Lecture Notes

on

Machine Learning (15A05706)

UNIT-1

1. What Is Machine Learning?

Machine learning is programming computers to optimize a performance criterion using example data or past experience. We have a model defined up to some parameters, and learning is the execution of a computer program to optimize the parameters of the model using the training data or past experience. The model may be *predictive* to make predictions in the future, or *descriptive* to gain knowledge from data, or both.

Machine learning uses the theory of statistics in building mathematical models, because the core task is making inference from a sample. The role of computer science is twofold: First, in training, we need efficient algorithms to solve the optimization problem, as well as to store and process the massive amount of data we generally have. Second, once a model is learned, its representation and algorithmic solution for inference needs to be efficient as well. In certain applications, the efficiency of the learning or inference algorithm, namely, its space and time complexity, may be as important as its predictive accuracy. Application of machine learning methods to large databases is called *data mining*. The analogy is that a large volume of earth and raw material is extracted from a mine, which when processed leads to a small amount of very precious material.

Its application areas are abundant: In addition to retail, in finance banks analyze their past data to build models to use in credit applications, fraud detection, and the stock market. In manufacturing, learning models are used for optimization, control, and troubleshooting. In medicine, learning programs are used for medical diagnosis. In telecommunications, call patterns are analyzed for network optimization and maximizing the quality of service. In science, large amounts of data in physics, astronomy, and biology can only be analyzed fast enough by computers. The World Wide Web is huge; it is constantly growing, and searching for relevant information cannot be done manually. But machine learning is not just a database problem; it is also a part of artificial intelligence. To be intelligent, a system that is in a changing environment should have the ability to learn. If the system can learn and adapt to such changes, the system designer need not foresee and provide solutions for all possible situations. Machine learning also helps us find solutions to many problems in vision, speech recognition, and robotics.

2. Learning a Class from Examples

Let us say we want to learn the *class*, C, of a "family car." We have a set of examples of cars, and we have a group of people that we survey to whom we show these cars. The people look at the cars and label them; the cars that they believe are family cars are *positive examples*, and the other cars are *negative examples*. Class learning is finding a description that is shared by all positive examples and none of the negative examples.

Doing this, we can make a prediction: Given a car that we have not seen before, by checking with the description learned, we will be able to say whether it is a family car or not. Or we can do knowledge extraction: This study may be sponsored by a car company, and the aim may be to understand what people expect from a family car.

After some discussions with experts in the field, let us say that we reach the conclusion that among all features a car may have, the features that separate a family car from other cars are the price and engine power. These two attributes are the *inputs* to the class recognizer. Note that when we decide on this particular *input representation*, we are ignoring various other attributes as irrelevant. Though one may

think of other attributes such as seating capacity and color that might be important for distinguishing among car types, we will consider only price and engine power to keep this example simple.

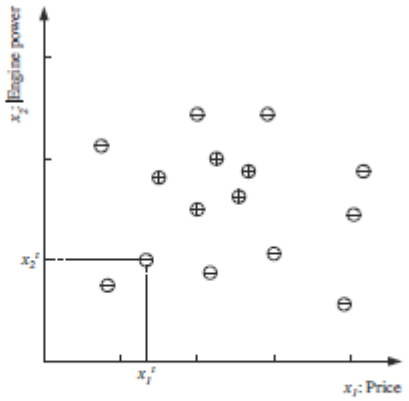


Figure 2.1 Training set for the class of a "family car." Each data point corresponds to one example car, and the coordinates of the point indicate the price and engine power of that car. '+' denotes a positive example of the class (a family car), and '−' denotes a negative example (not a family car); it is another type of car.

Let us denote price as the first input attribute $x1$ (e.g., in U.S. dollars) and engine power as the second attribute $x2$ (e.g., engine volume in cubic centimeters). Thus we represent each car using two numeric values.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

and its label denotes its type

$$r = \begin{cases} 1 & \text{if } \mathbf{x} \text{ is a positive example} \\ 0 & \text{if } \mathbf{x} \text{ is a negative example} \end{cases}$$

Each car is represented by such an ordered pair $(\mathbf{x}, r)$ and the training set contains N such examples

$$\mathcal{X} = \{\mathbf{x}^t, r^t\}_{t=1}^N$$

where t indexes different examples in the set; it does not represent time or any such order.

we may have reason to believe that for a car to be a family car, its price and engine power should be in a certain range.

$$(p_1 \leq \text{price} \leq p_2) \text{ AND } (e_1 \leq \text{engine power} \leq e_2)$$

for suitable values of p_1, p_2, e_1, and e_2. Equation 2.4 thus assumes C to be a rectangle in the price-engine power space (see figure 2.2).

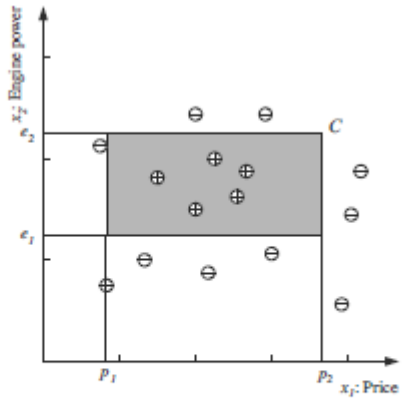


Figure 2.2 Example of a hypothesis class. The class of family car is a rectangle in the price-engine power space.

Margin: The *margin*, which is the distance between the boundary and the instances closest to it.

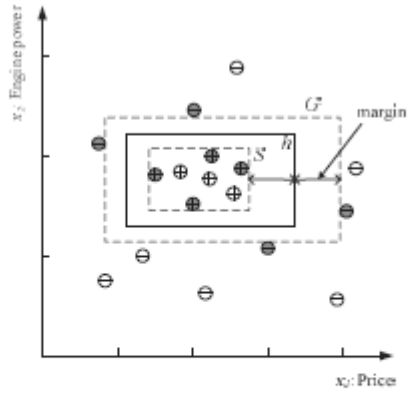


Figure 2.5 We choose the hypothesis with the largest margin, for best separation. The shaded instances are those that define (or support) the margin; other instances can be removed without affecting h.

Doubt:
In some applications, a wrong decision may be very costly and in such a case, we can say that any instance that falls in between *S* and *G* is a case of *doubt*, which we cannot label with certainty due to lack of data. In such a case, the system *rejects* the instance and defers the decision to a human expert.

3. Vapnik-Chervonenkis (VC) Dimension

Let us say we have a dataset containing *N* points. These *N* points can be labeled in 2^N ways as positive and negative. Therefore, 2^N different learning problems can be defined by *N* data points. If for any of these problems, we can find a hypothesis $h \in H$ that separates the positive examples from the negative, then we say H *shatters N* points. That is, any learning problem definable by *N* examples can be learned with no error by a hypothesis drawn from H. The maximum number of points that can be shattered by H is called the *Vapnik-Chervonenkis* (VC) *dimension* of H, is denoted as *VC(*H*)*, and measures the *capacity* of H.

In figure 2.6, we see that an axis-aligned rectangle can shatter four points in two dimensions. Then *VC(*H*)*, when H is the hypothesis class of axis-aligned rectangles in two dimensions, is four. In calculating

the VC dimension, it is enough that we find four points that can be shattered; it is not necessary that we be able to shatter *any* four points in two dimensions.

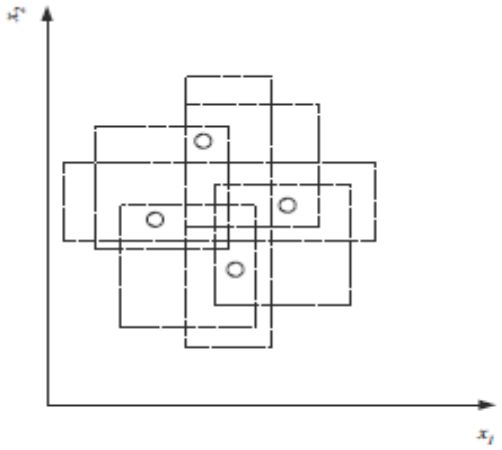


Figure 2.6 An axis-aligned rectangle can shatter four points. Only rectangles covering two points are shown.

VC dimension may seem pessimistic. It tells us that using a rectangle as our hypothesis class, we can learn only datasets containing four points and not more.

4. **Probably Approximately Correct (PAC) Learning**

Using the tightest rectangle, *S*, as our hypothesis, we would like to find how many examples we need. We would like our hypothesis to be approximately correct, namely, that the error probability be bounded by some value. We also would like to be confident in our hypothesis in that we want to know that our hypothesis will be correct most of the time (if not always); so we want to be probably correct as well (by a probability we can specify).

PAC learning In *Probably Approximately Correct* (PAC) *learning*, given a class, C, and examples drawn from some unknown but fixed probability distribution, *p(x)*, we want to find the number of examples, *N*, such that with probability at least $1 - \delta$, the hypothesis *h* has error at most ϵ, for arbitrary.

$$\delta \leq 1/2 \text{ and } \epsilon > 0$$

$$P\{C \Delta h \leq \epsilon\} \geq 1 - \delta$$

where $C \Delta h$ is the region of difference between C and h.

In our case, because S is the tightest possible rectangle, the error region between C and $h = S$ is the sum of four rectangular strips (see figure 2.7). We would like to make sure that the probability of a positive example falling in here (and causing an error) is at most ϵ. For any of these strips, if we can guarantee that the probability is upper bounded by $\epsilon/4$, the error is at most $4(\epsilon/4) = \epsilon$. Note that we count the overlaps in the corners twice, and the total actual error in this case is less than $4(\epsilon/4)$. The probability that a randomly drawn example misses this strip is $1 - \epsilon/4$. The probability that all N independent draws miss the strip is $(1-\epsilon/4)^N$, and the probability that all N independent draws miss any of the four strips is at most $4(1 - \epsilon/4)^N$, which we would like to be at most δ. We have the inequality

$$(1 - x) \leq \exp[-x]$$

So if we choose N and δ such that we have

$$4 \exp[-\epsilon N/4] \leq \delta$$

we can also write $4(1 - \epsilon/4)^N \leq \delta$. Dividing both sides by 4, taking (natural) log and rearranging terms, we have

$$N \geq (4/\epsilon) \log(4/\delta)$$

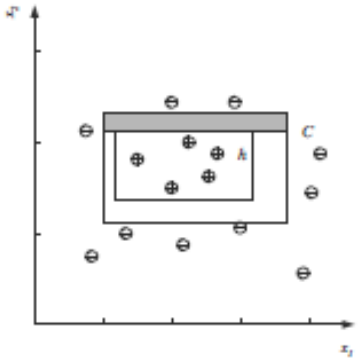


Figure 2.7 The difference between h and C is the sum of four rectangular strips, one of which is shaded.

Therefore, provided that we take at least $(4/\epsilon)\log(4/\delta)$ independent examples from C and use the tightest rectangle as our hypothesis h, with *confidence probability* at least $1 - \delta$, a given point will be misclassified with *error probability* at most ϵ. We can have arbitrary large confidence by decreasing δ and arbitrary small error by decreasing ϵ, and we see in equation 2.7 that the number of examples is a slowly growing function of $1/\epsilon$ and $1/\delta$, linear and logarithmic, respectively.

5. **Noise**

Noise is any unwanted anomaly in the data and due to noise, the class may be more difficult to learn and zero error may be infeasible with a simple hypothesis class (see figure 2.8). There are several interpretations of noise:

_ There may be imprecision in recording the input attributes, which may shift the data points in the input space.

_ There may be errors in labeling the data points, which may relabel positive instances as negative and vice versa. This is sometimes called *teacher noise*.

_ There may be additional attributes, which we have not taken into account, that affect the label of an instance. Such attributes may be *hidden* or *latent* in that they may be unobservable. The effect of these neglected attributes is thus modeled as a random component and is included in "noise."

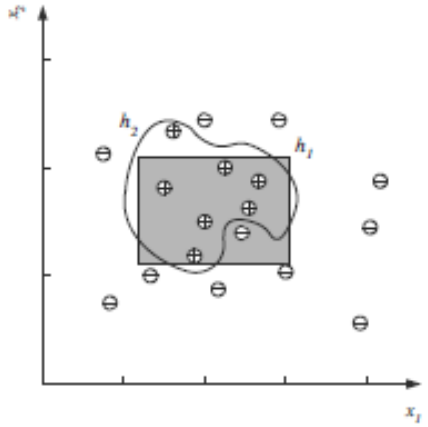


Figure 2.8 When there is noise, there is not a simple boundary between the positive and negative instances, and zero misclassification error may not be possible with a simple hypothesis. A rectangle is a simple hypothesis with four parameters defining the corners. An arbitrary closed form can be drawn by piecewise functions with a larger number of control points.

Using the simple rectangle (unless its training error is much bigger) makes more sense because of the following:

1. It is a simple model to use. It is easy to check whether a point is inside or outside a rectangle and we can easily check, for a future data instance, whether it is a positive or a negative instance.

2. It is a simple model to train and has fewer parameters. It is easier to find the corner values of a rectangle than the control points of an arbitrary shape. With a small training set when the training instances differ a little bit, we expect the simpler model to change less than a complex model: A simple model is thus said to have less *variance*. On the other hand, a too simple model assumes more, is more rigid, and may fail if

indeed the underlying class is not that simple: A simpler model has more *bias*. Finding the optimal model corresponds to minimizing both the bias and the variance.

3. It is a simple model to explain. A rectangle simply corresponds to defining intervals on the two attributes. By learning a simple model, we can extract information from the raw data given in the training set.

4. If indeed there is mislabeling or noise in input and the actual class is really a simple model like the rectangle, then the simple rectangle, because it has less variance and is less affected by single instances, will be a better discriminator than the wiggly shape, although the simple one may make slightly more errors on the training set. Given comparable empirical error, we say that a simple (but not too simple) model would generalize better than a complex model. This principle
is known as *Occam's razor*, which states that *simpler explanations are more plausible* and any unnecessary complexity should be shaved off.

6. Learning Multiple Classes

In the general case, we have K classes denoted as Ci, $i = 1, \ldots, K$, and an input instance belongs to one and exactly one of them. The training set is now of the form

$$\mathcal{X} = \{\boldsymbol{x}^t, \boldsymbol{r}^t\}_{t=1}^N$$

where $\boldsymbol{r}$ has K dimensions and

$$r_i^t = \begin{cases} 1 & \text{if } \boldsymbol{x}^t \in C_i \\ 0 & \text{if } \boldsymbol{x}^t \in C_j, j \neq i \end{cases}$$

An example is given in figure 2.9 with instances from three classes: family car, sports car, and luxury sedan. In machine learning for classification, we would like to learn the boundary separating the instances of one class from the instances of all other classes. Thus we view a K-class classification problem as K two-class problems. The training examples belonging to C_i are the positive instances of hypothesis h_i and the examples of all other classes are the negative instances of h_i . Thus in a K-class problem, we have K hypotheses to learn such that

$$h_i(\boldsymbol{x}^t) = \begin{cases} 1 & \text{if } \boldsymbol{x}^t \in C_i \\ 0 & \text{if } \boldsymbol{x}^t \in C_j, j \neq i \end{cases}$$

The total empirical error takes a sum over the predictions for all classes over all instances:

$$E(\{h_i\}_{i=1}^K | \mathcal{X}) = \sum_{t=1}^N \sum_{i=1}^K 1(h_i(\boldsymbol{x}^t) \neq r_i^t)$$

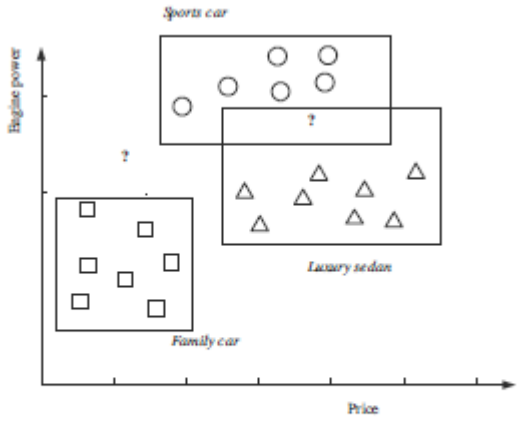


Figure 2.9 There are three classes: family car, sports car, and luxury sedan. There are three hypotheses induced, each one covering the instances of one class and leaving outside the instances of the other two classes. '?' are reject regions where no, or more than one, class is chosen.

For a given $\boldsymbol{x}$, ideally only one of $h_i(\boldsymbol{x})$, $i = 1, \ldots, K$ is 1 and we can choose a class. But when no, or two or more, $h_i(\boldsymbol{x})$ is 1, we cannot choose a class, and this is the case of *doubt* and the classifier *rejects* such cases.
If in a dataset, we expect to have all classes with similar distribution— shapes in the input space—then the same hypothesis class can be used for all classes.

7. **Regression**

In classification, given an input, the output that is generated is Boolean;
it is a yes/no answer. When the output is a numeric value, what we would like to learn is not a class, $C(\boldsymbol{x}) \in \{0, 1\}$, but is a numeric function. In machine learning, the function is not known but we have a training set of examples drawn from it.

$$\mathcal{X} = \{\boldsymbol{x}^t, r^t\}_{t=1}^N$$

where $r^t \in \Re$. If there is no noise, the task is *interpolation*. We would like to find the function $f(x)$ that passes through these points such that we have

$$r^t = f(\boldsymbol{x}^t)$$

In *polynomial interpolation*, given N points, we find the $(N-1)$st degree polynomial that we can use to predict the output for any $\boldsymbol{x}$. This is called *extrapolation* if $\boldsymbol{x}$ is outside of the range of $\boldsymbol{x}^t$ in the training set. In time-series prediction, for example, we have data up to the present and we want to predict the value for the future. In *regression*, there is noise added to the output of the unknown function

$$r^t = f(\boldsymbol{x}^t) + \epsilon$$

where $f(\boldsymbol{x}) \in \Re$ is the unknown function and ϵ is random noise. The explanation for noise is that there are extra *hidden* variables that we cannot observe

$$r^t = f^*(\boldsymbol{x}^t, \boldsymbol{z}^t)$$

where $\boldsymbol{z}^t$ denote those hidden variables. We would like to approximate the output by our model $g(\boldsymbol{x})$. The empirical error on the training set $\mathcal{X}$ is

$$E(g|\mathcal{X}) = \frac{1}{N}\sum_{t=1}^{N}[r^t - g(\boldsymbol{x}^t)]^2$$

Because r and $g(\boldsymbol{x})$ are numeric quantities, for example, $\in \Re$, there is an ordering defined on their values and we can define a *distance* between values, as the square of the difference, which gives us more information than equal/not equal, as used in classification. The square of the difference is one error (loss) function that can be used; another is the absolute value of the difference. We will see other examples in the coming chapters.

Our aim is to find $g(\cdot)$ that minimizes the empirical error. Again our approach is the same; we assume a hypothesis class for $g(\cdot)$ with a small set of parameters. If we assume that $g(\boldsymbol{x})$ is linear, we have

$$g(\boldsymbol{x}) = w_1 x_1 + \cdots + w_d x_d + w_0 = \sum_{j=1}^{d} w_j x_j + w_0$$

We have

$$g(x) = w_1 x + w_0$$

where w_1 and w_0 are the parameters to learn from data. The w_1 and w_0 values should minimize

$$E(w_1, w_0|\mathcal{X}) = \frac{1}{N}\sum_{t=1}^{N}[r^t - (w_1 x^t + w_0)]^2$$

Its minimum point can be calculated by taking the partial derivatives of E with respect to w_1 and w_0, setting them equal to 0, and solving for the two unknowns:

$$\begin{aligned} w_1 &= \frac{\sum_t x^t r^t - \overline{xr}N}{\sum_t (x^t)^2 - N\overline{x}^2} \\ w_0 &= \overline{r} - w_1 \overline{x} \end{aligned}$$

where $\overline{x} = \sum_t x^t / N$ and $\overline{r} = \sum_t r^t / N$. The line found is shown in figure 1.2.

If the linear model is too simple, it is too constrained and incurs a large approximation error, and in such a case, the output may be taken as a higher-order function of the input—for example, quadratic

$$g(x) = w_2 x^2 + w_1 x + w_0$$

8. **Dimensions of a Supervised Machine Learning Algorithm**

Let us now recapitulate and generalize. We have a sample

$$\mathcal{X} = \{x^t, r^t\}_{t=1}^N$$

The sample is *independent and identically distributed* (*iid*); the ordering is not important and all instances are drawn from the same joint distribution *p(x, r)*. *t* indexes one of the *N* instances, *xt* is the arbitrary dimensional input, and *rt* is the associated desired output. *rt* is 0/1 for two-class learning, is a *K*-dimensional binary vector (where exactly one of the dimensions is 1 and all others 0) for *(K* > 2*)*-class classification, and is a real value in regression.

The aim is to build a good and useful approximation to *rt* using the model *g(xt |θ)*. In doing this, there are three decisions we must make:

1. *Model* we use in learning, denoted as
g(x|θ)
where *g(·)* is the model, *x* is the input, and *θ* are the parameters.
g(·) defines the hypothesis class H, and a particular value of *θ* instantiates one hypothesis *h* ∈ H.

For example, in class learning, we have taken a rectangle as our model whose four coordinates make up *θ*; in linear regression, the model is the linear function of the input whose slope and intercept are the parameters learned from the data.

The model (inductive bias), or H, is fixed by the machine learning system designer based on his or her knowledge of the application and the hypothesis *h* is chosen (parameters are tuned) by a learning algorithm using the training set, sampled from *p(x, r)*.
2. *Loss function*, *L(·)*, to compute the difference between the desired output, r^t , and our approximation to it, $g(x^t |\theta)$, given the current value of the parameters, *θ*. The *approximation error*, or *loss*, is the sum of losses over the individual instances.

$$E(\theta|\mathcal{X}) = \sum_t L(r^t, g(x^t|\theta))$$

In class learning where outputs are 0/1, *L(·)* checks for equality or not; in regression, because the output is a numeric value, we have ordering information for distance and one possibility is to use the square of the difference.
3. *Optimization procedure* to find $\theta*$ that minimizes the total error

$$\theta^* = \arg\min_{\theta} E(\theta|X)$$

9. **Decision tree representation**

Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. Learned trees can also be re-represented as sets of if-then rules to improve human readability.

These learning methods are among the most popular of inductive inference algorithms
and have been successfully applied to a broad range of tasks from learning to diagnose medical cases to learning to assess credit risk of loan applicants.

Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some attribute of the instance, and each branch descending from that node corresponds to one of the possible values for this attribute. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node.

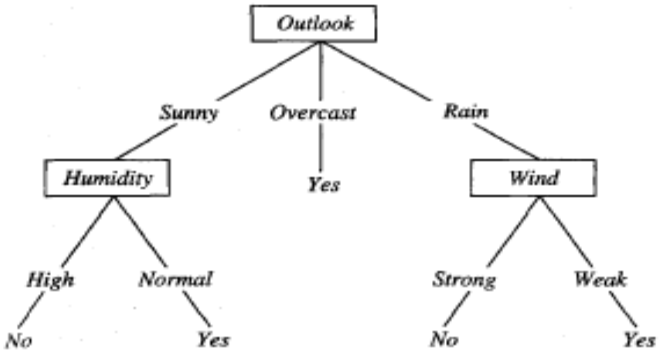


FIGURE 3.1
A decision tree for the concept *PlayTennis*. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf (in this case, *Yes* or *No*). This tree classifies Saturday mornings according to whether or not they are suitable for playing tennis.

Figure ***3.1*** illustrates a typical learned decision tree. This decision tree classifies
Saturday mornings according to whether they are suitable for playing tennis.
For example, the instance
(Outlook = ***Sunny, Temperature*** = ***Hot, Humidity*** = ***High, Wind*** = ***Strong***

In general, decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances. Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree itself to a disjunction of these conjunctions. For example, the decision tree shown in Figure **3.1** corresponds to the expression
(Outlook = ***Sunny*** **A** ***Humidity*** = ***Normal)***
V ***(Outlook*** = ***Overcast)***
v ***(Outlook*** = ***Rain A Wind*** = ***Weak)***

10. Appropriate problems for decision tree learning

Although a variety of decision tree learning methods have been developed with somewhat differing capabilities and requirements, decision tree learning is generally best suited to problems with the following characteristics:

Znstances are represented by attribute-value pairs. Instances are described by a fixed set of attributes (e.g., ***Temperature)*** and their values (e.g., ***Hot).*** The easiest situation for decision tree learning is when each attribute takes on a small number of disjoint possible values (e.g., ***Hot, Mild, Cold).*** However, extensions to the basic algorithm (discussed in Section 3.7.2) allow handling real-valued attributes as well (e.g., representing ***Temperature*** numerically).

The targetfunction has discrete output values. The decision tree in Figure 3.1 assigns a boolean classification (e.g., ***yes*** or ***no)*** to each example. Decision tree methods easily extend to learning functions with more than two possible output values. A more substantial extension allows learning target functions with real-valued outputs, though the application of decision trees in this
setting is less common.

0 Disjunctive descriptions may be required. As noted above, decision trees naturally represent disjunctive expressions.

0 The training data may contain errors. Decision tree learning methods are robust to errors, both errors in classifications of the training examples and errors in the attribute values that describe these examples.

0 The training data may contain missing attribute values. Decision tree methods can be used even when some training examples have unknown values (e.g., if the ***Humidity*** of the day is known for only some of the training.

11. The basic decision tree learning algorithm

A. Which Attribute Is the Best Classifier?

The central choice in the ID3 algorithm is selecting which attribute to test at each node in the tree. We would like to select the attribute that is most useful for classifying examples. What is a good quantitative measure of the worth of an attribute? We will define a statistical property, called ***informution gain,*** that measures how well a given attribute separates the training examples according to their target classification. ID3 uses this information gain measure to select among
the candidate attributes at each step while growing the tree.

B. Entropy measures homogeneity of examples

In order to define information gain precisely, we begin by defining a measure commonly
used in information theory, called ***entropy,*** that characterizes the (im)purity of an arbitrary collection of examples. Given a collection S, containing positive and negative examples of some target concept, the entropy of S relative to this boolean classification is

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

ID3($Examples$, $Target\_attribute$, $Attributes$)

Examples are the training examples. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given Examples.

- Create a *Root* node for the tree
- If all $Examples$ are positive, Return the single-node tree *Root*, with label = +
- If all $Examples$ are negative, Return the single-node tree *Root*, with label = −
- If $Attributes$ is empty, Return the single-node tree *Root*, with label = most common value of $Target\_attribute$ in $Examples$
- Otherwise Begin
  - $A \leftarrow$ the attribute from $Attributes$ that best* classifies $Examples$
  - The decision attribute for *Root* $\leftarrow A$
  - For each possible value, v_i, of A,
    - Add a new tree branch below *Root*, corresponding to the test $A = v_i$
    - Let $Examples_{v_i}$ be the subset of $Examples$ that have value v_i for A
    - If $Examples_{v_i}$ is empty
      - Then below this new branch add a leaf node with label = most common value of $Target\_attribute$ in $Examples$
      - Else below this new branch add the subtree ID3($Examples_{v_i}$, $Target\_attribute$, $Attributes - \{A\}$))
- End
- Return *Root*

* The best attribute is the one with highest *information gain*, as defined in Equation (3.4).

TABLE 3.1
Summary of the ID3 algorithm specialized to learning boolean-valued functions. ID3 is a greedy algorithm that grows the tree top-down, at each node selecting the attribute that best classifies the local training examples. This process continues until the tree perfectly classifies the training examples, or until all attributes have been used.

where $p_\oplus$ is the proportion of positive examples in S and $p_\ominus$ is the proportion of negative examples in S. In all calculations involving entropy we define $0 \log 0$ to be 0.

To illustrate, suppose S is a collection of 14 examples of some boolean concept, including 9 positive and 5 negative examples (we adopt the notation [9+, 5−] to summarize such a sample of data). Then the entropy of S relative to this boolean classification is

$$Entropy([9+, 5-]) = -(9/14)\log_2(9/14) - (5/14)\log_2(5/14)$$
$$= 0.940 \qquad (3.2)$$

Notice that the entropy is 0 if all members of S belong to the same class. For example, if all members are positive ($p_\oplus = 1$), then $p_\ominus$ is 0, and $Entropy(S) = -1 \cdot \log_2(1) - 0 \cdot \log_2 0 = -1 \cdot 0 - 0 \cdot \log_2 0 = 0$. Note the entropy is 1 when the collection contains an equal number of positive and negative examples. If the collection contains unequal numbers of positive and negative examples, the

entropy is between 0 and 1. Figure 3.2 shows the form of the entropy function relative to a boolean classification, as $p_\oplus$ varies between 0 and 1.

One interpretation of entropy from information theory is that it specifies the minimum number of bits of information needed to encode the classification of an arbitrary member of S (i.e., a member of S drawn at random with uniform probability). For example, if $p_\oplus$ is 1, the receiver knows the drawn example will be positive, so no message need be sent, and the entropy is zero. On the other hand, if $p_\oplus$ is 0.5, one bit is required to indicate whether the drawn example is positive or negative. If $p_\oplus$ is 0.8, then a collection of messages can be encoded using on average less than 1 bit per message by assigning shorter codes to collections of positive examples and longer codes to less likely negative examples.

Thus far we have discussed entropy in the special case where the target classification is boolean. More generally, if the target attribute can take on c different values, then the entropy of S relative to this c-wise classification is defined as

$$Entropy(S) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i \qquad (3.3)$$

where p_i is the proportion of S belonging to class i. Note the logarithm is still base 2 because entropy is a measure of the expected encoding length measured in *bits*. Note also that if the target attribute can take on c possible values, the entropy can be as large as $\log_2 c$.

C. Information gain measures the expected reduction in entropy

Given entropy as a measure of the impurity in a collection of training examples, we can now define a measure of the effectiveness of an attribute in classifying the training data. The measure we will use, called ***information gain,*** is simply the expected reduction in entropy caused by partitioning the examples according to this attribute. More precisely, the information gain, ***Gain(S, A)*** of ***an*** attribute **A,** relative to a collection of examples ***S,*** is defined as

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

For example, suppose S is a collection of training-example days described by attributes including *Wind*, which can have the values *Weak* or *Strong*. As before, assume S is a collection containing 14 examples, [9+, 5−]. Of these 14 examples, suppose 6 of the positive and 2 of the negative examples have *Wind* = *Weak*, and the remainder have *Wind* = *Strong*. The information gain due to sorting the original 14 examples by the attribute *Wind* may then be calculated as

$$\begin{aligned}
Values(Wind) &= Weak, Strong \\
S &= [9+, 5-] \\
S_{Weak} &\leftarrow [6+, 2-] \\
S_{Strong} &\leftarrow [3+, 3-] \\
Gain(S, Wind) &= Entropy(S) - \sum_{v \in \{Weak, Strong\}} \frac{|S_v|}{|S|} Entropy(S_v) \\
&= Entropy(S) - (8/14) Entropy(S_{Weak}) \\
&\quad - (6/14) Entropy(S_{Strong}) \\
&= 0.940 - (8/14)0.811 - (6/14)1.00 \\
&= 0.048
\end{aligned}$$

12. Hypothesis space search in decision tree learning

As with other inductive learning methods, ID3 can be characterized as searching a space of hypotheses for one that fits the training examples. The hypothesis space searched by ID3 is the set of possible decision trees. ID3 performs a simple-tocomplex, hill-climbing search through this hypothesis space, beginning with the empty tree, then considering progressively more elaborate hypotheses in search of a decision tree that correctly classifies the training data. The evaluation function

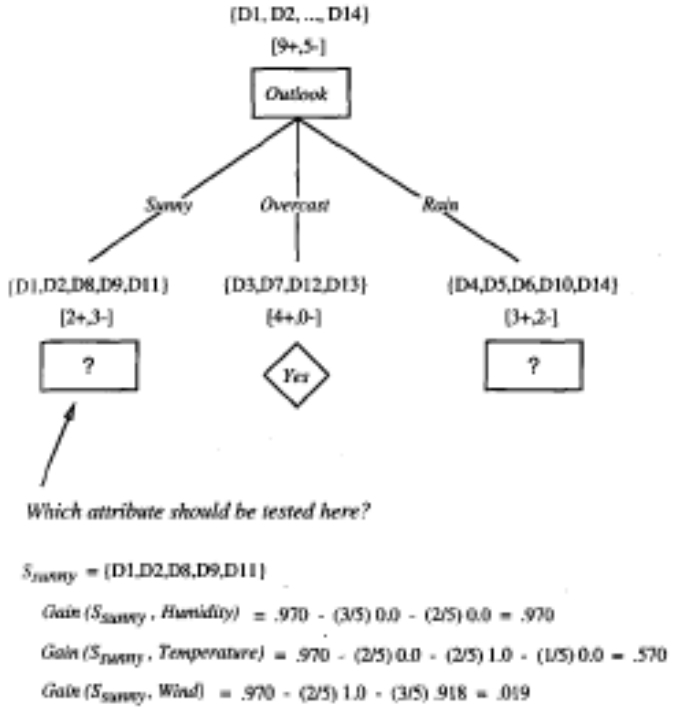


FIGURE 3.4
The partially learned decision tree resulting from the first step of ID3. The training examples are sorted to the corresponding descendant nodes. The *Overcast* descendant has only positive examples and therefore becomes a leaf node with classification *Yes*. The other two nodes will be further expanded, by selecting the attribute with highest information gain relative to the new subsets of examples.

that guides this hill-climbing search is the information gain measure. This search is depicted in Figure 3.5. By viewing ID in terms of its search space and search strategy, we can get some insight into its capabilities and limitations. 1~3's h ypothesis space of all decision trees is a ***complete*** space of finite discrete-valued functions, relative to the available attributes. Because every finite discrete-valued function can be represented by some decision tree, ID3 avoids one of the major risks of methods that search incomplete hypothesis spaces (such as methods that consider only conjunctive hypotheses): that the hypothesis space might not contain the target function.

1. ID3 maintains only a single current hypothesis as it searches through the space of decision trees. This contrasts, for example, with the earlier version space candidate-~lirninat-od, which maintains the set of ***all*** hypotheses consistent with the available training examples. By determining only a single hypothesis, ID^ loses the capabilities that follow from explicitly representing all consistent hypotheses. For example, it does not have the ability to determine how many alternative decision trees are consistent with the available training data, or to pose new instance queries that optimally resolve among these competing hypotheses.

2. **ID3** in its pure form performs no backtracking in its search. Once it,selects an attribute to test at a particular level in the tree, it never backtracks to reconsider this choice. Therefore, it is susceptible to the usual risks of hill-climbing search without backtracking: converging to locally optimal solutions that are not globally optimal. In the case of **ID3,** a locally optimal solution corresponds to the decision tree it selects along the single search path it explores. However, this locally optimal solution may be less desirable than trees that would have been encountered along a different branch of the search. Below we discuss an extension that adds a form of backtracking (post-pruning the decision tree).

3. **ID3** uses all training examples at each step in the search to make statistically based decisions regarding how to refine its current hypothesis. This contrasts with methods that make decisions incrementally, based on individual training examples (e.g., FIND-CANDIDATE-ELIMINATION) advantage of using statistical properties of all the examples (e.g., information gain) is that the resulting search is much less sensitive to errors in individual training examples. **ID3** can be easily extended to handle noisy training data by modifying its termination criterion to accept hypotheses that imperfectly fit the training data.

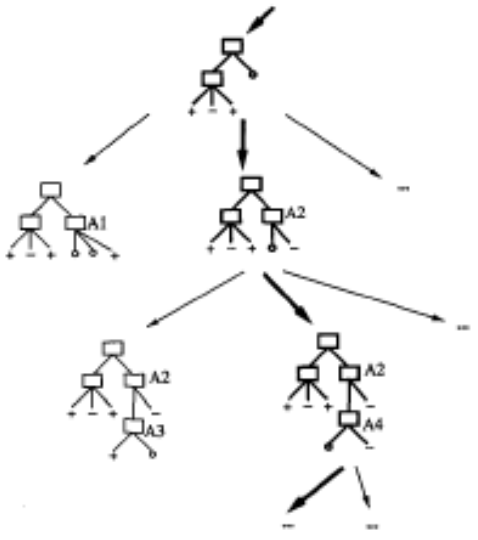


FIGURE 3.5
Hypothesis space search by ID3. ID3 searches through the space of possible decision trees from simplest to increasingly complex, guided by the information gain heuristic.

13. **Inductive bias in decision tree learning**

Given a collection of training examples, there are typically many decision trees consistent with these examples. Describing the inductive bias of ID3 therefore consists of describing the basis by which it chooses one of these consistent hypotheses over the others. Which of these decision trees does ID3 choose?

It chooses the first acceptable tree it encounters in its simple-to-complex, hill climbing search through the space of possible trees. Roughly speaking, then, the ID3 search strategy
(a) selects in favor of shorter trees over longer ones, and
(b) selects trees that place the attributes with highest information gain closest to the root. Because of the subtle interaction between the attribute selection heuristic used by ID3 and the particular training examples it encounters, it is difficult to characterize precisely the inductive bias exhibited by ID3. However, we can approximately characterize its bias as a preference for short decision trees over complex trees.

Approximate inductive bias of ID3: Shorter trees are preferred over larger trees.
In fact, one could imagine an algorithm similar to ID3 that exhibits precisely this inductive bias. Consider an algorithm that begins with the empty tree and searches ***breadth Jirst*** through progressively more complex trees, first considering all trees of depth 1, then all trees of depth 2, etc. Once it finds a decision tree consistent with the training data, it returns the smallest consistent tree at that search depth (e.g., the tree with the fewest nodes). Let us call this breadth-first
search algorithm BFS-ID3. BFS-ID3 finds a shortest decision tree and thus exhibits precisely the bias "shorter trees are preferred over longer trees." ID3 can be viewed as an efficient approximation to BFS-ID3, using a greedy heuristic search to attempt to find the shortest tree without conducting the entire breadth-first search through the hypothesis space.

Because ID3 uses the information gain heuristic and a hill climbing strategy, it exhibits a more complex bias than BFS-ID3. In particular, it does not always find the shortest consistent tree, and it is biased to favor trees that place attributes with high information gain closest to the root.
A closer approximation to the inductive bias of ID3: Shorter trees are preferred over longer trees. Trees that place high information gain attributes close to the root are preferred over those that do not.

A. Restriction Biases and Preference Biases

Consider the difference between the hypothesis space search in these two approaches: ID3 searches a complete hypothesis space (i.e., one capable of expressing any finite discrete-valued function). It searches incompletely through this space, from simple to complex hypotheses, until its termination condition is met (e.g., until it finds a hypothesis consistent with the data). Its inductive bias is solely a consequence of the ordering of hypotheses by its search strategy. Its hypothesis space introduces no additional bias.

The version space CANDIDATE-ELIMINATION algorithm searches an incomplete hypothesis space (i.e., one that can express only a subset of the potentially teachable concepts). It searches this space completely, finding every hypothesis consistent with the training data. Its inductive bias is solely a consequence of the expressive power of its hypothesis representation. Its search strategy introduces no additional bias.

The inductive bias of ID3 is thus a preference for certain hypotheses over others (e.g., for shorter hypotheses), with no hard restriction on the hypotheses that can be eventually enumerated. This form of bias is typically called a preference bias (or, alternatively, a search bias). In contrast, the bias of the CANDIDATEELIMINATION Algorithm is in the form of a categorical restriction on the set of hypotheses considered. This form of bias is typically called a restriction bias (or, alternatively, a language bias).
.

B. **Why Prefer Short Hypotheses?**

Is ID3's inductive bias favoring shorter decision trees a sound basis for generalizing beyond the training data?
William of Occam was one of the first to discuss the question:

Occam's razor: Prefer the simplest hypothesis that fits the data.

Upon closer examination, it turns out there is a major difficulty with the above argument. By the same reasoning we could have argued that one should prefer decision trees containing exactly 17 leaf nodes

with 11 nonleaf nodes, that use the decision attribute **A1** at the root, and test attributes **A2** through **All,** in numerical order.

A second problem with the above argument for Occam's razor is that the size of a hypothesis is determined by the particular representation used ***internally*** by the learner. Two learners using different internal representations could therefore anive at different hypotheses, both justifying their contradictory conclusions by Occam's razor!.

This last argument shows that Occam's razor will produce two different hypotheses from the same training examples when it is applied by two learners that perceive these examples in terms of different internal representations.

16. Issues in decision tree learning

A. Avoiding Overfitting the Data

The algorithm (DECISION TREE) grows each branch of the tree just deeply enough to perfectly classify the training examples. While this is sometimes a reasonable strategy, in fact it can lead to difficulties when there is noise in the data, or when the number of training examples is too small to produce a representative sample of the true target function. In either of these cases, this simple algorithm can produce trees that ***overfit*** the training examples.

Definition: **Given a hypothesis space H, a hypothesis $h \in H$ is said to overfit the training data if there exists some alternative hypothesis $h' \in H$, such that h has smaller error than h' over the training examples, but h' has a smaller error than h over the entire distribution of instances.**

Overfitting is a significant practical difficulty for decision tree learning and many other learning methods.
There are several approaches to avoiding overfitting in decision tree learning.
These can be grouped into two classes:
- approaches that stop growing the tree earlier, before it reaches the point where it perfectly classifies the training data,
- approaches that allow the tree to overfit the data, and then post-prune the tree.

Although the first of these approaches might seem.more direct, the second approach of post-pruning overfit trees has been found to be more successful in practice. This is due to the difficulty in the first approach of estimating precisely when to stop growing the tree. Regardless of whether the correct tree size is found by stopping early or by post-pruning, a key question is what criterion is to be used to determine the
correct final tree size.

Approaches include:

- Use a separate set of examples, distinct from the training examples, to evaluate the utility of post-pruning nodes from the tree.
- Use all the available data for training, but apply a statistical test to estimate whether expanding (or pruning) a particular node is likely to produce an improvement beyond the training set. For example, Quinlan (1986) uses a chi-square test to estimate whether further expanding a node is likely to improve performance over the entire instance distribution, or only on the current sample of training data.
- Use an explicit measure of the complexity for encoding the training examples and the decision tree, halting growth of the tree when this encoding size is minimized. This approach, based on a heuristic called the Minimum Description Length principle.

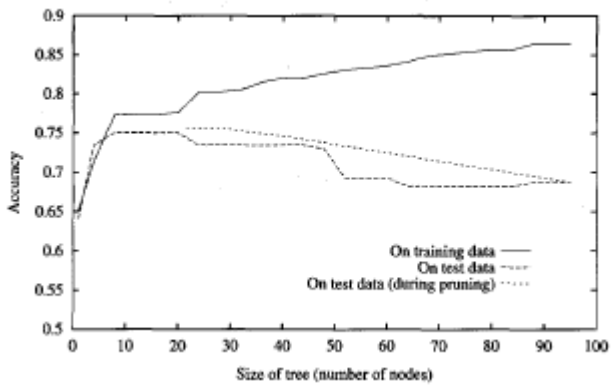
B. Reduced error pruning

How exactly might we use a validation set to prevent overfitting? One approach, called reduced-error pruning, is to consider each of the decision nodes in the.tree to be candidates for pruning. Pruning a decision node consists of removing the subtree rooted at that node, making it a leaf node, and assigning it the most common classification of the training examples affiliated with that node.

Nodes are removed only if the resulting pruned tree performs no worse than-the original over the validation set. This has the effect that any leaf node added due to coincidental regularities in the training set is likely to be pruned because these

same coincidences are unlikely to occur in the validation set. Nodes are pruned iteratively, always choosing the node whose removal most increases the decision tree accuracy over the validation set. Pruning of nodes continues until further pruning is harmful.

The impact of reduced-error pruning on the accuracy of the decision tree is illustrated in Figure **3.7.**



C. Rule post-pruning

Rule post-pruning involves the following steps:
1. Infer the decision tree from the training set, growing the tree until the training data is fit as well as possible and allowing overfitting to occur.
2. Convert the learned tree into an equivalent set of rules by creating one rule for each path from the root node to a leaf node.
3. Prune (generalize) each rule by removing any preconditions that result in improving its estimated accuracy.
4. Sort the pruned rules by their estimated accuracy, and consider them in this sequence when classifying subsequent instances.

Why convert the decision tree to rules before pruning? There are three main advantages.

- Converting to rules allows distinguishing among the different contexts in which a decision node is used. Because each distinct path through the decision tree node produces a distinct rule, the pruning decision regarding that attribute test can be made differently for each path. In contrast, if the tree itself were pruned, the only two choices would be to remove the decision node completely, or to retain it in its original form.
- Converting to rules removes the distinction between attribute tests that occur near the root of the tree and those that occur near the leaves. Thus, we avoid messy bookkeeping issues such as how to reorganize the tree if the root node is pruned while retaining part of the subtree below this test.
- Converting to rules improves readability. Rules are often easier for to understand.

D. Handling attributes with differing costs

In some learning tasks the instance attributes may have associated costs. For example, in learning to classify medical diseases we might describe patients in terms of attributes such **as** Temperature, BiopsyResult, Pulse, BloodTestResults, etc. These attributes vary significantly in their costs, both in terms of monetary cost and cost to patient comfort. In such tasks, we would prefer decision trees that use low-cost attributes where possible, relying on high-cost attributes only when needed to produce reliable classifications.

ID3 can be modified to take into account attribute costs by introducing a cost term into the attribute selection measure. For example, we might divide the ***Gain*** by the cost of the attribute, so that lower-cost attributes would be preferred. While such cost-sensitive measures do not guarantee finding an optimal cost-sensitive
decision tree, they do bias the search in favor of low-cost attributes.

Attribute cost is measured by the number of seconds required to obtain the attribute value by positioning and operating the sonar. They demonstrate that more efficient recognition strategies are learned, without sacrificing

classification accuracy, by replacing the information gain attribute selection measure by the following measure.

$$\frac{Gain^2(S, A)}{Cost(A)}$$

E. Handling training examples with missing attribute values

In certain cases, the available data may be missing values for some attributes. For example, in a medical domain in which we wish to predict patient outcome based on various laboratory tests, it may be that the lab test Blood-Test-Result is available only for a subset of the patients. In such cases, it is common to estimate the missing attribute value based on other examples for which this attribute has a known value.

Consider the situation in which ***Gain(S, A)*** is to be calculated at node ***n*** in the decision tree to evaluate whether the attribute ***A*** is the best attribute to test at this decision node. Suppose that ***(x, c(x))*** is one of the training examples in ***S*** and that the value ***A(x)*** is unknown.

One strategy for dealing with the missing attribute value is to assign it the value that is most common among training examples at node ***n.*** Alternatively, we might assign it the most common value among examples at node ***n*** that have the classification ***c(x).*** The elaborated training example using this estimated value for ***A(x)*** can then be used directly by the existing decision tree learning algorithm.

A second, more complex procedure is to assign a probability to each of the possible values of ***A*** rather than simply assigning the most common value to ***A(x).*** These probabilities can be estimated again based on the observed frequencies of the various values for ***A*** among the examples at node ***n.***

F. Incorporating continuous-valued attributes

Our initial definition of ID3 is restricted to attributes that take on a discrete set of values. First, the target attribute whose value is predicted by the learned tree must be discrete valued. Second, the attributes tested in the decision nodes of the tree must also be discrete valued. This second restriction can easily be removed so that continuous-valued decision attributes can be incorporated into the learned tree. This can be accomplished by dynamically defining new discrete valued attributes that partition the continuous attribute value into a discrete set of intervals.

14. Artificial neural networks

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known.

Neural network representations:

System ALVINN, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways. The input to the neural network is a 30 x 32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle. The network output is the direction in which the vehicle is steered. The ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes. ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways (driving in the left lane of a divided public highway, with other vehicles present).

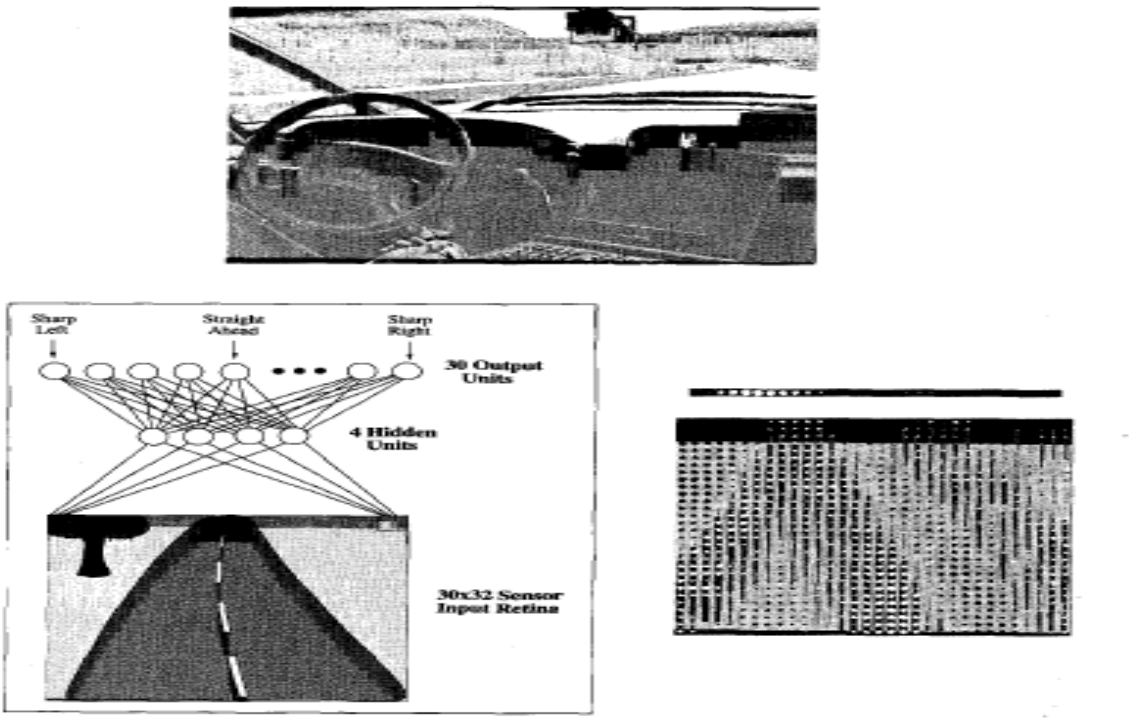


FIGURE 4.1

Figure 4.1 illustrates the neural network representation used in one version of the ALVINN system, and illustrates the kind of representation typical of many ANN systems. The network is shown on the left side of the figure, with the input camera image depicted below it. Each node (i.e., circle) in the network diagram corresponds to the output of a single network ***unit,*** and the lines entering the node from below are its inputs. As can be seen, there are four units that receive inputs directly from all of the 30 x 32 pixels in the image.

These are called "hidden" units because their output is available only within the network and is not available as part of the global network output. Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs. These hidden unit outputs are then used as inputs to a second layer of 30 "output" units.

Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.

The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN. The large matrix of black and white boxes on the lower right depicts the weights from the 30 x 32 pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude.

The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units. The network structure of **ALVINN** is typical of many ANNs. Here the individual units are interconnected in layers that form a directed acyclic graph. In general, ANNs can be graphs with many types of structures-acyclic or cyclic, directed or undirected. This chapter will focus on the most common and practical ANN approaches, which are based on the BACKPROPAGATION algorithm.

17. P**erceptron**

One type of ANN system is based on a unit called a *perceptron*, illustrated in Figure 4.2. A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise. More precisely, given inputs x_1 through x_n, the output $o(x_1, \ldots, x_n)$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

where each w_i is a real-valued constant, or *weight*, that determines the contribution of input x_i to the perceptron output. Notice the quantity $(-w_0)$ is a threshold that the weighted combination of inputs $w_1 x_1 + \cdots + w_n x_n$ must surpass in order for the perceptron to output a 1.

To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as $\sum_{i=0}^{n} w_i x_i > 0$, or in vector form as $\vec{w} \cdot \vec{x} > 0$. For brevity, we will sometimes write the perceptron function as

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

where

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$. Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors.

$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

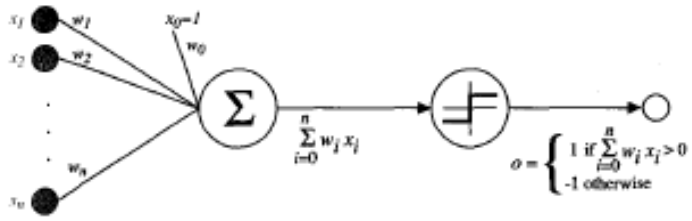


FIGURE 4.2
A perceptron.

A. Representational Power of Perceptrons

We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points). The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side, as illustrated in Figure 4.3. The equation for this decision hyperplane is $\vec{w} \cdot \vec{x} = 0.$. Of course, some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called linearly separable sets of examples.

A single perceptron can be used to represent many boolean functions. For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights *wo* = ***-3,*** and *wl* = *wz* = *.5.*

Perceptrons can represent all of the primitive boolean functions AND, OR, NAND (1 AND), and NOR (1 OR). Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if $x_1 \neq x_2$.

B. The Perceptron Training Rule

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct if 1 output for each of the given training examples.
Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta rule.
These two algorithms are guaranteed to converge to somewhat different acceptable hypotheses, under somewhat different conditions. They are important to ANNs because they provide the basis for learning networks of many units. One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly. Weights are modified at each step according to the ***perceptron training rule,*** which revises the weight w_i associated with input x_i according to the rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Here ***t*** is the target output for the current training example, ***o*** is the output generated by the perceptron, and *q* is a positive constant called the ***learning rate.***

C. Gradient Descent and the Delta Rule

Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable. **A** second training rule, called the ***delta rule,*** is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.

The key idea behind the delta rule is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples. This rule is important because gradient descent provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units. It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

The delta training rule is best understood by considering the task of training an ***unthresholded*** perceptron; that is, a ***linear unit*** for which the output ***o*** is given by

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.
In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples. Although there are many ways to define this error, one common measure that will turn out to be especially convenient is

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

where D is the set of training examples, ***td*** is the target output for training example *d,* and ***od*** is the output of the linear unit for training example d.

D. Derivation of the gradient descent rule

How can we calculate the direction of steepest descent along the error surface? This direction can be found by computing the derivative of E with respect to each component of the vector $\vec{w}$. This vector derivative is called the *gradient* of E with respect to $\vec{w}$, written $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right] \tag{4.3}$$

Notice $\nabla E(\vec{w})$ is itself a vector, whose components are the partial derivatives of E with respect to each of the w_i. *When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in* E. The negative of this vector therefore gives the direction of steepest decrease. For example, the arrow in Figure 4.4 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the w_0, w_1 plane.

Since the gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$$

where

$$\Delta \vec{w} = -\eta \nabla E(\vec{w}) \tag{4.4}$$

Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search. The negative sign is present because we want to move the weight vector in the direction that *decreases* E. This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \tag{4.5}$$

which makes it clear that steepest descent is achieved by altering each component w_i of $\vec{w}$ in proportion to $\frac{\partial E}{\partial w_i}$.

gradient can be obtained by differentiating E from Equation (4.2), as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_{d \in D} (t_d - o_d)(-x_{id}) \qquad (4.6)
\end{aligned}$$

where x_{id} denotes the single input component x_i for training example d. We now have an equation that gives $\frac{\partial E}{\partial w_i}$ in terms of the linear unit inputs x_{id}, outputs O_d, and target values t_d associated with the training examples. Substituting Equation (4.6) into Equation (4.5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id} \tag{4.7}$$

E. **Stochastic approximation to gradient descent**

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever
(1) the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and (2) the error can be differentiated with respect to these hypothesis parameters. The key practical difficulties in applying gradient descent are (1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and (2) if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

One common variation on gradient descent intended to alleviate these difficulties is called ***incremental gradient descent,*** or alternatively ***stochastic gradient descent.***

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

Single perceptrons can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the **BACKPROPAGATION** algorithm are capable of expressing a rich variety of nonlinear decision surfaces.

This section discusses how to learn such multilayer networks using a gradient descent algorithm similar to that discussed in the previous section.

- ### A Differentiable Threshold Unit

At first we might be tempted to choose the linear units discussed in the previous section, for which we have already derived a gradient descent learning rule. However, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions. The perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent. What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

The sigmoid unit is illustrated in Figure 4.6. Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of the sigmoid unit, however, the threshold output is a continuous function of its input. More precisely, the sigmoid unit computes its
output ***o*** as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

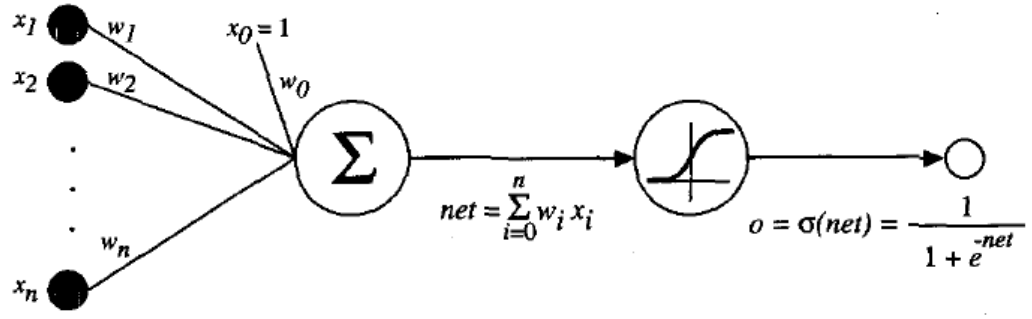


FIGURE 4.6
The sigmoid threshold unit.

- σ is often called the sigmoid function or, alternatively, the logistic function. Note its output ranges Between 0 and 1, increasing monotonically with its input.

The BACKPROPAGATION Algorithm

The BACKPROPAGATION algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs. This section presents the BACKPROPAGATION algorithm, and the following section gives the derivation for the gradient descent weight update rule used by BACKPROPAGATION.

Because we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

where $outputs$ is the set of output units in the network, and t_{kd} and o_{kd} are the target and output values associated with the kth output unit and training example d.

The learning problem faced by BACKPROPAGATION is to search a large hypothesis space defined by all possible weight values for all the units in the network. The situation can be visualized in terms of an error surface similar to that shown for linear units in Figure 4.4. The error in that diagram is replaced by our new definition of E, and the other dimensions of the space correspond now to all of the weights associated with all of the units in the network. As in the case of training a single unit, gradient descent can be used to attempt to find a hypothesis to minimize E.

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji} \tag{T4.5}$$

TABLE 4.2
The stochastic gradient descent version of the BACKPROPAGATION algorithm for feedforward networks

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, *where* $\vec{x}$ *is the vector of network input values, and* $\vec{t}$ *is the vector of target network output values.*

η *is the learning rate (e.g., .05).* n_{in} *is the number of network inputs,* n_{hidden} *the number of units in the hidden layer, and* n_{out} *the number of output units.*

The input from unit i *into unit* j *is denoted* x_{ji}, *and the weight from unit* i *to unit* j *is denoted* w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \tag{T4.5}$$

TABLE 4.2
The stochastic gradient descent version of the BACKPROPAGATION algorithm for feedforward networks containing two layers of sigmoid units.

One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface shown in Figure 4.4. Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error. Despite this obstacle, in practice BACKPROPAGATION has been found to produce excellent results in many real-world applications.

The BACKPROPAGATION algorithm is presented in Table 4.2. The algorithm as described here applies to layered feedforward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer. This is the incremental, or stochastic, gradient descent version of BACKPROPAGATION. The notation used here is the same as that used in earlier sections, with the following extensions:

Notice the algorithm in Table 4.2 begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values. Given this fixed network structure, the main loop of the algorithm then repeatedly iterates over the training examples. For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on this example, then updates all weights in the network. This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.

The gradient descent weight-update rule (Equation [T4.5] in Table 4.2) is similar to the delta training rule (Equation [4.10]). Like the delta rule, it updates each weight in proportion to the learning rate η, the input value x_{ji} to which the weight is applied, and the error in the output of the unit. The only difference is that the error $(t - o)$ in the delta rule is replaced by a more complex error term, δ_j. The exact form of δ_j follows from the derivation of the weight-tuning rule given in Section 4.5.3. To understand it intuitively, first consider how δ_k is computed for each network *output* unit k (Equation [T4.3] in the algorithm). δ_k is simply the familiar $(t_k - o_k)$ from the delta rule, multiplied by the factor $o_k(1 - o_k)$, which is the derivative of the sigmoid squashing function. The δ_h value for each *hidden* unit h has a similar form (Equation [T4.4] in the algorithm). However, since training examples provide target values t_k *only* for network outputs, no target values are directly available to indicate the error of hidden units' values. Instead, the error term for hidden unit h is calculated by summing the error terms δ_k for each output unit influenced by h, weighting each of the δ_k's by w_{kh}, the weight from hidden unit h to output unit k. This weight characterizes the degree to which hidden unit h is "responsible for" the error in output unit k.

15. Remarks on the back propagation algorithm

A. Convergence and Local Minima

Despite the lack of assured convergence to the global minimum error, BACKPROPAGATION is a highly effective function approximation method in practice. When gradient descent falls into a local minimum with respect to one of these weights, it will not necessarily be in a local minimum with respect to the other weights. In fact, the more weights in the network, the more dimensions that might provide "escape routes" for gradient descent to fall away from the local minimum with respect to this single weight.

A second perspective on local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases. Notice that if network weights are initialized to values near zero, then during

early gradient descent steps the network will represent a very smooth function that is approximately linear in its inputs. This is because the sigmoid threshold function itself is approximately linear when the weights are close to zero .

Despite the above comments, gradient descent over the complex error surfaces represented by ANNs is still poorly understood, and no methods are known to predict with certainty when local minima will cause difficulties. Common heuristics to attempt to alleviate the problem of local minima include:

-Add a momentum term to the weight-update rule.
-Use stochastic gradient descent rather than true gradient descent.
-Train multiple networks using the same data, but initializing each network with different random weights.

B. Representational Power of Feedforward Networks

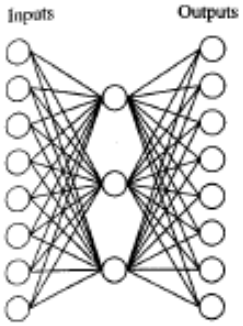
Function classes can be known
Boolean functions: Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs.
Continuous functions: Every bounded continuous function can be approximated with arbitrarily small error (under a finite norm) by a network with two layers of units.
Arbitraryfunctions:Any function can be approximated to arbitrary accuracy by a network with three layers of units.

c. Hidden Layer Representations

One intriguing property of BACKPROPAGATION ability to discover useful intermediate representations at the hidden unit layers inside the network. Because training examples constrain only the network inputs and outputs, the weight-tuning procedure is free to set weights that define whatever hidden unit representation is most effective at minimizing the squared error E. This can lead BACKPROPAGATION to define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.

Consider, for example, the network shown in Figure 4.7. Here, the eight network inputs are connected to three hidden units, which are in turn connected to the eight output units. Because of this structure, the three hidden units will be forced to re-represent the eight input values in some way that captures their relevant features, so that this hidden layer representation can be used by the output units to compute the correct target values.

Consider training the network shown in Figure 4.7 to learn the simple target function f ***(2)*** = 2, where ***2*** is a vector containing seven **0's** and a single 1. The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network in this case is constrained to use only three hidden units. Therefore, the essential information from all eight input units must be captured by the three learned hidden units.



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

FIGURE 4.7

When BACKPROPAGATION lied to this task, using each of the eight possible vectors as training examples, it successfully learns the target function. What hidden layer representation is created by the gradient descent BACKPROPAGATION algorithm? By examining the hidden unit values generated by the learned network for each of

the eight possible input vectors, it is easy to see that the learned encoding is similar to the familiar standard binary encoding of eight values using three bits (e.g., 000,001,010,. . . , 111). The exact values of the hidden units for one typical run of BACKPROPAGATION in Figure 4.7.

AN ILLUSTRATIVE EXAMPLE: FACE RECOGNITION

To illustrate some of the practical design choices involved in applying BACKPROPAGATION, this section discusses applying it to a learning task involving face recognition. All image data and code used to produce the examples described in this section are available at World Wide Web site http://www.cs.cmu.edu/~tom/mlbook.html, along with complete documentation on how to use the code. Why not try it yourself?

The learning task here involves classifying camera images of faces of various people in various poses. Images of 20 different people were collected, including approximately 32 images per person, varying the person's expression (happy, sad, angry, neutral), the direction in which they were looking (left, right, straight ahead, up), and whether or not they were wearing sunglasses. As can be seen from the example images in Figure 4.10, there is also variation in the background behind the person, the clothing worn by the person, and the position of the person's face within the image. In total, 624 greyscale images were collected, each with a resolution of 120×128, with each image pixel described by a greyscale intensity value between 0 (black) and 255 (white).

A variety of target functions can be learned from this image data. For example, given an image as input we could train an ANN to output the identity of the person, the direction in which the person is facing, the gender of the person, whether or not they are wearing sunglasses, etc. All of these target functions can be learned to high accuracy from this image data, and the reader is encouraged to try out these experiments. In the remainder of this section we consider one particular task: learning the direction in which the person is facing (to their left, right, straight ahead, or upward).

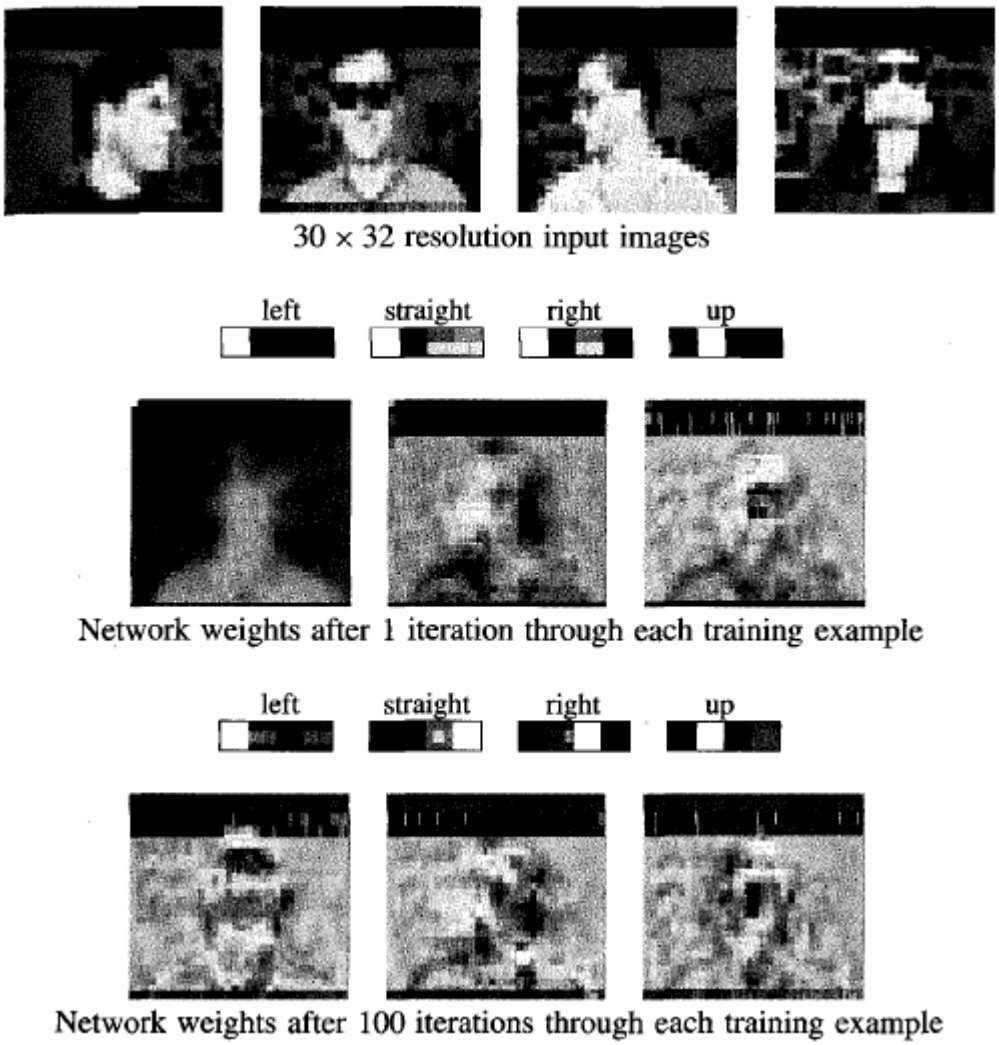


FIGURE 4.10
Learning an artificial neural network to recognize face pose. Here a $960 \times 3 \times 4$ network is trained on grey-level images of faces (see top), to predict whether a person is looking to their left, right, ahead, or up. After training on 260 such images, the network achieves an accuracy of 90% over a separate test set. The learned network weights are shown after one weight-tuning iteration through the training examples and after 100 iterations. Each output unit (left, straight, right, up) has four weights, shown by dark (negative) and light (positive) blocks. The leftmost block corresponds to the weight w_0, which determines the unit threshold, and the three blocks to the right correspond to weights on inputs from the three hidden units. The weights from the image pixels into each hidden unit are also shown, with each weight plotted in the position of the corresponding image pixel.

UNIT-2

EVALUATING HYPOTHESES

MOTIVATION:-

In many cases it is important to evaluate the performance of learned hypotheses as precisely as possible.

One reason is simply to understand whether to use the hypothesis. For instance, when learning from a limited-size database indicating the effectiveness of different medical treatments, it is important to understand as precisely as possible the accuracy of the learned hypotheses. A second reason is that evaluating hypotheses is an integral component of many learning methods. For example, in post-pruning decision trees to avoid overfitting, we must evaluate the impact of possible pruning steps on the accuracy of the resulting decision tree.

Therefore it is important to understand the likely errors inherent in estimating the accuracy of the pruned and unpruned tree. Estimating the accuracy of a hypothesis is relatively straightforward when data is plentiful. However, when we must learn a hypothesis and estimate its future accuracy given only a limited set of data, two key difficulties arise:

- ***Bias in the estimate:-*** First, the observed accuracy of the learned hypothesis over the training examples is often a poor estimator of its accuracy over future examples. Because the learned hypothesis was derived from these examples, they will typically provide an optimistically biased estimate of hypothesis accuracy over future examples. This is especially likely when the learner considers a very rich hypothesis space, enabling it to overfit the training examples.
- ***Variance in the estimate.*** Second, even if the hypothesis accuracy is measured over an unbiased set of test examples independent of the training examples, the measured accuracy can still vary from the true accuracy, depending on the makeup of the particular set of test examples. The smaller the set of test examples, the greater the expected variance.

ESTIMATING HYPOTHESIS ACCURACY:-

When evaluating a learned hypothesis we are most often interested in estimating the accuracy with which it will classify future instances. At the same time, we would like to know the probable error in this accuracy estimate (i.e., what error bars to associate with this estimate).

There is some space of possible instances X (e.g., the set of all people) over which various target functions may be defined (e.g., people who plan to purchase new skis this year). We assume that different instances in X may be encountered with different frequencies. A convenient way to model this is to assume there is some unknown probability distribution ***D*** that defines the probability of encountering each instance in X (e-g., *23* might assign a higher probability to encountering 19-year-old people than 109-year-old people). Notice *23* says nothing about whether x is a positive or negative example; it only detennines the probability that ***x*** will be encountered. The learning task is to learn the target concept or target function f by considering a space H of possible hypotheses. Training examples of the target function f are provided to the learner by a trainer who draws each instance independently, according to the distribution D, and who then forwards the instance x along with its correct target value f ***(x)*** to the learner.

To illustrate, consider learning the target function "people who plan to purchase new skis this year," given a sample of training data collected by surveying people as they arrive at a ski resort. In this case the instance space X is the space of all people, who might be described by attributes such as their age, occupation, how many times they skied last year, etc. The distribution D specifies for each person ***x*** the probability that x will be encountered as the next person arriving at the ski resort. The target function

f : X + **{O,1)** classifies each person according to whether or not they plan to purchase skis this year.

Sample Error and True Error:-

To answer these questions, we need to distinguish carefully between two notions of accuracy or, equivalently, error. One is the error rate of the hypothesis over the sample of data that is available. The other is the error rate of the hypothesis over the entire unknown distribution ***D*** of examples. We will call these the ***sample error*** and the ***true error*** respectively.

The ***sample error*** of a hypothesis with respect to some sample *S* of instances drawn from X is the fraction of *S* that it misclassifies:

Definition: The **sample error** (denoted ***errors(h))*** of hypothesis ***h*** with respect to target function f and data sample **S** is

$$error_S(h) \equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$

where ***n*** is the number of examples in **S,** and the quantity S(f ***(x), h(x))*** is ***1*** if f ***(x)*** # ***h(x),*** and ***0*** otherwise.

The ***true error*** of a hypothesis is the probability that it will misclassify a single randomly drawn instance from the distribution ***D.***

Definition: The **true error** (denoted $error_D(h)$)of hypothesis ***h*** with respect to target function f and distribution ***D,*** is the probability that ***h*** will misclassify an instance drawn at random according to ***D.***

$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)]$$

Here the notation $\Pr_{x \in \mathcal{D}}$ denotes that the probability is taken over the instan distribution $\mathcal{D}$. What we usually wish to know is the true error $error_D(h)$ of the hypothesis, because this is the error we can expect when applying the hypothesis to future examples. All we can measure, however, is the sample error $error_S(h)$ of the hypothesis for the data sample ***S*** that we happen to have in hand. The main question considered in this section is "How good an estimate of $error_D(h)$ is provided by $error_S(h)$?"

Confidence Intervals for Discrete-Valued Hypotheses:-

Here we give an answer to the question "How good an estimate of $error_D(h)$ is provided by $error_S(h)$?' for the case in which ***h*** is a discrete-valued hypothesis. More specifically, suppose we wish to estimate the true error for some discrete-valued hypothesis ***h,*** based on its observed sample error over a sample S, where

- the sample ***S*** contains ***n*** examples drawn independent of one another, and independent of ***h,*** according to the probability distribution ***D***
- $n \geq 30$
- hypothesis ***h*** commits ***r*** errors over these ***n*** examples (i.e., ***errors(h) = r / n).***

Under these conditions, statistical theory allows us to make the following assertions:

1. Given no other information, the most probable value of $error_D(h)$ is $error_S(h)$.

2. With approximately ***95%*** probability, the true error ***$error_D(h)$*** lies in the interval

$$error_S(h) \pm 1.96\sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

To illustrate, suppose the data sample S contains $n = 40$ examples and that hypothesis h commits $r = 12$ errors over this data. In this case, the sample error $error_S(h) = 12/40 = .30$. Given no other information, the best estimate of the true error $error_{\mathcal{D}}(h)$ is the observed sample error .30. However, we do not expect this to be a perfect estimate of the true error. If we were to collect a second sample S' containing 40 new randomly drawn examples, we might expect the sample error $error_{S'}(h)$ to vary slightly from the sample error $error_S(h)$. We expect a difference due to the random differences in the makeup of S and S'. In fact, if we repeated this experiment over and over, each time drawing a new sample S_i containing 40 new examples, we would find that for approximately 95% of these experiments, the calculated interval would contain the true error. For this reason, we call this interval the 95% confidence interval estimate for $error_{\mathcal{D}}(h)$. In the current example, where $r = 12$ and $n = 40$, the 95% confidence interval is, according to the above expression, $0.30 \pm (1.96 \cdot .07) = 0.30 \pm .14$.

The above expression for the 95% confidence interval can be generalized to any desired confidence level. The constant 1.96 is used in case we desire a 95% confidence interval. A different constant, z_N, is used to calculate the $N\%$ confidence interval. The general expression for approximate $N\%$ confidence intervals for $error_{\mathcal{D}}(h)$ is

$$error_S(h) \pm z_N\sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

where the constant z_N is chosen depending on the desired confidence level, using the values of z_N given in Table 5.1.

Confidence level $N\%$:	50%	68%	80%	90%	95%	98%	99%
Constant z_N:	0.67	1.00	1.28	1.64	1.96	2.33	2.58

TABLE 5.1
Values of z_N for two-sided $N\%$ confidence intervals.

A more accurate rule of thumb is that the above approximation works well when

$$n\ error_S(h)(1 - error_S(h)) \geq 5$$

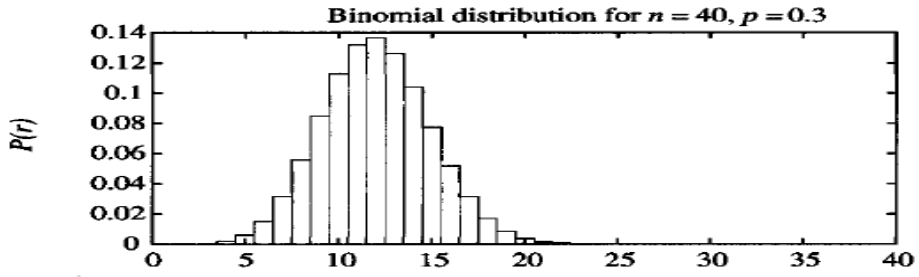
BASICS OF SAMPLING THEORY:-

Error Estimation and Estimating Binomial Proportions:-

Precisely how does the deviation between sample error and true error depend on the size of the data sample? This question is an instance of a well-studied problem in statistics: the problem of estimating the proportion of a population that exhibits some property, given the observed proportion over some random sample of the population. In our case, the property of interest is that h misclassifies the example.

The key to answering this question is to note that when we measure the sample error we are performing an experiment with a random outcome. We first collect a random sample **S** of ***n*** independently drawn instances from the distribution ***D*** and then measure the sample error ***error$_S$(h).*** As noted in the previous section, if we were to repeat this experiment many times, each time drawing a different random sample ***S$_i$*** of size n, we would expect to observe different values for the various error***$_{Si}$***(h), depending on random differences in the makeup of the various ***S$_i$.*** We say in such cases that error***$_{Si}$***(h), the outcome of the ith such experiment, is a random variable. In general, one can think of a random variable as the name of an experiment with a random outcome. The value of the random variable is the observed outcome of the random experiment.

As taking the k no-of random experiments, histogram table describes a particular probability distribution called the Binomial distribution



A Binomial distribution gives the probability of observing ***r*** heads in a sample of ***n*** independent coin tosses, when the probability of heads on a single coin toss is ***p.*** It is defined by the probability function

$$P(r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r}$$

If the random variable ***X*** follows a Binomial distribution, then:

- The probability ***Pr(X = r)*** that ***X*** will take on the value ***r*** is given by ***P(r)***

- The expected, or mean value of X, ***E[X],*** is

$$E[X] = np$$

- The variance of ***X, Var(X),*** is

$$Var(X) = np(1-p)$$

- The standard deviation of X, ***ax,*** is

$$\sigma_X = \sqrt{np(1-p)}$$

For sufficiently large values of ***n*** the Binomial distribution is closely approximated by a Normal distribution with the same mean and variance. Most statisticians recommend using the Normal approximation only when ***np(1- p)*** >= ***5.***

The Binomial Distribution:-

A good way to understand the Binomial distribution is to consider the following problem. You are given a worn and bent coin and asked to estimate the probability that the coin will turn up heads when tossed. Let us call this unknown probability of heads *p*. You toss the coin n times and record the number of times *r* that it turns up heads. A reasonable estimate of *p* is *r* / *n*. Note that if the experiment were rerun, generating a new set of n coin tosses, we might expect the number of heads *r* to vary somewhat from the value measured in the first experiment, yielding a somewhat different estimate for *p*. The Binomial distribution describes for each possible value of r (i.e., from 0 to n), the probability of observing exactly r heads given a sample of *n* independent tosses of a coin whose true probability of heads is *p*.

The detailed form of the Binomial distribution depends on the specific sample size n and the specific probability *p* or ***error$_D$(h)***.

The general setting to which the Binomial distribution applies is:

1. There is a base, or underlying, experiment (e.g., toss of the coin) whose outcome can be described by a random variable, say *Y*. The random variable *Y* can take on two possible values (e.g., *Y* = 1 if heads, Y = 0 if tails).

2. The probability that *Y* = 1 on any single trial of the underlying experiment is given by some constant p, independent of the outcome of any other experiment. The probability that *Y* = 0 is therefore (1 - p). Typically, p is not known in advance, and the problem is to estimate it.

3. A series of n independent trials of the underlying experiment is performed (e.g., n independent coin tosses), producing the sequence of independent, identically distributed random variables $Y_1, Y_2, \ldots, Y_n$. Let *R* denote the number of trials for which $Y_i = 1$ in this series of n experiments

$$R = \sum_{i=1}^{n} Y_i$$

4. The probability that the random variable ***R*** will take on a specific value *r* (e.g., the probability of observing exactly *r* heads) is given by the Binomial distribution

$$\Pr(R = r) = \frac{n!}{r!(n-r)!}\, p^r (1-p)^{n-r}$$

The Binomial distribution characterizes the probability of observing *r* heads from n coin flip experiments, as well as the probability of observing *r* errors in a data sample containing n randomly drawn instances.

Mean and Variance:-

Two properties of a random variable that are often of interest are its expected value (also called its mean value) and its variance. The expected value is the average of the values taken on by repeatedly sampling the random variable. More precisely

Definition: Consider a random variable *Y* that takes on the possible values $y_1, \ldots y_n$. The **expected value** of *Y*, *E[Y]*, is

$$E[Y] \equiv \sum_{i=1}^{n} y_i \Pr(Y = y_i)$$

In case the random variable *Y* is governed by a Binomial distribution, then it can be shown that

$$E[Y] = np$$

where n and p are the parameters of the Binomial distribution defined in above Equation

A second property, the variance, captures the "width or "spread" of the probability distribution; that is, it captures how far the random variable is expected to vary from its mean value.

Definition: The **variance** of a random variable *Y*, *Var[Y]*, is

$$Var[Y] \equiv E[(Y - E[Y])^2]$$

The variance describes the expected squared error in using a single observation of Y to estimate its mean *E[Y]*. The square root of the variance is called the *standard deviation* of *Y* , denoted σ_Y.

Definition: The **standard deviation** of a random variable *Y*, σ_Y, is

$$\sigma_Y \equiv \sqrt{E[(Y - E[Y])^2]}$$

In case the random variable Y is governed by a Binomial distribution, then the variance and standard deviation are given by

$$Var[Y] = np(1 - p)$$
$$\sigma_Y = \sqrt{np(1 - p)}$$

Estimators, Bias, and Variance :-

Now that we have shown that the random variable $error_S(h)$ obeys a Binomial distribution, we return to our primary question: What is the likely difference between $error_S(h)$ and the true error $error_D(h)$**?** Let us describe $error_S(h)$ and $error_D(h)$ using the terms in Equation defining the Binomial distribution. We then have

$$error_S(h) = \frac{r}{n}$$
$$error_D(h) = p$$

where ***n*** is the number of instances in the sample S, ***r*** is the number of instances from S misclassified by ***h,*** and p is the probability of misclassifying a single instance drawn from *D***.**

Statisticians call $error_S(h)$ an ***estimator*** for the true error $error_D(h)$. In general, an estimator is any random variable used to estimate some parameter of the underlying population from which the sample is drawn. An obvious question to ask about any estimator is whether on average it gives the right estimate. We define the ***estimation bias*** to be the difference between the expected value of the estimator and the true value of the parameter.

Definition: The **estimation bias** of **an** estimator Y for **an** arbitrary parameter p is

$$E[Y] - p$$

If the estimation bias is zero, we say that Y is an ***unbiased estimator*** for ***p.*** Notice this will be the case if the average of many random values of Y generated by repeated random experiments (i.e., E[Y]) converges toward p. Is $error_S(h)$ an unbiased estimator for $error_D(h)$**?** Yes, because for a Binomial distribution the expected value of ***r*** is equal to ***np*** . It follows, given that ***n*** is a constant, that the expected value of ***r / n*** is p. Two quick remarks are in order regarding the estimation bias. First, that testing the hypothesis on the training examples provides an optimistically biased estimate of hypothesis error,it is exactly this notion of estimation bias to which we were referring. In order for $error_S(h)$ to give an unbiased estimate of $error_D(h)$**,** the hypothesis ***h*** and sample ***S*** must be chosen independently. Second, this notion of ***estimation bias*** should not be confused with the ***inductive bias*** of a learner. The estimation bias is a numerical quantity, whereas the inductive bias is a set of assertions.

A second important property of any estimator is its variance. Given a choice among alternative unbiased estimators, it makes sense to choose the one with least variance. By our definition of variance, this choice will yield the smallest expected squared error between the estimate and the true value of the parameter.

In general, given *r* errors in a sample of *n* independently drawn test examples, the standard deviation for $error_S(h)$ is given by

$$\sigma_{error_S(h)} = \frac{\sigma_r}{n} = \sqrt{\frac{p(1-p)}{n}}$$

which can be approximated by substituting $r / n = error_S(h)$ for *p*

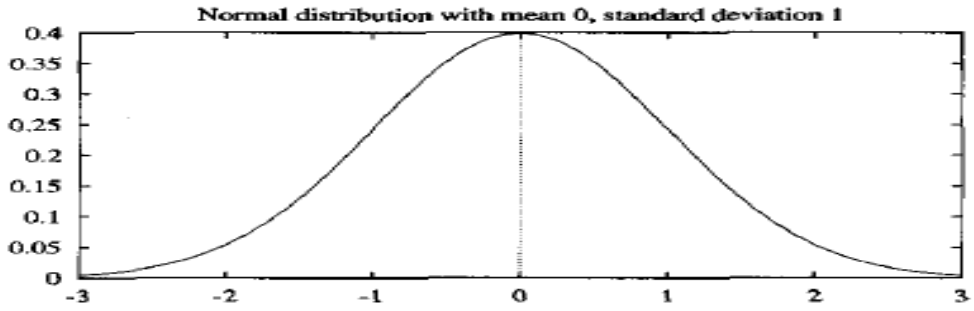
$$\sigma_{error_S(h)} \approx \sqrt{\frac{error_S(h)(1-error_S(h))}{n}}$$

Confidence Intervals:-

One common way to describe the uncertainty associated with an estimate is to give an interval within which the true value is expected to fall, along with the probability with which it is expected to fall into this interval. Such estimates are called *confidence interval* estimates.

Definition: An ***N%*** **confidence interval** for some parameter p is an interval that is expected with probability ***N%*** to contain ***p.***

For a given value of ***N*** how can we find the size of the interval that contains ***N%*** of the probability mass? Unfortunately, for the Binomial distribution this calculation can be quite tedious. Fortunately, however, an easily calculated and very good approximation can be found in most cases, based on the fact that for sufficiently large sample sizes the Binomial distribution can be closely approximated by the Normal distribution. The Normal distribution, summarized in is perhaps the most well-studied probability distribution in statistics. It is a bell-shaped distribution fully specified by its mean μ and standard deviation σ. For large n, any Binomial distribution is very closely approximated by a Normal distribution with the same mean and variance.



A Normal distribution (also called a Gaussian distribution) is a bell-shaped distribution defined by the probability density function

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

A Normal distribution is fully determined by two parameters in the above formula: μ and σ.

If the random variable ***X*** follows a normal distribution, then:

- The probability that X will fall into the interval (a, b***)*** is given by
$$\int_a^b p(x)dx$$
- The expected, or mean value of ***X, E[X],*** is
$$E[X] = \mu$$
- The variance of X, ***Var(X),*** is
$$Var(X) = \sigma^2$$
- The standard deviation of ***X,*** σ_X , is
$$\sigma_X = \sigma$$

The Central Limit Theorem (Section 5.4.1) states that the sum of a large number of independent,

identically distributed random variables follows a distribution that is approximately Normal.

A GENERAL APPROACH FOR DERIVING CONFIDENCE INTERVALS:-

It describe in detail how to derive confidence interval estimates for one particular case: estimating $error_V(h)$ for a discrete-valued hypothesis ***h,*** based on a sample of ***n*** independently drawn instances. The approach described there illustrates a general approach followed in many estimation problems. In particular, we can see this as a problem of estimating the mean (expected value) of a population based on the mean of a randomly drawn sample of size ***n.***

The general process includes the following steps:

1. Identify the underlying population parameter p to be estimated, for example, $error_D(h)$.

2. Define the estimator Y (e.g., $error_S(h)$). It is desirable to choose a minimum variance, unbiased estimator.

3. Determine the probability distribution D_Y that governs the estimator Y, including its mean and variance.

4. Determine the ***N%*** confidence interval by finding thresholds L and ***U*** such that ***N%*** of the mass in the probability distribution D_Y falls between L and U.

Central Limit Theorem:-

One essential fact that simplifies attempts to derive confidence intervals is the Central Limit Theorem. Consider again our general setting, in which we observe the values of ***n*** independently drawn random variables $Y_1 \ldots Y_n$ that obey the same unknown underlying probability distribution (e.g., ***n*** tosses of the same coin). Let *p* denote the mean of the unknown distribution governing each of the Y_i and let a denote the standard deviation. We say that these variables Y_i are ***independent, identically distributed*** random variables, because they describe independent experiments, each obeying the same underlying probability distribution. In an attempt to estimate the mean *p* of the distribution governing the Yi, we calculate the sample mean $\bar{Y}_n \equiv \frac{1}{n}\sum_{i=1}^{n} Y_i$ .The Central Limit Theorem states that the probability distribution governing $\bar{Y}_n$ approaches a Normal distribution as $n \to \infty$, ***regardless of the distribution that governs the underlying random variables*** Y_i. Furthermore, the mean of the distribution governing $\bar{Y}_n$ approaches μ and the standard deviation approaches $\frac{\sigma}{\sqrt{n}}$.

More precisely, Then **as** $n \to \infty$, the distribution governing

$$\frac{\bar{Y}_n - \mu}{\frac{\sigma}{\sqrt{n}}}$$

approaches a Normal distribution, with zero mean and standard deviation equal to 1.

DIFFERENCE IN ERROR OF TWO HYPOTHESES:-

Consider the case where we have two hypotheses h_1 and h_2 for some discrete valued target function. Hypothesis h_1 has been tested on a sample S_1 containing n_1 randomly drawn examples, and h_2 has been tested on an independent sample S_2 containing n_2 examples drawn from the same distribution. Suppose we wish to estimate the difference d between the true errors of these two hypotheses.

$$d \equiv error_{\mathcal{D}}(h_1) - error_{\mathcal{D}}(h_2)$$

We will use the generic four-step procedure to derive a confidence interval estimate for d. Having identified d as the parameter to be estimated, we next define an estimator. The obvious choice for an estimator in this case is the difference between the sample errors, which we denote by $\hat{d}$

$$\hat{d} \equiv error_{S_1}(h_1) - error_{S_2}(h_2)$$

Although we will not prove it here, it can be shown that $\hat{d}$ gives an unbiased estimate of d;

that is $E[\hat{d}] = d$

to obtain the approximate variance of each of these distributions, we have

$$\sigma_{\hat{d}}^2 \approx \frac{error_{S_1}(h_1)(1-error_{S_1}(h_1))}{n_1} + \frac{error_{S_2}(h_2)(1-error_{S_2}(h_2))}{n_2}$$

Using the approximate variance a; given above, this approximate ***N%*** confidence interval estimate for d is

$$\hat{d} \pm z_N \sqrt{\frac{error_{S_1}(h_1)(1-error_{S_1}(h_1))}{n_1} + \frac{error_{S_2}(h_2)(1-error_{S_2}(h_2))}{n_2}}$$

we redefine $\hat{d}$ as

$$\hat{d} \equiv error_S(h_1) - error_S(h_2)$$

COMPARING LEARNING ALGORITHMS:-

Often we are interested in comparing the performance of two learning algorithms L_A and L_B, rather than two specific hypotheses. What is an appropriate test for comparing learning algorithms, and how can we determine whether an observed difference between the algorithms is statistically significant? Although there is active debate within the machine-learning research community regarding the best method for comparison, we present here one reasonable approach. A discussion of alternative methods is given by Dietterich (1996).

As usual, we begin by specifying the parameter we wish to estimate. Suppose we wish to determine which of L_A and L_B is the better learning method on average for learning some particular target function f. A reasonable way to define "on average" is to consider the relative performance of these two algorithms averaged over all the training sets of size n that might be drawn from the underlying instance distribution ***D.*** In other words, we wish to estimate the expected value of the difference in their errors

$$\underset{S \subset \mathcal{D}}{E}[error_{\mathcal{D}}(L_A(S)) - error_{\mathcal{D}}(L_B(S))]$$

where L(S) denotes the hypothesis output by learning method ***L*** when given the sample S of training data and where the subscript S C D indicates that the expected value is taken over samples S drawn according to the underlying instance distribution D. The above expression describes the expected value of the difference in errors between learning methods L_A and L_B.

Of course in practice we have only a limited sample D_o of data when comparing learning methods. In such cases, one obvious approach to estimating the above quantity is to divide D_o into a training set S_o and a disjoint test set T_o.

The training data can be used to train both L_A and L_B, and the test data can be used to compare the accuracy of the two learned hypotheses. In other words, we measure the quantity

$$error_{T_0}(L_A(S_0)) - error_{T_0}(L_B(S_0))$$

Notice two key differences between this estimator and the quantity in Equation. First, we are using $error_{T0}(h)$ to approximate $error_D(h)$. Second, we are only measuring the difference in errors for one training set S_0 rather than taking the expected value of this difference over all samples S that might be drawn from the distribution D.

One way to improve on the estimator given by Equation is to repeatedly partition the data D_0 into disjoint training and test sets and to take the mean of the test set errors for these different experiments. This procedure first partitions the data into k disjoint subsets of equal size, where this size is at least 30. It then trains and tests the learning

algorithms k times, using each of the k subsets in turn as the test set, and using all remaining data as the training set. In this way, the learning algorithms are tested on k independent test sets, and the mean difference in errors $\bar{\delta}$ is returned as an estimate of the difference between the two learning algorithms.

The quantity $\bar{\delta}$ returned by the procedure that can be taken as an estimate of the desired quantity from Equation 5.14. More appropriately, we can view $\bar{\delta}$ as an estimate of the quantity

$$\underset{S \subset D_0}{E} [error_{\mathcal{D}}(L_A(S)) - error_{\mathcal{D}}(L_B(S))]$$

BAYESIAN LEARNING

INTRODUCTION

Bayesian learning methods are relevant to our study of machine learning for two different reasons. First, Bayesian learning algorithms that calculate explicit probabilities for hypotheses, such as the naive Bayes classifier, are among the most practical approaches to certain types of learning problems. For example, Michie et **al.** (1994) provide a detailed study comparing the naive Bayes classifier to other learning algorithms, including decision tree and neural network algorithms.

These researchers show that the naive Bayes classifier is competitive with these other learning algorithms in many cases and that in some cases it outperforms these other methods. In this chapter we describe the naive Bayes classifier and provide a detailed example of its use. In particular, we discuss its application to the problem of learning to classify text documents such as electronic news articles.

For such learning tasks, the naive Bayes classifier is among the most effective algorithms known.

The second reason that Bayesian methods are important to our study of machine learning is that they provide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities. We also use a Bayesian analysis to justify a key design choice in neural network learning algorithms:choosing to minimize the sum of squared errors when searching the space of possible neural networks. We also derive an alternative error function, cross entropy, that is more appropriate than sum of squared errors when learning target functions that predict probabilities. We use a Bayesian perspective to analyze the inductive bias of decision tree learning algorithms that favor short decision trees and examine the closely related Minimum Description Length principle. A basic familiarity with Bayesian methods is important to understanding and characterizing the operation of many algorithms in machine learning.

Features of Bayesian learning methods include:

- Each observed training example can incrementally decrease or increase the estimated probability that a hypothesis is correct. This provides a more flexible approach to learning than algorithms that completely eliminate a hypothesis if it is found to be inconsistent with any single example.
- Prior knowledge can be combined with observed data to determine the final probability ~f a hypothesis. In Bayesian learning, prior knowledge is provided by asserting (1) a prior probability for each candidate hypothesis, and (2) a probability distribution over observed data for each possible hypothesis.
- Bayesian methods can accommodate hypotheses that make probabilistic predictions.
- New instances can be classified by combining the predictions of multiple hypotheses, weighted by their probabilities.
- Even in cases where Bayesian methods prove computationally intractable, they can provide a standard of optimal decision making against which other practical methods can be measured.

BAYES THEOREM:-

In machine learning we are often interested in determining the best hypothesis from some space H, given the observed training data D. One way to specify what we mean by the ***best*** hypothesis is to say that we demand the ***most probable***

hypothesis, given the data ***D*** plus any initial knowledge about the prior probabilities of the various hypotheses in H. Bayes theorem provides a direct method for calculating such probabilities. More precisely, Bayes theorem provides a way to calculate the probability of a hypothesis based on its prior probability, the probabilities of observing various data given the hypothesis, and the observed data itself.

To define Bayes theorem precisely, let us first introduce a little notation. We shall write ***P(h)*** to denote the initial probability that hypothesis ***h*** holds, before we have observed the training data. ***P(h)*** is often called the ***prior probability*** of ***h*** and may reflect any background knowledge we have about the chance that ***h*** is a correct hypothesis. If we have no such prior knowledge, then we might simply assign the same prior probability to each candidate hypothesis. Similarly, we will write ***P(D)*** to denote the prior probability that training data D will be observed. Next, we will write ***P(D/h)*** to denote the probability of observing data ***D*** given some world in which hypothesis ***h*** holds. More generally, we write ***P(x/y)*** to denote the probability of x given ***y.*** In machine learning problems we are interested in the probability ***P (h/D)*** that ***h*** holds given the observed training data ***D. P (h/D)*** is called the ***posterior probability*** of ***h,*** because it reflects our confidence that ***h*** holds after we have seen the training data ***D.*** Notice the posterior probability ***P(h/D)*** reflects the influence of the training data ***D,*** in contrast to the prior probability ***P(h)***, which is independent of D.

Bayes theorem is the cornerstone of Bayesian learning methods because it provides a way to calculate the posterior probability ***P(h/D),*** from the prior probability ***P(h),*** together with ***P(D)*** and ***P(D/h).***

Bayes theorem:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

As one might intuitively expect, ***P(h/D)*** increases with ***P(h)*** and with ***P(D/h)*** according to Bayes theorem. It is also reasonable to see that ***P(h/D)*** decreases as ***P(D)*** increases, because the more probable it is that ***D*** will be observed independent of ***h,*** the less evidence ***D*** provides in support of ***h.***

In many learning scenarios, the learner considers some set of candidate hypotheses H and is interested in finding the most probable hypothesis $h \in H$ given the observed data ***D.*** Any such maximally probable hypothesis is called a ***Maximum a Posteriori (MAP) Hypothesis***. We can determine the MAP hypotheses by using Bayes theorem to calculate the posterior probability of each candidate hypothesis. More precisely, we will say that h_{MAP} is a MAP hypothesis provided.

$$\begin{aligned} h_{MAP} &\equiv \underset{h \in H}{\operatorname{argmax}} P(h|D) \\ &= \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)} \\ &= \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h) \end{aligned}$$

In some cases, we will assume that every hypothesis in H is equally probable a priori ***($P(h_i) = P(h_j)$*** for all h_i and h_j in H). In this case we can further simplify Equation and need only consider the term ***P(D / h)*** to find the most probable hypothesis. ***P(D/h)*** is often called the ***likelihood*** of the data ***D*** given ***h,*** and any hypothesis that maximizes ***P(D/h)*** is called a ***maximum likelihood*** (ML) hypothesis, h_{ML}

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(D|h)$$

BAYES THEOREM AND CONCEPT LEARNING:-

Bayes theorem provides a principled way to calculate the posterior probability of each hypothesis given the training data, we can use it as the basis for a straightforward learning algorithm that calculates the probability for each possible hypothesis, then outputs the most probable. This section considers such a brute-force Bayesian concept learning algorithm, then compares it to concept learning algorithms.

Product rule: probability of a conjunction of two events A and B

$$P(A \wedge B) = P(A|B)P(B) = P(B|A)P(A)$$

Sum rule: probability of a disjunction of two events ***A*** and B

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

Bayes theorem: the posterior probability ***P(h/D)*** of ***h*** given ***D***

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

Theorem of total probability: if events $A_1, \ldots, A_n$ are mutually exclusive with $\sum_{i=1}^{n} P(A_i) = 1$,

$$P(B) = \sum_{i=1}^{n} P(B|A_i)P(A_i)$$

.

BRUTE-FORCE MAP LEARNING ALGORITHM

1. For each hypothesis ***h*** in H, calculate the posterior probability

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

2. Output the hypothesis h_{MAP} with the highest posterior probability

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}} P(h|D)$$

This algorithm may require significant computation, because it applies Bayes theorem to each hypothesis in H to calculate ***P(h/D).*** While this may prove impractical for large hypothesis spaces, the algorithm is still of interest because it provides a standard against which we may judge the performance of other concept learning algorithms.

In order specify a Iearning problem for the **BRUTE-FORCE MAP LEARNING** algorithm we must specify what values are to be used for ***P(h)*** and for ***P(D/h)***. We may choose the probability distributions ***P(h)*** and ***P(D/h)*** in any way we wish, to describe our prior knowledge about the learning task. Here let us choose them to be consistent with the following assumptions:

1. The training data ***D*** is noise free (i.e., $d_i = c(x_i)$).

2. The target concept ***c*** is contained in the hypothesis space H

3. We have no a priori reason to believe that any hypothesis is more probable than any other.

Together these constraints imply that we should choose

$$P(h) = \frac{1}{|H|} \quad \text{for all } h \text{ in } H$$

noise-free training data, the probability of observing classification d_i given h is just 1 if $d_i = h(x_i)$ and 0 if $d_i \neq h(x_i)$. Therefore,

$$P(D|h) = \begin{cases} 1 \text{ if } d_i = h(x_i) \text{ for all } d_i \text{ in } D \\ 0 \text{ otherwise} \end{cases}$$

In other words, the probability of data ***D*** given hypothesis ***h*** is 1 if ***D*** is consistent with ***h,*** and 0 otherwise.

Given these choices for ***P(h)*** and for ***P(D/h)*** we now have a fully-defined problem for the above **BRUTE-FORCE MAP LEARNING a**lgorithm. **Let** us consider the first step of this algorithm, which uses Bayes theorem to compute the posterior probability ***P(h/D)*** of each hypothesis ***h*** given the observed training data ***D.***

Recalling Bayes theorem, we have

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

First consider the case where *h* is inconsistent with the training data *D.* Since Equation defines *P(D/h)* to be *0* when *h* is inconsistent with *D,* we have

$$P(h|D) = \frac{0 \cdot P(h)}{P(D)} = 0 \text{ if } h \text{ is inconsistent with } D$$

The posterior probability of a hypothesis inconsistent with *D* is zero. Now consider the case where *h* is consistent with *D.* Since Equation defines *P(D/h)* to be 1 when *h* is consistent with *D,* we have *P(D/h)* to be 1 if *h* is consistent with *D*

$$\begin{aligned} P(h|D) &= \frac{1 \cdot \frac{1}{|H|}}{P(D)} \\ &= \frac{1 \cdot \frac{1}{|H|}}{\frac{|VS_{H,D}|}{|H|}} \\ &= \frac{1}{|VS_{H,D}|} \text{ if } h \text{ is consistent with } D \end{aligned}$$

where $VS_{H,D}$ is the subset of hypotheses from H that are consistent with *D.* It is easy to verify that $P(D) = \frac{|VS_{H,D}|}{|H|}$ above, because the sum over all hypotheses of *P(h /D)* must be one and because the number of hypotheses from H consistent with *D* is by definition $|VS_{H,D}|$. Alternatively, we can derive *P(D)* from the theorem of total probability and the fact that the hypotheses are mutually exclusive

$$
\begin{aligned}
P(D) &= \sum_{h_i \in H} P(D|h_i) P(h_i) \\
&= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|} + \sum_{h_i \notin VS_{H,D}} 0 \cdot \frac{1}{|H|} \\
&= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|} \\
&= \frac{|VS_{H,D}|}{|H|}
\end{aligned}
$$

To summarize, Bayes theorem implies that the posterior probability *P(h /D)* under our assumed *P(h)* and *P(D/h)* is

$$
P(h|D) = \begin{cases} \frac{1}{|VS_{H,D}|} & \text{if } h \text{ is consistent with } D \\ 0 & \text{otherwise} \end{cases}
$$

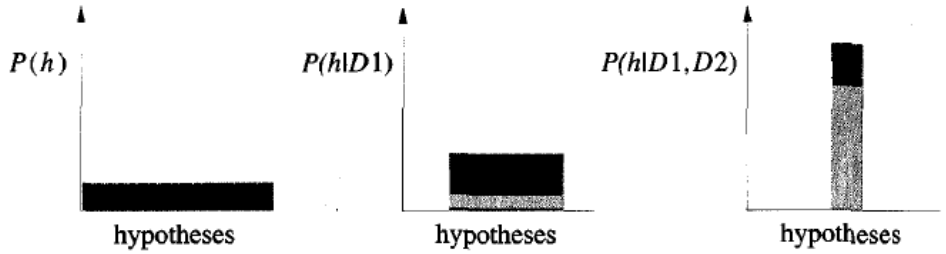
the posterior probability for inconsistent hypotheses becomes zero while the total probability summing to one is shared equally among the remaining consistent hypotheses. The above analysis implies that under our choice for *P(h)* and *P(D/h),* every *consistent* hypothesis has posterior probability $\frac{1}{|VS_{H,D}|}$, and every inconsistent hypothesis has posterior probability **0.** Every consistent hypothesis is, therefore, a MAP hypothesis.

MAP Hypotheses and Consistent Learners:-

The above analysis shows that in the given setting, every hypothesis consistent with *D* is a MAP hypothesis. This statement translates directly into an interesting statement about a general class of learners that we might call *consistent learners.*

We will say that a learning algorithm is a *consistent learner* provided it outputs a hypothesis that commits zero errors over the training examples. Given the above analysis, we can conclude that *every consistent learner outputs a MAP hypothesis, i f* **we** *assume a uniform prior probability distribution over H (i.e., P(hi) = P(hj) for all* **i,** *j), and if we assume deterministic, noise free training data (i.e., P(D / h) =1 if D and h are consistent, and 0 otherwise).*

FIND-S searches the hypothesis space H from specific to general hypotheses, outputting a maximally specific consistent hypothesis (i.e., a maximally specific member of the version space). Because FIND-S outputs a consistent hypothesis, we know that it will output a MAP hypothesis under the probability distributions *P(h)* and *P(D/h)* defined above. Of course FIND-S does not explicitly manipulate probabilities at all-it simply outputs a maximally specific member



of the version space. However, by identifying distributions for ***P(h)*** and ***P(D(h)*** under which its output hypotheses will be MAP hypotheses, we have a useful way of characterizing the behavior of FIND-S. Are there other probability distributions for ***P(h)*** and ***P(D1h)*** under which FIND-S outputs MAP hypotheses? Yes. Because FIND-S outputs a ***maximally specz$***

c hypothesis from the version space, its output hypothesis will be a MAP hypothesis relative to any prior probability distribution that favors more specific hypotheses. More precisely, suppose 3-1 is any probability distribution ***P(h)*** over

H that assigns ***P(h1) 2 P(hz)*** if ***hl*** is more specific than ***h2.*** Then it can be shown that FIND-S outputs a MAP hypothesis assuming the prior distribution $3-1$ and the same distribution ***P(D1h)*** discussed above.

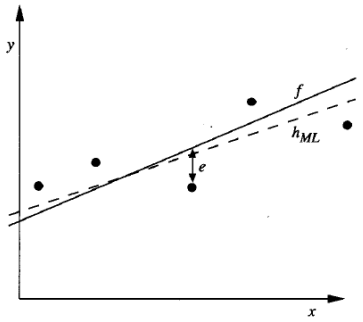
The Bayesian framework allows one way to characterize the behavior of learning algorithms, even when the learning algorithm does not explicitly manipulate probabilities. By identifying probability distributions ***P(h)*** and ***P(D/h)*** under which the algorithm outputs optimal (i.e., MAP) hypotheses, we can characterize the implicit assumptions, under which this algorithm behaves optimally.

MAXIMUM LIKELIHOOD AND LEAST-SQUARED ERROR HYPOTHESES:-

Bayesian analysis can sometimes be used to show that a particular learning algorithm outputs MAP hypotheses even though it may not explicitly use Bayes rule or calculate probabilities in any form.

We consider the problem of learning a continuous-valued target function-a problem faced by many learning approaches such as neural network learning, linear regression, and polynomial curve fitting. A straightforward Bayesian analysis will show that ***under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will output a maximum likelihood hypothesis.*** The significance of this result is that it provides a Bayesian justification (under certain assumptions) for many neural network and other curve fitting methods that attempt to minimize the sum of squared errors over the training data.

A simple example of such a problem is learning a linear function, though our analysis applies to learning arbitrary real-valued functions illustrates a linear target function f depicted by the solid line, and a set of noisy training examples of this target function. The dashed line corresponds to the hypothesis h_{ML} with least-squared training error, hence the maximum likelihood hypothesis. Notice that the maximum likelihood hypothesis is not necessarily identical to the correct hypothesis, f, because it is inferred from only a limited sample of noisy training data.



Before showing why a hypothesis that minimizes the sum of squared errors in this setting is also a maximum likelihood hypothesis, let us quickly review two basic concepts from probability theory: probability densities and Normal distributions. First, in order to discuss probabilities over continuous variables such as e, we must introduce probability densities. The reason, roughly, is that we wish for the total probability over all possible values of the random variable to sum to one. In the case of continuous variables we cannot achieve this by assigning a finite probability to each of the infinite set of possible values for the random variable. Instead, we speak of a probability density for continuous variables such as e and require that the integral of this probability density over all possible values be one. In general we will use lower case p to refer to the probability density function, to distinguish it from a finite probability ***P*** (which we will sometimes refer to as a probability mass). The probability density ***p(x0)*** is the limit as ε goes to zero, of times the probability that ***x*** will take on a value in the interval ***[xo, xo + 6).***

Probability density function:

$$p(x_0) \equiv \lim_{\epsilon \to 0} \frac{1}{\epsilon} P(x_0 \leq x < x_0 + \epsilon)$$

Second, we stated that the random noise variable e is generated by a Normal probability distribution. A Normal distribution is a smooth, bell-shaped distribution that can be completely characterized by its mean ***p*** and its standard deviation for a precise definition.

Given this background we now return to the main issue: showing that the least-squared error hypothesis is, in fact, the maximum likelihood hypothesis within our problem setting. We will show this by deriving the maximum likelihood hypothesis starting with our earlier definition Equation,but using lower case p to refer to the probability density

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} p(D|h)$$

we can write ***P(D/h)*** as the product of the various P***(d/hi)***

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} p(d_i|h)$$

we are writing the expression for the probability of d_i given that ***h*** is the correct description of the target function f, we will also substitute *p* = f ***(xi)*** = ***h(xi),*** yielding

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i - \mu)^2}$$

$$= \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i - h(x_i))^2}$$

We now apply a transformation that is common in maximum likelihood calculations:

Rather than maximizing the above complicated expression we shall choose to maximize its (less complicated) logarithm. This is justified because ln is a monotonic function of ***p.*** Therefore maximizing **ln** also maximizes ***p.***

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} \ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

The first term in this expression is a constant independent of ***h,*** and can therefore be discarded, yielding

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} -\frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

Maximizing this negative quantity is equivalent to minimizing the corresponding positive quantity.

$$h_{ML} = \underset{h \in H}{\operatorname{argmin}} \sum_{i=1}^{m} \frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

Finally, we can again discard constants that are independent of ***h.***

$$h_{ML} = \underset{h \in H}{\operatorname{argmin}} \sum_{i=1}^{m} (d_i - h(x_i))^2$$

MAXIMUM LIKELIHOOD HYPOTHESES FOR PREDICTING PROBABILITIES:-

we determined that the maximum likelihood hypothesis is the one that minimizes the sum of squared errors over the training examples. In this section we derive an analogous criterion for a second setting that is common in neural network learning: learning to predict probabilities.

Consider the setting in which we wish to learn a nondeterministic (probabilistic) function ***f*** : X -+ {0, 11, which has two discrete output values. For example, the instance space X might represent medical patients in terms of their symptoms, and the target function ***f*** (x) might be 1 if the patient survives the disease and 0 if not. Alternatively, X might represent loan applicants in terms of their past credit history, and ***f*** (x) might be 1 if the applicant successfully repays their next loan and 0 if not. In both of these cases we might well expect ***f*** to be probabilistic. For example, among a collection of patients exhibiting the same set of observable symptoms, we might find that 92% survive, and 8% do not. This unpredictability could arise from our inability to observe all the important distinguishing features of the patients, or from some genuinely probabilistic mechanism in the evolution of the disease. Whatever the source of the problem, the effect is that we have a target function f (x) whose output is a probabilistic function of the input.

Given this problem setting, we might wish to learn a neural network (or other real-valued function approximator) whose output is the ***probability*** that ***f*** (x) = 1. In other words, we seek to learn the target function, f' : X + [O, 11, such that

f '(x) = P(f (x) = 1). In the above medical patient example, if x is one of those indistinguishable patients of which 92% survive, then f'(x) = 0.92 whereas the probabilistic function ***f*** (x) will be equal to 1 in 92% of cases and equal to 0 in

the remaining 8%.

How can we learn ***f'*** using, say, a neural network? One obvious, bruteforce way would be to first collect the observed frequencies of 1's and 0's for each possible value of x and to then train the neural network to output the target frequency for each x. As we shall see below, we can instead train a neural network directly from the observed training examples of ***f,*** yet still derive a maximum likelihood hypothesis for f'.

What criterion should we optimize in order to find a maximum likelihood hypothesis for ***f'*** in this setting? To answer this question we must first obtain an expression for P(D/h). Let us assume the training data D is of the form

D = {(x_l, d_l) . . . (x_m, d_m)}, where d_i is the observed 0 or 1 value for f (x_i). Recall that in the maximum likelihood, least-squared error analysis of the previous section, we made the simplifying assumption that the instances (x_l . . . x_m)were fixed. This enabled us to characterize the data by considering only the target values d_i. Although we could make a similar simplifying assumption in this case, let us avoid it here in order to demonstrate that it has no impact on the final outcome. Thus treating both x_i and d_i as random variables, and assuming that each training example is drawn independently, we can write P(D/h) as

$$P(D|h) = \prod_{i=1}^{m} P(x_i, d_i|h)$$

It is reasonable to assume, furthermore, that the probability of encountering any particular instance x_i is independent of the hypothesis h. For example, the probability that our training set contains a particular ***patient*** x_i is independent of our hypothesis about survival rates (though of course the ***survival d,*** of the patient does depend strongly on ***h).*** When ***x*** is independent of ***h*** we can rewrite the above expression as

$$P(D|h) = \prod_{i=1}^{m} P(x_i, d_i|h) = \prod_{i=1}^{m} P(d_i|h, x_i)P(x_i)$$

Now what is the probability $P(d_i/h, x_i)$ of observing $d_i = 1$ for a single instance x_i, given a world in which hypothesis h holds? Recall that h is our hypothesis regarding the target function, which computes this very probability.

Therefore, $P(di = 1 / h, x_i) = h(x_i)$, and in general

$$P(d_i|h, x_i) = \begin{cases} h(x_i) & \text{if } d_i = 1 \\ (1 - h(x_i)) & \text{if } d_i = 0 \end{cases}$$

In order to substitute this into the Equation ***(6.8)*** for ***P(D/h),*** let us first " re-express it in a more mathematically manipulable form, as

$$P(d_i|h, x_i) = h(x_i)^{d_i}(1 - h(x_i))^{1-d_i}$$

MINIMUM DESCRIPTION LENGTH PRINCIPLE:-

The discussion of Occam's razor, a popular inductive bias that can be summarized as "choose the shortest explanation for the observed data." we discussed several arguments in the long-standing debate regarding Occam's razor. Here we consider a Bayesian perspective on this issue and a closely related principle called the Minimum Description Length (MDL)principle.

The Minimum Description Length principle is motivated by interpreting the definition of h_{MAP} in the light of basic concepts from information theory. Consider again the now familiar definition of h_{MAP}.

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h)$$

which can be equivalently expressed in terms of maximizing the log,

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}} \log_2 P(D|h) + \log_2 P(h)$$

or alternatively, minimizing the negative of this quantity

$$h_{MAP} = \underset{h \in H}{\operatorname{argmin}} -\log_2 P(D|h) - \log_2 P(h)$$

Somewhat surprisingly, Equation can be interpreted as a statement that short hypotheses are preferred, assuming a particular representation scheme for encoding hypotheses and data. To explain this, let us introduce a basic result from information theory: Consider the problem of designing a code to transmit messages drawn at random, where the probability of encountering message ***i*** is ***pi .*** We are interested here in the most compact code; that is, we are interested in the code that minimizes the expected number of bits we must transmit in order to encode a message drawn at random. Clearly, to minimize the expected code length we should assign shorter codes to messages that are more probable**.** We will refer to the number of bits required to encode message ***i*** using code ***C*** as the ***description length of message i with respect to C,*** which we denote by ***Lc(i)*** Let us interpret Equation in light of the above result from coding theory.

- -log, ***P(h)*** is the description length of ***h*** under the optimal encoding for the hypothesis space H. In other words, this is the size of the description of hypothesis ***h*** using this optimal representation. In our notation, ***LC, (h)*** =

- log, ***P(h),*** where ***CH*** is the optimal code for hypothesis space H.

- -log2 ***P(D1h)*** is the description length of the training data ***D*** given hypothesis ***h,*** under its optimal encoding. In our notation, ***Lc,,,(Dlh)*** =

- log, ***P(Dlh),*** where ***CD,***is~ t he optimal code for describing data ***D*** assuming that both the sender and receiver know the hypothesis ***h.***

- Therefore we can rewrite Equation (6.16) to show that h_{MAP} is the hypothesis h that minimizes the sum given by the description length of the hypothesis plus the description length of the data given the hypothesis.

$$h_{MAP} = \underset{h}{\operatorname{argmin}} L_{C_H}(h) + L_{C_{D|h}}(D|h)$$

where C_H and $C_{D/h}$ are the optimal encodings for H and for ***D*** given h, respectively.

The Minimum Description Length (MDL) principle recommends choosing the hypothesis that minimizes the sum of these two description lengths. Of course to apply this principle in practice we must choose specific encodings or representations appropriate for the given learning task. Assuming we use the codes C_1 and C_2 to represent the hypothesis and the data given the hypothesis, we can state the MDL principle as

Minimum Description Length principle: Choose h_{MDL} where

$$h_{MDL} = \underset{h \in H}{\operatorname{argmin}} L_{C_1}(h) + L_{C_2}(D|h)$$

The above analysis shows that if we choose C_1 to be the optimal encoding of hypotheses C_H, and if we choose C_2 to be the optimal encoding $C_{D/h}$ then $h_{MDL} = h_{MAP}$.

Intuitively, we can think of the MDL principle as recommending the shortest method for re-encoding the training data, where we count both the size of the hypothesis and any additional cost of encoding the data given this hypothesis. Let us consider an example. Thus the MDL principle provides a way of trading off hypothesis complexity for the number of errors committed by the hypothesis. It might select a shorter hypothesis that makes a few errors over a longer hypothesis that perfectly classifies the training data. Viewed in this light, it provides one method for dealing with the issue of ***overfitting*** the data.

BAYES OPTIMAL CLASSIFIER:-

we have considered the question "what is the most probable ***hypothesis*** given the training data?' In fact, the question that is often of most significance is the closely related question "what is the most probable ***classification*** of the new instance given the training data?'Although it may seem that this second question can be answered by simply applying the MAP hypothesis to the new instance, in fact it is possible to do better.

To develop some intuitions consider a hypothesis space containing three hypotheses, h_l, h_2, and h_3. Suppose that the posterior probabilities of these hypotheses given the training data are .4, **.3,** and **.3** respectively. Thus, h_l is the MAP hypothesis. Suppose a new instance x is encountered, which is classified positive by ***hl,*** but negative by h_2 and h_3. Taking all hypotheses into account, the probability that x is positive is .4 (the probability associated with h_i), and the probability that it is negative is therefore .6. The most probable classification (negative) in this case is different from the classification generated by the MAP hypothesis.

In general, the most probable classification of the new instance is obtained by combining the predictions of all hypotheses, weighted by their posterior probabilities. If the possible classification of the new example can take on any value v_j from some set V, then the probability $P(v_j/D)$ that the correct classification for the new instance is v_j, is just

$$P(v_j|D) = \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

The optimal classification of the new instance is the value v_j , for which P (v_j / D) is maximum.

Bayes optimal classification:

$$\underset{v_j \in V}{\operatorname{argmax}} \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

To illustrate in terms of the above example, the set of possible classifications of the new instance is $V = \{\oplus, \ominus\}$, and

$$P(h_1|D) = .4,\ P(\ominus|h_1) = 0,\ P(\oplus|h_1) = 1$$
$$P(h_2|D) = .3,\ P(\ominus|h_2) = 1,\ P(\oplus|h_2) = 0$$
$$P(h_3|D) = .3,\ P(\ominus|h_3) = 1,\ P(\oplus|h_3) = 0$$

therefore

$$\sum_{h_i \in H} P(\oplus|h_i)P(h_i|D) = .4$$
$$\sum_{h_i \in H} P(\ominus|h_i)P(h_i|D) = .6$$

and

$$\underset{v_j \in \{\oplus, \ominus\}}{\operatorname{argmax}} \sum_{h_i \in H} P(v_j|h_i)P(h_i|D) = \ominus$$

Any system that classifies new instances according to Equation is called a ***Bayes optimal classifier,*** or Bayes optimal learner. No other classification method using the same hypothesis space and same prior knowledge can outperform this method on average. This method maximizes the probability that the new instance is classified correctly, given the available data, hypothesis space, and prior probabilities over the hypotheses.

Note one curious property of the Bayes optimal classifier is that the predictions it makes can correspond to a hypothesis not contained in H! Imagine using Equation to classify every instance in X. The labeling of instances defined in this way need not correspond to the instance labeling of any single hypothesis ***h*** from H. One way to view this situation is to think of the Bayes optimal classifier as effectively considering a hypothesis space H' different from the space of hypotheses H to which Bayes theorem is being applied. In particular, ***H'*** effectively includes hypotheses that perform comparisons between linear combinations of predictions from multiple hypotheses in H.

GIBBS ALGORITHM:-

Although the Bayes optimal classifier obtains the best performance that can be achieved from the given training data, it can be quite costly to apply. The expense is due to the fact that it computes the posterior probability for every hypothesis in H and then combines the predictions of each hypothesis to classify each new instance.

An alternative, less optimal method is the Gibbs algorithm (see Opper and Haussler 1991), defined as follows:

1. Choose a hypothesis ***h*** from H at random, according to the posterior probability distribution over H.

2. Use ***h*** to predict the classification of the next instance ***x.***

Given a new instance to classify, the Gibbs algorithm simply applies a hypothesis drawn at random according to the current posterior probability distribution. Surprisingly, it can be shown that under certain conditions the expected misclassification error for the Gibbs algorithm is at most twice the expected error of the Bayes optimal classifier (Haussler et al. 1994). More precisely, the expected value is taken over target concepts drawn at random according to the prior probability distribution assumed by the learner. Under this condition, the expected value of the error of the Gibbs algorithm is at worst twice the expected value of the error of the Bayes optimal classifier.

This result has an interesting implication for the concept learning problem described earlier. In particular, it implies that if the learner assumes a uniform prior over H, and if target concepts are in fact drawn from such a distribution when presented to the learner, ***then classifying the next instance according to a hypothesis drawn at random from the current version space (according to a uniform distribution), will have expected error at most twice that of the Bayes optimal classifier.*** Again, we have an example where a Bayesian analysis of a non-Bayesian algorithm yields insight into the performance of that algorithm.

NAIVE BAYES CLASSIFIER:-

One highly practical Bayesian learning method is the naive Bayes learner, often called the ***naive Bayes classifier.*** In some domains its performance has been shown to be comparable to that of neural network and decision tree learning. This section introduces the naive Bayes classifier; the next section applies it to the practical problem of learning to classify natural language text documents.

The naive Bayes classifier applies to learning tasks where each instance ***x*** is described by a conjunction of attribute values and where the target function f ***(x)*** can take on any value from some finite set V. A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values ***(a_1, a_2.. a_n).*** The learner is asked to predict the target value, or classification, for this new instance.

The Bayesian approach to classifying the new instance is to assign the most probable target value, V_{MAP}, given the attribute values ***(a_1,a_2 . . .a_n)*** that describe the instance.

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j | a_1, a_2 \ldots a_n)$$

We can use Bayes theorem to rewrite this expression as

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2 \ldots a_n | v_j) P(v_j)}{P(a_1, a_2 \ldots a_n)}$$

$$= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2 \ldots a_n | v_j) P(v_j)$$

Now we could attempt to estimate the two terms in Equation based on the training data. It is easy to estimate each of the ***$P(v_j)$*** simply by counting the frequency with which each target value ***v_j*** occurs in the training data. However, estimating

the different $P(a_1, a_2 .. . a_n|v_j)$ terms in this fashion is not feasible unless we have a very, very large set of training data. The problem is that the number of these terms is equal to the number of possible instances times the number of possible target values. Therefore, we need to see every instance in the instance space many times in order to obtain reliable estimates. The naive Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value. In other words, the assumption is that given the target value of the instance, the probability of observing the conjunction $a_1, a_2 .. .a_n$, is just the product of the probabilities for the individual attributes: $P(a_1, a_2 . . . a_n| v_j) = \prod_i P(a_i / v_j)$. Substituting this into Equation, we have the approach used by the naive Bayes classifier.

Naive Bayes classifier:

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i|v_j)$$

where v_{NB} denotes the target value output by the naive Bayes classifier. Notice that in a naive Bayes classifier the number of distinct $P(a_i|v_j)$ terms that must be estimated from the training data is just the number of distinct attribute values times the number of distinct target values-a much smaller number than if we were to estimate the $P(a_1, a_2 . . . a_n |v_j)$ terms as first contemplated.

To summarize, the naive Bayes learning method involves a learning step in which the various $P(v_j)$ and $P(a_i | v_j)$ terms are estimated, based on their frequencies over the training data. The set of these estimates corresponds to the learned hypothesis. This hypothesis is then used to classify each new instance by applying the rule in Equation. Whenever the naive Bayes assumption of conditional independence is satisfied, this naive Bayes classification v_{NB} is identical to the MAP classification.

One interesting difference between the naive Bayes learning method and other learning methods we have considered is that there is no explicit search through the space of possible hypotheses (in this case, the space of possible hypotheses is the space of possible values that can be assigned to the various $P(v_j)$ and $P(a_i | v_j)$ terms). Instead, the hypothesis is formed without searching, simply by counting the frequency of various data combinations within the training examples.

ESTIMATING PROBABILITIES

Up to this point we have estimated probabilities by the fraction of times the event is observed to occur over the total number of opportunities. For example, in the above case we estimated ***P(Wind = strong | Play Tennis = no)*** by the fraction $\frac{n_c}{n}$ where $n = 5$ is the total number of training examples for which ***PlayTennis = no,*** and $n_c = 3$ is the number of these for which ***Wind = strong.*** While this observed fraction provides a good estimate of the probability in many cases, it provides poor estimates when n_c is very small. To see the difficulty, imagine that, in fact, the value of ***P(Wind = strong | PlayTennis = no)*** is ***.08*** and that we have a sample containing only 5 examples for which ***PlayTennis = no.***

Then the most probable value for n_c is ***0.*** This raises two difficulties. Fir $\frac{n_c}{n}$ st, produces a biased underestimate of the probability. Second, when this probability estimate is zero, this probability term will dominate the Bayes classifier if the future query contains ***Wind = strong.*** The reason is that the quantity calculated in Equation requires multiplying all the other probability terms by this zero value. To avoid this difficulty we can adopt a Bayesian approach to estimating the probability, using the m-estimate defined as follows.

m-estimate of probability:

$$\frac{n_c + mp}{n + m}$$

Here, n_c and n are defined as before, p is our prior estimate of the probability we wish to determine, and m is a constant called the ***equivalent sample size,*** which determines how heavily to weight p relative to the observed data. A typical method for choosing p in the absence of other information is to assume uniform priors; that is, if an attribute has k possible values we set $p = 1/k$. For example, in estimating ***P(Wind = strong | PlayTennis = no)*** we note the attribute ***Wind*** has two possible values, so uniform priors would correspond to choosing p = .5. Note that if m is zero, the m-estimate is equivalent to the simple fraction $\frac{n_c}{n}$. If both n and m are nonzero, then the observed fraction $\frac{n_c}{n}$ and prior p will be combined according to the weight m. The reason m is called the equivalent sample size is that Equation can be interpreted as augmenting the n actual observations by an additional m virtual samples distributed according to p.

BAYESIAN BELIEF NETWORKS:-

As discussed in the previous two sections, the naive Bayes classifier makes significant set of the assumption that the values of the attributes $a_1 \ldots a_n$, are conditionally independent given the target value v. This assumption dramatically reduces the complexity of learning the target function. When it is met, the naive Bayes classifier outputs the optimal Bayes classification. However, in many cases this conditional independence assumption is clearly overly restrictive.

A Bayesian belief network describes the probability distribution governing a set of variables by specifying a set of conditional independence assumptions along with a set of conditional probabilities. In contrast to the naive Bayes classifier, which assumes that all the variables are conditionally independent given the value of the target variable, Bayesian belief networks allow stating conditional independence assumptions that apply to ***subsets*** of the variables. Thus, Bayesian belief networks provide an intermediate approach that is less constraining than the global assumption of conditional independence made by the naive Bayes classifier, but more tractable than avoiding conditional independence assumptions altogether. Bayesian belief networks are an active focus of current research, and a variety of algorithms have been proposed for learning them and for using them for inference.

We define the ***joint space*** of the set of variables Y to be the cross product $V(Y_1) \times V(Y_2) \times \ldots V(Y_n)$. In other words, each item in the joint space corresponds to one of the possible assignments of values to the tuple of variables $(Y_1 \ldots Y_n)$. The probability distribution over this joint space is called the ***joint probability distribution.*** The joint probability distribution specifies the probability for each of the possible variable bindings for the tuple $(Y_1 \ldots Y_n)$. A Bayesian belief network describes the joint probability distribution for a set of variables.

Conditional Independence:-

Let us begin our discussion of Bayesian belief networks by defining precisely the notion of conditional independence. Let X, Y, and Z be three discrete-valued random variables. We say that X is ***conditionally independent*** of Y given Z if the probability distribution governing X is independent of the value of Y given a value for Z that is, if

$$(\forall x_i, y_j, z_k)\ P(X = x_i | Y = y_j, Z = z_k) = P(X = x_i | Z = z_k)$$

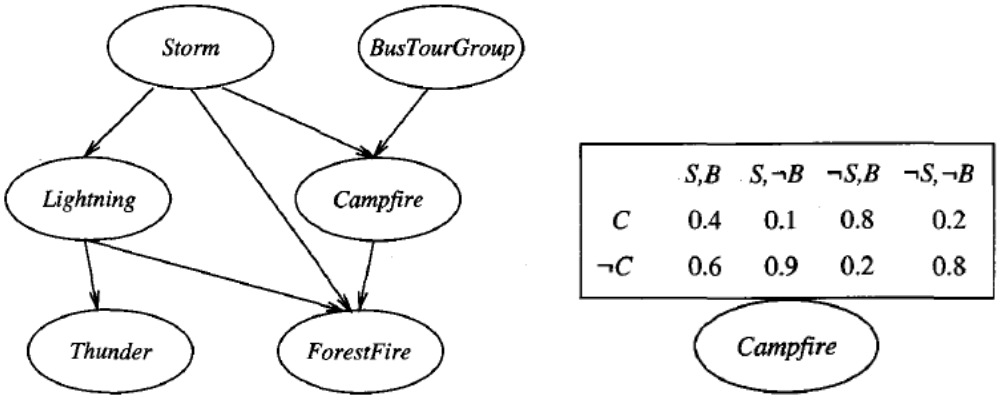
where $x_i \in V(X)$, $y_j \in V(Y)$, and $Z_k \in V(Z)$. We commonly write the above expression in abbreviated form as P(X|Y, Z) = P(X|Z). This definition of conditional independence can be extended to sets of variables as well. We say that the set of variables $X_1 \ldots X_l$ is conditionally independent of the set of variables $Y_1 \ldots Y_m$ given the set of variables $Z_1 \ldots Z_n$, if

$$P(X_1 \ldots X_l | Y_1 \ldots Y_m, Z_1 \ldots Z_n) = P(X_1 \ldots X_l | Z_1 \ldots Z_n)$$

Note the correspondence between this definition and our use of conditional independence in the definition of the naive Bayes classifier. The naive Bayes classifier assumes that the instance attribute A_1 is conditionally independent of instance attribute A_2 given the target value V. This allows the naive Bayes classifier to calculate $P(A_1, A_2 \mid V)$ in Equation as follows

$$\begin{aligned} P(A_1, A_2|V) &= P(A_1|A_2, V)P(A_2|V) \\ &= P(A_1|V)P(A_2|V) \end{aligned}$$

Equation is just the general form of the product rule of probability. Equation follows because if A_1 is conditionally independent of A_2 given V, then by our definition of conditional independence $P(A_1|A_2, V) = P(A_1|V)$.



Representation:-

A Bayesian belief network (Bayesian network for short) represents the joint probability distribution for a set of variables. represents the joint probability distribution over the boolean variables ***Storm, Lightning, Thunder, ForestFire, Campfire,*** and ***BusTourGroup.*** In general, a Bayesian network represents the joint probability distribution by specifying a set of conditional independence assumptions (represented by a directed acyclic graph), together with sets of local conditional probabilities. Each variable in the joint space is represented by a node in the Bayesian network. For each variable two types of information are specified. First, the network arcs represent the assertion that the variable is conditionally independent of its nondescendants in the network given its immediate predecessors in the network. We say X is a ***descendant*** of , ***Y*** if there is a directed path from ***Y*** to X. Second, a conditional probability table is given for each variable, describing the probability distribution for that variable given the values of its immediate predecessors. The joint probability for any desired assignment of values $(y_1, \ldots, y_n)$ to the tuple of network variables $(Y_1 \ldots Y_n)$ can be computed by the formula

$$P(y_1, \ldots, y_n) = \prod_{i=1}^{n} P(y_i | Parents(Y_i))$$

where $Parents(Y_i)$ denotes the set of immediate predecessors of Y_i in the network. Note the values of $P(yi|P arents(Y_i))$ are precisely the values stored in the conditional probability table associated with node Y_i. To illustrate, the Bayesian network represents the joint probability distribution over the boolean variables ***Storm, Lightning, Thunder, Fire, Campfire,*** and ***BusTourGroup.*** Consider the node ***Campfire.*** The network nodes and arcs represent the assertion that ***Campfire*** is conditionally independent of its nondescendants ***Lightning*** and ***Thunder,*** given its immediate parents ***Storm*** and ***BusTourGroup.*** This means that once we know the value of the variables ***Storm*** and ***BusTourGroup,*** the variables ***Lightning*** and ***Thunder*** provide no additional information about ***Campfire.*** The right side of the figure shows the

conditional probability table associated with the variable ***Campfire.*** The top left entry in this table, for example, expresses the assertion that

$$P(Campfire = True|Storm = True, BusTourGroup = True) = 0.4$$

Note this table provides only the conditional probabilities of ***Campjire*** given its parent variables ***Storm*** and B***usTourGroup.*** The set of local conditional probability tables for all the variables, together with the set of conditional independence assumptions described by the network, describe the full joint probability distribution for the network.

One attractive feature of Bayesian belief networks is that they allow a convenient way to represent causal knowledge such as the fact that ***Lightning*** causes ***Thunder.*** In the terminology of conditional independence, we express this by stating that ***Thunder*** is conditionally independent of other variables in the network, given the value of ***Lightning.*** Note this conditional independence assumption is implied by the arcs in the Bayesian network.

Inference

We might wish to use a Bayesian network to infer the value of some target variable (e.g., ***ForestFire)*** given the observed values of the other variables. Of course, given that we are dealing with random variables it will not generally be correct to assign the target variable a single determined value. What we really wish to infer is the probability distribution for the target variable, which specifies the probability that it will take on each of its possible values given the observed values of the other variables. This inference step can be straightforward if values for all of the other variables in the network are known exactly. In the more general case we may wish to infer the probability distribution for some variable (e.g., ***ForestFire)*** given observed values for only a subset of the other variables (e.g., ***Thunder*** and ***BusTourGroup*** may be the only observed values available). In general, a Bayesian network can be used to compute the probability distribution for any subset of network variables given the values or distributions for any subset of the remaining variables.

Exact inference of probabilities in general for an arbitrary Bayesian network is known to be NP-hard.

Learning Bayesian Belief Networks

Can we devise effective algorithms for learning Bayesian belief networks from training data? This question is a focus of much current research. Several different settings for this learning problem can be considered. First, the network structure might be given in advance, or it might have to be inferred from the training data.

Second, all the network variables might be directly observable in each training example, or some might be unobservable.

In the case where the network structure is given in advance and the variables are fully observable in the training examples, learning the conditional probability tables is straightforward. We simply estimate the conditional probability table entries just as we would for a naive Bayes classifier.

In the case where the network structure is given but only some of the variable values are observable in the training data, the learning problem is more difficult.

This problem is somewhat analogous to learning the weights for the hidden units in an artificial neural network, where the input and output node values are given but the hidden unit values are left unspecified by the training examples.

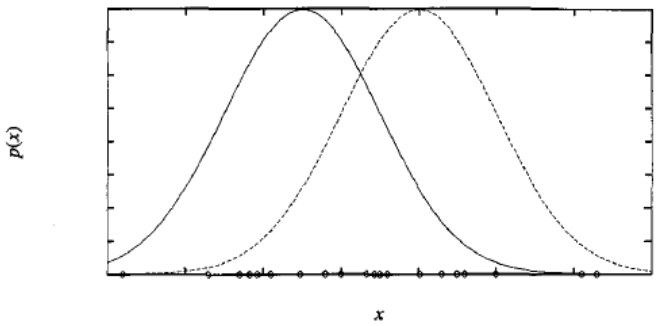
THE EM ALGORITHM

In many practical learning settings, only a subset of the relevant instance features might be observable. For example, in training or using the Bayesian belief network, we might have data where only a subset of the network variables ***Storm, Lightning, Thunder, ForestFire, Campfire,*** and ***BusTourGroup*** have been observed. Many approaches have been proposed to handle the problem of learning in the presence of unobserved variables. if some variable is sometimes observed and sometimes not, then we can use the cases for which it has been observed to learn to predict its values when it is not. In this section we describe the EM algorithm (Dempster et al. 1977), a widely used approach to learning in the presence of unobserved variables. The EM algorithm can be used even for variables whose value is never directly observed, provided the general form of the probability distribution governing these variables is known.

The EM algorithm has been used to train Bayesian belief networks (see Heckerman 1995) as well as radial basis function networks. The EM algorithm is also the basis for many unsupervised clustering algorithms (e.g., Cheeseman et al. 1988), and it is the basis for the widely used Baum-Welch forward-backward algorithm for learning Partially Observable Markov Models (Rabiner 1989).

Estimating Means of k Gaussians

The easiest way to introduce the EM algorithm is via an example. Consider a problem in which the data D is a set of instances generated by a probability distribution that is a mixture of k distinct Normal distributions. for the case where k = 2 and where the instances are the points shown along the ***x*** axis. Each instance is generated using a two-step process. First, one of the k Normal distributions is selected at random. Second, a single random instance ***xi*** is generated according to this selected distribution. This process is repeated to generate a set of data points as shown in the figure. To simplify our discussion, we consider the special case where the selection of the single Normal distribution at each step is based on choosing each with uniform probability, where each of the k Normal distributions has the same variance a2, and where a2 is known. The learning task is to output a hypothesis h = (FI, . . . ***pk)*** that describes the means of each of the k distributions. We would like to find a maximum likelihood hypothesis for these means; that is, a hypothesis h that maximizes ***p(D*** lh).



Note it is easy to calculate the maximum likelihood hypothesis for the mean of a single Normal distribution given the observed data instances ***x1, x2,*** . . . , ***xm*** drawn from this single distribution. This problem of finding the mean of a single distribution is just a special case of the problem discussed, Equation, where we showed that the maximum likelihood hypothesis is the one that minimizes the sum of squared errors over the m training instances. Restating Equation using our current notation, we have

$$\mu_{ML} = \underset{\mu}{\operatorname{argmin}} \sum_{i=1}^{m} (x_i - \mu)^2$$

In this case, the sum of squared errors is minimized by the sample mean

$$\mu_{ML} = \frac{1}{m} \sum_{i=1}^{m} x_i$$

Our problem here, however, involves a mixture of k different Normal distributions, and we cannot observe which instances were generated by which distribution.

Thus, we have a prototypical example of a problem involving hidden variables. In the example of Figure , we can think of the full description of each instance as the triple ***(xi,zi 1 , ziz),*** where ***xi*** is the observed value of the ith instance and where ***zi1*** and ***zi2*** indicate which of the two Normal distributions was used to generate the value ***xi.*** In particular, ***zij*** has the value 1 if ***xi*** was created by the jth Normal distribution and **0** otherwise. Here ***xi*** is the observed variable in the description of the instance, and ***zil*** and ***zi2*** are hidden variables. If the values of ***zi1*** and ***zi2*** were observed, we could use Equation to solve for the means ***p1*** and ***p2.*** Because they are not, we will instead use the EM algorithm. Applied to our k-means problem the EM algorithm searches for a maximum likelihood hypothesis by repeatedly re-estimating the expected values of the hidden variables ***zij*** given its current hypothesis ***(pI . . . pk),*** then recalculating the maximum likelihood hypothesis using these expected values for the hidden variables.

We will first describe this instance of the EM algorithm, and later state the EM algorithm in its general form.

Applied to the problem of estimating the two means for Figure, the EM algorithm first initializes the hypothesis to h = **(Pl, p2),** where **p1** and **p2** are arbitrary initial values. It then iteratively re-estimates h by repeating the following

two steps until the procedure converges to a stationary value for h.

Step 1: Calculate the expected value ***E[zij]*** of each hidden variable ***zi,,*** assuming the current hypothesis h = **(p1, p2)** holds.

Step 2: Calculate a new maximum likelihood hypothesis h' = **(pi, p;),** assuming the value taken on by each hidden variable ***zij*** is its expected value ***E[zij]*** calculated in Step 1. Then replace the hypothesis h = **(pl, p2)** by the new hypothesis h' = **(pi, pi)** and iterate.

Let us examine how both of these steps can be implemented in practice. Step 1 must calculate the expected value of each ***zi,.*** This ***E[4]*** is just the probability that instance ***xi*** was generated by the jth Normal distribution

$$E[z_{ij}] = \frac{p(x = x_i|\mu = \mu_j)}{\sum_{n=1}^{2} p(x = x_i|\mu = \mu_n)}$$
$$= \frac{e^{-\frac{1}{2\sigma^2}(x_i-\mu_j)^2}}{\sum_{n=1}^{2} e^{-\frac{1}{2\sigma^2}(x_i-\mu_n)^2}}$$

Thus the first step is implemented by substituting the current values **(μ_1, μ_2)** and the observed X_i into the above expression.

In the second step we use the $E[z_{ij}]$ calculated during Step 1 to derive a new maximum likelihood hypothesis h' = **(μ'_1, μ'_2).** AS we will discuss later, the maximum likelihood hypothesis in this case is given by

$$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E[z_{ij}]\ x_i}{\sum_{i=1}^{m} E[z_{ij}]}$$

Note this expression is similar to the sample mean from Equation that is used to estimate μ for a single Normal distribution. Our new expression is just the weighted sample mean for μ_j with each instance weighted by the expectation $E[z_{ij}]$ that it was generated by the j^{th} Normal distribution.

The above algorithm for estimating the means of a mixture of k Normal distributions illustrates the essence of the EM approach: The current hypothesis is used to estimate the unobserved variables, and the expected values of these variables are then used to calculate an improved hypothesis. It can be proved that on each iteration through this loop, the EM algorithm increases the likelihood *P(D/h)* unless it is at a local maximum. The algorithm thus converges to a local maximum likelihood hypothesis for (μ_1, μ_2).

General Statement of EM Algorithm

Above we described an EM algorithm for the problem of estimating means of a mixture of Normal distributions. More generally, the EM algorithm can be applied in many settings where we wish to estimate some set of parameters Θ that describe an underlying probability distribution, given only the observed portion of the full data produced by this distribution. In the above two-means example the parameters of interest were $\Theta = (\mu_1, \mu_2)$, and the full data were the triples (x_i, z_{i1}, z_{i2}) of which only the x_i were observed. In general let $X = \{x_1, \ldots, x_m\}$ denote the observed data in a set of m independently drawn instances, let $Z = \{z_1, \ldots, z_m\}$ denote the unobserved data in these same instances, and let $Y = X \cup Z$ denote the full data. Note the unobserved Z can be treated as a random variable whose probability distribution depends on the unknown parameters Θ and on the observed data X. Similarly, Y is a random variable because it is defined in terms of the random variable Z. In the remainder of this section we describe the general form of the EM algorithm. We use h to denote the current hypothesized values of the parameters Θ, and h' to denote the revised hypothesis that is estimated on each iteration of the EM algorithm.

The EM algorithm searches for the maximum likelihood hypothesis h' by seeking the h' that maximizes $E[\ln P(Y|h')]$. This expected value is taken over the probability distribution governing Y, which is determined by the unknown parameters Θ. Let us consider exactly what this expression signifies. First, $P(Y/h')$ is the likelihood of the full data Y given hypothesis h'. It is reasonable that we wish to find a h' that maximizes some function of this quantity. Second, maximizing the logarithm of this quantity In $P(Y|h')$ also maximizes $P(Y|h^1)$, as we have discussed on several occasions already. Third, we introduce the expected value $E[\ln P(Y|h')]$ because the full data Y is itself a random variable. Given that the full data Y is a combination of the observed data X and unobserved data Z, we must average over the possible values of the unobserved Z, weighting each according to its probability. In other words we take the expected value $E[\ln P(Y|h')]$ over the probability distribution governing the random variable Y. The distribution governing Y is determined by the completely known values for X, plus the distribution governing Z.

What is the probability distribution governing Y? In general we will not know this distribution because it is determined by the parameters Θ that we are trying to estimate. Therefore, the EM algorithm uses its current hypothesis h in place of the actual parameters Θ to estimate the distribution governing Y. Let us define a function $Q(h^1/h)$ that gives $E[\ln P(Y|h')]$ as a function of h', under the assumption that $\Theta = h$ and given the observed portion X of the full data Y.

$$Q(h'|h) = E[\ln p(Y|h')|h, X]$$

We write this function Q in the form $Q(h^1/h)$ to indicate that it is defined in part by the assumption that the current hypothesis h is equal to Θ. In its general form, the EM algorithm repeats the following two steps until convergence:

Step 1: ***Estimation* (E) *step:*** Calculate $Q(h^1/h)$ using the current hypothesis h and the observed data X to estimate the probability distribution over Y.

$$Q(h'|h) \leftarrow E[\ln P(Y|h')|h, X]$$

Step 2: ***Maximization (M) step:*** Replace hypothesis h by the hypothesis h' that maximizes this Q function.

$$h \leftarrow \underset{h'}{\operatorname{argmax}}\, Q(h'|h)$$

When the function ***Q*** is continuous, the EM algorithm converges to a stationary point of the likelihood function ***P(Y/h[l]).*** When this likelihood function has a single maximum, EM will converge to this global maximum likelihood estimate for h'. Otherwise, it is guaranteed only to converge to a local maximum. In this respect, EM shares some of the same limitations as other optimization methods such as gradient descent, line search, and conjugate gradient.

Derivation of the k Means Algorithm

To illustrate the general EM algorithm, let us use it to derive the algorithm given for estimating the means of a mixture of *k* Normal distributions. As discussed above, the k-means problem is to estimate the parameters $\Theta = (\mu_1 \ldots \mu_k)$

that define the means of the k Normal distributions. We are given the observed data X = $\{(x_i)\}$. The hidden variables Z = $\{(z_{il}, \ldots, z_{ik})\}$ in this case indicate which of the *k* Normal distributions was used to generate x_i.

To apply EM we must derive an expression for ***Q(h|h')*** that applies to our *k*-means problem. First, let us derive an expression for ***1n p(Y| h').N***ote the probability ***p(yi|h')*** of a single instance $y_i = (x_i, z_{il}, \ldots z_{ik})$ of the full data can

be written

$$p(y_i|h') = p(x_i, z_{i1}, \ldots, z_{ik}|h') = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}\sum_{j=1}^{k} z_{ij}(x_i-\mu'_j)^2}$$

To verify this note that only one of the z_{ij} can have the value 1, and all others must be 0. Therefore, this expression gives the probability distribution for x_i generated by the selected Normal distribution. Given this probability for a single instance ***p(yi|h'),*** the logarithm of the probability ***1n p(Y| h')***for all ***m*** instances in the data is

$$\begin{aligned}
\ln P(Y|h') &= \ln \prod_{i=1}^{m} p(y_i|h') \\
&= \sum_{i=1}^{m} \ln p(y_i|h') \\
&= \sum_{i=1}^{m} \left(\ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{j=1}^{k} z_{ij}(x_i - \mu'_j)^2 \right)
\end{aligned}$$

Finally we must take the expected value of this ***1n p(Y| h')***over the probability distribution governing Y or, equivalently, over the distribution governing the unobserved components z_{ij} of Y. Note the above expression for In ***p(yi|h')***is a linear function of these z_{ij}. In general, for any function f (z) that is a linear function of z, the following equality holds

$$E[f(z)] = f(E[z])$$

UNIT – 3

DIMENSIONALITY REDUCTION

Why Reduce Dimensionality?

- **Reduces time complexity: Less computation**
- **Reduces space complexity: Less parameters**
- **Saves the cost of observing the feature**
- **Simpler models are more robust on small datasets**
- **More interpretable; simpler explanation**
- **Data visualization (structure, groups, outliers, etc) if plotted in 2 or 3 dimensions**

Feature Selection vs Extraction

- **Feature selection: Choosing $k<d$ important features, ignoring the remaining $d - k$**

  Subset selection algorithms

- **Feature extraction: Project the**

  original x_i , i =1,...,d dimensions to

  new $k<d$ dimensions, z_j , j =1,...,k

  Principal components analysis (PCA), linear discriminant analysis (LDA), factor analysis (FA)

Subset Selection

- **There are 2^d subsets of d features**
- **Forward search: Add the best feature at each step**
  - **Set of features F initially Ø.**

- **At each iteration, find the best new feature**

  $j = \text{argmin}_i E(F \cup x_i)$

- **Add x_j to F if $E(F \cup x_j) < E(F)$**

- **Hill-climbing $O(d^2)$ algorithm**
- **Backward search: Start with all features and remove one at a time, if possible.**
- **Floating search (Add k, remove l)**

Principal Components Analysis (PCA)

- **Find a low-dimensional space such that when x is projected there, information loss is minimized.**
- **The projection of x on the direction of w is: $z = w^T x$**
- **Find w such that Var(z) is maximized**

$$\text{Var}(z) = \text{Var}(w^T x) = E[(w^T x - w^T \mu)^2]$$

$$= E[(w^T x - w^T \mu)(w^T x - w^T \mu)]$$

$$= E[w^T (x - \mu)(x - \mu)^T w]$$

$$= w^T E[(x - \mu)(x - \mu)^T] w = w^T \Sigma w$$

where $\text{Var}(x) = E[(x - \mu)(x - \mu)^T] = \Sigma$

- **Maximize Var(z) subject to $||w||=1$**

$\Sigma w_1 = \alpha w_1$ **that is,** w_1 **is an eigenvector of** Σ

Choose the one with the largest eigenvalue for Var(z) to be max

- **Second principal component: Max Var(z_2), s.t., $||w_2||=1$ and orthogonal to w_1**

$\Sigma w_2 = \alpha w_2$ **that is,** w_2 **is another eigenvector of** Σ

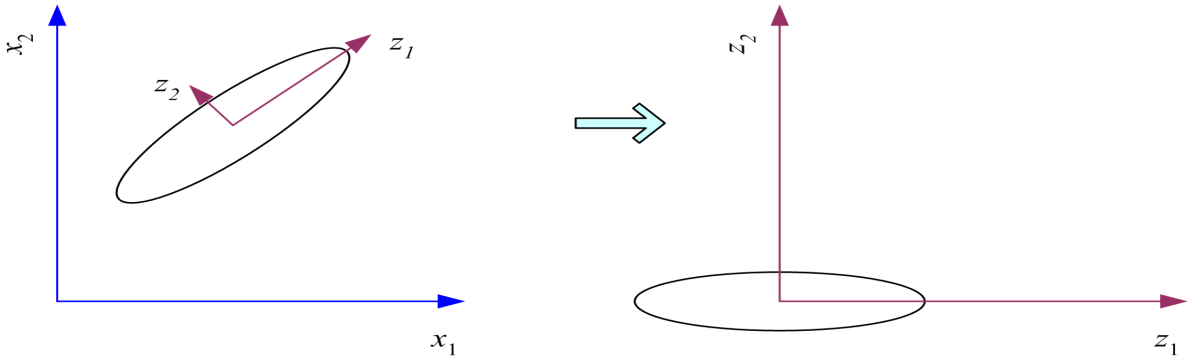
and so on.

What PCA does

$z = W^T(x - m)$

where the columns of W are the eigenvectors of ∑, and *m* is sample mean

Centers the data at the origin and rotates the axes



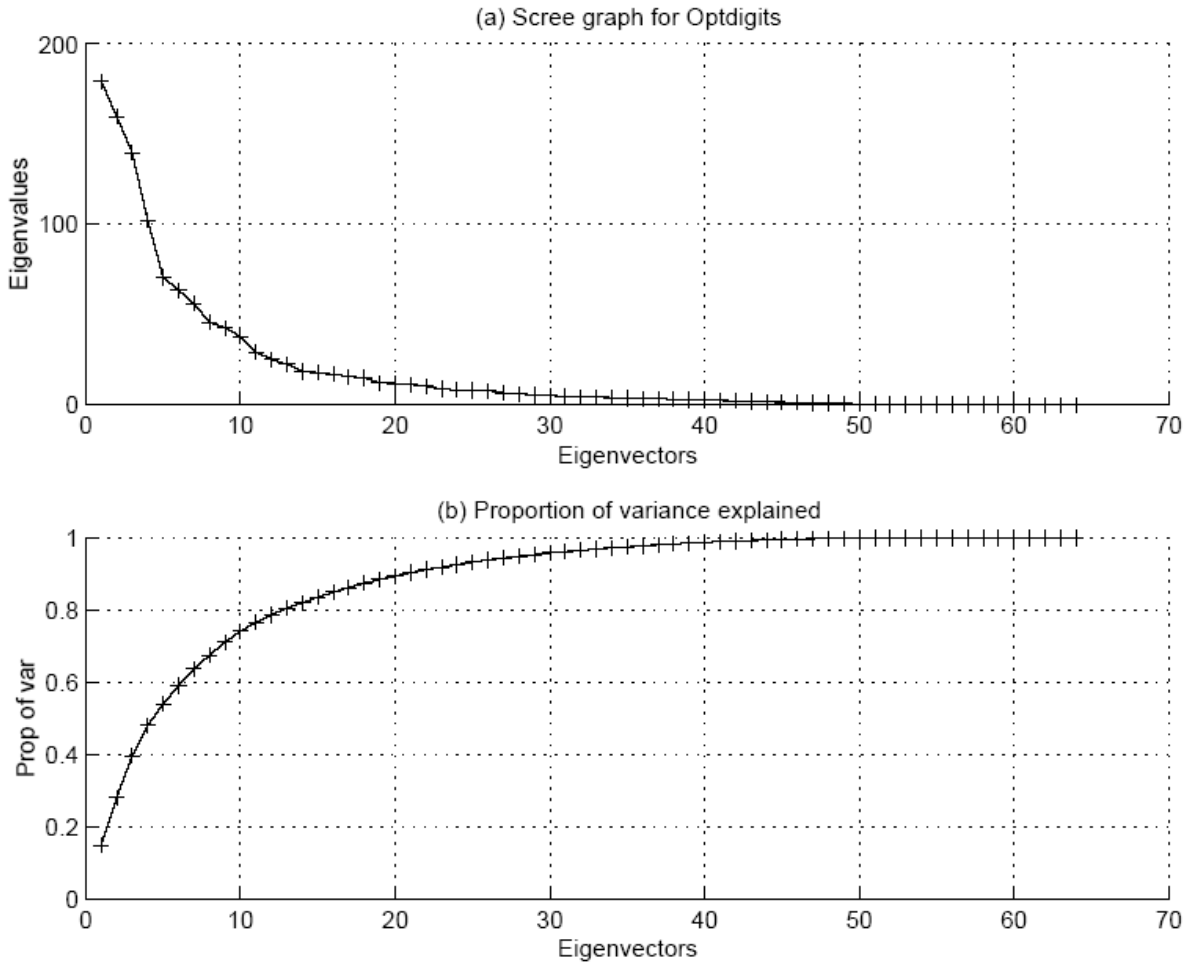
How to choose k ?

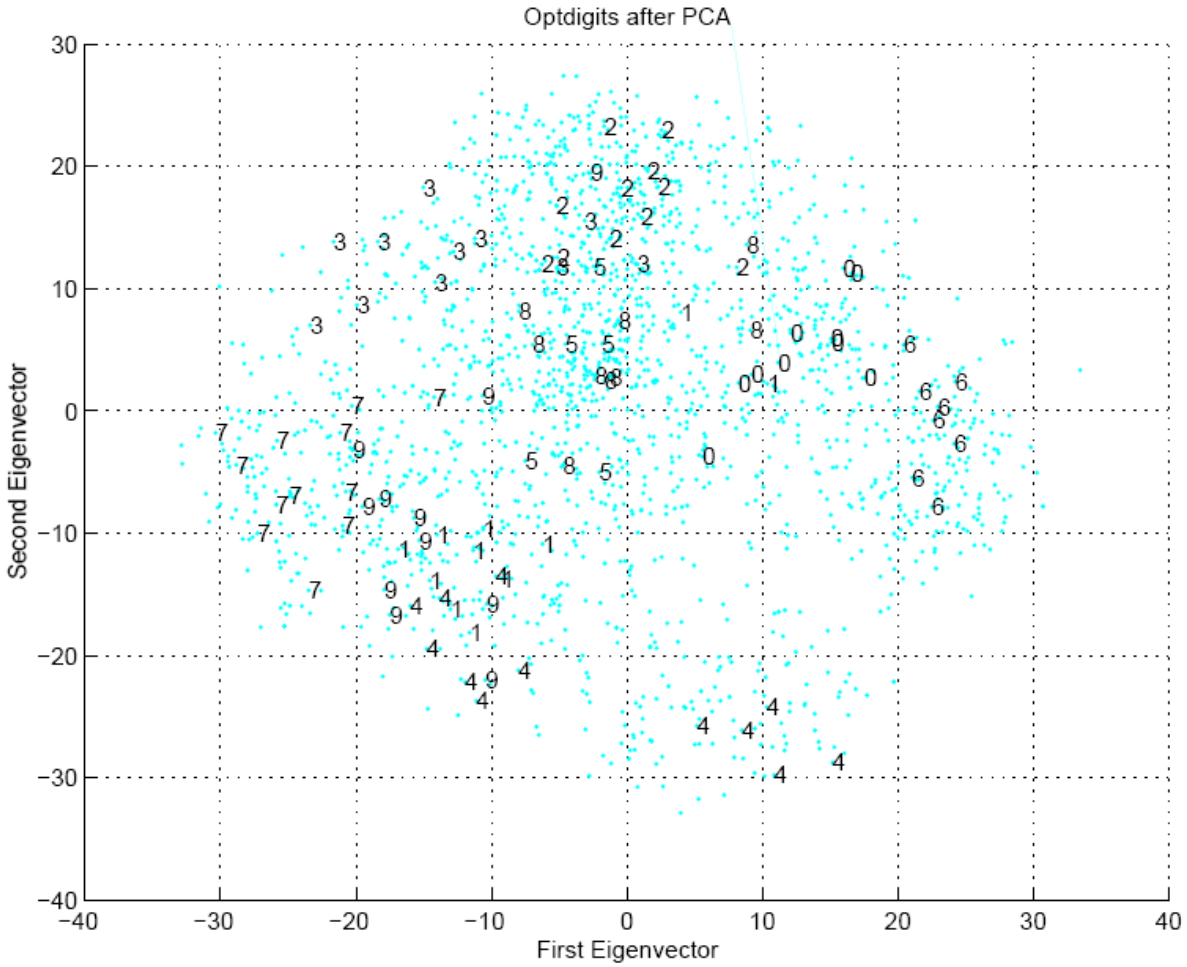
- **Proportion of Variance (PoV) explained**

$$\frac{\lambda_1 + \lambda_2 + \cdots + \lambda_k}{\lambda_1 + \lambda_2 + \cdots + \lambda_k + \cdots + \lambda_d}$$

when λ_i are sorted in descending order

- **Typically, stop at PoV>0.9**
- **Scree graph plots of PoV vs *k*, stop at "elbow"**


(a) Scree graph for Optdigits
Eigenvalues
Eigenvectors
0
100
200
0
10
20
30
40
50
60
70
(b) Proportion of variance explained
Prop of var
Eigenvectors
0
0.2
0.4
0.6
0.8
1
0
10
20
30
40
50
60
70



Factor Analysis

- **Find a small number of factors *z*, which when combined generate *x* :**

$$x_i - \mu_i = v_{i1}z_1 + v_{i2}z_2 + \ldots + v_{ik}z_k + \varepsilon_i$$

where z_j, j =1,...,k are the latent factors with

$$E[z_j]=0,\ \mathrm{Var}(z_j)=1,\ \mathrm{Cov}(z_i, z_j)=0,\ i \neq j,$$

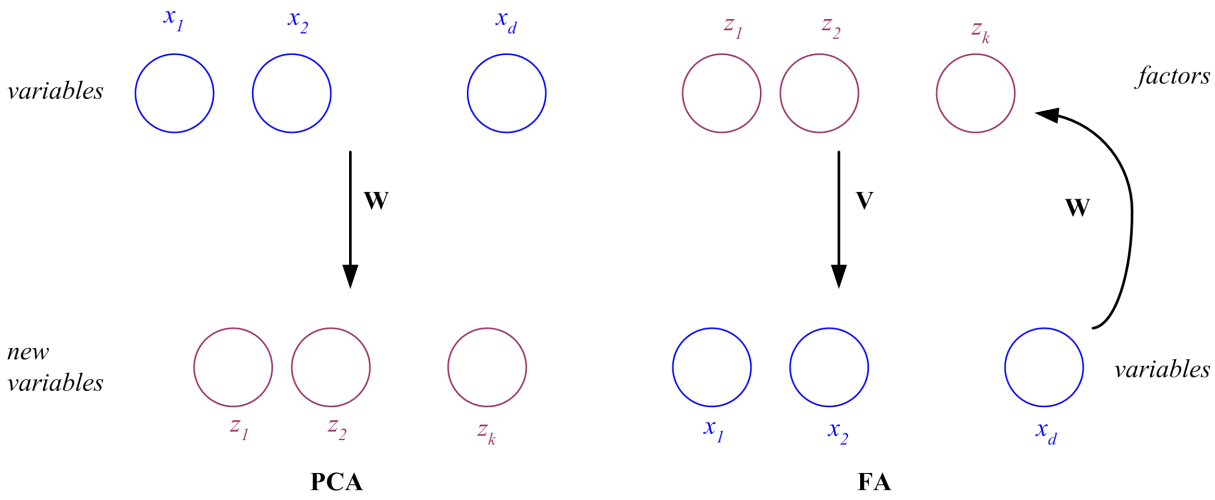
ε_i are the noise sources

$$E[\varepsilon_i]=\psi_i,\ \mathrm{Cov}(\varepsilon_i, \varepsilon_j)=0,\ i \neq j,\ \mathrm{Cov}(\varepsilon_i, z_j)=0,$$

and v_{ij} are the factor loadings

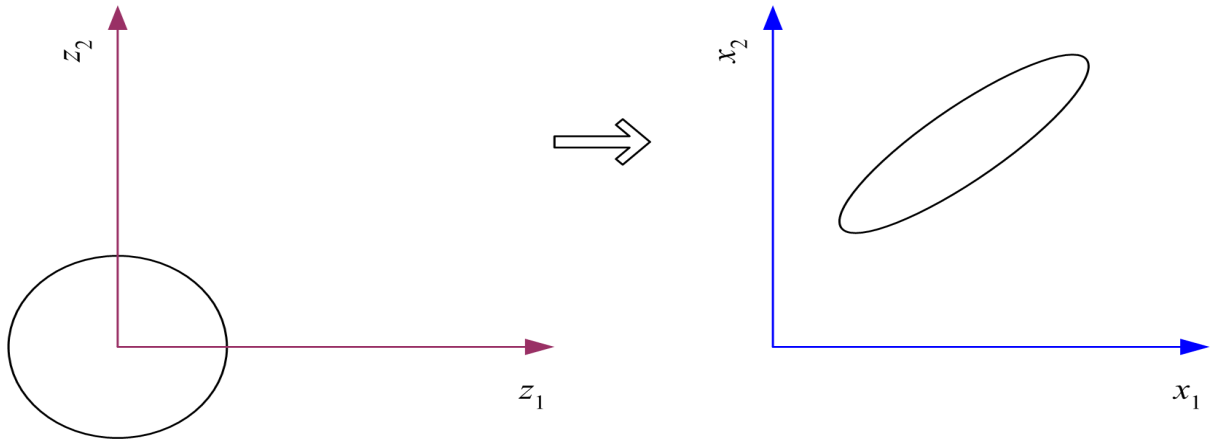
PCA vs FA

- **PCA From x to z** $\quad z = W^T(x - \mu)$
- **FA** $\quad$ **From z to x** $\quad x - \mu = Vz + \varepsilon$



Factor Analysis

- **In FA, factors z_j are stretched, rotated and translated to generate x**



Multidimensional Scaling

- **Given pairwise distances between N points,**

$$d_{ij},\ i,j = 1,\ldots,N$$

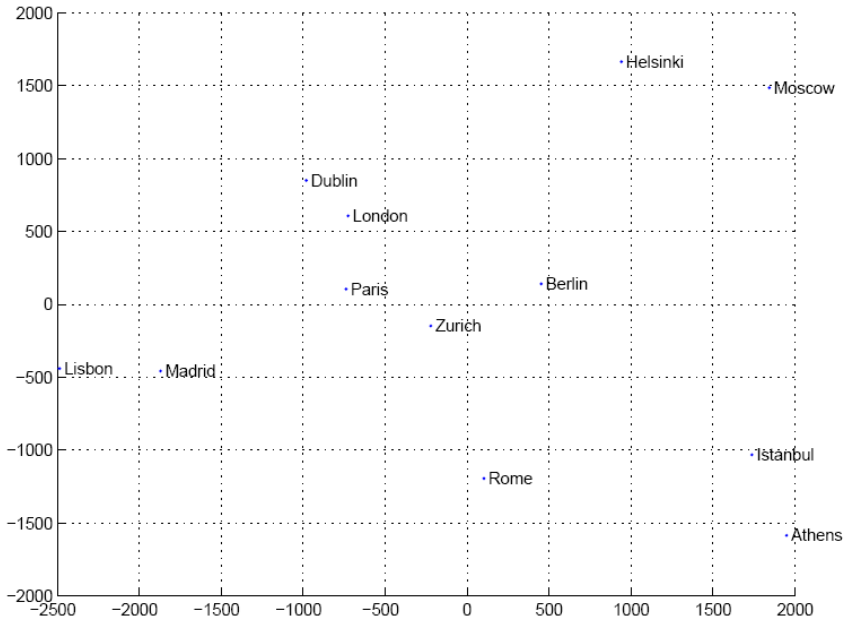
place on a low-dim map s.t. distances are preserved.

$z = g(x \mid \theta)$ **Find θ that min Sammon stress**

$$E(\theta \mid \mathcal{X}) = \sum_{r,s} \frac{\left(\left\|\mathbf{z}^r - \mathbf{z}^s\right\| - \left\|\mathbf{x}^r - \mathbf{x}^s\right\|\right)^2}{\left\|\mathbf{x}^r - \mathbf{x}^s\right\|^2}$$

$$= \sum_{r,s} \frac{\left(\left\|\mathbf{g}\left(\mathbf{x}^r \mid \theta\right) - \mathbf{g}\left(\mathbf{x}^s \mid \theta\right)\right\| - \left\|\mathbf{x}^r - \mathbf{x}^s\right\|\right)^2}{\left\|\mathbf{x}^r - \mathbf{x}^s\right\|^2}$$

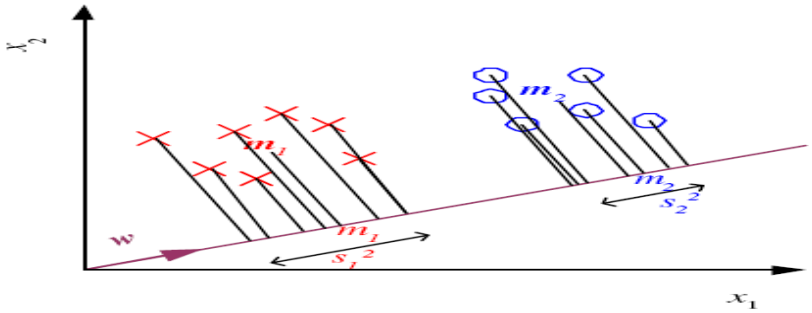
Map of Europe by MDS



Map from CIA – The World Factbook: http://www.cia.gov/

Linear Discriminant Analysis

- **Find a low-dimensional space such that when *x* is projected, classes are well-separated.**



- **Find *w* that maximizes**

$$J(\mathbf{w}) = \frac{(m_1 - m_2)^2}{s_1^2 + s_2^2}$$

$$m_1 = \frac{\sum_t \mathbf{w}^T \mathbf{x}^t r^t}{\sum_t r^t} \quad s_1^2 = \sum_t \left(\mathbf{w}^T \mathbf{x}^t - m_1\right)^2 r^t$$

- **Between-class scatter:**

$$\begin{aligned}(m_1 - m_2)^2 &= \left(\mathbf{w}^T \mathbf{m}_1 - \mathbf{w}^T \mathbf{m}_2\right)^2 \\ &= \mathbf{w}^T (\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T \mathbf{w} \\ &= \mathbf{w}^T \mathbf{S}_B \mathbf{w} \text{ where } \mathbf{S}_B = (\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T\end{aligned}$$

- **Within-class scatter:**

$$s_1^2 = \sum_t \left(\mathbf{w}^T \mathbf{x}^t - m_1\right)^2 r^t$$
$$= \sum_t \mathbf{w}^T \left(\mathbf{x}^t - \mathbf{m}_1\right)\left(\mathbf{x}^t - \mathbf{m}_1\right)^T \mathbf{w} r^t = \mathbf{w}^T \mathbf{S}_1 \mathbf{w}$$

where $\mathbf{S}_1 = \sum_t \left(\mathbf{x}^t - \mathbf{m}_1\right)\left(\mathbf{x}^t - \mathbf{m}_1\right)^T r^t$

$s_1^2 + s_1^2 = \mathbf{w}^T \mathbf{S}_W \mathbf{w}$ where $\mathbf{S}_W = \mathbf{S}_1 + \mathbf{S}_2$

Fisher's Linear Discriminant

- **Find *w* that max**

$$J(\mathbf{w}) = \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}} = \frac{\left|\mathbf{w}^T \left(\mathbf{m}_1 - \mathbf{m}_2\right)\right|^2}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}}$$

- **LDA soln:**

$$\mathbf{w} = c \cdot \mathbf{S}_W^{-1}\left(\mathbf{m}_1 - \mathbf{m}_2\right)$$

- **Parametric soln:**

$$\mathbf{w} = \Sigma^{-1}\left(\mu_1 - \mu_2\right)$$
when $p(\mathbf{x} \mid C_i) \sim \mathcal{N}(\mu_i, \Sigma)$

K>2 Classes

- Within-class scatter:

$$\mathbf{S}_W = \sum_{i=1}^{K} \mathbf{S}_i \qquad \mathbf{S}_i = \sum_t r_i^t \left(\mathbf{x}^t - \mathbf{m}_i\right)\left(\mathbf{x}^t - \mathbf{m}_i\right)^T$$

- Between-class scatter:

$$\mathbf{S}_B = \sum_{i=1}^{K} N_i \left(\mathbf{m}_i - \mathbf{m}\right)\left(\mathbf{m}_i - \mathbf{m}\right)^T \qquad \mathbf{m} = \frac{1}{K}\sum_{i=1}^{K} \mathbf{m}_i$$

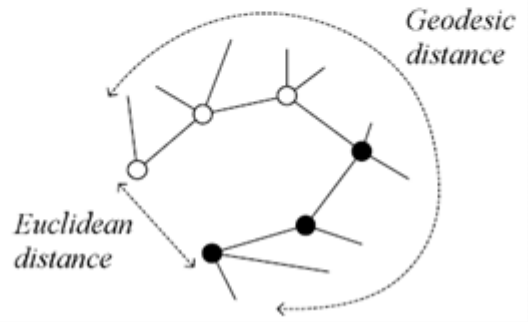
- Find $\mathbf{W}$ that max

$$J(\mathbf{W}) = \frac{\left|\mathbf{W}^T \mathbf{S}_B \mathbf{W}\right|}{\left|\mathbf{W}^T \mathbf{S}_W \mathbf{W}\right|}$$

The largest eigenvectors of $\mathbf{S}_W^{-1}\mathbf{S}_B$
Maximum rank of K-1

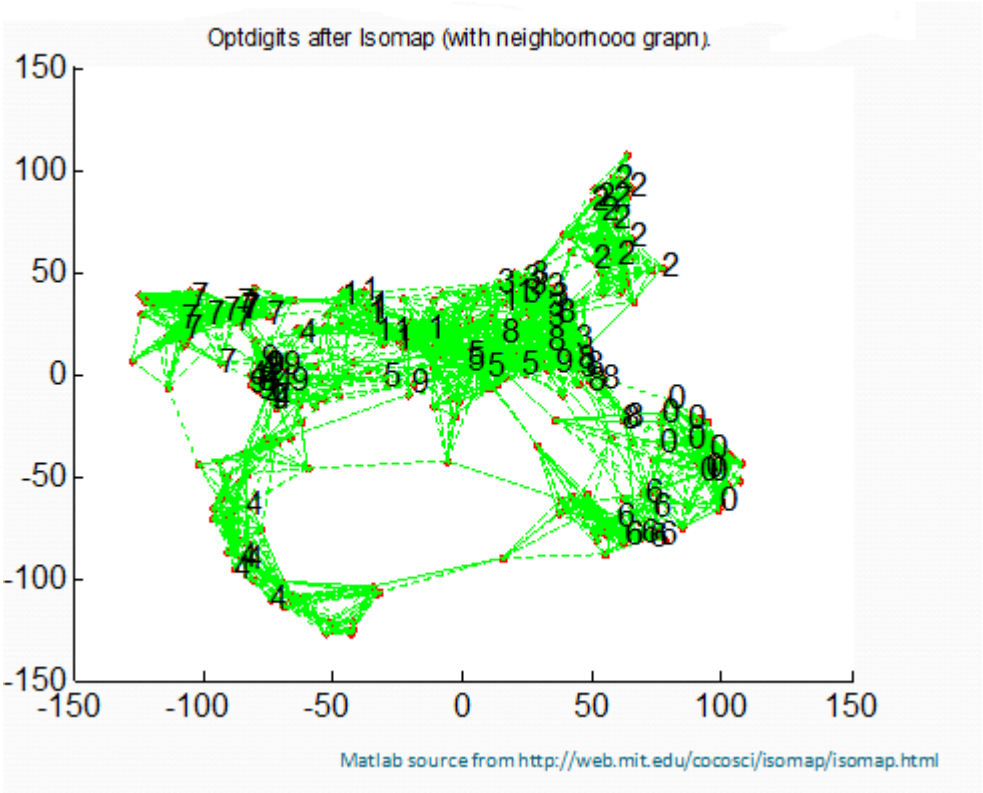
Isomap

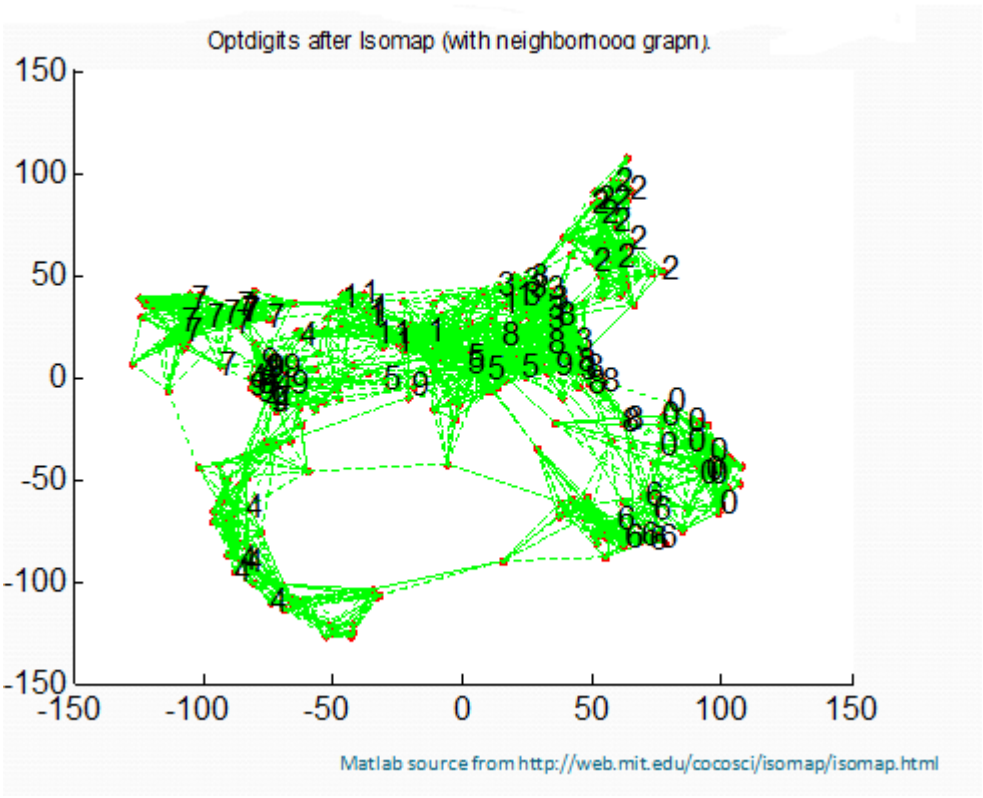
- Geodesic distance is the distance along the manifold that the data lies in, as opposed to the Euclidean distance in the input space



Isomap

- Instances r and s are connected in the graph if
  $||\boldsymbol{x}^r-\boldsymbol{x}^s||<\varepsilon$ or if $\boldsymbol{x}^s$ is one of the k neighbors of $\boldsymbol{x}^r$
  The edge length is $||\boldsymbol{x}^r-\boldsymbol{x}^s||$
- For two nodes r and s not connected, the distance is equal to the shortest path between them
- Once the NxN distance matrix is thus formed, use MDS to find a lower-dimensional mapping





Clustering

Semiparametric Density Estimation

- **Parametric:**
  Assume a single model for $p(x|C_i)$ (Chapter 4 and 5)

- **Semiparametric:**
  $p(x|C_i)$ is a mixture of densities

  Multiple possible explanations/prototypes:

  Different handwriting styles, accents in speech

- **Nonparametric:**
  No model; data speaks for itself

Classes vs. Clusters

- Supervised: $\mathcal{X} = \{ \boldsymbol{x}^t, \boldsymbol{r}^t \}_t$
- Classes C_i, $i = 1,\ldots,K$

$$p(\boldsymbol{x}) = \sum_{i=1}^{K} p(\boldsymbol{x} \mid C_i) P(C_i)$$

where $p(\boldsymbol{x} \mid C_i) \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$

- $\Phi = \{P(C_i), \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\}_{i=1}^{K}$

$$\hat{P}(C_i) = \frac{\sum_t r_i^t}{N} \quad \boldsymbol{m}_i = \frac{\sum_t r_i^t \boldsymbol{x}^t}{\sum_t r_i^t}$$

$$\mathbf{S}_i = \frac{\sum_t r_i^t (\boldsymbol{x}^t - \boldsymbol{m}_i)(\boldsymbol{x}^t - \boldsymbol{m}_i)^T}{\sum_t r_i^t}$$

- Unsupervised : $\mathcal{X} = \{ \boldsymbol{x}^t \}_t$
- Clusters $\mathcal{G}_i$, $i = 1,\ldots,k$

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

where $p(\boldsymbol{x} \mid \mathcal{G}_i) \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$

- $\Phi = \{P(\mathcal{G}_i), \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\}_{i=1}^{k}$

Labels, $\boldsymbol{r}^t_i$?

Mixture Densities

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

where $\mathcal{G}_i$: the mixture components/groups/clusters

$P(\mathcal{G}_i)$: mixture proportions (priors)

$p(\boldsymbol{x} \mid \mathcal{G}_i)$: component densities

Gaussian mixture where $p(\boldsymbol{x} \mid \mathcal{G}_i) \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$

parameters $\Phi = \{P(\mathcal{G}_i), \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\}_{i=1}^{k}$

unlabeled sample $\mathcal{X} = \{\boldsymbol{x}^t\}_{t=1}^{N}$ (unsupervised learning)

k-Means Clustering

- Find k reference vectors (prototypes/codebook vectors/ codewords) which best represent data
- Reference vectors, $\boldsymbol{m}_j$, j =1,...,k
- Use nearest (most similar) reference:

$$\left\|\boldsymbol{x}^t - \boldsymbol{m}_i\right\| = \min_j \left\|\boldsymbol{x}^t - \boldsymbol{m}_j\right\|$$

- Reconstruction error

$$E\left(\{\mathbf{m}_i\}_{i=1}^k \mid \mathcal{X}\right) = \sum_t \sum_i b_i^t \left\|\mathbf{x}^t - \mathbf{m}_i\right\|^2$$

$$b_i^t = \begin{cases} 1 & \text{if } \left\|\mathbf{x}^t - \mathbf{m}_i\right\| = \min_j \left\|\mathbf{x}^t - \mathbf{m}_j\right\| \\ 0 & \text{otherwise} \end{cases}$$

k-means Clustering

Initialize $\boldsymbol{m}_i, i = 1, \ldots, k$, for example, to k random $\boldsymbol{x}^t$

Repeat

For all $\boldsymbol{x}^t \in \mathcal{X}$

$$b_i^t \leftarrow \begin{cases} 1 & \text{if } \|\boldsymbol{x}^t - \boldsymbol{m}_i\| = \min_j \|\boldsymbol{x}^t - \boldsymbol{m}_j\| \\ 0 & \text{otherwise} \end{cases}$$

For all $\boldsymbol{m}_i, i = 1, \ldots, k$

$$\boldsymbol{m}_i \leftarrow \sum_t b_i^t \boldsymbol{x}^t / \sum_t b_i^t$$

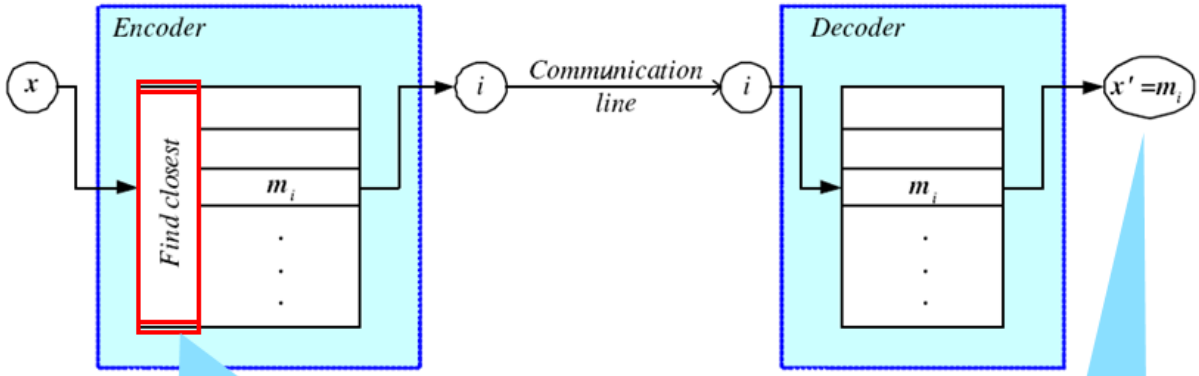
Until $\boldsymbol{m}_i$ converge

k-means Clustering

- One disadvantage:
  - A local search procedure
  - The final $\boldsymbol{m}_i$ highly depend on the initial $\boldsymbol{m}_i$

- The methods to initial $\boldsymbol{m}_i$
  - Randomly select k instances
  - Calculate the mean of all data and add small random vectors
  - Calculate the principal component partitioning the data into k groups, and then take the means of these groups

Encoding/Decoding



$$b_i^t = \begin{cases} 1 & \text{if } \left\|\boldsymbol{x}^t - \boldsymbol{m}_i\right\| = \min_j \left\|\boldsymbol{x}^t - \boldsymbol{m}_j\right\| \\ 0 & \text{otherwise} \end{cases}$$

Error is $\left\|\boldsymbol{x}^t - \boldsymbol{m}_i\right\|^2$

Expectation-Maximization Algorithm

- An **expectation-maximization (EM) algorithm** is used in statistics for finding maximum likelihood estimates of parameters in probabilistic models, where the model depends on unobserved latent variables.
- EM is an iterative method which alternates between performing an expectation (E) step,
  - which computes an expectation of the log likelihood with respect to the current estimate of the distribution for the latent variables,

  and a maximization (M) step,
  - which computes the parameters which maximize the expected log likelihood found on the E step.

E- and M-steps

- Iterate the two steps

1. E-step: Estimate the expectation of the log likelihood given $\mathcal{X}$ and current Φ
2. M-step: Find new Φ' given z, $\mathcal{X}$, and old Φ.

$$\text{E-step: } \mathcal{Q}\left(\Phi|\Phi^{l}\right) = E\left[\mathcal{L}_{c}\left(\Phi|\mathcal{X},\mathcal{Z}\right)|\mathcal{X},\Phi^{l}\right]$$

$$\text{M-step: } \Phi^{l+1} = \arg\max_{\Phi} \mathcal{Q}\left(\Phi|\Phi^{l}\right)$$

An increase in $\mathcal{Q}$ increases incomplete likelihood

(proven by Laird and Rubin (1977))

$$\mathcal{L}\left(\Phi^{l+1}|\mathcal{X}\right) \geq \mathcal{L}\left(\Phi^{l}|\mathcal{X}\right)$$

EM in Gaussian Mixtures

- Define indicator variables $\mathbf{z}^t = \{z^t_1, z^t_2, \ldots, z^t_k\}$

  $z^t_i = 1$ if $\mathbf{x}^t$ belongs to $\mathcal{G}_i$, 0 otherwise; assume $p(\mathbf{x}|\mathcal{G}_i) \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$

  - $\mathbf{z}$ is the multinomial distribution from k categories with prior probabilities π_i.

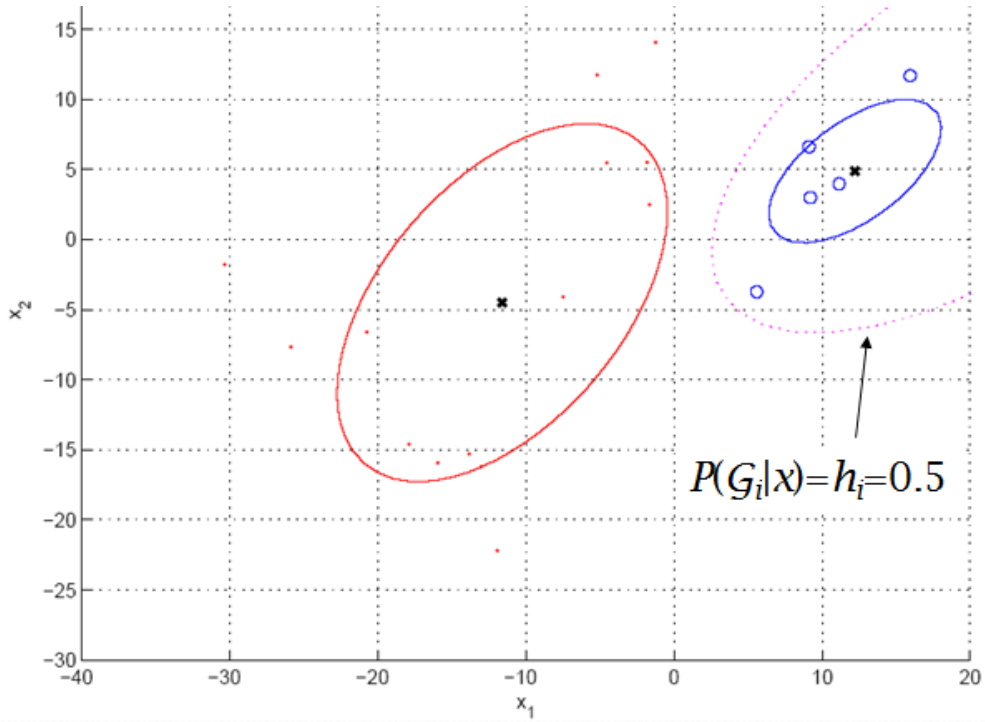
- E-step: (See p. 169 - p. 170)

$$\mathcal{Q}(\Phi|\Phi^l) = E\left[\mathcal{L}_c(\Phi|\mathcal{X},\mathcal{Z})|\mathcal{X},\Phi^l\right]$$

$$= \sum_t \sum_i E\left[z^t_i \,|\mathcal{X},\Phi^l\right]\left[\log \pi_i + \log p_i\left(\mathbf{x}^t|\Phi^l\right)\right]$$

where $$E\left[z^t_i \,|\mathcal{X},\Phi^l\right] = \frac{p\left(\mathbf{x}^t|\mathcal{G}_i,\Phi^l\right)P\left(\mathcal{G}_i\right)}{\sum_j p\left(\mathbf{x}^t|\mathcal{G}_j,\Phi^l\right)P\left(\mathcal{G}_j\right)}$$

$$= P\left(\mathcal{G}_i|\mathbf{x}^t,\Phi^l\right) \equiv h^t_i$$



Data points and the fitted Gaussians by EM, initialized by one k-means iteration of Figure 7.2. Unlike in k-means, EM allows estimating the covariance matrices. h_i indicates the contours of the estimated Gaussian densities.

UNIT-4

Introduction

In classification we define a set of discriminant functions $g_j(\mathbf{x})$, $j = 1, \ldots, K$, and then we choose C_i if $g_i(\mathbf{x}) = K \max_{j=1} g_j(\mathbf{x})$Previously, when we discussed methods for classification, we first estimated the prior probabilities, $\hat{P}(C_i)$, and the class likelihoods, $\hat{p}(\mathbf{x}|C_i)$, then used Bayes' rule to calculate the posterior densities. We then defined the discriminant functions in terms of the posterior, for example,
$g_i(\mathbf{x}) = \log \hat{P}(C_i|\mathbf{x})$

This is called *likelihood-based classification.*

We are now going to discuss *discriminant-based classification* where we assume a model directly for the discriminant, bypassing the estimation of likelihoods or posteriors. The discriminant-based approach, as inthe case of decision trees, makes an assumption on the form of the discriminant between the classes and makes no assumption about, or requires no knowledge of the densities—for example, whether they are Gaussian, or whether the inputs are correlated, and so forth.
We define a model for the discriminant $g_i(\mathbf{x}|\Phi_i)$ explicitly parameterized with the set of parameters Φ_i , as opposed to a likelihood-based scheme that has implicit parameters in defining the likelihood densities. This is a different inductive bias: instead of making an assumption on the form of the class densities, we make an assumption on the form of the boundaries separating classes.

Learning is the optimization of the model parameters $\boldsymbol{\Phi}_i$ to maximize the quality of the separation, that is, the classification accuracy on a given labeled training set. This differs from the likelihood-based methods that search for the parameters that maximize sample likelihoods, separately for each class.

In the discriminant-based approach, we do not care about correctly estimating the densities inside class regions; all we care about is the correct estimation of the *boundaries* between the class regions. Those who advocate the discriminant-based approach (e.g., Vapnik 1995) state that estimating the class densities is a harder problem than estimating the class discriminants, and it does not make sense to solve a hard problem to solve an easier problem. This is of course true only when the discriminant can be approximated by a simple function.
In this chapter, we concern ourselves with the simplest case where the discriminant functions are linear in $\mathbf{x}$:

$$g_i(\mathbf{x}|\mathbf{w}_i, w_{i0}) = \mathbf{w}_i^T\mathbf{x} + w_{i0} = \sum_{j=1}^{d} w_{ij}x_j + w_{i0}$$

The *linear discriminant* is used frequently mainly due to its simplicity: both the space and time complexities are $O(d)$. The linear model is easy to understand: the final output is a weighted sum of the input attributes x_j . The magnitude of the weight w_j shows the importance of x_j and its sign indicates if the effect is positive or negative. Most functions are additive in that the output is the sum of the effects of several attributes where the weights may be positive (enforcing) or negative (inhibiting).For example, when a customer applies for credit, financial institutions calculate the applicant's credit score that is generally written as a sum of the effects of various attributes; for example, yearly income has a positive effect (higher incomes increase the score.

Generalizing the Linear Model

When a linear model is not flexible enough, we can use the *quadratic discriminant* function and increase complexity, but this approach is $O(d^2)$ and we again have the bias/variance dilemma:

$$g_i(\mathbf{x}|\mathbf{W}_i, \mathbf{w}_i, w_{i0}) = \mathbf{x}^T\mathbf{W}_i\mathbf{x} + \mathbf{w}_i\mathbf{x} + w_{i0}$$

the quadratic model, though is more general, requires much larger training sets and may overfit on small samples. An equivalent way is to preprocess the input by adding *higher-order* also called *product terms*. For example, with two inputs $x1$ and $x2$, we can define new variables
$z1 = x1$, $z2 = x2$, $z3 = x21$, $z4 = x22$, $z5 = x1x2$ and take $\boldsymbol{z} = [z1, z2, z3, z4, z5]^T$ as the input. The linear function defined in the five-dimensional $\boldsymbol{z}$ space corresponds to a nonlinear function in
the two-dimensional $\boldsymbol{x}$ space. Instead of defining a nonlinear function (discriminant or regression) in the original space, what we do is to define a suitable nonlinear transformation to a new space where the function can be written in a linear form.
We write the discriminant as

$$g_i(\boldsymbol{x}) = \sum_{j=1}^{k} w_j \phi_{ij}(\boldsymbol{x})$$

where $\phi_{ij}(\boldsymbol{x})$ are *basis functions.*

Geometry of the Linear Discriminant

- **Two Classes**

Let us start with the simpler case of two classes. In such a case, one discriminant function is sufficient:

$$\begin{aligned} g(\boldsymbol{x}) &= g_1(\boldsymbol{x}) - g_2(\boldsymbol{x}) \\ &= (\boldsymbol{w}_1^T \boldsymbol{x} + w_{10}) - (\boldsymbol{w}_2^T \boldsymbol{x} + w_{20}) \\ &= (\boldsymbol{w}_1 - \boldsymbol{w}_2)^T \boldsymbol{x} + (w_{10} - w_{20}) \\ &= \boldsymbol{w}^T \boldsymbol{x} + w_0 \end{aligned}$$

and we

$$\text{choose} \begin{cases} C_1 & \text{if } g(\boldsymbol{x}) > 0 \\ C_2 & \text{otherwise} \end{cases}$$

This defines a hyperplane where $\boldsymbol{w}$ is the *weight vector* and $w0$ is the otherwise. The hyperplane divides the input space into two half-spaces:
the decision region R1 for C1 andR2 for C2. Any $\boldsymbol{x}$ inR1 is on the *positive* side of the hyperplane and any $\boldsymbol{x}$ in R2 is on its *negative* side. When $\boldsymbol{x}$ is $\mathbf{0}$, $g(\boldsymbol{x}) = w0$ and we see that if $w0 > 0$, the origin is on the positive side of the hyperplane, and if $w0 < 0$, the origin is on the negative side, and if $w0 = 0$, the hyperplane passes through the origin.
Take two points $\boldsymbol{x}1$ and $\boldsymbol{x}2$ both on the decision surface; that is, $g(\boldsymbol{x}1) = g(\boldsymbol{x}2) = 0$,

$$\begin{aligned} \boldsymbol{w}^T \boldsymbol{x}_1 + w_0 &= \boldsymbol{w}^T \boldsymbol{x}_2 + w_0 \\ \boldsymbol{w}^T (\boldsymbol{x}_1 - \boldsymbol{x}_2) &= 0 \end{aligned}$$

and we see that $\boldsymbol{w}$ is normal to any vector lying on the hyperplane. Let us rewrite $\boldsymbol{x}$ as (Duda, Hart, and Stork 2001)

$$\boldsymbol{x} = \boldsymbol{x}_p + r \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}$$

where $\boldsymbol{x}p$ is the normal projection of $\boldsymbol{x}$ onto the hyperplane and r gives us the distance from $\boldsymbol{x}$ to the hyperplane, negative if $\boldsymbol{x}$ is on the negative *threshold*. This latter name comes from the fact that the

decision rule can be rewritten as follows: choose C1 if $\boldsymbol{w}^T\boldsymbol{x} > -w0$, and choose C2 side, and positive if $\boldsymbol{x}$ is on the positive side. Calculating $g(\boldsymbol{x})$ and noting that $g(\boldsymbol{x}p) = 0$, we have

$$r = \frac{g(\boldsymbol{x})}{\|\boldsymbol{w}\|}$$

We see then that the distance to origin is

$$r_0 = \frac{w_0}{\|\boldsymbol{w}\|}$$

Thus w_0 determines the location of the hyperplane with respect to the origin, and $\boldsymbol{w}$ determines its orientation.

- **Multiple Classes**

When there are $K > 2$ classes, there are K discriminant functions. When they are linear, we have

$$g_i(\boldsymbol{x}|\boldsymbol{w}_i, w_{i0}) = \boldsymbol{w}_i^T\boldsymbol{x} + w_{i0}$$

We are going to talk about learning later on but for now, we assume that the parameters, $\boldsymbol{w}i$,$wi0$, are computed so as to have

$$g_i(\boldsymbol{x}|\boldsymbol{w}_i, w_{i0}) = \begin{cases} > 0 & \text{if } \boldsymbol{x} \in C_i \\ \leq 0 & \text{otherwise} \end{cases}$$

for all $\boldsymbol{x}$ in the training set. Using such discriminant functions corresponds to assuming that all classes are *linearly separable*; that is, for each class C_i, there exists a hyperplane H_i such that all $\boldsymbol{x} \in C_i$ lie on its positive side and all $\boldsymbol{x} \in C_j, j \neq i$ lie on its negative side.

During testing, given $\boldsymbol{x}$, ideally, we should have only one $g_j(\boldsymbol{x}), j = 1, \ldots, K$ greater than 0 and all others should be less than 0, but this is not always the case: The positive half-spaces of the hyperplanes may overlap, or, we may have a case where all $g_j(\boldsymbol{x}) < 0$. These may be taken as *reject* cases, but the usual approach is to assign $\boldsymbol{x}$ to the class having the highest discriminant:

Choose C_i if $g_i(\boldsymbol{x}) = \max_{j=1}^{K} g_j(\boldsymbol{x})$

- **Pairwise Separation**

If the classes are not linearly separable, one approach is to divide it into a set of linear problems. One possibility is *pairwise separation* of classes (Duda, Hart, and Stork 2001). It uses $K(K-1)/2$ linear discriminants, $g_{ij}(\boldsymbol{x})$, one for every pair of distinct classes:

$$g_{ij}(\boldsymbol{x}|\boldsymbol{w}_{ij}, w_{ij0}) = \boldsymbol{w}_{ij}^T\boldsymbol{x} + w_{ij0}$$

The parameters $\boldsymbol{w}_{ij}, j \neq i$ are computed during training so as to have

$$g_{ij}(\boldsymbol{x}) = \begin{cases} > 0 & \text{if } \boldsymbol{x} \in C_i \\ \leq 0 & \text{if } \boldsymbol{x} \in C_j \\ \text{don't care} & \text{otherwise} \end{cases} \quad i, j = 1, \ldots, K \text{ and } i \neq j$$

that is, if $\boldsymbol{x}^t \in C_k$ where $k \neq i, k \neq j$, then $\boldsymbol{x}^t$ is not used during training of $g_{ij}(\boldsymbol{x})$.

- **Parametric Discrimination Revisited**

if the class densities, $p(\boldsymbol{x}|C_i)$, are Gaussian and share a common covariance matrix, the discriminant function is linear

$$g_i(\boldsymbol{x}) = \boldsymbol{w}_i^T\boldsymbol{x} + w_{i0}$$

where the parameters can be analytically calculated as

$$\begin{aligned} \boldsymbol{w}_i &= \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_i \\ w_{i0} &= -\frac{1}{2}\boldsymbol{\mu}_i^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_i + \log P(C_i) \end{aligned}$$

Given a dataset, we first calculate the estimates for $\boldsymbol{\mu}_i$ and $\boldsymbol{\Sigma}$ and then plug the estimates, $\boldsymbol{m}_i$, S, in equation 10.12 and calculate the parameters of the linear discriminant.

Let us again see the special case where there are two classes. We define $y \equiv P(C_1|\boldsymbol{x})$ and $P(C_2|\boldsymbol{x}) = 1 - y$. Then in classification, we

$$\text{choose } C_1 \text{ if } \begin{cases} y > 0.5 \\ \frac{y}{1-y} > 1 \\ \log \frac{y}{1-y} > 0 \end{cases} \text{ and } C_2 \text{ otherwise}$$

$\log y/(1-y)$ is known as the *logit* transformation or *log odds* of y. In the case of two normal classes sharing a common covariance matrix, the log odds is linear:

$$\begin{aligned} \text{logit}(P(C_1|\boldsymbol{x})) &= \log \frac{P(C_1|\boldsymbol{x})}{1 - P(C_1|\boldsymbol{x})} = \log \frac{P(C_1|\boldsymbol{x})}{P(C_2|\boldsymbol{x})} \\ &= \log \frac{p(\boldsymbol{x}|C_1)}{p(\boldsymbol{x}|C_2)} + \log \frac{P(C_1)}{P(C_2)} \\ &= \log \frac{(2\pi)^{-d/2}|\boldsymbol{\Sigma}|^{-1/2}\exp[-(1/2)(\boldsymbol{x}-\boldsymbol{\mu}_1)^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_1)]}{(2\pi)^{-d/2}|\boldsymbol{\Sigma}|^{-1/2}\exp[-(1/2)(\boldsymbol{x}-\boldsymbol{\mu}_2)^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_2)]} + \log \frac{P(C_1)}{P(C_2)} \\ &= \boldsymbol{w}^T\boldsymbol{x} + w_0 \end{aligned}$$

where

$$\begin{aligned} \boldsymbol{w} &= \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \\ w_0 &= -\frac{1}{2}(\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2)^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) + \log \frac{P(C_1)}{P(C_2)} \end{aligned}$$

The inverse of logit

$$\log \frac{P(C_1|\boldsymbol{x})}{1 - P(C_1|\boldsymbol{x})} = \boldsymbol{w}^T\boldsymbol{x} + w_0$$

is the *logistic* function, also called the *sigmoid* function

$$P(C_1|\boldsymbol{x}) = \text{sigmoid}(\boldsymbol{w}^T\boldsymbol{x} + w_0) = \frac{1}{1 + \exp[-(\boldsymbol{w}^T\boldsymbol{x} + w_0)]}$$

During training, we estimate $\boldsymbol{m}_1, \boldsymbol{m}_2$, S and plug these estimates in equation 10.14 to calculate the discriminant parameters. During testing, given $\boldsymbol{x}$, we can either

1. calculate $g(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + w_0$ and choose C_1 if $g(\boldsymbol{x}) > 0$, or
2. calculate $y = \text{sigmoid}(\boldsymbol{w}^T\boldsymbol{x} + w_0)$ and choose C_1 if $y > 0.5$,

because $\text{sigmoid}(0) = 0.5$. In this latter case, sigmoid transforms the discriminant value to a posterior probability. This is valid when there are two classes and one discriminant; we see in section 10.7 how we can estimate posterior probabilities for $K > 2$.

Gradient Descent

In likelihood-based classification, the parameters were the sufficient statistics of $p(\boldsymbol{x}|C_i)$ and $P(C_i)$, and the method we used to estimate the parameters is maximum likelihood. In the discriminant-based approach,the parameters are those of the discriminants, and they are optimized to minimize the classification error on the training set. When $\boldsymbol{w}$ denotes the set of parameters and $E(\boldsymbol{w}|X)$ is the error with parameters $\boldsymbol{w}$ on the given training set X, we look for

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} E(\boldsymbol{w}|X)$$

In many cases, some of which we will see shortly, there is no analytical solution and we need to resort to iterative optimization methods, the most commonly employed being that of *gradient descent*. When $E(\boldsymbol{w})$ is differentiable function of a vector of variables.

Multilayer Perceptrons

Artificial neural network models, one of which is the *perceptron*, take their inspiration from the brain. There are cognitive scientists and neuroscientists whose aim is to understand the functioning of the brain (Posner 1989; Thagard 2005), and toward this aim, build models of the natural neural networks in the brain and make simulation studies.

- **Understanding the Brain**

According to Marr (1982), understanding an information processing system has three levels, called the levels of analysis:

1. *Computational theory* corresponds to the goal of computation and an abstract definition of the task.
2. *Representation and algorithm* is about how the input and the output are represented and about the specification of the algorithm for the transformation from the input to the output.
3. *Hardware implementation* is the actual physical realization of the system.

One example is sorting: The computational theory is to order a given set of elements. The representation may use integers, and the algorithm may be Quicksort. After compilation, the executable code for a particular processor sorting integers represented in binary is one hardware implementation. The idea is that for the same computational theory, there may be multiple representations and algorithms manipulating symbols in that representation. Similarly, for any given representation and algorithm, there may be multiple hardware implementations. We can use one of various sorting algorithms, and even the same algorithm can be compiled on computers with different processors and lead to different hardware implementations.

To take another example, ‘6’, ‘VI’, and ‘110’ are three different representations of the number six. There is a different algorithm for addition depending on the representation used. Digital computers use binary representation and have circuitry to add in this representation, which is one particular hardware implementation. Numbers are represented differently and addition corresponds to a different set of instructions on an abacus, which is another hardware implementation. When we add two numbers in our head, we use another representation and an algorithm suitable to that representation, which is implemented by the neurons. But all these different hardware implementations — for example, abacus, digital computer—implement the same computational theory, addition.

The classic example is the difference between natural and artificial flying machines. A sparrow flaps its wings; a commercial airplane does not flap its wings but uses jet engines. The sparrow and the airplane are two hardware implementations built for different purposes, satisfying different constraints. But they both implement the same theory, which is aerodynamics.

Neural Networks as a Paradigm for Parallel Processing

Since the 1980s, computer systems with thousands of processors have been commercially available. The software for such parallel architectures, however, has not advanced as quickly as hardware. The reason for this is that almost all our theory of computation up to that point was based on serial, one-processor machines. We are not able to use the parallel machines we have efficiently because we cannot program them efficiently.

There parallel processing are mainly two paradigms for *parallel processing*: In Single Instruction Multiple Data (SIMD) machines, all processors execute the same instruction but on different pieces of data. In Multiple Instruction Multiple Data (MIMD) machines, different processors may execute different instructions on different data. SIMD machines are easier to program because there is only one program to write. However, problems rarely have such a regular structure that they can be parallelized over a SIMD machine.

MIMD machines are more general, but it is not an easy task to write separate programs for all the individual processors; additional problems are related to synchronization, data transfer between processors, and so forth. SIMD machines are also easier to build, and machines with more processors can be constructed if they are SIMD. In MIMD machines, processors are more complex, and a more complex communication network should be constructed for the processors to exchange data arbitrarily.

Assume now that we can have machines where processors are a little bit more complex than SIMD processors but not as complex as MIMD processors. Assume we have simple processors with a small amount of local memory where some parameters can be stored. Each processor implements a fixed function and executes the same instructions as SIMD processors; but by loading different values into the local memory, they can be doing different things and the whole operation can be distributed over such processors. We will then have what we can call Neural Instruction Multiple Data (NIMD) machines, where each processor corresponds to a neuron, local parameters correspond to its synaptic weights, and the whole structure is a neural network. If the function implemented in each processor is simple and if the local memory is small, then many such processors can be fit on a single chip.

The problem now is to distribute a task over a network of such processors and to determine the local parameter values. This is where learning comes into play: We do not need to program such machines and determine the parameter values ourselves if such machines can learn from examples.

Thus, artificial neural networks are a way to make use of the parallel hardware we can build with current technology and—thanks to learning—they need not be programmed. Therefore, we also save ourselves the effort of programming them.

MLP as a Universal Approximator

We can represent any Boolean function as a disjunction of conjunctions,and such a Boolean expression can be implemented by a multilayer perceptron with one hidden layer. Each conjunction is implemented by one hidden unit and the disjunction by the output unit. For example,

$$x_1 \text{ XOR } x_2 = (x_1 \text{ AND } \sim x_2) \text{ OR } (\sim x_1 \text{ AND } x_2)$$

We have seen previously how to implement AND and OR using perceptrons. So two perceptrons can in parallel implement the two AND, and another perceptron on top can OR them together. We see that the first layer maps inputs from the *(x*1, *x*2*)* to the *(z*1, *z*2*)* space defined by the first-layer perceptrons. Note that both inputs, (0,0) and (1,1),are mapped to (0,0) in the *(z*1, *z*2*)* space, allowing linear separability in this second space.

Thus in the binary case, for every input combination where the output is 1, we define a hidden unit that checks for that particular conjunction of the input. The output layer then implements the disjunction. Note that this is just an existence proof, and such networks may not be practical

as up to 2^d hidden units may be necessary when there are *d* inputs. Such an architecture implements table lookup and does not generalize.

We can extend this to the case where inputs are continuous to show that similarly, any arbitrary function with continuous input and outputs can be approximated with a multilayer perceptron. The proof of *universal approximation* is easy with two hidden layers. For every input case or region, that region can be delimited by hyperplanes on all sides using hidden units on the first hidden layer. A hidden unit in the second layer then ANDs them together to bound the region.

Backpropagation Algorithm

Training a multilayer perceptron is the same as training a perceptron; the only difference is that now the output is a nonlinear function of the input thanks to the nonlinear basis function in the hidden units. Considering the hidden units as inputs, the second layer is a perceptron and we already know how to update the parameters, v_{ij}, in this case, given the inputs z_h. For the first-layer weights, w_{hj} , we use the chain rule to calculate the gradient:

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_h}\frac{\partial z_h}{\partial w_{hj}}$$

It is as if the error propagates from the output *y* back to the inputs and hence the name *backpropagation* was coined.

A) Nonlinear Regression

Let us first take the case of nonlinear regression (with a single output) calculated as

$$y^t = \sum_{h=1}^{H} v_h z_h^t + v_0$$

with z_h computed by equation

$$z_h = \text{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}) = \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}\right)\right]}, \quad h = 1, \ldots, H$$

The output y_i are perceptrons in the second layer taking the hidden units as their inputs

The error function over the whole sample in regression is

$$E(\mathbf{W}, \mathbf{v} | \mathcal{X}) = \frac{1}{2} \sum_t (r^t - y^t)^2$$

The second layer is a perceptron with hidden units as the inputs, and we use the least-squares rule to update the second-layer weights:

$$\Delta v_h = \eta \sum_t (r^t - y^t) z_h^t$$

The first layer are also perceptrons with the hidden units as the output units but in updating the first-layer weights, we cannot use the least squares rule directly as we do not have a desired output specified for the hidden units. This is where the chain rule comes into play. We write

$$\begin{aligned}
\Delta w_{hj} &= -\eta \frac{\partial E}{\partial w_{hj}} \\
&= -\eta \sum_t \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial z_h^t} \frac{\partial z_h^t}{\partial w_{hj}} \\
&= -\eta \sum_t \underbrace{-(r^t - y^t)}_{\partial E^t/\partial y^t} \underbrace{v_h}_{\partial y^t/\partial z_h^t} \underbrace{z_h^t (1 - z_h^t) x_j^t}_{\partial z_h^t/\partial w_{hj}} \\
&= \eta \sum_t (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t
\end{aligned}$$

The product of the first two terms $(r^t - y^t)v_h$ acts like the error term for hidden unit h. This error is *backpropagated* from the error to the hidden unit. $(r^t - y^t)$ is the error in the output, weighted by the "responsibility"of the hidden unit as given by its weight v_h. In the third term, $z_h(1 - z_h)$ is the derivative of the sigmoid and $x^t{}_j$ is the derivative of the weighted sum with respect to the weight w_{hj} . Note that the change in the first layer weight, Δw_{hj} , makes use of the second-layer weight, v_h. Therefore,we should calculate the changes in both layers and update the first-layer
weights, making use of the *old* value of the second-layer weights, then update the second-layer weights. Weights, w_{hj}, v_h are started from small random values initially, for example, in the range $[-0.01, 0.01]$, so as not to saturate the sigmoids. It is also a good idea to normalize the inputs so that they all have 0 mean and unit variance and have the same scale, since we use a single η parameter.
With the learning equations given here, for each pattern, we compute the direction in which each parameter needs be changed and the magnitude of this change.
In *batch learning*, we accumulate these changes over all patterns and make the change once after a complete pass over the whole training set is made, as shown in the previous update equations.

B) Two-Class Discrimination

When there are two classes, one output unit suffices:

$$y^t = \text{sigmoid}\left(\sum_{h=1}^{H} v_h z_h^t + v_0\right)$$

which approximates $P(C_1|\mathbf{x}^t)$ and $\hat{} P(C_2|\mathbf{x}^t) \equiv 1 - y^t$.

The error function in this case is

$$E(\mathbf{W}, \mathbf{v}|\mathcal{X}) = -\sum_t r^t \log y^t + (1 - r^t) \log(1 - y^t)$$

The update equations implementing gradient descent are

$$\begin{aligned} \Delta v_h &= \eta \sum_t (r^t - y^t) z_h^t \\ \Delta w_{hj} &= \eta \sum_t (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t \end{aligned}$$

As in the simple perceptron, the update equations for regression and classification are identical (which does not mean that the values are).

C)Multi-class Discrimination

In a $(K > 2)$-class classification problem, there are K outputs

$$o_i^t = \sum_{h=1}^{H} v_{ih} z_h^t + v_{i0}$$

and we use softmax to indicate the dependency between classes; namely, they are mutually exclusive and exhaustive:

$$y_i^t = \frac{\exp o_i^t}{\sum_k \exp o_k^t}$$

Initialize all v_{ih} and w_{hj} to rand$(-0.01, 0.01)$
Repeat
  For all $(\mathbf{x}^t, r^t) \in \mathcal{X}$ in random order
    For $h = 1, \ldots, H$
      $z_h \leftarrow \text{sigmoid}(\mathbf{w}_h^T \mathbf{x}^t)$
    For $i = 1, \ldots, K$
      $y_i = \mathbf{v}_i^T \mathbf{z}$
    For $i = 1, \ldots, K$
      $\Delta \mathbf{v}_i = \eta (r_i^t - y_i^t) \mathbf{z}$
    For $h = 1, \ldots, H$
      $\Delta \mathbf{w}_h = \eta (\sum_i (r_i^t - y_i^t) v_{ih}) z_h (1 - z_h) \mathbf{x}^t$
    For $i = 1, \ldots, K$
      $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta \mathbf{v}_i$
    For $h = 1, \ldots, H$
      $\mathbf{w}_h \leftarrow \mathbf{w}_h + \Delta \mathbf{w}_h$
Until convergence

Figure Backpropagation algorithm for training a multilayer perceptron for regression with K outputs. This code can easily be adapted for two-class classification (by setting a single sigmoid output) and to $K > 2$ classification (by using softmax outputs).

where y_i approximates $P(C_i|\mathbf{x}^t)$. The error function is

$$E(\mathbf{W},\mathbf{V}|\mathcal{X}) = -\sum_t \sum_i r_i^t \log y_i^t$$

and we get the update equations using gradient descent:

$$\Delta v_{ih} = \eta \sum_t (r_i^t - y_i^t) z_h^t$$

$$\Delta w_{hj} = \eta \sum_t \left[\sum_i (r_i^t - y_i^t) v_{ih} \right] z_h^t (1 - z_h^t) x_j^t$$

Richard and Lippmann (1991) have shown that given a network of enough complexity and sufficient training data, a suitably trained multilayer perceptron estimates posterior probabilities.

Training Procedures

- **Improving Convergence**

Gradient descent has various advantages. It is simple. It is local; namely, the change in a weight uses only the values of the presynaptic and postsynaptic units and the error (suitably backpropagated). When online training is used, it does not need to store the training set and can adapt as the task to be learned changes. Because of these reasons, it can be (and is) implemented in hardware. But by itself, gradient descent converges slowly. When learning time is important, one can use more sophisticated optimization methods (Battiti 1992). Bishop (1995) discusses in detail the application of conjugate gradient and second-order methods to the training of multilayer perceptrons. However, there are two frequently used simple techniques that improve the performance of the gradient descent considerably, making gradient-based methods feasible in real applications.

Let us say *wi* is any weight in a multilayer perceptron in any layer, including the biases. At each parameter update, successive $\Delta wt\ i$ values may be so different that large oscillations may occur and slow convergence. *t* is the time index that is the epoch number in batch learning and the iteration number in online learning. The idea is to take a running average by incorporating the previous update in the current change as if there is a momentum due to previous updates:

$$\Delta w_i^t = -\eta \frac{\partial E^t}{\partial w_i} + \alpha \Delta w_i^{t-1}$$

α is generally taken between 0.5 and 1.0. This approach is especially useful when online learning is used, where as a result we get an effect of averaging and smooth the trajectory during convergence. The disadvantage is that the past Δw_i^{t-1} values should be stored in extra memory.

Adaptive Learning Rate

In gradient descent, the learning factor η determines the magnitude of change to be made in the parameter. It is generally taken between 0.0 and 1.0, mostly less than or equal to 0.2. It can be made adaptive for faster convergence, where it is kept large when learning takes place and is decreased when learning slows down:

$$\Delta\eta = \begin{cases} +a & \text{if } E^{t+\tau} < E^t \\ -b\eta & \text{otherwise} \end{cases}$$

Thus we increase η by a constant amount if the error on the training set decreases and decrease it geometrically if it increases. Because E may oscillate from one epoch to another, it is a better idea to take the average of the past few epochs as E^t.

- **Overtraining**

A multilayer perceptron with d inputs, H hidden units, and K outputs has $H(d+1)$ weights in the first layer and $K(H+1)$ weights in the second layer. Both the space and time complexity of an MLP is $\mathcal{O}(H \cdot (K+d))$. When e denotes the number of training epochs, training time complexity is $\mathcal{O}(e \cdot H \cdot (K+d))$.

In an application, d and K are predefined and H is the parameter that we play with to tune the complexity of the model. We know from previous chapters that an overcomplex model memorizes the noise in the training set and does not generalize to the validation set. For example, we have previously seen this phenomenon in the case of polynomial regression where we noticed that in the presence of noise or small samples, increasing the polynomial order leads to worse generalization. Similarly in an MLP, when the number of hidden units is large, the generalization accuracy deteriorates, and the bias/variance dilemma also holds for the MLP, as it does for any statistical estimator (Geman, Bienenstock, and Doursat 1992).

- **Structuring the Network**

In some applications, we may believe that the input has a local structure. For example, in vision we know that nearby pixels are correlated and there are local features like edges and corners; any object, for example, a handwritten digit, may be defined as a combination of such primitives.

Similarly, in speech, locality is in time and inputs close in time can be grouped as speech primitives. By combining these primitives, longer utterances, for example, speech phonemes, may be defined. In such a case when designing the MLP, hidden units are not connected to all input units because not all inputs are correlated. Instead, we define hidden units that define a window over the input space and are connected to

only a small local subset of the inputs. This decreases the number of connections and therefore the number of free parameters (Le Cun et al. 1989).

Tuning the Network Size

Previously we saw that when the network is too large and has too many free parameters, generalization may not be well. To find the optimal network size, the most common approach is to try many different architectures,train them all on the training set, and choose the one that generalizes best to the validation set. Another approach is to incorporate this *structural adaptation* into the learning algorithm. There are two ways this can be done:

1. In the *destructive* approach, we start with a large network and gradually remove units and/or connections that are not necessary.
2. In the *constructive* approach, we start with a small network and gradually add units and/or connections to improve performance.

One destructive method is *weight decay* where the idea is to remove unnecessary connections. Ideally to be able to determine whether a unit or connection is necessary, we need to train once with and once without and check the difference in error on a separate validation set. This is costly since it should be done for all combinations of such units/connections.

Bayesian View of Learning

The Bayesian approach in training neural networks considers the parameters, namely, connection weights, w_i, as random variables drawn from a prior distribution $p(w_i)$ and computes the posterior probability given the data

$$p(\mathbf{w}|\mathcal{X}) = \frac{p(\mathcal{X}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{X})}$$

where $\mathbf{w}$ is the vector of all weights of the network. The MAP estimate $\hat{\mathbf{w}}$ is the mode of the posterior

$$\hat{\mathbf{w}}_{MAP} = \arg\max_{\mathbf{w}} \log p(\mathbf{w}|\mathcal{X})$$

Taking the log of equation

$$\log p(\mathbf{w}|\mathcal{X}) = \log p(\mathcal{X}|\mathbf{w}) + \log p(\mathbf{w}) + C$$

The first term on the right is the log likelihood, and the second is the log of the prior. If the weights are independent and the prior is taken as Gaussian, $\mathcal{N}(0, 1/2\lambda)$

$$p(\mathbf{w}) = \prod_i p(w_i) \text{ where } p(w_i) = c \cdot \exp\left[-\frac{w_i^2}{2(1/2\lambda)}\right]$$

the MAP estimate minimizes the augmented error function

$$E' = E + \lambda\|\mathbf{w}\|^2$$

where E is the usual classification or regression error (negative log likelihood). This augmented error is exactly the error function we used in weight decay. Using a large λ assumes small variability in parameters, puts a larger force on them to be close to 0, and takes the prior more into account than the data; if λ is small, then the allowed variability of the parameters is larger. This approach of removing unnecessary parameters is known as *ridge regression* in statistics.

Dimensionality Reduction

IN AN APPLICATION, whether it is classification or regression, observation data that we believe contain information are taken as inputs and fed to the system for decision making. Ideally, we should not need feature selection or extraction as a separate process; the classifier (or regressor) should be able to use whichever features are necessary, discarding the irrelevant. However, there are several reasons why we are interested in reducing dimensionality as a separate preprocessing step:

- In most learning algorithms, the complexity depends on the number of input dimensions, d, as well as on the size of the data sample, N, and for reduced memory and computation, we are interested in reducing the dimensionality of the problem. Decreasing d also decreases the complexity of the inference algorithm during testing.
- When an input is decided to be unnecessary, we save the cost of extracting it.
- Simpler models are more robust on small datasets. Simpler models

have less variance, that is, they vary less depending on the particulars of a sample, including noise, outliers, and so forth.

- When data can be explained with fewer features, we get a better idea about the process that underlies the data and this allows knowledge extraction.
- When data can be represented in a few dimensions without loss of information, it can be plotted and analyzed visually for structure and outliers.

There are two main methods for reducing dimensionality: feature selection and feature extraction. In *feature selection*, we are interested in finding k of the d dimensions that give us the most information and we discard the other $(d - k)$ dimensions. We are going to discuss *subset selection* as a feature selection method.

In *feature extraction*, we are interested in finding a new set of k dimensions that are combinations of the original d dimensions. These methods may be supervised or unsupervised depending on whether or not they use the output information. The best known and most widely used feature extraction methods are *Principal Components Analysis* (PCA) and *Linear Discriminant Analysis* (LDA), which are both linear projection methods, unsupervised and supervised respectively. PCA bears much similarity to two other unsupervised linear projection methods, which we also discuss—namely, *Factor Analysis* (FA) and *Multidimensional Scaling* (MDS). As examples of *nonlinear* dimensionality reduction, we are going to see *Isometric feature mapping* (Isomap) and *Locally Linear Embedding* (LLE).

Deep learning

Deep learning is a class of machine learning algorithms that:

- use a cascade of multiple layers of nonlinear processing units for feature extraction and transformation. Each successive layer uses the output from the previous layer as input.
- learn in supervised (e.g., classification) and/or unsupervised (e.g., pattern analysis) manners.
- learn multiple levels of representations that correspond to different levels of abstraction; the levels form a hierarchy of concepts.

In deep learning, each level learns to transform its input data into a slightly more abstract and composite representation. In an image recognition application, the raw input may be a matrix of pixels; the first representational layer may abstract the pixels and encode edges; the second layer may compose and encode arrangements of edges; the third layer may encode a nose and eyes; and the fourth layer may recognize that the image contains a face. Importantly, a deep learning process can learn which features to optimally place in which level *on its own*.

The "deep" in "deep learning" refers to the number of layers through which the data is transformed. More precisely, deep learning systems have a substantial *credit assignment path* (CAP) depth. The CAP is the chain of transformations from input to output. CAPs describe potentially causal connections between input and output. For a feedforward neural network, the depth of the CAPs is that of the network and is the

number of hidden layers plus one (as the output layer is also parameterized). For recurrent neural networks, in which a signal may propagate through a layer more than once, the CAP depth is potentially unlimited. No universally agreed upon threshold of depth divides shallow learning from deep learning, but most researchers agree that deep learning involves CAP depth > 2. CAP of depth 2 has been shown to be a universal approximator in the sense that it can emulate any function. Beyond that more layers do not add to the function approximator ability of the network. The extra layers help in learning features.

Deep learning architectures are often constructed with a greedy layer-by-layer method. Deep learning helps to disentangle these abstractions and pick out which features improve performance.

For supervised learning tasks, deep learning methods obviate feature engineering, by translating the data into compact intermediate representations akin to principal components, and derive layered structures that remove redundancy in representation.

Deep learning algorithms can be applied to unsupervised learning tasks. This is an important benefit because unlabeled data are more abundant than labeled data. Examples of deep structures that can be trained in an unsupervised manner are neural history compressors and deep belief networks.

UNIT-5

Kernel Machines

Support vector machine (SVM) and later generalized under the name *kernel machine*, has been popular in recent years for a number of reasons:

1. It is a discriminant-based method and uses Vapnik's principle to never solve a more complex problem as a first step before the actual problem (Vapnik 1995). For example, in classification, when the task is to learn the discriminant, it is not necessary to estimate where the class densities $p(\boldsymbol{x}|C_i)$ or the exact posterior probability values $P(C_i|\boldsymbol{x})$; we
only need to estimate where the class boundaries lie, that is, $\boldsymbol{x}$ where $P(C_i|\boldsymbol{x}) = P(C_j|\boldsymbol{x})$. Similarly, for outlier detection, we do not need to estimate the full density $p(\boldsymbol{x})$; we only need to find the boundary separating those $\boldsymbol{x}$ that have low $p(\boldsymbol{x})$, that is, $\boldsymbol{x}$ where $p(\boldsymbol{x}) < \theta$, for some threshold $\theta \in (0, 1)$.

2. After training, the parameter of the linear model, the weight vector, can be written down in terms of a subset of the training set, which are the so-called *support vectors*. In classification, these are the cases that are close to the boundary and as such, knowing them allows knowledge extraction: those are the uncertain or erroneous cases that lie in the vicinity of the boundary between two classes. Their number gives us an estimate of the generalization error, and, as we see below, being able to write the model parameter in terms of a set of instances allows kernelization.

3. As we will see shortly, the output is written as a sum of the influences of support vectors and these are given by *kernel functions* that are application-specific measures of similarity between data instances.

Previously, we talked about nonlinear basis functions allowing us to map the input to another space where a linear (smooth) solution is possible; the kernel function uses the same idea.

4. Typically in most learning algorithms, data points are represented as vectors, and either dot product (as in the multilayer perceptrons) or Euclidean distance (as in radial basis function networks) is used. A kernel function allows us to go beyond that. For example, $G1$ and $G2$ may be two graphs and $K(G1,G2)$ may correspond to the number of shared paths, which we can calculate without needing to represent $G1$ or $G2$ explicitly as vectors.

5. Kernel-based algorithms are formulated as convex optimization problems, and there is a single optimum that we can solve for analytically. Therefore we are no longer bothered with heuristics for learning rates, initializations, checking for convergence, and such. Of course, this does not mean that we do not have any hyperparameters for model selection; we do—any method needs them, to match the algorithm to the data at hand.

2.Optimal Separating Hyperplane

Let us start again with two classes and use labels $-1/+1$ for the two classes. The sample is $X = \{\boldsymbol{x}^t, r^t\}$ where $r^t = +1$ if $\boldsymbol{x}^t \in C1$ and $r^t = -1$ if $\boldsymbol{x}^t \in C2$. We would like to find $\boldsymbol{w}$ and w_0 such that $\boldsymbol{w}^T\boldsymbol{x}^t + w_0 \geq +1$ for $r^t = +1$
$\boldsymbol{w}^T\boldsymbol{x}^t + w_0 \leq -1$ for $r^t = -1$
which can be rewritten as
$r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) \geq +1$
Note that we do not simply require

rt (***w****T****x****t* + *w*0) ≥ 0

Not only do we want the instances to be on the right side of the hyperplane, but we also want them some distance away, for better generalization.

The distance from the hyperplane to the instances closest to it on either side is called the *margin*, which we want to maximize for best generalization. Very early on, in section 2.1, we talked about the concept of the margin when we were talking about fitting a rectangle, and we said that it is better to take a rectangle halfway between *S* and *G*, to get a breathing space. This is so that in case noise shifts a test instance slightly, it will still be on the right side of the boundary.

Similarly, now that we are using the hypothesis class of lines, the *optimal separating hyperplane* is the one that maximizes the margin. We remember from section 10.3 that the distance of ***x****t* to the discriminant is

$$\frac{|\boldsymbol{w}^T\boldsymbol{x}^t + w_0|}{\|\boldsymbol{w}\|}$$

which, when $r^t \in \{-1, +1\}$, can be written as

$$\frac{r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0)}{\|\boldsymbol{w}\|}$$

and we would like this to be at least some value ρ:

$$\frac{r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0)}{\|\boldsymbol{w}\|} \geq \rho, \forall t$$

We would like to maximize ρ but there are an infinite number of solutions

that we can get by scaling ***w*** and for a unique solution, we fix $\rho\|\boldsymbol{w}\| = 1$ and thus, to maximize the margin, we minimize $\|\boldsymbol{w}\|$.

The task can therefore be defined as to

$$\min \frac{1}{2}\|\boldsymbol{w}\|^2 \text{ subject to } r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) \geq +1, \forall t$$

This is a standard quadratic optimization problem, whose complexity depends on *d*, and it can be solved directly to find ***w*** and *w*0. Then, on both sides of the hyperplane, there will be instances that are 1/||***w***|| away from the hyperplane and the total margin will be 2/||w||.

In finding the optimal hyperplane, we can convert the optimization problem to a form whose complexity depends on *N*, the number of training instances, and not on *d*. Another advantage of this new formulation is that it will allow us to rewrite the basis functions in terms of kernel

Functions. To get the new formulation,

$$\begin{aligned} L_p &= \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{t=1}^{N} \alpha^t [r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) - 1] \\ &= \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_t \alpha^t r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) + \sum_t \alpha^t \end{aligned}$$

This should be minimized with respect to ***w***,*w*0 and maximized with respect to $\alpha t \geq 0$. The saddle point gives the solution. This is a convex quadratic optimization problembecause themain term

is convex and the linear constraints are also convex. Therefore, we can equivalently solve the dual problem, making use of the Karush-Kuhn- Tucker conditions. The dual is to *maximize Lp* with respect to αt, subject to the constraints that the gradient of *Lp* with respect to $\boldsymbol{w}$ and $w0$ are0 which we maximize with respect to αt only, subject to the constraints

and also that $\alpha^t \geq 0$:

$$\frac{\partial L_p}{\partial \boldsymbol{w}} = 0 \quad \Rightarrow \quad \boldsymbol{w} = \sum_t \alpha^t r^t \boldsymbol{x}^t$$

$$\frac{\partial L_p}{\partial w_0} = 0 \quad \Rightarrow \quad \sum_t \alpha^t r^t = 0$$

Plugging these into equation 13.4, we get the dual

$$\begin{aligned} L_d &= \frac{1}{2}(\boldsymbol{w}^T\boldsymbol{w}) - \boldsymbol{w}^T \sum_t \alpha^t r^t \boldsymbol{x}^t - w_0 \sum_t \alpha^t r^t + \sum_t \alpha^t \\ &= -\frac{1}{2}(\boldsymbol{w}^T\boldsymbol{w}) + \sum_t \alpha^t \\ &= -\frac{1}{2}\sum_t \sum_s \alpha^t \alpha^s r^t r^s (\boldsymbol{x}^t)^T \boldsymbol{x}^s + \sum_t \alpha^t \end{aligned}$$

$$\sum_t \alpha^t r^t = 0, \text{ and } \alpha^t \geq 0, \forall t$$

This can be solved using quadratic optimization methods. The size of the dual depends on *N*, sample size, and not on *d*, the input dimensionality. The upper bound for time complexity is O*(N*3*)*, and the upper bound for space complexity is O*(N*2*)*.

Once we solve for αt , we see that though there are *N* of them, most vanish with $\alpha t = 0$ and only a small percentage have $\alpha t > 0$. The set of $\boldsymbol{x}t$ whose $\alpha t > 0$ are the *support vectors*, and as we see in equation 13.5, $\boldsymbol{w}$ is written as the weighted sum of these training instances that are selected as the support vectors. These are the $\boldsymbol{x}t$ that satisfy

$$r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) = 1$$

For numerical stability, it is advised that this be done for all support vectors and an average be taken. The discriminant thus found is called the *support vector machine* (SVM),

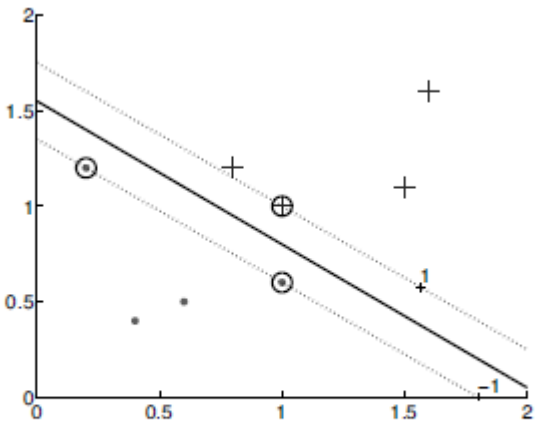


Figure 13.1 For a two-class problem where the instances of the classes are shown by plus signs and dots, the thick line is the boundary and the dashed lines define the margins on either side. Circled instances are the support vectors.

During testing, we do not enforce a margin. We calculate $g(\boldsymbol{x}) = \boldsymbol{w}T\ \boldsymbol{x}+$

*w*0, and choose according to the sign of $g(\boldsymbol{x})$: Choose C1 if $g(\boldsymbol{x}) > 0$ and C2 otherwise.

3.The Nonseparable Case: Soft Margin Hyperplane

If the data is not linearly separable, the algorithm we discussed earlier will not work. In such a case, if the two classes are not linearly separable such that there is no hyperplane to separate them, we look for the one that incurs slack variables the least error. We define *slack variables*, $\xi t \geq 0$, which store the deviation from the margin. There are two types of deviation: An instance may lie on the wrong side of the hyperplane and be misclassified. Or, it may be on the right side but may lie in the margin, namely, not sufficiently away from the hyperplane. Relaxing

$$r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) \geq 1 - \xi^t$$

If $\xi t = 0$, there is no problem with $\boldsymbol{x}t$. If $0 < \xi t < 1$, $\boldsymbol{x}t$ is correctly
classified but in the margin. If $\xi t \geq 1$, $\boldsymbol{x}t$ is misclassified .
The number of misclassifications is $\#\{\xi t > 1\}$, and the number of non separable points is $\#\{\xi t > 0\}$. We define *soft error* as

$$\sum_t \xi^t$$

and add this as a penalty term:

$$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t \xi^t$$

subject to the constraint of equation 13.9. C is the penalty factor as in any regularization scheme trading off complexity, as measured by the $L2$ norm of the weight vector (similar to weight decay in multilayer perceptrons; and data misfit, as measured by the number of nonseparable points. Note that we are penalizing not only the misclassified points but also the ones in the margin for better generalization, though these latter would be correctly classified during testing.
Finally we will get

$$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t \xi^t - \sum_t \alpha^t[r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) - 1 + \xi^t] - \sum_t \mu^t\xi^t$$

where μt are the new Lagrange parameters to guarantee the positivity of ξt. When we take the derivatives with respect to the parameters and set them to 0, we get:

$$\frac{\partial L_p}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_t \alpha^t r^t \boldsymbol{x}^t = 0 \Rightarrow \boldsymbol{w} = \sum_t \alpha^t r^t \boldsymbol{x}^t$$

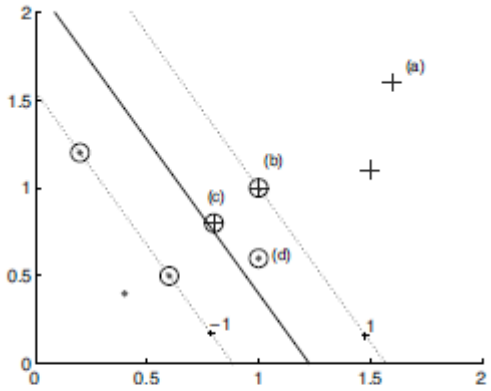


Figure 13.2 In classifying an instance, there are four possible cases: In (a), the instance is on the correct side and far away from the margin; $r^t g(\boldsymbol{x}^t) > 1$, $\xi^t = 0$. In (b), $\xi^t = 0$; it is on the right side and on the margin. In (c), $\xi^t = 1 - g(\boldsymbol{x}^t)$, $0 < \xi < 1$; it is on the right side but is in the margin and not sufficiently away. In (d), $\xi^t = 1 + g(\boldsymbol{x}^t) > 1$; it is on the wrong side—this is a misclassification. All cases except (a) are support vectors. In terms of the dual variable, in (a), $\alpha^t = 0$; in (b), $\alpha^t < C$; in (c) and (d), $\alpha^t = C$.

$$\frac{\partial L_p}{\partial w_0} = \sum_t \alpha^t r^t = 0$$

$$\frac{\partial L_p}{\partial \xi^t} = C - \alpha^t - \mu^t = 0$$

Since $\mu^t \geq 0$, this last implies that $0 \leq \alpha^t \leq C$. Plugging these into equation 13.11, we get the dual that we maximize with respect to α^t:

$$L_d = \sum_t \alpha^t - \frac{1}{2}\sum_t \sum_s \alpha^t \alpha^s r^t r^s (\mathbf{x}^t)^T \mathbf{x}^s$$

subject to

$$\sum_t \alpha^t r^t = 0 \text{ and } 0 \leq \alpha^t \leq C, \forall t$$

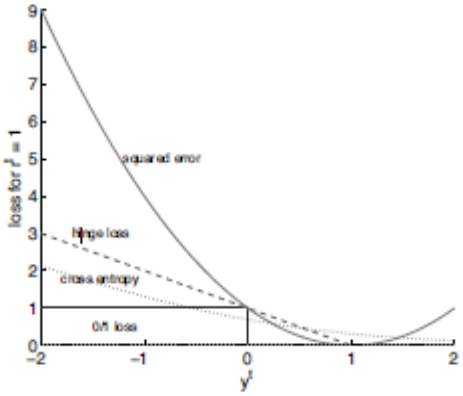
Solving this, we see that as in the separable case, instances that lie on the correct side of the boundary with sufficient margin vanish with their $\alpha t = 0$ (see figure 13.2). The support vectors have their $\alpha t > 0$ and they define $\mathbf{w}$, Of these, those whose $\alpha t < C$ are the ones that are on the margin, and we can use them to calculate $w0$; they have $\xi t = 0$ and satisfy $rt\ (\mathbf{w}T\mathbf{x}t + w0) = 1$. Again, it is better to take an average over these $w0$ estimates. Those instances that are in the margin or misclassified have their $\alpha t = C$. The nonseparable instances that we store as support vectors are the instances that we would have trouble correctly classifying if they were not in the training set; they would either be misclassified or classified correctly but not with enough confidence. We can say that the number
of support vectors is an upper-bound estimate for the expected number of errors. And, actually, Vapnik (1995) has shown that the expected test error rate is

$$E_N[P(error)] \leq \frac{E_N[\#\text{ of support vectors}]}{N}$$

where $EN[\cdot]$ denotes expectation over training sets of size N. The nice implication of this is that it shows that the error rate depends on the number of support vectors and not on the input dimensionality.it

implies that we define error if the instance is on the wrong side or if the margin is less than 1. This is called the *hinge loss*. If $y^t = \mathbf{w}^T\mathbf{x}^t + w_0$ is the output and r^t is the desired output, hinge loss is defined as

$$L_{hinge}(y^t, r^t) = \begin{cases} 0 & \text{if } y^t r^t \geq 1 \\ 1 - y^t r^t & \text{otherwise} \end{cases}$$



***4.v*-SVM**

There is another, equivalent formulation of the soft margin hyperplane that uses a parameter $\nu \in [0, 1]$ instead of C. The objective function is

$$\min \frac{1}{2}\|\mathbf{w}\|^2 - \nu\rho + \frac{1}{N}\sum_t \xi^t$$

subject to

$$r^t(\mathbf{w}^T\mathbf{x}^t + w_0) \geq \rho - \xi^t,\ \xi^t \geq 0,\ \rho \geq 0$$

ρ is a new parameter that is a variable of the optimization problem and scales the margin: the margin is now $2\rho/\|\mathbf{w}\|$. ν has been shown to be a lower bound on the fraction of support vectors and an upper bound on the fraction of instances having margin errors ($\sum_t \#\{\xi^t > 0\}$). The dual is

$$L_d = -\frac{1}{2}\sum_t\sum_s \alpha^t\alpha^s r^t r^s (\mathbf{x}^t)^T\mathbf{x}^s$$

subject to

$$\sum_t \alpha^t r^t = 0,\ 0 \leq \alpha^t \leq \frac{1}{N},\ \sum_t \alpha^t \geq \nu$$

5. **Kernel Trick**

Instead of trying to fit a nonlinear model, we can map the problem to a new space by doing a nonlinear transformation using suitably chosen basis functions and then use a linear model in this new space. The linear model in the new space corresponds to a nonlinear model in the original space. This approach can be used in both classification and regression problems, and in the special case of classification, it can be used with any scheme. In the particular case of support vector machines, it leads to certain simplifications that we now discuss.

Let us say we have the new dimensions calculated through the basis functions

$\mathbf{z} = \boldsymbol{\varphi}(\mathbf{x})$ where $z_j = \varphi_j(\mathbf{x}),\ j = 1, \ldots, k$

mapping from the d-dimensional $\mathbf{x}$ space to the k-dimensional $\mathbf{z}$ space

where we write the discriminant as

$$\begin{aligned} g(\mathbf{z}) &= \mathbf{w}^T\mathbf{z} \\ g(\mathbf{x}) &= \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}) \\ &= \sum_{j=1}^{k} w_j\phi_j(\mathbf{x}) \end{aligned}$$

where we do not use a separate w_0; we assume that $z_1 = \varphi_1(\mathbf{x}) \equiv 1$. Generally, k is much larger than d and k may also be larger than N, and there lies the advantage of using the dual form whose complexity depends on N, whereas if we used the primal it would depend on k. We also use the more general case of the soft margin hyperplane here because we have no guarantee that the problem is linearly separable in this new space.

The problem is the same

$$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t \xi^t$$

except that now the constraints are defined in the new space

$$r^t \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}^t) \geq 1 - \xi^t$$

The Lagrangian is

$$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_t \xi^t - \sum_t \alpha^t \left[r^t \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}^t) - 1 + \xi^t\right] - \sum_t \mu^t \xi^t$$

When we take the derivatives with respect to the parameters and set them to 0, we get

$$\frac{\partial L_p}{\partial \boldsymbol{w}} = \boldsymbol{w} = \sum_t \alpha^t r^t \boldsymbol{\phi}(\boldsymbol{x}^t)$$

$$\frac{\partial L_p}{\partial \xi^t} = C - \alpha^t - \mu^t = 0$$

The dual is now

$$L_d = \sum_t \alpha^t - \frac{1}{2}\sum_t \sum_s \alpha^t \alpha^s r^t r^s \boldsymbol{\phi}(\boldsymbol{x}^t)^T \boldsymbol{\phi}(\boldsymbol{x}^s)$$

subject to

$$\sum_t \alpha^t r^t = 0 \text{ and } 0 \leq \alpha^t \leq C, \forall t$$

The idea in *kernel machines* is to replace the inner product of basis functions, $\boldsymbol{\phi}(\boldsymbol{x}^t)^T \boldsymbol{\phi}(\boldsymbol{x}^s)$, by a *kernel function*, $K(\boldsymbol{x}^t, \boldsymbol{x}^s)$, between instances in the original input space. So instead of mapping two instances $\boldsymbol{x}^t$ and $\boldsymbol{x}^s$ to the $\boldsymbol{z}$-space and doing a dot product there, we directly apply the kernel function in the original space.

$$L_d = \sum_t \alpha^t - \frac{1}{2}\sum_t \sum_s \alpha^t \alpha^s r^t r^s K(\boldsymbol{x}^t, \boldsymbol{x}^s)$$

The kernel function also shows up in the discriminant

$$g(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) = \sum_t \alpha^t r^t \boldsymbol{\phi}(\boldsymbol{x}^t)^T \boldsymbol{\phi}(\boldsymbol{x})$$

$$= \sum_t \alpha^t r^t K(\boldsymbol{x}^t, \boldsymbol{x})$$

This implies that if we have the kernel function, we do not need to map it to the new space at all. Actually, for any valid kernel, there does exist a corresponding mapping function, but it may be much simpler to use $K(\boldsymbol{xt}, \boldsymbol{x})$ rather than calculating $\boldsymbol{\varphi(xt)}$, $\boldsymbol{\varphi(x)}$ and taking the dot product. Many algorithms have been *kernelized*, as we will see in later sections, and that is why we have the name "kernel machines." Thematrix of kernel values, **K**, where **K***ts* Gram matrix = $K(\boldsymbol{xt}, \boldsymbol{xs})$, is called the *Gram matrix*, which should be symmetric and positive semidefinite.

6.Vectorial Kernels

The most popular, general-purpose kernel functions are

- *polynomials* of degree q:

$$K(\boldsymbol{x}^t, \boldsymbol{x}) = (\boldsymbol{x}^T \boldsymbol{x}^t + 1)^q$$

where q is selected by the user. For example, when $q = 2$ and $d = 2$,

$$\begin{aligned} K(\boldsymbol{x}, \boldsymbol{y}) &= (\boldsymbol{x}^T \boldsymbol{y} + 1)^2 \\ &= (x_1 y_1 + x_2 y_2 + 1)^2 \\ &= 1 + 2x_1 y_1 + 2x_2 y_2 + 2x_1 x_2 y_1 y_2 + x_1^2 y_1^2 + x_2^2 y_2^2 \end{aligned}$$

corresponds to the inner product of the basis function (Cherkassky and Mulier 1998):

$$\phi(\boldsymbol{x}) = [1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1 x_2, x_1^2, x_2^2]^T$$

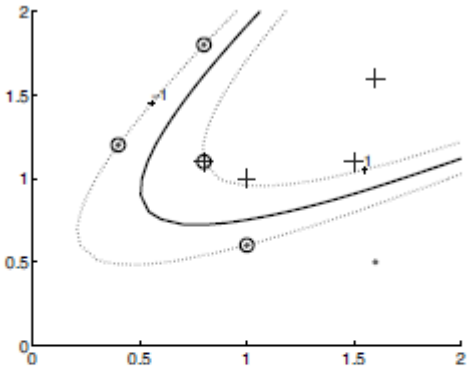


Figure 13.4 The discriminant and margins found by a polynomial kernel of degree 2. Circled instances are the support vectors.

An example is given in figure 13.4. When $q = 1$, we have the *linear kernel* that corresponds to the original formulation.

- *radial-basis functions*:

.30) $$K(\boldsymbol{x}^t, \boldsymbol{x}) = \exp\left[-\frac{\|\boldsymbol{x}^t - \boldsymbol{x}\|^2}{2s^2}\right]$$

defines a spherical kernel as in Parzen windows (chapter 8) where $\boldsymbol{x}t$ is the center and s, supplied by the user, defines the radius.

One can have a Mahalanobis kernel, generalizing from the Euclidean Distance:

$$K(\boldsymbol{x}^t, \boldsymbol{x}) = \exp\left[-\frac{1}{2}(\boldsymbol{x}^t - \boldsymbol{x})^T \mathbf{S}^{-1}(\boldsymbol{x}^t - \boldsymbol{x})\right]$$

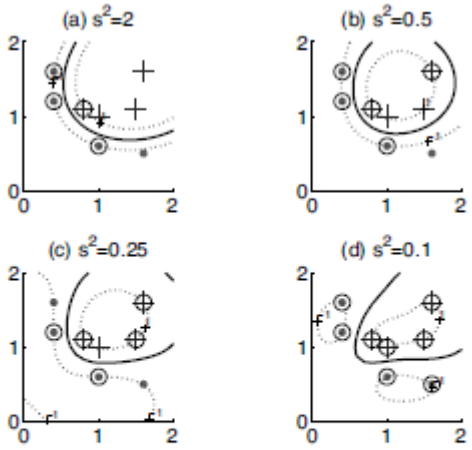


Figure 13.5 The boundary and margins found by the Gaussian kernel with different spread values, s^2. We get smoother boundaries with larger spreads.

where **S** is a covariance matrix. Or, in the most general case,

$$K(\boldsymbol{x}^t, \boldsymbol{x}) = \exp\left[-\frac{\mathcal{D}(\boldsymbol{x}^t, \boldsymbol{x})}{2s^2}\right]$$

for some distance function $\mathcal{D}(\boldsymbol{x}^t, \boldsymbol{x})$.

- *sigmoidal functions*:

$$K(\boldsymbol{x}^t, \boldsymbol{x}) = \tanh(2\boldsymbol{x}^T\boldsymbol{x}^t + 1)$$

7. Multiple Kernel Learning

It is possible to construct new kernels by combining simpler kernels. If $K1(\boldsymbol{x},\boldsymbol{y})$ and $K2(\boldsymbol{x},\boldsymbol{y})$ are valid kernels and c a constant, then

$$K(\boldsymbol{x}, \boldsymbol{y}) = \begin{cases} cK_1(\boldsymbol{x}, \boldsymbol{y}) \\ K_1(\boldsymbol{x}, \boldsymbol{y}) + K_2(\boldsymbol{x}, \boldsymbol{y}) \\ K_1(\boldsymbol{x}, \boldsymbol{y}) \cdot K_2(\boldsymbol{x}, \boldsymbol{y}) \end{cases}$$

are also valid.

Different kernels may also be using different subsets of $\boldsymbol{x}$. We can therefore see combining kernels as another way to fuse information from different sources where each kernel measures similarity according to its domain. When we have input from two representations A and B

$$\begin{aligned} K_A(\boldsymbol{x}_A, \boldsymbol{y}_A) + K_B(\boldsymbol{x}_B, \boldsymbol{y}_B) &= \boldsymbol{\phi}_A(\boldsymbol{x}_A)^T\boldsymbol{\phi}_A(\boldsymbol{y}_A) + \boldsymbol{\phi}_B(\boldsymbol{x}_B)^T\boldsymbol{\phi}_B(\boldsymbol{y}_B) \\ &= \boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\phi}(\boldsymbol{y}) \\ &= K(\boldsymbol{x}, \boldsymbol{y}) \end{aligned}$$

where $\boldsymbol{x} = [\boldsymbol{x}A, \boldsymbol{x}B]$ is the concatenation of the two representations. That is, taking a sum of two kernels corresponds to doing a dot product in the concatenated feature vectors. One can generalize to a number of kernels

$$K(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{m} K_i(\boldsymbol{x}, \boldsymbol{y})$$

which, similar to taking an average of classifiers (section 17.4), this time averages over kernels and frees us from the need to choose one particular kernel. It is also possible to take a weighted sum and also learn the weights from data

$$K(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{m} \eta_i K_i(\mathbf{x}, \mathbf{y})$$

subject to $\eta i \geq 0$, with or without the constraint of $\sum_i \eta_i = 1$, respectively known as convex or conic combination. This is called *multiple kernel learning* where we replace a single kernel with a weighted sum

Finally kernel objective function becomes

$$L_d = \sum_t \alpha^t - \frac{1}{2} \sum_t \sum_s \alpha^t \alpha^s r^t r^s \sum_i \eta_i K_i(\mathbf{x}^t, \mathbf{x}^s)$$

which we solve for both the support vector machine parameters αt and the kernel weights ηi . Then, the combination of multiple kernels also appear in the discriminant

$$g(\mathbf{x}) = \sum_t \alpha^t r^t \sum_i \eta_i K_i(\mathbf{x}^t, \mathbf{x})$$

After training, ηi will take values depending on how the corresponding kernel $Ki(\mathbf{x}t , \mathbf{x})$ is useful in discriminating. It is also possible to localize kernels by defining kernel weights as a parameterized function of the input $\mathbf{x}$, rather like the gating function in mixture of experts

$$g(\mathbf{x}) = \sum_t \alpha^t r^t \sum_i \eta_i(\mathbf{x}|\theta_i) K_i(\mathbf{x}^t, \mathbf{x})$$

7. **Multiclass Kernel Machines**

When there are $K > 2$ classes, the straightforward, *one-vs.-all* way is to define K two-class problems, each one separating one class from all other classes combined and learn K support vector machines $gi(\mathbf{x})$, $i = 1, \ldots, K$.

That is, in training $gi(\mathbf{x})$, examples of Ci are labeled +1 and examples of Ck, $k \neq i$ are labeled as −1. During testing, we calculate all $gi(\mathbf{x})$ and choose the maximum.

Platt (1999) proposed to fit a sigmoid to the output of a single (2-class) SVM output to convert to a posterior probability. Similarly, one can train one layer of softmax outputs to minimize cross-entropy to generate $K >$ 2 posterior probabilities

$$y_i(\mathbf{x}) = \sum_{j=1}^{K} v_{ij} f_j(\mathbf{x}) + v_{i0}$$

where $fj(\mathbf{x})$ are the SVM outputs and yi are the posterior probability outputs. Weights vij are trained to minimize cross-entropy. Note, however, that as in stacking (section 17.9), the data on which we train vij should be different from the data used to train the base SVMs $fj(\mathbf{x})$, to alleviate overfitting. Instead of the usual approach of building K two-class SVM classifiers to separate one from all the rest, as with any other classifier, one can build $K(K − 1)/2$ *pairwise* classifiers (see also section 10.4), each $gij(\mathbf{x})$ taking examples of Ci with the label +1, examples of Cj with

the label −1, and not using examples of the other classes. Separating classes in pairs is normally expected to be an easier job, with the additional advantage that because we use less data, the optimizations will be faster, noting however that we have O*(K*2*)* discriminants to train instead of O*(K)*.

In the general case, both one-vs.-all and pairwise separation are special cases of the *error-correcting output codes* that decompose a multiclass problem to a set of two-class problems Another possibility is to write a single *multiclass* optimization problem involving all classes

$$\min \frac{1}{2}\sum_{i=1}^{K}\|\boldsymbol{w}_i\|^2 + C\sum_t\sum_{\text{¶}}\xi_i^t$$

subject to

$$\boldsymbol{w}_{z^t}\boldsymbol{x}^t + w_{z^t 0} \geq \boldsymbol{w}_i\boldsymbol{x}^t + w_{i0} + 2 - \xi_i^t, \forall i \neq z^t \text{ and } \xi_i^t \geq 0$$

where *zt* contains the class index of ***xt*** . The regularization terms minimizes the norms of all hyperplanes simultaneously, and the constraints are there to make sure that the margin between the actual class and any other class is at least 2. The output for the correct class should be at least +1, the output of any other class should be at least −1, and the slack variables are defined to make up any difference.

8.One-Class Kernel Machines

Support vector machines, originally proposed for classification, are extended to regression by defining slack variables for deviations around the regression line, instead of the discriminant. We now see how SVM can be used for a restricted type of unsupervised learning, namely, for estimating regions of high density. We are not doing a full density estimation; rather, we want to find a boundary (so that it reads like a classification problem) that separates volumes of high density from volumes of low density (Tax and Duin 1999). Such a boundary can then be used for *novelty* or *outlier detection*. This is also called *one-class classification*.

We consider a sphere with center ***a*** and radius *R* that we want to enclose as much as possible of the density, measured empirically as the enclosed training set percentage. At the same time, trading off with it, we want to find the smallest radius. We define slack variables for instances that lie outside (we only have one type of slack variable because we have examples from one class and we do not have any penalty for those inside), and we have a smoothness measure that is proportional to the radius:

$$\min R^2 + C \sum_t \xi^t$$

subject to

$$\|\boldsymbol{x}^t - \boldsymbol{a}\|^2 \le R^2 + \xi^t \text{ and } \xi^t \ge 0, \forall t$$

Adding the constraints, we get the Lagrangian, which we write keeping in mind that $\|\boldsymbol{x}^t - \boldsymbol{a}\|^2 = (\boldsymbol{x}^t - \boldsymbol{a})^T(\boldsymbol{x}^t - \boldsymbol{a})$:

$$L_p = R^2 + C \sum_t \xi^t - \sum_t \alpha^t \left(R^2 + \xi^t - \left[(\boldsymbol{x}^t)^T\boldsymbol{x}^t - 2\boldsymbol{a}^T\boldsymbol{x}^t + \boldsymbol{a}^T\boldsymbol{a}\right]\right) - \sum_t \gamma^t \xi^t$$

with $\alpha^t \ge 0$ and $\gamma_t \ge 0$ being the Lagrange multipliers. Taking the derivative with respect to the parameters, we get

$$\begin{aligned}
\frac{\partial L}{\partial R} &= 2R - 2R\sum_t \alpha^t = 0 \Rightarrow \sum_t \alpha^t = 1 \\
\frac{\partial L}{\partial \boldsymbol{a}} &= \sum_t \alpha^t (2\boldsymbol{x}^t - 2\boldsymbol{a}) = 0 \Rightarrow \boldsymbol{a} = \sum_t \alpha^t \boldsymbol{x}^t \\
\frac{\partial L}{\partial \xi^t} &= C - \alpha^t - \gamma^t = 0
\end{aligned}$$

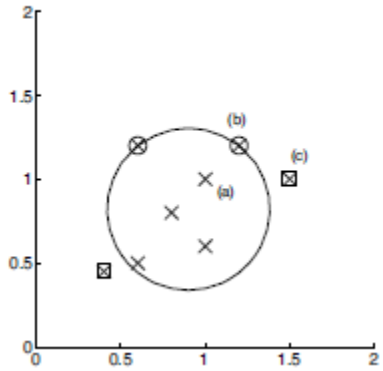


Fig: One class svm

Since $\gamma t \ge 0$, we can write this last as the constraint: $0 \le \alpha t \le C$. Plugging these into equation 13.53, we get the dual that we maximize with respect to αt :

$$L_d = \sum_t \alpha^t (\boldsymbol{x}^t)^T\boldsymbol{x}^t - \sum_t \sum_s \alpha^t \alpha^s (\boldsymbol{x}^t)^T\boldsymbol{x}^s$$

subject to

$$0 \le \alpha^t \le C \text{ and } \sum_t \alpha^t = 1|$$

When we solve this, we again see that most of the instances vanish with their $\alpha t = 0$; these are the typical, highly likely instances that fall inside the sphere (figure 13.10). There are two type of support vectors with $\alpha t > 0$: There are instances that satisfy $0 < \alpha t < C$ and lie on the boundary, $\|\boldsymbol{x}^t - \boldsymbol{a}\|^2 = R^2 (\xi^t = 0)$($\xi t = 0$), which we use to calculate R. Instances.

that satisfy $\alpha^t = C$ ($\xi^t > 0$) lie outside the boundary and are the outliers. From equation 13.55, we see that the center $\boldsymbol{a}$ is written as a weighted sum of the support vectors.

Then given a test input $\boldsymbol{x}$, we say that it is an outlier if

$$\|\boldsymbol{x} - \boldsymbol{a}\|^2 > R^2$$

or

$$\boldsymbol{x}^t\boldsymbol{x} - 2\boldsymbol{a}^T\boldsymbol{x} + \boldsymbol{a}^T\boldsymbol{a} > R^2$$

Using kernel functions, allow us to go beyond a sphere and define boundaries of arbitrary shapes. Replacing the dot product with a kernel function, we get (subject to the same constraints):

$$L_d = \sum_t \alpha^t K(\boldsymbol{x}^t, \boldsymbol{x}^t) - \sum_t \sum_s \alpha^t \alpha^s K(\boldsymbol{x}^t, \boldsymbol{x}^s)$$

For example, using a polynomial kernel of degree 2 allows arbitrary quadratic surfaces to be used. If we use a Gaussian kernel (equation 13.30), we have a union of local spheres. We reject $\boldsymbol{x}$ as an outlier if

$$K(\boldsymbol{x}, \boldsymbol{x}) - 2\sum_t \alpha^t K(\boldsymbol{x}, \boldsymbol{x}^t) + \sum_t \sum_s \alpha^t \alpha^s K(\boldsymbol{x}^t, \boldsymbol{x}^s) > R^2$$

The third term does not depend on $\boldsymbol{x}$ and is therefore a constant (we use this as an equality to solve for R where $\boldsymbol{x}$ is an instance on the margin). In the case of a Gaussian kernel where $K(\boldsymbol{x}, \boldsymbol{x}) = 1$, the condition reduces to

$$\sum_t \alpha^t K_G(\boldsymbol{x}, \boldsymbol{x}^t) < R_c$$

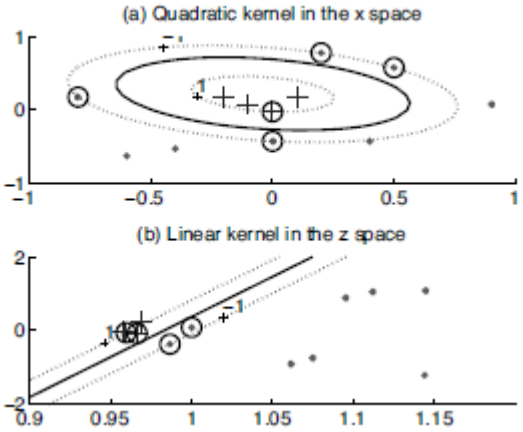
9.Kernel Dimensionality Reduction

In the kernelized version, we work in the space of $\boldsymbol{\varphi(x)}$ instead of the original $\boldsymbol{x}$ and because, as usual, the dimensionality d of this new space may be much larger than the data set size N, we prefer to work with the $N \times N$ matrix $\mathbf{X}\mathbf{X}T$ instead of the $d \times d$ matrix $\mathbf{X}T\mathbf{X}$. The projected data matrix is $\mathbf{\Phi} = \boldsymbol{\varphi}(\mathbf{X})$, and hence we work on the eigenvectors of $\mathbf{\Phi}T\mathbf{\Phi}$ and hence of the kernel matrix $\mathbf{K}$.

Kernel PCA uses the eigenvectors and eigenvalues of the kernel matrix and this corresponds to doing a linear dimensionality reduction in the $\varphi(\boldsymbol{x})$ space. When $\boldsymbol{ci}$ and λi are the corresponding eigenvectors and eigenvalues, the projected new k-dimensional values can be calculated as

$$z_j^t = \sqrt{\lambda_t}\boldsymbol{c}_j^t, j = 1, \ldots, k, \; t = 1, \ldots, N$$

An example is given in figure 13.12 where we first use a quadratic kernel and then decrease dimensionality to two (out of five) using kernel PCA and implement a linear SVM there. Note that in the general case (e.g., with a Gaussian kernel), the eigenvalues do not necessarily decay and there is no guarantee that we can reduce dimensionality using kernel PCA. What we are doing here is multidimensional scaling using kernel values as the similarity values. For example, by taking $k = 2$, one can visualize the data in the space induced by the kernel matrix, which can give us information as to how similarity is defined by the used kernel. Linear discriminality reduction can similarly be kernelized.



Part- B(Graphical Models)

1.Introduction:

Graphical models, also called *Bayesian networks*, *belief networks*, or *probabilistic networks*, are composed of nodes and arcs between the nodes. Each node corresponds to a random variable, X, and has a value corresponding to the probability of the random variable, $P(X)$. If there is a directed arc from node X to node Y, this indicates that X has a *direct influence* on Y. This influence is specified by the conditional probability $P(Y|X)$. The network is a *directed acyclic graph* (DAG); namely, there are no cycles. The nodes and the arcs between the nodes define the *structure* of the network, and the conditional probabilities are the *parameters* given the structure.

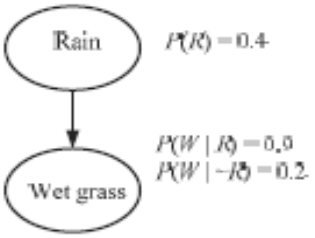


Figure 16.1 Bayesian network modeling that rain is the cause of wet grass.

We see that these three values completely specify the joint distribution of $P(R, W)$. If $P(R) = 0.4$, then $P(\sim R) = 0.6$, and similarly $P(\sim W|R) = 0.1$ and $P(\sim W|\sim R) = 0.8$. The joint is written as

$$P(R, W) = P(R)P(W|R)$$

We can calculate the individual (marginal) probability of wet grass by summing up over the possible values that its parent node can take:

$$\begin{aligned} P(W) &= \sum_R P(R, W) = P(W|R)P(R) + P(W|\sim R)P(\sim R) \\ &= 0.9 \cdot 0.4 + 0.2 \cdot 0.6 = 0.48 \end{aligned}$$

If we knew that it rained, the probability of wet grass would be 0.9; if we knew for sure that it did not, it would be as low as 0.2; not knowing whether it rained or not, the probability is 0.48.

For example, knowing that the grass is wet, the probability that it rained can be calculated as follows:

$$P(R|W) = \frac{P(W|R)P(R)}{P(W)} = 0.75$$

Knowing that the grass is wet increased the probability of rain from 0.4 to 0.75; this is because $P(W|R)$ is high and $P(W|\sim R)$ is low.

We form graphs by adding nodes and arcs and generate dependencies.

X and Y are *independent events* if

$$p(X,Y) = P(X)P(Y)$$

X and Y are *conditionally independent events* given a third event Z if

$$P(X,Y|Z) = P(X|Z)P(Y|Z)$$

which can also be rewritten as

$$P(X|Y,Z) = P(X|Z)$$

X and *Y* are *conditionally independent events* given a third event *Z* if

$P(X,Y|Z) = P(X|Z)P(Y|Z)$

which can also be rewritten as

$P(X|Y,Z) = P(X|Z)$

In a graphical model, not all nodes are connected; actually, in general, a node is connected to only a small number of other nodes. Certain subgraphs imply conditional independence statements, and these allow us to break down a complex graph into smaller subsets in which inferences can be done locally and whose results are later propagated over the graph. There are three canonical cases and larger graphs are constructed using these as subgraphs.

2.Canonical Cases for Conditional Independence

Case 1: Head-to-tail Connection

We see here that *X* and *Z* are independent given *Y*: Knowing *Y* tells *Z* everything; knowing the state of *X* does not add any extra knowledge for *Z*; we write *P(Z|Y,X) = P(Z|Y)*. We say that *Y blocks* the path from *X* to *Z*, or in other words, it *separates* them in the sense that if *Y* is removed, there is no path between *X* to *Z*. In this case, the joint is written as

P(X, Y,Z) = P(X)P(Y|X)P(Z|Y)

Writing the joint this way implies independence

$$P(Z|X,Y) = \frac{P(X,Y,Z)}{P(X,Y)} = \frac{P(X)P(Y|X)P(Z|Y)}{P(X)P(Y|X)} = P(Z|Y)$$

Typically, *X* is the cause of *Y* and *Y* is the cause of *Z*. For example, as seen in figure 16.2b, *X* can be cloudy sky, *Y* can be rain, and *Z* can be wet grass. We can propagate information along the chain. If we do not know the state of cloudy, we have

$P(R) = P(R|C)P(C) + P(R|\sim C)P(\sim C) = 0.38$

$P(W) = P(W|R)P(R) + P(W|\sim R)P(\sim R) = 0.47$

Let us say, in the morning we see that the weather is cloudy; what can we say about the probability that the grass will be wet? To do this, we

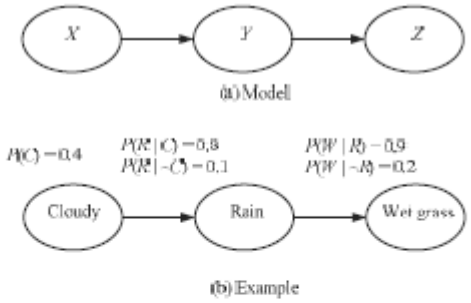


Figure 16.2 Head-to-tail connection. (a) Three nodes are connected serially. X and Z are independent given the intermediate node Y: $P(Z|Y,X) = P(Z|Y)$. (b) Example: Cloudy weather causes rain, which in turn causes wet grass.

need to propagate evidence first to the intermediate node R, and then to the query node W.

$$P(W|C) = P(W|R)P(R|C) + P(W|\sim R)P(\sim R|C) = 0.76$$

Knowing that the weather is cloudy increased the probability of wet grass. We can also propagate evidence back using Bayes' rule. Let us say that we were traveling and on our return, see that our grass is wet; what is the probability that the weather was cloudy that day? We use Bayes' rule to invert the direction:

$$P(C|W) = \frac{P(W|C)P(C)}{P(W)} = 0.65$$

Knowing that the grass is wet increased the probability of cloudy weather from its default (prior) value of 0.4 to 0.65.

Case 2: Tail-to-tail Connection

X may be the parent of two nodes Y and Z, as shown in figure 16.3a. The joint density is written as

$$P(X,Y,Z) = P(X)P(Y|X)P(Z|X)$$

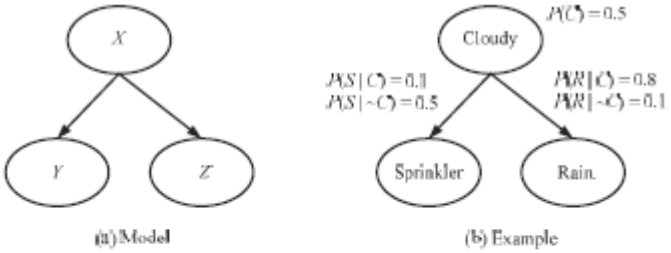


Figure 16.3 Tail-to-tail connection. X is the parent of two nodes Y and Z. The two child nodes are independent given the parent: $P(Y|X,Z) = P(Y|X)$. In the example, cloudy weather causes rain and also makes us less likely to turn the sprinkler on.

Normally Y and Z are dependent through X; given X, they become independent:

$$P(Y,Z|X) = \frac{P(X,Y,Z)}{P(X)} = \frac{P(X)P(Y|X)P(Z|X)}{P(X)} = P(Y|X)P(Z|X) \quad (16.7)$$

When its value is known, X blocks the path between Y and Z, or in other words, separates them.
we see an example where cloudy weather influences both rain and the use of the sprinkler, one positively and the other negatively. Knowing that it rained, for example, we can invert the dependency using Bayes' rule and infer the cause:

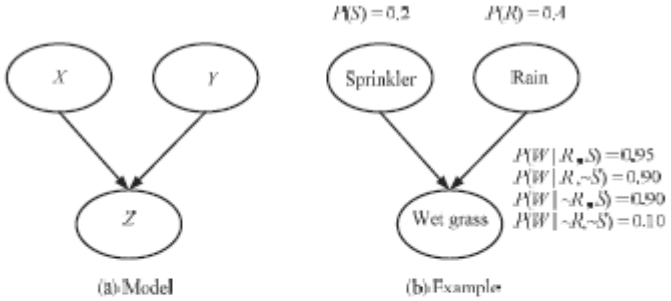


Figure 16.4 Head-to-head connection. A node has two parents that are independent unless the child is given. For example, an event may have two independent causes.

$$\begin{aligned} &= P(R|C)\frac{P(S|C)P(C)}{P(S)} + P(R|\sim C)\frac{P(S|\sim C)P(\sim C)}{P(\sim S)} \\ &= 0.22 \end{aligned}$$

$$\begin{aligned} P(C|R) &= \frac{P(R|C)P(C)}{P(R)} = \frac{P(R|C)P(C)}{\sum_C P(R,C)} \\ &= \frac{P(R|C)P(C)}{P(R|C)P(C) + P(R|\sim C)P(\sim C)} = 0.89 \end{aligned}$$

Note that this value is larger than $P(C)$; knowing that it rained increased the probability that the weather is cloudy.

In figure 16.3a, if X is not known, knowing Y, for example, we can infer X which we can then use to infer Z. In figure 16.3b, knowing the state of the sprinkler has an effect on the probability that it rained. If we know that the sprinkler is on,

$$P(R|S) = \sum_C P(R,C|S) = P(R|C)P(C|S) + P(R|\sim C)P(\sim C|S)$$

:

This is less than $P(R) = 0.45$; that is, knowing that the sprinkler is on decreases the probability that it rained because sprinkler and rain happens for different states of cloudy weather. If the sprinkler is known to be off, using the same approach, we find that $P(R|\sim S) = 0.55$; the probability of rain increases this time.

Case 3: Head-to-head Connection

In a head-to-head node, there are two parents X and Y to a single node Z, as shown in figure 16.4a. The joint density is written as $P(X, Y,Z) = P(X)P(Y)P(Z|X, Y)$ X and Y are independent: $P(X,Y) = P(X) \cdot P(Y)$ (exercise 2); they become

dependent when Z is known. The concept of blocking or separation is different for this case: The path between X and Y is blocked, or they are separated, when Z is *not* observed; when Z (or any of its descendants)

is observed, they are not blocked, separated, nor are independent .

We see for example in figure 16.4b that node W has two parents, R and S, and thus its probability is conditioned on the values of those two, $P(W|R,S)$.

Not knowing anything else, the probability that grass is wet is calculated by marginalizing over the joint:

$$\begin{aligned} P(W) &= \sum_{R,S} P(W,R,S) \\ &= P(W|R,S)P(R,S) + P(W|\sim R,S)P(\sim R,S) \\ &\quad + P(W|R,\sim S)P(R,\sim S) + P(W|\sim R,\sim S)P(\sim R,\sim S) \\ &= P(W|R,S)P(R)P(S) + P(W|\sim R,S)P(\sim R)P(S) \\ &\quad + P(W|R,\sim S)P(R)P(\sim S) + P(W|\sim R,\sim S)P(\sim R)P(\sim S) \\ &= 0.52 \end{aligned}$$

Now, let us say that we know that the sprinkler is on, and we check how this affects the probability. This is a causal (predictive) inference:

$$\begin{aligned} P(W|S) &= \sum_{R} P(W,R|S) \\ &= P(W|R,S)P(R|S) + P(W|\sim R,S)P(\sim R|S) \\ &= P(W|R,S)P(R) + P(W|\sim R,S)P(\sim R) \\ &= 0.92 \end{aligned}$$

We see that $P(W|S) > P(W)$; knowing that the sprinkler is on, the probability of wet grass increases.

We can also calculate the probability that the sprinkler is on, given that the grass is wet. This is a diagnostic inference.

$$P(S|W) = \frac{P(W|S)P(S)}{P(W)} = 0.35$$

$P(S|W) > P(S)$, that is, knowing that the grass is wet increased the probability of having the sprinkler on. Now let us assume that it rained. Then we have

$$\begin{aligned} P(S|R,W) &= \frac{P(W|R,S)P(S|R)}{P(W|R)} = \frac{P(W|R,S)P(S)}{P(W|R)} \\ &= 0.21 \end{aligned}$$

which is less than $P(S|W)$. This is called *explaining away*: given that we know it rained, the probability of sprinkler causing the wet grass decreases. Knowing that the grass is wet, rain and sprinkler become dependent. Similarly, $P(S|\sim R,W) > P(S|W)$. We see the same behavior when we compare $P(R|W)$ and $P(R|W,S)$ (exercise 3).

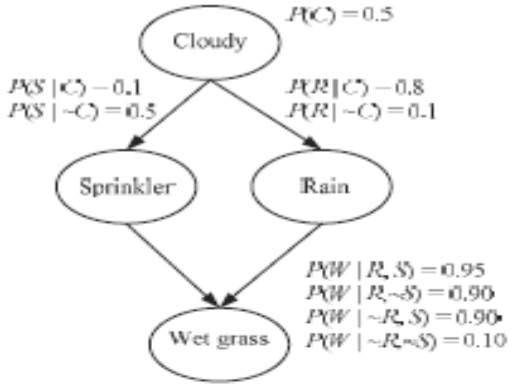


Figure 16.5 Larger graphs are formed by combining simpler subgraphs over which information is propagated using the implied conditional independencies.

We can construct larger graphs by combining such subgraphs. For example, in figure 16.5 where we combine the two subgraphs, we can, for example, calculate the probability of having wet grass if it is cloudy:

$$
\begin{aligned}
P(W|C) &= \sum_{R,S} P(W,R,S|C) \\
&= P(W,R,S|C) + P(W,\sim R,S|C) \\
&\quad + P(W,R,\sim S|C) + P(W,\sim R,\sim S|C) \\
&= P(W|R,S,C)P(R,S|C) \\
&\quad + P(W|\sim R,S,C)P(\sim R,S|C) \\
&\quad + P(W|R,\sim S,C)P(R,\sim S|C) \\
&\quad + P(W|\sim R,\sim S,C)P(\sim R,\sim S|C) \\
&= P(W|R,S)P(R|C)P(S|C) \\
&\quad + P(W|\sim R,S)P(\sim R|C)P(S|C) \\
&\quad + P(W|R,\sim S)P(R|C)P(\sim S|C) \\
&\quad + P(W|\sim R,\sim S)P(\sim R|C)P(\sim S|C)
\end{aligned}
$$

where we have used that $P(W|R, S,C) = P(W|R, S)$; given R and S, W is independent of C: R and S between them block the path between W and C. Similarly, $P(R,S|C) = P(R|C)P(S|C)$; given C, R and S are independent.
We see the advantage of Bayesian networks here, which explicitly encode independencies and allow breaking down inference into calculation over small groups of variables that are propagated from evidence nodes to query nodes.
We can calculate $P(C|W)$ and have a diagnostic inference:

$$P(C|W) = \frac{P(W|C)P(C)}{P(W)}$$

3.Example Graphical Models

A. Naive Bayes' Classifier

For the case of classification, the corresponding graphical model is shown in figure 16.6a, with $\boldsymbol{x}$ as the input and C a multinomial variable taking

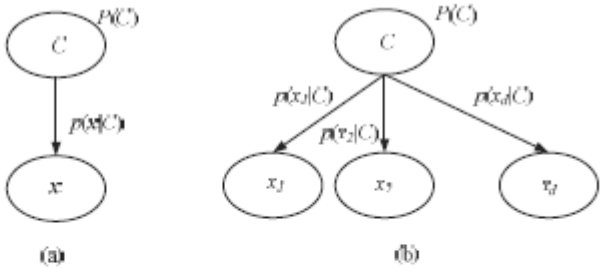


Figure 16.6 (a) Graphical model for classification. (b) Naive Bayes' classifier assumes independent inputs.

one of K states for the class code. Bayes' rule allows a diagnosis, as in the rain and wet grass case we saw in figure 16.1:

$$P(C|\mathbf{x}) = \frac{P(C)p(\mathbf{x}|C)}{P(\mathbf{x})}$$

If the inputs are independent,which is called the *naive Bayes' classifier*, because it ignores possible dependencies, namely, correlations, among the inputs and reduces a multivariate problem to a group of univariate problems:

$$p(\mathbf{x}|C) = \prod_{j=1}^{d} p(x_j|C)$$

Figure 16.6a is a *generative model* of the process that creates the data. It is as if we first pick a class C at random by sampling from *P(C)*, and then having fixed C, we pick an ***x*** by sampling from *p(**x**|C)*. 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 where an author decides to write a document on a certain topic and then chooses the set of words accordingly. In bioinformatics, one area among many where a graphical approach used is the modelling of a *phylogenetic tree*; namely, a directed graph whose leaves are the current species, nonterminal nodes are past ancestors that split into multiple species during a speciation event, and the conditional probabilities depend on the evolutionary distance between a species and its ancestor.

B.Hidden Markov Model:

Hidden Markov models (HMM) are an example of case 1 where three successive states q_{t-2}, q_{t-1}, q_t correspond to 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 **A** (see figure 16.7). Each hidden variable generates a discrete observation that is observed, as given by the observation probability matrix **B**. The forward-backward procedure of hidden Markov models is a special case of belief propagation that we will discuss shortly.

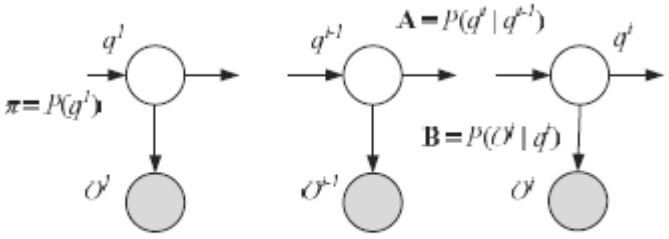
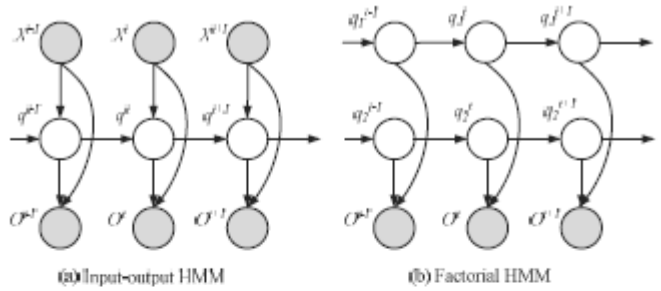


Figure 16.7 Hidden Markov model can be drawn as a graphical model where q^t are the hidden states and shaded O^t are observed.



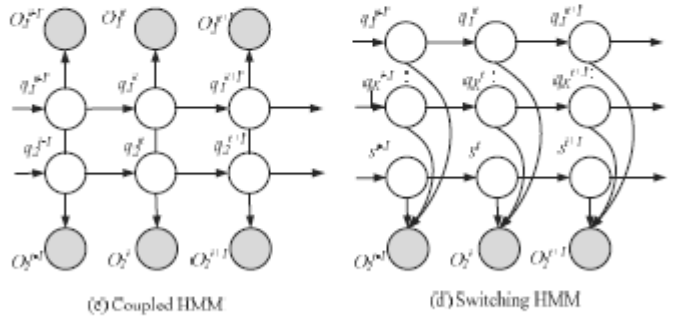


Figure 16.8 Different types of HMM model different assumptions about the way the observed data (shown shaded) is generated from Markov sequences of latent variables.

C. **Linear Regression**

Linear regression can be visualized as a graphical model, as shown in figure 16.9. Input ***xt*** is drawn from a prior ***p(x)*** and the dependent variable *rt* depend on the input ***x***, weights ***w*** (drawn from a prior parameterized by α, i.e., $p(\boldsymbol{w}) \sim \mathcal{N}(\mathbf{0}, \alpha^{-1}\mathbf{I})$), and noise ϵ .

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

There are N such pairs in the training set, which is shown by the rectangular *plate* in the figure. Given a new input $\boldsymbol{x}'$, the aim is to estimate r', which will be $E[r'|\boldsymbol{x}', \boldsymbol{w}]$.

The weights $\boldsymbol{w}$ are not given but they can be estimated using the training set of $[\mathbf{X}, \boldsymbol{r}]$. Just as in equation 16.9, where C is the cause of R and S, where we used

$$P(R|S) = \sum_C P(R, C|S) = P(R|C)P(C|S) + P(R|\sim C)P(\sim C|S)$$

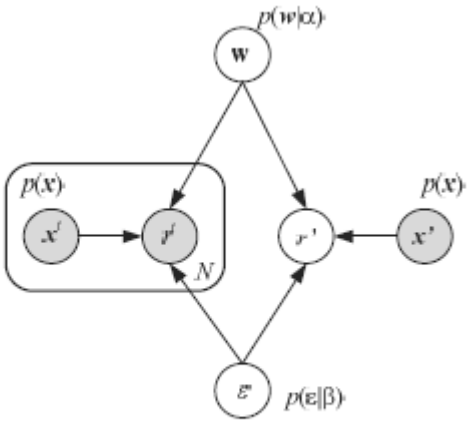


Figure 16.9 Bayesian network for linear regression.

filling in *C* using *S*, which we in turn used to estimate *R*. Here, we write

$$\begin{aligned} p(r'|\mathbf{x}', \mathbf{r}, \mathbf{X}) &= \int p(r'|\mathbf{x}', w) p(w|\mathbf{X}, \mathbf{r}) dw \\ &= \int p(r'|\mathbf{x}', w) \frac{p(\mathbf{r}|\mathbf{X}, w) p(w)}{p(\mathbf{r})} dw \\ &\propto \int p(r'|\mathbf{x}', w) \prod_t p(r^t|\mathbf{x}^t, w) p(w) dw \end{aligned}$$

4. **d-Separation**

We now generalize the concept of blocking and separation under the name of *d-separation*, and we define it in a way so that for arbitrary subsets of nodes *A*, *B*, and *C*, we can check if *A* and *B* are independent given *C*. Jordan visualizes this as a ball bouncing over the graph and calls this the *Bayes' ball*. We set the nodes in *C* to their values, place a

ball at each node in *A*, let the balls move around according to a set of rules, and check whether a ball reaches any node in B. If this is the case, they are dependent; otherwise, they are independent.

To check whether *A* and *B* are d-separated given *C*, we consider all possible paths between any node in *A* and any node in *B*. Any such path is *blocked* if

(a) the directions of the edges on the path either meet head-to-tail (case 1)\or tail-to-tail (case 2) and the node is in *C*, or

(b) the directions of the edges on the path meet head-to-head (case 3) and neither that node nor any of its descendant is in *C*. If all paths are blocked, we say that *A* and *B* are d-separated, that is, independent, given *C*; otherwise, they are dependent. Examples are given in figure 16.10.

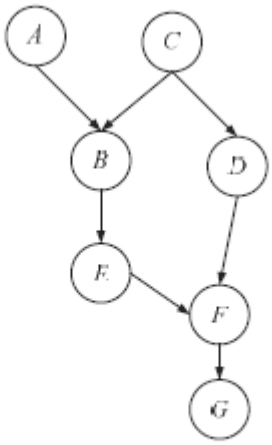


Figure 16.10 Examples of d-separation. The path *BCDF* is blocked given *C* because *C* is a tail-to-tail node. *BEFG* is blocked by *F* because *F* is a head-to-tail node. *BEFD* is blocked unless *F* (or *G*) is given.

5. Belief Propagation

Having discussed some inference examples by hand, we now are interested in an algorithm that can answer queries such as $P(X|E)$ where X is any *query node* in the graph and E is any subset of *evidence nodes* whose values are set to certain value. Following Pearl (1988), we start with the simplest case of chains and gradually move on to more complex graphs. Our aim is to find the graph operation counterparts of probabilistic procedures such as Bayes' rule or marginalization, so that the task of inference can be mapped to general purpose graph algorithms.

A.Chains

A *chain* is a sequence of head-to-tail nodes with one *root* node without any parent; all other nodes have exactly one parent node, and all nodes except the very last, *leaf*, have a single child. If evidence is in the ancestors of X, we can just do a diagnostic inference and propagate evidence down the chain; if evidence is in the descendants of X, we can do a causal inference and propagate upward using Bayes' rule. Let us see the general case where we have evidence in both directions, up the chain $E+$ and down the chain $E-$ (see figure 16.11). Note that any evidence node separates X from the nodes on the chain on the other side of the evidence and their values do not affect $p(X)$; this is true in both directions.

We consider each node as a processor that receives messages from its neighbors and pass it along after some local calculation. Each node X locally calculates and stores two values: $\lambda(X) \equiv P(E-|X)$ is the propagated $E-$ that X receives from its child and forwards to its parent, and $\pi(X) \equiv P(X|E+)$ is the propagated $E+$ that X receives from its parent

and passes on to its child.

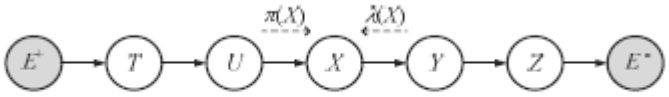


Figure 16.11 Inference along a chain.

$$
\begin{aligned}
P(X|E) &= \frac{P(E|X)P(X)}{P(E)} = \frac{P(E^+,E^-|X)P(X)}{P(E)} \\
&= \frac{P(E^+|X)P(E^-|X)P(X)}{P(E)} \\
&= \frac{P(X|E^+)P(E^+)P(E^-|X)P(X)}{P(X)P(E)} \\
&= \alpha P(X|E^+)P(E^-|X) = \alpha\pi(X)\lambda(X)
\end{aligned}
$$

Given these initial conditions, we can devise recursive formulas to propagate evidence along the chain.

For the π-messages, we have

$$
\begin{aligned}
\pi(X) &= P(X|E^+) = \sum_U P(X|U,E^+)P(U|E^+) \\
&= \sum_U P(X|U)P(U|E^+) = \sum_U P(X|U)\pi(U)
\end{aligned}
$$

where the second line follows from the fact that *U* blocks the path between *X* and *E*+.

For the λ-messages, we have

$$
\begin{aligned}
\lambda(X) &= P(E^-|X) = \sum_Y P(E^-|X,Y)P(Y|X) \\
&= \sum_Y P(E^-|Y)P(Y|X) = \sum_U P(Y|X)\lambda(Y)
\end{aligned}
$$

B. Trees

Chains are restrictive because each node can have only a single parent and a single child, that is, a single cause and a single symptom. In a *tree*, each node may have several children but all nodes, except the single root, have exactly one parent. The same belief propagation also applies here with the difference from chains being that a node receives different λ-messages from its children, $\lambda Y(X)$ denoting the message *X* receives from its child *Y*, and sends different π-messages to its children, $\pi Y(X)$ denoting the message *X* sends to its child *Y*. Again, we divide possible evidence to two parts, *E*− are nodes that are in the subtree rooted at the query node *X*, and *E*+ are evidence nodes elsewhere (see figure 16.12). Note that this second need not be an ancestor of *X* but may also be in a subtree rooted at a sibling of *X*. The

important point is that again *X* separates *E*+ and *E*− so that we can write

$P(E+, E-|X) = P(E+|X)P(E-|X)$, and hence have

$P(X|E) = \alpha\pi(X)\lambda(X)$

where again α is a normalizing constant. $\lambda(X)$ is the evidence in the subtree rooted at *X*, and if *X* has two children *Y* and *Z*, it can be calculated as

$$\begin{aligned}\lambda(X) &= P(E_X^-|X) = P(E_Y^-, E_Z^-|X)\\ &= P(E_Y^-|X)P(E_Z^-|X) = \lambda_Y(X)\lambda_Z(X)\end{aligned}$$

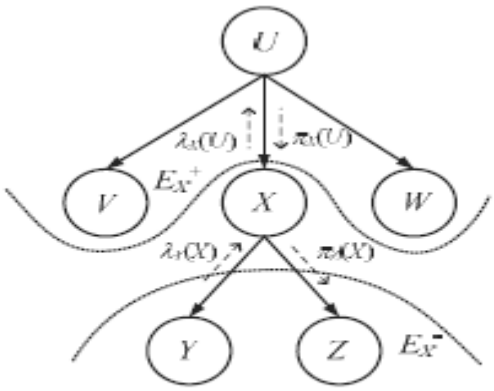


Figure 16.12 In a tree, a node may have several children but a single parent.

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

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

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

$$\lambda_X(U) = \sum_X \lambda(X)P(X|U)$$

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:

$$\pi(X) = P(X|E_X^+) = \sum_U P(X|U)P(U|E_X^+) = \sum_U P(X|U)\pi_X(U)$$

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 16.12):

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

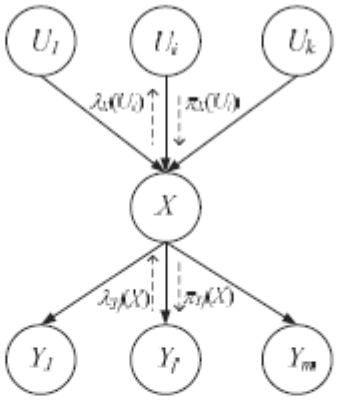


Figure 16.13 In a polytree, a node may have several children and several parents, but the graph is singly connected; that is, there is a single chain between U_i and Y_j passing through X.

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

$$= \alpha\lambda_Z(X)\pi(X) \tag{16.22}$$

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

$$\pi_{Y_j}(X) = \alpha \prod_{s \neq j} \lambda_{Y_s}(X)\pi(X) \tag{16.23}$$

C. Polytrees

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 EX into $E+$

X and $E- X$, which are independent given X .

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

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

$$\begin{aligned}\pi(X) &= P(X|E_X^+) = P(X, E_{U_1X}^+, E_{U_2X}^+, \ldots, E_{U_kX}^+) \\ &= \sum_{U_1}\sum_{U_2}\cdots\sum_{U_k} P(X|U_1, U_2, \ldots, U_k)P(U_1|E_{U_1X}^+)\cdots P(U_k|E_{U_kX}^+) \\ &= \sum_{U_1}\sum_{U_2}\cdots\sum_{U_k} P(X|U_1, U_2, \ldots, U_k)\prod_{i=1}^{k}\pi_X(U_i) \end{aligned} \tag{16.24}$$

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

$$\pi_{Y_j}(X) = \alpha\prod_{s\neq j}\lambda_{Y_s}(X)\pi(X) \tag{16.25}$$

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 $E_{U_iX}^-$:

$$\begin{aligned}\lambda_X(U_i) &= P(E_{U_iX}^-|X) \\ &= \sum_X\sum_{U_{r\neq i}} P(E_X^-, E_{U_{r\neq i}X}^+, X, U_{r\neq i}|U_i) \\ &= \sum_X\sum_{U_{r\neq i}} P(E_X^-, E_{U_{r\neq i}X}^+|X, U_{r\neq i}, U_i)P(X, U_{r\neq i}|U_i) \\ &= \sum_X\sum_{U_{r\neq i}} P(E_X^-|X)P(E_{U_{r\neq i}X}^+|U_{r\neq i})P(X|U_{r\neq i}, U_i)P(U_{r\neq i}|U_i) \\ &= \sum_X\sum_{U_{r\neq i}} P(E_X^-|X)\frac{P(U_{r\neq i}|E_{U_{r\neq i}X}^+)P(E_{U_{r\neq i}X}^+)}{P(U_{r\neq i})}P(X|U_{r\neq i}, U_i)P(U_{r\neq i}|U_i) \\ &= \beta\sum_X\sum_{U_{r\neq i}} P(E_X^-|X)P(U_{r\neq i}|E_{U_{r\neq i}X}^+)P(X|U_{r\neq i}, U_i) \\ &= \beta\sum_X\sum_{U_{r\neq i}}\lambda(X)\prod_{r\neq i}\pi_X(U_r)P(X|U_1, \ldots, U_k) \\ &= \beta\sum_X\lambda(X)\sum_{U_{r\neq i}} P(X|U_1, \ldots, U_k)\prod_{r\neq i}\pi_X(U_r) \end{aligned} \tag{16.26}$$

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

$$\lambda(X) = \prod_{j=1}^{m}\lambda_{Y_j}(X) \tag{16.27}$$

D.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. We can then run the same belief propagation algorithm with some modifications.

This is the basic idea behind the *junction tree algorithm.*

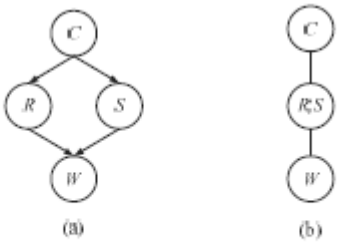


Figure 16.14 (a) A multiply connected graph, and (b) its corresponding junction tree with nodes clustered.

6. Learning the Structure of a Graphical Model

Learning a graphical model has two parts. The first is the learning of parameters given a structure; this is relatively easier , and, in graphical models, conditional probability tables or their parameterizations can be trained to maximize the likelihood, or by using a Bayesian approach if suitable priors are known.

The second, more difficult, and interesting part is to learn the graph structure .This is basically a model selection problem, and just like the incremental approaches for learning the structure of a multilayer perceptron , 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.

7. **Influence Diagrams**

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 .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. Given the input, the decision node decides on a class, and for each choice we incur a certain utility (risk).

8. **Undirected Graphs: Markov Random Fields**

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.

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.

In the case of an undirected graph, we do not talk about the parent or clique 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 ψC(XC)* where *XC* 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

$$p(X) = \frac{1}{Z}\prod_{C}\psi_C(X_C)$$

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

$$Z = \sum_X \prod_C \psi_C(X)$$

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.

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

$$p(X) = \frac{1}{Z}\prod_{s} f_s(X_s)$$

It is possible to generalize the belief propagation algorithm to work on factor graphs; this is called the *sum-product algorithm* 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 messages.