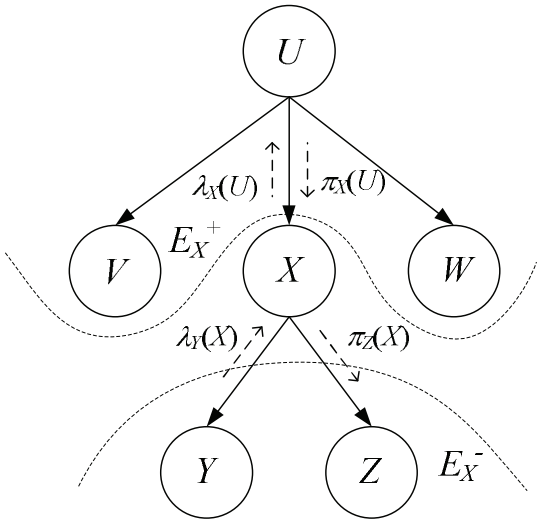


Figure 14.10 In a tree, a node may have several children but a single parent.

In the general case, if X has m children, $Y_j, j = 1, \ldots, m$, then we multiply all their λ values:

$$\lambda(X) = \prod_{j=1}^{m} \lambda_{Y_j}(X) \tag{14.19}$$

Once X accumulates λ evidence from its children's λ-messages, it propagates it up to its parent:

$$\lambda_X(U) = \sum_X \lambda(X) P(X|U) \tag{14.20}$$

Similarly and in the other direction, $\pi(X)$ is the evidence elsewhere that is accumulated in $P(U|E^+)$ and passed on to X as a π-message:

$$\pi(X) \equiv P(X|E_X^+) = \sum_U P(X|U) P(U|E_X^+) = \sum_U P(X|U) \pi_X(U) \tag{14.21}$$

This calculated π value is then propagated down to X's children. Note that what Y receives from X is what X receives from its parent U and also from its other child Z; together they make up E_Y^+ (see figure 14.10):

$$\pi_Y(X) \quad \equiv \quad P(X|E_Y^+) = P(X|E_X^+, E_Z^-)$$

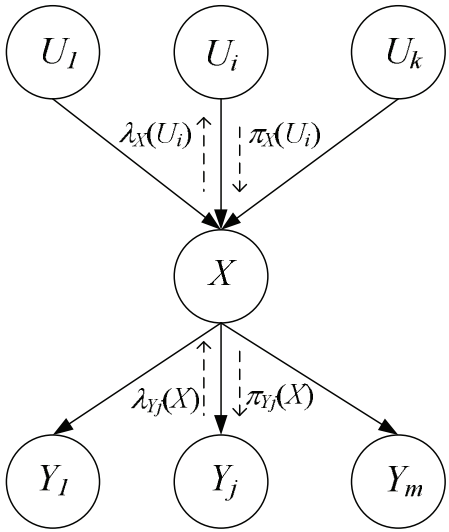


Figure 14.11 In a polytree, a node may have several children and several parents, but the graph is singly connected; that is, there is a single chain between U_i and Y_j passing through X.

$$= \frac{P(E_Z^-|X,E_X^+)P(X|E_X^+)}{P(E_Z^-)} = \frac{P(E_Z^-|X)P(X|E_X^+)}{P(E_Z^-)}$$

$$= \alpha\lambda_Z(X)\pi(X) \tag{14.22}$$

Again, if Y has not one sibling Z but multiple, we need to take a product over all their λ values:

$$\pi_{Y_j}(X) = \alpha \prod_{s\neq j} \lambda_{Y_s}(X)\pi(X) \tag{14.23}$$

14.5.3 Polytrees

POLYTREE

In a tree, a node has a single parent, that is, a single cause. In a *polytree*, a node may have multiple parents, but we require that the graph be singly connected, which means that there is a single chain between any two nodes. If we remove X, the graph will split into two components. This is necessary so that we can continue splitting E_X into E_X^+ and E_X^-, which are independent given X (see figure 14.11).

If X has multiple parents $U_i, i = 1,\ldots,k$, it receives π-messages from

all of them, $\pi_X(U_i)$, which it combines as follows:

$$\begin{aligned}
\pi(X) &\equiv P(X|E_X^+) = P(X, E_{U_1X}^+, E_{U_2X}^+, \ldots, E_{U_kX}^+) \\
&= \sum_{U_1}\sum_{U_2}\cdots\sum_{U_k} P(X|U_1, U_2, \ldots, U_k) P(U_1|E_{U_1X}^+)\cdots P(U_k|E_{U_kX}^+) \\
&= \sum_{U_1}\sum_{U_2}\cdots\sum_{U_k} P(X|U_1, U_2, \ldots, U_k) \prod_{i=1}^{k} \pi_X(U_i) \qquad (14.24)
\end{aligned}$$

and passes it on to its several children $Y_j, j = 1, \ldots, m$:

$$\pi_{Y_j}(X) = \alpha \prod_{s\neq j} \lambda_{Y_s}(X)\pi(X) \qquad (14.25)$$

In this case when X has multiple parents, a λ-message X passes on to one of its parents U_i combines not only the evidence X receives from its children but also the π-messages X receives from its other parents $U_r, r \neq i$; they together make up $E_{U_iX}^-$:

$$\begin{aligned}
\lambda_X(U_i) &\equiv P(E_{U_iX}^-|X) \\
&= \sum_X \sum_{U_{r\neq i}} P(E_X^-, E_{U_{r\neq i}X}^+, X, U_{r\neq i}|U_i) \\
&= \sum_X \sum_{U_{r\neq i}} P(E_X^-, E_{U_{r\neq i}X}^+|X, U_{r\neq i}, U_i) P(X, U_{r\neq i}|U_i) \\
&= \sum_X \sum_{U_{r\neq i}} P(E_X^-|X) P(E_{U_{r\neq i}X}^+|U_{r\neq i}) P(X|U_{r\neq i}, U_i) P(U_{r\neq i}|U_i) \\
&= \sum_X \sum_{U_{r\neq i}} P(E_X^-|X) \frac{P(U_{r\neq i}|E_{U_{r\neq i}X}^+) P(E_{U_{r\neq i}X}^+)}{P(U_{r\neq i})} P(X|U_{r\neq i}, U_i) P(U_{r\neq i}|U_i) \\
&= \beta \sum_X \sum_{U_{r\neq i}} P(E_X^-|X) P(U_{r\neq i}|E_{U_{r\neq i}X}^+) P(X|U_{r\neq i}, U_i) \\
&= \beta \sum_X \sum_{U_{r\neq i}} \lambda(X) \prod_{r\neq i} \pi_X(U_r) P(X|U_1, \ldots, U_k) \\
&= \beta \sum_X \lambda(X) \sum_{U_{r\neq i}} P(X|U_1, \ldots, U_k) \prod_{r\neq i} \pi_X(U_r) \qquad (14.26)
\end{aligned}$$

As in a tree, to find its overall λ, the parent multiplies the λ-messages it receives from its children:

$$\lambda(X) = \prod_{j=1}^{m} \lambda_{Y_j}(X) \qquad (14.27)$$

In this case of multiple parents, we need to store and manipulate the conditional probability given all the parents, $p(X|U_1,\ldots,U_k)$, which is costly for large k. Approaches have been proposed to decrease the complexity from exponential in k to linear. For example, in a *noisy OR gate*, any of the parents is sufficient to cause the event and the likelihood does not decrease when multiple parent events occur. If the probability that X happens when only cause U_i happens is $1-q_i$

NOISY OR

$$P(X|U_i,\sim U_{p\neq j}) = 1-q_i \tag{14.28}$$

the probability that X happens when a subset T of them occur is calculated as

$$P(X|T) = 1-\prod_{u_i\in T} q_i \tag{14.29}$$

For example, let us say wet grass has two causes, rain and a sprinkler, with $q_R = q_S = 0.1$; that is, both singly have a 90 percent probability of causing wet grass. Then, $P(W|R,\sim S) = 0.9$ and $P(W|R,S) = 0.99$.

Another possibility is to write the conditional probability as some function given a set of parameters, for example, as a linear model

$$P(X|U_1,\ldots,U_k,w_0,w_1,\ldots,w_k) = \text{sigmoid}\left(\sum_{i=1}^{k} w_i U_i + w_0\right) \tag{14.30}$$

where sigmoid guarantees that the output is a probability between 0 and 1. During training, we can learn the parameters $w_i, i=0,\ldots,d$, for example, to maximize the likelihood on a sample.

14.5.4 Junction Trees

If there is a loop, that is, if there is a cycle in the underlying undirected graph—for example, if the parents of X share a common ancestor—the algorithm we discussed earlier does not work. In such a case, there is more than one path on which to propagate evidence and, for example, while evaluating the probability at X, we cannot say that X separates E into E_X^+ and E_X^- as causal (upward) and diagnostic (downward) evidence; removing X does not split the graph into two. Conditioning them on X does not make them independent and the two can interact through some other path not involving X.

We can still use the same algorithm if we can convert the graph to a polytree. We define *clique nodes* that correspond to a set of original variables and connect them so that they form a tree (see figure 14.12). We

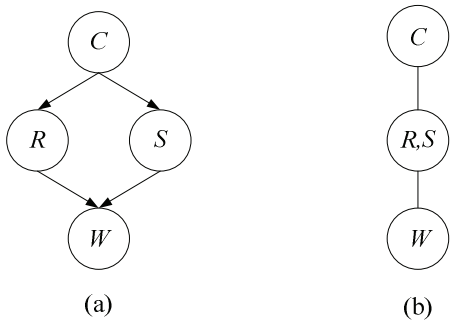


Figure 14.12 (a) A multiply connected graph, and (b) its corresponding junction tree with nodes clustered.

JUNCTION TREE

can then run the same belief propagation algorithm with some modifications. This is the basic idea behind the *junction tree algorithm* (Lauritzen and Spiegelhalter 1988; Jensen 1996; Jordan 2004).

14.6 Undirected Graphs: Markov Random Fields

MARKOV RANDOM FIELD

Up to now, we have discussed directed graphs where the influences are undirectional and have used Bayes' rule to invert the arcs. If the influences are symmetric, we represent them using an undirected graphical model, also known as a *Markov random field*. For example, neighboring pixels in an image tend to have the same color—that is, are correlated—and this correlation goes both ways.

Directed and undirected graphs define conditional independence differently, and, hence, there are probability distributions that are represented by a directed graph and not by an undirected graph, and vice versa (Pearl 1988).

Because there are no directions and hence no distinction between the head or the tail of an arc, the treatment of undirected graphs is simpler. For example, it is much easier to check if A and B are independent given C. We just check if after removing all nodes in C, we still have a path between a node in A and a node in B. If so, they are dependent, otherwise, if all paths between nodes in A and nodes in B pass through nodes in C such that removal of C leaves nodes of A and nodes of B in separate components, we have independence.

CLIQUE

In the case of an undirected graph, we do not talk about the parent or the child but about *cliques*, which are sets of nodes such that there exists a link between any two nodes in the set. A *maximal* clique has the maximum number of elements. Instead of conditional probabilities (implying a direction), in undirected graphs we have *potential functions* $\psi_C(X_C)$ where X_C is the set of variables in clique C, and we define the joint distribution as the product of the potential functions of the maximal cliques of the graph

POTENTIAL FUNCTION

$$p(X) = \frac{1}{Z}\prod_C \psi_C(X_C) \tag{14.31}$$

where Z is the normalization constant to make sure that $\sum_X p(X) = 1$:

$$Z = \sum_X \prod_C \psi_C(X) \tag{14.32}$$

It can be shown that a directed graph is already normalized (exercise 5).

Unlike in directed graphs, the potential functions in an undirected graph do not need to have a probabilistic interpretation, and one has more freedom in defining them. In general, we can view potential functions as expressing local constraints, that is, favoring some local configurations over others. For example, in an image, we can define a pairwise potential function between neighboring pixels, which takes a higher value if their colors are similar than the case when they are different (Bishop 2006). Then, setting some of the pixels to their values given as evidence, we can estimate the values of other pixels that are not known, for example, due to occlusion.

If we have the directed graph, it is easy to redraw it as an undirected graph, simply by dropping all the directions, and if a node has a single parent, we can set the pairwise potential function simply to the conditional probability. If the node has more than one parent, however, the "explaining away" phenomenon due to the head-to-head node makes the parents dependent, and hence we should have the parents in the same clique so that the clique potential includes all the parents. This is done by connecting all the parents of a node by links so that they are completely connected among them and form a clique. This is called "marrying" the parents, and the process is called *moralization*. Incidentally, moralization is one of the steps in generating a junction tree, which is undirected.

MORALIZATION

It is straightforward to adapt the belief propagation algorithm to work on undirected graphs, and it is easier because the potential function is

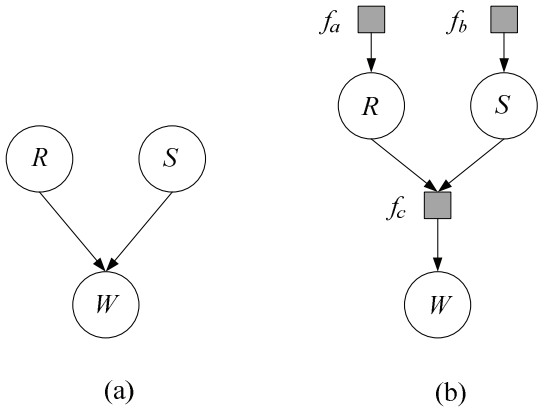


Figure 14.13 (a) A directed graph that would have a loop after moralization, and (b) its corresponding factor graph that is a tree. The three factors are $f_a(R) \equiv P(R)$, $f_b(S) \equiv P(S)$, and $f_c(R,S,W) \equiv P(W|R,S)$.

symmetric and we do not need to make a difference between causal and diagnostic evidence. Thus, we can do inference on undirected chains and trees. But in polytrees where a node has multiple parents and moralization necessarily creates loops, this would not work. One trick is to convert it to a *factor graph* that uses a second kind of *factor nodes* in addition to the variable nodes, and we write the joint distribution as a product of factors (Kschischang, Frey, and Loeliger 2001)

FACTOR GRAPH

$$p(X) = \frac{1}{Z}\prod_S f_S(X_S) \tag{14.33}$$

where X_s denotes a subset of the variable nodes used by factor S. Directed graphs are a special case where factors correspond to local conditional distributions, and undirected graphs are another special case where factors are potential functions over maximal cliques. The advantage is that, as we can see in figure 14.13, the tree structure can be kept even after moralization.

It is possible to generalize the belief propagation algorithm to work on factor graphs; this is called the *sum-product algorithm* (Bishop 2006; Jordan 2004) where there is the same idea of doing local computations once and propagating them through the graph as messages. The difference now is that there are two types of messages because there are two kinds of nodes, factors and variables, and we make a distinction between their

SUM-PRODUCT ALGORITHM

messages. Note, however, that a factor graph is bipartite, and one kind of node can have a close encounter only with the second kind.

In belief propagation, or the sum-product algorithm, the aim is to find the probability of a set of nodes X given that another set of evidence nodes E are clamped to a certain value, that is, $P(X|E)$. In some applications, we may be interested in finding the setting of all X that maximizes the full joint probability distribution $p(X)$. For example, in the undirected case where potential functions code locally consistent configurations, such an approach would propagate local constraints over the whole graph and find a solution that maximizes global consistency. In a graph where nodes correspond to pixels and pairwise potential functions favor correlation, this approach would implement noise removal (Bishop 2006). The algorithm for this, named the *max-product algorithm* (Bishop 2006; Jordan 2004) is the same as the sum-product algorithm where we take the maximum (choose the most likely value) rather than the sum (marginalize). This is analogous to the difference between the forward-backward procedure and the Viterbi algorithm in hidden Markov models that we discussed in chapter 15.

MAX-PRODUCT ALGORITHM

Note that the nodes need not correspond to low-level concepts like pixels; in a vision application, for instance, we may have nodes for corners of different types or lines of different orientations with potential functions checking for compatibility, so as to see if they can be part of the same interpretation—remember the Necker cube, for example—so that overall consistent solutions emerge after the consolidation of local evidences.

The complexity of the inference algorithms on polytrees or junction trees is determined by the maximum number of parents or the size of the largest clique, and when this is large, exact inference may be infeasible. In such a case, one needs to use an approximation or a sampling algorithm (Jordan 1999; Bishop 2006).

14.7 Learning the Structure of a Graphical Model

As in any approach, learning a graphical model has two parts. The first is the learning of parameters given a structure; this is relatively easier (Buntine 1996), and, in graphical models, conditional probability tables or their parameterizations (as in equation 14.30) can be trained to maximize the likelihood, or by using a Bayesian approach if suitable priors are known (chapter 16).

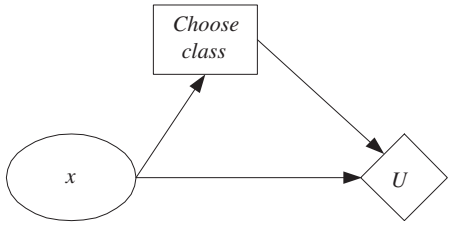


Figure 14.14 Influence diagram corresponding to classification. Depending on input x, a class is chosen that incurs a certain utility (risk).

The second, more difficult, and interesting part is to learn the graph structure (Cowell et al. 1999). This is basically a model selection problem, and just like the incremental approaches for learning the structure of a multilayer perceptron (section 11.9), we can see this as a search in the space of all possible graphs. One can, for example, consider operators that can add/remove arcs and/or hidden nodes and then do a search evaluating the improvement at each step (using parameter learning at each intermediate iteration). Note, however, that to check for overfitting, one should regularize properly, corresponding to a Bayesian approach with a prior that favors simpler graphs (Neapolitan 2004). However, because the state space is large, it is most helpful if there is a human expert who can manually define causal relationships among variables and creates subgraphs of small groups of variables.

In chapter 16, we discuss the Bayesian approach and in section 16.8, we discuss the nonparametric Bayesian methods where model structure can be made more complex in time as more data arrives.

14.8 Influence Diagrams

INFLUENCE DIAGRAMS

Just as in chapter 3, we generalized from probabilities to actions with risks, *influence diagrams* are graphical models that allow the generalization of graphical models to include decisions and utilities. An influence diagram contains *chance nodes* representing random variables that we use in graphical models (see figure 14.14). It also has decision nodes and a utility node. A *decision node* represents a choice of actions. A *utility node* is where the utility is calculated. Decisions may be based on chance nodes and may affect other chance nodes and the utility node.

Inference on an influence diagram is an extension to belief propagation on a graphical model. Given evidence on some of the chance nodes, this evidence is propagated, and for each possible decision, the utility is calculated and the decision having the highest utility is chosen. The influence diagram for classification of a given input is shown in figure 14.14. Given the input, the decision node decides on a class, and for each choice we incur a certain utility (risk).

14.9 Notes

Graphical models have two advantages. One is that we can visualize the interaction of variables and have a better understanding of the process, for example, by using a causal generative model. The second is that by finding graph operations that correspond to basic probabilistic procedures such as Bayes' rule or marginalization, the task of inference can be mapped to general-purpose graph algorithms that can be efficiently represented and implemented.

The idea of visual representation of variables and dependencies between them as a graph, and the related factorization of a complicated global function of many variables as a product of local functions involving a small subset of the variables for each, seems to be used in different domains in decision making, coding, and signal processing; Kschischang, Frey, and Loeliger (2001) give a review.

The complexity of the inference algorithms on polytrees or junction trees is determined by the maximum number of parents or the size of the largest clique, and when this is large exact inference may be infeasible. In such a case, one needs to use an approximation or a sampling algorithm. Variational approximations and Markov chain Monte Carlo (MCMC) algorithms are discussed in Jordan et al. 1999, MacKay 2003, Andrieu et al. 2003, Bishop 2006, and Murphy 2012.

Graphical models are especially suited to represent Bayesian approaches where in addition to nodes for observed variables, we also have nodes for hidden variables as well as the model parameters. We may also introduce a hierarchy where we have nodes for hyperparameters—that is, second-level parameters for the priors of the first-level parameters.

Thinking of data as sampled from a causal generative model that can be visualized as a graph can ease understanding and also inference in many domains. For example, in text categorization, generating a text may be

thought of as the process whereby an author decides to write a document on a number of topics and then chooses a set of words for each topic. In bioinformatics, one area among many where a graphical approach used is the modeling of a *phylogenetic tree*; namely, it is a directed graph whose leaves are the current species, whose nonterminal nodes are past ancestors that split into multiple species during a speciation event, and whose conditional probabilities depend on the evolutionary distance between a species and its ancestor (Jordan 2004).

PHYLOGENETIC TREE

The hidden Markov model we discuss in chapter 15 is one type of graphical model where inputs are dependent sequentially, as in speech recognition, where a word is a particular sequence of basic speech sounds called phonemes (Ghahramani 2001). Such *dynamic graphical models* find applications in many areas where there is a temporal dimension, such as speech, music, and so on (Zweig 2003; Bilmes and Bartels 2005).

Graphical models are also used in computer vision—for example, in information retrieval (Barnard et al. 2003) and scene analysis (Sudderth et al. 2008). A review of the use of graphical models in bioinformatics (and related software) is given in Donkers and Tuyls 2008.

14.10 Exercises

1. With two independent inputs in a classification problem, that is, $p(x_1, x_2|C) = p(x_1|C)p(x_2|C)$, how can we calculate $p(x_1|x_2, C)$? Derive the formula for $p(x_j|C_i) \sim \mathcal{N}(\mu_{ij}, \sigma_{ij}^2)$.

2. For a head-to-head node, show that equation 14.10 implies $P(X, Y) = P(X) \cdot P(Y)$.

   SOLUTION: We know that $P(X, Y, Z) = P(Z|X, Y)P(X, Y)$, and if we also know that $P(X, Y, Z) = P(X)P(Y)P(Z|X, Y)$, we see that $P(X, Y) = P(X)P(Y)$.

3. In figure 14.4, calculate $P(R|W)$, $P(R|W, S)$, and $P(R|W, \sim S)$.

   SOLUTION:

$$
\begin{aligned}
P(R|W) &= \frac{P(R, W)}{P(W)} = \frac{\sum_S P(R, W, S)}{\sum_R \sum_S P(R, W, S)} \\
&= \frac{\sum_S P(R)P(S)P(W|R, S)}{\sum_R \sum_S P(R)P(S)P(W|R, S)} \\
P(R|W, S) &= \frac{P(R, W, S)}{P(W, S)} = \frac{P(R)P(S)P(W|R, S)}{\sum_R P(R)P(S)P(W|R, S)} \\
P(R|W, \sim S) &= \frac{P(R, W, \sim S)}{P(W, \sim S)} = \frac{P(R)P(\sim S)P(W|R, \sim S)}{\sum_R P(R)P(\sim S)P(W|R, \sim S)}
\end{aligned}
$$

4. In equation 14.30, X is binary. How do we need to modify it if X can take one of K discrete values?

   SOLUTION: Let us say there are $j = 1, \ldots, K$ states. Then, keeping the model linear, we need to parameterize each by a separate $\mathbf{w}_j$ and use softmax to map to probabilities:

$$P(X = j|U_1, \ldots, U_k, \{w_{ji}\}) = \frac{\exp \sum_{i=1}^{k} w_{ji} U_i + w_{j0}}{\sum_{l=1}^{K} \exp \sum_{i=1}^{k} w_{li} U_i + w_{l0}}$$

5. Show that in a directed graph where the joint distribution is written as equation 14.12, $\sum_{\mathbf{x}} p(\mathbf{x}) = 1$.

   SOLUTION: The terms cancel when we sum up over all possible values because these are probabilities. Let us, for example, take figure 14.3:

$$\begin{aligned}
P(X, Y, Z) &= P(X)P(Y|X)P(Z|X) \\
\sum_X \sum_Y \sum_Z P(X, Y, Z) &= \sum_X \sum_Y \sum_Z P(X)P(Y|X)P(Z|X) \\
&= \sum_X \sum_Y P(X)P(Y|X) \sum_Z P(Z|X) \\
&= \sum_X \sum_Y P(X)P(Y|X) \sum_Z \frac{P(Z, X)}{P(X)} \\
&= \sum_X \sum_Y P(X)P(Y|X) \frac{P(X)}{P(X)} \\
&= \sum_X \sum_Y P(X)P(Y|X) \\
&= \sum_X P(X) \sum_Y P(Y|X) = \sum_X P(X) = 1
\end{aligned}$$

6. Draw the Necker cube as a graphical model defining links to indicate mutually reinforcing or inhibiting relations between different corner interpretations.

   SOLUTION: We are going to have nodes corresponding to corners, and they take values depending on the interpretation; there will be positive, enforcing, excitatory connections between corners that are part of the same interpretation, and negative, inhibitory connections between corners that are part of different interpretations (see figure 14.15).

7. Write down the graphical model for linear logistic regression for two classes in the manner of figure 14.7.

8. Propose a suitable goodness measure that can be used in learning graph structure as a state-space search. What are suitable operators?

   SOLUTION: We need a *score function* that is the sum of two parts, one quantifying a goodness of fit, that is, how likely is the data given the model, and one quantifying the complexity of the graph, to alleviate overfitting. In measuring complexity, we must take into account the total number of nodes and

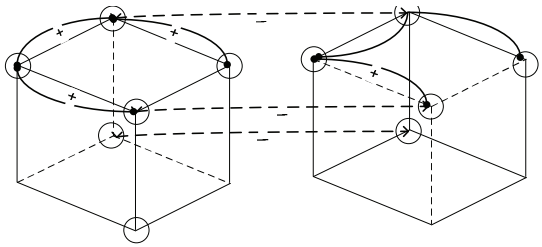


Figure 14.15 Two different interpretations of the Necker cube. Solid lines, marked by '+,' are excitatory and dashed lines, marked by '−,' are inhibitory.

the number of parameters needed to represent the conditional probability distributions. For example, we should try to have nodes with as few parents as possible. Possible operators are there to add/remove an edge and add/remove a hidden node.

9. Generally, in a newspaper, a reporter writes a series of articles on successive days related to the same topics as the story develops. How can we model this using a graphical model?

14.11 References

Andrieu, C., N. de Freitas, A. Doucet, and M. I. Jordan. 2003. "An Introduction to MCMC for Machine Learning." *Machine Learning* 50:5–43.

Barnard, K., P. Duygulu, D. Forsyth, N. de Freitas, D. M. Blei, and M. I. Jordan. 2003. "Matching Words and Pictures." *Journal of Machine Learning Research* 3:1107–1135.

Bilmes, J., and C. Bartels. 2005. "Graphical Model Architectures for Speech Recognition." *IEEE Signal Processing Magazine* 22:89–100.

Bishop, C. M. 2006. *Pattern Recognition and Machine Learning*. New York: Springer.

Buntine, W. 1996. "A Guide to the Literature on Learning Probabilistic Networks from Data." *IEEE Transactions on Knowledge and Data Engineering* 8:195–210.

Cowell, R. G., A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. 1999. *Probabilistic Networks and Expert Systems*. New York: Springer.

Donkers, J., and K. Tuyls. 2008. "Belief Networks in Bioinformatics." In *Computational Intelligence in Bioinformatics*, ed. A. Kelemen, A. Abraham, and Y. Chen, 75–111. Berlin: Springer.

Ghahramani, Z. 2001. "An Introduction to Hidden Markov Models and Bayesian Networks." *International Journal of Pattern Recognition and Artificial Intelligence* 15:9–42.

Jensen, F. 1996. *An Introduction to Bayesian Networks*. New York: Springer.

Jordan, M. I., ed. 1999. *Learning in Graphical Models*. Cambridge, MA: MIT Press.

Jordan, M. I. 2004. "Graphical Models." *Statistical Science* 19:140–155.

Jordan, M. I., Z. Ghahramani, T. S. Jaakkola, and L. K. Saul. 1999. "An Introduction to Variational Methods for Graphical Models." In *Learning in Graphical Models*, ed. M. I. Jordan, 105–161. Cambridge, MA: MIT Press.

Kschischang, F. R., B. J. Frey, and H.-A. Loeliger. 2001. "Factor Graphs and the Sum-Product Algorithm." *IEEE Transactions on Information Theory* 47:498–519.

Lauritzen, S. L., and D. J. Spiegelhalter. 1988. "Local Computations with Probabilities on Graphical Structures and their Application to Expert Systems." *Journal of Royal Statistical Society B* 50:157–224.

MacKay, D. J. C. 2003. *Information Theory, Inference, and Learning Algorithms*. Cambridge, UK: Cambridge University Press.

Murphy, K. P. 2012. *Machine Learning: A Probabilistic Perspective*. Cambridge, MA: MIT Press.

Neapolitan, R. E. 2004. *Learning Bayesian Networks*. Upper Saddle River, NJ: Pearson.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA: Morgan Kaufmann.

Pearl, J. 2000. *Causality: Models, Reasoning, and Inference*. Cambridge, UK: Cambridge University Press.

Sudderth, E. B., A. Torralba, W. T. Freeman, and A. S. Willsky. 2008. "Describing Visual Scenes Using Transformed Objects and Parts." *International Journal of Computer Vision* 77:291–330.

Zweig, G. 2003. "Bayesian Network Structures and Inference Techniques for Automatic Speech Recognition." *Computer Speech and Language* 17:173–193.

15 *Hidden Markov Models*

We relax the assumption that instances in a sample are independent and introduce Markov models to model input sequences as generated by a parametric random process. We discuss how this modeling is done as well as introduce an algorithm for learning the parameters of such a model from example sequences.

15.1 Introduction

UNTIL NOW, we assumed that the instances that constitute a sample are iid. This has the advantage that the likelihood of the sample is simply the product of the likelihoods of the individual instances. This assumption, however, is not valid in applications where successive instances are dependent. For example, in a word successive letters are dependent; in English 'h' is very likely to follow 't' but not 'x'. Such processes where there is a *sequence* of observations—for example, letters in a word, base pairs in a DNA sequence—cannot be modeled as simple probability distributions. A similar example is speech recognition where speech utterances are composed of speech primitives called phonemes; only certain sequences of phonemes are allowed, which are the words of the language. At a higher level, words can be written or spoken in certain sequences to form a sentence as defined by the syntactic and semantic rules of the language.

A sequence can be characterized as being generated by a *parametric random process*. In this chapter, we discuss how this modeling is done and also how the parameters of such a model can be learned from a training sample of example sequences.

15.2 Discrete Markov Processes

Consider a system that at any time is in one of a set of N distinct states: $S_1, S_2, \ldots, S_N$. The state at time t is denoted as $q_t, t = 1, 2, \ldots$, so, for example, $q_t = S_i$ means that at time t, the system is in state S_i. Though we write "time" as if this should be a temporal sequence, the methodology is valid for any sequencing, be it in time, space, position on the DNA string, and so forth.

At regularly spaced discrete times, the system moves to a state with a given probability, depending on the values of the previous states:

$$P(q_{t+1} = S_j | q_t = S_i, q_{t-1} = S_k, \cdots)$$

MARKOV MODEL

For the special case of a first-order *Markov model*, the state at time $t+1$ depends only on state at time t, regardless of the states in the previous times:

$$P(q_{t+1} = S_j | q_t = S_i, q_{t-1} = S_k, \cdots) = P(q_{t+1} = S_j | q_t = S_i) \tag{15.1}$$

This corresponds to saying that, given the present state, the future is independent of the past. This is just a mathematical version of the saying, Today is the first day of the rest of your life.

We further simplify the model—that is, regularize—by assuming that the *transition probability* from S_i to S_j is independent of time:

TRANSITION PROBABILITY

$$a_{ij} \equiv P(q_{t+1} = S_j | q_t = S_i) \tag{15.2}$$

satisfying

$$a_{ij} \geq 0 \text{ and } \sum_{j=1}^{N} a_{ij} = 1 \tag{15.3}$$

So, going from S_i to S_j has the same probability no matter when it happens, or where it happens in the observation sequence. $\mathbf{A} = [a_{ij}]$ is a $N \times N$ matrix whose rows sum to 1.

STOCHASTIC AUTOMATON

This can be seen as a *stochastic automaton* (see figure 15.1). From each state S_i, the system moves to state S_j with probability a_{ij}, and this probability is the same for any t. The only special case is the first state.

INITIAL PROBABILITY

We define the *initial probability*, π_i, which is the probability that the first state in the sequence is S_i:

$$\pi_i \equiv P(q_1 = S_i) \tag{15.4}$$

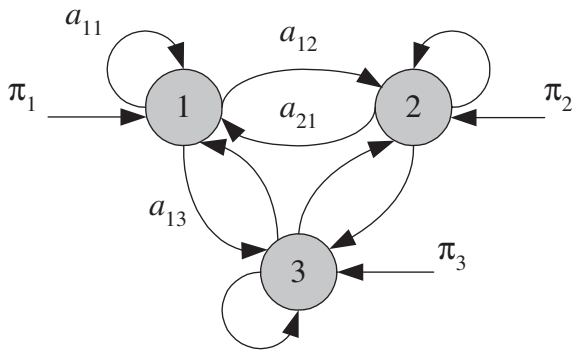


Figure 15.1 Example of a Markov model with three states. This is a stochastic automaton where π_i is the probability that the system starts in state S_i, and a_{ij} is the probability that the system moves from state S_i to state S_j.

satisfying

$$\sum_{i=1}^{N} \pi_i = 1 \tag{15.5}$$

$\mathbf{\Pi} = [\pi_i]$ is a vector of N elements that sum to 1.

OBSERVABLE MARKOV MODEL

In an *observable Markov model*, the states are observable. At any time t, we know q_t, and as the system moves from one state to another, we get an observation sequence that is a sequence of states. The output of the process is the set of states at each instant of time where each state corresponds to a physical observable event.

We have an observation sequence O that is the state sequence $O = Q = \{q_1 q_2 \cdots q_T\}$, whose probability is given as

$$P(O = Q|\mathbf{A}, \mathbf{\Pi}) = P(q_1) \prod_{t=2}^{T} P(q_t|q_{t-1}) = \pi_{q_1} a_{q_1 q_2} \cdots a_{q_{T-1} q_T} \tag{15.6}$$

π_{q_1} is the probability that the first state is q_1, $a_{q_1 q_2}$ is the probability of going from q_1 to q_2, and so on. We multiply these probabilities to get the probability of the whole sequence.

Let us now see an example (Rabiner and Juang 1986) to help us demonstrate. Assume we have N urns where each urn contains balls of only one color. So there is an urn of red balls, another of blue balls, and so forth.

Somebody draws balls from urns one by one and shows us their color. Let q_t denote the color of the ball drawn at time t. Let us say we have three states:

S_1 : red, S_2 = blue, S_3 : green

with initial probabilities:

$\mathbf{\Pi} = [0.5, 0.2, 0.3]^T$

a_{ij} is the probability of drawing from urn j (a ball of color j) after drawing a ball of color i from urn i. The transition matrix is, for example,

$$\mathbf{A} = \begin{bmatrix} 0.4 & 0.3 & 0.3 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{bmatrix}$$

Given $\mathbf{\Pi}$ and $\mathbf{A}$, it is easy to generate K random sequences each of length T. Let us see how we can calculate the probability of a sequence. Assume that the first four balls are "red, red, green, green." This corresponds to the observation sequence $O = \{S_1, S_1, S_3, S_3\}$. Its probability is

$$\begin{aligned} P(O|\mathbf{A}, \mathbf{\Pi}) &= P(S_1) \cdot P(S_1|S_1) \cdot P(S_3|S_1) \cdot P(S_3|S_3) \\ &= \pi_1 \cdot a_{11} \cdot a_{13} \cdot a_{33} \\ &= 0.5 \cdot 0.4 \cdot 0.3 \cdot 0.8 = 0.048 \end{aligned} \tag{15.7}$$

Now, let us see how we can learn the parameters, $\mathbf{\Pi}, \mathbf{A}$. Given K sequences of length T, where q_t^k is the state at time t of sequence k, the initial probability estimate is the number of sequences starting with S_i divided by the number of sequences:

$$\hat{\pi}_i = \frac{\#\{\text{sequences starting with } S_i\}}{\#\{\text{sequences}\}} = \frac{\sum_k 1(q_1^k = S_i)}{K} \tag{15.8}$$

where $1(b)$ is 1 if b is true and 0 otherwise.

As for the transition probabilities, the estimate for a_{ij} is the number of transitions from S_i to S_j divided by the total number of transitions from S_i over all sequences:

$$\hat{a}_{ij} = \frac{\#\{\text{transitions from } S_i \text{ to } S_j\}}{\#\{\text{transitions from } S_i\}} = \frac{\sum_k \sum_{t=1}^{T-1} 1(q_t^k = S_i \text{ and } q_{t+1}^k = S_j)}{\sum_k \sum_{t=1}^{T-1} 1(q_t^k = S_i)} \tag{15.9}$$

$\hat{a}_{12}$ is the number of times a blue ball follows a red ball divided by the total number of red ball draws over all sequences.

15.3 Hidden Markov Models

HIDDEN MARKOV MODEL

In a *hidden Markov model* (HMM), the states are not observable, but when we visit a state, an observation is recorded that is a probabilistic function of the state. We assume a discrete observation in each state from the set $\{v_1, v_2, \ldots, v_M\}$:

$$b_j(m) \equiv P(O_t = v_m | q_t = S_j) \tag{15.10}$$

OBSERVATION PROBABILITY
EMISSION PROBABILITY

$b_j(m)$ is the *observation probability*, or *emission probability*, that we observe the value $v_m, m = 1, \ldots, M$ in state S_j. We again assume a homogeneous model in which the probabilities do not depend on t. The values thus observed constitute the observation sequence O. The state sequence Q is not observed, that is what makes the model "hidden," but it should be inferred from the observation sequence O. Note that there are typically many different state sequences Q that could have generated the same observation sequence O, but with different probabilities; just as, given an iid sample from a normal distribution, there are an infinite number of (μ, σ) value pairs possible, we are interested in the one having the highest likelihood of generating the sample.

Note also that in this case of a hidden Markov model, there are two sources of randomness. In addition to randomly moving from one state to another, the observation in a state is also random.

Let us go back to our example. The hidden case corresponds to the urn-and-ball example where each urn contains balls of different colors. Let $b_j(m)$ denote the probability of drawing a ball of color m from urn j. We again observe a sequence of ball colors but without knowing the sequence of urns from which the balls were drawn. So it is as if now the urns are placed behind a curtain and somebody picks a ball at random from one of the urns and shows us only the ball, without showing us the urn from which it is picked. The ball is returned to the urn to keep the probabilities the same. The number of ball colors may be different from the number of urns. For example, let us say we have three urns and the observation sequence is

$$O = \{\text{red, red, green, blue, yellow}\}$$

In the previous case, knowing the observation (ball color), we knew the state (urn) exactly because there were separate urns for separate colors and each urn contained balls of only one color. The observable model is a special case of the hidden model where $M = N$ and $b_j(m)$ is 1 if $j = m$

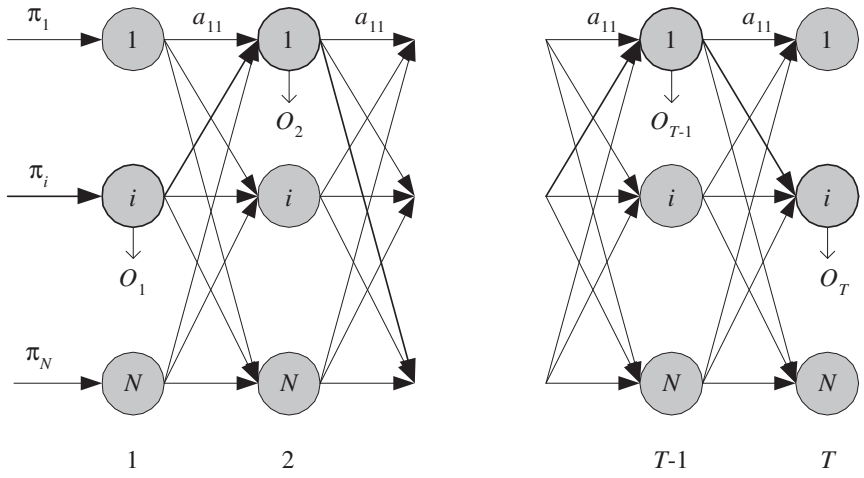


Figure 15.2 An HMM unfolded in time as a lattice (or trellis) showing all the possible trajectories. One path, shown in thicker lines, is the actual (unknown) state trajectory that generated the observation sequence.

and 0 otherwise. But in the case of a hidden model, a ball could have been picked from any urn. In this case, for the same observation sequence O, there may be many possible state sequences Q that could have generated O (see figure 15.2).

To summarize and formalize, an HMM has the following elements:

1. N: Number of states in the model

   $S = \{S_1, S_2, \ldots, S_N\}$

2. M: Number of distinct observation symbols in the *alphabet*

   $V = \{v_1, v_2, \ldots, v_M\}$

3. State transition probabilities:

   $\mathbf{A} = [a_{ij}]$ where $a_{ij} \equiv P(q_{t+1} = S_j | q_t = S_i)$

4. Observation probabilities:

   $\mathbf{B} = [b_j(m)]$ where $b_j(m) \equiv P(O_t = v_m | q_t = S_j)$

5. Initial state probabilities:

$$\mathbf{\Pi} = [\pi_i] \text{ where } \pi_i \equiv P(q_1 = S_i)$$

N and M are implicitly defined in the other parameters so $\lambda = (\mathbf{A}, \mathbf{B}, \mathbf{\Pi})$ is taken as the parameter set of an HMM. Given λ, the model can be used to generate an arbitrary number of observation sequences of arbitrary length, but as usual, we are interested in the other direction, that of estimating the parameters of the model given a training set of sequences.

15.4 Three Basic Problems of HMMs

Given a number of sequences of observations, we are interested in three problems:

1. Given a model λ, we would like to evaluate the probability of any given observation sequence, $O = \{O_1 O_2 \cdots O_T\}$, namely, $P(O|\lambda)$.

2. Given a model λ and an observation sequence O, we would like to find out the state sequence $Q = \{q_1 q_2 \cdots q_T\}$, which has the highest probability of generating O; namely, we want to find Q^* that maximizes $P(Q|O, \lambda)$.

3. Given a training set of observation sequences, $\mathcal{X} = \{O^k\}_k$, we would like to learn the model that maximizes the probability of generating $\mathcal{X}$; namely, we want to find λ^* that maximizes $P(\mathcal{X}|\lambda)$.

Let us see solutions to these one by one, with each solution used to solve the next problem, until we get to calculating λ or learning a model from data.

15.5 Evaluation Problem

Given an observation sequence $O = \{O_1 O_2 \cdots O_T\}$ and a state sequence $Q = \{q_1 q_2 \cdots q_T\}$, the probability of observing O given the state sequence Q is simply

$$P(O|Q, \lambda) = \prod_{t=1}^{T} P(O_t|q_t, \lambda) = b_{q_1}(O_1) \cdot b_{q_2}(O_2) \cdots b_{q_T}(O_T) \tag{15.11}$$

which we cannot calculate because we do not know the state sequence. The probability of the state sequence Q is

$$P(Q|\lambda) = P(q_1)\prod_{t=2}^{T} P(q_t|q_{t-1}) = \pi_{q_1} a_{q_1 q_2} \cdots a_{q_{T-1} q_T} \tag{15.12}$$

Then the joint probability is

$$\begin{aligned} P(O,Q|\lambda) &= P(q_1)\prod_{t=2}^{T} P(q_t|q_{t-1})\prod_{t=1}^{T} P(O_t|q_t) \\ &= \pi_{q_1} b_{q_1}(O_1) a_{q_1 q_2} b_{q_2}(O_2) \cdots a_{q_{T-1} q_T} b_{q_T}(O_T) \end{aligned} \tag{15.13}$$

We can compute $P(O|\lambda)$ by marginalizing over the joint, namely, by summing up over all possible Q:

$$P(O|\lambda) = \sum_{\text{all possible } Q} P(O,Q|\lambda)$$

However, this is not practical since there are N^T possible Q, assuming that all the probabilities are nonzero. Fortunately, there is an efficient procedure to calculate $P(O|\lambda)$, which is called the *forward-backward procedure* (see figure 15.3). It is based on the idea of dividing the observation sequence into two parts: the first one starting from time 1 until time t, and the second one from time $t+1$ until T.

FORWARD-BACKWARD PROCEDURE

FORWARD VARIABLE

We define the *forward variable* $\alpha_t(i)$ as the probability of observing the partial sequence $\{O_1 \cdots O_t\}$ until time t and being in S_i at time t, given the model λ:

$$\alpha_t(i) \equiv P(O_1 \cdots O_t, q_t = S_i|\lambda) \tag{15.14}$$

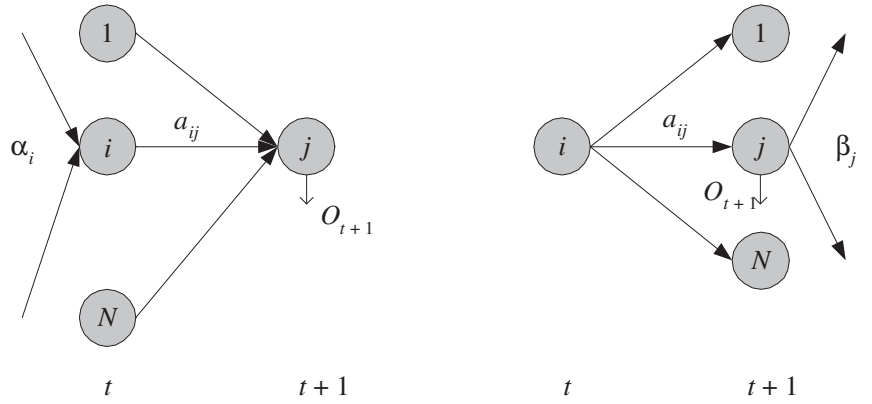
The nice thing about this is that it can be calculated recursively by accumulating results on the way.

- Initialization:

$$\begin{aligned} \alpha_1(i) &\equiv P(O_1, q_1 = S_i|\lambda) \\ &= P(O_1|q_1 = S_i, \lambda) P(q_1 = S_i|\lambda) \\ &= \pi_i b_i(O_1) \end{aligned} \tag{15.15}$$

- Recursion (see figure 15.3a):

$$\alpha_{t+1}(j) \equiv P(O_1 \cdots O_{t+1}, q_{t+1} = S_j|\lambda)$$



(a) Forward (b) Backward

Figure 15.3 Forward-backward procedure: (a) computation of $\alpha_t(j)$ and (b) computation of $\beta_t(i)$.

$$
\begin{aligned}
&= P(O_1 \cdots O_{t+1} | q_{t+1} = S_j, \lambda) P(q_{t+1} = S_j | \lambda) \\
&= P(O_1 \cdots O_t | q_{t+1} = S_j, \lambda) P(O_{t+1} | q_{t+1} = S_j, \lambda) P(q_{t+1} = S_j | \lambda) \\
&= P(O_1 \cdots O_t, q_{t+1} = S_j | \lambda) P(O_{t+1} | q_{t+1} = S_j, \lambda) \\
&= P(O_{t+1} | q_{t+1} = S_j, \lambda) \sum_i P(O_1 \cdots O_t, q_t = S_i, q_{t+1} = S_j | \lambda) \\
&= P(O_{t+1} | q_{t+1} = S_j, \lambda) \\
&\quad \sum_i P(O_1 \cdots O_t, q_{t+1} = S_j | q_t = S_i, \lambda) P(q_t = S_i | \lambda) \\
&= P(O_{t+1} | q_{t+1} = S_j, \lambda) \\
&\quad \sum_i P(O_1 \cdots O_t | q_t = S_i, \lambda) P(q_{t+1} = S_j | q_t = S_i, \lambda) P(q_t = S_i | \lambda) \\
&= P(O_{t+1} | q_{t+1} = S_j, \lambda) \\
&\quad \sum_i P(O_1 \cdots O_t, q_t = S_i | \lambda) P(q_{t+1} = S_j | q_t = S_i, \lambda) \\
&= \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij} \right] b_j(O_{t+1}) \qquad (15.16)
\end{aligned}
$$

$\alpha_t(i)$ explains the first t observations and ends in state S_i. We multiply this by the probability a_{ij} to move to state S_j, and because there are

N possible previous states, we need to sum up over all such possible previous S_i. $b_j(O_{t+1})$ then is the probability we generate the $(t+1)$st observation while in state S_j at time $t+1$.

When we calculate the forward variables, it is easy to calculate the probability of the observation sequence:

$$\begin{aligned} P(O|\lambda) &= \sum_{i=1}^{N} P(O, q_T = S_i|\lambda) \\ &= \sum_{i=1}^{N} \alpha_T(i) \end{aligned} \tag{15.17}$$

$\alpha_T(i)$ is the probability of generating the full observation sequence and ending up in state S_i. We need to sum up over all such possible final states.

Computing $\alpha_t(i)$ is $\mathcal{O}(N^2T)$, and this solves our first evaluation problem in a reasonable amount of time. We do not need it now but let us similarly define the *backward variable*, $\beta_t(i)$, which is the probability of being in S_i at time t and observing the partial sequence $O_{t+1} \cdots O_T$:

BACKWARD VARIABLE

$$\beta_t(i) \equiv P(O_{t+1} \cdots O_T|q_t = S_i, \lambda) \tag{15.18}$$

This can again be recursively computed as follows, this time going in the backward direction:

- Initialization (arbitrarily to 1):

$$\beta_T(i) = 1$$

- Recursion (see figure 15.3b):

$$\begin{aligned} \beta_t(i) &\equiv P(O_{t+1} \cdots O_T|q_t = S_i, \lambda) \\ &= \sum_j P(O_{t+1} \cdots O_T, q_{t+1} = S_j|q_t = S_i, \lambda) \\ &= \sum_j P(O_{t+1} \cdots O_T|q_{t+1} = S_j, q_t = S_i, \lambda) P(q_{t+1} = S_j|q_t = S_i, \lambda) \\ &= \sum_j P(O_{t+1}|q_{t+1} = S_j, q_t = S_i, \lambda) \\ &\quad P(O_{t+2} \cdots O_T|q_{t+1} = S_j, q_t = S_i, \lambda) P(q_{t+1} = S_j|q_t = S_i, \lambda) \\ &= \sum_j P(O_{t+1}|q_{t+1} = S_j, \lambda) \end{aligned}$$

$$P(O_{t+2}\cdots O_T|q_{t+1}=S_j,\lambda)P(q_{t+1}=S_j|q_t=S_i,\lambda)$$

$$= \sum_{j=1}^{N} a_{ij}b_j(O_{t+1})\beta_{t+1}(j) \tag{15.19}$$

When in state S_i, we can go to N possible next states S_j, each with probability a_{ij}. While there, we generate the $(t+1)$st observation and $\beta_{t+1}(j)$ explains all the observations after time $t+1$, continuing from there.

One word of caution about implementation is necessary here: Both α_t and β_t values are calculated by multiplying small probabilities, and with long sequences we risk getting underflow. To avoid this, at each time step, we normalize $\alpha_t(i)$ by multiplying it with

$$c_t = \frac{1}{\sum_j \alpha_t(j)}$$

We also normalize $\beta_t(i)$ by multiplying it with the same c_t ($\beta_t(i)$ do not sum to 1). We cannot use equation 15.17 after normalization; instead, we have (Rabiner 1989)

$$P(O|\lambda) = \frac{1}{\prod_t c_t} \text{ or } \log P(O|\lambda) = -\sum_t \log c_t \tag{15.20}$$

15.6 Finding the State Sequence

We now move on to the second problem, that of finding the state sequence $Q = \{q_1 q_2 \cdots q_T\}$ having the highest probability of generating the observation sequence $O = \{O_1 O_2 \cdots O_T\}$, given the model λ.

Let us define $\gamma_t(i)$ as the probability of being in state S_i at time t, given O and λ, which can be computed as follows:

$$\begin{aligned}
\gamma_t(i) &\equiv P(q_t = S_i|O,\lambda) \qquad (15.21)\\
&= \frac{P(O|q_t=S_i,\lambda)P(q_t=S_i|\lambda)}{P(O|\lambda)}\\
&= \frac{P(O_1\cdots O_t|q_t=S_i,\lambda)P(O_{t+1}\cdots O_T|q_t=S_i,\lambda)P(q_t=S_i|\lambda)}{\sum_{j=1}^{N} P(O,q_t=S_j|\lambda)}\\
&= \frac{P(O_1\cdots O_t,q_t=S_i|\lambda)P(O_{t+1}\cdots O_T|q_t=S_i,\lambda)}{\sum_{j=1}^{N} P(O|q_t=S_j,\lambda)P(q_t=S_j|\lambda)}\\
&= \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N}\alpha_t(j)\beta_t(j)} \qquad (15.22)
\end{aligned}$$

Here we see how nicely $\alpha_t(i)$ and $\beta_t(i)$ split the sequence between them: The forward variable $\alpha_t(i)$ explains the starting part of the sequence until time t and ends in S_i, and the backward variable $\beta_t(i)$ takes it from there and explains the ending part until time T.

The numerator $\alpha_t(i)\beta_t(i)$ explains the whole sequence given that at time t, the system is in state S_i. We need to normalize by dividing this over all possible intermediate states that can be traversed at time t, and guarantee that $\sum_i \gamma_t(i) = 1$.

To find the state sequence, for each time step t, we can choose the state that has the highest probability:

$$q_t^* = \arg\max_i \gamma_t(i) \tag{15.23}$$

but this may choose S_i and S_j as the most probable states at time t and $t+1$ even when $a_{ij} = 0$. To find the single best state *sequence* (path), we use the *Viterbi algorithm*, based on dynamic programming, which takes such transition probabilities into account.

VITERBI ALGORITHM

Given state sequence $Q = q_1 q_2 \cdots q_T$ and observation sequence $O = O_1 \cdots O_T$, we define $\delta_t(i)$ as the probability of the highest probability path at time t that accounts for the first t observations and ends in S_i:

$$\delta_t(i) \equiv \max_{q_1 q_2 \cdots q_{t-1}} p(q_1 q_2 \cdots q_{t-1}, q_t = S_i, O_1 \cdots O_t | \lambda) \tag{15.24}$$

Then we can recursively calculate $\delta_{t+1}(i)$ and the optimal path can be read by backtracking from T, choosing the most probable at each instant. The algorithm is as follows:

1. Initialization:

$$\begin{aligned} \delta_1(i) &= \pi_i b_i(O_1) \\ \psi_1(i) &= 0 \end{aligned}$$

2. Recursion:

$$\begin{aligned} \delta_t(j) &= \max_i \delta_{t-1}(i) a_{ij} \cdot b_j(O_t) \\ \psi_t(j) &= \arg\max_i \delta_{t-1}(i) a_{ij} \end{aligned}$$

3. Termination:

$$\begin{aligned} p^* &= \max_i \delta_T(i) \\ q_T^* &= \arg\max_i \delta_T(i) \end{aligned}$$

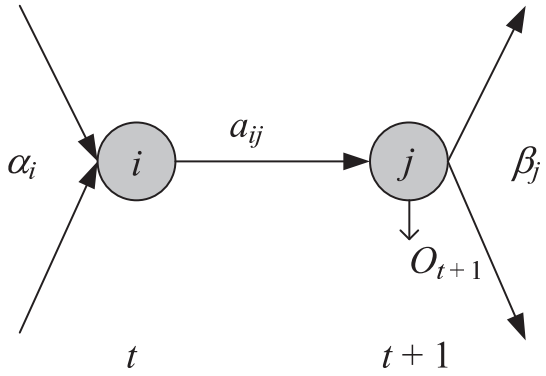


Figure 15.4 Computation of arc probabilities, $\xi_t(i,j)$.

4. Path (state sequence) backtracking:

$$q_t^* = \psi_{t+1}(q_{t+1}^*),\ t = T-1, T-2, \ldots, 1$$

Using the lattice structure of figure 15.2, $\psi_t(j)$ keeps track of the state that maximizes $\delta_t(j)$ at time $t-1$, that is, the best previous state. The Viterbi algorithm has the same complexity with the forward phase, where instead of the sum, we take the maximum at each step.

15.7 Learning Model Parameters

We now move on to the third problem, learning an HMM from data. The approach is maximum likelihood, and we would like to calculate λ^* that maximizes the likelihood of the sample of training sequences, $\mathcal{X} = \{O^k\}_{k=1}^K$, namely, $P(\mathcal{X}|\lambda)$. We start by defining a new variable that will become handy later on.

We define $\xi_t(i,j)$ as the probability of being in S_i at time t and in S_j at time $t+1$, given the whole observation O and λ:

$$\xi_t(i,j) \equiv P(q_t = S_i, q_{t+1} = S_j|O,\lambda) \tag{15.25}$$

which can be computed as follows (see figure 15.4):

$$\xi_t(i,j) \equiv P(q_t = S_i, q_{t+1} = S_j|O,\lambda)$$

$$
\begin{aligned}
&= \frac{P(O|q_t = S_i, q_{t+1} = S_j, \lambda)P(q_t = S_i, q_{t+1} = S_j|\lambda)}{P(O|\lambda)} \\
&= \frac{P(O|q_t = S_i, q_{t+1} = S_j, \lambda)P(q_{t+1} = S_j|q_t = S_i, \lambda)P(q_t = S_i|\lambda)}{P(O|\lambda)} \\
&= \left(\frac{1}{P(O|\lambda)}\right) P(O_1 \cdots O_t|q_t = S_i, \lambda)P(O_{t+1}|q_{t+1} = S_j, \lambda) \\
&\quad P(O_{t+2} \cdots O_T|q_{t+1} = S_j, \lambda)a_{ij}P(q_t = S_i|\lambda) \\
&= \left(\frac{1}{P(O|\lambda)}\right) P(O_1 \cdots O_t, q_t = S_i|\lambda)P(O_{t+1}|q_{t+1} = S_j, \lambda) \\
&\quad P(O_{t+2} \cdots O_T|q_{t+1} = S_j, \lambda)a_{ij} \\
&= \frac{\alpha_t(i)b_j(O_{t+1})\beta_{t+1}(j)a_{ij}}{\sum_k \sum_l P(q_t = S_k, q_{t+1} = S_l, O|\lambda)} \\
&= \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\sum_k \sum_l \alpha_t(k)a_{kl}b_l(O_{t+1})\beta_{t+1}(l)} \qquad (15.26)
\end{aligned}
$$

$\alpha_t(i)$ explains the first t observations and ends in state S_i at time t. We move on to state S_j with probability a_{ij}, generate the $(t+1)$st observation, and continue from S_j at time $t+1$ to generate the rest of the observation sequence. We normalize by dividing for all such possible pairs that can be visited at time t and $t+1$.

If we want, we can also calculate the probability of being in state S_i at time t by marginalizing over the arc probabilities for all possible next states:

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j) \qquad (15.27)$$

Note that if the Markov model were not hidden but observable, both $\gamma_t(i)$ and $\xi_t(i,j)$ would be 0/1. In this case when they are not, we estimate them with posterior probabilities that give us *soft counts*. This is just like the difference between supervised classification and unsupervised clustering where we did and did not know the class labels, respectively. In unsupervised clustering using EM (section 7.4), not knowing the class labels, we estimated them first (in the E-step) and calculated the parameters with these estimates (in the M-step).

SOFT COUNTS

BAUM-WELCH ALGORITHM

Similarly here we have the *Baum-Welch algorithm*, which is an EM procedure. At each iteration, first in the E-step, we compute $\xi_t(i,j)$ and $\gamma_t(i)$ values given the current $\lambda = (\mathbf{A}, \mathbf{B}, \mathbf{\Pi})$, and then in the M-step, we recalculate λ given $\xi_t(i,j)$ and $\gamma_t(i)$. These two steps are alternated until convergence during which, it has been shown, $P(O|\lambda)$ never decreases.

Assume indicator variables z_i^t as

$$z_i^t = \begin{cases} 1 & \text{if } q_t = S_i \\ 0 & \text{otherwise} \end{cases} \tag{15.28}$$

and

$$z_{ij}^t = \begin{cases} 1 & \text{if } q_t = S_i \text{ and } q_{t+1} = S_j \\ 0 & \text{otherwise} \end{cases} \tag{15.29}$$

These are 0/1 in the case of an observable Markov model and are hidden random variables in the case of an HMM. In this latter case, we estimate them in the E-step as

$$\begin{aligned} E[z_i^t] &= \gamma_t(i) \\ E[z_{ij}^t] &= \xi_t(i,j) \end{aligned} \tag{15.30}$$

In the M-step, we calculate the parameters given these estimated values. The expected number of transitions from S_i to S_j is $\sum_t \xi_t(i,j)$ and the total number of transitions from S_i is $\sum_t \gamma_t(i)$. The ratio of these two gives us the probability of transition from S_i to S_j at any time:

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{15.31}$$

Note that this is the same as equation 15.9, except that the actual counts are replaced by estimated soft counts.

The probability of observing v_m in S_j is the expected number of times v_m is observed when the system is in S_j over the total number of times the system is in S_j:

$$\hat{b}_j(m) = \frac{\sum_{t=1}^{T} \gamma_t(j) 1(O_t = v_m)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{15.32}$$

When there are multiple observation sequences

$$\mathcal{X} = \{O^k\}_{k=1}^K$$

which we assume to be independent

$$P(\mathcal{X}|\lambda) = \prod_{k=1}^{K} P(O^k|\lambda)$$

the parameters are now averages over all observations in all sequences:

$$\begin{aligned} \hat{a}_{ij} &= \frac{\sum_{k=1}^{K}\sum_{t=1}^{T_k-1}\xi_t^k(i,j)}{\sum_{k=1}^{K}\sum_{t=1}^{T_k-1}\gamma_t^k(i)} \\ \hat{b}_j(m) &= \frac{\sum_{k=1}^{K}\sum_{t=1}^{T_k}\gamma_t^k(j)1(O_t^k=v_m)}{\sum_{k=1}^{K}\sum_{t=1}^{T_k}\gamma_t^k(j)} \\ \hat{\pi}_i &= \frac{\sum_{k=1}^{K}\gamma_1^k(i)}{K} \end{aligned} \tag{15.33}$$

15.8 Continuous Observations

In our discussion, we assumed discrete observations modeled as a multinomial

$$P(O_t|q_t=S_j,\lambda)=\prod_{m=1}^{M} b_j(m)^{r_m^t} \tag{15.34}$$

where

$$r_m^t=\begin{cases} 1 & \text{if } O_t=v_m \\ 0 & \text{otherwise} \end{cases} \tag{15.35}$$

If the inputs are continuous, one possibility is to discretize them and then use these discrete values as observations. Typically, a vector quantizer (section 7.3) is used for this purpose of converting continuous values to the discrete index of the closest reference vector. For example, in speech recognition, a word utterance is divided into short speech segments corresponding to phonemes or part of phonemes; after preprocessing, these are discretized using a vector quantizer and an HMM is then used to model a word utterance as a sequence of them.

We remember that k-means used for vector quantization is the hard version of a Gaussian mixture model:

$$p(O_t|q_t=S_j,\lambda)=\sum_{l=1}^{L} P(\mathcal{G}_l)p(O_t|q_t=S_j,\mathcal{G}_l,\lambda) \tag{15.36}$$

where

$$p(O_t|q_t=S_j,\mathcal{G}_l,\lambda)\sim\mathcal{N}(\boldsymbol{\mu}_l,\boldsymbol{\Sigma}_l) \tag{15.37}$$

and the observations are kept continuous. In this case of Gaussian mixtures, EM equations can be derived for the component parameters (with

suitable regularization to keep the number of parameters in check) and the mixture proportions (Rabiner 1989).

Let us see the case of a scalar continuous observation, $O_t \in \Re$. The easiest is to assume a normal distribution:

$$p(O_t|q_t = S_j, \lambda) \sim \mathcal{N}(\mu_j, \sigma_j^2) \tag{15.38}$$

which implies that in state S_j, the observation is drawn from a normal with mean μ_j and variance σ_j^2. The M-step equations in this case are

$$\begin{aligned} \hat{\mu}_j &= \frac{\sum_t \gamma_t(j) O_t}{\sum_t \gamma_t(j)} \\ \hat{\sigma}_j^2 &= \frac{\sum_t \gamma_t(j)(O_t - \hat{\mu}_j)^2}{\sum_t \gamma_t(j)} \end{aligned} \tag{15.39}$$

15.9 The HMM as a Graphical Model

We discussed graphical models in chapter 14, and the hidden Markov model can also be depicted as a graphical model. The three successive states q_{t-2}, q_{t-1}, q_t correspond to the three states on a chain in a first-order Markov model. The state at time t, q_t, depends only on the state at time $t-1$, q_{t-1}, and given q_{t-1}, q_t is independent of q_{t-2}

$$P(q_t|q_{t-1}, q_{t-2}) = P(q_t|q_{t-1})$$

as given by the state transition probability matrix $\mathbf{A}$ (see figure 15.5). Each hidden variable generates a discrete observation that is observed, as given by the observation probability matrix $\mathbf{B}$. The forward-backward procedure of hidden Markov models we discuss in this chapter is an application of belief propagation that we discussed in section 14.5.

Continuing with the graphical formalism, different HMM types can be devised and depicted as different graphical models. In figure 15.6a, an *input-output HMM* is shown where there are two separate observed input-output sequences and there is also a sequence of hidden states (Bengio and Frasconi 1996). In some applications this is the case, namely, additional to the observation sequence O_t, we have an input sequence, x_t, and we know that the observation depends also on the input. In such a case, we condition the observation O_t in state S_j on the input x^t and write $P(O_t|q_t = S_j, x_t)$. When the observations are numeric, for example, we replace equation 15.38 with a generalized model

INPUT-OUTPUT HMM

$$p(O_t|q_t = S_j, x_t, \lambda) \sim \mathcal{N}(g_j(x^t|\theta_j), \sigma_j^2) \tag{15.40}$$

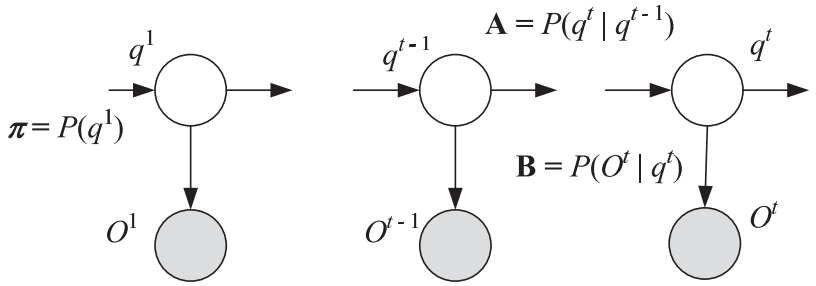


Figure 15.5 A hidden Markov model can be drawn as a graphical model where q^t are the hidden states and shaded O^t are observed.

where, for example, assuming a linear model, we have

$$g_j(x^t|w_j, w_{j0}) = w_j x^t + w_{j0} \tag{15.41}$$

If the observations are discrete and multinomial, we have a classifier taking x^t as input and generating a 1-of-M output, or we can generate posterior class probabilities and keep the observations continuous.

Similarly, the state transition probabilities can also be conditioned on the input, namely, $P(q_{t+1} = S_j|q_t = S_i, x_t)$, which is implemented by a classifier choosing the state at time $t+1$ as a function of the state at time t and the input. This is a *Markov mixture of experts* (Meila and Jordan 1996) and is a generalization of the mixture of experts architecture (section 12.8) where the gating network keeps track of the decision it made in the previous time step. This has the advantage that the model is no longer homogeneous; different observation and transition probabilities are used at different time steps. There is still a single model for each state, parameterized by θ_j, but it generates different transition or observation probabilities depending on the input seen. It is possible that the input is not a single value but a window around time t making the input a vector; this allows handling applications where the input and observation sequences have different lengths.

MARKOV MIXTURE OF EXPERTS

Even if there is no other explicit input sequence, an HMM with input can be used by generating an "input" through some prespecified function of previous observations

$$\boldsymbol{x}_t = \boldsymbol{f}(O_{t-\tau}, \ldots, O_{t-1})$$

thereby providing a window of size τ of contextual input.

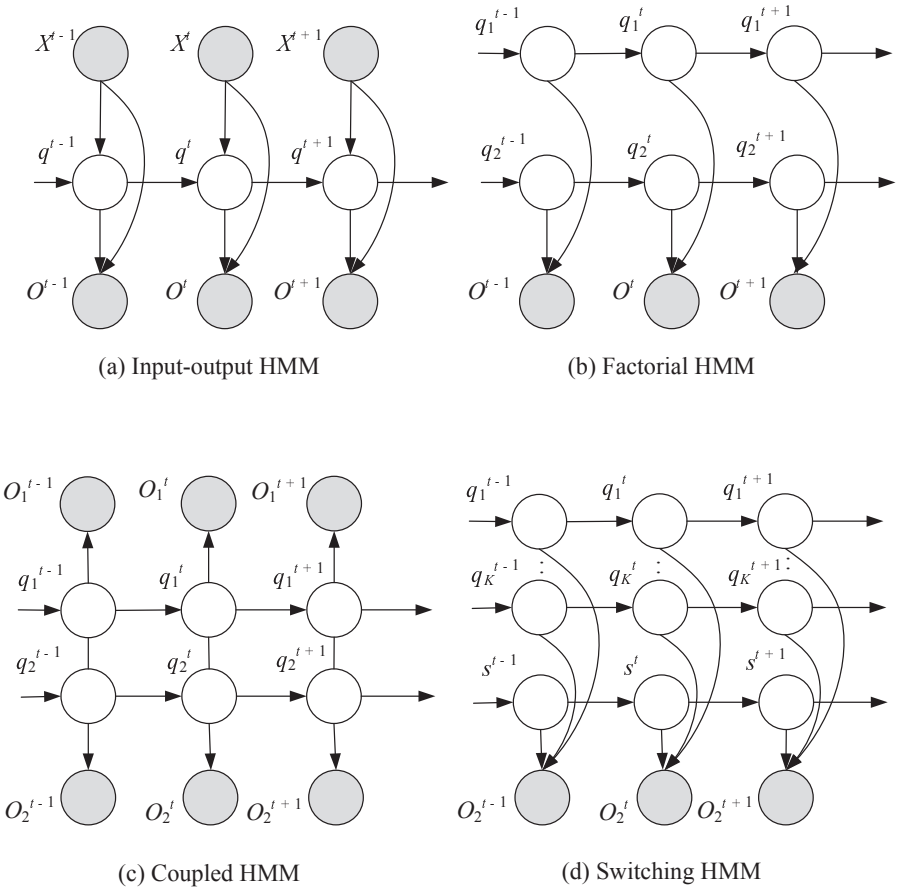


Figure 15.6 Different types of HMM model different assumptions about the way the observed data (shown shaded) is generated from Markov sequences of latent variables.

FACTORIAL HMM

PEDIGREE

Another HMM type that can be easily visualized is a *factorial HMM*, where there are multiple separate hidden sequences that interact to generate a single observation sequence. An example is a *pedigree* that displays the parent-child relationship (Jordan 2004); figure 15.6b models *meiosis* where the two sequences correspond to the chromosomes of the father and the mother (which are independent), and at each locus (gene), the offspring receives one allele from the father and the other allele from the mother.

COUPLED HMM

A *coupled HMM*, shown in figure 15.6c, models two parallel but interacting hidden sequences that generate two parallel observation sequences. For example, in speech recognition, we may have one observed acoustic sequence of uttered words and one observed visual sequence of lip images, each having its hidden states where the two are dependent.

SWITCHING HMM

In a *switching HMM*, shown in figure 15.6d, there are K parallel independent hidden state sequences, and the state variable S at any one time picks one of them and the chosen one generates the output. That is, we switch between state sequences as we go along.

LINEAR DYNAMICAL SYSTEM
KALMAN FILTER

In HMM proper, though the observation may be continuous, the state variable is discrete; in a *linear dynamical system*, also known as the *Kalman filter*, both the state and the observations are continuous. In the basic case, the state at time t is a linear function of the state at $t-1$ with additive zero-mean Gaussian noise, and, at each state, the observation is another linear function of the state with additive zero-mean Gaussian noise. The two linear mappings and the covariances of the two noise sources make up the parameters. All HMM variants we discussed earlier can similarly be generalized to use continuous states.

By suitably modifying the graphical model, we can adapt the architecture to the characteristics of the process that generates the data. This process of matching the model to the data is a model selection procedure to best trade off bias and variance. The disadvantage is that exact inference may no longer be possible on such extended HMMs, and we would need approximation or sampling methods (Ghahramani 2001; Jordan 2004).

15.10 Model Selection in HMMs

Just like any model, the complexity of an HMM should be tuned so as to balance its complexity with the size and properties of the data at hand. One possibility is to tune the topology of the HMM. In a fully connected (ergodic) HMM, there is transition from a state to any other state, which makes $\mathbf{A}$ a full $N \times N$ matrix. In some applications, only certain transitions are allowed, with the disallowed transitions having their $a_{ij} = 0$. When there are fewer possible next states, $N' < N$, the complexity of forward-backward passes and the Viterbi procedure is $\mathcal{O}(NN'T)$ instead of $\mathcal{O}(N^2T)$.

LEFT-TO-RIGHT HMMS

For example, in speech recognition, *left-to-right HMMs* are used, which

have their states ordered in time so that as time increases, the state index increases or stays the same. Such a constraint allows modeling sequences whose properties change over time as in speech, and when we get to a state, we know approximately the states preceding it. There is the property that we never move to a state with a smaller index, namely, $a_{ij} = 0, \text{for } j < i$. Large changes in state indices are not allowed either, namely, $a_{ij} = 0, \text{for } j > i + \tau$. The example of the left-to-right HMM given in figure 15.7 with $\tau = 2$ has the state transition matrix

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{bmatrix}$$

Another factor that determines the complexity of an HMM is the number of states N. Because the states are hidden, their number is not known and should be chosen before training. This is determined using prior information and can be fine-tuned by cross-validation, namely, by checking the likelihood of validation sequences.

When used for classification, we have a set of HMMs, each one modeling the sequences belonging to one class. For example, in spoken word recognition, examples of each word train a separate model, λ_i. Given a new word utterance O to classify, all of the separate word models are evaluated to calculate $P(O|\lambda_i)$. We then use Bayes' rule to get the posterior probabilities

$$P(\lambda_i|O) = \frac{P(O|\lambda_i)P(\lambda_i)}{\sum_j P(O|\lambda_j)P(\lambda_j)} \tag{15.42}$$

where $P(\lambda_i)$ is the prior probability of word i. The utterance is assigned to the word having the highest posterior. This is the likelihood-based approach; there is also work on discriminative HMM trained directly to maximize the posterior probabilities. When there are several pronunciations of the same word, these are defined as parallel paths in the HMM for the word.

PHONES

In the case of a continuous input like speech, the difficult task is that of segmenting the signal into small discrete observations. Typically, *phones* are used that are taken as the primitive parts, and combining them, longer sequences (e.g., words) are formed. Each phone is recognized in parallel (by the vector quantizer), then the HMM is used to combine them serially. If the speech primitives are simple, then the HMM becomes complex and vice versa. In connected speech recognition where the words are

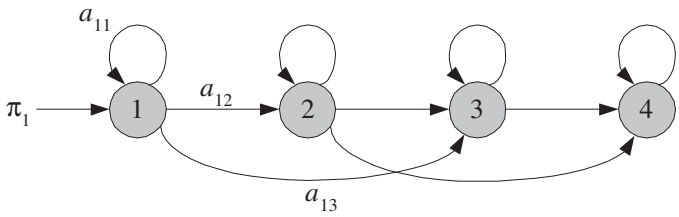


Figure 15.7 Example of a left-to-right HMM.

not uttered one by one with clear pauses between them, there is a hierarchy of HMMs at several levels; one combines phones to recognize words, another combines words to recognize sentences by building a language model, and so forth.

Hybrid neural network/HMM models were also used for speech recognition (Morgan and Bourlard 1995). In such a model, a multilayer perceptron (chapter 11) is used to capture temporally local but possibly complex and nonlinear primitives, for example, phones, while the HMM is used to learn the temporal structure. The neural network acts as a preprocessor and translates the raw observations in a time window to a form that is easier to model than the output of a vector quantizer.

An HMM can be visualized as a graphical model and evaluation in an HMM is a special case of the belief propagation algorithm that we discuss in chapter 14. The reason we devote a special chapter is the widespread successful use of this particular model, especially in automatic speech recognition. But the basic HMM architecture can be extended—for example, by having multiple sequences, or by introducing hidden (latent) variables, as we discuss in section 15.9.

In chapter 16, we discuss the Bayesian approach and in section 16.8, we discuss the nonparametric Bayesian methods where the model structure can be made more complex over time as more data arrives. One application of that is the *infinite HMM* (Beal, Ghahramani, and Rasmussen 2002).

15.11 Notes

The HMM is a mature technology, and there are HMM-based commercial speech recognition systems in actual use (Rabiner and Juang 1993; Jelinek 1997). In section 11.12, we discussed how to train multilayer

perceptrons for recognizing sequences. HMMs have the advantage over time delay neural networks in that no time window needs to be defined a priori, and they train better than recurrent neural networks. HMMs are applied to diverse sequence recognition tasks. Applications of HMMs to bioinformatics is given in Baldi and Brunak 1998, and to natural language processing in Manning and Schütze 1999. It is also applied to online handwritten character recognition, which differs from optical recognition in that the writer writes on a touch-sensitive pad and the input is a sequence of (x, y) coordinates of the pen tip as it moves over the pad and is not a static image. Bengio et al. (1995) explain a hybrid system for online recognition where an MLP recognizes individual characters, and an HMM combines them to recognize words. Various applications of the HMM and several extensions, for example, discriminative HMMs, are discussed in Bengio 1999. A more recent survey of what HMMs can and cannot do is Bilmes 2006.

In any such recognition system, one critical point is to decide how much to do things in parallel and what to leave to serial processing. In speech recognition, phonemes may be recognized by a parallel system that corresponds to assuming that all the phoneme sound is uttered in one time step. The word is then recognized serially by combining the phonemes. In an alternative system, phonemes themselves may be designed as a sequence of simpler speech sounds, if the same phoneme has many versions, for example, depending on the previous and following phonemes. Doing things in parallel is good but only to a degree; one should find the ideal balance of parallel and serial processing. To be able to call anyone at the touch of a button, we would need millions of buttons on our telephone; instead, we have ten buttons and we press them in a sequence to dial the number.

We discussed graphical models in chapter 14, and we know that HMMs can be considered a special class of graphical models and inference and learning operations on HMMs are analogous to their counterparts in graphical models (Smyth, Heckerman, and Jordan 1997). There are various extensions to HMMs, such as *factorial HMMs*, where at each time step, there are a number of states that collectively generate the observation and *tree-structured HMMs* where there is a hierarchy of states. The general formalism also allows us to treat continuous as well as discrete states, known as *linear dynamical systems*. For some of these models, exact inference is not possible and one needs to use approximation or sampling methods (Ghahramani 2001).

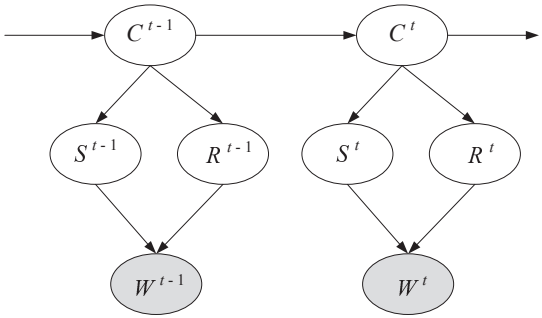


Figure 15.8 A dynamic version where we have a chain of graphs to show dependency in weather on consecutive days.

Actually, any graphical model can be extended in time by unfolding it in time and adding dependencies between successive copies. In fact, a hidden Markov model is nothing but a sequence of clustering problems where the cluster index at time t is dependent not only on observation at time t but also on the index at time $t-1$, and the Baum-Welch algorithm is expectation-maximization extended to also include this dependency in time. In section 6.5, we discussed factor analysis where a small number of hidden factors generate the observation; similarly, a linear dynamical system may be viewed as a sequence of such factor analysis models where the current factors also depend on the previous factors.

This dynamic dependency may be added when needed. For example, figure 14.5 models the cause of wet grass for a particular day; if we believe that yesterday's weather has an influence on today's weather (and we should—it tends to be cloudy on successive days, then sunny for a number of days, and so on), we can have the dynamic graphical model shown in figure 15.8 where we model this dependency.

15.12 Exercises

1. Given the observable Markov model with three states, S_1, S_2, S_3, initial probabilities

   $$\mathbf{\Pi} = [0.5, 0.2, 0.3]^T$$

and transition probabilities

$$\mathbf{A} = \begin{bmatrix} 0.4 & 0.3 & 0.3 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{bmatrix}$$

generate 100 sequences of 1,000 states.

2. Using the data generated by the previous exercise, estimate Π, $\mathbf{A}$ and compare with the parameters used to generate the data.
3. Formalize a second-order Markov model. What are the parameters? How can we calculate the probability of a given state sequence? How can the parameters be learned for the case of an observable model?

   SOLUTION: In a second-order model, the current state depends on the two previous states:

   $a_{ijk} \equiv P(q_{t+2} = S_k | q_{t+1} = S_j, q_t = S_i)$

   Initial state probability defines the probability of the first state:

   $\pi_i \equiv P(q_1 = S_i)$

   We also need parameters to define the probability of the second state given the first state:

   $\theta_{ij} \equiv P(q_2 = S_j | q_1 = S_i)$

   Given a second-order observable MM with parameters $\lambda = (\Pi, \Theta, \mathbf{A})$, the probability of an observed state sequence is

$$\begin{aligned} P(O = Q|\lambda) &= P(q_1)P(q_2|q_1)\prod_{t=3}^{T} P(q_t|q_{t-1}, q_{t-2}) \\ &= \pi_{q_1}\theta_{q_2q_1}a_{q_3q_2q_1}a_{q_4q_3q_2}\cdots a_{q_Tq_{T-1}q_{T-2}} \end{aligned}$$

   The probabilities are estimated as proportions:

$$\begin{aligned} \hat{\pi}_i &= \frac{\sum_k 1(q_1^k = S_i)}{K} \\ \hat{\theta}_{ij} &= \frac{\sum_k 1(q_2^k = S_j \text{ and } q_1^k = S_i)}{\sum_k 1(q_1^k = S_i)} \\ \hat{a}_{ijk} &= \frac{\sum_k \sum_{t=3}^{T} 1(q_t^k = S_k \text{ and } q_{t-1}^k = S_j \text{ and } q_{t-2}^k = S_i)}{\sum_k \sum_{t=3}^{T} 1(q_{t-1}^k = S_j \text{ and } q_{t-2}^k = S_i)} \end{aligned}$$

4. Show that any second- (or higher-order) Markov model can be converted to a first-order Markov model.

   SOLUTION: In a second-order model, each state depends on the two previous states. We can define a new set of states corresponding to the Cartesian product of the original set of states with itself. A first-order model defined on this new N^2 states corrresponds to a second-order model defined on the original N states.

5. Some researchers define a Markov model as generating an observation while traversing an arc, instead of on arrival at a state. Is this model any more powerful than what we have discussed?

   SOLUTION: Similar to the case of the previous exercise, if the output depends not only on the current state but also on the next state, we can define new states corresponding to this pair and have the output generated by this (joint) state.

6. Generate training and validation sequences from an HMM of your choosing. Then train different HMMs by varying the number of hidden states on the same training set and calculate the validation likelihoods. Observe how the validation likelihood changes as the number of states increases.

7. If in equation 15.38 we have multivariate observations, what will the M-step equations be?

   SOLUTION: If we have d-dimensional $\mathbf{O}_t \in \Re^d$, drawn from d-variate Gaussians with their mean vectors and covariance matrices

   $$p(\mathbf{O}_t|q_t = S_j, \lambda) \sim \mathcal{N}(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$$

   the M-step equations are

   $$\hat{\boldsymbol{\mu}}_j = \frac{\sum_t \gamma_t(j)\mathbf{O}_t}{\sum_t \gamma_t(j)}$$

   $$\hat{\boldsymbol{\Sigma}}_j = \frac{\sum_t \gamma_t(j)(\mathbf{O}_t - \hat{\boldsymbol{\mu}}_j)(\mathbf{O}_t - \hat{\boldsymbol{\mu}}_j)^T}{\sum_t \gamma_t(j)}$$

8. Consider the urn-and-ball example where we draw *without replacement*. How will it be different?

   SOLUTION: If we draw without replacement, then at each iteration, the number of balls change, which means that the observation probabilities, $\mathbf{B}$, change. We will no longer have a homogenous model.

9. Let us say at any time we have two observations from two different alphabets; for example, let us say we are observing the values of two currencies every day. How can we implement this using HMM?

   SOLUTION: In such a case, what we have is a hidden state generating two different observations. That is, we have two $\mathbf{B}$, each trained with its own observation sequence. These two observations then need to be combined to estimate $\mathbf{A}$ and π.

10. How can we have an incremental HMM where we add new hidden states when necessary?

    SOLUTION: Again, this is a state space search. Our aim may be to maximize validation log likelihood, and an operator allows us to add a hidden state. We do then a forward search. There are structure learning algorithms for the more general case of graphical models, which we discussed in chapter 14.

15.13 References

Baldi, P., and S. Brunak. 1998. *Bioinformatics: The Machine Learning Approach*. Cambridge, MA: MIT Press.

Beal, M. J., Z. Ghahramani, and C. E. Rasmussen. 2002. "The Infinite Hidden Markov Model." In *Advances in Neural Information Processing Systems 14*, ed. T. G. Dietterich, S. Becker, and Z. Ghahramani, 577–585. Cambridge, MA: MIT Press.

Bengio, Y. 1999. "Markovian Models for Sequential Data." *Neural Computing Surveys* 2: 129–162.

Bengio, Y., and P. Frasconi. 1996. "Input-Output HMMs for Sequence Processing." *IEEE Transactions on Neural Networks* 7:1231–1249.

Bengio, Y., Y. Le Cun, C. Nohl, and C. Burges. 1995. "LeRec: A NN/HMM Hybrid for On-line Handwriting Recognition." *Neural Computation* 7:1289–1303.

Bilmes, J. A. 2006. "What HMMs Can Do." *IEICE Transactions on Information and Systems* E89-D:869–891.

Ghahramani, Z. 2001. "An Introduction to Hidden Markov Models and Bayesian Networks." *International Journal of Pattern Recognition and Artificial Intelligence* 15:9–42.

Jelinek, F. 1997. *Statistical Methods for Speech Recognition*. Cambridge, MA: MIT Press.

Jordan, M. I. 2004. "Graphical Models." *Statistical Science* 19:140–155.

Manning, C. D., and H. Schütze. 1999. *Foundations of Statistical Natural Language Processing*. Cambridge, MA: MIT Press.

Meila, M., and M. I. Jordan. 1996. "Learning Fine Motion by Markov Mixtures of Experts." In *Advances in Neural Information Processing Systems 8*, ed. D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, 1003–1009. Cambridge, MA: MIT Press.

Morgan, N., and H. Bourlard. 1995. "Continuous Speech Recognition: An Introduction to the Hybrid HMM/Connectionist Approach." *IEEE Signal Processing Magazine* 12:25–42.

Smyth, P., D. Heckerman, and M. I. Jordan. 1997. "Probabilistic Independence Networks for Hidden Markov Probability Models." *Neural Computation* 9:227–269.

Rabiner, L. R. 1989. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition." *Proceedings of the IEEE* 77:257–286.

Rabiner, L. R., and B. H. Juang. 1986. "An Introduction to Hidden Markov Models." *IEEE Acoustics, Speech, and Signal Processing Magazine* 3:4–16.

Rabiner, L. R., and B. H. Juang. 1993. *Fundamentals of Speech Recognition.* New York: Prentice Hall.

16 *Bayesian Estimation*

In the Bayesian approach, we consider parameters as random variables with a distribution allowing us to model our uncertainty in estimating them. We continue from where we left off in section 4.4 and discuss estimating both the parameters of a distribution and the parameters of a model for regression, classification, clustering, or dimensionality reduction. We also discuss nonparametric Bayesian modeling where model complexity is not fixed but depends on the data.

16.1 Introduction

BAYESIAN ESTIMATION, which we introduced in section 4.4, treats a parameter θ as a random variable with a probability distribution. The maximum likelihood approach we discussed in section 4.2 treats a parameter as an unknown constant. For example, if the parameter we want to estimate is the mean μ, its maximum likelihood estimator is the sample average $\overline{X}$. We calculate $\overline{X}$ over our training set, plug it in our model, and use it, for example, for classification. However, we know that especially with small samples, the maximum likelihood estimator can be a poor estimator and has variance—as the training set varies, we may calculate different values of $\overline{X}$, which in turn may lead to different discriminants with different generalization accuracies.

In Bayesian estimation, we make use of the fact that we have uncertainty in estimating θ and instead of a single θ_{ML}, we use all θ weighted by our estimated distribution, $p(\theta|\mathcal{X})$. That is, we average over our uncertainty in estimating θ.

While estimating $p(\theta|\mathcal{X})$, we can make use of the prior information we

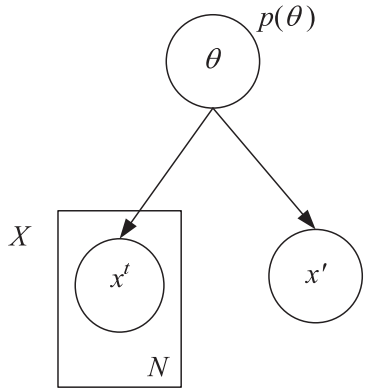


Figure 16.1 The generative graphical model (see chapter 14). The arcs are in the direction of sampling; first we pick θ from $p(\theta)$, and then we generate data by sampling from $p(x|\theta)$. The rectangular *plate* contains N independent instances drawn, and they make up the training set $\mathcal{X}$. The new x' is independently drawn given θ. This is the iid assumption. If θ is not known, they are dependent. We infer θ from the past instances using Bayes' rule, which is then used to make inference about the new x'.

may have regarding the value of the parameter. Such prior beliefs are especially important when we have a small sample (and when the variance of the maximum likelihood estimator is high). In such a case, we are interested in combining what the data tells us, namely, the value calculated from the sample, and our prior information. As we first discussed in section 4.4, we code this information using a *prior probability* distribution. For example, before looking at a sample to estimate the mean, we may have some prior belief that it is close to 2, between 1 and 3, and in such a case, we write $p(\mu)$ in such a way that the bulk of the density lies in the interval $[1, 3]$.

PRIOR PROBABILITY

Using Bayes' rule, we combine the prior and the likelihood and calculate the *posterior probability* distribution:

POSTERIOR PROBABILITY

$$p(\theta|\mathcal{X}) = \frac{p(\theta)p(\mathcal{X}|\theta)}{p(\mathcal{X})} \tag{16.1}$$

Here, $p(\theta)$ is the prior density; it is what we know regarding the possible values that θ may take *before* looking at the sample. $p(\mathcal{X}|\theta)$ is the *sample likelihood*; it tells us how likely our sample $\mathcal{X}$ is if the parameter of the distribution takes the value θ. For example, if the instances in our

sample are between 5 and 10, such a sample is likely if μ is 7 but is less likely if μ is 3 and even less likely if μ is 1. $p(\mathcal{X})$ in the denominator is a normalizer to make sure that the *posterior* $p(\theta|\mathcal{X})$ integrates to 1. It is called the posterior probability because it tells us how likely θ takes a certain value *after* looking at the sample. The Bayes' rule takes the prior distribution, combines it with what the data reveals, and generates the posterior distribution. We then use this posterior distribution later for making inference.

Let us say that we have a past sample $\mathcal{X} = \{\boldsymbol{x}^t\}_{t=1}^N$ drawn from sone distribution with unknown parameter θ. We can then draw one more instance x', and we would like to calculate its probability distribution. We can visualize this as a graphical model (chapter 14) as shown in figure 16.1. What is shown here is a *generative model* representing how the data is generated. We first sample θ from $p(\theta)$ and then sample from $p(x|\theta)$ to first generate the training instances $\boldsymbol{x}^t$ and also the new test x'.

GENERATIVE MODEL

We write the joint as

$$p(x', \mathcal{X}, \theta) = p(\theta)p(\mathcal{X}|\theta)p(x'|\theta)$$

We can estimate the probability distribution for the new x given the sample $\mathcal{X}$:

$$\begin{aligned} p(x'|\mathcal{X}) &= \frac{p(x', \mathcal{X})}{p(\mathcal{X})} = \frac{\int p(x', \mathcal{X}, \theta)d\theta}{p(\mathcal{X})} = \frac{\int p(\theta)p(\mathcal{X}|\theta)p(x'|\theta)d\theta}{p(\mathcal{X})} \\ &= \int p(x'|\theta)p(\theta|\mathcal{X})d\theta \end{aligned} \tag{16.2}$$

In calculating $p(\theta|\mathcal{X})$, Bayes' rule inverts the direction of the arc and makes a diagnostic inference. This inferred (posterior) distribution is then used to derive a predictive distribution for new x.

We see that our estimate is a weighted sum (we replace $\int d\theta$ by $\sum_\theta$ if θ is discrete valued) of estimates using all possible values of θ weighted by how likely each θ is, given the sample $\mathcal{X}$.

This is the *full Bayesian treatment* and it may not be possible if the posterior is not easy to integrate. As we saw in section 4.4, in the case of the *maximum a posteriori (MAP) estimate*, we use the mode of the posterior:

MAXIMUM A POSTERIORI (MAP) ESTIMATE

$$\theta_{MAP} = \arg\max_\theta p(\theta|\mathcal{X}) \text{ and } p_{MAP}(x'|\mathcal{X}) = p(x'|\theta_{MAP})$$

The MAP estimate corresponds to assuming that the posterior makes a very narrow peak around a single point, that is, the mode. If the prior

$p(\theta)$ is uniform over all θ, then the mode of the posterior $p(\theta|\mathcal{X})$ and the mode of the likelihood $p(\mathcal{X}|\theta)$ are at the same point, and the MAP estimate is equal to the maximum likelihood (ML) estimate:

$$\theta_{ML} = \arg\max_{\theta} p(\mathcal{X}|\theta) \text{ and } p_{ML}(x'|\mathcal{X}) = p(x'|\theta_{ML})$$

This implies that using ML corresponds to assuming no a priori distinction between different values of θ.

Basically, the Bayesian approach has two advantages:

1. The prior helps us ignore the values that θ is unlikely to take and concentrate on the region where it is likely to lie. Even a weak prior with long tails can be very helpful.

2. Instead of using a single θ estimate in prediction, we generate a set of possible θ values (as defined by the posterior) and use all of them in prediction, weighted by how likely they are.

If we use the MAP estimate instead of integrating over θ, we make use of the first advantage but not the second—if we use the ML estimate, we lose both advantages. If we use an uninformative (uniform) prior, we make use of the second advantage but not the first. Actually it is this second advantage, rather than the first, that makes the Bayesian approach interesting, and in chapter 17, we discuss combining multiple models where we see methods that are very similar, though not always Bayesian.

This approach can be used in different types of distributions and for different types of applications. The parameter θ can be the parameter of a distribution. For example, in classification, it can be the unknown class mean, for which we define a prior and get its posterior; then, we get a different discriminant for each possible value of the mean and hence the Bayesian approach will average over all possible discriminants whereas in the ML approach there is a single mean estimate and hence a single discriminant.

The unknown parameter, as we will see shortly, can also be the parameters of a fitted model. For example, in linear regression, we can define a prior distribution on the slope and the intercept parameters and calculate a posterior on them, that is, a distribution over lines. We will then be averaging over the prediction of all possible lines, weighted by how likely they are as specified by their prior weights and how well they fit the given data.

One of the most critical aspects of Bayesian estimation is evaluating the integral in equation 16.2. For some cases, we can calculate it, but mostly we cannot, and in such cases, we need to approximate it, and we will see methods for this in the next few sections, namely, Laplace and variational approximations, and Markov chain Monte Carlo (MCMC) sampling.

Now, let us see these and other applications of the Bayesian approach in more detail, starting from simple and incrementally making them more complex.

16.2 Bayesian Estimation of the Parameters of a Discrete Distribution

16.2.1 $K > 2$ States: Dirichlet Distribution

Let us say that each instance is a multinomial variable taking one of K distinct states (section 4.2.2). We say $x_i^t = 1$ if instance t is in state i and $x_j^t = 0, \forall j \neq i$. The parameters are the probabilities of states, $\boldsymbol{q} = [q_1, q_2, \ldots, q_K]^T$ with $q_i, i = 1, \ldots, K$ satisfying $q_i \geq 0, \forall i$ and $\sum_i q_i = 1$.

For example, x^t may correspond to news documents and states may correspond to K different news categories: sports, politics, arts, and so on. The probabilities q_i then correspond to the proportions of different news categories, and priors on them allow us to code our prior beliefs in these proportions; for example, we may expect to have more news related to sports than news related to arts.

The sample likelihood is

$$p(\mathcal{X}|\boldsymbol{q}) = \prod_{t=1}^{N} \prod_{i=1}^{K} q_i^{x_i^t}$$

DIRICHLET DISTRIBUTION

The prior distribution of $\boldsymbol{q}$ is the *Dirichlet distribution*:

$$\text{Dirichlet}(\boldsymbol{q}|\boldsymbol{\alpha}) = \frac{\Gamma(\alpha_0)}{\Gamma(\alpha_1)\cdots\Gamma(\alpha_K)} \prod_{i=1}^{K} q_i^{\alpha_i - 1}$$

where $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_K]^T$ and $\alpha_0 = \sum_i \alpha_i$. α_i, the parameters of the prior, are called the *hyperparameters*. $\Gamma(x)$ is the *gamma function* defined as

GAMMA FUNCTION

$$\Gamma(x) \equiv \int_0^\infty u^{x-1} e^{-u} du$$

Given the prior and the likelihood, we can derive the posterior:

$$p(\boldsymbol{q}|\mathcal{X}) \quad \propto \quad p(\mathcal{X}|\boldsymbol{q}) p(\boldsymbol{q}|\boldsymbol{\alpha})$$

$$\propto \prod_i q_i^{\alpha_i + N_i - 1} \tag{16.3}$$

where $N_i = \sum_{t=1}^{N} x_i^t$. We see that the posterior has the same form as the prior, and we call such a prior a *conjugate prior*. Both the prior and the likelihood have the form of product of powers of q_i, and we combine them to make up the posterior:

CONJUGATE PRIOR

$$\begin{aligned} p(\boldsymbol{q}|\mathcal{X}) &= \frac{\Gamma(\alpha_0 + N)}{\Gamma(\alpha_1 + N_1) \cdots \Gamma(\alpha_K + N_K)} \prod_{i=1}^{K} q_i^{\alpha_i + N_i - 1} \\ &= \text{Dirichlet}(\boldsymbol{q}|\boldsymbol{\alpha} + \boldsymbol{n}) \end{aligned} \tag{16.4}$$

where $\boldsymbol{n} = [N_1, \ldots, N_K]^T$ and $\sum_i N_i = N$.

Looking at equation 16.3, we can bring an interpretation to the hyperparameters α_i (Bishop 2006). Just as n_i are counts of occurrences of state i in a sample of N, we can view α_i as counts of occurences of state i in some imaginary sample of α_0 instances. In defining the prior, we are subjectively saying the following: In a sample of α_0, I would expect α_i of them to belong to state i. Note that larger α_0 implies that we have a higher confidence (a more peaked distribution) in our subjective proportions: Saying that I expect to have 60 out of 100 occurrences belong to state 1 has higher confidence than saying that I expect to have 6 out of 10. The posterior then is another Dirichlet that sums up the counts of the occurences of states, imagined and actual, given by the prior and the likelihood, respectively.

The conjugacy has a nice implication. In a sequential setting where we receive a sequence of instances, because the posterior and the prior have the same form, the current posterior accumulates information from all past instances and becomes the prior for the next instance.

16.2.2 $K = 2$ States: Beta Distribution

When the variable is binary, $x^t \in \{0, 1\}$, the multinomial sample becomes Bernoulli:

$$p(\mathcal{X}|q) = \prod_t q^{x^t} (1 - q)^{1 - x^t}$$

BETA DISTRIBUTION

and the Dirichlet prior reduces to the *beta distribution*:

$$\text{beta}(q|\alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} q^{\alpha - 1} (1 - q)^{\beta - 1}$$

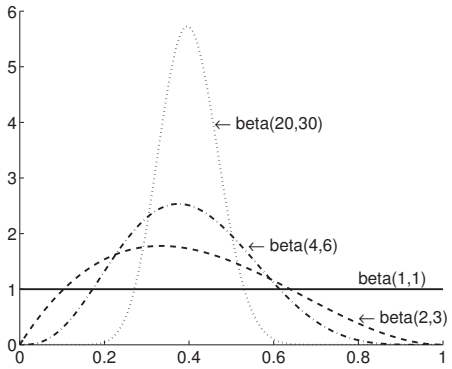


Figure 16.2 Plots of beta distributions for different sets of (α, β).

For example, x^t may be 0 or 1 depending on whether email with index t in a random sample of size N is legitimate or spam, respectively. Then defining a prior on q allows us to define a prior belief on the spam probability: I would expect, on the average, $\alpha/(\alpha+\beta)$ of my emails to be spam.

Beta is a conjugate prior, and for the posterior we get

$$p(q|A, N, \alpha, \beta) \propto q^{A+\alpha-1}(1-p)^{N-A+\beta-1}$$

where $A = \sum_t x^t$, and we see again that we combine the occurrences in the imaginary and the actual samples. Note that when $\alpha = \beta = 1$, we have a uniform prior and the posterior has the same shape as the likelihood. As the two counts, whether α and β for the prior or $\alpha+A$ and $\beta+N-A$ for the posterior, increase and their difference increases, we get a distribution that is more peaked with smaller variance (see figure 16.2). As we see more data (imagined or actual), the variance decreases.

16.3 Bayesian Estimation of the Parameters of a Gaussian Distribution

16.3.1 Univariate Case: Unknown Mean, Known Variance

We now consider the case where instances are Gaussian distributed. Let us start with the univariate case, $p(x) \sim \mathcal{N}(\mu, \sigma^2)$, where the parame-

ters are μ and σ^2; we discussed this briefly in section 4.4. The sample likelihood is

$$p(\mathcal{X}|\mu,\sigma^2) = \prod_t \frac{1}{\sqrt{2\pi}\sigma}\exp\left[-\frac{(x^t-\mu)^2}{2\sigma^2}\right] \tag{16.5}$$

The conjugate prior for μ is Gaussian, $p(\mu) \sim \mathcal{N}(\mu_0^2, \sigma_0^2)$, and we write the posterior as

$$\begin{aligned} p(\mu|\mathcal{X}) &\propto p(\mu)p(\mathcal{X}|\mu) \\ &\sim \mathcal{N}(\mu_N, \sigma_N^2) \end{aligned}$$

where

$$\mu_N = \frac{\sigma^2}{N\sigma_0^2+\sigma^2}\mu_0 + \frac{N\sigma_0^2}{N\sigma_0^2+\sigma^2}m \tag{16.6}$$

$$\frac{1}{\sigma_N^2} = \frac{1}{\sigma_0^2} + \frac{N}{\sigma^2} \tag{16.7}$$

where $m = \sum_t x^t/N$ is the sample average. We see that the mean of the posterior density (which is the MAP estimate), μ_N, is a weighted average of the prior mean μ_0 and the sample mean m, with weights being inversely proportional to their variances (see figure 16.3 for an example). Note that because both coefficients are between 0 and 1 and sum to 1, μ_N is always between μ_0 and m. When the sample size N or the variance of the prior σ_0^2 is large, the posterior mean is close to m, relying more on the information provided by the sample. When σ_0^2 is small—that is, when we have little prior uncertainty regarding the correct value of μ, or when we have a small sample—our prior guess μ_0 has higher effect.

σ_N gets smaller when either of σ_0 or σ gets smaller or if N is larger. Note also that σ_N is smaller than both σ_0 and $\sigma/\sqrt{N}$, that is, the posterior variance is smaller than both prior variance and that of m. Incorporating both results in a better posterior estimate than using any of the prior or sample alone.

If σ^2 is known, for new x, we can integrate over this posterior to make a prediction:

$$\begin{aligned} p(x|\mathcal{X}) &= \int p(x|\mu)p(\mu|\mathcal{X})d\mu \\ &\sim \mathcal{N}(\mu_N, \sigma_N^2+\sigma^2) \end{aligned} \tag{16.8}$$

We see that x is still Gaussian, that it is centered at the posterior mean, and that its variance now includes the uncertainty due to the estimation

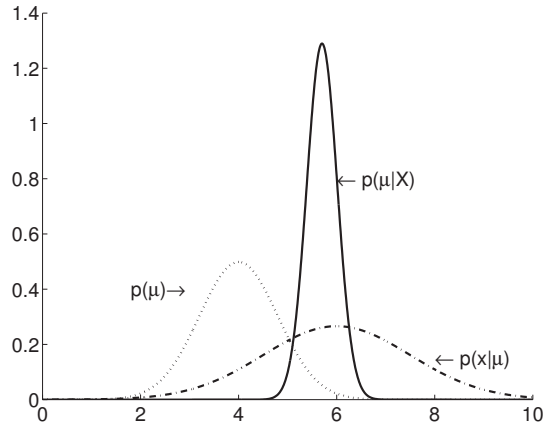


Figure 16.3 Twenty data points are drawn from $p(x) \sim \mathcal{N}(6, 1.5^2)$, prior is $p(\mu) \sim \mathcal{N}(4, 0.8^2)$, and posterior is then $p(\mu|\mathcal{X}) \sim \mathcal{N}(5.7, 0.3^2)$.

of the mean and the new sampled instance x. We can write $x = \mu + x'$ where $x' \sim \mathcal{N}(0, \sigma^2)$; then $E[x] = E[\mu] + E[x'] = \mu_N$ and $\text{Var}(x) = \text{Var}(\mu) + \text{Var}(x') = \sigma_N^2 + \sigma^2$, where this last follows from the fact that the new x' is an independent draw.

Once we get a distribution for $p(x|\mathcal{X})$, we can use it for different purposes. For example in classification, this approach corresponds to assuming Gaussian classes where means have a Gaussian prior and they are trained using $\mathcal{X}_i$, the subset of $\mathcal{X}$ labeled by class C_i. Then, $p(x|\mathcal{X}_i)$ as calculated above, corresponds to $p(x|C_i)$, which we combine with prior $P(C_i)$ to get the posterior and hence a discriminant.

16.3.2 Univariate Case: Unknown Mean, Unknown Variance

PRECISION

If we do not know σ^2, we also need to estimate it. For the case of variance, we work with the *precision*, the reciprocal of the variance, $\lambda \equiv 1/\sigma^2$. Using this, the sample likelihood is written as

$$\begin{aligned} p(\mathcal{X}|\lambda) &= \prod_t \frac{\lambda^{1/2}}{\sqrt{2\pi}} \exp\left[-\frac{\lambda}{2}(x^t - \mu)^2\right] \\ &= \lambda^{N/2}(2\pi)^{-N/2} \exp\left[-\frac{\lambda}{2}\sum_t (x^t - \mu)^2\right] \end{aligned} \tag{16.9}$$

GAMMA DISTRIBUTION

The conjugate prior for the precision is the *gamma distribution*:

$$p(\lambda) \sim \text{gamma}(a_0, b_0) = \frac{1}{\Gamma(a_0)} b_0^{a_0} \lambda^{a_0-1} \exp(-b_0\lambda)$$

where we define $a_0 \equiv \nu_0/2$ and $b_0 \equiv (\nu_0/2)s_0^2$ such that s_0^2 is our prior estimate of variance and ν_0 is our confidence in this prior—it may be thought of as the size of the imaginary sample on which we believe s_0^2 is estimated.

The posterior then is also gamma:

$$\begin{aligned} p(\lambda|\mathcal{X}) &\propto p(\mathcal{X}|\lambda)p(\lambda) \\ &\sim \text{gamma}(a_N, b_N) \end{aligned}$$

where

$$\begin{aligned} a_N &= a_0 + N/2 = \frac{\nu_0 + N}{2} \\ b_N &= b_0 + \frac{N}{2}s^2 = \frac{\nu_0}{2}s_0^2 + \frac{N}{2}s^2 \end{aligned} \tag{16.10}$$

where $s^2 = \sum_t (x^t - \mu)^2/N$ is the sample variance. Again, we see that posterior estimates are weighted sum of priors and sample statistics.

To make a prediction for new x, when both μ and σ^2 are unknown, we need the joint posterior that we write as

$$p(\mu, \lambda) = p(\mu|\lambda)p(\lambda)$$

where $p(\lambda) \sim \text{gamma}(a_0, b_0)$ and $p(\mu|\lambda) \sim \mathcal{N}(\mu_0, 1/(\kappa_0\lambda))$. Here again, κ_0 may be thought of as the size of the imaginary sample and as such it defines our confidence in the prior. The conjugate prior for the joint in this case is called the *normal-gamma distribution*

NORMAL-GAMMA DISTRIBUTION

$$\begin{aligned} p(\mu, \lambda) &\sim \text{normal-gamma}(\mu_0, \kappa_0, a_0, b_0) \\ &= \mathcal{N}(\mu, 1/(\kappa_0\lambda)) \cdot \text{gamma}(a_0, b_0) \end{aligned}$$

The posterior is

$$p(\mu, \lambda|\mathcal{X}) \sim \text{normal-gamma}(\mu_N, \kappa_N, a_N, b_N) \tag{16.11}$$

where

$$\begin{aligned} \kappa_N &= \kappa_0 + N \\ \mu_N &= \frac{\kappa_0\mu_0 + Nm}{\kappa_N} \\ a_N &= a_0 + N/2 \\ b_N &= b_0 + \frac{N}{2}s^2 + \frac{\kappa_0 N}{2\kappa_N}(m - \mu_0)^2 \end{aligned} \tag{16.12}$$

To make a prediction for new x, we integrate over the posterior:

$$p(x|\mathcal{X}) = \int\int p(x|\mu,\lambda)p(\mu,\lambda|\mathcal{X})d\mu d\lambda \tag{16.13}$$

$$\sim t_{2a_N}\left(\mu_N, \frac{b_N(\kappa_N+1)}{a_N\kappa_N}\right) \tag{16.14}$$

That is, we get a (nonstandardized) t distribution having the given mean and variance values with $2a_N$ degrees of freedom. In equation 16.8, we have a Gaussian distribution; here the mean is the same but because σ^2 is unknown, its estimation adds uncertainty, and we get a t distribution with wider tails. Sometimes, equivalently, instead of modeling the precision λ, we model σ^2 and for this, we can use the inverse gamma or the inverse chi-squared distribution; see Murphy 2007.

16.3.3 Multivariate Case: Unknown Mean, Unknown Covariance

If we have multivariate $\boldsymbol{x} \in \Re^d$, we use exactly the same approach, except for the fact that we need to use the multivariate versions of the distributions (Murphy 2012). We have

$$p(\boldsymbol{x}) \sim \mathcal{N}_d(\boldsymbol{\mu}, \boldsymbol{\Lambda})$$

PRECISION MATRIX

where $\boldsymbol{\Lambda} \equiv \boldsymbol{\Sigma}^{-1}$ is the *precision matrix*. We use a Gaussian prior (conditioned on $\boldsymbol{\Lambda}$) for the mean:

$$p(\boldsymbol{\mu}|\boldsymbol{\Lambda}) \sim \mathcal{N}_d(\boldsymbol{\mu}_0, (1/\kappa_0)\boldsymbol{\Lambda})$$

and for the precision matrix, the multivariate version of the gamma distribution is called the *Wishart distribution*:

WISHART DISTRIBUTION

$$p(\boldsymbol{\Lambda}) \sim \text{Wishart}(\nu_0, \mathbf{V}_0)$$

where ν_0, as with κ_0, corresponds to the strength of our prior belief.

NORMAL-WISHART DISTRIBUTION

The conjugate joint prior is the *normal-Wishart distribution*:

$$p(\mu,\boldsymbol{\Lambda}) = p(\boldsymbol{\mu}|\boldsymbol{\Lambda})p(\boldsymbol{\Lambda})$$

$$\sim \text{normal-Wishart}(\boldsymbol{\mu}_0, \kappa_0, \nu_0, \mathbf{V}_0) \tag{16.15}$$

and the posterior is

$$p(\mu,\boldsymbol{\Lambda}|\mathcal{X}) \sim \text{normal-Wishart}(\boldsymbol{\mu}_N, \kappa_N, \nu_N, \mathbf{V}_N)$$

where

$$
\begin{aligned}
\kappa_N &= \kappa_0 + N \\
\boldsymbol{\mu}_N &= \frac{\kappa_0 \boldsymbol{\mu}_0 + N\boldsymbol{m}}{\kappa_N} \\
\nu_N &= \nu_0 + N \\
\mathbf{V}_N &= \left(\mathbf{V}_0^{-1} + \mathbf{C} + \frac{\kappa_0 N}{\kappa_N}(\boldsymbol{m}-\boldsymbol{\mu}_0)(\boldsymbol{m}-\boldsymbol{\mu}_0)^T\right)^{-1}
\end{aligned} \tag{16.16}
$$

and $\mathbf{C} = \sum_t (\boldsymbol{x}^t - \boldsymbol{m})(\boldsymbol{x}^t - \boldsymbol{m})^T$ is the scatter matrix.

To make a prediction for new $\boldsymbol{x}$, we integrate over the joint posterior:

$$
p(\boldsymbol{x}|\mathcal{X}) = \int\int p(\boldsymbol{x}|\boldsymbol{\mu},\Lambda)p(\boldsymbol{\mu},\Lambda|\mathcal{X})d\boldsymbol{\mu}d\Lambda \tag{16.17}
$$

$$
\sim t_{\nu_N - d + 1}\left(\boldsymbol{\mu}_N, \frac{\kappa_N + 1}{\kappa_N(\nu_N - d + 1)}(\mathbf{V}_N)^{-1}\right) \tag{16.18}
$$

That is, we get a (nonstandardized) multivariate t distribution having this mean and covariance with $\nu_N - d + 1$ degrees of freedom.

16.4 Bayesian Estimation of the Parameters of a Function

We now discuss the case where we estimate the parameters, not of a distribution, but some function of the input, for regression or classification. Again, our approach is to consider these parameters as random variables with a prior distribution and use Bayes' rule to calculate a posterior distribution. We can then either evaluate the full integral, approximate it, or use the MAP estimate.

16.4.1 Regression

Let us take the case of a linear regression model:

$$
r = \boldsymbol{w}^T\boldsymbol{x} + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 1/\beta) \tag{16.19}
$$

where β is the precision of the additive noise (assume that one of the d inputs is always +1).

The parameters are the weights $\boldsymbol{w}$ and we have a sample $\mathcal{X} = \{\boldsymbol{x}^t, r^t\}_{t=1}^N$ where $\boldsymbol{x} \in \Re^d$ and $r^t \in \Re$. We can break it down into a matrix of inputs and a vector of desired outputs as $\mathcal{X} = [\mathbf{X}, \boldsymbol{r}]$ where $\mathbf{X}$ is $N \times d$ and $\boldsymbol{r}$ is $N \times 1$. From equation 16.19, we have

$$
p(r^t|\boldsymbol{x}^t, \boldsymbol{w}, \beta) \sim \mathcal{N}(\boldsymbol{w}^T\boldsymbol{x}, 1/\beta)
$$

We saw previously in section 4.6 that the log likelihood is

$$\begin{aligned}\mathcal{L}(\boldsymbol{w}|\mathcal{X}) \equiv \log p(\mathcal{X}|\boldsymbol{w}) &= \log p(\boldsymbol{r},\mathbf{X}|\boldsymbol{w}) \\ &= \log p(\boldsymbol{r}|\mathbf{X},\boldsymbol{w}) + \log p(\mathbf{X})\end{aligned}$$

where the second term is a constant, independent of the parameters. We expand the first term as

$$\begin{aligned}\log p(\boldsymbol{r}|\mathbf{X},\boldsymbol{w},\beta) &= \log \prod_t p(r^t|\boldsymbol{x}^t,\boldsymbol{w},\beta) \\ &= -N\log(\sqrt{2\pi}) + N\log\sqrt{\beta} - \frac{\beta}{2}\sum_t (r^t - \boldsymbol{w}^T\boldsymbol{x}^t)^2 \qquad (16.20)\end{aligned}$$

For the case of the ML estimate, we find $\boldsymbol{w}$ that maximizes this, or equivalently, minimizes the last term that is the sum of the squared error. It can be rewritten as

$$\begin{aligned}E &= \sum_{t=1}^{N}(r^t - \boldsymbol{w}^T\boldsymbol{x}^t)^2 = (\boldsymbol{r} - \mathbf{X}\boldsymbol{w})^T(\boldsymbol{r} - \mathbf{X}\boldsymbol{w}) \\ &= \boldsymbol{r}^T\boldsymbol{r} - 2\boldsymbol{w}^T\mathbf{X}^T\boldsymbol{r} + \boldsymbol{w}^T\mathbf{X}^T\mathbf{X}\boldsymbol{w}\end{aligned}$$

Taking the derivative with respect to $\boldsymbol{w}$ and setting it to 0

$$-2\mathbf{X}^T\boldsymbol{r} + 2\mathbf{X}^T\mathbf{X}\boldsymbol{w} = 0 \Rightarrow \mathbf{X}^T\mathbf{X}\boldsymbol{w} = \mathbf{X}^T\boldsymbol{r}$$

we get the maximum likelihood estimator (we have previously derived this in section 5.8):

$$\boldsymbol{w}_{ML} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\boldsymbol{r} \qquad (16.21)$$

Having calculated the parameters, we can now do prediction. Given new input $\boldsymbol{x}'$, the response is calculated as

$$r' = \boldsymbol{w}_{ML}^T\boldsymbol{x}' \qquad (16.22)$$

In the general case, for any model, $g(\boldsymbol{x}|\boldsymbol{w})$, for example, a multilayer perceptron where $\boldsymbol{w}$ are all the weights, we minimize, for example, using gradient descent:

$$E(\mathcal{X}|\boldsymbol{w}) = \left[r^t - g(x^t|\boldsymbol{w})\right]^2$$

and $\boldsymbol{w}_{LSQ}$ that minimizes it is called the *least squares estimator*. Then for new x', the prediction is calculated as

$$r' = g(\boldsymbol{x}'|\boldsymbol{w}_{LSQ})$$

In the case of the Bayesian approach, we define a Gaussian prior for the parameters:

$$p(\boldsymbol{w}) \sim \mathcal{N}(\mathbf{0}, (1/\alpha)\mathbf{I})$$

which is a conjugate prior, and for the posterior, we get

$$p(\boldsymbol{w}|\mathcal{X}, \boldsymbol{r}) \sim \mathcal{N}(\boldsymbol{\mu}_N, \boldsymbol{\Sigma}_N)$$

where

$$\begin{aligned} \boldsymbol{\mu}_N &= \beta \boldsymbol{\Sigma}_N \mathbf{X}^T \boldsymbol{r} \\ \boldsymbol{\Sigma}_N &= (\alpha \mathbf{I} + \beta \mathbf{X}^T \mathbf{X})^{-1} \end{aligned} \tag{16.23}$$

To calculate the output for new $\boldsymbol{x}'$, we integrate over the full posterior

$$r' = \int (\boldsymbol{w}^T \boldsymbol{x}') p(\boldsymbol{w}|\mathcal{X}, \boldsymbol{r}) d\boldsymbol{w}$$

The graphical model for this is shown in figure 14.7.

If we want to use a point estimate, the MAP estimator is

$$\boldsymbol{w}_{MAP} = \boldsymbol{\mu}_N = \beta(\alpha \mathbf{I} + \beta \mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \boldsymbol{r} \tag{16.24}$$

and in calculating the output for input $\boldsymbol{x}'$, we replace the density with a single point, namely, the mean:

$$r' = \boldsymbol{w}_{MAP}^T \boldsymbol{x}'$$

We can also calculate the variance of our estimate:

$$\text{Var}(r') = 1/\beta + (\boldsymbol{x}')^T \boldsymbol{\Sigma}_N \boldsymbol{x}' \tag{16.25}$$

Comparing equation 16.24 with the ML estimate of equation 16.21, this can be seen as regularization—that is, we add a constant α to the diagonal to better condition the matrix to be inverted.

The prior, $p(\boldsymbol{w}) \sim \mathcal{N}(\mathbf{0}, (1/\alpha)\mathbf{I})$, says that we expect the parameters to be close to 0 with spread inversely proportional to α. When $\alpha \to 0$, we have a flat prior and the MAP estimate converges to the ML estimate.

We see in figure 16.4 that if we increase α, we force parameters to be closer to 0 and the posterior distribution moves closer to the origin and shrinks. If we decrease β, we assume noise with higher variance and the posterior also has higher variance.

If we take the log of the posterior, we have

$$\begin{aligned} \log p(\boldsymbol{w}|\mathbf{X}, \boldsymbol{r}) &\propto \log p(\boldsymbol{r}|\boldsymbol{w}, \mathbf{X}) + \log p(\boldsymbol{w}) \\ &= -\frac{\beta}{2} \sum_t (r^t - \boldsymbol{w}^T \boldsymbol{x}^t)^2 - \frac{\alpha}{2} \boldsymbol{w}^T \boldsymbol{w} + c \end{aligned}$$

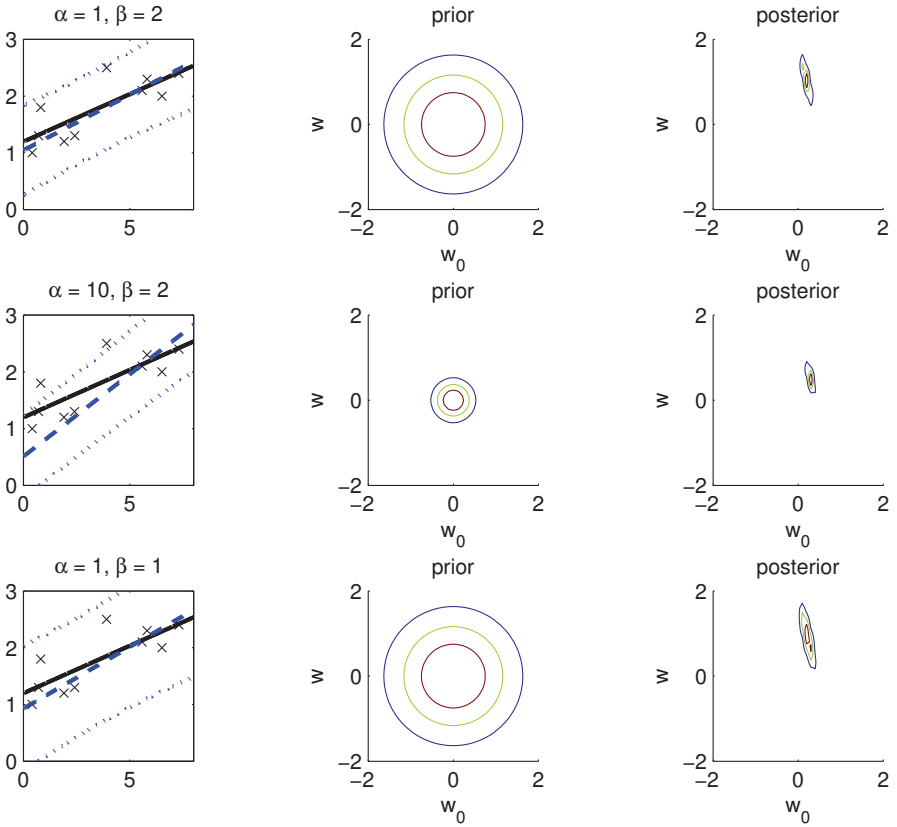


Figure 16.4 Bayesian linear regression for different values of α and β. To the left: crosses that are the data points and the straight line that is the ML solution. The MAP solution with one standard deviation error bars are also shown dashed. Center: prior density centered at 0 and variance $1/\alpha$. To the right: posterior density whose mean is the MAP solution. We see that when α is increased, the variance of the prior shrinks and the line moves closer to the flat 0 line. When β is decreased, more noise is assumed and the posterior density has higher variance.

which we maximize to find the MAP estimate. In the general case, given our model $g(\boldsymbol{x}|\boldsymbol{w})$, we can write an augmented error function

$$E_{ridge}(\boldsymbol{w}|\mathcal{X}) = \sum_t [r^t - g(\boldsymbol{x}^t|\boldsymbol{w})]^2 + \lambda \sum_i w_i^2$$

RIDGE REGRESSION

with $\lambda \equiv \alpha/\beta$. This is known as *parameter shrinkage* or *ridge regression* in statistics. In section 4.8, we called this *regularization*, and in section 11.9, we called this *weight decay* in neural networks. The first term is the negative log of the likelihood, and the second term penalizes w_i away from 0 (as dictated by α of the prior).

Though this approach reduces $\sum_i w_i^2$, it does not force individual w_i to 0; that is, it cannot be used for feature selection, namely, to determine which x_i are redundant. For this, one can use a *Laplacian prior* that uses the L_1 norm instead of the L_2 norm (Figueiredo 2003):

LAPLACIAN PRIOR

$$p(\boldsymbol{w}|\alpha) = \prod_i \frac{\alpha}{2} \exp(-\alpha|w_i|) = \left(\frac{\alpha}{2}\right)^d \exp\left(-\alpha \sum_i |w_i|\right)$$

The posterior probability is no longer Gaussian and the MAP estimate is found by minimizing

$$E_{lasso}(\boldsymbol{w}|\mathcal{X}) = \sum_t (r^t - \boldsymbol{w}^T\boldsymbol{x}^t)^2 + 2\sigma^2\alpha \sum_i |w_i|$$

where σ^2 is the variance of noise (for which we plug in our estimate). This is known as *lasso* (least absolute shrinkage and selection operator) (Tibshirani 1996). To see why L_1 induces sparseness, let us consider the case with two weights $[w_1, w_2]^T$ (Figueiredo 2003): $\|[1,0^T\|_2 = \|[1/\sqrt{2}, 1/\sqrt{2}]^T\|_2 = 1$, whereas $\|[1,0]^T\|_1 = 1 < \|[1/\sqrt{2}, 1/\sqrt{2}]^T\|_1 = \sqrt{2}$, and therefore L_1 prefers to set w_2 to 0 and use a large w_1, rather than having small values for both.

LASSO

16.4.2 Regression with Prior on Noise Precision

Above, we assume that β, the precision of noise, is known and $\boldsymbol{w}$ is the only parameter we integrated on. If we do not know β, we can also define a prior on it. Just as we do in section 16.3, we can define a gamma prior:

$$p(\beta) \sim \text{gamma}(a_0, b_0)$$

and a prior on $\boldsymbol{w}$ conditioned on β:

$$p(\boldsymbol{w}|\beta) \sim \mathcal{N}(\boldsymbol{\mu}_0, \beta\boldsymbol{\Sigma}_0)$$

If $\boldsymbol{\mu}_0 = \mathbf{0}$ and $\boldsymbol{\Sigma}_0 = \alpha\mathbf{I}$, we get ridge regression, as we discussed above. We now can write a conjugate normal-gamma prior on parameters $\boldsymbol{w}$ and β:

$$p(\boldsymbol{w}, \beta) = p(\beta)p(\boldsymbol{w}|\beta) \sim \text{normal-gamma}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0, a_0, b_0)$$

It can be shown (Hoff 2009) that the posterior is

$$p(\boldsymbol{w}, \beta|\mathbf{X}, \boldsymbol{r}) \sim \text{normal-gamma}(\boldsymbol{\mu}_N, \boldsymbol{\Sigma}_N, a_N, b_N)$$

where

$$\begin{aligned} \boldsymbol{\Sigma}_N &= (\mathbf{X}^T\mathbf{X} + \boldsymbol{\Sigma}_0)^{-1} \\ \boldsymbol{\mu}_N &= \boldsymbol{\Sigma}_N(\mathbf{X}^T\boldsymbol{r} + \boldsymbol{\Sigma}_0\boldsymbol{\mu}_0) \\ a_N &= a_0 + N/2 \\ b_N &= b_0 + \frac{1}{2}(\boldsymbol{r}^T\boldsymbol{r} + \boldsymbol{\mu}_0^T\boldsymbol{\Sigma}_0\boldsymbol{\mu}_0 - \boldsymbol{\mu}_N^T\boldsymbol{\Sigma}_N\boldsymbol{\mu}_N) \end{aligned} \tag{16.26}$$

An example is given in figure 16.5 where we fit a polynomial of different degrees on a small set of instances—$\boldsymbol{w}$ corresponds to the vector of coefficients of the polynomial. We see that the maximum likelihood starts to overfit as the degree is increased.

MARKOV CHAIN MONTE CARLO SAMPLING

We use *Markov chain Monte Carlo sampling* to get the Bayesian fit as follows: We draw a β value from $p(\beta) \sim \text{gamma}(a_N, b_N)$, and then we draw a $\boldsymbol{w}$ from $p(\boldsymbol{w}|\beta) \sim \mathcal{N}(\boldsymbol{\mu}_N, \beta\boldsymbol{\Sigma}_N)$, which gives us one sampled model from the posterior $p(\boldsymbol{w}, \beta)$. Ten such samples are drawn for each degree, as shown in figure 16.5. The thick line is the average of those ten models and is an approximation of the full integral; we see that even with ten samples, we get a reasonable and very smooth fit to the data. Note that any of the sampled models from the posterior is not necessarily any better than the maximum likelihood estimator; it is the averaging that leads to a smoother and hence better fit.

16.4.3 The Use of Basis/Kernel Functions

Using the Bayes' estimate of equation 16.23, the prediction is written as

$$\begin{aligned} r' &= (\boldsymbol{x}')^T\boldsymbol{w} \\ &= \beta(\boldsymbol{x}')^T\boldsymbol{\Sigma}_N\mathbf{X}^T\boldsymbol{r} \\ &= \sum_t \beta(\boldsymbol{x}')^T\boldsymbol{\Sigma}_N\boldsymbol{x}^t r^t \end{aligned}$$

DUAL REPRESENTATION

This is the *dual representation*. When we can write the parameter in

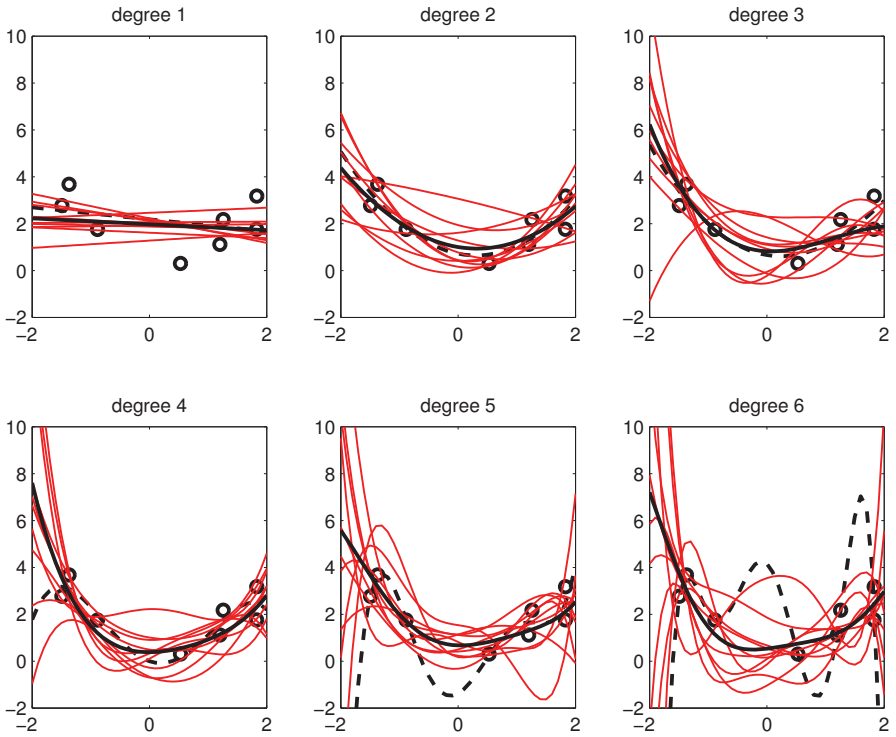


Figure 16.5 Bayesian polynomial regression example. Circles are the data points and the dashed line is the maximum likelihood fit, which overfits as the degree of the polynomial is increased. Thin lines are ten samples from the posterior $p(\boldsymbol{w}, \beta)$ and the thick line is their average.

terms of the training data, or a subset of it as in support vector machines (chapter 13), we can write the prediction as a function of the current input and past data. We can rewrite this as

$$r' = \sum_t K(\boldsymbol{x}', \boldsymbol{x}^t) r^t \tag{16.27}$$

where we define

$$K(\boldsymbol{x}', \boldsymbol{x}^t) = \beta(\boldsymbol{x}')^T \boldsymbol{\Sigma}_N \boldsymbol{x}^t \tag{16.28}$$

We know that we can generalize the linear kernel of equation 16.28 by using a nonlinear *basis function* $\phi(\boldsymbol{x})$ to map to a new space where we

BASIS FUNCTION

fit the linear model. In such a case, instead of the d-dimensional $\boldsymbol{x}$ we have the k-dimensional $\boldsymbol{\phi}(\boldsymbol{x})$ where k is the number of basis functions and instead of $N \times d$ data matrix $\mathbf{X}$, we have $N \times k$ image of the basis functions $\boldsymbol{\Phi}$.

During test, we have

$$\begin{aligned} r' &= \boldsymbol{\phi}(\boldsymbol{x}')^T \boldsymbol{w} \text{ where } \boldsymbol{w} = \beta \boldsymbol{\Sigma}_N^{\phi} \boldsymbol{\Phi}^T \boldsymbol{r} \text{ and } \boldsymbol{\Sigma}_N^{\phi} = \left(\alpha \mathbf{I} + \beta \boldsymbol{\Phi}^T \boldsymbol{\Phi}\right)^{-1} \\ &= \beta \boldsymbol{\phi}(\boldsymbol{x}')^T \boldsymbol{\Sigma}_N^{\phi} \boldsymbol{\Phi}^T \boldsymbol{r} \\ &= \sum_t \beta \boldsymbol{\phi}(\boldsymbol{x}')^T \boldsymbol{\Sigma}_N^{\phi} \boldsymbol{\phi}(\boldsymbol{x}^t) r^t \\ &= \sum_t K(\boldsymbol{x}', \boldsymbol{x}^t) r^t \end{aligned} \tag{16.29}$$

where we define

$$K(\boldsymbol{x}', \boldsymbol{x}^t) = \beta \boldsymbol{\phi}(\boldsymbol{x}')^T \boldsymbol{\Sigma}_N^{\phi} \boldsymbol{\phi}(\boldsymbol{x}^t) \tag{16.30}$$

as the equivalent kernel. This is the dual representation in the space of $\boldsymbol{\phi}(\boldsymbol{x})$. We see that we can write our estimate as a weighted sum of the effects of instances in the training set where the effect is given by the *kernel function* $K(\boldsymbol{x}', \boldsymbol{x}^t)$; this is similar to the nonparametric kernel smoothers we discussed in chapter 8, or the kernel machines of chapter 13.

KERNEL FUNCTION

Error bars can be defined using

$$\text{Var}(r') = \beta^{-1} + \boldsymbol{\phi}(\boldsymbol{x}')^T \boldsymbol{\Sigma}_N^{\phi} \boldsymbol{\phi}(\boldsymbol{x}')$$

An example is given in figure 16.6 for the linear, quadratic, and sixth-degree kernels. This is equivalent to the polynomial regression we see in figure 16.5, except that here we use the dual representation and the polynomial coefficients $\boldsymbol{w}$ are embedded in the kernel function. We see that just as in regression proper where we can work on the original $\boldsymbol{x}$ or $\boldsymbol{\phi}(\boldsymbol{x})$, in Bayesian regression too we can work on the preprocessed $\boldsymbol{\phi}(\boldsymbol{x})$, defining parameters in that space. Later on in this chapter, we are going to see Gaussian processes where we can define and use $K(\boldsymbol{x}, \boldsymbol{x}^t)$ directly without needing to calculate $\boldsymbol{\phi}(\boldsymbol{x})$.

16.4.4 Bayesian Classification

In a two-class problem, we have a single output, and assuming a linear model, we have

$$P(C_1 | \boldsymbol{x}^t) = y^t = \text{sigmoid}(\boldsymbol{w}^T \boldsymbol{x}^t)$$

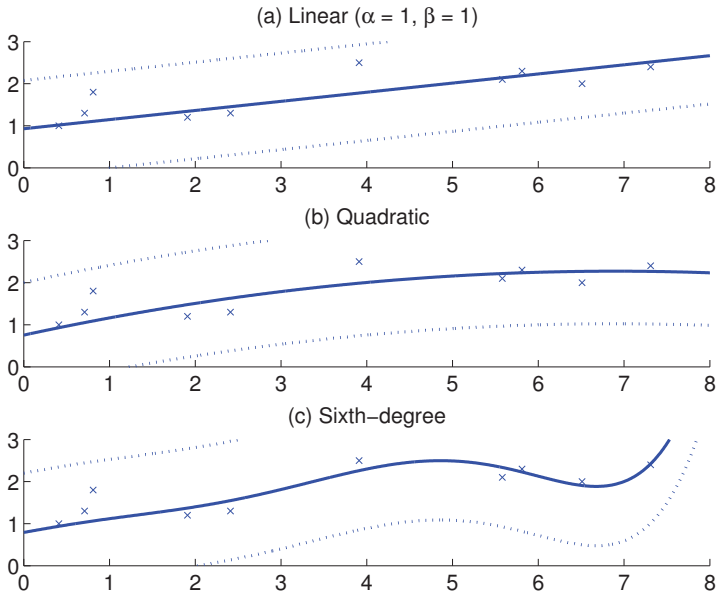


Figure 16.6 Bayesian regression using kernels with one standard deviation error bars: (a) linear: $\boldsymbol{\phi}(x) = [1, x]^T$, (b) quadratic: $\boldsymbol{\phi}(x) = [1, x, x^2]^T$, and (c) sixth degree: $\boldsymbol{\phi}(x) = [1, x, x^2, x^3, x^4, x^5, x^6]^T$.

The log likelihood of a Bernoulli sample is given as

$$\mathcal{L}(\boldsymbol{r}|\mathbf{X}) = \sum_t r^t \log y_t + (1 - r^t) \log(1 - y^t)$$

which we maximize, or minimize its negative log—the cross-entropy—to find the ML estimate, for example, using gradient descent. This is called *logistic discrimination* (section 10.7).

In the case of the Bayesian approach, we assume a Gaussian prior

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{m}_0, \mathbf{S}_0) \tag{16.31}$$

and the log of the posterior is given as

$$\begin{aligned} \log p(\boldsymbol{w}|\boldsymbol{r}, \mathbf{X}) &\propto \log p(\boldsymbol{w}) + \log p(\boldsymbol{r}|\boldsymbol{w}, \mathbf{X}) \\ &= -\frac{1}{2}(\boldsymbol{w} - \boldsymbol{m}_0)^T \mathbf{S}_0^{-1}(\boldsymbol{w} - \boldsymbol{m}_0) \\ &\quad + \sum_t r^t \log y_t + (1 - r^t) \log(1 - y^t) + c \end{aligned} \tag{16.32}$$

LAPLACE APPROXIMATION

This posterior distribution is no longer Gaussian, and we cannot integrate exactly. We can use *Laplace approximation*, which works as follows (MacKay 2003). Let us say we want to approximate some distribution $f(x)$, not necessarily normalized (to integrate to 1). In Laplace approximation, we find the mode of $f(x)$, x_0, fit a Gaussian $q(x)$ centered there with covariance given by the curvature of $f(x)$ around that mean, and then if we want to integrate, we integrate this fitted Gaussian instead.

To find the variance of the Gaussian, we consider the Taylor expansion of $f(\cdot)$ at $x = x_0$

$$\log f(x) = \log f(x_0) - \frac{1}{2}a(x - x_0)^2 + \cdots$$

where

$$a \equiv - \frac{d}{dx^2} \log f(x) \Big|_{x=x_0}$$

Note that the first, linear term disappears because the first derivative is 0 at the mode. Taking exp, we have

$$f(x) = f(x_0) \exp\left[-\frac{a}{2}(x - x_0)^2\right]$$

To normalize $f(x)$, we consider that in a Gaussian distribution

$$\int \frac{1}{\sqrt{2\pi}(1/\sqrt{a})} \exp\left[-\frac{a}{2}(x - x_0)^2\right] = 1 \Rightarrow \int \exp\left[-\frac{a}{2}(x - x_0)^2\right] = \sqrt{a/2\pi}$$

and therefore

$$q(x) = \sqrt{a/2\pi} \exp\left[-\frac{a}{2}(x - x_0)^2\right] \sim \mathcal{N}(x_0, 1/a)$$

In the multivariate setting where $\boldsymbol{x} \in \Re^d$, we have

$$\log f(\boldsymbol{x}) = \log f(\boldsymbol{x}_0) - \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_0)^T \mathbf{A}(\boldsymbol{x} - \boldsymbol{x}_0) + \cdots$$

where $\mathbf{A}$ is the (Hessian) matrix of second derivatives:

$$\mathbf{A} = - \nabla\nabla \log f(\boldsymbol{x}) \big|_{\boldsymbol{x}=\boldsymbol{x}_0}$$

The Laplace approximation is then

$$f(\boldsymbol{x}) = \frac{|\mathbf{A}|^{1/2}}{(2\pi)^{d/2}} \exp\left[-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_0)^T \mathbf{A}(\boldsymbol{x} - \boldsymbol{x}_0)\right] \sim \mathcal{N}_d(\boldsymbol{x}_0, \mathbf{A}^{-1})$$

Having now discussed how to approximate, we can now use it for the posterior density. The MAP estimate, $\boldsymbol{w}_{MAP}$—the mode of $p(\boldsymbol{w}|\boldsymbol{r}, \mathbf{X})$—is

taken as the mean, and the covariance matrix is given by the inverse of the matrix of the second derivatives of the negative log likelihood:

$$\mathbf{S}_N = -\nabla\nabla \log p(\boldsymbol{w}|\boldsymbol{r},\mathbf{X}) = \mathbf{S}_0^{-1} + \sum_t y^t(1-y^t)\boldsymbol{x}^t(\boldsymbol{x}^t)^T$$

We then integrate over this Gaussian to estimate the class probability:

$$P(C_1|\boldsymbol{x}) = y = \int \text{sigmoid}(\boldsymbol{w}^T\boldsymbol{x})q(\boldsymbol{w})d\boldsymbol{w}$$

where $q(\boldsymbol{w}) \sim \mathcal{N}(\boldsymbol{w}_{MAP}, \mathbf{S}_N^{-1})$. A further complication is that we cannot integrate analytically over a Gaussian convolved with a sigmoid. If we use the *probit function* instead, which has the same S-shape as the sigmoid, an analytical solution is possible (Bishop 2006).

PROBIT FUNCTION

16.5 Choosing a Prior

Defining the prior is the subjective part of Bayesian estimation and as such should be done with care. It is best to define robust priors with heavy tails so as not to limit the parameter space too much; in the extreme case of no prior preference, one can use an uninformative prior and methods have been proposed for this purpose, for example, Jeffreys prior (Murphy 2012). Sometimes our choice of a prior is also motivated by simplicity—for example, a conjugate prior makes inference quite easy.

One critical decision is when to take a parameter as a constant and when to define it as a random variable with a prior and to be integrated (averaged) out. For example, in section 16.4.1, we assume that we know the noise precision whereas in section 16.4.2, we assume we do not and define a gamma prior on it. Similarly for the spread of weights in linear regression, we assume a constant α value but can also define a prior on it and average it out if we want. Of course, this makes the prior more complicated and the whole inference more difficult but averaging over α should be preferred if we do not know what the good value for α is.

Another decision is how high to go in defining the priors. Let us say we have parameter θ and we define a posterior on it. In prediction, we have

$$\text{Level I: } p(x|\mathcal{X}) = \int p(x|\theta)p(\theta|\mathcal{X})d\theta$$

where $p(\theta|\mathcal{X}) \propto p(\mathcal{X}|\theta)p(\theta)$. If we believe that we cannot define a good

$p(\theta)$ but that it depends on some other variable, we can condition θ on a hyper parameter α and integrate it out:

$$\text{Level II: } p(x|\mathcal{X}) = \int p(x|\theta)p(\theta|\mathcal{X},\alpha)p(\alpha)d\theta d\alpha$$

This is called a *hierarchical prior*. This really makes the inference rather difficult because we need to integrate on two levels. One shortcut is to test different values α on the data, choose the best α^*, and just use that value:

$$\text{Level II ML: } p(x|\mathcal{X}) = \int p(x|\theta)p(\theta|\mathcal{X},\alpha^*)d\theta$$

This is called *level II maximum likelihood* or *empirical Bayes*.

16.6 Bayesian Model Comparison

Assume we have many models $\mathcal{M}_j$, each with its own set of parameters θ_j, and we want to compare these models. For example, in figure 16.5, we have polynomials of different degrees and let us say we want to check how well they fit the data.

For a given model $\mathcal{M}$ and parameter θ, the likelihood of data is $p(\mathcal{X}|\mathcal{M},\theta)$.

MARGINAL LIKELIHOOD

To get the Bayesian *marginal likelihood* for a given model, we average over θ:

$$p(\mathcal{X}|\mathcal{M}) = \int p(\mathcal{X}|\theta,\mathcal{M})p(\theta|\mathcal{M})d\theta \tag{16.33}$$

MODEL EVIDENCE

This is also called *model evidence*. For example, in the polynomial regression example above, for a given degree, we have

$$p(\boldsymbol{r}|\mathbf{X},\mathcal{M}) = \int\int p(\boldsymbol{r}|\mathbf{X},\boldsymbol{w},\beta,\mathcal{M})p(\boldsymbol{w},\beta|\mathcal{M})d\boldsymbol{w}d\beta$$

where $p(\boldsymbol{w},\beta|\mathcal{M})$ is the prior assumed for model $\mathcal{M}$. We can then calculate the posterior probability of a model given the data:

$$p(\mathcal{M}|\mathcal{X}) = \frac{p(\mathcal{X}|\mathcal{M})p(\mathcal{M})}{p(\mathcal{X})} \tag{16.34}$$

where $P(\mathcal{M})$ is the prior distribution defined over models. The nice property of the Bayesian approach is that even if those priors are taken uniform, the marginal likelihood, because it averages over all θ, favors simpler models. Let us assume we have models in increasing complexity, for example, polynomials with increasing degree.

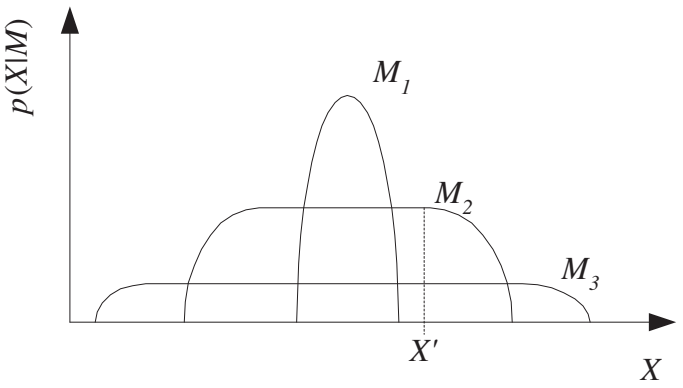


Figure 16.7 Bayesian model comparison favors simpler models. $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ are three models in increasing complexity. The x axis is the space of all datasets with N instances. A complex model can fit more datasets but spreads itself thin over the space of all possible datasets of size N; a simpler model can fit fewer datasets but each with a heavier probability. For a particular dataset $\mathcal{X}'$, if both can fit, the simpler model will have higher marginal likelihood (MacKay 2003).

Let us say we have a dataset $\mathcal{X}$ with N instances. A more complex model will be able to fit more of such datasets reasonably well compared with a simpler model—consider choosing randomly three points in a plane; the number of such triples that can be fitted by a line is much fewer than the number of triples that can be fitted by a quadratic. Given that $\sum_{\mathcal{X}} p(\mathcal{X}|\mathcal{M}) = 1$, because for a complex model there are more possible $\mathcal{X}$ where it can make a reasonable fit, if there is a fit, the value of $p(\mathcal{X}'|\mathcal{M})$ for some particular $\mathcal{X}'$ is going to be smaller—see figure 16.7. Hence for a simpler model $p(\mathcal{M}|\mathcal{X})$ will be higher (even if we assume that the priors, $p(\mathcal{M})$, are equal); this is the Bayesian interpretation of Occam's razor (MacKay 2003).

For the polynomial fitting example of figure 16.5, a comparison of likelihood and the marginal likelihood is shown in figure 16.8. We see that likelihood increases when complexity increases, which implies overfitting, but the marginal likelihood increases until the correct degree and then starts decreasing; this is because there are many more complex models that fit badly to the data and they pull the likelihood down as

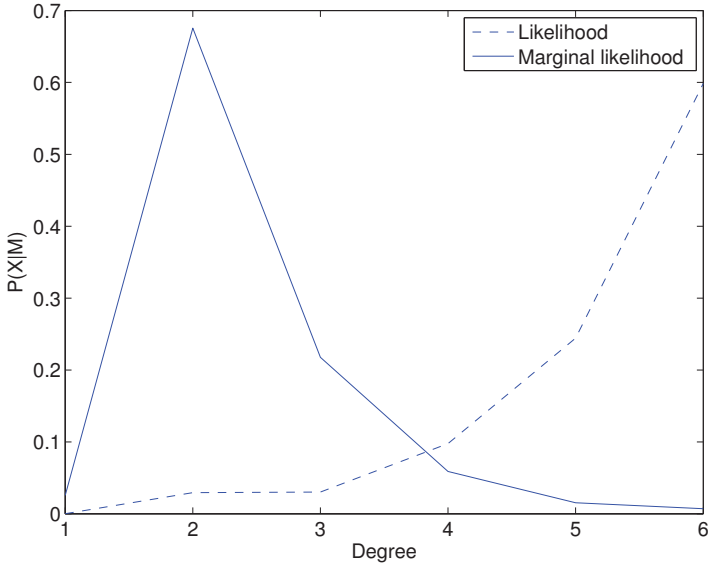


Figure 16.8 Likelihood versus marginal likelihood for the polynomial regression example. Though the likelihood increases as the degree of the polynomial increases, the marginal likelihood that averages over parameter values make a peak at the right complexity and then levels off.

we average over them.

If we have two models $\mathcal{M}_0$ and $\mathcal{M}_1$, we can compare them

$$\frac{P(\mathcal{M}_1|\mathcal{X})}{P(\mathcal{M}_0|\mathcal{X})} = \frac{P(\mathcal{X}|\mathcal{M}_1)}{P(\mathcal{X}|\mathcal{M}_0)} \frac{P(\mathcal{M}_1)}{P(\mathcal{M}_0)}$$

and we have higher belief in $\mathcal{M}_1$ if this ratio is higher than 1, and in $\mathcal{M}_0$ otherwise.

BAYES FACTOR

There are two important points here: One, the ratio of the two marginal likelihoods is called the *Bayes factor* and is enough for model selection even if the two priors are taken equal. Second, in the Bayesian approach, we do not choose among models and we do not do model selection; but in keeping with the spirit of the Bayesian approach, we average over their predictions rather than choosing one and discarding the rest. For instance, in the polynomial regression example above, rather than choosing one degree, it is best to take a weighted average over all degrees weighted by their marginal likelihoods.

BAYESIAN INFORMATION CRITERION

A related approach is the *Bayesian information criterion* (BIC) where using Laplace approximation (section 16.4.4), equation 16.33 is approximated as

$$\log p(\mathcal{X}|\mathcal{M}) \approx \text{BIC} \equiv \log p(\mathcal{X}|\theta_{ML}, \mathcal{M}) - \frac{|\mathcal{M}|}{2} \log N \tag{16.35}$$

The first term is the likelihood using the ML estimator and the second term is a penalty for complex models: $|\mathcal{M}|$ is a measure of model complexity, in other words, the degrees of freedom in the model—for example, the number of coefficients in a linear regression model. As model complexity increases, the first term may be higher but the second penalty term compensates for this.

AKAIKE'S INFORMATION CRITERION

A related, but not Bayesian, approach is *Akaike's information criterion* (AIC), which is written as

$$\text{AIC} \equiv \log p(\mathcal{X}|\theta_{ML}, \mathcal{M}) - |\mathcal{M}| \tag{16.36}$$

where again we see a penalty term that is proportional to the model complexity. It is important to note here that in such criteria, $|\mathcal{M}|$ represents the "effective" degrees of freedom and not simply the number of adjustable parameters in the model. For example in a multilayer perceptron (chapter 11), the effective degrees of freedom is much less than the number of adjustable connection weights.

One interpretation of the penalty term is as a term of "optimism" (Hastie, Tibshirani, and Friedman 2011). In a complex model, the ML estimator would overfit and hence be a very optimistic indicator of model performance; therefore, it should be cut back proportional to the model complexity.

16.7 Bayesian Estimation of a Mixture Model

In section 7.2, we discuss the mixture model where we write the density as a weighted sum of component densities. Let us remember equation 7.1:

$$p(\boldsymbol{x}) = \sum_{i=1}^{k} P(\mathcal{G}_i) p(\boldsymbol{x}|\mathcal{G}_i)$$

where $P(\mathcal{G}_i)$ are the mixture proportions and $p(\boldsymbol{x}|\mathcal{G}_i)$ are the component densities. For example, in Gaussian mixtures, we have $p(\boldsymbol{x}|\mathcal{G}_i) \sim$

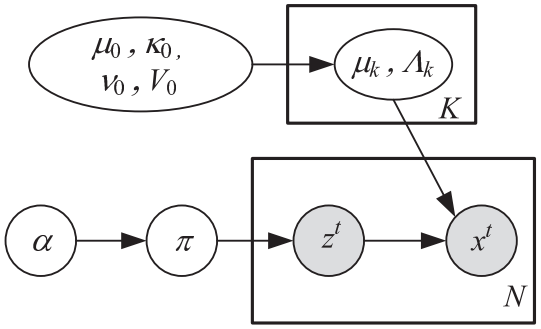


Figure 16.9 The generative graphical representation of a Gaussian mixture model.

$\mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)$, and defining $\pi_i \equiv P(\mathcal{G}_i)$, we have the parameter vector $\Phi = \{\pi_i, \boldsymbol{\mu}_i, \Sigma_i\}_{i=1}^k$ that we need to learn from data $\mathcal{X} = \{\boldsymbol{x}^t\}_{t=1}^N$.

In section 7.4, we discussed the EM algorithm that is a maximum likelihood procedure:

$$\Phi_{MLE} = \arg\max_{\Phi} \log p(\mathcal{X}|\Phi)$$

If we have a prior distribution $p(\Phi)$, we can devise a Bayesian approach. For example, the MAP estimator is

$$\Phi_{MAP} = \arg\max_{\Phi} \log p(\Phi|\mathcal{X}) = \arg\max_{\Phi} \log p(\mathcal{X}|\Phi) + \log p(\Phi) \tag{16.37}$$

Let us now write down the prior. Π_i are multinomial variables and for them, we can use a Dirichlet prior as we discuss in section 16.2.1. For the Gaussian components, for the mean and precision (inverse covariance) matrix, we can use a normal-Wishart prior as we discuss in section 16.3:

$$\begin{aligned} p(\Phi) &= p(\boldsymbol{\pi}) \prod_i p(\boldsymbol{\mu}_i, \boldsymbol{\Lambda}_i) \\ &= \text{Dirichlet}(\boldsymbol{\pi}|\boldsymbol{\alpha}) \prod_i \text{normal-Wishart}(\boldsymbol{\mu}_0, \kappa_0, \nu_0, \mathbf{V}_0) \end{aligned} \tag{16.38}$$

So in using EM in this case, the E-step does not change, but in the M-step we maximize the posterior with this prior (Murphy 2012). Adding log of the posterior, equation 7.10 becomes

$$\begin{aligned} \mathcal{Q}(\Phi|\Phi^l) &= \sum_t \sum_i h_i^t \log \pi_i + \sum_t \sum_i h_i^t \log p_i(\boldsymbol{x}^t|\Phi^l) + \log p(\boldsymbol{\pi}) + \\ &\quad \sum_i \log p(\boldsymbol{\mu}_i, \boldsymbol{\Lambda}_i) \end{aligned} \tag{16.39}$$

where $h_i^t \equiv E[z_i^t]$ are the soft labels estimated in the E-step using the current values of Φ. The M-step MAP estimate for the mixture proportions are as follows (based on equation 16.4):

$$\pi_i^{l+1} = \frac{\alpha_i + N_i - 1}{\sum_i \alpha_i + N - k} \tag{16.40}$$

where $N_i = \sum_i h_i^t$. The M-step MAP estimates for the Gaussian component density parameters are as follows (based on equation 16.16):

$$\begin{aligned} \boldsymbol{\mu}_i^{l+1} &= \frac{\kappa_0 \boldsymbol{\mu}_0 + N_i \boldsymbol{m}_i}{\kappa_0 + N_i} \\ \boldsymbol{\Lambda}_i^{l+1} &= \left(\frac{\mathbf{V}_0^{-1} + \mathbf{C}_i + \mathbf{S}_i}{\nu_0 + N_i + d + 2} \right)^{-1} \end{aligned} \tag{16.41}$$

where $\boldsymbol{m}_i = \sum_t h_i^t / N_i$ is the component mean, $\mathbf{C}_i = \sum_t h_i^t (\boldsymbol{x}^t - \boldsymbol{m}_i)(\boldsymbol{x}^t - \boldsymbol{m}_i)^T$ is the within-scatter matrix for component i, and $\mathbf{S}_i = (\kappa_0 N_i)/(\kappa_0 + N_i)(\boldsymbol{m}_i - \boldsymbol{\mu}_0)(\boldsymbol{m}_i - \boldsymbol{\mu}_0)^T$ is the between-scatter of component i around the prior mean.

If we take $\alpha_i = 1/K$, this is a uniform prior. We can take $\kappa_0 = 0$ not to bias the mean estimates unless we do have some prior information about them. We can take $\mathbf{V}_0$ as the identity matrix and hence the MAP estimate has a regularizing effect.

The mixture density is shown as a generative graphical model in figure 16.9.

Once we know how to do basic blocks in a Bayesian manner, we can combine them to get more complicated models. For example, combining the mixture model we have here and the linear regression model we discuss in section 16.4.1, we can write the Bayesian version of the mixture of experts model (section 12.8) where we cluster the data into components and learn a separate linear regression model in each component at the same time. The posterior turns out to be rather nasty and Waterhouse et al. 1996 use variational approximation, which, roughly speaking, works as follows.

VARIATIONAL APPROXIMATION

We remember that in Laplace approximation, we approximate $p(\theta|\mathcal{X})$ by a Gaussian and integrate over the Gaussian instead. In *variational approximation*, we approximate the posterior by a density $q(\mathcal{Z}|\psi)$ whose parameters ψ are adjustable (Jordan et al. 1999; MacKay 2003; Bishop 2006). Hence, it is more general because we are not restricted to use a Gaussian density. Here, $\mathcal{Z}$ contains all the latent variables in the model

and the parameters θ, and ψ of the approximating model $q(\mathcal{Z}|\psi)$ are adjusted such that $q(\mathcal{Z}|\psi)$ is as close as possible to $p(\mathcal{Z}|\mathcal{X})$.

KULLBACK-LEIBLER DISTANCE

We define as the *Kullback-Leibler distance* between the two:

$$D_{\mathrm{KL}}(q||p) = \sum_{\mathcal{Z}} q(\mathcal{Z}|\psi) \log \frac{q(\mathcal{Z}|\psi)}{p(\mathcal{Z}|\mathcal{X})} \tag{16.42}$$

To make life easier, the set of latent variables (including the parameters) is assumed to be partitioned into subsets $\mathcal{Z}_i, i = 1, \ldots, k$, such that the variational distribution can be factorized:

$$q(\mathcal{Z}|\psi) = \prod_{i=1}^{k} q_i(\mathcal{Z}_i|\psi_i) \tag{16.43}$$

Adjustment of the parameters ψ_i in each factor is iterative, rather like the expectation-maximization algorithm we discussed in section 7.4. We start from (possibly random) initial values and while adjusting each, we use the expected values of the $\mathcal{Z}_j, j \neq i$ in a circular manner. This is called the *mean-field approximation*.

MEAN-FIELD APPROXIMATION

This factorization is an approximation. For example, in section 16.4.2 when we discuss regression, we write

$$p(\boldsymbol{w}, \beta) = p(\beta)p(\boldsymbol{w}|\beta)$$

because $\boldsymbol{w}$ is conditioned on β. A variational approximation would assume

$$p(\boldsymbol{w}, \beta) = p(\beta)p(\boldsymbol{w})$$

For example, in the mixture of experts model, the latent parameters are the component indices and the parameters are the parameters in the gating model, the regression weights in the local experts, the variance of the noise, and the hyperparameters of the priors for gating and regression weights; they are all factors (Waterhouse, MacKay, and Robinson 1996).

16.8 Nonparametric Bayesian Modeling

The models we discuss earlier in this chapter are all parametric, in the sense that we have models of fixed complexity with a set of parameters and these parameters are optimized using the data and the prior information. In chapter 8, we discussed nonparametric models where the training data makes up the model and hence model complexity depends on the

size of the data. Now we address how such a nonparametric approach can be used in the Bayesian setting.

A nonparametric model does not mean that the model has no parameters; it means that the number of parameters is not fixed and that their number can grow depending on the size of the data, or better still, depending on the complexity of the regularity that underlies the data. Such models are also sometimes called *infinite* models, in the sense that their complexity can keep on increasing with more data. In section 11.9, we discuss incremental neural network models where new hidden units are added when necessary and network is grown during training, but usually in parametric learning, adjusting model complexity is handled in an outer loop by checking performance on a separate validation set. The nonparametric Bayesian approach includes model adjustment in parameter training by using a suitable prior (Gershman and Blei 2012). This makes such models more flexible, and would have normally made them prone to overfitting if not for the Bayesian approach that alleviates this risk.

Because it is the parameters that grow, the priors on such parameters should be able to handle that growth and we will discuss three example prior distributions for three different type of machine learning applications, namely, Gaussian processes for supervised learning, Dirichlet processes for clustering, and beta processes for dimensionality reduction.

16.9 Gaussian Processes

Let us say we have the linear model $y = \mathbf{w}^T\mathbf{x}$. Then, for each $\mathbf{w}$, we have one line. Given a prior distribution $p(\mathbf{w})$, we get a distribution of lines, or to be more specific, for any $\mathbf{w}$, we get a distribution of y values calculated at x as $y(\mathbf{x}|\mathbf{w})$ when $\mathbf{w}$ is sampled from $p(\mathbf{w})$, and this is what we mean when we talk about a *Gaussian process*. We know that if $p(\mathbf{w})$ is Gaussian, each y is a linear combination of Gaussians and is also Gaussian; in particular, we are interested in the joint distribution of y values calculated at the N input data points, $\mathbf{x}^t, t = 1, \ldots, N$ (MacKay 1998).

GAUSSIAN PROCESS

We assume a zero-mean Gaussian prior

$$p(\mathbf{w}) \sim \mathcal{N}(\mathbf{0}, (1/\alpha)\mathbf{I})$$

Given the $N \times d$ data points $\mathbf{X}$ and the $d \times 1$ weight vector, we write the

y outputs as

$$\boldsymbol{y} = \mathbf{X}\boldsymbol{w} \tag{16.44}$$

which is N-variate Gaussian with

$$\begin{aligned} E[\boldsymbol{y}] &= \mathbf{X}E[\boldsymbol{w}] = \mathbf{0} \\ \text{Cov}(\boldsymbol{y}) &= E[\boldsymbol{y}\boldsymbol{y}^T] = \mathbf{X}E[\boldsymbol{w}\boldsymbol{w}^T]\mathbf{X}^T = \frac{1}{\alpha}\mathbf{X}\mathbf{X}^T \equiv \mathbf{K} \end{aligned} \tag{16.45}$$

where $\mathbf{K}$ is the (Gram) matrix with elements

$$K_{i,j} \equiv K(\boldsymbol{x}^i, \boldsymbol{x}^j) = \frac{(\boldsymbol{x}^i)^T\boldsymbol{x}^j}{\alpha}$$

COVARIANCE FUNCTION

This is known as the *covariance function* in the literature of Gaussian processes and the idea is the same as in kernel functions: If we use a set of basis functions $\boldsymbol{\phi}(\boldsymbol{x})$, we generalize from the dot product of the original inputs to the dot product of basis functions by a kernel

$$K_{i,j} = \frac{\boldsymbol{\phi}(\boldsymbol{x}^i)^T\boldsymbol{\phi}(\boldsymbol{x}^j)}{\alpha}$$

The actual observed output r is given by the line with added noise, $r = y + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \beta^{-1})$. For all N data points, we write it as

$$\boldsymbol{r} \sim \mathcal{N}_N(\mathbf{0}, \mathbf{C}_N) \text{ where } \mathbf{C}_N = \beta^{-1}\mathbf{I} + \mathbf{K} \tag{16.46}$$

To make a prediction, we consider the new data as the $(N+1)$st data point pair $(\boldsymbol{x}', r')$, and write the joint using all $N+1$ data points. We have

$$\boldsymbol{r}_{N+1} \sim \mathcal{N}_N(\mathbf{0}, \mathbf{C}_{N+1}) \tag{16.47}$$

where

$$\mathbf{C}_{N+1} = \begin{bmatrix} \mathbf{C}_N & \boldsymbol{k} \\ \boldsymbol{k}^T & c \end{bmatrix}$$

with $\boldsymbol{k}$ being the $N \times 1$ dimensional vector of $K(\boldsymbol{x}', \boldsymbol{x}^t), t = 1, \ldots, N$ and $c = K(\boldsymbol{x}', \boldsymbol{x}') + \beta^{-1}$. Then to make a prediction, we calculate $p(r'|\boldsymbol{x}', \mathbf{X}, \boldsymbol{r})$, which is Gaussian with

$$\begin{aligned} E[r'|\boldsymbol{x}'] &= \boldsymbol{k}^T\mathbf{C}_N^{-1}\boldsymbol{r} \\ \text{Var}(r'|\boldsymbol{x}') &= c - \boldsymbol{k}^T\mathbf{C}_N^{-1}\boldsymbol{k} \end{aligned}$$

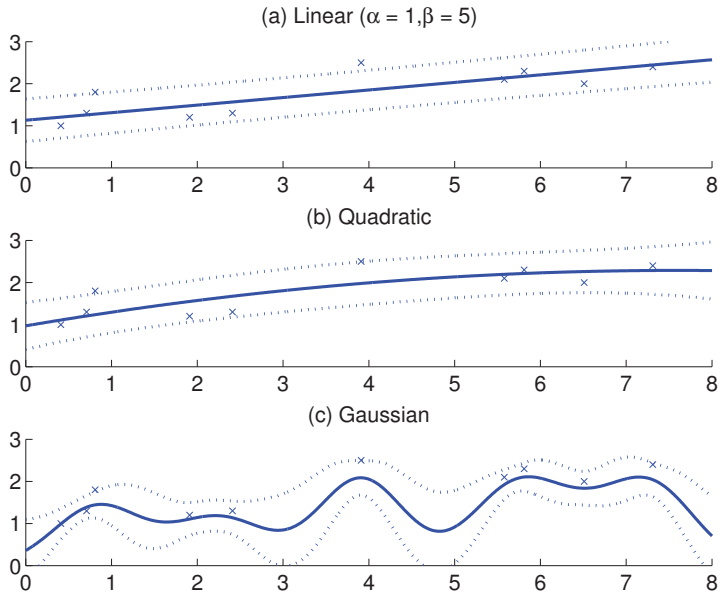


Figure 16.10 Gaussian process regression with one standard deviation error bars: (a) linear kernel, (b) quadratic kernel, and (c) Gaussian kernel with spread $s^2 = 0.5$.

An example shown in figure 16.10 uses linear, quadratic, and Gaussian kernels. The first two are defined as the dot product of their corresponding basis functions; the Gaussian kernel is defined directly as

$$K_G(\mathbf{x}^i, \mathbf{x}^j) = \exp\left[-\frac{\|\mathbf{x}^i - \mathbf{x}^j\|^2}{s^2}\right]$$

The mean, which is our point estimate (if we do not integrate over the full distribution), can also be written as a weighted sum of the kernel effects

$$E[r'|\mathbf{x}'] = \sum_t a^t K(\mathbf{x}^t, \mathbf{x}') \tag{16.48}$$

where a^t is the tth component of $\mathbf{C}_N^{-1}\mathbf{r}$. Or, we can write it as a weighted sum of the outputs of the training data points where weights are given by the kernel function

$$E[r'|\mathbf{x}'] = \sum_t r^t w^t \tag{16.49}$$

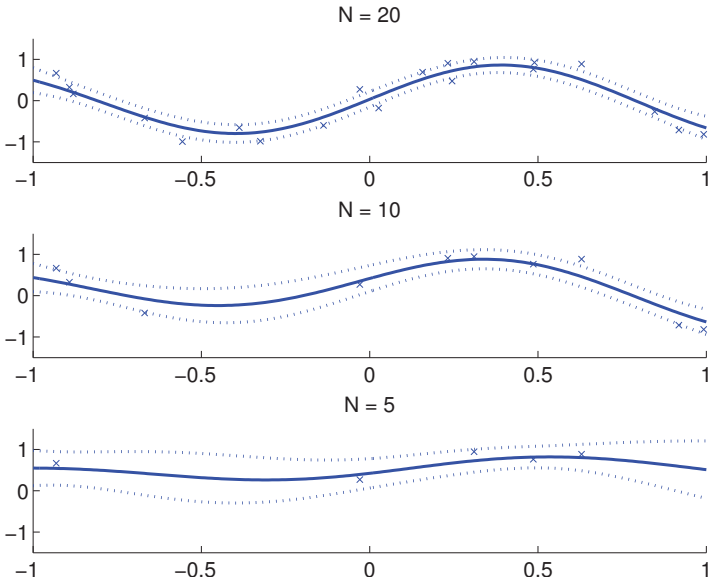


Figure 16.11 Gaussian process regression using a Gaussian kernel with $s^2 = 0.5$ and varying number of training data. We see how variance of the prediction is larger where there is few data.

where w^t is the tth component of $\boldsymbol{k}^T \mathbf{C}_N^{-1}$.

Note that we can also calculate the variance of a prediction at a point to get an idea about uncertainty in there, and it depends on the instances that affect the prediction in there. In the case of a Gaussian kernel, only instances within a locality are effective and prediction variance is high where there is little data in the vicinity (see figure 16.11).

Kernel functions can be defined and used for any application, as we have previously discussed in the context of kernel machines in chapter 13. The possibility of using kernel functions directly without needing to calculate or store the basis functions offers a great flexibility. Normally, given a training set, we first calculate the parameters, for example using equation 16.21, and then use the parameters to make predictions using equation 16.22, never needing the training set any more. This makes sense because generally the dimensionality of the parameters, which is generally $\mathcal{O}(d)$, is much lower than the size of the training set N.

When we work with basis functions, however, calculating the parameter explicitly may no longer be the case, because the dimensionality of the basis functions may be very high, even infinite. In such a case, it is cheaper to use the dual representation, taking into account the effects of training instances using kernel functions, as we do here. This idea is also used in nonparametric smoothers (chapter 8) and kernel machines (chapter 13).

The requirement here is that $\mathbf{C}_N$ be invertible and hence positive definite. For this, $\mathbf{K}$ should be semidefinite so that after adding $\beta^{-1} > 0$ to the diagonals, we get positive definiteness. We also see that the costliest operation is this inversion of the $N \times N$ matrix, which fortunately needs to be calculated only once (during training) and stored. Still, for large N, one may need an approximation.

When we use it for classification for a two-class problem, the output is filtered through a sigmoid, $y = \text{sigmoid}(\boldsymbol{w}^T\boldsymbol{x})$, and the distribution of y is no longer Gaussian. The derivation is similar except that the conditional $p(r_{N+1}|\boldsymbol{x}_{N+1}, \mathbf{X}, \boldsymbol{r})$ is not Gaussian either and we need to approximate, for example, using Laplace approximation (Bishop 2006; Rasmussen and Williams 2006).

16.10 Dirichlet Processes and Chinese Restaurants

To explain a Dirichlet process, let us start with a metaphor: There is a Chinese restaurant with a lot of tables. Customers enter one by one; we start with the first customer who sits at the first table, and any subsequent customer can either sit at one of the occupied tables or go and start a new table. The probability that a customer sits at an occupied table is proportional to the number of customers already sitting at the table, and the probability that he or she sits at a new table depends on a parameter α. This is called a *Chinese restaurant process*:

CHINESE RESTAURANT PROCESS

$$\begin{aligned} \text{Join existing table } i \text{ with } P(z_i = 1) &= \frac{n_i}{\alpha + n - 1}, i = 1, \ldots, k \\ \text{Start new table with } P(z_{k+1} = 1) &= \frac{\alpha}{\alpha + n - 1} \end{aligned}$$

where n_i is the number of customers already starting at table i, $n = \sum_{i=1}^{k} n_i$ is the total number of customers. α is the propensity to start a new table and is the parameter of the process. Note that at each step,

DIRICHLET PROCESS

the sitting arrangement of customers define a partition of integers 1 to n into k subsets. This is called a *Dirichlet process* with parameter α.

We can apply this to clustering by making customer choices not only dependent on the table occupancies but also on the input. Let us say that this is not a Chinese restaurant but the dinner of a large conference, for example, NIPS. There is a large dining lounge with many tables, and in the evening, the conference participants enter the lounge one by one. They want to eat, but they also want to participate in interesting conversation. For that, they want to sit at a table where there are already many people sitting, but they also want to sit next to people having similar research interests. If they see no such table, they start a new table and expect incoming similar participants to find and join them.

Assume that instance/participant t is represented by a d-dimensional vector $\mathbf{x}^t$, and let us assume that such $\mathbf{x}^t$ are locally Gaussian distributed. This defines a Gaussian mixture over the whole space/dining lounge, and to have it Bayesian, we define priors on the parameters of the Gaussian components, as we discuss in section 16.7. To make it nonparametric, we define a Dirichlet process as the prior so a new component can be added when necessary, as follows:

$$\begin{aligned}\text{Join component } i \text{ with } P(z_i^t = 1) &\propto \frac{n_i}{\alpha + n - 1} p(\mathbf{x}^t | \mathcal{X}_i), i = 1, \ldots, k \\ \text{Start new component with } P(z_{k+1}^t) &\propto \frac{\alpha}{\alpha + n - 1} p(\mathbf{x}^t)\end{aligned}$$

$\mathcal{X}_i$ is the set of instances previously assigned to component i; using their data and the priors, we can calculate a posterior and integrating over it, we can calculate $p(\mathbf{x}^t|\mathcal{X}_i)$. Roughly speaking, the probability this new instance is assigned to component i will be high if there are already many instances in the component, that is, due to a high prior, or if $\mathbf{x}^t$ is similar to the instances already in $\mathcal{X}_i$. If none of the existing components have a high probability, a new component is added: $p(\mathbf{x}^t)$ is the marginal probability (integrated over the component parameter priors because there is no data).

Different α may lead to different numbers of clusters. To adjust α, we can use empirical Bayes, or also define a prior on it and average it out.

In chapter 7, when we talk about k-means clustering (section 7.3), we discuss leader-cluster algorithms where new clusters are added during training and as an example of that, in section 12.2.2, we discuss adaptive resonance theory where we add a new cluster if the distance to the center

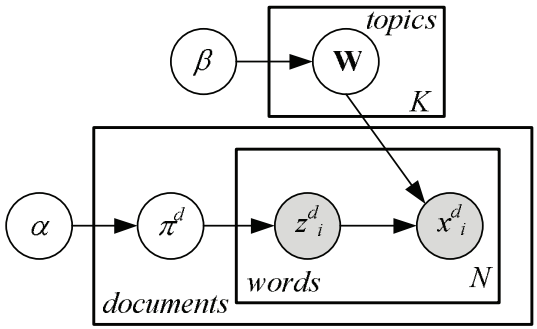


Figure 16.12 The graphical model for latent Dirichlet allocation.

of the nearest cluster is more than a vigilance value. What we have here is very similar: Assuming Gaussian components and diagonal covariance matrices, if the Euclidean distance to all clusters is high, all posteriors will be small and a new component will be added.

16.11 Latent Dirichlet Allocation

TOPIC MODELING

Let us see an application of the Bayesian approach in text processing, namely *topic modeling* (Blei 2012). In this age, there are digital repositories containing a very large number of documents—scientific articles, web pages, emails, blog posts, and so on—but finding a relevant topic for a query is very difficult, unless documents are manually annotated with topics, such as "arts," "sports," and so on. What we would like to is do this annotation automatically.

Assume we have a vocabulary with M words. Each document contains N words chosen from a number of topics in different proportions—that is, each document is a probability distribution over topics. A document may be partially "arts" and partially "politics," for example. Each topic in turn is defined as a mixture of the M words—that is, each topic corresponds to a probability distribution over words. For example, for the topic arts, the words "painting" and "sculpture" have a high probability, but the word "knee" has a low probability.

LATENT DIRICHLET ALLOCATION

In *latent Dirichlet allocation*, we define a generative process as follows (figure 16.12)—there are K topics, a vocabulary of M words, and all documents contain N words (Blei, Ng, and Jordan 2003):

To generate each document d, we first decide on the topics it will be about. These topic probabilities, $\pi_k^d, k = 1, \ldots, K$, define a multinomial distribution and are drawn from a Dirichlet prior with hyperparameter $\boldsymbol{\alpha}$ (section 16.2.1):

$$\boldsymbol{\pi}^d \sim \text{Dirichlet}_K(\boldsymbol{\alpha})$$

Once we know the topic distribution for document d, we generate its N words using it. In generating word i, first we decide on its particular topic by sampling from $\boldsymbol{\pi}$: We roll a die with K faces where face k has probability π_k. We define z_i^d as the outcome, it will be a value between 1 and K:

$$z_i^d \sim \text{Mult}_K(\boldsymbol{\pi}^d)$$

Now we know that in document d, the ith word will be about topic $z_i^d \in \{1, \ldots, K\}$. We have a $K \times M$ matrix of probabilities $\mathbf{W}$ whose row k, $\boldsymbol{w}_k \equiv [w_{k1}, \ldots, w_{kM}]^T$ gives us the probabilities of occurrences of the M words in topic k. So knowing that the topic for word i needs to come from topic z_i^d, we will sample from the multinomial distribution whose parameters are given by row z_i^d of $\mathbf{W}$ to get the word x_i^d (which is a value between 1 and M):

$$x_i^d \sim \text{Mult}_M(\boldsymbol{w}_{z_i^d})$$

This is a multinomial draw, and we define a Dirichlet prior with hyperparameter $\boldsymbol{\beta}$ on these rows of multinomial probabilities:

$$\boldsymbol{w}_k \sim \text{Dirichlet}(\boldsymbol{\beta})$$

This completes the process to generate one word. To generate the N words for the document, we do this N times; namely, for each word, we decide on a topic, then given the topic, we choose a word (inner plate in the figure). When we get to the next document, we sample another topic distribution $\boldsymbol{\pi}$ (outer plate), and then sample N words from that topic distribution.

On all the documents, we always use the same $\mathbf{W}$, and in learning, we are given a large corpus of documents, that is, only x_i^d values are observed. We can then write a posterior distribution as usual and learn $\mathbf{W}$, the word probabilities for topics shared across all documents.

Once $\mathbf{W}$ is learned, each of its rows correspond to one topic. By looking at the words with high probabilities, we can assign some meaning to these

topics. Note, however, that we will always learn some $\mathbf{W}$; whether the rows will be meaningful or not is another matter.

The model we have just discussed is parametric, and its size is fixed; we can make it nonparametric by making K, the number of topics, which is the hidden complexity parameter, increase as necessary and adapt to data using a Dirichlet process. We need to be careful though. Each document contains N words that come from some topics, but we have several documents and they all need to share the same set of topics; that is, we need to tie the Dirichlets that generate the topics. For this, we define a hierarchy; we define a higher Dirichlet process from which we draw the Dirichlets for individual documents. This is a *hierarchical Dirichlet process* (Teh et al. 2006) that allows topics learned for one document be shared by all.

HIERARCHICAL DIRICHLET PROCESS

16.12 Beta Processes and Indian Buffets

Now let us see an application of the Bayesian approach to dimensionality reduction in factor analysis. Remember that given the $N \times d$ matrix of data $\mathbf{X}$, we want to find k features or latent factors, each of which are d-dimensional such that the data can be written as a linear combination of them. That is, we want to find $\mathbf{Z}$ and $\mathbf{A}$ such that

$$\mathbf{X} = \mathbf{Z}\mathbf{A}$$

where $\mathbf{A}$ is the $k \times d$ matrix whose row j is the d-dimensional feature vector (similar to the eigenvector in PCA (section 6.3) and $\mathbf{Z}$ is $N \times k$ matrix whose row t defines instance t as a vector of features.

Let us assume that z_j^t are binary and are drawn from Bernoulli distributions with probability μ_j:

$$z_j^t = \begin{cases} 1 & \text{with probability } \mu_j \\ 0 & \text{with probability } 1 - \mu_j \end{cases} \tag{16.50}$$

So z_j^t indicates the absence/presence of hidden factor j in constructing instance t. If the corresponding factor is present, row j of $\mathbf{A}$ is chosen and the sum of all such rows chosen make up row t of $\mathbf{X}$.

We are being Bayesian so we define priors. We define a Gaussian prior on $\mathbf{A}$ and a beta conjugate prior on μ_j of Bernoulli z_j^t:

$$\mu_j \sim \text{beta}(\alpha, 1) \tag{16.51}$$

where α is the hyperparameter. We can write down the posterior and estimate the matrix $\mathbf{A}$. Looking at the rows of $\mathbf{A}$, we can get an idea about what the hidden factors represent; for example, if k is small (e.g., 2), we can plot and visualize the data.

We assume a certain k; hence this model is parametric. We can make it nonparametric and allow k increase with more data (Griffiths and Ghahramani 2011). This defines a *beta process* and the corresponding metaphor is called the *Indian buffet process*, which defines a generative model that works as follows.

BETA PROCESS
INDIAN BUFFET PROCESS

There is an Indian restaurant with a buffet that contains k dishes and each customer can take a serving of any subset of these dishes. The first customer (instance) enters and takes servings of the first m dishes; we assume m is a random variable generated from a Poisson distribution with parameter α. Then each subsequent customer n can take a serving of any existing dish j with probability n_j/n where n_j is the number of customers before who took a serving of dish j, and once he or she is done sampling the existing dishes, that customer can also ask for Poisson(α/n) additional new dishes, hence growing the model. When applied to the context of latent factor model earlier, this corresponds to a model where the number of factors need not be fixed and instead grows as the complexity inherent in data grows.

16.13 Notes

Bayesian approaches are becoming more popular recently. The use of generative graphical models corresponds quite well to the Bayesian formalism, and we are seeing interesting applications in various domains from natural language processing to computer vision to bioinformatics.

The recent field of Bayesian nonparametrics is also interesting in that adapting model complexity is now a part of training and is not an outer loop of model complexity adjustment; we expect to see more work along this direction in the near future. One example of this is the infinite hidden Markov models (Beal, Ghahramani, and Rasmussen 2002) where the number of hidden states is automatically adjusted with more data.

Due to lack of space and the need to keep the chapter to a reasonable length, the approximation and sampling methods are not discussed in detail in this chapter; see MacKay 2003, Bishop 2006, or Murphy 2012 for

more information about variational methods and Markov chain Monte Carlo sampling.

Bayesian approach is interesting and promising, and has already worked successfully in many cases, but it is far from completely supplanting the nonBayesian, or frequentist, approach. For tractability, generative models may be quite simple—for example, latent Dirichlet analysis loses the ordering or words—or the approximation methods may be hard to derive, and sampling methods slow to converge; hence frequentist shortcuts, (e.g., empirical Bayes), may be preferred in certain cases. Hence, it is best to look for an ideal compromise between the two worlds rather than fully committing to one.

16.14 Exercises

1. For the setting of figure 16.3, observe how the posterior changes as we change N, σ^2, and σ_0^2.
2. Let us denote by x the number of spam emails I receive in a random sample of n. Assume that the prior for q, the proportion of spam emails is uniform in $[0, 1]$. Find the posterior distribution for $p(q|x)$.
3. As above, except that assume that $p(q) \sim \mathcal{N}(\mu_0, \sigma_0^2)$. Also assume n is large so that you can use central limit theorem and approximate binomial by a Gaussian. Derive $p(q|x)$.
4. What is $\text{Var}(r')$ when the maximum likelihood estimator is used? Compare it with equation 16.25.
5. In figure 16.10, how does the fit change when we change s^2?

   SOLUTION: As usual, s is the smoothing parameter and we get smoother fits as we increase s.
6. Propose a filtering algorithm to choose a subset of the training set in Gaussian processes.

   SOLUTION: One nice property of Gaussian processes is that we can calculate the variance at a certain point. For any instance from the training set, we can calculate the leave-one-out estimate there and check whether the actual output is in, for example, the 95 percent prediction interval. If it is, this means that we do not need that instance and it can be left out. Those that cannot be pruned will be just like the support vectors in a kernel machine, namely, those instances that are stored and needed, to bound the total error of the fit.

ACTIVE LEARNING

7. *Active learning* is when the learner is able to generate x itself and ask a su-

pervisor to provide the corresponding r value during learning one by one, instead of passively being given a training set. How can we implement active learning using Gaussian processes? (Hint: Where do we have the largest uncertainty?)

SOLUTION: This is just like the previous exercise, except that we add instead of prune. Using the same logic, we can see that we need instances where the prediction interval is large. Given the variance as a function of $\mathbf{x}$, we search for its local maxima. In the case of a Gaussian kernel, we expect points that are distant from training data to have high variance, but this need not be the case for all kernels. While searching, we need to make sure that we do not go out of the valid input bounds.

8. Let us say we have a set of documents where for each document, we have one copy in English and one in French. How can we extend latent Dirichlet allocation for this case?

16.15 References

Beal, M. J., Z. Ghahramani, and C. E. Rasmussen. 2002. "The Infinite Hidden Markov Model." In *Advances in Neural Information Processing Systems 14*, ed. T. G. Dietterich, S. Becker, and Z. Ghahramani, 577–585. Cambridge, MA: MIT Press.

Bishop, C. M. 2006. *Pattern Recognition and Machine Learning*. New York: Springer.

Blei, D. M. 2012. "Probabilistic Topic Models." *Communications of the ACM* 55 (4): 77–84.

Blei, D. M., A. Y. Ng, and M. I. Jordan. 2003. "Latent Dirichlet Allocation." *Journal of Machine Intelligence* 3:993–1022.

Figueiredo, M. A. T. 2003. "Adaptive Sparseness for Supervised Learning." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 25:1150–1159.

Gershman, S. J., and D. M. Blei. 2012. "A Tutorial on Bayesian Nonparametric Models." *Journal of Mathematical Psychology* 56:1–12.

Griffiths, T. L., and Z. Ghahramani. 2011. "The Indian Buffet Process: An Introduction and Review." *Journal of Machine Learning Research* 12:1185–1224.

Hastie, T., R. Tibshirani, and J. Friedman. 2011. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. New York: Springer.

Hoff, P. D. 2009. *A First Course in Bayesian Statistical Methods*. New York: Springer.

Jordan, M. I., Z. Ghahramani, T. S. Jaakkola, L. K. Saul. 1999. "An Introduction to Variational Methods for Graphical Models." *Machine Learning* 37:183–233.

MacKay, D. J. C. 1998. "Introduction to Gaussian Processes." In *Neural Networks and Machine Learning*, ed. C. M. Bishop, 133–166. Berlin: Springer.

MacKay, D. J. C. 2003. *Information Theory, Inference, and Learning Algorithms*. Cambridge, UK: Cambridge University Press.

Murphy, K. P. 2007. "Conjugate Bayesian Analysis of the Gaussian Distribution." `http://www.cs.ubc.ca/~murphyk/Papers/bayesGauss.pdf`.

Murphy, K. P. 2012. *Machine Learning: A Probabilistic Perspective*. Cambridge, MA: MIT Press.

Rasmussen, C. E. , and C. K. I. Williams. 2006. *Gaussian Processes for Machine Learning*. Cambridge, MA: MIT Press.

Teh, Y. W., M. I. Jordan, M. J. Beal, and D. M. Blei. 2006. "Hierarchical Dirichlet Processes." *Journal of Americal Statistical Association* 101: 1566–1581.

Tibshirani, R. 1996. "Regression Shrinkage and Selection via the Lasso." *Journal of the Royal Statistical Society B* 58: 267–288.

Waterhouse, S., D. MacKay, and T. Robinson. 1996. "Bayesian Methods for Mixture of Experts." In *Advances in Neural Information Processing Systems 8*, ed. D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, 351–357. Cambridge, MA: MIT Press.

17 *Combining Multiple Learners*

We discussed many different learning algorithms in the previous chapters. Though these are generally successful, no one single algorithm is always the most accurate. Now, we are going to discuss models composed of multiple learners that complement each other so that by combining them, we attain higher accuracy.

17.1 Rationale

IN ANY APPLICATION, we can use one of several learning algorithms, and with certain algorithms, there are hyperparameters that affect the final learner. For example, in a classification setting, we can use a parametric classifier or a multilayer perceptron, and, for example, with a multilayer perceptron, we should also decide on the number of hidden units. The No Free Lunch Theorem states that there is no single learning algorithm that in any domain always induces the most accurate learner. The usual approach is to try many and choose the one that performs the best on a separate validation set.

Each learning algorithm dictates a certain model that comes with a set of assumptions. This inductive bias leads to error if the assumptions do not hold for the data. Learning is an ill-posed problem and with finite data, each algorithm converges to a different solution and fails under different circumstances. The performance of a learner may be fine-tuned to get the highest possible accuracy on a validation set, but this fine-tuning is a complex task and still there are instances on which even the best learner is not accurate enough. The idea is that there may be another learner that is accurate on these. By suitably combining multiple *base-learners* then, accuracy can be improved. Recently with computation and

BASE-LEARNER

memory getting cheaper, such systems composed of multiple learners have become popular (Kuncheva 2004).

There are basically two questions here:

1. How do we generate base-learners that complement each other?
2. How do we combine the outputs of base-learners for maximum accuracy?

Our discussion in this chapter will answer these two related questions. We will see that model combination is not a trick that always increases accuracy; model combination does always increase time and space complexity of training and testing, and unless base-learners are trained carefully and their decisions combined smartly, we will only pay for this extra complexity without any significant gain in accuracy.

17.2 Generating Diverse Learners

DIVERSITY

Since there is no point in combining learners that always make similar decisions, the aim is to be able to find a set of *diverse* learners who differ in their decisions so that they complement each other. At the same time, there cannot be a gain in overall success unless the learners are accurate, at least in their domain of expertise. We therefore have this double task of maximizing individual accuracies and the diversity between learners. Let us now discuss the different ways to achieve this.

Different Algorithms

We can use different learning algorithms to train different base-learners. Different algorithms make different assumptions about the data and lead to different classifiers. For example, one base-learner may be parametric and another may be nonparametric. When we decide on a single algorithm, we give emphasis to a single method and ignore all others. Combining multiple learners based on multiple algorithms, we free ourselves from taking a decision and we no longer put all our eggs in one basket.

Different Hyperparameters

We can use the same learning algorithm but use it with different hyperparameters. Examples are the number of hidden units in a multilayer

perceptron, k in k-nearest neighbor, error threshold in decision trees, the kernel function in support vector machines, and so forth. With a Gaussian parametric classifier, whether the covariance matrices are shared or not is a hyperparameter. If the optimization algorithm uses an iterative procedure such as gradient descent whose final state depends on the initial state, such as in backpropagation with multilayer perceptrons, the initial state, for example, the initial weights, is another hyperparameter. When we train multiple base-learners with different hyperparameter values, we average over this factor and reduce variance, and therefore error.

Different Input Representations

Separate base-learners may be using different *representations* of the same input object or event, making it possible to integrate different types of sensors/measurements/modalities. Different representations make different characteristics explicit allowing better identification. In many applications, there are multiple sources of information, and it is desirable to use all of these data to extract more information and achieve higher accuracy in prediction.

For example, in speech recognition, to recognize the uttered words, in addition to the acoustic input, we can also use the video image of the speaker's lips and shape of the mouth as the words are spoken. This is similar to *sensor fusion* where the data from different sensors are integrated to extract more information for a specific application. Another example is information, for example, image retrieval where in addition to the image itself, we may also have text annotation in the form of keywords. In such a case, we want to be able to combine both of these sources to find the right set of images; this is also sometimes called *multi-view learning*.

SENSOR FUSION

MULTI-VIEW LEARNING

The simplest approach is to concatenate all data vectors and treat it as one large vector from a single source, but this does not seem theoretically appropriate since this corresponds to modeling data as sampled from one multivariate statistical distribution. Moreover, larger input dimensionalities make the systems more complex and require larger samples for the estimators to be accurate. The approach we take is to make separate predictions based on different sources using separate base-learners, then combine their predictions.

Even if there is a single input representation, by choosing random subsets from it, we can have classifiers using different input features; this is

RANDOM SUBSPACE

called the *random subspace method* (Ho 1998). This has the effect that different learners will look at the same problem from different points of view and will be robust; it will also help reduce the curse of dimensionality because inputs are fewer dimensional.

Different Training Sets

Another possibility is to train different base-learners by different subsets of the training set. This can be done randomly by drawing random training sets from the given sample; this is called *bagging*. Or, the learners can be trained serially so that instances on which the preceding base-learners are not accurate are given more emphasis in training later base-learners; examples are *boosting* and *cascading*, which actively try to generate complementary learners, instead of leaving this to chance.

The partitioning of the training sample can also be done based on locality in the input space so that each base-learner is trained on instances in a certain local part of the input space; this is what is done by the *mixture of experts* that we discussed in chapter 12 but that we revisit in this context of combining multiple learners. Similarly, it is possible to define the main task in terms of a number of subtasks to be implemented by the base-learners, as is done by *error-correcting output codes*.

Diversity vs. Accuracy

One important note is that when we generate multiple base-learners, we want them to be reasonably accurate but do not require them to be very accurate individually, so they are not, and need not be, optimized separately for best accuracy. The base-learners are not chosen for their accuracy, but for their simplicity. We do require, however, that the base-learners be diverse, that is, accurate on different instances, specializing in subdomains of the problem. What we care for is the final accuracy when the base-learners are combined, rather than the accuracies of the base-learners we started from. Let us say we have a classifier that is 80 percent accurate. When we decide on a second classifier, we do not care for the overall accuracy; we care only about how accurate it is on the 20 percent that the first classifier misclassifies, as long as we know when to use which one.

This implies that the required accuracy and diversity of the learners also depend on how their decisions are to be combined, as we will dis-

cuss next. If, as in a voting scheme, a learner is consulted for all inputs, it should be accurate everywhere and diversity should be enforced everywhere; if we have a partioning of the input space into regions of expertise for different learners, diversity is already guaranteed by this partitioning and learners need to be accurate only in their own local domains.

17.3 Model Combination Schemes

There are also different ways the multiple base-learners are combined to generate the final output:

MULTIEXPERT COMBINATION

- *Multiexpert combination* methods have base-learners that work in *parallel*. These methods can in turn be divided into two:
  - In the *global* approach, also called *learner fusion*, given an input, all base-learners generate an output and all these outputs are used. Examples are *voting* and *stacking*.
  - In the *local* approach, or *learner selection*, for example, in *mixture of experts*, there is a *gating* model, which looks at the input and chooses one (or very few) of the learners as responsible for generating the output.

MULTISTAGE COMBINATION

- *Multistage combination* methods use a *serial* approach where the next base-learner is trained with or tested on only the instances where the previous base-learners are not accurate enough. The idea is that the base-learners (or the different representations they use) are sorted in increasing complexity so that a complex base-learner is not used (or its complex representation is not extracted) unless the preceding simpler base-learners are not confident. An example is *cascading*.

Let us say that we have L base-learners. We denote by $d_j(x)$ the prediction of base-learner $\mathcal{M}_j$ given the arbitrary dimensional input x. In the case of multiple representations, each $\mathcal{M}_j$ uses a different input representation x_j. The final prediction is calculated from the predictions of the base-learners:

$$y = f(d_1, d_2, \ldots, d_L|\Phi) \tag{17.1}$$

where $f(\cdot)$ is the combining function with Φ denoting its parameters.

When there are K outputs, for each learner there are $d_{ji}(x), i = 1, \ldots, K$, $j = 1, \ldots, L$, and, combining them, we also generate K values, $y_i, i =$

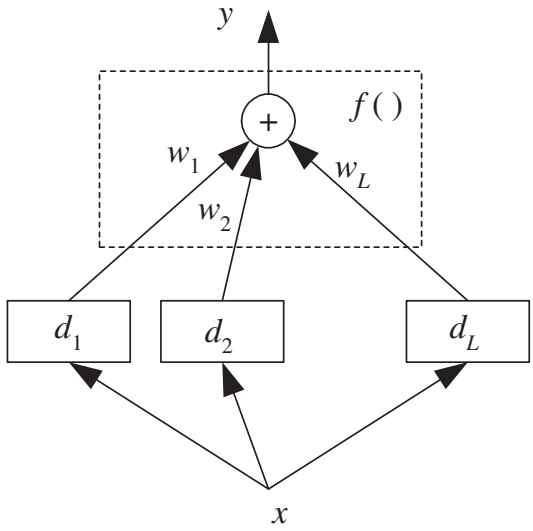


Figure 17.1 Base-learners are d_j and their outputs are combined using $f(\cdot)$. This is for a single output; in the case of classification, each base-learner has K outputs that are separately used to calculate y_i, and then we choose the maximum. Note that here all learners observe the same input; it may be the case that different learners observe different representations of the same input object or event.

$1, \ldots, K$ and then for example in classification, we choose the class with the maximum y_i value:

$$\text{Choose } C_i \text{ if } y_i = \max_{k=1}^{K} y_k$$

17.4 Voting

VOTING

The simplest way to combine multiple classifiers is by *voting*, which corresponds to taking a linear combination of the learners (see figure 17.1):

$$y_i = \sum_j w_j d_{ji} \text{ where } w_j \geq 0, \sum_j w_j = 1 \tag{17.2}$$

ENSEMBLES
LINEAR OPINION POOLS

This is also known as *ensembles* and *linear opinion pools*. In the simplest case, all learners are given equal weight and we have *simple voting*

Table 17.1 Classifier combination rules

Rule	Fusion function $f(\cdot)$
Sum	$y_i = \frac{1}{L}\sum_{j=1}^{L} d_{ji}$
Weighted sum	$y_i = \sum_j w_j d_{ji}, w_j \geq 0, \sum_j w_j = 1$
Median	$y_i = \text{median}_j d_{ji}$
Minimum	$y_i = \min_j d_{ji}$
Maximum	$y_i = \max_j d_{ji}$
Product	$y_i = \prod_j d_{ji}$

Table 17.2 Example of combination rules on three learners and three classes

	C_1	C_2	C_3
d_1	0.2	0.5	0.3
d_2	0.0	0.6	0.4
d_3	0.4	0.4	0.2
Sum	0.2	**0.5**	0.3
Median	0.2	**0.5**	0.4
Minimum	0.0	**0.4**	0.2
Maximum	0.4	**0.6**	0.4
Product	0.0	**0.12**	0.032

that corresponds to taking an average. Still, taking a (weighted) sum is only one of the possibilities and there are also other combination rules, as shown in table 17.1 (Kittler et al. 1998). If the outputs are not posterior probabilities, these rules require that outputs be normalized to the same scale (Jain, Nandakumar, and Ross 2005).

An example of the use of these rules is shown in table 17.2, which demonstrates the effects of different rules. Sum rule is the most intuitive and is the most widely used in practice. Median rule is more robust to outliers; minimum and maximum rules are pessimistic and optimistic, respectively. With the product rule, each learner has veto power; regardless of the other ones, if one learner has an output of 0, the overall output goes to 0. Note that after the combination rules, y_i do not necessarily sum up to 1.

In weighted sum, d_{ji} is the vote of learner j for class C_i and w_j is the weight of its vote. Simple voting is a special case where all voters have equal weight, namely, $w_j = 1/L$. In classification, this is called *plurality voting* where the class having the maximum number of votes is the winner. When there are two classes, this is *majority voting* where the winning class gets more than half of the votes (exercise 1). If the voters can also supply the additional information of how much they vote for each class (e.g., by the posterior probability), then after normalization, these can be used as weights in a *weighted voting* scheme. Equivalently, if d_{ji} are the class posterior probabilities, $P(C_i|x, \mathcal{M}_j)$, then we can just sum them up ($w_j = 1/L$) and choose the class with maximum y_i.

In the case of regression, simple or weighted averaging or median can be used to fuse the outputs of base-regressors. Median is more robust to noise than the average.

Another possible way to find w_j is to assess the accuracies of the learners (regressor or classifier) on a separate validation set and use that information to compute the weights, so that we give more weights to more accurate learners. These weights can also be learned from data, as we will discuss when we discuss stacked generalization in section 17.9.

Voting schemes can be seen as approximations under a Bayesian framework with weights approximating prior model probabilities, and model decisions approximating model-conditional likelihoods. This is *Bayesian model combination*—see section 16.6. For example, in classification we have $w_j \equiv P(\mathcal{M}_j)$, $d_{ji} = P(C_i|x, \mathcal{M}_j)$, and equation 17.2 corresponds to

BAYESIAN MODEL COMBINATION

$$P(C_i|x) = \sum_{\text{all models } \mathcal{M}_j} P(C_i|x, \mathcal{M}_j)P(\mathcal{M}_j) \tag{17.3}$$

Simple voting corresponds to a uniform prior. If we have a prior distribution preferring simpler models, this would give larger weights to them. We cannot integrate over all models; we only choose a subset for which we believe $P(\mathcal{M}_j)$ is high, or we can have another Bayesian step and calculate $P(\mathcal{M}_j|\mathcal{X})$, the probability of a model given the sample, and sample high probable models from this density.

Hansen and Salamon (1990) have shown that given independent two-class classifiers with success probability higher than 1/2, namely, better than random guessing, by taking a majority vote, the accuracy increases as the number of voting classifiers increases.

Let us assume that d_j are iid with expected value $E[d_j]$ and variance $\text{Var}(d_j)$, then when we take a simple average with $w_j = 1/L$, the expected

value and variance of the output are

$$E[y] = E\left[\sum_j \frac{1}{L} d_j\right] = \frac{1}{L} LE[d_j] = E[d_j]$$

$$\text{Var}(y) = \text{Var}\left(\sum_j \frac{1}{L} d_j\right) = \frac{1}{L^2}\text{Var}\left(\sum_j d_j\right) = \frac{1}{L^2} L\text{Var}(d_j) = \frac{1}{L}\text{Var}(d_j) \tag{17.4}$$

We see that the expected value does not change, so the bias does not change. But variance, and therefore mean square error, decreases as the number of independent voters, L, increases. In the general case,

$$\text{Var}(y) = \frac{1}{L^2}\text{Var}\left(\sum_j d_j\right) = \frac{1}{L^2}\left[\sum_j \text{Var}(d_j) + 2\sum_j \sum_{i<j} \text{Cov}(d_j, d_i)\right] \tag{17.5}$$

which implies that if learners are positively correlated, variance (and error) increase. We can thus view using different algorithms and input features as efforts to decrease, if not completely eliminate, the positive correlation. In section 17.10, we discuss pruning methods to remove learners with high positive correlation fron an ensemble.

We also see here that further decrease in variance is possible if the voters are not independent but negatively correlated. The error then decreases if the accompanying increase in bias is not higher because these aims are contradictory; we cannot have a number of classifiers that are all accurate *and* negatively correlated. In mixture of experts for example, where learners are localized, the experts are negatively correlated but biased (Jacobs 1997).

If we view each base-learner as a random noise function added to the true discriminant/regression function and if these noise functions are uncorrelated with 0 mean, then the averaging of the individual estimates is like averaging over the noise. In this sense, voting has the effect of smoothing in the functional space and can be thought of as a regularizer with a smoothness assumption on the true function (Perrone 1993). We saw an example of this in figure 4.5d, where, averaging over models with large variance, we get a better fit than those of the individual models. This is the idea in voting: We vote over models with high variance and low bias so that after combination, the bias remains small and we reduce the variance by averaging. Even if the individual models are biased, the decrease in variance may offset this bias and still a decrease in error is possible.

17.5 Error-Correcting Output Codes

ERROR-CORRECTING OUTPUT CODES

In *error-correcting output codes* (ECOC) (Dietterich and Bakiri 1995), the main classification task is defined in terms of a number of subtasks that are implemented by the base-learners. The idea is that the original task of separating one class from all other classes may be a difficult problem. Instead, we want to define a set of simpler classification problems, each specializing in one aspect of the task, and combining these simpler classifiers, we get the final classifier.

Base-learners are binary classifiers having output $-1/+1$, and there is a *code matrix* $\mathbf{W}$ of $K \times L$ whose K rows are the binary codes of classes in terms of the L base-learners d_j. For example, if the second row of $\mathbf{W}$ is $[-1,+1,+1,-1]$, this means that for us to say an instance belongs to C_2, the instance should be on the negative side of d_1 and d_4, and on the positive side of d_2 and d_3. Similarly, the columns of the code matrix defines the task of the base-learners. For example, if the third column is $[-1,+1,+1]^T$, we understand that the task of the third base-learner, d_3, is to separate the instances of C_1 from the instances of C_2 and C_3 combined. This is how we form the training set of the base-learners. For example in this case, all instances labeled with C_2 and C_3 form $\mathcal{X}_3^+$ and instances labeled with C_1 form $\mathcal{X}_3^-$, and d_3 is trained so that $x^t \in \mathcal{X}_3^+$ give output $+1$ and $x^t \in \mathcal{X}_3^-$ give output -1.

The code matrix thus allows us to define a polychotomy ($K > 2$ classification problem) in terms of dichotomies ($K = 2$ classification problem), and it is a method that is applicable using any learning algorithm to implement the dichotomizer base-learners—for example, linear or multilayer perceptrons (with a single output), decision trees, or SVMs whose original definition is for two-class problems.

The typical one discriminant per class setting corresponds to the diagonal code matrix where $L = K$. For example, for $K = 4$, we have

$$\mathbf{W} = \begin{bmatrix} +1 & -1 & -1 & -1 \\ -1 & +1 & -1 & -1 \\ -1 & -1 & +1 & -1 \\ -1 & -1 & -1 & +1 \end{bmatrix}$$

The problem here is that if there is an error with one of the base-learners, there may be a misclassification because the class code words are so similar. So the approach in error-correcting codes is to have $L > K$ and increase the Hamming distance between the code words. One pos-

sibility is *pairwise separation* of classes where there is a separate base-learner to separate C_i from C_j, for $i < j$ (section 10.4). In this case, $L = K(K-1)/2$ and with $K = 4$, the code matrix is

$$\mathbf{W} = \begin{bmatrix} +1 & +1 & +1 & 0 & 0 & 0 \\ -1 & 0 & 0 & +1 & +1 & 0 \\ 0 & -1 & 0 & -1 & 0 & +1 \\ 0 & 0 & -1 & 0 & -1 & -1 \end{bmatrix}$$

where a 0 entry denotes "don't care." That is, d_1 is trained to separate C_1 from C_2 and does not use the training instances belonging to the other classes. Similarly, we say that an instance belongs to C_2 if $d_1 = -1$ and $d_4 = d_5 = +1$, and we do not consider the values of d_2, d_3, and d_6. The problem here is that L is $\mathcal{O}(K^2)$, and for large K pairwise separation may not be feasible.

If we can have L high, we can just randomly generate the code matrix with $-1/+1$ and this will work fine, but if we want to keep L low, we need to optimize $\mathbf{W}$. The approach is to set L beforehand and then find $\mathbf{W}$ such that the distances between rows, and at the same time the distances between columns, are as large as possible, in terms of Hamming distance. With K classes, there are $2^{(K-1)} - 1$ possible columns, namely, two-class problems. This is because K bits can be written in 2^K different ways and complements (e.g., "0101" and "1010," from our point of view, define the same discriminant) dividing the possible combinations by 2 and then subtracting 1 because a column of all 0s (or 1s) is useless. For example, when $K = 4$, we have

$$\mathbf{W} = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & +1 & +1 & +1 & +1 \\ -1 & +1 & +1 & -1 & -1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 \end{bmatrix}$$

When K is large, for a given value of L, we look for L columns out of the $2^{(K-1)}-1$. We would like these columns of $\mathbf{W}$ to be as different as possible so that the tasks to be learned by the base-learners are as different from each other as possible. At the same time, we would like the rows of $\mathbf{W}$ to be as different as possible so that we can have maximum error correction in case one or more base-learners fail.

ECOC can be written as a voting scheme where the entries of $\mathbf{W}$, w_{ij},

are considered as vote weights:

$$y_i = \sum_{j=1}^{L} w_{ij} d_j \tag{17.6}$$

and then we choose the class with the highest y_i. Taking a weighted sum and then choosing the maximum instead of checking for an exact match allows d_j to no longer need to be binary but to take a value between -1 and $+1$, carrying soft certainties instead of hard decisions. Note that a value p_j between 0 and 1, for example, a posterior probability, can be converted to a value d_j between -1 and $+1$ simply as

$$d_j = 2p_j - 1$$

The difference between equation 17.6 and the generic voting model of equation 17.2 is that the weights of votes can be different for different classes, namely, we no longer have w_j but w_{ij}, and also that $w_j \geq 0$ whereas w_{ij} are -1, 0, or $+1$.

One problem with ECOC is that because the code matrix $\mathbf{W}$ is set a priori, there is no guarantee that the subtasks as defined by the columns of $\mathbf{W}$ will be simple. Dietterich and Bakiri (1995) report that the dichotomizer trees may be larger than the polychotomizer trees and when multilayer perceptrons are used, there may be slower convergence by backpropagation.

17.6 Bagging

BAGGING

Bagging is a voting method whereby base-learners are made different by training them over slightly different training sets. Generating L slightly different samples from a given sample is done by bootstrap, where given a training set $\mathcal{X}$ of size N, we draw N instances randomly from $\mathcal{X}$ *with replacement*. Because sampling is done with replacement, it is possible that some instances are drawn more than once and that certain instances are not drawn at all. When this is done to generate L samples $\mathcal{X}_j, j = 1, \ldots, L$, these samples are similar because they are all drawn from the same original sample, but they are also slightly different due to chance. The base-learners d_j are trained with these L samples $\mathcal{X}_j$.

UNSTABLE ALGORITHM

A learning algorithm is an *unstable algorithm* if small changes in the training set causes a large difference in the generated learner, namely, the

learning algorithm has high variance. Bagging, short for bootstrap aggregating, uses bootstrap to generate L training sets, trains L base-learners using an unstable learning procedure, and then, during testing, takes an average (Breiman 1996). Bagging can be used both for classification and regression. In the case of regression, to be more robust, one can take the median instead of the average when combining predictions.

We saw before that averaging reduces variance only if the positive correlation is small; an algorithm is stable if different runs of the same algorithm on resampled versions of the same dataset lead to learners with high positive correlation. Algorithms such as decision trees and multilayer perceptrons are unstable. Nearest neighbor is stable, but condensed nearest neighbor is unstable (Alpaydın 1997). If the original training set is large, then we may want to generate smaller sets of size $N' < N$ from them using bootstrap, since otherwise the bootstrap replicates $\mathcal{X}_j$ will be too similar, and d_j will be highly correlated.

17.7 Boosting

In bagging, generating complementary base-learners is left to chance and to the unstability of the learning method. In boosting, we actively try to generate complementary base-learners by training the next learner on the mistakes of the previous learners. The original *boosting* algorithm (Schapire 1990) combines three weak learners to generate a strong learner. A *weak learner* has error probability less than 1/2, which makes it better than random guessing on a two-class problem, and a *strong learner* has arbitrarily small error probability.

BOOSTING

WEAK LEARNER

STRONG LEARNER

Given a large training set, we randomly divide it into three. We use $\mathcal{X}_1$ and train d_1. We then take $\mathcal{X}_2$ and feed it to d_1. We take all instances misclassified by d_1 and also as many instances on which d_1 is correct from $\mathcal{X}_2$, and these together form the training set of d_2. We then take $\mathcal{X}_3$ and feed it to d_1 and d_2. The instances on which d_1 and d_2 disagree form the training set of d_3. During testing, given an instance, we give it to d_1 and d_2; if they agree, that is the response, otherwise the response of d_3 is taken as the output. Schapire (1990) has shown that this overall system has reduced error rate, and the error rate can arbitrarily be reduced by using such systems recursively, that is, a boosting system of three models used as d_j in a higher system.

Though it is quite successful, the disadvantage of the original boost-

Training:
For all $\{x^t, r^t\}_{t=1}^N \in \mathcal{X}$, initialize $p_1^t = 1/N$
For all base-learners $j = 1, \ldots, L$
Randomly draw $\mathcal{X}_j$ from $\mathcal{X}$ with probabilities p_j^t
Train d_j using $\mathcal{X}_j$
For each (x^t, r^t), calculate $y_j^t \leftarrow d_j(x^t)$
Calculate error rate: $\epsilon_j \leftarrow \sum_t p_j^t \cdot 1(y_j^t \neq r^t)$
If $\epsilon_j > 1/2$, then $L \leftarrow j - 1$; stop
$\beta_j \leftarrow \epsilon_j / (1 - \epsilon_j)$
For each (x^t, r^t), decrease probabilities if correct:
If $y_j^t = r^t$, then $p_{j+1}^t \leftarrow \beta_j p_j^t$ Else $p_{j+1}^t \leftarrow p_j^t$
Normalize probabilities:
$Z_j \leftarrow \sum_t p_{j+1}^t$; $p_{j+1}^t \leftarrow p_{j+1}^t / Z_j$
Testing:
Given x, calculate $d_j(x), j = 1, \ldots, L$
Calculate class outputs, $i = 1, \ldots, K$:
$y_i = \sum_{j=1}^L \left(\log \frac{1}{\beta_j}\right) d_{ji}(x)$

Figure 17.2 AdaBoost algorithm.

ing method is that it requires a very large training sample. The sample should be divided into three and furthermore, the second and third classifiers are only trained on a subset on which the previous ones err. So unless one has a quite large training set, d_2 and d_3 will not have training sets of reasonable size. Drucker et al. (1994) use a set of 118,000 instances in boosting multilayer perceptrons for optical handwritten digit recognition.

ADABOOST

Freund and Schapire (1996) proposed a variant, named *AdaBoost*, short for adaptive boosting, that uses the same training set over and over and thus need not be large, but the classifiers should be simple so that they do not overfit. AdaBoost can also combine an arbitrary number of base-learners, not three.

Many variants of AdaBoost have been proposed; here, we discuss the original algorithm AdaBoost.M1 (see figure 17.2). The idea is to modify the probabilities of drawing the instances as a function of the error. Let us say p_j^t denotes the probability that the instance pair (x^t, r^t) is drawn to train the jth base-learner. Initially, all $p_1^t = 1/N$. Then we add new

base-learners as follows, starting from $j = 1$: ϵ_j denotes the error rate of d_j. AdaBoost requires that learners are weak, that is, $\epsilon_j < 1/2, \forall j$; if not, we stop adding new base-learners. Note that this error rate is not on the original problem but on the dataset used at step j. We define $\beta_j = \epsilon_j/(1 - \epsilon_j) < 1$, and we set $p^t_{j+1} = \beta_j p^t_j$ if d_j correctly classifies x^t; otherwise, $p^t_{j+1} = p^t_j$. Because p^t_{j+1} should be probabilities, there is a normalization where we divide p^t_{j+1} by $\sum_t p^t_{j+1}$, so that they sum up to 1. This has the effect that the probability of a correctly classified instance is decreased, and the probability of a misclassified instance increases. Then a new sample of the same size is drawn from the original sample according to these modified probabilities, p^t_{j+1}, with replacement, and is used to train d_{j+1}.

This has the effect that d_{j+1} focuses more on instances misclassified by d_j; that is why the base-learners are chosen to be simple and not accurate, since otherwise the next training sample would contain only a few outlier and noisy instances repeated many times over. For example, with decision trees, *decision stumps*, which are trees grown only one or two levels, are used. So it is clear that these would have bias but the decrease in variance is larger and the overall error decreases. An algorithm like the linear discriminant has low variance, and we cannot gain by AdaBoosting linear discriminants.

Once training is done, AdaBoost is a voting method. Given an instance, all d_j decide and a weighted vote is taken where weights are proportional to the base-learners' accuracies (on the training set): $w_j = \log(1/\beta_j)$. Freund and Schapire (1996) showed improved accuracy in twenty-two benchmark problems, equal accuracy in one problem, and worse accuracy in four problems.

MARGIN

Schapire et al. (1998) explain that the success of AdaBoost is due to its property of increasing the *margin*. If the margin increases, the training instances are better separated and an error is less likely. This makes AdaBoost's aim similar to that of support vector machines (chapter 13).

In AdaBoost, although different base-learners have slightly different training sets, this difference is not left to chance as in bagging, but is a function of the error of the previous base-learner. The actual performance of boosting on a particular problem is clearly dependent on the data and the base-learner. There should be enough training data and the base-learner should be weak but not too weak, and boosting is especially susceptible to noise and outliers.

AdaBoost has also been generalized to regression: One straightforward

way, proposed by Avnimelech and Intrator (1997), checks for whether the prediction error is larger than a certain threshold, and if so marks it as error, then uses AdaBoost proper. In another version (Drucker 1997), probabilities are modified based on the magnitude of error, such that instances where the previous base-learner commits a large error, have a higher probability of being drawn to train the next base-learner. Weighted average, or median, is used to combine the predictions of the base-learners.

17.8 The Mixture of Experts Revisited

MIXTURE OF EXPERTS

In voting, the weights w_j are constant over the input space. In the *mixture of experts* architecture, which we previously discussed in section 12.8) as a local method, as an extension of radial basis functions, there is a gating network whose outputs are weights of the experts. This architecture can then be viewed as a voting method where the votes depend on the input, and may be different for different inputs. The competitive learning algorithm used by the mixture of experts localizes the base-learners such that each of them becomes an expert in a different part of the input space and have its weight, $w_j(x)$, close to 1 in its region of expertise. The final output is a weighted average as in voting

$$y = \sum_{j=1}^{L} w_j(x) d_j \tag{17.7}$$

except in this case, both the base-learners and the weights are a function of the input (see figure 17.3).

Jacobs (1997) has shown that in the mixture of experts architecture, experts are biased but are negatively correlated. As training proceeds, bias decreases and expert variances increase but at the same time as experts localize in different parts of the input space, their covariances get more and more negative, which, due to equation 17.5, decreases the total variance, and thus the error. In section 12.8, we considered the case where both experts and gating are linear functions but a nonlinear method, for example, a multilayer perceptron with hidden units, can also be used for both. This may decrease the expert biases but risks increasing expert variances and overfitting.

DYNAMIC CLASSIFIER SELECTION

In *dynamic classifier selection*, similar to the gating network of mixture of experts, there is first a system which takes a test input and estimates

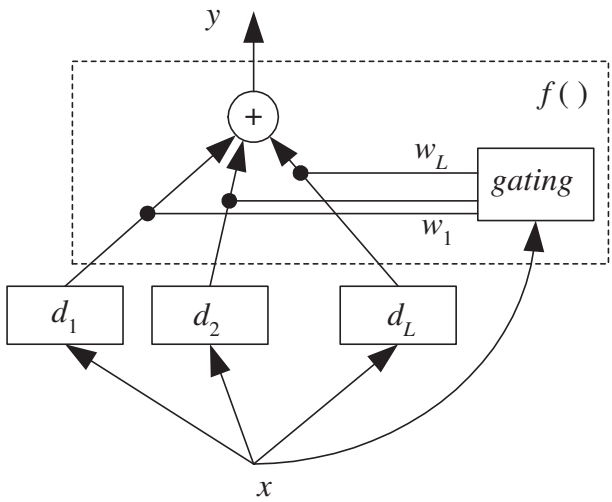


Figure 17.3 Mixture of experts is a voting method where the votes, as given by the gating system, are a function of the input. The combiner system f also includes this gating system.

the competence of base-classifiers in the vicinity of the input. It then picks the most competent to generate output and that output is given as the overall output. Woods, Kegelmeyer, and Bowyer (1997) find the k nearest training points of the test input, look at the accuracies of the base classifiers on those, and choose the one that performs the best on them. Only the selected base-classifier need be evaluated for that test input. To decrease variance, at the expense of more computation, one can take a vote over a few competent base-classifiers instead of using just a single one.

Note that in such a scheme, one should make sure that for any region of the input space, there is a competent base-classifier; this implies that there should be some partitioning of the learning of the input space among the base-classifiers. This is the nice property of mixture of experts, namely, the gating model that does the selection and the expert base-learners that it selects from are trained in a coupled manner. It would be straightforward to have a regression version of this dynamic learner selection algorithm (exercise 5).

17.9 Stacked Generalization

STACKED GENERALIZATION

Stacked generalization is a technique proposed by Wolpert (1992) that extends voting in that the way the output of the base-learners is combined need not be linear but is learned through a combiner system, $f(\cdot|\Phi)$, which is another learner, whose parameters Φ are also trained (see figure 17.4):

$$y = f(d_1, d_2, \ldots, d_L|\Phi) \tag{17.8}$$

The combiner learns what the correct output is when the base-learners give a certain output combination. We cannot train the combiner function on the training data because the base-learners may be memorizing the training set; the combiner system should actually learn how the base-learners make errors. Stacking is a means of estimating and correcting for the biases of the base-learners. Therefore, the combiner should be trained on data unused in training the base-learners.

If $f(\cdot|w_1, \ldots, w_L)$ is a linear model with constraints, $w_i \geq 0, \sum_j w_j = 1$, the optimal weights can be found by constrained regression, but of course we do not need to enforce this; in stacking, there is no restriction on the combiner function and unlike voting, $f(\cdot)$ can be nonlinear. For example, it may be implemented as a multilayer perceptron with Φ its connection weights.

The outputs of the base-learners d_j define a new L-dimensional space in which the output discriminant/regression function is learned by the combiner function.

In stacked generalization, we would like the base-learners to be as different as possible so that they will complement each other, and, for this, it is best if they are based on different learning algorithms. If we are combining classifiers that can generate continuous outputs, for example, posterior probabilities, it is better that they be the combined rather than hard decisions.

When we compare a trained combiner as we have in stacking, with a fixed rule such as in voting, we see that both have their advantages: A trained rule is more flexible and may have less bias, but adds extra parameters, risks introducing variance, and needs extra time and data for training. Note also that there is no need to normalize classifier outputs before stacking.

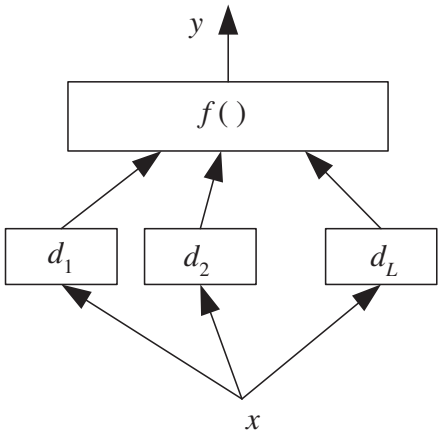


Figure 17.4 In stacked generalization, the combiner is another learner and is not restricted to being a linear combination as in voting.

17.10 Fine-Tuning an Ensemble

Model combination is not a magical formula that is always guaranteed to decrease error; base-learners should be diverse and accurate—that is, they should provide useful information. If a base-learner does not add to accuracy, it can be discarded; also, of the two base-learners that are highly correlated, one is not needed. Note that an inaccurate learner can worsen accuracy, for example, majority voting assumes more than half of the classifiers to be accurate for an input. Therefore, given a set of candidate base-learners, it may not be a good idea to use them as they are, and instead, we may want to do some preprocessing.

We can actually think of the outputs of our base-learners as forming a feature vector for the later stage of combination, and we remember from chapter 6 that we have the same problem with features. Some may be just useless, and some may be highly correlated. Hence, we can use the same ideas of feature selection and extraction here too. Our first approach is to select a subset from the set of base-learners, keeping some and discarding the rest, and the second approach is to define few, new, uncorrelated metalearners from the original base-learners.

17.10.1 Choosing a Subset of the Ensemble

ENSEMBLE SELECTION

Choosing a subset from an ensemble of base-learners is similar to input feature selection, and the possible approaches for *ensemble selection* are the same. We can have a forward/incremental/growing approach where at each iteration, from a set of candidate base-learners, we add to the ensemble the one that most improves accuracy, we can have a backward/decremental/pruning approach where at each iteration, we remove the base-learner whose absence leads to highest improvement, or we can have a floating approach where both additions and removals are allowed.

The combination scheme can be a fixed rule, such as voting, or it can be a trained stacker. Such a selection scheme would not include inaccurate learners, ones that are not diverse enough or are correlated (Caruana et al. 2004; Ruta and Gabrys 2005). So discarding the useless also decreases the overall complexity. Different learners may be using different representations, and such an approach also allows choosing the best complementary representations (Demir and Alpaydın 2005). Note that if we use a decision tree as the combiner, it acts both as a selector and a combiner (Ulaş et al. 2009).

17.10.2 Constructing Metalearners

No matter how we vary the learning algorithms, hyperparameters, resampled folds, or input features, we get positively correlated classifiers (Ulaş, Yıldız, and Alpaydın 2012), and postprocessing is needed to remove this correlation that may be harmful. One possibility is to discard some of the correlated ones, as we discussed earlier; another is to apply a feature extraction method where from the space of the outputs of base-learners, we go to a new, lower-dimensional space where we define uncorrelated metalearners that will also be fewer in number.

Merz (1999) proposes the SCANN algorithm that uses correspondence analysis—a variant of principal components analysis (section 6.3)—on the crisp outputs of base classifiers and combines them using the nearest mean classifier. Actually, any linear or nonlinear feature extraction method we discussed in chapter 6 can be used and its (preferrably continuous) output can be fed to any learner, as we do in stacking.

Let us say we have L learners each having K outputs. Then, for example, using principal component analysis, we can map from the $K \cdot L$-dimensional space to a new space of lower-dimensional, uncorrelated

space of "eigenlearners" (Ulaş, Yıldız, and Alpaydın 2012). We can then train the combiner in this new space (using a separate dataset unused to train the base-learners and the dimensionality reducer). Actually, by looking at the coefficients of the eigenvectors, we can also understand the contribution of the base-learners and assess their utility.

It has been shown by Jacobs (1995) that L dependent learners are worth the same as L' independent learners where $L' \leq L$, and this is exactly the idea here. Another point to note is that rather than drastically discarding or keeping a subset of the ensemble, this approach uses all the base-learners, and hence all the information, but at the expense of more computation.

17.11 Cascading

The idea in cascaded classifiers is to have a *sequence* of base-classifiers d_j sorted in terms of their space or time complexity, or the cost of the representation they use, so that d_{j+1} is costlier than d_j (Kaynak and Alpaydın 2000). *Cascading* is a multistage method, and we use d_j only if all preceding learners, $d_k, k < j$ are not confident (see figure 17.5). For this, associated with each learner is a *confidence* w_j such that we say d_j is confident of its output and can be used if $w_j > \theta_j$ where $1/K < \theta_j \leq \theta_{j+1} < 1$ is the confidence threshold. In classification, the confidence function is set to the highest posterior: $w_j \equiv \max_i d_{ji}$; this is the strategy used for rejections (section 3.3).

CASCADING

We use learner d_j if all the preceding learners are not confident:

$$y_i = d_{ji} \text{ if } w_j > \theta_j \text{ and } \forall k < j, w_k < \theta_k \tag{17.9}$$

Starting with $j = 1$, given a training set, we train d_j. Then we find all instances from a separate validation set on which d_j is not confident, and these constitute the training set of d_{j+1}. Note that unlike in AdaBoost, we choose not only the misclassified instances but the ones for which the previous base-learner is not confident. This covers the misclassifications as well as the instances for which the posterior is not high enough; these are instances on the right side of the boundary but for which the distance to the discriminant, namely, the margin, is not large enough.

The idea is that an early simple classifier handles the majority of instances, and a more complex classifier is used only for a small percentage, thereby not significantly increasing the overall complexity. This is

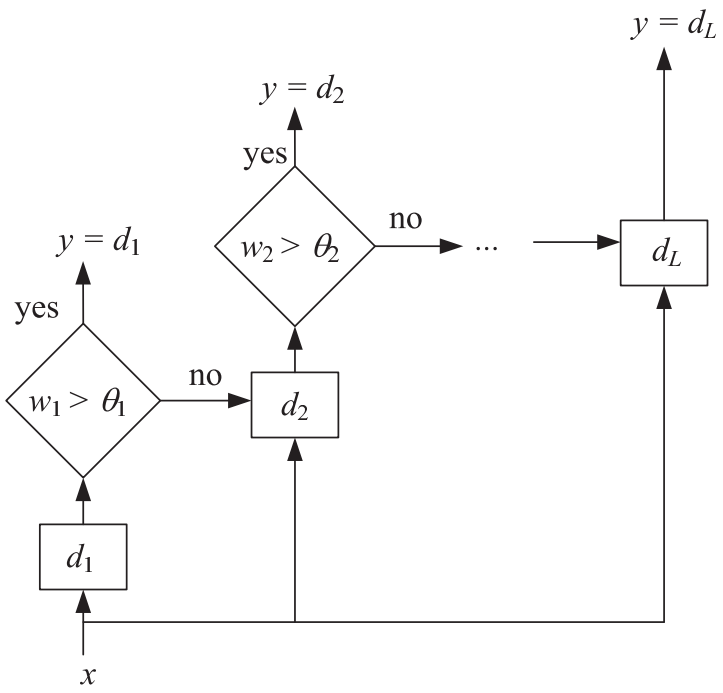


Figure 17.5 Cascading is a multistage method where there is a sequence of classifiers, and the next one is used only when the preceding ones are not confident.

contrary to the multiexpert methods like voting where all base-learners generate their output for any instance. If the problem space is complex, a few base-classifiers may be cascaded increasing the complexity at each stage. In order not to increase the number of base-classifiers, the few instances not covered by any are stored as they are and are treated by a nonparametric classifier, such as k-NN.

The inductive bias of cascading is that the classes can be explained by a small number of "rules" in increasing complexity, with an additional small set of "exceptions" not covered by the rules. The rules are implemented by simple base-classifiers, for example, perceptrons of increasing complexity, which learn general rules valid over the whole input space. Exceptions are localized instances and are best handled by a nonparametric model.

Cascading thus stands between the two extremes of parametric and

nonparametric classification. The former—for example, a linear model—finds a single rule that should cover all the instances. A nonparametric classifier—for example, k-NN—stores the whole set of instances without generating any simple rule explaining them. Cascading generates a rule (or rules) to explain a large part of the instances as cheaply as possible and stores the rest as exceptions. This makes sense in a lot of learning applications. For example, most of the time the past tense of a verb in English is found by adding a "–d" or "–ed" to the verb; there are also irregular verbs—for example, "go"/"went"—that do not obey this rule.

17.12 Notes

The idea in combining learners is to divide a complex task into simpler tasks that are handled by separately trained base-learners. Each base-learner has its own task. If we had a large learner containing all the base-learners, then it would risk overfitting. For example, consider taking a vote over three multilayer perceptrons, each with a single hidden layer. If we combine them all together with the linear model combining their outputs, this is a large multilayer perceptron with two hidden layers. If we train this large model with the whole sample, it very probably overfits. When we train the three multilayer perceptrons separately, for example, using ECOC, bagging, and so forth, it is as if we define a required output for the second-layer hidden nodes of the large multilayer perceptron. This puts a constraint on what the overall learner should learn and simplifies learning.

One disadvantage of combining is that the combined system is not interpretable. For example, even though decision trees are interpretable, bagged or boosted trees are not interpretable. Error-correcting codes with their weights as $-1/0/+1$ allow some form of interpretability. Mayoraz and Moreira (1997) discuss incremental methods for learning the error-correcting output codes where base-learners are added when needed. Allwein, Schapire, and Singer (2000) discuss various methods for coding multiclass problems as two-class problems. Alpaydın and Mayoraz (1999) consider the application of ECOC where linear base-learners are combined to get nonlinear discriminants, and they also propose methods to learn the ECOC matrix from data.

The earliest and most intuitive approach is voting. Kittler et al. (1998) give a review of fixed rules and also discuss an application where multi-

ple representations are combined. The task is person identification using three representations: frontal face image, face profile image, and voice. The error rate of the voting model is lower than the error rates when a single representation is used. Another application is given in Alimoğlu and Alpaydın 1997 where for improved handwritten digit recognition, two sources of information are combined: One is the temporal pen movement data as the digit is written on a touch-sensitive pad, and the other is the static two-dimensional bitmap image once the digit is written. In that application, the two classifiers using either of the two representations have around 5 percent error, but combining the two reduces the error rate to 3 percent. It is also seen that the critical stage is the design of the complementary learners and/or representations, the way they are combined is not as critical.

BIOMETRICS

Combining different modalities is used in *biometrics*, where the aim is authentication using different input sources, fingerprint, signature, face, and so on. In such a case, different classifiers use these modalities separately and their decisions are combined. This both improves accuracy and makes *spoofing* more difficult.

Noble (2004) makes a distinction between three type of combination strategies when we have information coming from multiple sources in different representations or modalities:

- In *early integration*, all these inputs are concatenated to form a single vector that is then fed to a single classifier. Previously we discussed why this is not a very good idea.
- In *late integration*, which we advocated in this chapter, different inputs are fed to separate classifiers whose outputs are then combined, by voting, stacking, or any other method we discussed.
- Kernel algorithms, which we discussed in chapter 13, allow a different method of integration that Noble (2004) calls *intermediate integration*, as being between early and late integration. This is the *multiple kernel learning* approach (see section 13.8) where there is a single kernel machine classifier that uses multiple kernels for different inputs and the combination is not in the input space as in early integration, or in the space of decisions as in late integration, but in the space of the basis functions that define the kernels. For different sources, there are different notions of similarity calculated by their kernels, and the classifier accumulates and uses them.

MULTIPLE KERNEL LEARNING

Some ensemble methods such as voting are similar to Bayesian averaging (chapter 16). For example when we do bagging and train the same model on different resampled training sets, we may consider them as being samples from the posterior distribution, but other combination methods such as mixture of experts and stacking go much beyond averaging over parameters or models.

When we are combining multiple views/representations, concatenating them is not really a good idea but one interesting possibility is to do some combined dimensionality reduction. We can consider a generative model (section 14.3) where we assume that there is a set of latent factors that generate these multiple views in parallel, and from the observed views, we can go back to that latent space and do classification there (Chen et al. 2012).

Combining multiple learners has been a popular topic in machine learning since the early 1990s, and research has been going on ever since. Kuncheva (2004) discusses different aspects of classifier combination; the book also includes a section on combination of multiple clustering results.

AdaBoosted decision trees are considered to be one of the best machine learning algorithms. There are also versions of AdaBoost where the next base-learner is trained on the residual of the previous base-learner (Hastie, Tibshirani, and Friedman 2001). Recently, it has been noticed that ensembles do not always improve accuracy and research has started to focus on the criteria that a good ensemble should satisfy or how to form a good one. A survey of the role of diversity in ensembles is given in Kuncheva 2005.

17.13 Exercises

1. If each base-learner is iid and correct with probability $p > 1/2$, what is the probability that a majority vote over L classifiers gives the correct answer?

   SOLUTION: It is given by a binomial distribution (see figure 17.6).

   $$P(X \geq \lfloor L/2 \rfloor + 1) = \sum_{i=\lfloor L/2 \rfloor + 1}^{L} \binom{L}{i} p^i (1-p)^{L-i}$$

2. In bagging, to generate the L training sets, what would be the effect of using L-fold cross-validation instead of bootstrap?

3. Propose an incremental algorithm for learning error-correcting output codes

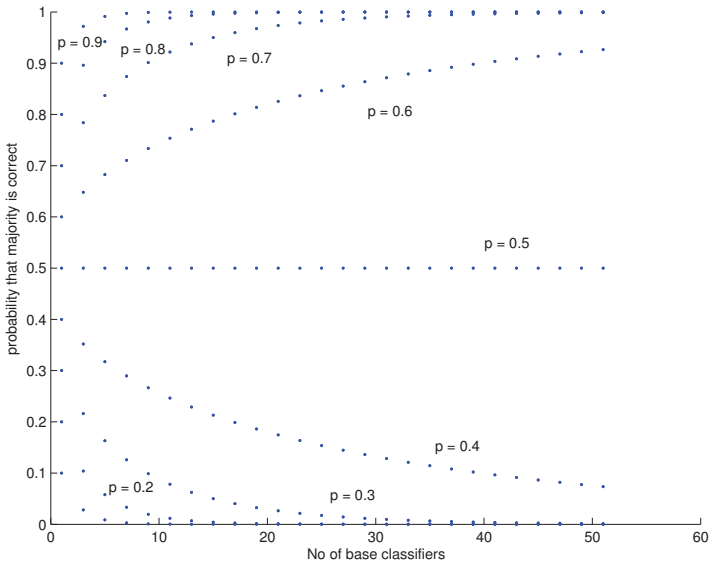


Figure 17.6 Probability that a majority vote is correct as a function of the number of base-learners for different p. The probabity increases only for $p > 0.5$.

where new two-class problems are added as they are needed to better solve the multiclass problem.

4. In the mixture of experts architecture, we can have different experts use different input representations. How can we design the gating network in such a case?

5. Propose a dynamic regressor selection algorithm.

6. What is the difference between voting and stacking using a linear perceptron as the combiner function?

   SOLUTION: If the voting system is also trained, the only difference would be that with stacking, the weights need not be positive or sum up to 1, and there is also a bias term. Of course, the main advantage of stacking is when the combiner is nonlinear.

7. In cascading, why do we require $\theta_{j+1} \geq \theta_j$?

   SOLUTION: Instances on which the confidence is less than θ_j have already been filtered out by d_j; we require the threshold to increase so that we can have higher confidences.

8. To be able to use cascading for regression, during testing, a regressor should

be able to say whether it is confident of its output. How can we implement this?

9. How can we combine the results of multiple clustering solutions?

   SOLUTION: The easiest is the following: Let us take any two training instances. Each clustering solution either places them in the same cluster or not; denote it as 1 and 0. The average of these counts over all clustering solutions is the overall probability that those two are in the same cluster (Kuncheva 2004).

10. In section 17.10, we discuss that if we use a decision tree as a combiner in stacking, it works both as a selector and a combiner. What are the other advantages and disadvantages?

    SOLUTION: A tree uses only a subset of the classifiers and not the whole. Using the tree is fast, and we need to evaluate only the nodes on our path, which may be short. See Ulaş et al. 2009 for more detail. The disadvantage is that the combiner cannot look at combinations of classifier decisions (assuming that the tree is univariate). Using a subset may also be harmful; we do not get the redundancy we need if some classifiers are faulty.

17.14 References

Alimoğlu, F., and E. Alpaydın. 1997. "Combining Multiple Representations and Classifiers for Pen-Based Handwritten Digit Recognition." In *Fourth International Conference on Document Analysis and Recognition*, 637–640. Los Alamitos, CA: IEEE Computer Society.

Allwein, E. L., R. E. Schapire, and Y. Singer. 2000. "Reducing Multiclass to Binary: A Unifying Approach for Margin Classifiers." *Journal of Machine Learning Research* 1:113–141.

Alpaydın, E. 1997. "Voting over Multiple Condensed Nearest Neighbors." *Artificial Intelligence Review* 11:115–132.

Alpaydın, E., and E. Mayoraz. 1999. "Learning Error-Correcting Output Codes from Data." In *Ninth International Conference on Artificial Neural Networks*, 743–748. London: IEE Press.

Avnimelech, R., and N. Intrator. 1997. "Boosting Regression Estimators." *Neural Computation* 11:499–520.

Breiman, L. 1996. "Bagging Predictors." *Machine Learning* 26:123–140.

Caruana, R., A. Niculescu-Mizil, G. Crew, and A. Ksikes. 2004. "Ensemble Selection from Libraries of Models." In *Twenty-First International Conference on Machine Learning*, ed. C. E. Brodley, 137–144. New York: ACM.

Chen, N., J. Zhu, F. Sun, and E. P. Xing. 2012. "Large-Margin Predictive Latent Subspace Learning for Multiview Data Analysis." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34:2365–2378.

Demir, C., and E. Alpaydın. 2005. "Cost-Conscious Classifier Ensembles." *Pattern Recognition Letters* 26:2206–2214.

Dietterich, T. G., and G. Bakiri. 1995. "Solving Multiclass Learning Problems via Error-Correcting Output Codes." *Journal of Artificial Intelligence Research* 2:263–286.

Drucker, H. 1997. "Improving Regressors using Boosting Techniques." In *Fourteenth International Conference on Machine Learning*, ed. D. H. Fisher, 107–115. San Mateo, CA: Morgan Kaufmann.

Drucker, H., C. Cortes, L. D. Jackel, Y. Le Cun, and V. Vapnik. 1994. "Boosting and Other Ensemble Methods." *Neural Computation* 6:1289–1301.

Freund, Y., and R. E. Schapire. 1996. "Experiments with a New Boosting Algorithm." In *Thirteenth International Conference on Machine Learning*, ed. L. Saitta, 148–156. San Mateo, CA: Morgan Kaufmann.

Hansen, L. K., and P. Salamon. 1990. "Neural Network Ensembles." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12:993–1001.

Hastie, T., R. Tibshirani, and J. Friedman. 2001. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer.

Ho, T. K. 1998. "The Random Subspace Method for Constructing Decision Forests." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20:832–844.

Jacobs, R. A. 1995. "Methods for Combining Experts' Probability Assessments." *Neural Computation* 7:867–888.

Jacobs, R. A. 1997. "Bias/Variance Analyses for Mixtures-of-Experts Architectures." *Neural Computation* 9:369–383.

Jain, A., K. Nandakumar, and A. Ross. 2005. "Score Normalization in Multimodal Biometric Systems." *Pattern Recognition* 38:2270–2285.

Kaynak, C., and E. Alpaydın. 2000. "MultiStage Cascading of Multiple Classifiers: One Man's Noise is Another Man's Data." In *Seventeenth International Conference on Machine Learning*, ed. P. Langley, 455–462. San Francisco: Morgan Kaufmann.

Kittler, J., M. Hatef, R. P. W. Duin, and J. Matas. 1998. "On Combining Classifiers." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20:226–239.

Kuncheva, L. I. 2004. *Combining Pattern Classifiers: Methods and Algorithms*. Hoboken, NJ: Wiley.

Kuncheva, L. I. 2005. Special issue on Diversity in Multiple Classifier Systems. *Information Fusion* 6:1–115.

Mayoraz, E., and M. Moreira. 1997. "On the Decomposition of Polychotomies into Dichotomies." In *Fourteenth International Conference on Machine Learning*, ed. D. H. Fisher, 219–226. San Mateo, CA: Morgan Kaufmann.

Merz, C. J. 1999. "Using Correspondence Analysis to Combine Classifiers." *Machine Learning* 36:33–58.

Noble, W. S. 2004. "Support Vector Machine Applications in Computational Biology." In *Kernel Methods in Computational Biology*, ed. B. Schölkopf, K. Tsuda, and J.-P. Vert, 71–92. Cambridge, MA: MIT Press.

Özen, A., M. Gönen, E. Alpaydın, and T. Haliloğlu. 2009. "Machine Learning Integration for Predicting the Effect of Single Amino Acid Substitutions on Protein Stability." *BMC Structural Biology* 9 (66): 1–17.

Perrone, M. P. 1993. "Improving Regression Estimation: Averaging Methods for Variance Reduction with Extensions to General Convex Measure." Ph.D. thesis, Brown University.

Ruta, D., and B. Gabrys. 2005. "Classifier Selection for Majority Voting." *Information Fusion* 6:63–81.

Schapire, R. E. 1990. "The Strength of Weak Learnability." *Machine Learning* 5:197–227.

Schapire, R. E., Y. Freund, P. Bartlett, and W. S. Lee. 1998. "Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods." *Annals of Statistics* 26:1651–1686.

Ulaş, A., M. Semerci, O. T. Yıldız, and E. Alpaydın. 2009. "Incremental Construction of Classifier and Discriminant Ensembles." *Information Sciences* 179:1298–1318.

Ulaş, A., O. T. Yıldız, and E. Alpaydın. 2012. "Eigenclassifiers for Combining Correlated Classifiers." *Information Sciences* 187:109–120.

Wolpert, D. H. 1992. "Stacked Generalization." *Neural Networks* 5:241–259.

Woods, K., W. P. Kegelmeyer Jr., and K. Bowyer. 1997. "Combination of Multiple Classifiers Using Local Accuracy Estimates." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19:405–410.

18 *Reinforcement Learning*

In reinforcement learning, the learner is a decision-making agent that takes actions in an environment and receives reward (or penalty) for its actions in trying to solve a problem. After a set of trial-and-error runs, it should learn the best policy, which is the sequence of actions that maximize the total reward.

18.1 Introduction

LET US SAY we want to build a machine that learns to play chess. In this case we cannot use a supervised learner for two reasons. First, it is very costly to have a teacher that will take us through many games and indicate us the best move for each position. Second, in many cases, there is no such thing as the best move; the goodness of a move depends on the moves that follow. A single move does not count; a sequence of moves is good if after playing them we win the game. The only feedback is at the end of the game when we win or lose the game.

Another example is a robot that is placed in a maze. The robot can move in one of the four compass directions and should make a sequence of movements to reach the exit. As long as the robot is in the maze, there is no feedback and the robot tries many moves until it reaches the exit and only then does it get a reward. In this case there is no opponent, but we can have a preference for shorter trajectories, implying that in this case we play against time.

These two applications have a number of points in common: There is a decision maker, called the *agent*, that is placed in an *environment* (see figure 18.1). In chess, the game-player is the decision maker and the environment is the board; in the second case, the maze is the environment

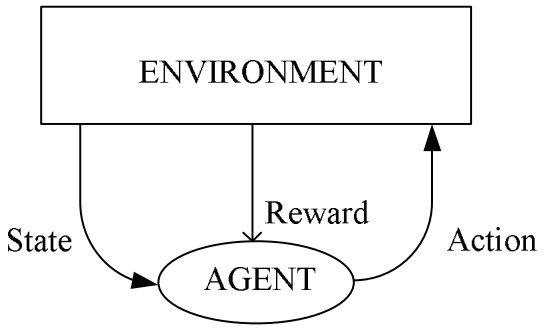


Figure 18.1 The agent interacts with an environment. At any state of the environment, the agent takes an action that changes the state and returns a reward.

of the robot. At any time, the environment is in a certain *state* that is one of a set of possible states—for example, the state of the board, the position of the robot in the maze. The decision maker has a set of *actions* possible: legal movement of pieces on the chess board, movement of the robot in possible directions without hitting the walls, and so forth. Once an action is chosen and taken, the state changes. The solution to the task requires a sequence of actions, and we get feedback, in the form of a *reward* rarely, generally only when the complete sequence is carried out. The reward defines the problem and is necessary if we want a *learning* agent. The learning agent learns the best sequence of actions to solve a problem where "best" is quantified as the sequence of actions that has the maximum cumulative reward. Such is the setting of *reinforcement learning*.

Reinforcement learning is different from the learning methods we discussed before in a number of respects. It is called "learning with a critic," as opposed to learning with a teacher which we have in supervised learning. A *critic* differs from a teacher in that it does not tell us what to do but only how well we have been doing in the past; the critic never informs in advance. The feedback from the critic is scarce and when it comes, it comes late. This leads to the *credit assignment* problem. After taking several actions and getting the reward, we would like to assess the individual actions we did in the past and find the moves that led us to win the reward so that we can record and recall them later on. As we see shortly, what a reinforcement learning program does is that it learns to generate

CRITIC

CREDIT ASSIGNMENT

an *internal value* for the intermediate states or actions in terms of how good they are in leading us to the goal and getting us to the real reward. Once such an internal reward mechanism is learned, the agent can just take the local actions to maximize it.

The solution to the task requires a *sequence* of actions, and from this perspective, we remember the Markov models we discussed in chapter 15. Indeed, we use a Markov decision process to model the agent. The difference is that in the case of Markov models, there is an external process that generates a sequence of signals, for example, speech, which we observe and model. In the current case, however, it is the agent that generates the sequence of actions. Previously, we also made a distinction between observable and hidden Markov models where the states are observed or hidden (and should be inferred) respectively. Similarly here, sometimes we have a partially observable Markov decision process in cases where the agent does not know its state exactly but should infer it with some uncertainty through observations using sensors. For example, in the case of a robot moving in a room, the robot may not know its exact position in the room, nor the exact location of obstacles nor the goal, and should make decisions through a limited image provided by a camera.

18.2 Single State Case: *K*-Armed Bandit

K-ARMED BANDIT

We start with a simple example. The K-*armed bandit* is a hypothetical slot machine with K levers. The action is to choose and pull one of the levers, and we win a certain amount of money that is the reward associated with the lever (action). The task is to decide which lever to pull to maximize the reward. This is a classification problem where we choose one of K. If this were supervised learning, then the teacher would tell us the correct class, namely, the lever leading to maximum earning. In this case of reinforcement learning, we can only try different levers and keep track of the best. This is a simplified reinforcement learning problem because there is only one state, or one slot machine, and we need only decide on the action. Another reason why this is simplified is that we immediately get a reward after a single action; the reward is not delayed, so we immediately see the value of our action.

Let us say $Q(a)$ is the value of action a. Initially, $Q(a) = 0$ for all a. When we try action a, we get reward $r_a \geq 0$. If rewards are deterministic, we always get the same r_a for any pull of a and in such a case, we can

just set $Q(a) = r_a$. If we want to exploit, once we find an action a such that $Q(a) > 0$, we can keep choosing it and get r_a at each pull. However, it is quite possible that there is another lever with a higher reward, so we need to explore.

We can choose different actions and store $Q(a)$ for all a. Whenever we want to exploit, we can choose the action with the maximum value, that is,

$$\text{choose } a^* \text{ if } Q(a^*) = \max_a Q(a) \tag{18.1}$$

If rewards are not deterministic but stochastic, we get a different reward each time we choose the same action. The amount of the reward is defined by the probability distribution $p(r|a)$. In such a case, we define $Q_t(a)$ as the estimate of the value of action a at time t. It is an average of all rewards received when action a was chosen before time t. An online update can be defined as

$$Q_{t+1}(a) \leftarrow Q_t(a) + \eta[r_{t+1}(a) - Q_t(a)] \tag{18.2}$$

where $r_{t+1}(a)$ is the reward received after taking action a at time $(t+1)$st time.

Note that equation 18.2 is the *delta rule* that we have used on many occasions in the previous chapters: η is the learning factor (gradually decreased in time for convergence), r_{t+1} is the desired output, and $Q_t(a)$ is the current prediction. $Q_{t+1}(a)$ is the *expected* value of action a at time $t+1$ and converges to the mean of $p(r|a)$ as t increases.

The full reinforcement learning problem generalizes this simple case in a number of ways. First, we have several states. This corresponds to having several slot machines with different reward probabilities, $p(r|s_i, a_j)$, and we need to learn $Q(s_i, a_j)$, which is the value of taking action a_j when in state s_i. Second, the actions affect not only the reward but also the next state, and we move from one state to another. Third, the rewards are delayed and we need to be able to estimate immediate values from delayed rewards.

18.3 Elements of Reinforcement Learning

The learning decision maker is called the *agent*. The agent interacts with the *environment* that includes everything outside the agent. The agent has sensors to decide on its *state* in the environment and takes an *action*

that modifies its state. When the agent takes an action, the environment provides a *reward*. Time is discrete as $t = 0, 1, 2, \ldots$, and $s_t \in S$ denotes the state of the agent at time t where S is the set of all possible states. $a_t \in \mathcal{A}(s_t)$ denotes the action that the agent takes at time t where $\mathcal{A}(s_t)$ is the set of possible actions in state s_t. When the agent in state s_t takes the action a_t, the clock ticks, reward $r_{t+1} \in \Re$ is received, and the agent moves to the next state, s_{t+1}. The problem is modeled using a *Markov decision process* (MDP). The reward and next state are sampled from their respective probability distributions, $p(r_{t+1}|s_t, a_t)$ and $P(s_{t+1}|s_t, a_t)$. Note that what we have is a *Markov* system where the state and reward in the next time step depend only on the current state and action. In some applications, reward and next state are deterministic, and for a certain state and action taken, there is one possible reward value and next state.

MARKOV DECISION PROCESS

Depending on the application, a certain state may be designated as the initial state and in some applications, there is also an absorbing terminal (goal) state where the search ends; all actions in this terminal state transition to itself with probability 1 and without any reward. The sequence of actions from the start to the terminal state is an *episode*, or a *trial*.

EPISODE

The *policy*, π, defines the agent's behavior and is a mapping from the states of the environment to actions: $\pi : S \to \mathcal{A}$. The policy defines the action to be taken in any state s_t: $a_t = \pi(s_t)$. The *value* of a policy π, $V^\pi(s_t)$, is the expected cumulative reward that will be received while the agent follows the policy, starting from state s_t.

POLICY

In the *finite-horizon* or *episodic* model, the agent tries to maximize the expected reward for the next T steps:

FINITE-HORIZON

$$V^\pi(s_t) = E[r_{t+1} + r_{t+2} + \cdots + r_{t+T}] = E\left[\sum_{i=1}^{T} r_{t+i}\right] \tag{18.3}$$

Certain tasks are continuing, and there is no prior fixed limit to the episode. In the *infinite-horizon* model, there is no sequence limit, but future rewards are discounted:

INFINITE-HORIZON

$$V^\pi(s_t) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots] = E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right] \tag{18.4}$$

where $0 \le \gamma < 1$ is the *discount rate* to keep the return finite. If $\gamma = 0$, then only the immediate reward counts. As γ approaches 1, rewards further in the future count more, and we say that the agent becomes more farsighted. γ is less than 1 because there generally is a time limit

DISCOUNT RATE

to the sequence of actions needed to solve the task. The agent may be a robot that runs on a battery. We prefer rewards sooner rather than later because we are not certain how long we will survive.

OPTIMAL POLICY

For each policy π, there is a $V^{\pi}(s_t)$, and we want to find the *optimal policy* π^* such that

$$V^*(s_t) = \max_{\pi} V^{\pi}(s_t), \forall s_t \tag{18.5}$$

In some applications, for example, in control, instead of working with the values of states, $V(s_t)$, we prefer to work with the values of state-action pairs, $Q(s_t, a_t)$. $V(s_t)$ denotes how good it is for the agent to be in state s_t, whereas $Q(s_t, a_t)$ denotes how good it is to perform action a_t when in state s_t. We define $Q^*(s_t, a_t)$ as the value, that is, the expected cumulative reward, of action a_t taken in state s_t and then obeying the optimal policy afterward. The value of a state is equal to the value of the best possible action:

$$\begin{aligned} V^*(s_t) &= \max_{a_t} Q^*(s_t, a_t) \\ &= \max_{a_t} E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right] \\ &= \max_{a_t} E\left[r_{t+1} + \gamma \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i+1}\right] \\ &= \max_{a_t} E\left[r_{t+1} + \gamma V^*(s_{t+1})\right] \end{aligned}$$

$$V^*(s_t) = \max_{a_t}\left(E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) V^*(s_{t+1})\right) \tag{18.6}$$

To each possible next state s_{t+1}, we move with probability $P(s_{t+1}|s_t, a_t)$, and continuing from there using the optimal policy, the expected cumulative reward is $V^*(s_{t+1})$. We sum over all such possible next states, and we discount it because it is one time step later. Adding our immediate expected reward, we get the total expected cumulative reward for action a_t. We then choose the best of possible actions. Equation 18.6 is known as *Bellman's equation* (Bellman 1957). Similarly, we can also write

BELLMAN'S EQUATION

$$Q^*(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \tag{18.7}$$

```
Initialize V(s) to arbitrary values
Repeat
    For all s ∈ S
        For all a ∈ A
            Q(s, a) ← E[r|s, a] + γ Σ_{s'∈S} P(s'|s, a)V(s')
        V(s) ← max_a Q(s, a)
Until V(s) converge
```

Figure 18.2 Value iteration algorithm for model-based learning.

Once we have $Q^*(s_t, a_t)$ values, we can then define our policy π as taking the action a_t^*, which has the highest value among all $Q^*(s_t, a_t)$:

$$\pi^*(s_t) : \text{Choose } a_t^* \text{ where } Q^*(s_t, a_t^*) = \max_{a_t} Q^*(s_t, a_t) \tag{18.8}$$

This means that if we have the $Q^*(s_t, a_t)$ values, then by using a greedy search at each *local* step we get the optimal sequence of steps that maximizes the *cumulative* reward.

18.4 Model-Based Learning

We start with model-based learning where we completely know the environment model parameters, $p(r_{t+1}|s_t, a_t)$ and $P(s_{t+1}|s_t, a_t)$. In such a case, we do not need any exploration and can directly solve for the optimal value function and policy using dynamic programming. The optimal value function is unique and is the solution to the simultaneous equations given in equation 18.6. Once we have the optimal value function, the optimal policy is to choose the action that maximizes the value in the next state:

$$\pi^*(s_t) = \arg\max_{a_t} \left(E[r_{t+1}|s_t, a_t] + \gamma \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t) V^*(s_t + 1) \right) \tag{18.9}$$

18.4.1 Value Iteration

VALUE ITERATION

To find the optimal policy, we can use the optimal value function, and there is an iterative algorithm called *value iteration* that has been shown to converge to the correct V^* values. Its pseudocode is given in figure 18.2.

Initialize a policy π' arbitrarily
Repeat
    $\pi \leftarrow \pi'$
    Compute the values using π by
        solving the linear equations
            $V^{\pi}(s) = E[r|s, \pi(s)] + \gamma \sum_{s' \in S} P(s'|s, \pi(s)) V^{\pi}(s')$
    Improve the policy at each state
        $\pi'(s) \leftarrow \arg\max_a (E[r|s, a] + \gamma \sum_{s' \in S} P(s'|s, a) V^{\pi}(s'))$
Until $\pi = \pi'$

Figure 18.3 Policy iteration algorithm for model-based learning.

We say that the values converged if the maximum value difference between two iterations is less than a certain threshold δ:

$$\max_{s \in S} |V^{(l+1)}(s) - V^{(l)}(s)| < \delta$$

where l is the iteration counter. Because we care only about the actions with the maximum value, it is possible that the policy converges to the optimal one even before the values converge to their optimal values. Each iteration is $\mathcal{O}(|S|^2|\mathcal{A}|)$, but frequently there is only a small number $k < |S|$ of next possible states, so complexity decreases to $\mathcal{O}(k|S||\mathcal{A}|)$.

18.4.2 Policy Iteration

In policy iteration, we store and update the policy rather than doing this indirectly over the values. The pseudocode is given in figure 18.3. The idea is to start with a policy and improve it repeatedly until there is no change. The value function can be calculated by solving for the linear equations. We then check whether we can improve the policy by taking these into account. This step is guaranteed to improve the policy, and when no improvement is possible, the policy is guaranteed to be optimal. Each iteration of this algorithm takes $\mathcal{O}(|\mathcal{A}||S|^2 + |S|^3)$ time that is more than that of value iteration, but policy iteration needs fewer iterations than value iteration.

18.5 Temporal Difference Learning

Model is defined by the reward and next state probability distributions, and as we saw in section 18.4, when we know these, we can solve for the optimal policy using dynamic programming. However, these methods are costly, and we seldom have such perfect knowledge of the environment. The more interesting and realistic application of reinforcement learning is when we do not have the model. This requires exploration of the environment to query the model. We first discuss how this exploration is done and later see model-free learning algorithms for deterministic and nondeterministic cases. Though we are not going to assume a full knowledge of the environment model, we will however require that it be stationary.

TEMPORAL DIFFERENCE

As we will see shortly, when we explore and get to see the value of the next state and reward, we use this information to update the value of the current state. These algorithms are called *temporal difference* algorithms because what we do is look at the difference between our current estimate of the value of a state (or a state-action pair) and the discounted value of the next state and the reward received.

18.5.1 Exploration Strategies

To explore, one possibility is to use ϵ-*greedy* search where with probability ϵ, we choose one action uniformly randomly among all possible actions, namely, explore, and with probability $1-\epsilon$, we choose the best action, namely, exploit. We do not want to continue exploring indefinitely but start exploiting once we do enough exploration; for this, we start with a high ϵ value and gradually decrease it. We need to make sure that our policy is *soft*, that is, the probability of choosing any action $a \in \mathcal{A}$ in state $s \in S$ is greater than 0.

We can choose probabilistically, using the softmax function to convert values to probabilities

$$P(a|s) = \frac{\exp Q(s,a)}{\sum_{b\in\mathcal{A}} \exp Q(s,b)} \tag{18.10}$$

and then sample according to these probabilities. To gradually move from exploration to exploitation, we can use a "temperature" variable T and define the probability of choosing action a as

$$P(a|s) = \frac{\exp[Q(s,a)/T]}{\sum_{b\in\mathcal{A}} \exp[Q(s,b)/T]} \tag{18.11}$$

When T is large, all probabilities are equal and we have exploration. When T is small, better actions are favored. So the strategy is to start with a large T and decrease it gradually, a procedure named *annealing*, which in this case moves from exploration to exploitation smoothly in time.

18.5.2 Deterministic Rewards and Actions

In model-free learning, we first discuss the simpler deterministic case, where at any state-action pair, there is a single reward and next state possible. In this case, equation 18.7 reduces to

$$Q(s_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{18.12}$$

and we simply use this as an assignment to update $Q(s_t, a_t)$. When in state s_t, we choose action a_t by one of the stochastic strategies we saw earlier, which returns a reward r_{t+1} and takes us to state s_{t+1}. We then update the value of *previous* action as

$$\hat{Q}(s_t, a_t) \leftarrow r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) \tag{18.13}$$

where the hat denotes that the value is an estimate. $\hat{Q}(s_{t+1}, a_{t+1})$ is a later value and has a higher chance of being correct. We discount this by γ and add the immediate reward (if any) and take this as the new estimate for the previous $\hat{Q}(s_t, a_t)$. This is called a *backup* because it can be viewed as taking the estimated value of an action in the next time step and "backing it up" to revise the estimate for the value of a current action.

BACKUP

For now we assume that all $\hat{Q}(s, a)$ values are stored in a table; we will see later on how we can store this information more succinctly when $|S|$ and $|\mathcal{A}|$ are large.

Initially all $\hat{Q}(s_t, a_t)$ are 0, and they are updated in time as a result of trial episodes. Let us say we have a sequence of moves and at each move, we use equation 18.13 to update the estimate of the Q value of the previous state-action pair using the Q value of the current state-action pair. In the intermediate states, all rewards and therefore values are 0, so no update is done. When we get to the goal state, we get the reward r and then we can update the Q value of the previous state-action pair as γr. As for the preceding state-action pair, its immediate reward is 0 and the contribution from the next state-action pair is discounted by γ because it is one step later. Then in another episode, if we reach this

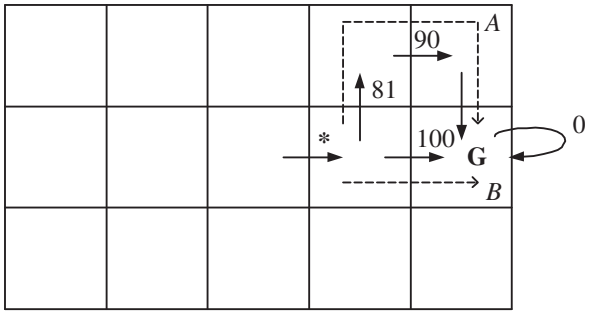


Figure 18.4 Example to show that Q values increase but never decrease. This is a deterministic grid-world where G is the goal state with reward 100, all other immediate rewards are 0, and $\gamma = 0.9$. Let us consider the Q value of the transition marked by asterisk, and let us just consider only the two paths A and B. Let us say that path A is seen before path B, then we have $\gamma \max(0, 81) = 72.9$; if afterward B is seen, a shorter path is found and the Q value becomes $\gamma \max(100, 81) = 90$. If B is seen before A, the Q value is $\gamma \max(100, 0) = 90$; then when A is seen, it does not change because $\gamma \max(100, 81) = 90$.

state, we can update the one preceding that as $\gamma^2 r$, and so on. This way, after many episodes, this information is backed up to earlier state-action pairs. Q values increase until they reach their optimal values as we find paths with higher cumulative reward, for example, shorter paths, but they never decrease (see figure 18.4).

Note that we do not know the reward or next state functions here. They are part of the environment, and it is as if we query them when we explore. We are not modeling them either, though that is another possibility. We just accept them as given and learn directly the optimal policy through the estimated value function.

18.5.3 Nondeterministic Rewards and Actions

If the rewards and the result of actions are not deterministic, then we have a probability distribution for the reward $p(r_{t+1}|s_t, a_t)$ from which rewards are sampled, and there is a probability distribution for the next state $P(s_{t+1}|s_t, a_t)$. These help us model the uncertainty in the system that may be due to forces we cannot control in the environment: for instance, our opponent in chess, the dice in backgammon, or our lack of

```
Initialize all Q(s, a) arbitrarily
For all episodes
    Initalize s
    Repeat
        Choose a using policy derived from Q, e.g., ε-greedy
        Take action a, observe r and s′
        Update Q(s, a):
            Q(s, a) ← Q(s, a) + η(r + γ max_a′ Q(s′, a′) − Q(s, a))
        s ← s′
    Until s is terminal state
```

Figure 18.5 Q learning, which is an off-policy temporal difference algorithm.

knowledge of the system. For example, we may have an imperfect robot which sometimes fails to go in the intended direction and deviates, or advances shorter or longer than expected.

In such a case, we have

$$Q(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{18.14}$$

We cannot do a direct assignment in this case because for the same state and action, we may receive different rewards or move to different next states. What we do is keep a running average. This is known as the Q *learning* algorithm:

Q LEARNING

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \eta(r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{18.15}$$

We think of $r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1})$ values as a sample of instances for each (s_t, a_t) pair and we would like $\hat{Q}(s_t, a_t)$ to converge to its mean. As usual η is gradually decreased in time for convergence, and it has been shown that this algorithm converges to the optimal Q^* values (Watkins and Dayan 1992). The pseudocode of the Q learning algorithm is given in figure 18.5.

We can also think of equation 18.15 as reducing the difference between the current Q value and the backed-up estimate, from one time step later. Such algorithms are called *temporal difference* (TD) algorithms (Sutton 1988).

TEMPORAL DIFFERENCE

This is an *off-policy* method as the value of the best next action is used without using the policy. In an *on-policy* method, the policy is used to

OFF-POLICY

ON-POLICY

```
Initialize all Q(s, a) arbitrarily
For all episodes
    Initalize s
    Choose a using policy derived from Q, e.g., ε-greedy
    Repeat
        Take action a, observe r and s′
        Choose a′ using policy derived from Q, e.g., ε-greedy
        Update Q(s, a):
            Q(s, a) ← Q(s, a) + η(r + γQ(s′, a′) − Q(s, a))
        s ← s′, a ← a′
    Until s is terminal state
```

Figure 18.6 Sarsa algorithm, which is an on-policy version of Q learning.

determine also the next action. The on-policy version of Q learning is the SARSA *Sarsa* algorithm whose pseudocode is given in figure 18.6. We see that instead of looking for all possible next actions a' and choosing the best, the on-policy Sarsa uses the policy derived from Q values to choose one next action a' and uses its Q value to calculate the temporal difference. On-policy methods estimate the value of a policy while using it to take actions. In off-policy methods, these are separated, and the policy used to generate behavior, called the *behavior* policy, may in fact be different from the policy that is evaluated and improved, called the *estimation* policy.

Sarsa converges with probability 1 to the optimal policy and state-action values if a *GLIE policy* is employed to choose actions. A GLIE (greedy in the limit with infinite exploration) policy is where (1) all state-action pairs are visited an infinite number of times, and (2) the policy converges in the limit to the greedy policy (which can be arranged, e.g., with ϵ-greedy policies by setting $\epsilon = 1/t$).

The same idea of temporal difference can also be used to learn $V(s)$ TD LEARNING values, instead of $Q(s,a)$. *TD learning* (Sutton 1988) uses the following update rule to update a state value:

$$V(s_t) \leftarrow V(s_t) + \eta[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{18.16}$$

This again is the delta rule where $r_{t+1} + \gamma V(s_{t+1})$ is the better, later prediction and $V(s_t)$ is the current estimate. Their difference is the temporal difference, and the update is done to decrease this difference. The

update factor η is gradually decreased, and TD is guaranteed to converge to the optimal value function $V^*(s)$.

18.5.4 Eligibility Traces

ELIGIBILITY TRACE

The previous algorithms are one-step—that is, the temporal difference is used to update only the previous value (of the state or state-action pair). An *eligibility trace* is a record of the occurrence of past visits that enables us to implement temporal credit assignment, allowing us to update the values of previously occurring visits as well. We discuss how this is done with Sarsa to learn Q values; adapting this to learn V values is straightforward.

To store the eligibility trace, we require an additional memory variable associated with each state-action pair, $e(s,a)$, initialized to 0. When the state-action pair (s,a) is visited, namely, when we take action a in state s, its eligibility is set to 1; the eligibilities of all other state-action pairs are multiplied by $\gamma\lambda$. $0 \le \lambda \le 1$ is the trace decay parameter.

$$e_t(s,a) = \begin{cases} 1 & \text{if } s = s_t \text{ and } a = a_t, \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise} \end{cases} \tag{18.17}$$

If a state-action pair has never been visited, its eligibility remains 0; if it has been, as time passes and other state-actions are visited, its eligibility decays depending on the value of γ and λ (see figure 18.7).

We remember that in Sarsa, the temporal error at time t is

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{18.18}$$

In Sarsa with an eligibility trace, named Sarsa(λ), *all* state-action pairs are updated as

$$Q(s,a) \leftarrow Q(s,a) + \eta\delta_t e_t(s,a), \ \forall s,a \tag{18.19}$$

This updates all eligible state-action pairs, where the update depends on how far they have occurred in the past. The value of λ defines the temporal credit: If $\lambda = 0$, only a one-step update is done. The algorithms we discussed in section 18.5.3 are such, and for this reason they are named $Q(0)$, Sarsa(0), or TD(0). As λ gets closer to 1, more of the previous steps are considered. When $\lambda = 1$, all previous steps are updated and the credit given to them falls only by γ per step. In online updating, all eligible values are updated immediately after each step; in offline updating, the updates are accumulated and a single update is done at

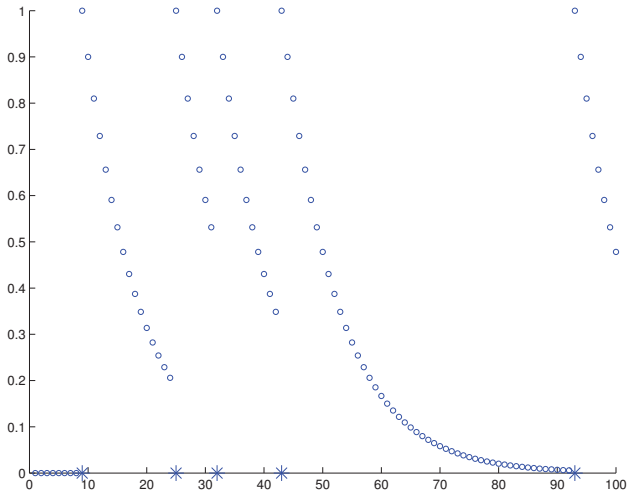


Figure 18.7 Example of an eligibility trace for a value. Visits are marked by an asterisk.

the end of the episode. Online updating takes more time but converges faster. The pseudocode for *Sarsa*(λ) is given in figure 18.8. $Q(\lambda)$ and TD(λ) algorithms can similarly be derived (Sutton and Barto 1998).

SARSA(λ)

18.6 Generalization

Until now, we assumed that the $Q(s,a)$ values (or $V(s)$, if we are estimating values of states) are stored in a lookup table, and the algorithms we considered earlier are called *tabular* algorithms. There are a number of problems with this approach: (1) when the number of states and the number of actions is large, the size of the table may become quite large; (2) states and actions may be continuous, for example, turning the steering wheel by a certain angle, and to use a table, they should be discretized which may cause error; and (3) when the search space is large, too many episodes may be needed to fill in all the entries of the table with acceptable accuracy.

Instead of storing the Q values as they are, we can consider this a regression problem. This is a supervised learning problem where we define a regressor $Q(s,a|\boldsymbol{\theta})$, taking s and a as inputs and parameterized by a

```
Initialize all Q(s,a) arbitrarily, e(s,a) ← 0, ∀s,a
For all episodes
    Initalize s
    Choose a using policy derived from Q, e.g., ε-greedy
    Repeat
        Take action a, observe r and s'
        Choose a' using policy derived from Q, e.g., ε-greedy
        δ ← r + γQ(s',a') − Q(s,a)
        e(s,a) ← 1
        For all s,a:
            Q(s,a) ← Q(s,a) + ηδe(s,a)
            e(s,a) ← γλe(s,a)
        s ← s', a ← a'
    Until s is terminal state
```

Figure 18.8 Sarsa(λ) algorithm.

vector of parameters, $\boldsymbol{\theta}$, to learn Q values. For example, this can be an artificial neural network with s and a as its inputs, one output, and $\boldsymbol{\theta}$ its connection weights.

A good function approximator has the usual advantages and solves the problems discussed previously. A good approximation may be achieved with a simple model without explicitly storing the training instances; it can use continuous inputs; and it allows generalization. If we know that similar (s, a) pairs have similar Q values, we can generalize from past cases and come up with good $Q(s, a)$ values even if that state-action pair has never been encountered before.

To be able to train the regressor, we need a training set. In the case of Sarsa(0), we saw before that we would like $Q(s_t, a_t)$ to get close to $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$. So, we can form a set of training samples where the input is the state-action pair (s_t, a_t) and the required output is $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$. We can write the squared error as

$$E^t(\boldsymbol{\theta}) = [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]^2 \tag{18.20}$$

Training sets can similarly be defined for $Q(0)$ and TD(0), where in the latter case we learn $V(s)$, and the required output is $r_{t+1} + \gamma V(s_{t+1})$.

Once such a set is ready, we can use any supervised learning algorithm for learning the training set.

If we are using a gradient descent method, as in training neural networks, the parameter vector is updated as

$$\Delta\boldsymbol{\theta} = \eta[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]\nabla_{\boldsymbol{\theta}_t} Q(s_t, a_t) \tag{18.21}$$

This is a one-step update. In the case of Sarsa(λ), the eligibility trace is also taken into account:

$$\Delta\boldsymbol{\theta} = \eta\delta_t \boldsymbol{e}_t \tag{18.22}$$

where the temporal difference error is

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

and the vector of eligibilities of parameters are updated as

$$\boldsymbol{e}_t = \gamma\lambda \boldsymbol{e}_{t-1} + \nabla_{\boldsymbol{\theta}_t} Q(s_t, a_t) \tag{18.23}$$

with $\boldsymbol{e}_0$ all zeros. In the case of a tabular algorithm, the eligibilities are stored for the state-action pairs because they are the parameters (stored as a table). In the case of an estimator, eligibility is associated with the parameters of the estimator. We also note that this is very similar to the momentum method for stabilizing backpropagation (section 11.8.1). The difference is that in the case of momentum previous weight changes are remembered, whereas here previous gradient vectors are remembered. Depending on the model used for $Q(s_t, a_t)$, for example, a neural network, we plug its gradient vector in equation 18.23.

In theory, any regression method can be used to train the Q function, but the particular task has a number of requirements. First, it should allow generalization; that is, we really need to guarantee that similar states and actions have similar Q values. This also requires a good coding of s and a, as in any application, to make the similarities apparent. Second, reinforcement learning updates provide instances one by one and not as a whole training set, and the learning algorithm should be able to do individual updates to learn the new instance without forgetting what has been learned before. For example, a multilayer perceptron using backpropagation can be trained with a single instance only if a small learning rate is used. Or, such instances may be collected to form a training set and learned altogether but this slows down learning as no learning happens while a sufficiently large sample is being collected.

Because of these reasons, it seems a good idea to use local learners to learn the Q values. In such methods, for example, radial basis functions, information is localized and when a new instance is learned, only a local part of the learner is updated without possibly corrupting the information in another part. The same requirements apply if we are estimating the state values as $V(s_t|\boldsymbol{\theta})$.

18.7 Partially Observable States

18.7.1 The Setting

In certain applications, the agent does not know the state exactly. It is equipped with sensors that return an *observation*, which the agent then uses to estimate the state. Let us say we have a robot that navigates in a room. The robot may not know its exact location in the room, or what else is there in the room. The robot may have a camera with which sensory observations are recorded. This does not tell the robot its state exactly but gives some indication as to its likely state. For example, the robot may only know that there is an obstacle to its right.

The setting is like a Markov decision process, except that after taking an action a_t, the new state s_{t+1} is not known, but we have an observation o_{t+1} that is a stochastic function of s_t and a_t: $p(o_{t+1}|s_t, a_t)$. This is called a *partially observable MDP* (POMDP). If $o_{t+1} = s_{t+1}$, then POMDP reduces to the MDP. This is just like the distinction between observable and hidden Markov models and the solution is similar; that is, from the observation, we need to infer the state (or rather a probability distribution for the states) and then act based on this. If the agent believes that it is in state s_1 with probability 0.4 and in state s_2 with probability 0.6, then the value of any action is 0.4 times the value of the action in s_1 plus 0.6 times the value of the action in s_2.

PARTIALLY OBSERVABLE MDP

The Markov property does not hold for observations. The next state observation does not only depend on the current action and observation. When there is limited observation, two states may appear the same but are different and if these two states require different actions, this can lead to a loss of performance, as measured by the cumulative reward. The agent should somehow compress the past trajectory into a current unique state estimate. These past observations can also be taken into account by taking a past window of observations as input to the policy,

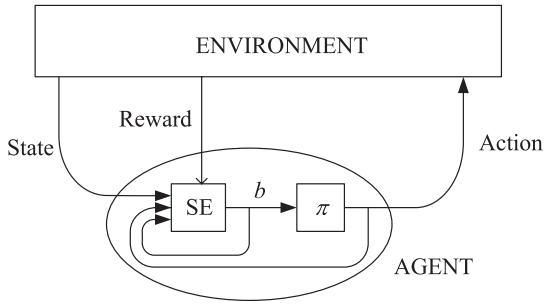


Figure 18.9 In the case of a partially observable environment, the agent has a state estimator (SE) that keeps an internal belief state b and the policy π generates actions based on the belief states.

or one can use a recurrent neural network (section 11.12.2) to maintain the state without forgetting past observations.

At any time, the agent may calculate the most likely state and take an action accordingly. Or it may take an action to gather information and reduce uncertainty, for example, search for a landmark, or stop to ask for direction. This implies the importance of the *value of information*, and indeed POMDPs can be modeled as *dynamic* influence diagrams (section 14.8). The agent chooses between actions based on the amount of information they provide, the amount of reward they produce, and how they change the state of the environment.

VALUE OF INFORMATION

BELIEF STATE

To keep the process Markov, the agent keeps an internal *belief state* b_t that summarizes its experience (see figure 18.9). The agent has a *state estimator* that updates the belief state b_{t+1} based on the last action a_t, current observation o_{t+1}, and its previous belief state b_t. There is a policy π that generates the next action a_{t+1} based on this belief state, as opposed to the actual state that we had in a completely observable environment. The belief state is a probability distribution over states of the environment given the initial belief state (before we did any actions) and the past observation-action history of the agent (without leaving out any information that could improve agent's performance). Q learning in such a case involves the belief state-action pair values, instead of the actual state-action pairs:

$$Q(b_t, a_t) = E[r_{t+1}] + \gamma \sum_{b_{t+1}} P(b_{t+1}|b_t, a_t) V(b_{t+1}) \tag{18.24}$$

18.7.2 Example: The Tiger Problem

We now discuss an example that is a slightly different version of the *Tiger problem* discussed in Kaelbling, Littman, and Cassandra 1998, modified as in the example in Thrun, Burgard, and Fox 2005. Let us say we are standing in front of two doors, one to our left and the other to other right, leading to two rooms. Behind one of the two doors, we do not know which, there is a crouching tiger, and behind the other, there is a treasure. If we open the door of the room where the tiger is, we get a large negative reward, and if we open the door of the treasure room, we get some positive reward. The hidden state, z_L, is the location of the tiger. Let us say p denotes the probability that tiger is in the room to the left and therefore, the tiger is in the room to the right with probability $1 - p$:

$$p \equiv P(z_L = 1)$$

The two actions are a_L and a_R, which respectively correspond to opening the left or the right door. The rewards are

$r(A, Z)$	Tiger left	Tiger right
Open left	−100	+80
Open right	+90	−100

We can calculate the expected reward for the two actions. There are no future rewards because the episode ends once we open one of the doors.

$$\begin{aligned} R(a_L) &= r(a_L, z_L)P(z_L) + r(a_L, z_R)P(z_R) = -100p + 80(1 - p) \\ R(a_R) &= r(a_R, z_L)P(z_L) + r(a_R, z_R)P(z_R) = 90p - 100(1 - p) \end{aligned}$$

Given these rewards, if p is close to 1, if we believe that there is a high chance that the tiger is on the left, the right action will be to choose the right door, and, similarly, for p close to 0, it is better to choose the left door.

The two intersect for p around 0.5, and there the expected reward is approximately −10. The fact that the expected reward is negative when p is around 0.5 (when we have uncertainty) indicates the importance of collecting information. If we can add sensors to to decrease uncertainty—that is, move p away from 0.5 to either close to 0 or close to 1—we can take actions with high positive rewards. That sensing action, a_S, may

have a small negative reward: $R(a_S) = -1$; this may be considered as the cost of sensing or equivalent to discounting future reward by $\gamma < 1$ because we are postponing taking the real action (of opening one of the doors).

In such a case, the expected rewards and value of the best action are shown in figure 18.10a:

$$V = \max(a_L, a_R, a_S)$$

Let us say as sensory input, we use microphones to check whether the tiger is behind the left or the right door. But we have unreliable sensors (so that we still stay in the realm of partial observability). Let us say we can only detect tiger's presence with 0.7 probability:

$$P(o_L|z_L) = 0.7 \qquad P(o_L|z_R) = 0.3$$
$$P(o_R|z_L) = 0.3 \qquad P(o_R|z_R) = 0.7$$

If we sense o_L, our belief in the tiger's position changes:

$$p' = P(z_L|o_L) = \frac{P(o_L|z_L)P(z_L)}{p(o_L)} = \frac{0.7p}{0.7p + 0.3(1-p)}$$

The effect of this is shown in figure 18.10b where we plot $R(a_L|o_L)$. Sensing o_L turns opening the right door into a better action for a wider range. The better sensors we have (if the probability of correct sensing moves from 0.7 closer to 1), the larger this range gets (exercise 9). Similarly, as we see in figure 18.10c, if we sense o_R, this increases the chances of opening the left door. Note that sensing also decreases the range where there is a need to sense (once more).

The expected rewards for the actions in this case are

$$\begin{aligned}
R(a_L|o_L) &= r(a_L, z_L)P(z_L|o_L) + r(a_L, z_R)P(z_R|o_L) \\
&= -100p' + 80(1-p') \\
&= -100 \cdot \frac{0.7 \cdot p}{p(o_L)} + 80 \cdot \frac{0.3 \cdot (1-p)}{p(o_L)} \\
R(a_R|o_L) &= r(a_R, z_L)P(z_L|o_L) + r(a_R, z_R)P(z_R|o_L) \\
&= 90p' - 100(1-p') \\
&= 90 \cdot \frac{0.7 \cdot p}{p(o_L)} - 100 \cdot \frac{0.3 \cdot (1-p)}{p(o_L)} \\
R(a_S|o_L) &= -1
\end{aligned}$$

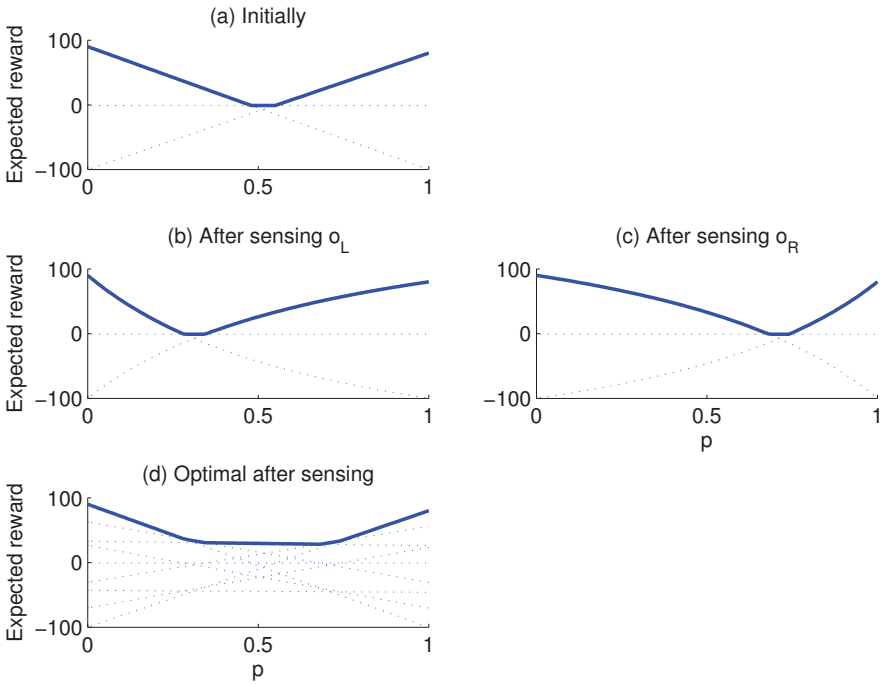


Figure 18.10 Expected rewards and the effect of sensing in the Tiger problem.

The best action is this case is the maximum of these three. Similarly, if we sense o_R, the expected rewards become

$$\begin{aligned} R(a_L|o_R) &= r(a_L,z_L)P(z_L|o_R) + r(a_L,z_R)P(z_R|o_R) \\ &= -100 \cdot \frac{0.3 \cdot p}{p(o_R)} + 80 \cdot \frac{0.7 \cdot (1-p)}{p(o_R)} \\ R(a_R|o_R) &= r(a_R,z_L)P(z_L|o_R) + r(a_R,z_R)P(z_R|o_R) \\ &= 90 \cdot \frac{0.3 \cdot p}{p(o_R)} - 100 \cdot \frac{0.7 \cdot (1-p)}{p(o_R)} \\ R(a_S|o_R) &= -1 \end{aligned}$$

To calculate the expected reward, we need to take average over both sensor readings weighted by their probabilities:

$$V' = \sum_j \left[\max_i R(a_i|o_j) \right] P(O_j)$$

$$
\begin{aligned}
&= \max(R(a_L|o_L), R(a_R|o_L), R(a_S|o_L))P(o_L) + \\
&\quad \max(R(a_L|o_R), R(a_R|o_R), R(a_S|o_R))P(o_R) \\
&= \max(-70p + 24(1-p), 63p - 30(1-p), -0.7p - 0.3(1-p)) + \\
&\quad \max(-30p + 56(1-p), 27p - 70(1-p), -0.3p - 0.7(1-p)) \\
&= \max \begin{pmatrix} -100p & +80(1-p) \\ -43p & -46(1-p) \\ 33p & +26(1-p) \\ 90p & -100(1-p) \end{pmatrix}
\end{aligned} \tag{18.25}
$$

Note that when we multiply by $P(o_L)$, it cancels out and we get functions linear in p. These five lines and the piecewise function that corresponds to their maximum are shown in figure 18.10d. Note that the line, $-40p - 5(1-p)$, as well as the ones involving a_S, are beneath others for all values of p and can safely be pruned. The fact that figure 18.10d is better than figure 18.10a indicates the *value of information*.

VALUE OF INFORMATION

What we calculate here is the value of the best action had we chosen a_S. For example, the first line corresponds to choosing a_L after a_S. So to find the best decision with an episode of length two, we need to back this up by subtracting -1, which is the reward of a_S, and get the expected reward for the action of sense. Equivalently, we can consider this as waiting that has an immediate reward of 0 but discounts the future reward by some $\gamma < 1$. We also have the two usual actions of a_L and a_R and we choose the best of three; the two immediate actions and the one discounted future action.

Let us now make the problem more interesting, as in the example of Thrun, Burgard, and Fox 2005. Let us assume that there is a door between the two rooms and without us seeing, the tiger can move from one room to the other. Let us say that this is a restless tiger and it stays in the same room with probability 0.2 and moves to the other room with probability 0.8. This means that p should also be updated as

$$p' = 0.2p + 0.8(1-p)$$

and this updated p should be used in equation 18.25 while choosing the best action after having chosen a_S:

$$V' = \max \begin{pmatrix} -100p' & +80(1-p') \\ 33p' & +26(1-p') \\ 90p' & -100(1-p') \end{pmatrix}$$

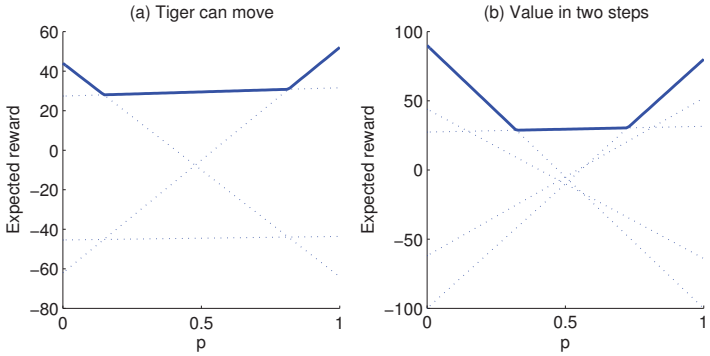


Figure 18.11 Expected rewards change (a) if the hidden state can change, and (b) when we consider episodes of length two.

Figure 18.11b corresponds to figure 18.10d with the updated p'. Now, when planning for episodes of length two, we have the two immediate actions of a_L and a_R, or we wait and sense when p changes and then we take the action and get its discounted reward (figure 18.11b):

$$V_2 = \max \begin{pmatrix} -100p & +80(1-p) \\ 90p & -100(1-p) \\ \max V' - 1 & \end{pmatrix}$$

We see that figure 18.11b is better than figure 18.10a; when wrong actions may lead to large penalty, it is better to defer judgment, look for extra information, and plan ahead. We can consider longer episodes by continuing the iterative updating of p and discounting by subtracting 1 and including the two immediate actions to calculate $V_t, t > 2$.

The algorithm we have just discussed where the value is represented by piecewise linear functions works only when the number of states, actions, observations, and the episode length are all finite. Even in applications where any of these is not small, or when any is continuous-valued, the complexity becomes high and we need to resort to approximate algorithms having reasonable complexity. Reviews of such algorithms are given in Hauskrecht 2000 and Thrun, Burgard, and Fox 2005.

18.8 Notes

More information on reinforcement learning can be found in the textbook by Sutton and Barto (1998) that discusses all the aspects, learning algorithms, and several applications. A comprehensive tutorial is Kaelbling, Littman, and Moore 1996. Recent work on reinforcement learning applied to robotics with some impressive applications is given in Thrun, Burgard, and Fox 2005.

Dynamic programming methods are discussed in Bertsekas 1987 and in Bertsekas and Tsitsiklis 1996, and TD(λ) and Q-learning can be seen as stochastic approximations to dynamic programming (Jaakkola, Jordan, and Singh 1994). Reinforcement learning has two advantages over classical dynamic programming: First, as they learn, they can focus on the parts of the space that are important and ignore the rest; and second, they can employ function approximation methods to represent knowledge that allows them to generalize and learn faster.

LEARNING AUTOMATA

A related field is that of *learning automata* (Narendra and Thathachar 1974), which are finite state machines that learn by trial and error for solving problems like the K-armed bandit. The setting we have here is also the topic of optimal control where there is a controller (agent) taking actions in a plant (environment) that minimize cost (maximize reward).

The earliest use of temporal difference method was in Samuel's checkers program written in 1959 (Sutton and Barto 1998). For every two successive positions in a game, the two board states are evaluated by the board evaluation function that then causes an update to decrease the difference. There has been much work on games because games are both easily defined and challenging. A game like chess can easily be simulated: The allowed moves are formal, and the goal is well defined. Despite the simplicity of defining the game, expert play is quite difficult.

TD-GAMMON

One of the most impressive application of reinforcement learning is the *TD-Gammon* program that learns to play backgammon by playing against itself (Tesauro 1995). This program is superior to the previous neurogammon program also developed by Tesauro, which was trained in a supervised manner based on plays by experts. Backgammon is a complex task with approximately 10^{20} states, and there is randomness due to the roll of dice. Using the TD(λ) algorithm, the program achieves master level play after playing 1,500,000 games against a copy of itself.

Another interesting application is in *job shop scheduling*, or finding a schedule of tasks satisfying temporal and resource constraints (Zhang

and Dietterich 1996). Some tasks have to be finished before others can be started, and two tasks requiring the same resource cannot be done simultaneously. Zhang and Dietterich used reinforcement learning to quickly find schedules that satisfy the constraints and are short. Each state is one schedule, actions are schedule modifications, and the program finds not only one good schedule but a schedule for a class of related scheduling problems.

Recently hierarchical methods have also been proposed where the problem is decomposed into a set of subproblems. This has the advantage that policies learned for the subproblems can be shared for multiple problems, which accelerates learning a new problem (Dietterich 2000). Each subproblem is simpler and learning them separately is faster. The disadvantage is that when they are combined, the policy may be suboptimal.

Though reinforcement learning algorithms are slower than supervised learning algorithms, it is clear that they have a wider variety of application and have the potential to construct better learning machines (Ballard 1997). They do not need any supervision, and this may actually be better since then they are not biased by the teacher. For example, Tesauro's TD-Gammon program in certain circumstances came up with moves that turned out to be superior to those made by the best players. The field of reinforcement learning is developing rapidly, and we may expect to see other impressive results in the near future.

18.9 Exercises

1. Given the grid world in figure 18.12, if the reward on reaching on the goal is 100 and $\gamma = 0.9$, calculate manually $Q^*(s,a)$, $V^*(S)$, and the actions of optimal policy.
2. With the same configuration given in exercise 1, use Q learning to learn the optimal policy.
3. In exercise 1, how does the optimal policy change if another goal state is added to the lower-right corner? What happens if a state of reward -100 (a very bad state) is defined in the lower-right corner?
4. Instead of having $\gamma < 1$, we can have $\gamma = 1$ but with a negative reward of $-c$ for all intermediate (nongoal) states. What is the difference?
5. In exercise 1, assume that the reward on arrival to the goal state is normal distributed with mean 100 and variance 40. Assume also that the actions are

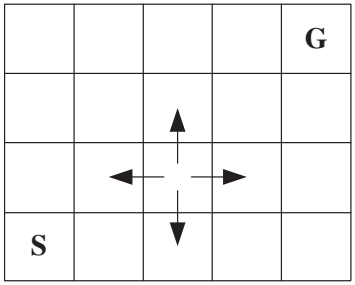


Figure 18.12 The grid world. The agent can move in the four compass directions starting from S. The goal state is G.

also stochastic in that when the robot advances in a direction, it moves in the intended direction with probability 0.5 and there is a 0.25 probability that it moves in one of the lateral directions. Learn $Q(s, a)$ in this case.

6. Assume we are estimating the value function for states $V(s)$ and that we want to use TD(λ) algorithm. Derive the tabular value iteration update.

   SOLUTION: The temporal error at time t is

   $$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

   All state values are updated as

   $$V(s) \leftarrow V(s) + \eta \delta_t e_t(s), \ \forall s$$

   where the eligibility of states decay in time:

   $$e_t(s) = \begin{cases} 1 & \text{if } s = s_t \\ \gamma \lambda e_{t-1}(s) & \text{otherwise} \end{cases}$$

7. Using equation 18.22, derive the weight update equations when a multilayer perceptron is used to estimate Q.

   SOLUTION: Let us say for simplicity we have one-dimensional state value s_t and one-dimensional action value a_t, and let us assume a linear model:

   $$Q(s, a) = w_1 s + w_2 a + w_3$$

   We can update the three parameters w_1, w_2, w_3 using gradient descent (equation 16.21):

   $$\begin{aligned} \Delta w_1 &= \eta \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] s_t \\ \Delta w_2 &= \eta \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] a_t \\ \Delta w_3 &= \eta \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \end{aligned}$$

In the case of a multilayer perceptron, only the last term will differ to update the weights on all layers.

In the case of Sarsa(λ), $\boldsymbol{e}$ is three-dimensional: e^1 for w_1, e^2 for w_2, and e^3 for w_0. We update the eligibilities (equation 18.23):

$$\begin{aligned} e_t^1 &= \gamma\lambda e_{t-1}^1 + s_t \\ e_t^2 &= \gamma\lambda e_{t-1}^2 + a_t \\ e_t^3 &= \gamma\lambda e_{t-1}^3 \end{aligned}$$

and we update the weights using the eligibilities (equation 18.22):

$$\begin{aligned} \Delta w_1 &= \eta\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] e_t^1 \\ \Delta w_2 &= \eta\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] e_t^2 \\ \Delta w_3 &= \eta\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] e_t^3 \end{aligned}$$

8. Give an example of a reinforcement learning application that can be modeled by a POMDP. Define the states, actions, observations, and reward.
9. In the tiger example, show that as we get a more reliable sensor, the range where we need to sense once again decreases.
10. Rework the tiger example using the following reward matrix

$r(A, Z)$	Tiger left	Tiger right
Open left	−100	+10
Open right	20	−100

18.10 References

Ballard, D. H. 1997. *An Introduction to Natural Computation.* Cambridge, MA: MIT Press.

Bellman, R. E. 1957. *Dynamic Programming.* Princeton: Princeton University Press.

Bertsekas, D. P. 1987. *Dynamic Programming: Deterministic and Stochastic Models.* New York: Prentice Hall.

Bertsekas, D. P., and J. N. Tsitsiklis. 1996. *Neuro-Dynamic Programming.* Belmont, MA: Athena Scientific.

Dietterich, T. G. 2000. "Hierarchical Reinforcement Learning with the MAXQ Value Decomposition." *Journal of Artificial Intelligence Research* 13:227–303.

Hauskrecht, M. 2000. "Value-Function Approximations for Partially Observable Markov Decision Processes." *Journal of Artificial Intelligence Research* 13:33–94.

Jaakkola, T., M. I. Jordan, and S. P. Singh. 1994. "On the Convergence of Stochastic Iterative Dynamic Programming Algorithms." *Neural Computation* 6:1185–1201.

Kaelbling, L. P., M. L. Littman, and A. R. Cassandra. 1998. "Planning and Acting in Partially Observable Stochastic Domains." *Artificial Intelligence* 101:99–134.

Kaelbling, L. P., M. L. Littman, and A. W. Moore. 1996. "Reinforcement Learning: A Survey." *Journal of Artificial Intelligence Research* 4:237–285.

Narendra, K. S., and M. A. L. Thathachar. 1974. "Learning Automata—A Survey." *IEEE Transactions on Systems, Man, and Cybernetics* 4:323–334.

Sutton, R. S. 1988. "Learning to Predict by the Method of Temporal Differences." *Machine Learning* 3:9–44.

Sutton, R. S., and A. G. Barto. 1998. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

Tesauro, G. 1995. "Temporal Difference Learning and TD-Gammon." *Communications of the ACM* 38 (3): 58–68.

Thrun, S., W. Burgard, and D. Fox. 2005. *Probabilistic Robotics*. Cambridge, MA: MIT Press.

Watkins, C. J. C. H., and P. Dayan. 1992. "Q-learning." *Machine Learning* 8:279–292.

Zhang, W., and T. G. Dietterich. 1996. "High-Performance Job-Shop Scheduling with a Time-Delay TD(λ) Network." In *Advances in Neural Information Processing Systems 8*, ed. D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, 1024–1030. Cambridge, MA: The MIT Press.

19 *Design and Analysis of Machine Learning Experiments*

We discuss the design of machine learning experiments to assess and compare the performances of learning algorithms in practice and the statistical tests to analyze the results of these experiments.

19.1 Introduction

IN PREVIOUS chapters, we discussed several learning algorithms and saw that, given a certain application, more than one is applicable. Now, we are concerned with two questions:

1. How can we assess the expected error of a learning algorithm on a problem? That is, for example, having used a classification algorithm to train a classifier on a dataset drawn from some application, can we say with enough confidence that later on when it is used in real life, its expected error rate will be less than, for example, 2 percent?

2. Given two learning algorithms, how can we say one has less error than the other one, for a given application? The algorithms compared can be different, for example, parametric versus nonparametric, or they can use different hyperparameter settings. For example, given a multilayer perceptron (chapter 11) with four hidden units and another one with eight hidden units, we would like to be able to say which one has less expected error. Or with the k-nearest neighbor classifier (chapter 8), we would like to find the best value of k.

We cannot look at the training set errors and decide based on those. The error rate on the training set, by definition, is always smaller than the error rate on a test set containing instances unseen during training.

Similarly, training errors cannot be used to compare two algorithms. This is because over the training set, the more complex model having more parameters will almost always give fewer errors than the simple one.

So as we have repeatedly discussed, we need a validation set that is different from the training set. Even over a validation set though, just one run may not be enough. There are two reasons for this: First, the training and validation sets may be small and may contain exceptional instances, like noise and outliers, which may mislead us. Second, the learning method may depend on other random factors affecting generalization. For example, with a multilayer perceptron trained using backpropagation, because gradient descent converges to the nearest local minimum, the initial weights affect the final weights, and given the exact same architecture and training set, starting from different initial weights, there may be multiple possible final classifiers having different error rates on the same validation set. We thus would like to have several runs to average over such sources of randomness. If we train and validate only once, we cannot test for the effect of such factors; this is only admissible if the learning method is so costly that it can be trained and validated only once.

We use a *learning algorithm* on a dataset and generate a *learner*. If we do the training once, we have one learner and one validation error. To average over randomness (in training data, initial weights, etc.), we use the same algorithm and generate multiple learners. We test them on multiple validation sets and record a sample of validation errors. (Of course, all the training and validation sets should be drawn from the same application.) We base our evaluation of the learning algorithm on the *distribution* of these validation errors. We can use this distribution for assessing the *expected error* of the learning algorithm for that problem, or compare it with the error rate distribution of some other learning algorithm.

EXPECTED ERROR

Before proceeding to how this is done, it is important to stress a number of points:

1. We should keep in mind that whatever conclusion we draw from our analysis is conditioned on the dataset we are given. We are not comparing learning algorithms in a domain independent way but on some particular application. We are not saying anything about the expected error of a learning algorithm, or comparing one learning algorithm with another algorithm, in general. Any result we have is only true for the particular application, and only insofar as that application is rep-

NO FREE LUNCH THEOREM

resented in the sample we have. And anyway, as stated by the *No Free Lunch Theorem* (Wolpert 1995), there is no such thing as the "best" learning algorithm. For any learning algorithm, there is a dataset where it is very accurate and another dataset where it is very poor. When we say that a learning algorithm is good, we only quantify how well its inductive bias matches the properties of the data.

2. The division of a given dataset into a number of training and validation set pairs is only for testing purposes. Once all the tests are complete and we have made our decision as to the final method or hyperparameters, to train the final learner, we can use all the labeled data that we have previously used for training or validation.

3. Because we also use the validation set(s) for testing purposes, for example, for choosing the better of two learning algorithms, or to decide where to stop learning, it effectively becomes part of the data we use. When after all such tests, we decide on a particular algorithm and want to report its expected error, we should use a separate *test set* for this purpose, unused during training this final system. This data should have never been used before for training or validation and should be large for the error estimate to be meaningful. So, given a dataset, we should first leave some part of it aside as the test set and use the rest for training and validation. Typically, we can leave one-third of the sample as the test set, then use two-thirds for cross-validation to generate multiple training/validation set pairs, as we will see shortly. So, the training set is used to optimize the parameters, given a particular learning algorithm and model structure; the validation set is used to optimize the hyperparameters of the learning algorithm or the model structure; and the test set is used at the end, once both these have been optimized. For example, with an MLP, the training set is used to optimize the weights, the validation set is used to decide on the number of hidden units, how long to train, the learning rate, and so forth. Once the best MLP configuration is chosen, its final error is calculated on the test set. With k-NN, the training set is stored as the lookup table; we optimize the distance measure and k on the validation set and test finally on the test set.

4. In general, we compare learning algorithms by their error rates, but it should be kept in mind that in real life, error is only one of the criteria that affect our decision. Some other criteria are (Turney 2000):

- risks when errors are generalized using loss functions, instead of 0/1 loss (section 3.3),
- training time and space complexity,
- testing time and space complexity,
- interpretability, namely, whether the method allows knowledge extraction which can be checked and validated by experts, and
- easy programmability.

The relative importance of these factors changes depending on the application. For example, if the training is to be done once in the factory, then training time and space complexity are not important; if adaptability during use is required, then they do become important. Most of the learning algorithms use 0/1 loss and take error as the single criterion to be minimized; recently, *cost-sensitive learning* variants of these algorithms have also been proposed to take other cost criteria into account.

COST-SENSITIVE LEARNING

When we train a learner on a dataset using a training set and test its accuracy on some validation set and try to draw conclusions, what we are doing is experimentation. Statistics defines a methodology to design experiments correctly and analyze the collected data in a manner so as to be able to extract significant conclusions (Montgomery 2005). In this chapter, we will see how this methodology can be used in the context of machine learning.

19.2 Factors, Response, and Strategy of Experimentation

As in other branches of science and engineering, in machine learning too, we do experiments to get information about the process under scrutiny. In our case, this is a learner, which, having been trained on a dataset, generates an output for a given input. An *experiment* is a test or a series of tests where we play with the *factors* that affect the output. These factors may be the algorithm used, the training set, input features, and so on, and we observe the changes in the *response* to be able to extract information. The aim may be to identify the most important factors, screen the unimportant ones, or find the configuration of the factors that optimizes the response—for example, classification accuracy on a given test set.

EXPERIMENT

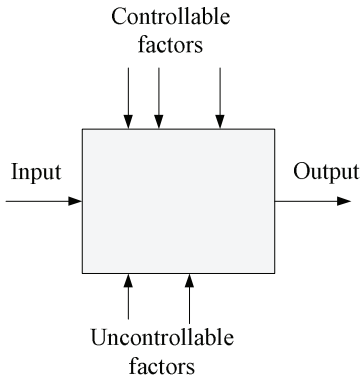


Figure 19.1 The process generates an output given an input and is affected by controllable and uncontrollable factors.

Our aim is to plan and conduct machine learning experiments and analyze the data resulting from the experiments, to be able to eliminate the effect of chance and obtain conclusions which we can consider *statistically significant*. In machine learning, we target a learner having the highest generalization accuracy and the minimal complexity (so that its implementation is cheap in time and space) and is robust, that is, minimally affected by external sources of variability.

A trained learner can be shown as in figure 19.1; it gives an output, for example, a class code for a test input, and this depends on two type of factors. The *controllable factors*, as the name suggests, are those we have control on. The most basic is the learning algorithm used. There are also the hyperparameters of the algorithm, for example, the number of hidden units for a multilayer perceptron, k for k-nearest neighbor, C for support vector machines, and so on. The dataset used and the input representation, that is, how the input is coded as a vector, are other controllable factors.

There are also *uncontrollable factors* over which we have no control, adding undesired variability to the process, which we do not want to affect our decisions. Among these are the noise in the data, the particular training subset if we are resampling from a large set, randomness in the optimization process, for example, the initial state in gradient descent with multilayer perceptrons, and so on.

We use the output to generate the *response* variable—for example, av-

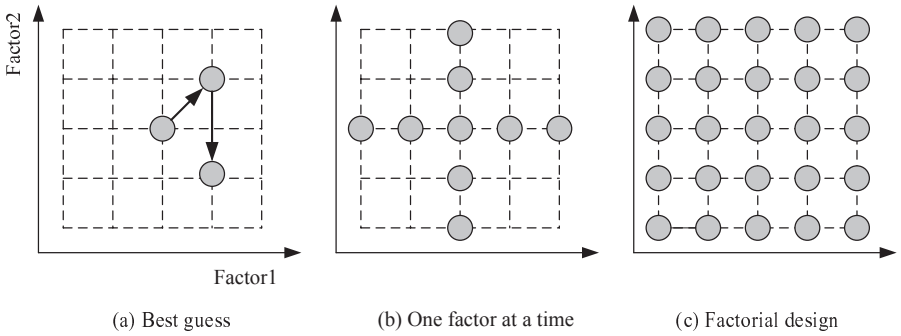


Figure 19.2 Different strategies of experimentation with two factors and five levels each.

erage classification error on a test set, or the expected risk using a loss function, or some other measure, such as precision and recall, as we will discuss shortly.

Given several factors, we need to find the best setting for best response, or in the general case, determine their effect on the response variable. For example, we may be using principal components analyzer (PCA) to reduce dimensionality to d before a k-nearest neighbor (k-NN) classifier. The two factors are d and k, and the question is to decide which combination of d and k leads to highest performance. Or, we may be using a support vector machine classifier with Gaussian kernel, and we have the regularization parameter C and the spread of the Gaussian s^2 to fine-tune together.

STRATEGIES OF EXPERIMENTATION

There are several *strategies of experimentation*, as shown in figure 19.2. In the *best guess* approach, we start at some setting of the factors that we believe is a good configuration. We test the response there and we fiddle with the factors one (or very few) at a time, testing each combination until we get to a state that we consider is good enough. If the experimenter has a good intuition of the process, this may work well; but note that there is no systematic approach to modify the factors and when we stop, we have no guarantee of finding the best configuration.

Another strategy is to modify *one factor at a time* where we decide on a baseline (default) value for all factors, and then we try different levels for one factor while keeping all other factors at their baseline. The major disadvantage of this is that it assumes that there is no *interaction* between the factors, which may not always be true. In the PCA/k-NN

cascade we discussed earlier, each choice for d defines a different input space for k-NN where a different k value may be appropriate.

FACTORIAL DESIGN

The correct approach is to use a *factorial design* where factors are varied together, instead of one at a time; this is colloquially called *grid search*. With F factors at L levels each, searching one factor at a time takes $\mathcal{O}(L \cdot F)$ time, whereas a factorial experiment takes $\mathcal{O}(L^F)$ time.

19.3 Response Surface Design

To decrease the number of runs necessary, one possibility is to run a fractional factorial design where we run only a subset, another is to try to use knowledge gathered from previous runs to estimate configurations that seem likely to have high response. In searching one factor at a time, if we can assume that the response is typically quadratic (with a single maximum, assuming we are maximizing a response value, such as the test accuracy), then instead of trying all values, we can have an iterative procedure where starting from some initial runs, we fit a quadratic, find its maximum analytically, take that as the next estimate, run an experiment there, add the resulting data to the sample, and then continue fitting and sampling, until we get no further improvement.

RESPONSE SURFACE DESIGN

With many factors, this is generalized as the *response surface design* method where we try to fit a parametric response function to the factors as

$$r = g(f_1, f_2, \ldots, f_F | \phi)$$

where r is the response and $f_i, i = 1, \ldots, F$ are the factors. This fitted parametric function defined given the parameters ϕ is our empirical model estimating the response for a particular configuration of the (controllable) factors; the effect of uncontrollable factors is modeled as noise. $g(\cdot)$ is a (typically quadratic) regression model and after a small number of runs around some baseline (as defined by a so-called *design matrix*), one can have enough data to fit $g(\cdot)$ on. Then, we can analytically calculate the values of f_i where the fitted g is maximum, which we take as our next guess, run an experiment there, get a data instance, add it to the sample, fit g once more, and so on, until there is convergence. Whether this approach will work well or not depends on whether the response can indeed be written as a quadratic function of the factors with a single maximum.

19.4 Randomization, Replication, and Blocking

Let us now talk about the three basic principles of experimental design.

RANDOMIZATION

- *Randomization* requires that the order in which the runs are carried out should be randomly determined so that the results are independent. This is typically a problem in real-world experiments involving physical objects; for example, machines require some time to warm up until they operate in their normal range so tests should be done in random order for time not to bias the results. Ordering generally is not a problem in software experiments.

REPLICATION

- *Replication* implies that for the same configuration of (controllable) factors, the experiment should be run a number of times to average over the effect of uncontrollable factors. In machine learning, this is typically done by running the same algorithm on a number of resampled versions of the same dataset; this is known as *cross-validation*, which we will discuss in section 19.6. How the response varies on these different replications of the same experiment allows us to obtain an estimate of the experimental error (the effect of uncontrollable factors), which we can in turn use to determine how large differences should be to be deemed *statistically significant*.

BLOCKING

- *Blocking* is used to reduce or eliminate the variability due to *nuisance factors* that influence the response but in which we are not interested. For example, defects produced in a factory may also depend on the different batches of raw material, and this effect should be isolated from the controllable factors in the factory, such as the equipment, personnel, and so on. In machine learning experimentation, when we use resampling and use different subsets of the data for different replicates, we need to make sure that for example if we are comparing learning algorithms, they should all use the same set of resampled subsets, otherwise the differences in accuracies would depend not only on the algorithms but also on the different subsets—to be able to measure the difference due to algorithms only, the different training sets in replicated runs should be identical; this is what we mean by blocking.

PAIRING

In statistics, if there are two populations, this is called *pairing* and is used in *paired testing*.

19.5 Guidelines for Machine Learning Experiments

Before we start experimentation, we need to have a good idea about what it is we are studying, how the data is to be collected, and how we are planning to analyze it. The steps in machine learning are the same as for any type of experimentation (Montgomery 2005). Note that at this point, it is not important whether the task is classification or regression, or whether it is an unsupervised or a reinforcement learning application. The same overall discussion applies; the difference is only in the sampling distribution of the response data that is collected.

A. Aim of the Study

We need to start by stating the problem clearly, defining what the objectives are. In machine learning, there may be several possibilities. As we discussed before, we may be interested in assessing the expected error (or some other response measure) of a learning algorithm on a particular problem and check that, for example, the error is lower than a certain acceptable level.

Given two learning algorithms and a particular problem as defined by a dataset, we may want to determine which one has less generalization error. These can be two different algorithms, or one can be a proposed improvement of the other, for example, by using a better feature extractor.

In the general case, we may have more than two learning algorithms, and we may want to choose the one with the least error, or order them in terms of error, for a given dataset.

In an even more general setting, instead of on a single dataset, we may want to compare two or more algorithms on two or more datasets.

B. Selection of the Response Variable

We need to decide on what we should use as the quality measure. Most frequently, error is used that is the misclassification error for classification and mean square error for regression. We may also use some variant; for example, generalizing from 0/1 to an arbitrary loss, we may use a risk measure. In information retrieval, we use measures such as precision and recall; we will discuss such measures in section 19.7. In a cost-sensitive

setting, not only the output but also system parameters, for example, its complexity, are taken into account.

C. Choice of Factors and Levels

What the factors are depend on the aim of the study. If we fix an algorithm and want to find the best hyperparameters, then those are the factors. If we are comparing algorithms, the learning algorithm is a factor. If we have different datasets, they also become a factor.

The levels of a factor should be carefully chosen so as not to miss a good configuration and avoid doing unnecessary experimentation. It is always good to try to normalize factor levels. For example, in optimizing k of k-nearest neighbor, one can try values such as 1, 3, 5, and so on, but in optimizing the spread h of Parzen windows, we should not try absolute values such as 1.0, 2.0, and so on, because that depends on the scale of the input; it is better to find some statistic that is an indicator of scale—for example, the average distance between an instance and its nearest neighbor—and try h as different multiples of that statistic.

Though previous expertise is a plus in general, it is also important to investigate all factors and factor levels that may be of importance and not be overly influenced by past experience.

D. Choice of Experimental Design

It is always better to do a factorial design unless we are sure that the factors do not interact, because mostly they do. Replication number depends on the dataset size; it can be kept small when the dataset is large; we will discuss this in the next section when we talk about resampling. However, too few replicates generate few data and this will make comparing distributions difficult; in the particular case of parametric tests, the assumptions of Gaussianity may not be tenable.

Generally, given some dataset, we leave some part as the test set and use the rest for training and validation, probably many times by resampling. How this division is done is important. In practice, using small datasets leads to responses with high variance, and the differences will not be significant and results will not be conclusive.

It is also important to avoid as much as possible toy, synthetic data and use datasets that are collected from real-world under real-life circumstances. Didactic one- or two-dimensional datasets may help provide

intuition, but the behavior of the algorithms may be completely different in high-dimensional spaces.

E. Performing the Experiment

Before running a large factorial experiment with many factors and levels, it is best if one does a few trial runs for some random settings to check that all is as expected. In a large experiment, it is always a good idea to save intermediate results (or seeds of the random number generator), so that a part of the whole experiment can be rerun when desired. All the results should be reproducable. In running a large experiment with many factors and factor levels, one should be aware of the possible negative effects of software aging.

It is important that an experimenter be unbiased during experimentation. In comparing one's favorite algorithm with a competitor, both should be investigated equally diligently. In large-scale studies, it may even be envisaged that testers be different from developers.

One should avoid the temptation to write one's own "library" and instead, as much as possible, use code from reliable sources; such code would have been better tested and optimized.

As in any software development study, the advantages of good documentation cannot be underestimated, especially when working in groups. All the methods developed for high-quality software engineering should also be used in machine learning experiments.

F. Statistical Analysis of the Data

This corresponds to analyzing data in a way so that whatever conclusion we get is not subjective or due to chance. We cast the questions that we want to answer in the framework of hypothesis testing and check whether the sample supports the hypothesis. For example, the question "Is A a more accurate algorithm than B?" becomes the hypothesis "Can we say that the average error of learners trained by A is significantly lower than the average error of learners trained by B?"

As always, visual analysis is helpful, and we can use histograms of error distributions, whisker-and-box plots, range plots, and so on.

G. Conclusions and Recommendations

Once all data is collected and analyzed, we can draw objective conclusions. One frequently encountered conclusion is the need for further experimentation. Most statistical, and hence machine learning or data mining, studies are iterative. It is for this reason that we never start with all the experimentation. It is suggested that no more than 25 percent of the available resources should be invested in the first experiment (Montgomery 2005). The first runs are for investigation only. That is also why it is a good idea not to start with high expectations, or promises to one's boss or thesis advisor.

We should always remember that statistical testing never tells us if the hypothesis is correct or false, but how much the sample seems to concur with the hypothesis. There is always a risk that we do not have a conclusive result or that our conclusions be wrong, especially if the data is small and noisy.

When our expectations are not met, it is most helpful to investigate why they are not. For example, in checking why our favorite algorithm A has worked awfully bad on some cases, we can get a splendid idea for some improved version of A. All improvements are due to the deficiencies of the previous version; finding a deficiency is but a helpful hint that there is an improvement we can make!

But we should not go to the next step of testing the improved version before we are sure that we have completely analyzed the current data and learned all we could learn from it. Ideas are cheap, and useless unless tested, which is costly.

19.6 Cross-Validation and Resampling Methods

For replication purposes, our first need is to get a number of training and validation set pairs from a dataset $\mathcal{X}$ (after having left out some part as the test set). To get them, if the sample $\mathcal{X}$ is large enough, we can randomly divide it into K parts, then randomly divide each part into two and use one half for training and the other half for validation. K is typically 10 or 30. Unfortunately, datasets are never large enough to do this. So we should do our best with small datasets. This is done by repeated use of the same data split differently; this is called *cross-validation*. The catch is that this makes the error percentages dependent as these different sets share data.

CROSS-VALIDATION

So, given a dataset $\mathcal{X}$, we would like to generate K training/validation set pairs, $\{\mathcal{T}_i, \mathcal{V}_i\}_{i=1}^{K}$, from this dataset. We would like to keep the training and validation sets as large as possible so that the error estimates are robust, and at the same time, we would like to keep the overlap between different sets as small as possible. We also need to make sure that classes are represented in the right proportions when subsets of data are held out, not to disturb the class prior probabilities; this is called *stratification*. If a class has 20 percent examples in the whole dataset, in all samples drawn from the dataset, it should also have approximately 20 percent examples.

STRATIFICATION

19.6.1 *K*-Fold Cross-Validation

K-FOLD CROSS-VALIDATION

In K-*fold cross-validation,* the dataset $\mathcal{X}$ is divided randomly into K equal-sized parts, $\mathcal{X}_i, i = 1, \ldots, K$. To generate each pair, we keep one of the K parts out as the validation set and combine the remaining $K - 1$ parts to form the training set. Doing this K times, each time leaving out another one of the K parts out, we get K pairs:

$$\begin{array}{ll} \mathcal{V}_1 = \mathcal{X}_1 & \mathcal{T}_1 = \mathcal{X}_2 \cup \mathcal{X}_3 \cup \cdots \cup \mathcal{X}_K \\ \mathcal{V}_2 = \mathcal{X}_2 & \mathcal{T}_2 = \mathcal{X}_1 \cup \mathcal{X}_3 \cup \cdots \cup \mathcal{X}_K \\ \quad \vdots & \\ \mathcal{V}_K = \mathcal{X}_K & \mathcal{T}_K = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \cdots \cup \mathcal{X}_{K-1} \end{array}$$

There are two problems with this. First, to keep the training set large, we allow validation sets that are small. Second, the training sets overlap considerably, namely, any two training sets share $K - 2$ parts.

K is typically 10 or 30. As K increases, the percentage of training instances increases and we get more robust estimators, but the validation set becomes smaller. Furthermore, there is the cost of training the classifier K times, which increases as K is increased. As N increases, K can be smaller; if N is small, K should be large to allow large enough training sets. One extreme case of K-fold cross-validation is *leave-one-out* where given a dataset of N instances, only one instance is left out as the validation set (instance) and training uses the $N - 1$ instances. We then get N separate pairs by leaving out a different instance at each iteration. This is typically used in applications such as medical diagnosis, where labeled data is hard to find. Leave-one-out does not permit stratification.

LEAVE-ONE-OUT

Recently, with computation getting cheaper, it has also become possible to have multiple runs of K-fold cross-validation, for example, 10×10-

fold, and use average over averages to get more reliable error estimates (Bouckaert 2003).

19.6.2 5 × 2 Cross-Validation

5 × 2 CROSS-VALIDATION

Dietterich (1998) proposed the *5 ×2 cross-validation*, which uses training and validation sets of equal size. We divide the dataset $\mathcal{X}$ randomly into two parts, $\mathcal{X}_1^{(1)}$ and $\mathcal{X}_1^{(2)}$, which gives our first pair of training and validation sets, $\mathcal{T}_1 = \mathcal{X}_1^{(1)}$ and $\mathcal{V}_1 = \mathcal{X}_1^{(2)}$. Then we swap the role of the two halves and get the second pair: $\mathcal{T}_2 = \mathcal{X}_1^{(2)}$ and $\mathcal{V}_2 = \mathcal{X}_1^{(1)}$. This is the first fold; $\mathcal{X}_i^{(j)}$ denotes half j of fold i.

To get the second fold, we shuffle $\mathcal{X}$ randomly and divide this new fold into two, $\mathcal{X}_2^{(1)}$ and $\mathcal{X}_2^{(2)}$. This can be implemented by drawing these from $\mathcal{X}$ randomly without replacement, namely, $\mathcal{X}_1^{(1)} \cup \mathcal{X}_1^{(2)} = \mathcal{X}_2^{(1)} \cup \mathcal{X}_2^{(2)} = \mathcal{X}$. We then swap these two halves to get another pair. We do this for three more folds and because from each fold, we get two pairs, doing five folds, we get ten training and validation sets:

$$
\begin{array}{ll}
\mathcal{T}_1 = \mathcal{X}_1^{(1)} & \mathcal{V}_1 = \mathcal{X}_1^{(2)} \\
\mathcal{T}_2 = \mathcal{X}_1^{(2)} & \mathcal{V}_2 = \mathcal{X}_1^{(1)} \\
\mathcal{T}_3 = \mathcal{X}_2^{(1)} & \mathcal{V}_3 = \mathcal{X}_2^{(2)} \\
\mathcal{T}_4 = \mathcal{X}_2^{(2)} & \mathcal{V}_4 = \mathcal{X}_2^{(1)} \\
\quad \vdots & \\
\mathcal{T}_9 = \mathcal{X}_5^{(1)} & \mathcal{V}_9 = \mathcal{X}_5^{(2)} \\
\mathcal{T}_{10} = \mathcal{X}_5^{(2)} & \mathcal{V}_{10} = \mathcal{X}_5^{(1)}
\end{array}
$$

Of course, we can do this for more than five folds and get more training/validation sets, but Dietterich (1998) points out that after five folds, the sets share many instances and overlap so much that the statistics calculated from these sets, namely, validation error rates, become too dependent and do not add new information. Even with five folds, the sets overlap and the statistics are dependent, but we can get away with this until five folds. On the other hand, if we do have fewer than five folds, we get less data (fewer than ten sets) and will not have a large enough sample to fit a distribution to and test our hypothesis on.

Table 19.1 Confusion matrix for two classes

	Predicted class		
True class	Positive	Negative	Total
Positive	tp : true positive	fn : false negative	p
Negative	fp : false positive	tn : true negative	n
Total	p'	n'	N

19.6.3 Bootstrapping

BOOTSTRAP

To generate multiple samples from a single sample, an alternative to cross-validation is the *bootstrap* that generates new samples by drawing instances from the original sample *with* replacement. We saw the use of bootstrapping in section 17.6 to generate training sets for different learners in bagging. The bootstrap samples may overlap more than cross-validation samples and hence their estimates are more dependent; but is considered the best way to do resampling for very small datasets.

In the bootstrap, we sample N instances from a dataset of size N with replacement. The original dataset is used as the validation set. The probability that we pick an instance is $1/N$; the probability that we do not pick it is $1 - 1/N$. The probability that we do not pick it after N draws is

$$\left(1 - \frac{1}{N}\right)^N \approx e^{-1} = 0.368$$

This means that the training data contains approximately 63.2 percent of the instances; that is, the system will not have been trained on 36.8 percent of the data, and the error estimate will be pessimistic. The solution is replication, that is, to repeat the process many times and look at the average behavior.

19.7 Measuring Classifier Performance

For classification, especially for two-class problems, a variety of measures has been proposed. There are four possible cases, as shown in table 19.1. For a positive example, if the prediction is also positive, this is a *true positive*; if our prediction is negative for a positive example, this is a *false negative*. For a negative example, if the prediction is also negative, we

Table 19.2 Performance measures used in two-class problems

Name	Formula
error	$(fp + fn)/N$
accuracy	$(tp + tn)/N = 1 - \text{error}$
tp-rate	tp/p
fp-rate	fp/n
precision	tp/p'
recall	tp/p = tp-rate
sensitivity	tp/p = tp-rate
specificity	$tn/n = 1 -$ fp-rate

have a *true negative*, and we have a *false positive* if we predict a negative example as positive.

In some two-class problems, we make a distinction between the two classes and hence the two types of errors, false positives and false negatives. Different measures appropriate in different settings are given in table 19.2. Let us envisage an authentication application where, for example, users log on to their accounts by voice. A false positive is wrongly logging on an impostor and a false negative is refusing a valid user. It is clear that the two type of errors are not equally bad; the former is much worse. True positive rate, *tp-rate*, also known as *hit rate*, measures what proportion of valid users we authenticate and false positive rate, *fp-rate*, also known as *false alarm rate*, is the proportion of impostors we wrongly accept.

Let us say the system returns $\hat{P}(C_1|x)$, the probability of the positive class, and for the negative class, we have $\hat{P}(C_2|x) = 1 - \hat{P}(C_1|x)$, and we choose "positive" if $\hat{P}(C_1|x) > \theta$. If θ is close to 1, we hardly choose the positive class; that is, we will have no false positives but also few true positives. As we decrease θ to increase the number of true positives, we risk introducing false positives.

RECEIVER OPERATING CHARACTERISTICS

For different values of θ, we can get a number of pairs of (tp-rate, fp-rate) values and by connecting them we get the *receiver operating characteristics* (ROC) curve, as shown in figure 19.3a. Note that different values of θ correspond to different loss matrices for the two types of error and the ROC curve can also be seen as the behavior of a classifier

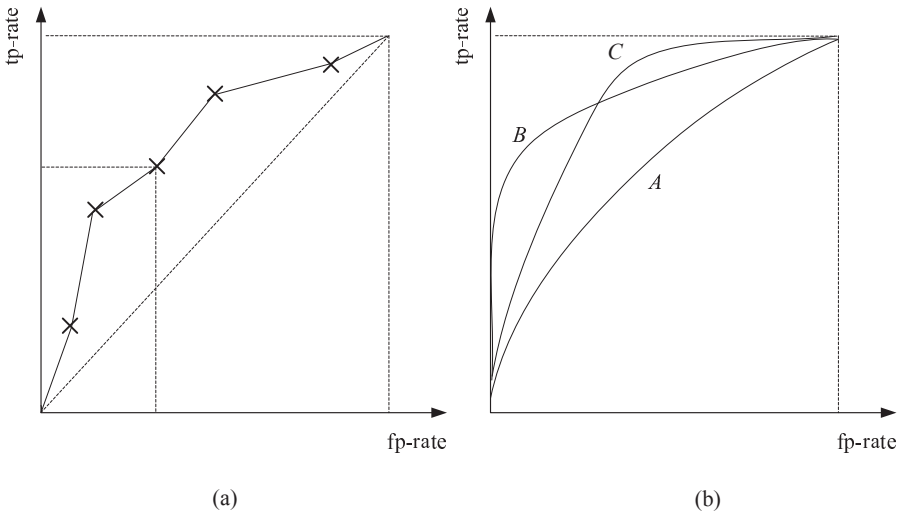


Figure 19.3 (a) Typical ROC curve. Each classifier has a threshold that allows us to move over this curve, and we decide on a point, based on the relative importance of hits versus false alarms, namely, true positives and false positives. The area below the ROC curve is called AUC. (b) A classifier is preferred if its ROC curve is closer to the upper-left corner (larger AUC). *B* and *C* are preferred over *A*; *B* and *C* are preferred under different loss matrices.

under different loss matrices (see exercise 1).

Ideally, a classifier has a tp-rate of 1 and an fp-rate of 0, and hence a classifier is better the more its ROC curve gets closer to the upper-left corner. On the diagonal, we make as many true decisions as false ones, and this is the worst one can do (any classifier that is below the diagonal can be improved by flipping its decision). Given two classifiers, we can say one is better than the other one if its ROC curve is above the ROC curve of the other one; if the two curves intersect, we can say that the two classifiers are better under different loss conditions, as seen in figure 19.3b.

AREA UNDER THE CURVE

ROC allows a visual analysis; if we want to reduce the curve to a single number we can do this by calculating the *area under the curve* (AUC) . A classifier ideally has an AUC of 1 and AUC values of different classifiers can be compared to give us a general performance averaged over different loss conditions.

INFORMATION RETRIEVAL

In *information retrieval*, there is a database of records; we make a query, for example, by using some keywords, and a system (basically a two-class classifier) returns a number of records. In the database, there are relevant records and for a query, the system may retrieve some of them (true positives) but probably not all (false negatives); it may also wrongly retrieve records that are not relevant (false positives). The set of relevant and retrieved records can be visualized using a Venn diagram, as shown in figure 19.4a. *Precision* is the number of retrieved and relevant records divided by the total number of retrieved records; if precision is 1, all the retrieved records may be relevant but there may still be records that are relevant but not retrieved. *Recall* is the number of retrieved relevant records divided by the total number of relevant records; even if recall is 1, all the relevant records may be retrieved but there may also be irrelevant records that are retrieved, as shown in figure19.4c. As in the ROC curve, for different threshold values, one can draw a curve for precision vs. recall.

PRECISION

RECALL

From another perspective but with the same aim, there are the two measures of *sensitivity* and *specificity*. Sensitivity is the same as tp-rate and recall. Specificity is how well we detect the negatives, which is the number of true negatives divided by the total number of negatives; this is equal to 1 minus the false alarm rate. One can also draw a sensitivity vs. specificity curve using different thresholds.

SENSITIVITY

SPECIFICITY

CLASS CONFUSION MATRIX

In the case of $K > 2$ classes, if we are using 0/1 error, the *class confusion matrix* is a $K \times K$ matrix whose entry (i, j) contains the number of instances that belong to C_i but are assigned to C_j. Ideally, all off-diagonals should be 0, for no misclassification. The class confusion matrix allows us to pinpoint what types of misclassification occur, namely, if there are two classes that are frequently confused. Or, one can define K separate two-class problems, each one separating one class from the other $K - 1$.

19.8 Interval Estimation

INTERVAL ESTIMATION

Let us now do a quick review of *interval estimation* that we will use in hypothesis testing. A point estimator, for example, the maximum likelihood estimator, specifies a value for a parameter θ. In interval estimation, we specify an interval within which θ lies with a certain degree of confidence. To obtain such an interval estimator, we make use of the probability distribution of the point estimator.

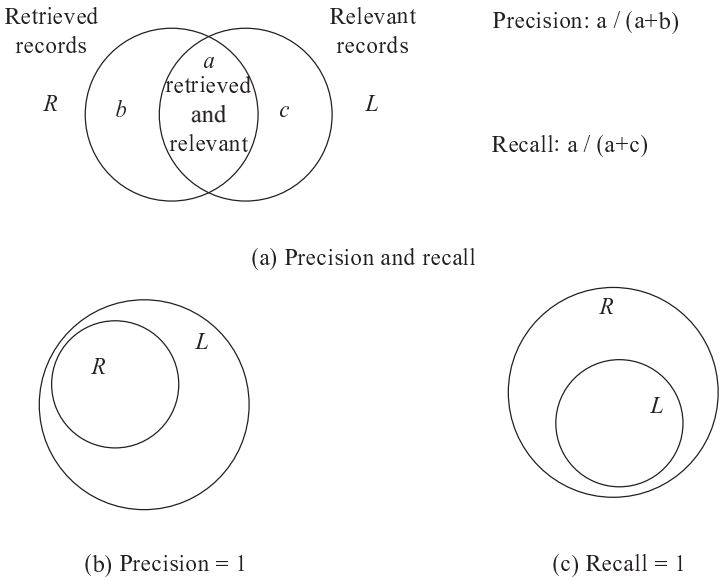


Figure 19.4 (a) Definition of precision and recall using Venn diagrams. (b) Precision is 1; all the retrieved records are relevant but there may be relevant ones not retrieved. (c) Recall is 1; all the relevant records are retrieved but there may also be irrelevant records that are retrieved.

For example, let us say we are trying to estimate the mean μ of a normal density from a sample $\mathcal{X} = \{x^t\}_{t=1}^N$. $m = \sum_t x^t/N$ is the sample average and is the point estimator to the mean. m is the sum of normals and therefore is also normal, $m \sim \mathcal{N}(\mu, \sigma^2/N)$. We define the statistic with a *unit normal distribution*:

UNIT NORMAL DISTRIBUTION

$$\frac{(m-\mu)}{\sigma/\sqrt{N}} \sim \mathcal{Z} \tag{19.1}$$

We know that 95 percent of $\mathcal{Z}$ lies in $(-1.96, 1.96)$, namely, $P\{-1.96 < \mathcal{Z} < 1.96\} = 0.95$, and we can write (see figure 19.5)

$$P\left\{-1.96 < \sqrt{N}\frac{(m-\mu)}{\sigma} < 1.96\right\} = 0.95$$

or equivalently

$$P\left\{m - 1.96\frac{\sigma}{\sqrt{N}} < \mu < m + 1.96\frac{\sigma}{\sqrt{N}}\right\} = 0.95$$

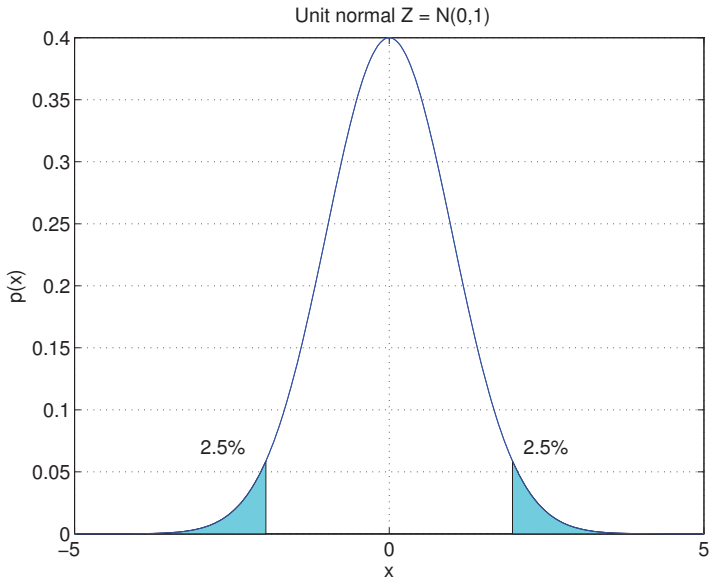


Figure 19.5 95 percent of the unit normal distribution lies between −1.96 and 1.96.

TWO-SIDED CONFIDENCE INTERVAL

That is, "with 95 percent confidence," μ will lie within $1.96\sigma/\sqrt{N}$ units of the sample average. This is a *two-sided confidence interval*. With 99 percent confidence, μ will lie in $(m - 2.58\sigma/\sqrt{N}, m + 2.58\sigma/\sqrt{N})$; that is, if we want more confidence, the interval gets larger. The interval gets smaller as N, the sample size, increases.

This can be generalized for any required confidence as follows. Let us denote z_α such that

$$P\{Z > z_\alpha\} = \alpha, \ 0 < \alpha < 1$$

Because Z is symmetric around the mean, $z_{1-\alpha/2} = -z_{\alpha/2}$, and $P\{X < -z_{\alpha/2}\} = P\{X > z_{\alpha/2}\} = \alpha/2$. Hence for any specified level of confidence $1 - \alpha$, we have

$$P\left\{-z_{\alpha/2} < Z < z_{\alpha/2}\right\} = 1 - \alpha$$

and

$$P\left\{-z_{\alpha/2} < \sqrt{N}\frac{(m-\mu)}{\sigma} < z_{\alpha/2}\right\} = 1 - \alpha$$

or

$$P\left\{m - z_{\alpha/2}\frac{\sigma}{\sqrt{N}} < \mu < m + z_{\alpha/2}\frac{\sigma}{\sqrt{N}}\right\} = 1 - \alpha \tag{19.2}$$

Hence a $100(1-\alpha)$ percent two-sided confidence interval for μ can be computed for any α.

Similarly, knowing that $P\{\mathcal{Z} < 1.64\} = 0.95$, we have (see figure 19.6)

$$P\left\{\sqrt{N}\frac{(m-\mu)}{\sigma} < 1.64\right\} = 0.95$$

or

$$P\left\{m - 1.64\frac{\sigma}{\sqrt{N}} < \mu\right\} = 0.95$$

ONE-SIDED CONFIDENCE INTERVAL

and $(m - 1.64\sigma/\sqrt{N}, \infty)$ is a 95 percent *one-sided upper confidence interval* for μ, which defines a lower bound. Generalizing, a $100(1-\alpha)$ percent one-sided confidence interval for μ can be computed from

$$P\left\{m - z_{\alpha}\frac{\sigma}{\sqrt{N}} < \mu\right\} = 1 - \alpha \tag{19.3}$$

Similarly, the one-sided *lower* confidence interval that defines an upper bound can also be calculated.

In the previous intervals, we used σ; that is, we assumed that the variance is known. If it is not, one can plug the sample variance

$$S^2 = \sum_t (x^t - m)^2/(N-1)$$

instead of σ^2. We know that when $x^t \sim \mathcal{N}(\mu, \sigma^2)$, $(N-1)S^2/\sigma^2$ is chi-square with $N-1$ degrees of freedom. We also know that m and S^2 are independent. Then, $\sqrt{N}(m-\mu)/S$ is t-distributed with $N-1$ degrees of freedom (section A.3.7), denoted as

$$\frac{\sqrt{N}(m-\mu)}{S} \sim t_{N-1} \tag{19.4}$$

Hence for any $\alpha \in (0, 1/2)$, we can define an interval, using the values specified by the t *distribution*, instead of the unit normal $\mathcal{Z}$

t DISTRIBUTION

$$P\left\{t_{1-\alpha/2,N-1} < \sqrt{N}\frac{(m-\mu)}{S} < t_{\alpha/2,N-1}\right\} = 1 - \alpha$$

or using $t_{1-\alpha/2,N-1} = -t_{\alpha/2,N-1}$, we can write

$$P\left\{m - t_{\alpha/2,N-1}\frac{S}{\sqrt{N}} < \mu < m + t_{\alpha/2,N-1}\frac{S}{\sqrt{N}}\right\} = 1 - \alpha$$

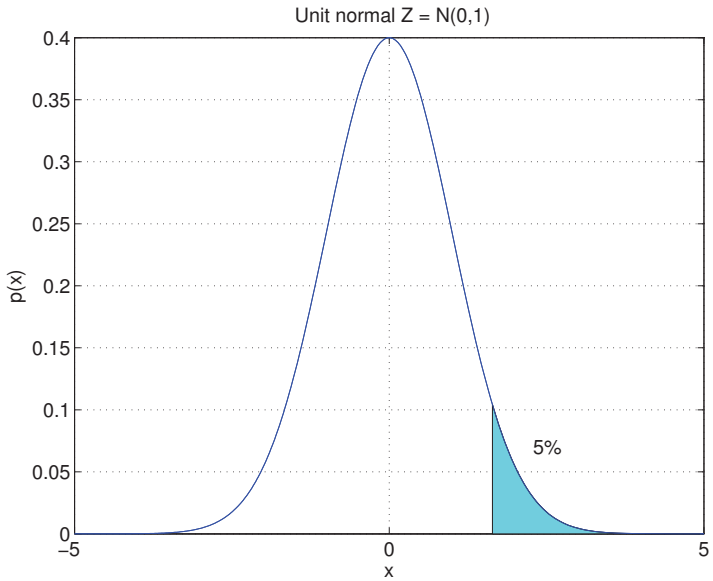


Figure 19.6 95 percent of the unit normal distribution lies before 1.64.

Similarly, one-sided confidence intervals can be defined. The t distribution has larger spread (longer tails) than the unit normal distribution, and generally the interval given by the t is larger; this should be expected since additional uncertainty exists due to the unknown variance.

19.9 Hypothesis Testing

Instead of explicitly estimating some parameters, in certain applications we may want to use the sample to test some particular hypothesis concerning the parameters. For example, instead of estimating the mean, we may want to test whether the mean is less than 0.02. If the random sample is consistent with the hypothesis under consideration, we "fail to reject" the hypothesis; otherwise, we say that it is "rejected." But when we make such a decision, we are not really saying that it is true or false but rather that the sample data appears to be consistent with it to a given degree of confidence or not.

HYPOTHESIS TESTING

In *hypothesis testing*, the approach is as follows. We define a statistic

Table 19.3 Type I error, type II error, and power of a test

	Decision	
Truth	Fail to reject	Reject
True	Correct	Type I error
False	Type II error	Correct (power)

that obeys a certain distribution if the hypothesis is correct. If the statistic calculated from the sample has very low probability of being drawn from this distribution, then we reject the hypothesis; otherwise, we fail to reject it.

Let us say we have a sample from a normal distribution with unknown mean μ and known variance σ^2, and we want to test a specific hypothesis about μ, for example, whether it is equal to a specified constant μ_0. It is denoted as H_0 and is called the *null hypothesis*

NULL HYPOTHESIS

$$H_0 : \mu = \mu_0$$

against the alternative hypothesis

$$H_1 : \mu \neq \mu_0$$

m is the point estimate of μ, and it is reasonable to reject H_0 if m is too far from μ_0. This is where the interval estimate is used. We fail to reject the hypothesis with *level of significance* α if μ_0 lies in the $100(1-\alpha)$ percent confidence interval, namely, if

LEVEL OF SIGNIFICANCE

$$\frac{\sqrt{N}(m-\mu_0)}{\sigma} \in (-z_{\alpha/2}, z_{\alpha/2}) \tag{19.5}$$

We reject the null hypothesis if it falls outside, on either side. This is a *two-sided test.*

TWO-SIDED TEST

If we reject when the hypothesis is correct, this is a *type I error* and thus α, set before the test, defines how much type I error we can tolerate, typical values being $\alpha = 0.1, 0.05, 0.01$ (see table 19.3). A *type II error* is if we fail to reject the null hypothesis when the true mean μ is unequal to μ_0. The probability that H_0 is not rejected when the true mean is μ is a function of μ and is given as

TYPE I ERROR

TYPE II ERROR

$$\beta(\mu) = P_\mu \left\{ -z_{\alpha/2} \leq \frac{m-\mu_0}{\sigma/\sqrt{N}} \leq z_{\alpha/2} \right\} \tag{19.6}$$

POWER FUNCTION

$1 - \beta(\mu)$ is called the *power function* of the test and is equal to the probability of rejection when μ is the true value. Type II error probability increases as μ and μ_0 gets closer, and we can calculate how large a sample we need for us to be able to detect a difference $\delta = |\mu - \mu_0|$ with sufficient power.

ONE-SIDED TEST

One can also have a *one-sided test* of the form

$$H_0 : \mu \leq \mu_0 \text{ vs } H_1 : \mu > \mu_0$$

as opposed to the two-sided test when the alternative hypothesis is $\mu \neq \mu_0$. The one-sided test with α level of significance defines the $100(1-\alpha)$ confidence interval bounded on one side in which m should lie for the hypothesis not to be rejected. We fail to reject if

$$\frac{\sqrt{N}}{\sigma}(m - \mu_0) \in (-\infty, z_\alpha) \tag{19.7}$$

and reject outside. Note that the null hypothesis H_0 also allows equality, which means that we get ordering information only if the test rejects. This tells us which of the two one-sided tests we should use. Whatever claim we have should be in H_1 so that rejection of the test would support our claim.

If the variance is unknown, just as we did in the interval estimates, we use the sample variance instead of the population variance and the fact that

$$\frac{\sqrt{N}(m - \mu_0)}{S} \sim t_{N-1} \tag{19.8}$$

For example, for $H_0 : \mu = \mu_0$ vs $H_1 : \mu \neq \mu_0$, we fail to reject at significance level α if

$$\frac{\sqrt{N}(m - \mu_0)}{S} \in (-t_{\alpha/2,N-1}, t_{\alpha/2,N-1}) \tag{19.9}$$

t TEST

which is known as the *two-sided* t *test*. A one-sided t test can be defined similarly.

19.10 Assessing a Classification Algorithm's Performance

Now that we have reviewed hypothesis testing, we are ready to see how it is used in testing error rates. We will discuss the case of classification error, but the same methodology applies for squared error in regression, log likelihoods in unsupervised learning, expected reward in

reinforcement learning, and so on, as long as we can write the appropriate parametric form for the sampling distribution. We will also discuss nonparametric tests when no such parametric form can be found.

We now start with error rate assessment, and, in the next section, we discuss error rate comparison.

19.10.1 Binomial Test

Let us start with the case where we have a single training set $\mathcal{T}$ and a single validation set $\mathcal{V}$. We train our classifier on $\mathcal{T}$ and test it on $\mathcal{V}$. We denote by p the probability that the classifier makes a misclassification error. We do not know p; it is what we would like to estimate or test a hypothesis about. On the instance with index t from the validation set $\mathcal{V}$, let us say x^t denotes the correctness of the classifier's decision: x^t is a 0/1 Bernoulli random variable that takes the value 1 when the classifier commits an error and 0 when the classifier is correct. The binomial random variable X denotes the total number of errors:

$$X = \sum_{t=1}^{N} x^t$$

We would like to test whether the error probability p is less than or equal to some value p_0 we specify:

$$H_0 : p \leq p_0 \text{ vs. } H_1 : p > p_0$$

If the probability of error is p, the probability that the classifier commits j errors out of N is

$$P\{X = j\} = \left(\begin{array}{c} N \\ j \end{array} \right) p^j (1-p)^{N-j}$$

BINOMIAL TEST

It is reasonable to reject $p \leq p_0$ if in such a case, the probability that we see $X = e$ errors or more is very unlikely. That is, the *binomial test* rejects the hypothesis if

$$P\{X \geq e\} = \sum_{x=e}^{N} \left(\begin{array}{c} N \\ x \end{array} \right) {p_0}^x (1-p_0)^{N-x} < \alpha \tag{19.10}$$

where α is the significance, for example, 0.05.

19.10.2 Approximate Normal Test

If p is the probability of error, our point estimate is $\hat{p} = X/N$. Then, it is reasonable to reject the null hypothesis if $\hat{p}$ is much larger than p_0. How large is large enough is given by the sampling distribution of $\hat{p}$ and the significance α.

Because X is the sum of independent random variables from the same distribution, the central limit theorem states that for large N, X/N is approximately normal with mean p_0 and variance $p_0(1-p_0)/N$. Then

$$\frac{X/N - p_0}{\sqrt{p_0(1-p_0)/N}} \doteq \mathcal{Z} \tag{19.11}$$

where $\doteq$ denotes "approximately distributed." Then, using equation 19.7, the *approximate normal test* rejects the null hypothesis if this value for $X = e$ is greater than z_α. $z_{0.05}$ is 1.64. This approximation will work well as long as N is not too small and p is not very close to 0 or 1; as a rule of thumb, we require $Np \geq 5$ and $N(1-p) \geq 5$.

APPROXIMATE NORMAL TEST

19.10.3 *t* Test

The two tests we discussed earlier use a single validation set. If we run the algorithm K times, on K training/validation set pairs, we get K error percentages, $p_i, i = 1, \ldots, K$ on the K validation sets. Let x_i^t be 1 if the classifier trained on $\mathcal{T}_i$ makes a misclassification error on instance t of $\mathcal{V}_i$; x_i^t is 0 otherwise. Then

$$p_i = \frac{\sum_{t=1}^{N} x_i^t}{N}$$

Given that

$$m = \frac{\sum_{i=1}^{K} p_i}{K}, \quad S^2 = \frac{\sum_{i=1}^{K} (p_i - m)^2}{K-1}$$

from equation 19.8, we know that we have

$$\frac{\sqrt{K}(m - p_0)}{S} \sim t_{K-1} \tag{19.12}$$

and the t test rejects the null hypothesis that the classification algorithm has p_0 or less error percentage at significance level α if this value is greater than $t_{\alpha,K-1}$. Typically, K is taken as 10 or 30. $t_{0.05,9} = 1.83$ and $t_{0.05,29} = 1.70$.

19.11 Comparing Two Classification Algorithms

Given two learning algorithms, we want to compare and test whether they construct classifiers that have the same expected error rate.

19.11.1 McNemar's Test

CONTINGENCY TABLE

Given a training set and a validation set, we use two algorithms to train two classifiers on the training set and test them on the validation set and compute their errors. A *contingency table*, like the one shown here, is an array of natural numbers in matrix form representing counts, or frequencies:

e_{00}: number of examples misclassified by both	e_{01}: number of examples misclassified by 1 but not 2
e_{10}: number of examples misclassified by 2 but not 1	e_{11}: number of examples correctly classified by both

Under the null hypothesis that the classification algorithms have the same error rate, we expect $e_{01} = e_{10}$ and these to be equal to $(e_{01}+e_{10})/2$. We have the chi-square statistic with one degree of freedom

$$\frac{(|e_{01} - e_{10}| - 1)^2}{e_{01} + e_{10}} \sim \chi^2_1 \tag{19.13}$$

MCNEMAR'S TEST

and *McNemar's test* rejects the hypothesis that the two classification algorithms have the same error rate at significance level α if this value is greater than $\chi^2_{\alpha,1}$. For $\alpha = 0.05$, $\chi^2_{0.05,1} = 3.84$.

19.11.2 *K*-Fold Cross-Validated Paired *t* Test

This set uses K-fold cross-validation to get K training/validation set pairs. We use the two classification algorithms to train on the training sets $\mathcal{T}_i, i = 1, \ldots, K$, and test on the validation sets $\mathcal{V}_i$. The error percentages of the classifiers on the validation sets are recorded as p_i^1 and p_i^2.

If the two classification algorithms have the same error rate, then we expect them to have the same mean, or equivalently, that the difference of their means is 0. The difference in error rates on fold i is $p_i = p_i^1 - p_i^2$. This is a *paired test*; that is, for each i, both algorithms see the same training and validation sets. When this is done K times, we have a distribution of p_i containing K points. Given that p_i^1 and p_i^2 are both (approximately)

PAIRED TEST

normal, their difference p_i is also normal. The null hypothesis is that this distribution has 0 mean:

$$H_0 : \mu = 0 \text{ vs. } H_1 : \mu \neq 0$$

We define

$$m = \frac{\sum_{i=1}^{K} p_i}{K}, \quad S^2 = \frac{\sum_{i=1}^{K} (p_i - m)^2}{K - 1}$$

Under the null hypothesis that $\mu = 0$, we have a statistic that is t-distributed with $K - 1$ degrees of freedom:

$$\frac{\sqrt{K}(m - 0)}{S} = \frac{\sqrt{K} \cdot m}{S} \sim t_{K-1} \tag{19.14}$$

K-FOLD CV PAIRED *t* TEST

Thus the K-*fold cv paired* t *test* rejects the hypothesis that two classification algorithms have the same error rate at significance level α if this value is outside the interval $(-t_{\alpha/2,K-1}, t_{\alpha/2,K-1})$. $t_{0.025,9} = 2.26$ and $t_{0.025,29} = 2.05$.

If we want to test whether the first algorithm has less error than the second, we need a one-sided hypothesis and use a one-tailed test:

$$H_0 : \mu \geq 0 \text{ vs. } H_1 : \mu < 0$$

If the test rejects, our claim that the first one has significantly less error is supported.

19.11.3 5 × 2 cv Paired *t* Test

In the 5 × 2 cv t test, proposed by Dietterich (1998), we perform five replications of twofold cross-validation. In each replication, the dataset is divided into two equal-sized sets. $p_i^{(j)}$ is the difference between the error rates of the two classifiers on fold $j = 1, 2$ of replication $i = 1, \ldots, 5$. The average on replication i is $\overline{p}_i = (p_i^{(1)} + p_i^{(2)})/2$, and the estimated variance is $s_i^2 = (p_i^{(1)} - \overline{p}_i)^2 + (p_i^{(2)} - \overline{p}_i)^2$.

Under the null hypothesis that the two classification algorithms have the same error rate, $p_i^{(j)}$ is the difference of two identically distributed proportions, and ignoring the fact that these proportions are not independent, $p_i^{(j)}$ can be treated as approximately normal distributed with 0 mean and unknown variance σ^2. Then $p_i^{(j)}/\sigma$ is approximately unit normal. If we assume $p_i^{(1)}$ and $p_i^{(2)}$ are independent normals (which is not strictly true because their training and test sets are not drawn independently of each other), then s_i^2/σ^2 has a chi-square distribution with

one degree of freedom. If each of the s_i^2 are assumed to be independent (which is not true because they are all computed from the same set of available data), then their sum is chi-square with five degrees of freedom:

$$M = \frac{\sum_{i=1}^{5} s_i^2}{\sigma^2} \sim \mathcal{X}_5^2$$

and

$$t = \frac{p_1^{(1)}/\sigma}{\sqrt{M/5}} = \frac{p_1^{(1)}}{\sqrt{\sum_{i=1}^{5} s_i^2/5}} \sim t_5 \tag{19.15}$$

5×2 CV PAIRED t TEST

giving us a t statistic with five degrees of freedom. The *5×2 cv paired* t *test* rejects the hypothesis that the two classification algorithms have the same error rate at significance level α if this value is outside the interval $(-t_{\alpha/2,5}, t_{\alpha/2,5})$. $t_{0.025,5} = 2.57$.

19.11.4 5×2 cv Paired F Test

We note that the numerator in equation 19.15, $p_1^{(1)}$, is arbitrary; actually, ten different values can be placed in the numerator, namely, $p_i^{(j)}, j = 1, 2, i = 1, \ldots, 5$, leading to ten possible statistics:

$$t_i^{(j)} = \frac{p_i^{(j)}}{\sqrt{\sum_{i=1}^{5} s_i^2/5}} \tag{19.16}$$

Alpaydın (1999) proposed an extension to the 5×2 cv t test that combines the results of the ten possible statistics. If $p_i^{(j)}/\sigma \sim \mathcal{Z}$, then $\left(p_i^{(j)}\right)^2/\sigma^2 \sim \mathcal{X}_1^2$ and their sum is chi-square with ten degrees of freedom:

$$N = \frac{\sum_{i=1}^{5}\sum_{j=1}^{2}\left(p_i^{(j)}\right)^2}{\sigma^2} \sim \mathcal{X}_{10}^2$$

Placing this in the numerator of equation 19.15, we get a statistic that is the ratio of two chi-square distributed random variables. Two such variables divided by their respective degrees of freedom is F-distributed with ten and five degrees of freedom (section A.3.8):

$$f = \frac{N/10}{M/5} = \frac{\sum_{i=1}^{5}\sum_{j=1}^{2}\left(p_i^{(j)}\right)^2}{2\sum_{i=1}^{5} s_i^2} \sim F_{10,5} \tag{19.17}$$

5×2 CV PAIRED F TEST

5×2 cv paired F *test* rejects the hypothesis that the classification algorithms have the same error rate at significance level α if this value is greater than $F_{\alpha,10,5}$. $F_{0.05,10,5} = 4.74$.

19.12 Comparing Multiple Algorithms: Analysis of Variance

In many cases, we have more than two algorithms, and we would like to compare their expected error. Given L algorithms, we train them on K training sets, induce K classifiers with each algorithm, and then test them on K validation sets and record their error rates. This gives us L groups of K values. The problem then is the comparison of these L samples for statistically significant difference. This is an experiment with a single factor with L levels, the learning algorithms, and there are K replications for each level.

ANALYSIS OF VARIANCE

In *analysis of variance* (ANOVA), we consider L independent samples, each of size K, composed of normal random variables of unknown mean μ_j and unknown common variance σ^2:

$$X_{ij} \sim \mathcal{N}(\mu_j, \sigma^2), j = 1, \ldots, L, \; i = 1, \ldots, K,$$

We are interested in testing the hypothesis H_0 that all means are equal:

$$H_0 : \mu_1 = \mu_2 = \cdots = \mu_L \text{ vs. } H_1 : \mu_r \neq \mu_s, \text{for at least one pair } (r, s)$$

The comparison of error rates of multiple classification algorithms fits this scheme. We have L classification algorithms, and we have their error rates on K validation folds. X_{ij} is the number of validation errors made by the classifier, which is trained by classification algorithm j on fold i. Each X_{ij} is binomial and approximately normal. If H_0 is not rejected, we fail to find a significant error difference among the error rates of the L classification algorithms. This is therefore a generalization of the tests we saw in section 19.11 that compared the error rates of two classification algorithms. The L classification algorithms may be different or may use different hyperparameters, for example, number of hidden units in a multilayer perceptron, number of neighbors in k-nn, and so forth.

The approach in ANOVA is to derive two estimators of σ^2. One estimator is designed such that it is true only when H_0 is true, and the second is always a valid estimator, regardless of whether H_0 is true or not. ANOVA then rejects H_0, namely, that the L samples are drawn from the same population, if the two estimators differ significantly.

Our first estimator to σ^2 is valid only if the hypothesis is true, namely, $\mu_j = \mu, j = 1, \ldots, L$. If $X_{ij} \sim \mathcal{N}(\mu, \sigma^2)$, then the group average

$$m_j = \sum_{i=1}^{K} \frac{X_{ij}}{K}$$

is also normal with mean μ and variance σ^2/K. If the hypothesis is true, then $m_j, j = 1, \ldots, L$ are L instances drawn from $\mathcal{N}(\mu, \sigma^2/K)$. Then *their* mean and variance are

$$m = \frac{\sum_{j=1}^{L} m_j}{L}, \quad S^2 = \frac{\sum_j (m_j - m)^2}{L - 1}$$

Thus an estimator of σ^2 is $K \cdot S^2$, namely,

$$\hat{\sigma}_b^2 = K \sum_{j=1}^{L} \frac{(m_j - m)^2}{L - 1} \tag{19.18}$$

Each of m_j is normal and $(L-1)S^2/(\sigma^2/K)$ is chi-square with $(L-1)$ degrees of freedom. Then, we have

$$\sum_j \frac{(m_j - m)^2}{\sigma^2/K} \sim \mathcal{X}_{L-1}^2 \tag{19.19}$$

We define SS_b, the between-group sum of squares, as

$$SS_b \equiv K \sum_j (m_j - m)^2$$

So, when H_0 is true, we have

$$\frac{SS_b}{\sigma^2} \sim \mathcal{X}_{L-1}^2 \tag{19.20}$$

Our second estimator of σ^2 is the average of group variances, S_j^2, defined as

$$S_j^2 = \frac{\sum_{i=1}^{K} (X_{ij} - m_j)^2}{K - 1}$$

and their average is

$$\hat{\sigma}_w^2 = \sum_{j=1}^{L} \frac{S_j^2}{L} = \sum_j \sum_i \frac{(X_{ij} - m_j)^2}{L(K-1)} \tag{19.21}$$

We define SS_w, the within-group sum of squares:

$$SS_w \equiv \sum_j \sum_i (X_{ij} - m_j)^2$$

Remembering that for a normal sample, we have

$$(K-1)\frac{S_j^2}{\sigma^2} \sim \mathcal{X}_{K-1}^2$$

and that the sum of chi-squares is also a chi-square, we have

$$(K-1)\sum_{j=1}^{L}\frac{S_j^2}{\sigma^2}\sim\chi^2_{L(K-1)}$$

So

$$\frac{SS_w}{\sigma^2}\sim\chi^2_{L(K-1)} \tag{19.22}$$

Then we have the task of comparing two variances for equality, which we can do by checking whether their ratio is close to 1. The ratio of two independent chi-square random variables divided by their respective degrees of freedom is a random variable that is F-distributed, and hence when H_0 is true, we have

$$F_0=\left(\frac{SS_b/\sigma^2}{L-1}\right)\Big/\left(\frac{SS_w/\sigma^2}{L(K-1)}\right)=\frac{SS_b/(L-1)}{SS_w/(L(K-1))}=\frac{\hat{\sigma}_b^2}{\hat{\sigma}_w^2}\sim F_{L-1,L(K-1)} \tag{19.23}$$

For any given significance value α, the hypothesis that the L classification algorithms have the same expected error rate is rejected if this statistic is greater than $F_{\alpha,L-1,L(K-1)}$.

Note that we are rejecting if the two estimators disagree significantly. If H_0 is not true, then the variance of m_j around m will be larger than what we would normally have if H_0 were true, and hence if H_0 is not true, the first estimator $\hat{\sigma}_b^2$ will overestimate σ^2, and the ratio will be greater than 1. For $\alpha=0.05$, $L=5$ and $K=10$, $F_{0.05,4,45}=2.6$. If X_{ij} vary around m with a variance of σ^2, then if H_0 is true, m_j vary around m by σ^2/K. If it seems as if they vary more, then H_0 should be rejected because the displacement of m_j around m is more than what can be explained by some constant added noise.

The name *analysis of variance* is derived from a partitioning of the total variability in the data into its components.

$$SS_T\equiv\sum_j\sum_i(X_{ij}-m)^2 \tag{19.24}$$

SS_T divided by its degree of freedom, namely, $K\cdot L-1$ (there are $K\cdot L$ data points, and we lose one degree of freedom because m is fixed), gives us the sample variance of X_{ij}. It can be shown that (exercise 5) the total sum of squares can be split into between-group sum of squares and within-group sum of squares

$$SS_T=SS_b+SS_w \tag{19.25}$$

Table 19.4 The analysis of variance (ANOVA) table for a single factor model

Source of variation	Sum of squares	Degrees of freedom	Mean square	F_0
Between groups	$SS_b \equiv K\sum_j (m_j - m)^2$	$L-1$	$MS_b = \frac{SS_b}{L-1}$	$\frac{MS_b}{MS_w}$
Within groups	$SS_w \equiv \sum_j \sum_i (X_{ij} - m_j)^2$	$L(K-1)$	$MS_w = \frac{SS_w}{L(K-1)}$	
Total	$SS_T \equiv \sum_j \sum_i (X_{ij} - m)^2$	$L \cdot K - 1$		

Results of ANOVA are reported in an ANOVA table as shown in table 19.4. This is the basic *one-way* analysis of variance where there is a single factor, for example, learning algorithm. We may consider experiments with multiple factors, for example, we can have one factor for classification algorithms and another factor for feature extraction algorithms used before, and this will be a *two-factor experiment with interaction*.

POST HOC TESTING

If the hypothesis is rejected, we only know that there is some difference between the L groups but we do not know where. For this, we do *post hoc testing*, that is, an additional set of tests involving subsets of groups, for example, pairs.

LEAST SQUARE DIFFERENCE TEST

Fisher's *least square difference test* compares groups in a pairwise manner. For each group, we have $m_i \sim \mathcal{N}(\mu_i, \sigma_w^2 = MS_w/K)$ and $m_i - m_j \sim \mathcal{N}(\mu_i - \mu_j, 2\sigma_w^2)$. Then, under the null hypothesis that $H_0 : \mu_i = \mu_j$, we have

$$t = \frac{m_i - m_j}{\sqrt{2}\sigma_w} \sim t_{L(K-1)}$$

We reject H_0 in favor of the alternative hypothesis $H_1 : \mu_1 \neq \mu_2$ if $|t| > t_{\alpha/2,L(K-1)}$. Similarly, one-sided tests can be defined to find pairwise orderings.

MULTIPLE COMPARISONS

When we do a number of tests to draw one conclusion, this is called *multiple comparisons*, and we need to keep in mind that if T hypotheses are to be tested, each at significance level α, then the probability that at least one hypothesis is incorrectly rejected is at most $T\alpha$. For example,

the probability that six confidence intervals, each calculated at 95 percent individual confidence intervals, will simultaneously be correct is at least 70 percent. Thus to ensure that the overall confidence interval is at least $100(1-\alpha)$, each confidence interval should be set at $100(1-\alpha/T)$. This is called a *Bonferroni correction*.

BONFERRONI CORRECTION

Sometimes it may be the case that ANOVA rejects and none of the post hoc pairwise tests find a significant difference. In such a case, our conclusion is that there is a difference between the means but that we need more data to be able to pinpoint the source of the difference.

Note that the main cost is the training and testing of L classification algorithms on K training/validation sets. Once this is done and the values are stored in a $K \times L$ table, calculating the ANOVA or pairwise comparison test statistics from those is very cheap in comparison.

19.13 Comparison over Multiple Datasets

Let us say we want to compare two or more algorithms on several datasets and not one. What makes this different is that an algorithm depending on how well its inductive bias matches the problem will behave differently on different datasets, and these error values on different datasets cannot be said to be normally distributed around some mean accuracy. This implies that the parametric tests that we discussed in the previous sections based on binomials being approximately normal are no longer applicable and we need to resort to *nonparametric tests*. The advantage of having such tests is that we can also use them for comparing other statistics that are not normal, for example, training times, number of free parameters, and so on.

NONPARAMETRIC TESTS

Parametric tests are generally robust to slight departures from normality, especially if the sample is large. Nonparametric tests are distribution free but are less efficient; that is, if both are applicable, a parametric test should be preferred. The corresponding nonparametric test will require a larger sample to achieve the same power. Nonparametric tests assume no knowledge about the distribution of the underlying population but only that the values can be compared or ordered, and, as we will see, such tests make use of this order information.

When we have an algorithm trained on a number of different datasets, the average of its errors on these datasets is not a meaningful value, and, for example, we cannot use such averages to compare two algorithms A

and B. To compare two algorithms, the only piece of information we can use is if on any dataset, A is more accurate than B; we can then count the number of times A is more accurate than B and check whether this could have been by chance if they indeed were equally accurate. With more than two algorithms, we will look at the average *ranks* of the learners trained by different algorithms. Nonparametric tests basically use this rank data and not the absolute values.

Before proceeding with the details of these tests, it should be stressed that it does not make sense to compare error rates of algorithms on a whole variety of applications. Because there is no such thing as the "best learning algorithm," such tests would not be conclusive. However, we can compare algorithms on a number of datasets, or versions, of the same application. For example, we may have a number of different datasets for face recognition but with different properties (resolution, lighting, number of subjects, and so on), and we may use a nonparametric test to compare algorithms on those; different properties of the datasets would make it impossible for us to lump images from different datasets together in a single set, but we can train algorithms separately on different datasets, obtain ranks separately, and then combine these to get an overall decision.

19.13.1 Comparing Two Algorithms

Let us say we want to compare two algorithms. We both train and validate them on $i = 1, \ldots, N$ different datasets in a paired manner—that is, all the conditions except the different algorithms should be identical. We get results e_i^1 and e_i^2 and if we use K-fold cross-validation on each dataset, these are averages or medians of the K values. The *sign test* is based on the idea that if the two algorithms have equal error, on each dataset, there should be $1/2$ probability that the first has less error than the second, and thus we expect the first to win on $N/2$ datasets. Let us define

SIGN TEST

$$X_i = \begin{cases} 1 & \text{if } e_i^1 < e_i^2 \\ 0 & \text{otherwise} \end{cases} \quad \text{and } X = \sum_{i=1}^{N} X_i$$

Let us say we want to test

$$H_0 : \mu_1 \geq \mu_2 \text{ vs. } H_1 : \mu_1 < \mu_2$$

If the null hypothesis is correct, X is binomial in N trials with $p = 1/2$. Let us say that we saw that the first one wins on $X = e$ datasets. Then, the probability that we have e or less wins when indeed $p = 1/2$ is

$$P\{X \leq e\} = \sum_{x=0}^{e} \left(\begin{array}{c} N \\ x \end{array} \right) \left(\frac{1}{2}\right)^x \left(\frac{1}{2}\right)^{N-x}$$

and we reject the null hypothesis if this probability is too small, that is, less than α. If there are ties, we divide them equally to both sides; that is, if there are t ties, we add $t/2$ to e (if t is odd, we ignore the odd one and decrease N by 1).

In testing

$$H_0 : \mu_1 \leq \mu_2 \text{ vs. } H_1 : \mu_1 > \mu_2$$

we reject if $P\{X \geq e\} < \alpha$.

For the two-sided test

$$H_0 : \mu_1 = \mu_2 \text{ vs. } H_1 : \mu_1 \neq \mu_2$$

we reject the null hypothesis if e is too small or too large. If $e < N/2$, we reject if $2P\{X \leq e\} < \alpha$; if $e > N/2$, we reject if $2P\{X \geq e\} < \alpha$—we need to find the corresponding tail, and we multiply it by 2 because it is a two-tailed test.

As we discussed before, nonparametric tests can be used to compare any measurements, for example, training times. In such a case, we see the advantage of a nonparametric test that uses order rather than averages of absolute values. Let us say we compare two algorithms on ten datasets, nine of which are small and have training times for both algorithms on the order of minutes, and one that is very large and whose training time is on the order of a day. If we use a parametric test and take the average of training times, the single large dataset will dominate the decision, but when we use the nonparameric test and compare values separately on each dataset, using the order will have the effect of normalizing separately for each dataset and hence will help us make a robust decision.

We can also use the sign test as a one sample test, for example, to check if the average error on all datasets is less than two percent, by comparing μ_1 not by the mean of a second population but by a constant μ_0. We can do this simply by plugging the constant μ_0 in place of all observations from a second sample and using the procedure used earlier;

that is, we will count how many times we get more or less than 0.02 and check if this is too unlikely under the null hypothesis. For large N, normal approximation to the binomial can be used (exercise 6), but in practice, the number of datasets may be smaller than 20. Note that the sign test is a test on the median of a population, which is equal to the mean if the distribution is symmetric.

The sign test only uses the sign of the difference and not its magnitude, but we may envisage a case where the first algorithm, when it wins, always wins by a large margin whereas the second algorithm, when it wins, always wins barely. The *Wilcoxon signed rank test* uses both the sign and the magniture of differences, as follows.

WILCOXON SIGNED RANK TEST

Let us say, in addition to the sign of differences, we also calculate $m_i = |e_i^1 - e_i^2|$ and then we order them so that the smallest, $\min_i m_i$, is assigned rank 1, the next smallest is assigned rank 2, and so on. If there are ties, their ranks are given the average value that they would receive if they differed slightly. For example, if the magnitudes are $2, 1, 2, 4$, the ranks are $2.5, 1, 2.5, 4$. We then calculate w_+ as the sum of all ranks whose signs are positive and w_- as the sum of all ranks whose signs are negative.

The null hypothesis $\mu_1 \leq \mu_2$ can be rejected in favor of the alternative $\mu_1 > \mu_2$ only if w_+ is much smaller than w_-. Similarly, the two-sided hypothesis $\mu_1 = \mu_2$ can be rejected in favor of the alternative $\mu_1 \neq \mu_2$ only if either w_+ or w_-, that is, $w = \min(w_+, w_-)$, is very small. The critical values for the Wilcoxon signed rank test are tabulated and for $N > 20$, normal approximations can be used.

19.13.2 Multiple Algorithms

KRUSKAL-WALLIS TEST

The *Kruskal-Wallis test* is the nonparametric version of ANOVA and is a multiple sample generalization of a rank test. Given the $M = L \cdot N$ observations, for example, error rates, of L algorithms on N datasets, $X_{ij}, i = 1, \ldots, L, j = 1, \ldots, N$, we rank them from the smallest to the largest and assign them ranks, R_{ij}, between 1 and M, again taking averages in case of ties. If the null hypothesis

$$H_0 : \mu_1 = \mu_2 = \cdots = \mu_L$$

is true, then the average of ranks of algorithm i should be approximately halfway between 1 and M, that is, $(M + 1)/2$. We denote the sample

average rank of algorithm i by $\overline{R}_{i\bullet}$ and we reject the hypothesis if the average ranks seem to differ from halfway. The test statistic

$$H = \frac{12}{(M+1)L}\sum_{i=1}^{L}\left(\overline{R}_{i\bullet} - \frac{M+1}{2}\right)$$

is approximately chi-square distributed with $L-1$ degrees of freedom and we reject the null hypothesis if the statistic exceeds $\mathcal{X}_{\alpha,L-1}$.

Just like the parametric ANOVA, if the null hypothesis is rejected, we can do post hoc testing to check for pairwise comparison of ranks. One method for this is *Tukey's test*, which makes use of the *studentized range statistic*

TUKEY'S TEST

$$q = \frac{\overline{R}_{max} - \overline{R}_{min}}{\sigma_w}$$

where $\overline{R}_{max}$ and $\overline{R}_{min}$ are the largest and smallest means (of ranks), respectively, out of the L means, and σ_w^2 is the average variance of ranks around group rank averages. We reject the null hypothesis that groups i and j have the same ranks in favor of the alternative hypothesis that they are different if

$$|\overline{R}_{i\bullet} - \overline{R}_{j\bullet}| > q_\alpha(L, L(K-1))\sigma_w$$

where $q_\alpha(L, L(K-1))$ are tabulated. One-sided tests can also be defined to order algorithms in terms of average rank.

Demsar (2006) proposes to use CD (critical difference) diagrams for visualization. On a scale of 1 to L, we mark the averages, $\overline{R}_{i\bullet}$, and draw lines of length given by the critical difference, $q_\alpha(L, L(K-1))\sigma_w$, between groups, so that lines connect groups that are not statistically significantly different.

19.14 Multivariate Tests

All the tests we discussed earlier in this chapter are univariate; that is, they use a single performance measure, for example, error, precision, area under the curve, and so on. However we know that different measures make different behavior explicit; for example, misclassification error is the sum of false positives and false negatives and a test on error cannot make a distinction between these two types of error. Instead, one can use a bivariate test on these two that will be more powerful than a

univariate test on error because it can also check for the type of misclassification. Similarly, we can define, for example, a bivariate test on [tp-rate, fp-rate] or [precision, recall] that checks for two measures together (Yıldız, Aslan, and Alpaydın 2011).

Let us say that we use p measures. If we compare in terms of (tp-rate, fp-rate) or (precision, recall), then $p = 2$. Actually, all of the performance measures shown in table 19.2, such as error, tp-rate, precision, and so on, are all calculated from the same four entries in table 19.1, and instead of using any predefined measure we can just go ahead and do a four-variate test on [tp, fp, fn, tn].

19.14.1 Comparing Two Algorithms

We assume that $\boldsymbol{x}_{ij}$ are p-variate normal distributions. We have $i = 1, \ldots, K$ folds and we start with the comparison of two algorithms, so $j = 1, 2$. We want to test whether the two populations have the same mean vector in the p-dimensional space:

$$H_0 : \boldsymbol{\mu}_1 = \boldsymbol{\mu}_2 \text{ vs. } H_1 : \boldsymbol{\mu}_1 \neq \boldsymbol{\mu}_2$$

For paired testing, we calculate the paired differences: $\boldsymbol{d}_i = \boldsymbol{x}_{1i} - \boldsymbol{x}_{2i}$, and we test whether these have zero mean:

$$H_0 : \boldsymbol{\mu}_d = \mathbf{0} \text{ vs. } H_1 : \boldsymbol{\mu}_d \neq \mathbf{0}$$

To test for this, we calculate the sample average and covariance matrix:

$$\begin{aligned} \boldsymbol{m} &= \sum_{i=1}^{K} \boldsymbol{d}_i / K \\ \mathbf{S} &= \frac{1}{K-1} \sum_i (\boldsymbol{d}_i - \boldsymbol{m})(\boldsymbol{d}_i - \boldsymbol{m})^T \end{aligned} \tag{19.26}$$

HOTELLING'S MULTIVARIATE TEST

Under the null hypothesis, the *Hotelling's multivariate test* statistic

$$T'^2 = K \boldsymbol{m}^T \mathbf{S}^{-1} \boldsymbol{m} \tag{19.27}$$

is Hotelling's T^2 distributed with p and $K-1$ degrees of freedom (Rencher 1995). We reject the null hypothesis if $T'^2 > T^2_{\alpha,p,K-1}$.

When $p = 1$, this multivariate test reduces to the paired t test we discuss in section 19.11.2. In equation 19.14, $\sqrt{K}m/S$ measures the normalized distance to 0 in one dimension, whereas here, $K\boldsymbol{m}^T\mathbf{S}^{-1}\boldsymbol{m}$ measures the squared Mahalanobis distance to $\mathbf{0}$ in p dimensions. In both cases,

we reject if the distance is so large that it can only occur at most $\alpha \cdot 100$ percent of the time.

If the multivariate test rejects the null hypothesis, we can do p separate post hoc univariate tests (using equation 19.14) to check which one(s) of the variates cause(s) rejection. For example, if a multivariate test on [fp, fn] rejects the null hypothesis, we can check whether the difference is due to a significant difference in false positives, false negatives, or both.

It may be the case that none of the univariate differences is significant whereas the multivariate one is; this is one of the advantages of multivariate testing. The linear combination of variates that causes the maximum difference can be calculated as

$$\mathbf{w} = \mathbf{S}^{-1}\mathbf{m} \tag{19.28}$$

We can then see the effect of the different univariate dimensions by looking at the corresponding elements of $\mathbf{w}$. Actually if $p = 4$, we can think of $\mathbf{w}$ as defining for us a new performance measure from the original four values in the confusion matrix. The fact that this is the Fisher's LDA direction (section 6.8) is not accidental—we are looking for the direction that maximizes the separation of two groups of data.

19.14.2 Comparing Multiple Algorithms

We can similarly get a multivariate test for comparing $L > 2$ algorithms by the multivariate version of ANOVA, namely, MANOVA. We test for

$$H_0 : \boldsymbol{\mu}_1 = \boldsymbol{\mu}_2 = \cdots = \boldsymbol{\mu}_L \text{ vs.}$$

$$H_1 : \boldsymbol{\mu}_r \neq \boldsymbol{\mu}_s \text{ for at least one pair } r, s$$

Let us say that $\mathbf{x}_{ij}, i = 1, \ldots, K, j = 1, \ldots, L$ denotes the p-dimensional performance vector of algorithm j on validation fold i. The multivariate ANOVA (MANOVA) calculates the two matrices of between- and within-scatter:

$$\mathbf{H} = K \sum_{j=1}^{L} (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^T$$

$$\mathbf{E} = \sum_{j=1}^{L} \sum_{i=1}^{K} (\mathbf{x}_{ij} - \mathbf{m}_j)(\mathbf{x}_{ij} - \mathbf{m}_j)^T$$

Then, the test statistic

$$\Lambda' = \frac{|\mathbf{E}|}{|\mathbf{E} + \mathbf{H}|} \tag{19.29}$$

is Wilks's Λ distributed with $p, L(K-1), L-1$ degrees of freedom (Rencher 1995). We reject the null hypothesis if $\Lambda' > \Lambda_{\alpha,p,L(K-1),L-1}$. Note that rejection is for small values of Λ': If the sample mean vectors are equal, we expect $\mathbf{H}$ to be 0 and Λ' to approach 1; as the sample means become more spread out, Λ' becomes "larger" than $\mathbf{E}$ and Λ' approaches 0.

If MANOVA rejects, we can do post hoc testing in a number of ways: We can do a set of pairwise multivariate tests as we discussed previously, to see which pairs are significantly different. Or, we can do p separate univariate ANOVA on each of the individual variates (section 19.12) to see which one(s) cause a reject.

If MANOVA rejects, the difference may be due to some linear combination of the variates: The mean vectors occupy a space whose dimensionality is given by $s = \min(p, L-1)$; its dimensions are the eigenvectors of $\mathbf{E}^{-1}\mathbf{H}$, and by looking at these eigenvectors, we can pinpoint the directions (new performance measures) that cause MANOVA to reject. For example, if $\lambda_i / \sum_{i=1}^{s} \lambda_i > 0.9$, we get roughly one direction, and plotting the projection of data along this direction allows for a univariate ordering of the algorithms.

19.15 Notes

The material related to experiment design follows the discussion from (Montgomery 2005), which here is adapted for machine learning. A more detailed discussion of interval estimation, hypothesis testing, and analysis of variance can be found in any introductory statistics book, for example, Ross 1987.

Dietterich (1998) discusses statistical tests and compares them on a number of applications using different classification algorithms. A review of ROC use and AUC calculation is given in Fawcett 2006. Demsar (2006) reviews statistical tests for comparing classifiers over multiple datasets.

When we compare two or more algorithms, if the null hypothesis that they have the same error rate is not rejected, we choose the simpler one, namely, the one with less space or time complexity. That is, we use our prior preference if the data does not prefer one in terms of error rate. For example, if we compare a linear model and a nonlinear model and if the test does not reject that they have the same expected error rate, we should go for the simpler linear model. Even if the test rejects, in choosing one algorithm over another, error rate is only one of the criteria.

Other criteria like training (space/time) complexity, testing complexity, and interpretability may override in practical applications.

This is how the post hoc test results are used in the MultiTest algorithm (Yıldız and Alpaydın 2006) to generate a full order. We do $L(L-1)/2$ one-sided pairwise tests to order the L algorithms, but it is very likely that the tests will not give a full order but only a partial order. The missing links are filled in using the prior complexity information to get a full order. A topological sort gives an ordering of algorithms using both types of information, error and complexity.

There are also tests to allow checking for *contrasts*. Let us say 1 and 2 are neural network methods and 3 and 4 are fuzzy logic methods. We can then test whether the average of 1 and 2 differs from the average of 3 and 4, thereby allowing us to compare methods in general.

Statistical comparison is needed not only to choose between learning algorithms but also for adjusting the hyperparameters of an algorithm, and the experimental design framework provides us with tools to do this efficiently; for example, response surface design can be used to learn weights in a multiple kernel learning scenario (Gönen and Alpaydın 2011).

Another important point to note is that if are comparing misclassification errors, this implies that from our point of view, all misclassifications have the same cost. When this is not the case, our tests should be based on risks taking a suitable loss function into account. Not much work has been done in this area. Similarly, these tests should be generalized from classification to regression, so as to be able to assess the mean square errors of regression algorithms, or to be able to compare the errors of two regression algorithms.

In comparing two classification algorithms, note that we are testing only whether they have the same expected error rate. If they do, this does not mean that they make the same errors. This is an idea that we used in chapter 17; we can combine multiple models to improve accuracy if different classifiers make different errors.

19.16 Exercises

1. In a two-class problem, let us say we have the loss matrix where $\lambda_{11} = \lambda_{22} = 0$, $\lambda_{21} = 1$ and $\lambda_{12} = \alpha$. Determine the threshold of decision as a function of α.

   SOLUTION: The risk of choosing the first class is $0 \cdot P(C_1|x) + \alpha \cdot P(C_2|x)$ and the risk of choosing the second class is $1 \cdot P(C_1|x) + 0 \cdot P(C_2|x)$ (section 3.3).

We choose C_1 if the former is less than the latter and given that $P(C_2|x) = 1 - P(C_1|x)$, we choose C_1 if

$$P(C_1|x) > \frac{\alpha}{1+\alpha}$$

That is, varying the threshold decision corresponds to varying the relative cost of false positives and false negatives.

2. We can simulate a classifier with error probability p by drawing samples from a Bernoulli distribution. Doing this, implement the binomial, approximate, and t tests for $p_0 \in (0, 1)$. Repeat these tests at least 1,000 times for several values of p and calculate the probability of rejecting the null hypothesis. What do you expect the probability of reject to be when $p_0 = p$?

3. Assume that $x^t \sim \mathcal{N}(\mu, \sigma^2)$ where σ^2 is known. How can we test for $H_0 : \mu \geq \mu_0$ vs. $H_1 : \mu < \mu_0$?

   SOLUTION: Under H_0, we have

   $$z = \frac{\sqrt{N}(m - \mu_0)}{\sigma} \sim \mathcal{Z}$$

   We accept H_0 if $z \in (-z_\alpha, \infty)$.

4. The K-fold cross-validated t test only tests for the equality of error rates. If the test rejects, we do not know which classification algorithm has the lower error rate. How can we test whether the first classification algorithm does not have higher error rate than the second one? Hint: We have to test $H_0 : \mu \leq 0$ vs. $H_1 : \mu > 0$.

5. Show that the total sum of squares can be split into between-group sum of squares and within-group sum of squares as $SS_T = SS_b + SS_w$.

6. Use the normal approximation to the binomial for the sign test.

   SOLUTION: Under the null hypothesis that the two are equally good, we have $p = 1/2$ and over N datasets, we expect the number of wins X to be approximately Gaussian with $\mu = pN = N/2$ and $\sigma^2 = p(1-p)N = N/4$. If there are e wins, we reject if $P(X < e) > \alpha$, or if $P(Z < \frac{e - N/2}{\sqrt{N/4}}) > \alpha$.

7. Let us say we have three classification algorithms. How can we order these three from best to worst?

8. If we have two variants of algorithm A and three variants of algorithm B, how can we compare the overall accuracies of A and B taking all their variants into account?

   SOLUTION: We can use *contrasts* (Montgomery 2005). Basically, what we would be doing is comparing the average of the two variants of A with the average of the three variants of B.

9. Propose a suitable test to compare the errors of two regression algorithms.
   SOLUTION: In regression, we minimize the sum of squares that is a measure of variance, which we know is chi-squared distributed. Since we use the F test to compare variances (as we did in ANOVA), we can also use it to compare the squared errors of two regression algorithms.
10. Propose a suitable test to compare the expected rewards of two reinforcement learning algorithms.

19.17 References

Alpaydın, E. 1999. "Combined 5×2 cv F Test for Comparing Supervised Classification Learning Algorithms." *Neural Computation* 11:1885–1892.

Bouckaert, R. R. 2003. "Choosing between Two Learning Algorithms based on Calibrated Tests." In *Twentieth International Conference on Machine Learning*, ed. T. Fawcett and N. Mishra, 51–58. Menlo Park, CA: AAAI Press.

Demsar, J. 2006. "Statistical Comparison of Classifiers over Multiple Data Sets." *Journal of Machine Learning Research* 7:1–30.

Dietterich, T. G. 1998. "Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms." *Neural Computation* 10:1895–1923.

Fawcett, T. 2006. "An Introduction to ROC Analysis." *Pattern Recognition Letters* 27:861–874.

Gönen, M. and, E. Alpaydın. 2011. "Regularizing Multiple Kernel Learning using Response Surface Methodology." *Pattern Recognition* 44:159–171.

Montgomery, D. C. 2005. *Design and Analysis of Experiments*. 6th ed. New York: Wiley.

Rencher, A. C. 1995. *Methods of Multivariate Analysis*. New York: Wiley.

Ross, S. M. 1987. *Introduction to Probability and Statistics for Engineers and Scientists*. New York: Wiley.

Turney, P. 2000. "Types of Cost in Inductive Concept Learning." Paper presented at Workshop on Cost-Sensitive Learning at the Seventeenth International Conference on Machine Learning, Stanford University, Stanford, CA, July 2.

Wolpert, D. H. 1995. "The Relationship between PAC, the Statistical Physics Framework, the Bayesian Framework, and the VC Framework." In *The Mathematics of Generalization*, ed. D. H. Wolpert, 117–214. Reading, MA: Addison-Wesley.

Yıldız, O. T., and E. Alpaydın. 2006. "Ordering and Finding the Best of $K > 2$ Supervised Learning Algorithms." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28:392–402.

Yıldız, O. T., Ö. Aslan, and E. Alpaydın. 2011. "Multivariate Statistical Tests for Comparing Classification Algorithms," In *Learning and Intelligent Optimization (LION) Conference*, ed. C. A. Coello Coello, 1–15. Heidelberg: Springer.

A *Probability*

We review briefly the elements of probability, the concept of a random variable, and example distributions.

A.1 Elements of Probability

A RANDOM experiment is one whose outcome is not predictable with certainty in advance (Ross 1987; Casella and Berger 1990). The set of all possible outcomes is known as the *sample space* S. A sample space is *discrete* if it consists of a finite (or countably infinite) set of outcomes; otherwise it is *continuous*. Any subset E of S is an *event*. Events are sets, and we can talk about their complement, intersection, union, and so forth.

One interpretation of probability is as a *frequency*. When an experiment is continually repeated under the exact same conditions, for any event E, the proportion of time that the outcome is in E approaches some constant value. This constant limiting frequency is the probability of the event, and we denote it as $P(E)$.

Probability sometimes is interpreted as a *degree of belief*. For example, when we speak of Turkey's probability of winning the World Soccer Cup in 2018, we do not mean a frequency of occurrence, since the championship will happen only once and it has not yet occurred (at the time of the writing of this book). What we mean in such a case is a subjective degree of belief in the occurrence of the event. Because it is subjective, different individuals may assign different probabilities to the same event.

A.1.1 Axioms of Probability

Axioms ensure that the probabilities assigned in a random experiment can be interpreted as relative frequencies and that the assignments are consistent with our intuitive understanding of relationships among relative frequencies:

1. $0 \leq P(E) \leq 1$. If E_1 is an event that cannot possibly occur, then $P(E_1) = 0$. If E_2 is sure to occur, $P(E_2) = 1$.
2. S is the sample space containing all possible outcomes, $P(S) = 1$.
3. If $E_i, i = 1, \ldots, n$ are mutually exclusive (i.e., if they cannot occur at the same time, as in $E_i \cap E_j = \emptyset, j \neq i$, where $\emptyset$ is the *null event* that does not contain any possible outcomes), we have

$$P\left(\bigcup_{i=1}^{n} E_i\right) = \sum_{i=1}^{n} P(E_i) \tag{A.1}$$

For example, letting E^c denote the *complement* of E, consisting of all possible outcomes in S that are not in E, we have $E \cap E^C = \emptyset$ and

$$P(E \cup E^c) = P(E) + P(E^c) = 1$$
$$P(E^c) = 1 - P(E)$$

If the intersection of E and F is not empty, we have

$$P(E \cup F) = P(E) + P(F) - P(E \cap F) \tag{A.2}$$

A.1.2 Conditional Probability

$P(E|F)$ is the probability of the occurrence of event E given that F occurred and is given as

$$P(E|F) = \frac{P(E \cap F)}{P(F)} \tag{A.3}$$

Knowing that F occurred reduces the sample space to F, and the part of it where E also occurred is $E \cap F$. Note that equation A.3 is well-defined only if $P(F) > 0$. Because $\cap$ is commutative, we have

$$P(E \cap F) = P(E|F)P(F) = P(F|E)P(E)$$

which gives us *Bayes' formula*:

$$P(F|E) = \frac{P(E|F)P(F)}{P(E)} \tag{A.4}$$

When F_i are mutually exclusive and exhaustive, namely, $\bigcup_{i=1}^{n} F_i = S$

$$E = \bigcup_{i=1}^{n} E \cap F_i$$

$$P(E) = \sum_{i=1}^{n} P(E \cap F_i) = \sum_{i=1}^{n} P(E|F_i)P(F_i) \tag{A.5}$$

Bayes' formula allows us to write

$$P(F_i|E) = \frac{P(E \cap F_i)}{P(E)} = \frac{P(E|F_i)P(F_i)}{\sum_j P(E|F_j)P(F_j)} \tag{A.6}$$

If E and F are *independent*, we have $P(E|F) = P(E)$ and thus

$$P(E \cap F) = P(E)P(F) \tag{A.7}$$

That is, knowledge of whether F has occurred does not change the probability that E occurs.

A.2 Random Variables

A *random variable* is a function that assigns a number to each outcome in the sample space of a random experiment.

A.2.1 Probability Distribution and Density Functions

The *probability distribution function* $F(\cdot)$ of a random variable X for any real number a is

$$F(a) = P\{X \leq a\} \tag{A.8}$$

and we have

$$P\{a < X \leq b\} = F(b) - F(a) \tag{A.9}$$

If X is a discrete random variable

$$F(a) = \sum_{\forall x \leq a} P(x) \tag{A.10}$$

where $P(\cdot)$ is the *probability mass function* defined as $P(a) = P\{X = a\}$. If X is a *continuous* random variable, $p(\cdot)$ is the *probability density function* such that

$$F(a) = \int_{-\infty}^{a} p(x)dx \tag{A.11}$$

A.2.2 Joint Distribution and Density Functions

In certain experiments, we may be interested in the relationship between two or more random variables, and we use the *joint* probability distribution and density functions of X and Y satisfying

$$F(x, y) = P\{X \le x, Y \le y\} \tag{A.12}$$

Individual *marginal* distributions and densities can be computed by marginalizing, namely, summing over the free variable:

$$F_X(x) = P\{X \le x\} = P\{X \le x, Y \le \infty\} = F(x, \infty) \tag{A.13}$$

In the discrete case, we write

$$P(X = x) = \sum_j P(x, y_j) \tag{A.14}$$

and in the continuous case, we have

$$p_X(x) = \int_{-\infty}^{\infty} p(x, y)dy \tag{A.15}$$

If X and Y are *independent*, we have

$$p(x, y) = p_X(x)p_Y(y) \tag{A.16}$$

These can be generalized in a straightforward manner to more than two random variables.

A.2.3 Conditional Distributions

When X and Y are random variables

$$P_{X|Y}(x|y) = P\{X = x|Y = y\} = \frac{P\{X = x, Y = y\}}{P\{Y = y\}} = \frac{P(x, y)}{P_Y(y)} \tag{A.17}$$

A.2.4 Bayes' Rule

When two random variables are jointly distributed with the value of one known, the probability that the other takes a given value can be computed using *Bayes' rule*:

$$P(y|x) = \frac{P(x|y)P_Y(y)}{P_X(x)} = \frac{P(x|y)P_Y(y)}{\sum_y P(x|y)P_Y(y)} \tag{A.18}$$

Or, in words

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}} \tag{A.19}$$

Note that the denominator is obtained by summing (or integrating if y is continuous) the numerator over all possible y values. The "shape" of $p(y|x)$ depends on the numerator with denominator as a normalizing factor to guarantee that $p(y|x)$ sum to 1. Bayes' rule allows us to modify a prior probability into a posterior probability by taking information provided by x into account.

Bayes' rule inverts dependencies, allowing us to compute $p(y|x)$ if $p(x|y)$ is known. Suppose that y is the "cause" of x, like y going on summer vacation and x having a suntan. Then $p(x|y)$ is the probability that someone who is known to have gone on summer vacation has a suntan. This is the *causal* (or predictive) way. Bayes' rule allows us a *diagnostic* approach by allowing us to compute $p(y|x)$: namely, the probability that someone who is known to have a suntan, has gone on summer vacation. Then $p(y)$ is the general probability of anyone's going on summer vacation and $p(x)$ is the probability that anyone has a suntan, including both those who have gone on summer vacation and those who have not.

A.2.5 Expectation

Expectation, *expected value*, or *mean* of a random variable X, denoted by $E[X]$, is the average value of X in a large number of experiments:

$$E[X] = \begin{cases} \sum_i x_i P(x_i) & \text{if } X \text{ is discrete} \\ \int x p(x) dx & \text{if } X \text{ is continuous} \end{cases} \tag{A.20}$$

It is a weighted average where each value is weighted by the probability that X takes that value. It has the following properties ($a, b \in \Re$):

$$\begin{aligned} E[aX + b] &= aE[X] + b \\ E[X + Y] &= E[X] + E[Y] \end{aligned} \tag{A.21}$$

For any real-valued function $g(\cdot)$, the expected value is

$$E[g(X)] = \begin{cases} \sum_i g(x_i)P(x_i) & \text{if } X \text{ is discrete} \\ \int g(x)p(x)dx & \text{if } X \text{ is continuous} \end{cases} \tag{A.22}$$

A special $g(x) = x^n$, called the nth moment of X, is defined as

$$E[X^n] = \begin{cases} \sum_i x_i^n P(x_i) & \text{if } X \text{ is discrete} \\ \int x^n p(x)dx & \text{if } X \text{ is continuous} \end{cases} \tag{A.23}$$

Mean is the first moment and is denoted by μ.

A.2.6 Variance

Variance measures how much X varies around the expected value. If $\mu \equiv E[X]$, the variance is defined as

$$\text{Var}(X) = E[(X-\mu)^2] = E[X^2] - \mu^2 \tag{A.24}$$

Variance is the second moment minus the square of the first moment. Variance, denoted by σ^2, satisfies the following property ($a, b \in \Re$):

$$\text{Var}(aX+b) = a^2\text{Var}(X) \tag{A.25}$$

$\sqrt{\text{Var}(X)}$ is called the *standard deviation* and is denoted by σ. Standard deviation has the same unit as X and is easier to interpret than variance.

Covariance indicates the relationship between two random variables. If the occurrence of X makes Y more likely to occur, then the covariance is positive; it is negative if X's occurrence makes Y less likely to happen and is 0 if there is no dependence.

$$\text{Cov}(X,Y) = E[(X-\mu_X)(Y-\mu_Y)] = E[XY] - \mu_X\mu_Y \tag{A.26}$$

where $\mu_X \equiv E[X]$ and $\mu_Y \equiv E[Y]$. Some other properties are

$$\begin{aligned} \text{Cov}(X,Y) &= \text{Cov}(Y,X) \\ \text{Cov}(X,X) &= \text{Var}(X) \\ \text{Cov}(X+Z,Y) &= \text{Cov}(X,Y) + \text{Cov}(Z,Y) \end{aligned}$$

$$\text{Cov}\left(\sum_i X_i, Y\right) = \sum_i \text{Cov}(X_i,Y) \tag{A.27}$$

$$\text{Var}(X+Y) = \text{Var}(X) + \text{Var}(Y) + 2\text{Cov}(X,Y) \tag{A.28}$$

$$\text{Var}\left(\sum_i X_i\right) = \sum_i \text{Var}(X_i) + \sum_i \sum_{j \neq i} \text{Cov}(X_i, X_j) \tag{A.29}$$

If X and Y are independent, $E[XY] = E[X]E[Y] = \mu_X\mu_Y$ and $\text{Cov}(X, Y) = 0$. Thus if X_i are independent

$$\text{Var}\left(\sum_i X_i\right) = \sum_i \text{Var}(X_i) \tag{A.30}$$

Correlation is a normalized, dimensionless quantity that is always between -1 and 1:

$$\text{Corr}(X, Y) = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}} \tag{A.31}$$

A.2.7 Weak Law of Large Numbers

Let $\mathcal{X} = \{X^t\}_{t=1}^N$ be a set of independent and identically distributed (iid) random variables each having mean μ and a finite variance σ^2. Then for any $\epsilon > 0$,

$$P\left\{\left|\frac{\sum_t X^t}{N} - \mu\right| > \epsilon\right\} \to 0 \text{ as } N \to \infty \tag{A.32}$$

That is, the average of N trials converges to the mean as N increases.

A.3 Special Random Variables

There are certain types of random variables that occur so frequently that names are given to them.

A.3.1 Bernoulli Distribution

A trial is performed whose outcome is either a "success" or a "failure." The random variable X is a 0/1 indicator variable and takes the value 1 for a success outcome and is 0 otherwise. p is the probability that the result of trial is a success. Then

$$P\{X = 1\} = p \text{ and } P\{X = 0\} = 1 - p \tag{A.33}$$

which can equivalently be written as

$$P\{X = i\} = p^i(1 - p)^{1-i}, i = 0, 1 \tag{A.34}$$

If X is Bernoulli, its expected value and variance are

$$E[X] = p,\ \text{Var}(X) = p(1 - p) \tag{A.35}$$

A.3.2 Binomial Distribution

If N identical independent Bernoulli trials are made, the random variable X that represents the number of successes that occurs in N trials is binomial distributed. The probability that there are i successes is

$$P\{X = i\} = \left(\begin{array}{c} N \\ i \end{array} \right) p^i (1-p)^{N-i}, i = 0 \ldots N \tag{A.36}$$

If X is binomial, its expected value and variance are

$$E[X] = Np, \ \mathrm{Var}(X) = Np(1-p) \tag{A.37}$$

A.3.3 Multinomial Distribution

Consider a generalization of Bernoulli where instead of two states, the outcome of a random event is one of K mutually exclusive and exhaustive states, each of which has a probability of occurring p_i where $\sum_{i=1}^{K} p_i = 1$. Suppose that N such trials are made where outcome i occurred N_i times with $\sum_{i=1}^{k} N_i = N$. Then the joint distribution of $N_1, N_2, \ldots, N_K$ is multinomial:

$$P(N_1, N_2, \ldots, N_K) = N! \prod_{i=1}^{K} \frac{p_i^{N_i}}{N_i!} \tag{A.38}$$

A special case is when $N = 1$; only one trial is made. Then N_i are 0/1 indicator variables of which only one of them is 1 and all others are 0. Then equation A.38 reduces to

$$P(N_1, N_2, \ldots, N_K) = \prod_{i=1}^{K} p_i^{N_i} \tag{A.39}$$

A.3.4 Uniform Distribution

X is uniformly distributed over the interval $[a, b]$ if its density function is given by

$$p(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases} \tag{A.40}$$

If X is uniform, its expected value and variance are

$$E[X] = \frac{a+b}{2}, \ \mathrm{Var}(X) = \frac{(b-a)^2}{12} \tag{A.41}$$

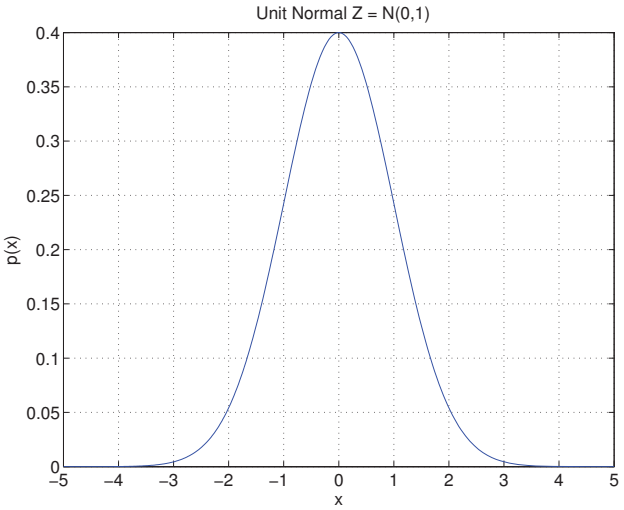


Figure A.1 Probability density function of $\mathcal{Z}$, the unit normal distribution.

A.3.5 Normal (Gaussian) Distribution

X is normal or Gaussian distributed with mean μ and variance σ^2, denoted as $\mathcal{N}(\mu, \sigma^2)$, if its density function is

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right], -\infty < x < \infty \tag{A.42}$$

Many random phenomena obey the bell-shaped normal distribution, at least approximately, and many observations from nature can be seen as a continuous, slightly different versions of a typical value—that is probably why it is called the *normal* distribution. In such a case, μ represents the typical value and σ defines how much instances vary around the prototypical value.

68.27 percent lie in $(\mu - \sigma, \mu + \sigma)$, 95.45 percent in $(\mu - 2\sigma, \mu + 2\sigma)$, and 99.73 percent in $(\mu - 3\sigma, \mu + 3\sigma)$. Thus $P\{|x - \mu| < 3\sigma\} \approx 0.99$. For practical purposes, $p(x) \approx 0$ if $x < \mu - 3\sigma$ or $x > \mu + 3\sigma$. $\mathcal{Z}$ is unit normal, namely, $\mathcal{N}(0, 1)$ (see figure A.1), and its density is written as

$$p_Z(x) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{x^2}{2}\right] \tag{A.43}$$

If $X \sim \mathcal{N}(\mu, \sigma^2)$ and $Y = aX + b$, then $Y \sim \mathcal{N}(a\mu + b, a^2\sigma^2)$. The sum of independent normal variables is also normal with $\mu = \sum_i \mu_i$ and $\sigma^2 = \sum_i \sigma_i^2$. If X is $\mathcal{N}(\mu, \sigma^2)$, then

$$\frac{X - \mu}{\sigma} \sim \mathcal{Z} \tag{A.44}$$

This is called z-normalization.

CENTRAL LIMIT THEOREM

Let $X_1, X_2, \ldots, X_N$ be a set of iid random variables all having mean μ and variance σ^2. Then the *central limit theorem* states that for large N, the distribution of

$$X_1 + X_2 + \ldots + X_N \tag{A.45}$$

is approximately $\mathcal{N}(N\mu, N\sigma^2)$. For example, if X is binomial with parameters (N, p), X can be written as the sum of N Bernoulli trials and $(X - Np)/\sqrt{Np(1-p)}$ is approximately unit normal.

Central limit theorem is also used to generate normally distributed random variables on computers. Programming languages have subroutines that return uniformly distributed (pseudo-)random numbers in the range $[0, 1]$. When U_i are such random variables, $\sum_{i=1}^{12} U_i - 6$ is approximately $\mathcal{Z}$.

Let us say $X^t \sim \mathcal{N}(\mu, \sigma^2)$. The estimated sample mean

$$m = \frac{\sum_{t=1}^{N} X^t}{N} \tag{A.46}$$

is also normal with mean μ and variance σ^2/N.

A.3.6 Chi-Square Distribution

If Z_i are independent unit normal random variables, then

$$X = Z_1^2 + Z_2^2 + \ldots + Z_n^2 \tag{A.47}$$

is chi-square with n degrees of freedom, namely, $X \sim \mathcal{X}_n^2$, with

$$E[X] = n, \ \mathrm{Var}(X) = 2n \tag{A.48}$$

When $X^t \sim \mathcal{N}(\mu, \sigma^2)$, the estimated sample variance is

$$S^2 = \frac{\sum_t (X^t - m)^2}{N - 1} \tag{A.49}$$

and we have

$$(N-1)\frac{S^2}{\sigma^2} \sim \mathcal{X}_{N-1}^2 \tag{A.50}$$

It is also known that m and S^2 are independent.

A.3.7 *t* Distribution

If $Z \sim \mathcal{Z}$ and $X \sim \mathcal{X}^2_n$ are independent, then

$$T_n = \frac{Z}{\sqrt{X/n}} \tag{A.51}$$

is t-distributed with n degrees of freedom with

$$E[T_n] = 0, n > 1,\ \mathrm{Var}(T_n) = \frac{n}{n-2}, n > 2 \tag{A.52}$$

Like the unit normal density, t is symmetric around 0. As n becomes larger, t density becomes more and more like the unit normal, the difference being that t has thicker tails, indicating greater variability than does normal.

A.3.8 *F* Distribution

If $X_1 \sim \mathcal{X}^2_n$ and $X_2 \sim \mathcal{X}^2_m$ are independent chi-square random variables with n and m degrees of freedom, respectively,

$$F_{n,m} = \frac{X_1/n}{X_2/m} \tag{A.53}$$

is F-distributed with n and m degrees of freedom with

$$E[F_{n,m}] = \frac{m}{m-2}, m > 2,\ \mathrm{Var}(F_{n,m}) = \frac{m^2(2m+2n-4)}{n(m-2)^2(m-4)}, m > 4 \tag{A.54}$$

A.4 References

Casella, G., and R. L. Berger. 1990. *Statistical Inference.* Belmont, CA: Duxbury.

Ross, S. M. 1987. *Introduction to Probability and Statistics for Engineers and Scientists.* New York: Wiley.

Index

Adaptive Computation and Machine Learning

Thomas Dietterich, Editor
Christopher Bishop, David Heckerman, Michael Jordan, and Michael Kearns, Associate Editors

Bioinformatics: The Machine Learning Approach, Pierre Baldi and Søren Brunak

Reinforcement Learning: An Introduction, Richard S. Sutton and Andrew G. Barto

Graphical Models for Machine Learning and Digital Communication, Brendan J. Frey

Learning in Graphical Models, Michael I. Jordan

Causation, Prediction, and Search, second edition, Peter Spirtes, Clark Glymour, and Richard Scheines

Principles of Data Mining, David Hand, Heikki Mannila, and Padhraic Smyth

Bioinformatics: The Machine Learning Approach, second edition, Pierre Baldi and Søren Brunak

Learning Kernel Classifiers: Theory and Algorithms, Ralf Herbrich

Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond, Bernhard Schölkopf and Alexander J. Smola

Introduction to Machine Learning, Ethem Alpaydın

Gaussian Processes for Machine Learning, Carl Edward Rasmussen and Christopher K. I. Williams

Semi-Supervised Learning, Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien, eds.

The Minimum Description Length Principle, Peter D. Grünwald

Introduction to Statistical Relational Learning, Lise Getoor and Ben Taskar, eds.

Probabilistic Graphical Models: Principles and Techniques, Daphne Koller and Nir Friedman

Introduction to Machine Learning, second edition, Ethem Alpaydın

Machine Learning in Non-Stationary Environments: Introduction to Covariate Shift Adaptation, Masashi Sugiyama and Motoaki Kawanabe

Boosting: Foundations and Algorithms, Robert E. Schapire and Yoav Freund

Machine Learning: A Probabilistic Perspective, Kevin P. Murphy

Foundations of Machine Learning, Mehryar Mohri, Afshin Rostami, and Ameet Talwalker

Introduction to Machine Learning, third edition, Ethem Alpaydın