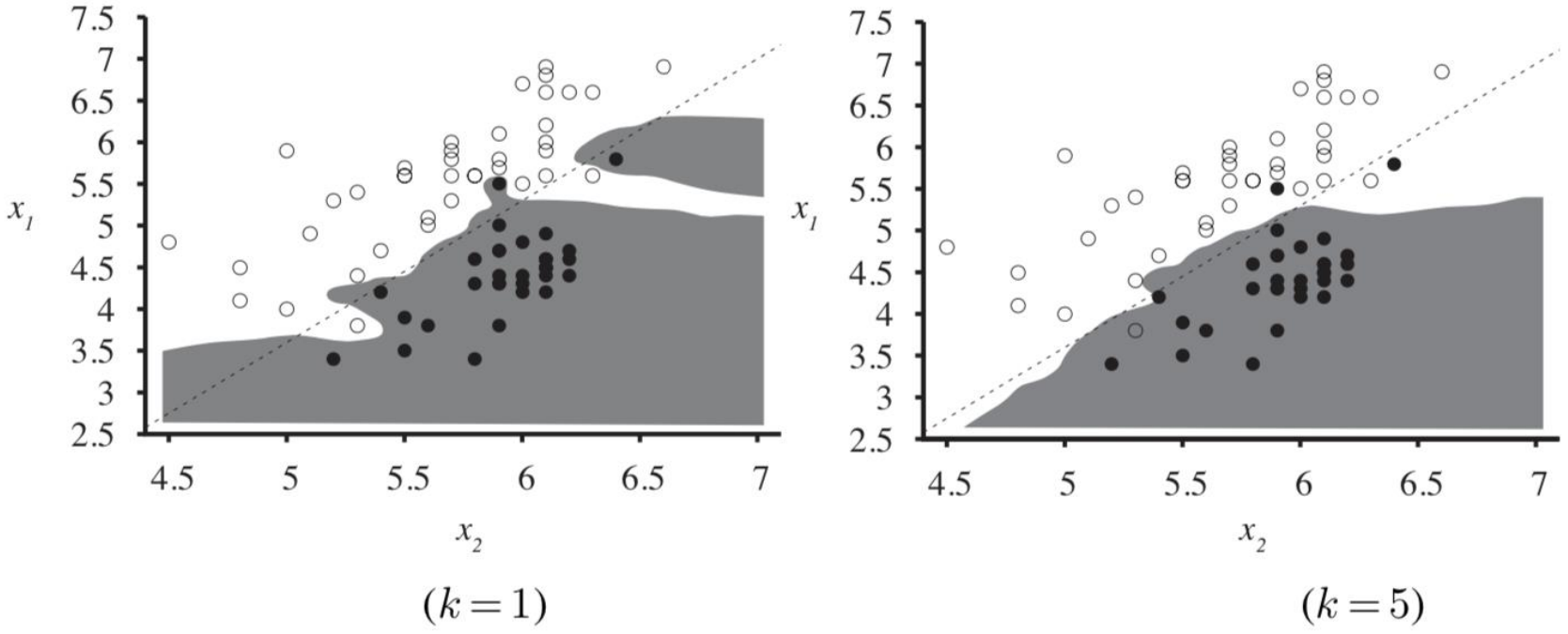


Figure 18.26 (a) A k-nearest-neighbor model showing the extent of the explosion class for the data in Figure 18.15, with $k = 1$. Overfitting is apparent. (b) With $k = 5$, the overfitting problem goes away for this data set.

Nonparametric Models

- **Nearest neighbor models**
  - Nonparametric methods are still subject to underfitting and overfitting, just like parametric methods. In this case 1-nearest neighbors is overfitting; it reacts too much to the black outlier in the upper right and the white outlier at (5.4, 3.7). The 5-nearest-neighbors decision boundary is good; higher k would underfit. As usual, cross-validation can be used to select the best value of k.

Nonparametric Models

- **Nearest neighbor models**
  - The 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 norm, defined as

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

    With $p = 2$ this is Euclidean distance and with $p = 1$ it is Manhattan distance.
  - With Boolean attribute values, the number of attributes on which the two points differ is called the **Hamming distance**.

Nonparametric Models

- **Nearest neighbor models**
  - The total distance will be affected by a change in scale in any dimension. If we change dimension i from centimeters to miles while keeping the other dimensions the same, we'll get different nearest neighbors.
  - 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.

Nonparametric Models

- **Curse of dimensionality**
  - In low-dimensional spaces with plenty of data, nearest neighbors works very well. However, it's not in high-dimensional spaces!
  - Consider k-nearest-neighbors on a data set of N points uniformly distributed throughout the interior of an n-dimensional unit hypercube. The k-neighborhood of a point is the smallest hypercube that contains the k-nearest neighbors. Let ℓ be the average side length of a neighborhood. Then the volume of the neighborhood (which contains k points) is ℓ^n and the volume of the full cube (which contains N points) is 1. So, on average, $\ell^n = k/N$. Taking nth roots of both sides we get $\ell = (k/N)^{1/n}$.

Nonparametric Models

- **Curse of dimensionality**
  - To be concrete, let k=10 and N=1,000,000. In two dimensions (n=2; a unit square), the average neighborhood has $\ell = 0.003$, a small fraction of the unit square, and in 3 dimensions ℓ is just 2% of the edge length of the unit cube. But by the time we get to 17 dimensions, ℓ is half the edge length of the unit hypercube, and in 200 dimensions it is 94%. This problem has been called the **curse of dimensionality**.

Nonparametric Models

- **Finding nearest neighbors with k-d trees**
  - A balanced binary tree over data with an arbitrary number of dimensions is called a **k-d tree**, for k-dimensional tree.
  - To construct a k-d tree, we start with a set of examples and at the root node we split them along the *i*th dimension by testing whether $x_i \leq m$. We chose the value m to be the median of the examples along the *i*th dimension; thus half the examples will be in the left branch of the tree and half in the right. We then recursively make a tree for the left and right sets of examples, stopping when there are fewer than two examples left.

Nonparametric Models

- **Finding nearest neighbors with k-d trees**
  - k-d trees are appropriate only when there are many more examples than dimensions, preferably at least 2^n examples. Thus, k-d trees work well with up to 10 dimensions with thousands of examples or up to 20 dimensions with millions of examples. If we don't have enough examples, lookup is no faster than a linear scan of the entire data set.

Nonparametric Models

- **Locality-sensitive hashing**
  - Hash tables have the potential to provide even faster lookup than binary trees. Hash codes randomly distribute values among the bins, but we want to have near points grouped together in the same bin; we want a **locality-sensitive hash** (LSH).

Nonparametric Models

- **Locality-sensitive hashing**
  - **Approximate near-neighbors problem**: Given a data set of example points and a query point $\mathbf{x}_q$, find, with high probability, an example point (or points) that is near $\mathbf{x}_q$.
  - To be more precise, we require that if there is a point $\mathbf{x}_j$ that is within a radius r of $\mathbf{x}_q$, then with high probability the algorithm will find a point $\mathbf{x}_{j'}$ that is within distance cr of $\mathbf{x}_q$. If there is no point within radius r then the algorithm is allowed to report failure. The values of c and "high probability" are parameters of the algorithm.

Nonparametric Models

- **Locality-sensitive hashing**
  - We need a hash function $g(\mathbf{x})$ that has the property that, for any two points $\mathbf{x}_j$ and $\mathbf{x}_{j'}$, the probability that they have the same hash code is small if their distance is more than cr, and is high if their distance is less than r.

Nonparametric Models

- **Locality-sensitive hashing**
  - The trick of LSH is to create *multiple* random projections and combine them. A random projection is just a random subset of the bit-string representation. We choose ℓ different random projections and create ℓ hash tables, $g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), \ldots, g_\ell(\boldsymbol{x})$. We then enter all the examples into each hash table. Then when given a query point $\mathbf{x}_q$, we fetch the set of points in bin $g_k(q)$ for each k, and union these sets together into a set of candidate points, C.

Nonparametric Models

- **Locality-sensitive hashing**
  - Then we compute the actual distance to $\mathbf{x}_q$ for each of the points in C and return the k closest points. With high probability, each of the points that are near to $\mathbf{x}_q$ will show up in at least one of the bins.
  - With large real-world problems, such as finding the near neighbors in a data set of 13 million Web images using 512 dimensions, locality-sensitive hashing needs to examine only a few thousand images out of 13 million to find nearest neighbors; a thousand-fold speedup over exhaustive or k-d tree approaches.

Nonparametric Models

- **Nonparametric regression**

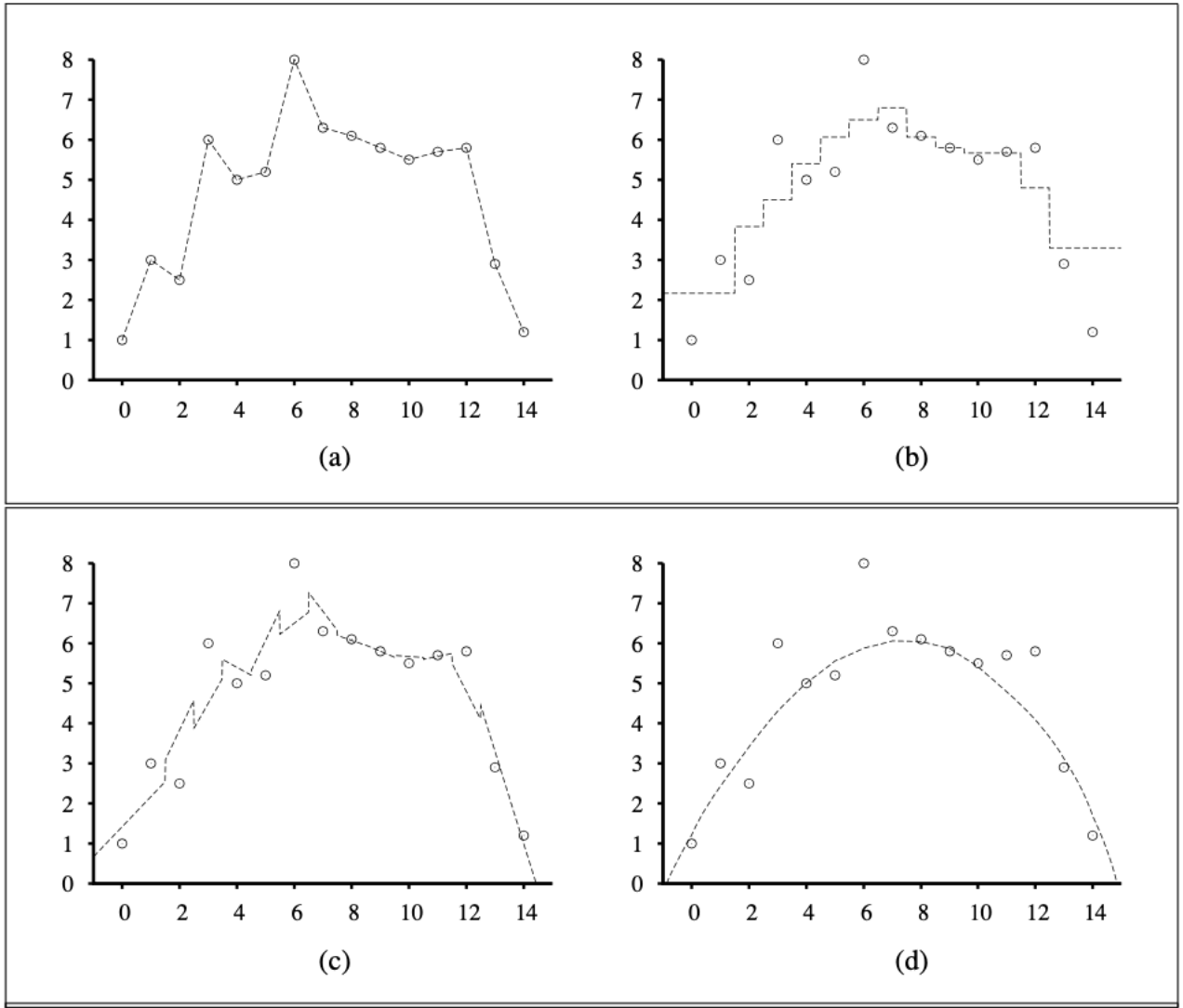


Figure 18.28 Nonparametric regression models: (a) connect the dots, (b) 3-nearest neighbors average, (c) 3-nearest-neighbors linear regression, (d) locally weighted regression with a quadratic kernel of width $k = 10$.

Nonparametric Models

- **Nonparametric regression**
  - (a) When noise is low, this trivial method is actually not too bad. But when the data are noisy, the resulting function is spiky, and does not generalize well.
  - (b) **k-nearest-neighbors regression** (Figure 18.28(b)) improves on connect-the-dots. Instead of using just the two examples to the left and right of a query point x_q, we use the k nearest neighbors (here 3). we have the k-nearest-neighbors average: $h(x)$ is the mean value of the k points,

$$\sum y_j / k$$

Nonparametric Models

- **Nonparametric regression**
  - (c) **k-nearest-neighbors linear regression** (Figure 18.28(c)) finds the best line through the k examples. This does a better job of capturing trends at the outliers, but is still discontinuous.
  - (d) **Locally weighted regression** (Figure 18.28(d)) gives us the advantages of nearest neighbors, without the discontinuities. The idea of locally weighted regression is that at each query point x_q, the examples that are close to x_q are weighted heavily, and the examples that are farther away are weighted less heavily or not at all. The decrease in weight over distance is always gradual, not sudden.

Nonparametric Models

- **Nonparametric regression**
  - We decide how much to weight each example with a function known as a **kernel**. A kernel function looks like a bump; in Figure 18.29 we see the specific kernel used to generate Figure 18.28(d).

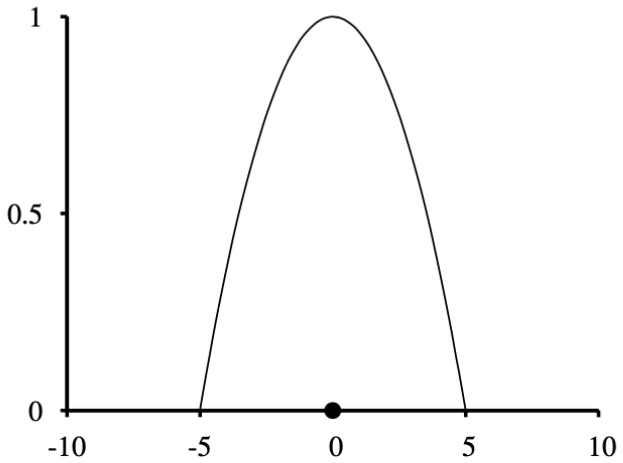


Figure 18.29 A quadratic kernel, $\mathcal{K}(d) = \max(0, 1 - (2|x|/k)^2)$, with kernel width $k = 10$, centered on the query point $x = 0$.

Nonparametric Models

- **Nonparametric regression**

Doing locally weighted regression with kernels is now straightforward. For a given query point $\mathbf{x}_q$ we solve the following weighted regression problem using gradient descent:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j \mathcal{K}(Distance(\mathbf{x}_q, \mathbf{x}_j)) \, (y_j - \mathbf{w} \cdot \mathbf{x}_j)^2 \, ,$$

where *Distance* is any of the distance metrics discussed for nearest neighbors. Then the answer is $h(\mathbf{x}_q) = \mathbf{w}^* \cdot \mathbf{x}_q$.

Support Vector Machines

- In the early 2000s, the support vector machines (SVM) model class was the most popular approach for "off-the-shelf" supervised learning, for when you don't have any specialized prior knowledge about a domain.
  - SVMs construct a **maximum margin separator**
  - SVMs create a linear separating hyperplane, but they have the ability to embed the data into a higher-dimensional space, using the so-called **kernel trick**.
  - SVMs are a nonparametric method

Support Vector Machines

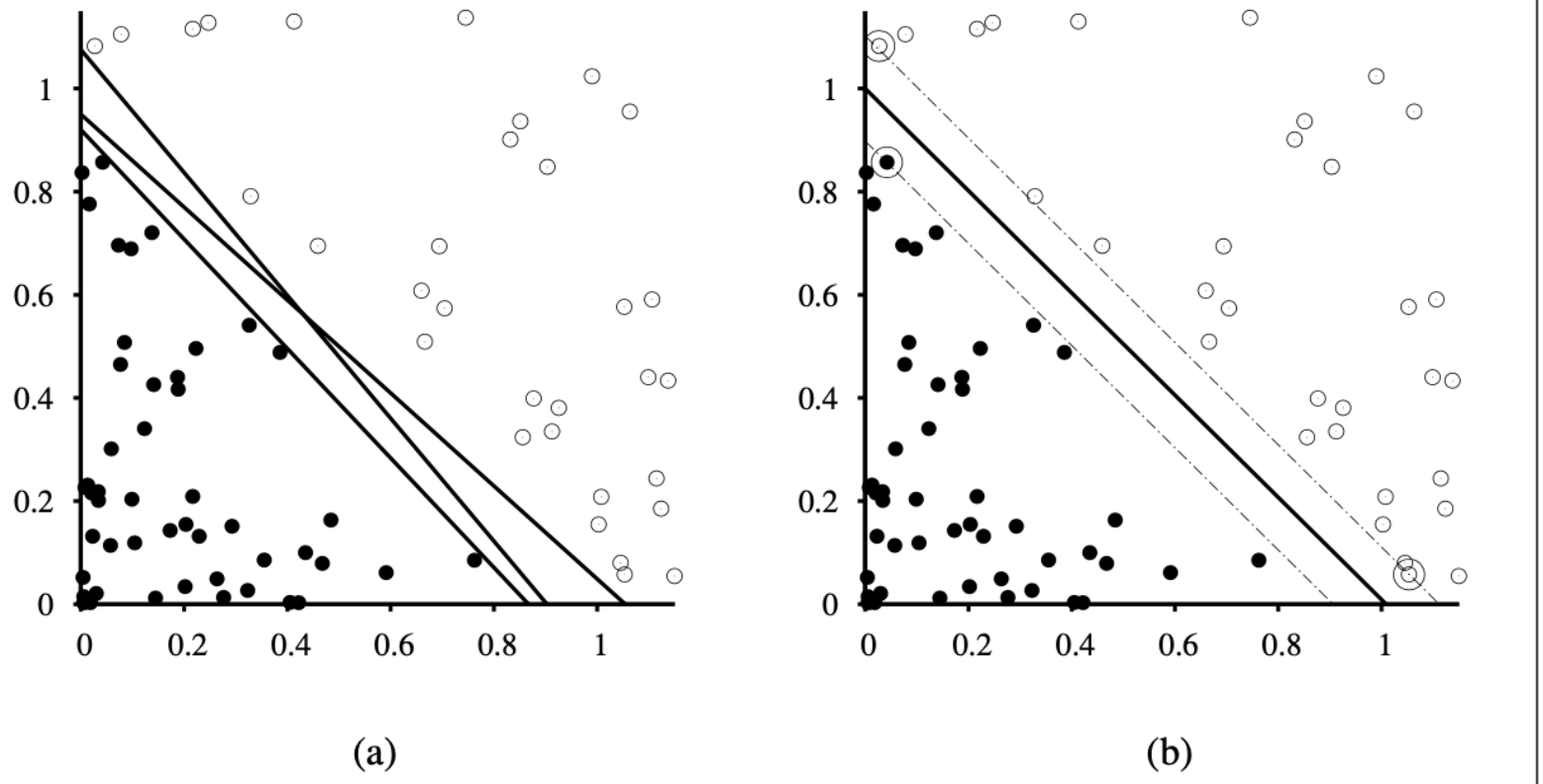


Figure 18.30 Support vector machine classification: (a) Two classes of points (black and white circles) and three candidate linear separators. (b) The maximum margin separator (heavy line), is at the midpoint of the **margin** (area between dashed lines). The **support vectors** (points with large circles) are the examples closest to the separator.

Support Vector Machines

- In Figure 18.30(a), we have a binary classification problem with three candidate decision boundaries, each a linear separator. Each of them is consistent with all the examples, so from the point of view of 0/1 loss, each would be equally good.
- Logistic regression would find some separating line; the exact location of the line depends on all the example points. The key insight of SVMs is that **some examples are more important than others, and that paying attention to them can lead to better generalization**.

Support Vector Machines

- SVMs address this issue: Instead of minimizing expected *empirical loss* on the training data, SVMs attempt to minimize expected *generalization* loss.
- We call this separator, shown in Figure 18.30(b) the **maximum margin separator**. The **margin** is the width of the area bounded by dashed lines in the figure—twice the distance from the separator to the nearest example point.
- How do we find this separator?

Support Vector Machines

- The separator is defined as the set of points $\{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} + b=0\}$. We could search the space of $\mathbf{w}$ and b with gradient descent to find the parameters that maximize the margin while correctly classifying all the examples.
- We don't directly solve it, but turns it into the dual representation. The optimal solution is found by solving

$$\underset{\alpha}{\operatorname{argmax}} \sum_j \alpha_j - \frac{1}{2} \sum_{j,k} \alpha_j \alpha_k y_j y_k (\mathbf{x}_j \cdot \mathbf{x}_k) \qquad (18.13)$$

$$\text{subject to the constraints } \alpha_j \geq 0 \text{ and } \textstyle\sum_j \alpha_j y_j = 0$$

Support Vector Machines

- This is a **quadratic programming** optimization problem.
- Once we have found the vector α we can get back to $\mathbf{w}$ with the equation $\mathbf{w} = \sum_j \alpha_j \mathbf{x}_j$, or we can stay in the dual representation.
- There are three important properties of Equation (18.13). First, the expression is convex; it has a single global maximum that can be found efficiently.

Support Vector Machines

- Second, *the data enter the expression only in the form of dot products of pairs of points*. This second property is also true of the equation for the separator itself; once the optimal α_i have been calculated, it is

$$h(\mathbf{x}) = \text{sign}\left(\sum_j \alpha_j y_j (\mathbf{x} \cdot \mathbf{x}_j) - b\right)$$

- A final important property is that the weights α_j associated with each data point are zero except for the **support vectors**—the points closest to the separator. Because there are usually many fewer support vectors than examples, SVMs gain some of the advantages of parametric models.

Support Vector Machines

- What if the examples are not linearly separable?

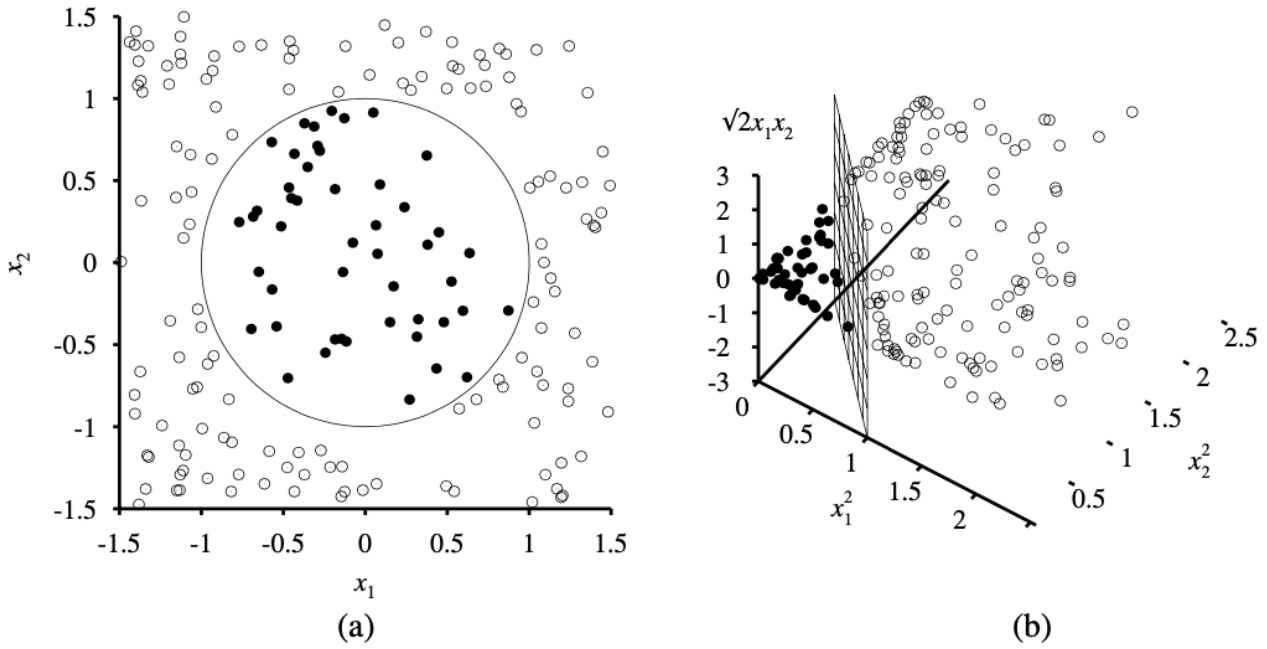


Figure 18.31 (a) A two-dimensional training set with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions. Figure 18.30(b) gives a closeup of the separator in (b).

Support Vector Machines

- Clearly, there is no linear separator for Fig. 18.31(a). Now, suppose we map each input vector **x** to a new vector of feature values, $F(\mathbf{x})$. In particular, let us use the three features

$$f_1 = x_1^2 , \qquad f_2 = x_2^2 , \qquad f_3 = \sqrt{2} x_1 x_2$$

- Figure 18.31(b) shows the data in the new, three-dimensional space defined by the three features; the data are linearly separable in this space! If data are mapped into a space of sufficiently high dimension, then they will almost always be linearly separable. In general (with some special cases excepted) if we have N data points then they will always be separable in spaces of $N - 1$ dimensions or more.

Support Vector Machines

- Now, we would not usually expect to find a linear separator in the input space $\mathbf{x}$, but we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_j \cdot \mathbf{x}_k$ in Equation(18.13) with $F(\mathbf{x}_j) \cdot F(\mathbf{x}_k)$.
- It turns out that $F(\mathbf{x}_j) \cdot F(\mathbf{x}_k)$ can often be computed without first computing F for each point. In our three-dimensional feature space defined by Equation (18.15), a little bit of algebra shows that

$$F(\mathbf{x}_j) \cdot F(\mathbf{x}_k) = (\mathbf{x}_j \cdot \mathbf{x}_k)^2$$

(That's why the $\sqrt{2}$ is in f_3.)

$$\boldsymbol{x}_j = (x_{1,j}, x_{2,j}) \quad F(\boldsymbol{x}_j) = (x_{1,j}^2, x_{2,j}^2, \sqrt{2}x_{1,j}x_{2,j})$$

$$\boldsymbol{x}_k = (x_{1,k}, x_{2,k}) \quad F(\boldsymbol{x}_k) = (x_{1,k}^2, x_{2,k}^2, \sqrt{2}x_{1,k}x_{2,k})$$

$$(\boldsymbol{x}_j \cdot \boldsymbol{x}_k)^2 = x_{1,j}^2 x_{1,k}^2 + x_{2,j}^2 x_{2,k}^2 + 2x_{1,j}x_{1,k}x_{2,j}x_{2,k}$$

Support Vector Machines

- The expression $(\mathbf{x}_j \cdot \mathbf{x}_k)^2$ is called a **kernel function**, and is usually written as $K(\mathbf{x}_j, \mathbf{x}_k)$. The kernel function can be applied to pairs of input data to evaluate dot products in some corresponding feature space. So, we can find linear separators in the higher-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_j \cdot \mathbf{x}_k$ in Equation (18.13) with a kernel function $K(\mathbf{x}_j, \mathbf{x}_k)$. Thus, we can learn in the higher-dimensional space, but we compute only kernel functions rather than the full list of features for each data point.
- Other kernel functions: $K(\mathbf{x}_j, \mathbf{x}_k) = (1 + \mathbf{x}_j \cdot \mathbf{x}_k)^d$

Support Vector Machines

- **Kernel trick**: Plugging these kernels into Equation (18.13), optimal linear separators can be found efficiently in feature spaces with billions of (or, in some cases, infinitely many) dimensions. The resulting linear separators, when mapped back to the original input space, can correspond to arbitrarily wiggly, nonlinear decision boundaries between the positive and negative examples.

Ensemble Learning

- The idea of ensemble learning methods is to select a collection, or ensemble, of hypotheses from the hypothesis space and combine their predictions.
- For example, during cross-validation we might generate twenty different decision trees, and have them vote on the best classification for a new example.
- Consider an ensemble of $K = 5$ hypotheses and suppose that we combine their predictions using simple majority voting. For the ensemble to misclassify a new example, at least three of the five hypotheses have to misclassify it. The hope is that this is much less likely than a misclassification by a single hypothesis.

Ensemble Learning

- Another way to think about the ensemble idea is as a generic way of enlarging the hypothesis space. That is, think of the ensemble itself as a hypothesis and the new hypothesis space as the set of all possible ensembles constructable from hypotheses in the original space.

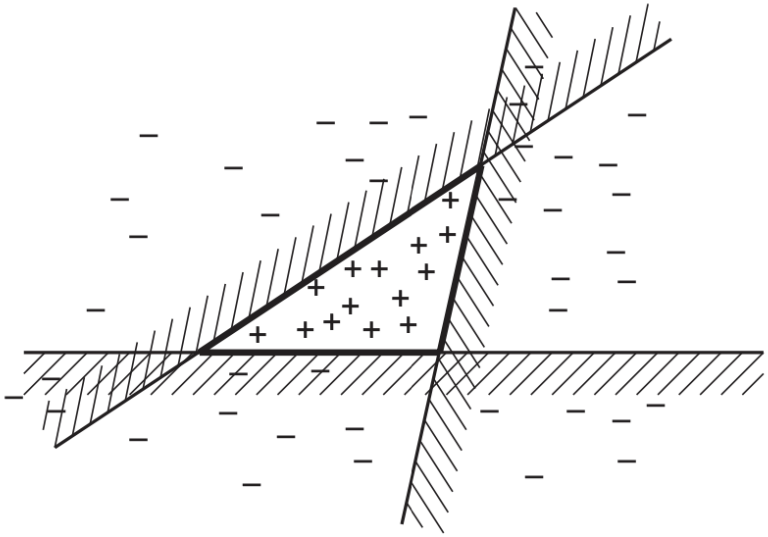


Figure 18.32 Illustration of the increased expressive power obtained by ensemble learning. We take three linear threshold hypotheses, each of which classifies positively on the unshaded side, and classify as positive any example classified positively by all three. The resulting triangular region is a hypothesis not expressible in the original hypothesis space.

Ensemble Learning -- Bagging

- We generate K distinct training sets by sampling with replacement from the original training set. We then run our machine learning algorithm on a training set to get hypothesis. Repeat this process K times, getting K different hypotheses. For classification, taking the majority vote. For regression, the final output is the average $h(\boldsymbol{x}) = \frac{1}{K}\sum_{i=1}^{K} h_i(\boldsymbol{x})$

- Bagging tends to reduce variance and is a standard approach when there is limited data or when the base model is seen to be overfitting.

Ensemble Learning – Random forests

- The **random forest** model is a form of decision tree bagging in which we take extra steps to make the ensemble of K trees more diverse.
- The key idea is to randomly vary the *attribute choices* (rather than the training examples). At each split point in constructing the tree, we select a random sampling of attributes, and then compute which of those gives the highest information gain.
- Further improvement: for each selected attribute, we randomly sample several candidate values from a uniform distribution. Then we select the value that has the highest information gain.

Ensemble Learning – Random forests

- All the hyperparameters can be trained by cross-validation: the number of trees *K*, the number of examples used by each tree *N*, the number of attributes used at each split point, and the number of random split points tried.
- Random forests are resistant to overfitting.
- Breiman (2001) gives a mathematical proof that (in almost all cases) as you add more trees to the forest, the error converges.

Ensemble Learning – Stacking

- The technique of **stacked generalization** (or **stacking** for short) combines multiple base models from different model classes trained on the same data.
- For example, given the restaurant data set, the first row is:

  $\mathbf{x}_1$=Yes, No, No, Yes, Some, $$$, No, Yes, French, 0-10; y_1=Yes

- We use the training set to train three separate base models – SVM, logistic regression, and a decision tree.
- In the next step we take the validation data set and augment each row with the predictions made from the three base models, giving us rows look like this

  $\mathbf{x}_2$=Yes, No, No, Yes, Full, $, No, No, Thai, 30-60, **Yes**, **No**, **No**; y_2=No

Ensemble Learning – Stacking

- We use this validation set to train a new ensemble model, let's say a logistic regression model. The ensemble model can use the predictions and the original data as it sees fit.
- This method is called "stacking" because it can be thought of as a layer of base models with an ensemble model stacked above it, operating on the output of the base models.
- Stacking reduces bias, and usually leads to performance that is better than any of the individual base models.

Ensemble Learning -- Boosting

- The most widely used ensemble method is called **boosting**.
- To understand the idea, we need first to explain the idea of a **weighted training set**. In such a training set, each example has an associated weight $w_j \geq 0$. The higher the weight of an example, the higher is the importance attached to it during the learning of a hypothesis.

Ensemble Learning -- Boosting

- Boosting starts with $w_j = 1$ for all the examples (i.e., a normal training set). From this set, it generates the first hypothesis, h_1. This hypothesis will classify some of the training examples correctly and some incorrectly. We would like the next hypothesis to *do better on the misclassified examples*, so we increase their weights while decreasing the weights of the correctly classified examples.
- From this new weighted training set, we generate hypothesis h_2. The process continues in this way until we have generated K hypotheses. The final ensemble hypothesis is a weighted-majority combination of all the K hypotheses, each weighted according to how well it performed on the training set.

Ensemble Learning -- Boosting

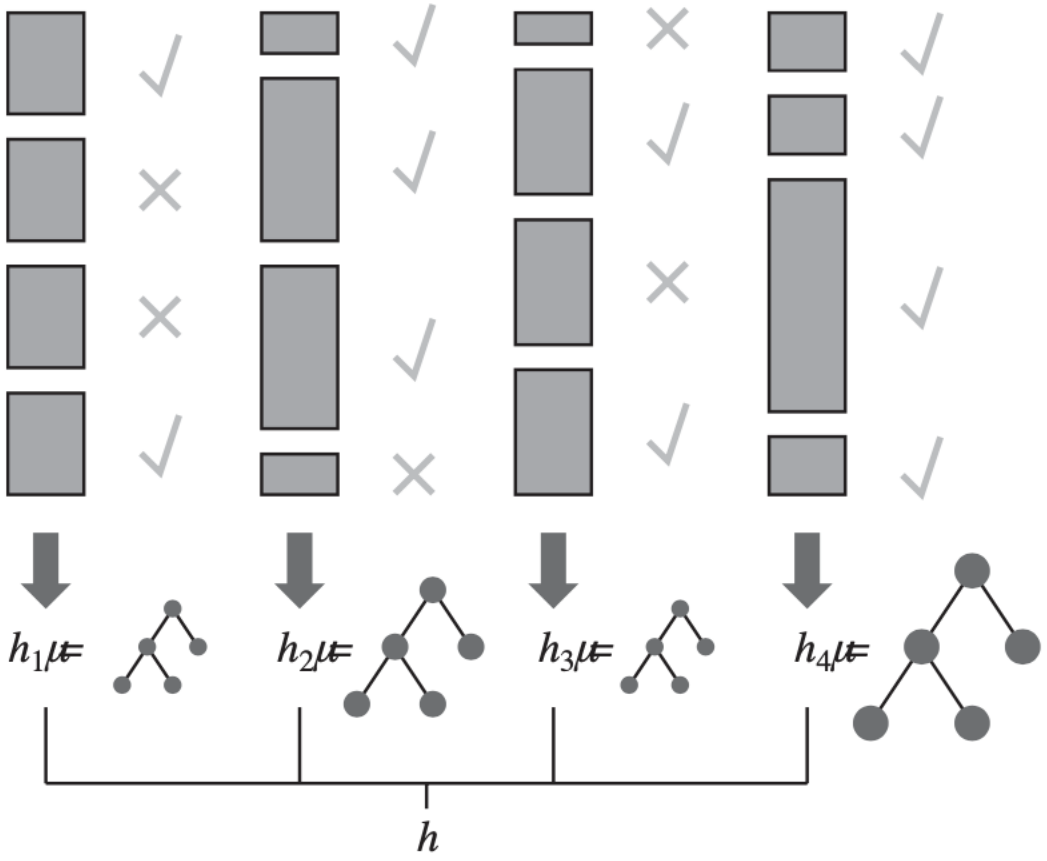


Figure 18.33 How the boosting algorithm works. Each shaded rectangle corresponds to an example; the height of the rectangle corresponds to the weight. The checks and crosses indicate whether the example was classified correctly by the current hypothesis. The size of the decision tree indicates the weight of that hypothesis in the final ensemble.

Ensemble Learning -- Boosting

- ADABOOST has a very important property: if the input learning algorithm L is a **weak learning algorithm**—which means that L always returns a hypothesis with accuracy on the training set that is slightly better than random guessing —then ADABOOST will return a hypothesis that classifies the training data perfectly for large enough K.

Ensemble Learning -- Boosting

- Let us see how well boosting does on the restaurant data. We will choose as our original hypothesis space the class of decision stumps, which are decision trees with just one test, at the root.
- The lower curve in Figure 18.35(a) shows that unboosted decision stumps are not very effective for this data set, reaching a prediction performance of only 81% on 100 training examples. When boosting is applied (with $K = 5$), the performance is better, reaching 93% after 100 examples.

Ensemble Learning -- Boosting

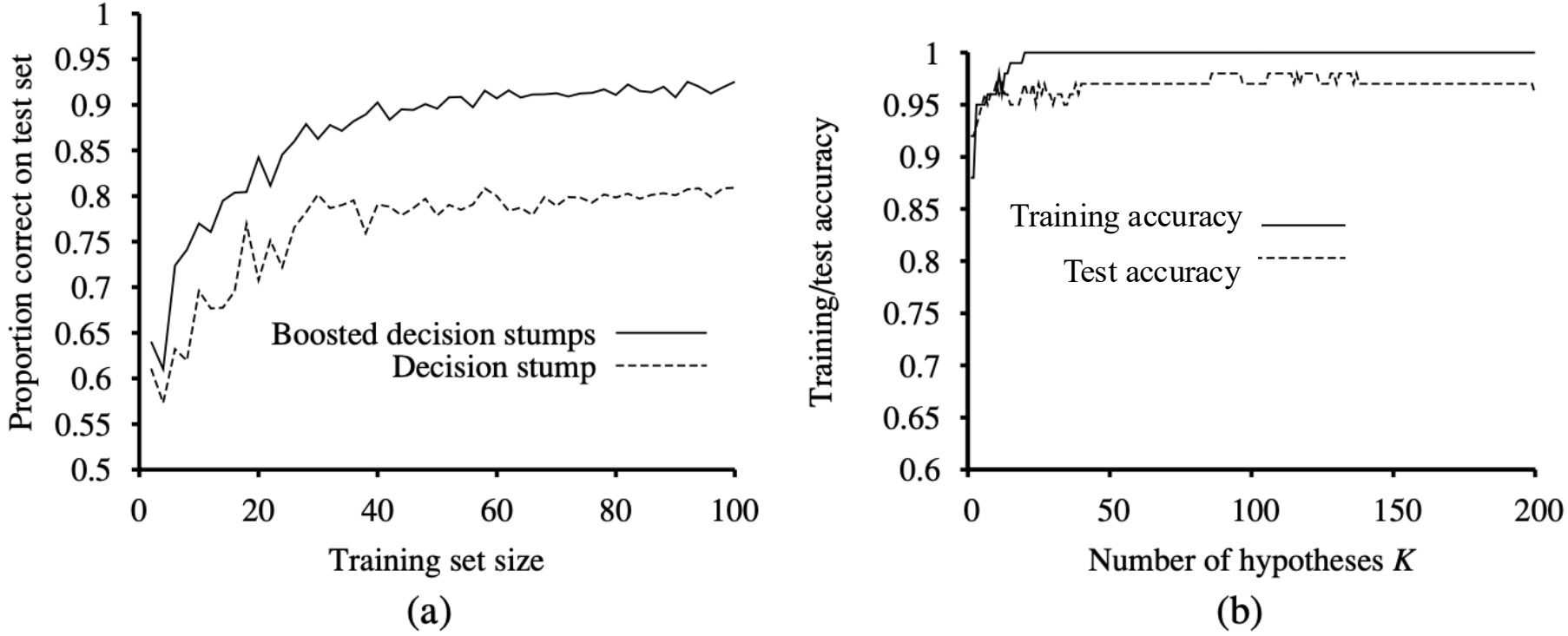


Figure 18.35 (a) Graph showing the performance of boosted decision stumps with $K = 5$ versus unboosted decision stumps on the restaurant data. (b) The proportion correct on the training set and the test set as a function of K, the number of hypotheses in the ensemble. Notice that the test set accuracy improves slightly even after the training accuracy reaches 1, i.e., after the ensemble fits the data exactly.

Ensemble Learning – Online learning

- So far, everything we have done in this chapter has relied on the assumption that the data are i.i.d. (independent and identically distributed). On the one hand, that is a sensible assumption: if the future bears no resemblance to the past, then how can we predict anything? On the other hand, it is too strong an assumption: it is rare that our inputs have captured all the information that would make the future truly independent of the past.

Ensemble Learning – Online learning

- What to do when the data are not i.i.d.?
- In this case, it matters *when* we make a prediction, so we will adopt the perspective called **online learning**: an agent receives an input x_j from nature, predicts the corresponding y_j, and then is told the correct answer. Then the process repeats with x_{j+1}, and so on.

Ensemble Learning – Online learning

- Let us consider the situation where our input consists of predictions from a panel of experts. For example, each day a set of *K* pundits (權威者) predicts whether the stock market will go up or down, and our task is to pool those predictions and make our own. One way to do this is to keep track of how well each expert performs, and choose to believe them in proportion to their past performance. This is called the **randomized weighted majority algorithm**.

Ensemble Learning – Online learning

- We can describe it more formally:

1. Initialize a set of weights $\{w_1, \ldots, w_K\}$ all to 1.
2. Receive the predictions $\{\hat{y}_1, \ldots, \hat{y}_K\}$ from the experts.
3. Randomly choose an expert k^*, in proportion to its weight: $P(k) = w_k / (\sum_{k'} w_{k'})$.
4. Predict $\hat{y}_{k^*}$.
5. Receive the correct answer y.
6. For each expert k such that $\hat{y}_k \neq y$, update $w_k \leftarrow \beta w_k$

Here β is a number, $0 < \beta < 1$, that tells how much to penalize an expert for each mistake.

Ensemble Learning – Online learning

- Online learning is helpful when the data may be changing rapidly over time. It is also useful for applications that involve a large collection of data that is constantly growing, even if changes are gradual.
- For most learning algorithms based on minimizing loss, there is an online version based on minimizing regret.

Developing Machine Learning Systems

- Problem formulation
  - You need to specify a loss function, which should be correlated with your true goals.
  - When you have decomposed your problems into parts, you may find that there are multiple components that can be handled by old-fashioned software engineering, not machine learning.
  - Part of problem formulation is deciding whether you are dealing with supervised, unsupervised, or reinforcement learning.

Developing Machine Learning Systems

- Data collection, assessment, and management
  - Every machine learning project needs data.
  - Freely available datasets, crowdsourcing
  - When data are limited, **data augmentation** can help.
  - Unbalanced classes problem – undersample or over-sample. You can use a weighted loss function that gives a larger penalty to missing a fraudulent case.
  - You should carefully consider **outliers** in your data. An outlier is a data point that is far from other points.

Developing Machine Learning Systems

- Feature engineering
  - You may preprocess the data to make it easier to digest.
  - Quantization, normalization, one-hot encoding
  - You can also introduce new attributes based on your domain knowledge
- Exploratory data analysis and visualization
  - Cluster your data and then visualize a prototype data point at the center of each cluster.
  - It is also helpful to detect outliers that are far from the prototypes

Developing Machine Learning Systems

- Exploratory data analysis and visualization
  - t-distributed stochastic neighbor embedding (t-SNE)

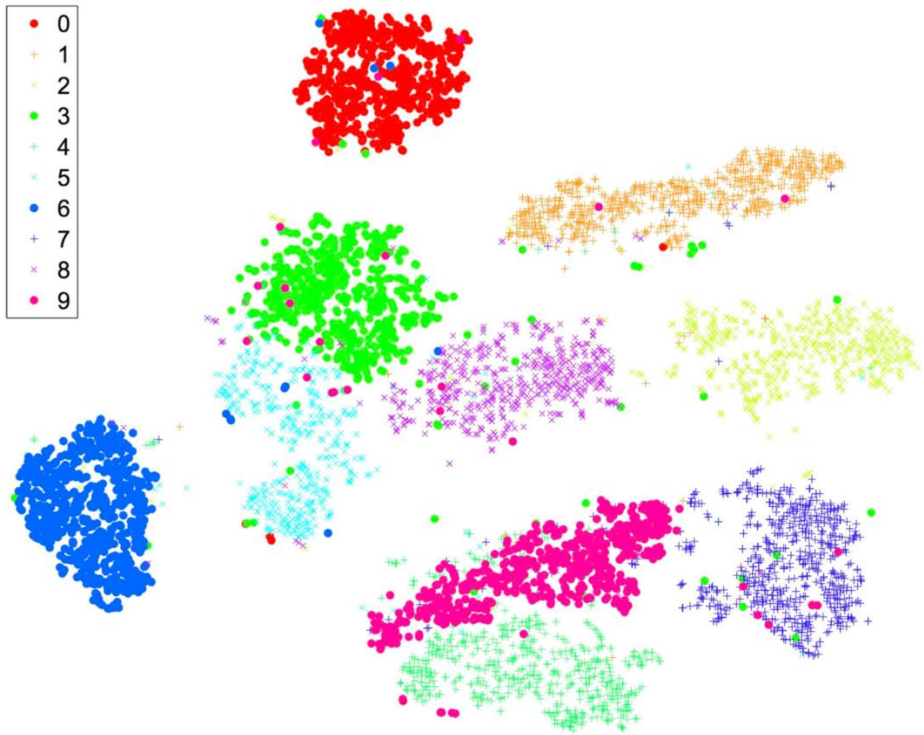


Figure 19.27 A two-dimensional t-SNE map of the MNIST data set, a collection of 60,000 images of handwritten digits, each 28 × 28 pixels and thus 784 dimensions. You can clearly see clusters for the ten digits, with a few confusions in each cluster; for example the top cluster is for the digit 0, but within the bounds of the cluster are a few data points representing the digits 3 and 6. The t-SNE algorithm finds a representation that accentuates the differences between clusters.

Developing Machine Learning Systems

- Model selection and training
  - There is no guaranteed way to pick the best model class, but there are some rough guidelines.
  - Random forests are good when there are a lot of categorical features and you believe that many of them may be irrelevant.
  - Nonparameteric methods are good when you have a lot of data and no prior knowledge.
  - Logistic regression does well when the data are linearly separable.

Developing Machine Learning Systems

- Model selection and training
  - SVMs are good to try when the data set is not too large.
  - Problems dealing with pattern recognition are most often approached with deep neural networks.

  - Choosing hyperparameters can be done with a combination of experience and search. As you run more experiments you will get ideas for different models to try.

Developing Machine Learning Systems

- Trust, interpretability, and explainability
  - Doing well on metric is a necessary but not sufficient condition for your to **trust** your model.
  - **Interpretability**: We say a machine learning model is interpretable if you can inspect the actual model and understand why it got a particular answer for a given input.
  - **Explainability**: An explainable model is one that can help you understand "*why* was this output produced for this input?"