Chapter 14
Statistical Learning Theory

Abstract A machine learning system, in general, learns from the environment, but statistical machine learning programs (systems) learn from the data. This chapter presents techniques for statistical machine learning using Support Vector Machines (SVM) to recognize the patterns and classify them, predicting structured objects using SVM, k-nearest neighbor method for classification, and Naive Bayes classifiers. The artificial neural networks are presented with brief introduction to error-correction rules, Boltzmann learning, Hebbian rule, competitive learning rule, and deep learning. The instance-based learning is treated in details with its algorithm and learning task. The chapter concludes with a summary, and a set of practice exercises.

Keywords Statistical machine learning · Support Vector Machines (SVM) · K-nearest method · Naive Bayes classifier · Artificial Neural Networks (ANN) · Boltzmann learning · Hebbian rule · Deep learning · Instance-Based Learning (IBL)

14.1 Introduction

Statistical machine learning systems help the programs automatically learn from the data. This is an attractive option as an alternative to manually coding every rule in a program. Machine learning is used in computer science and beyond, e.g., in spam filters in email, web searching, placement advertisement, stock trading, credit scoring, drug design, fraud detection, and in many more applications. The statistical machine learning can be said to be a kind of a data mining.

The field of machine learning developed in a new direction during 1979–80, with innovations like *decision trees* and *rule learning*. These methods were applied in expert systems. In the late 1980s, there were renewals of research interests in machine learning architectures that used *perceptron* as the basic building block. This was particularly because the limitations which were highlighted by *Minksy* and *Papert* in the early days of AI were overcome by *multilayer networks* that used simple computing elements. The latter used perceptrons like nodes called *neural networks*. These networks were trained using biologically inspired backpropagation

K. R. Chowdhary, *Fundamentals of Artificial Intelligence*,
https://doi.org/10.1007/978-81-322-3972-7_14

algorithms, which performed gradient descent on error-based cost functions. This new approach raised the hopes not only of creating machines that were capable to learn, but also of understanding the basic mechanisms used by biological learning systems.

The machine learning algorithms can determine how to perform important tasks through generalization from examples. A machine learning-based solution, at the end, turns out to be a feasible and cost-effective solution when compared with the manual programming approach. In statistical-based machine learning approaches, as more data becomes available, it becomes possible to tackle more ambitious problems.

In the previous chapter we concentrated on broad classification of machine learning techniques, and basic principles of each of them. In the present chapter we will discuss new and emerging techniques, which are mainly statistical based. For the purpose of illustration, we focus on the most mature and widely used classification for these techniques.

Learning Outcomes of This Chapter:

1. Apply the simple statistical learning algorithm such as Naive Bayesian Classifier to a classification task and measure the classifier's accuracy. [Usage]
2. Apply the simple statistical learning algorithm such as SVM to a classification task and measure the classifier's accuracy. [Usage]
3. Evaluate the performance of a simple learning system on a real-world data set. [Assessment]
4. Compare and contrast each of the following techniques, providing examples of, in what condition each strategy is superior: decision trees, neural networks, and belief networks, deep learning. [Assessment]

14.2 Classification

One of the sub-fields of predictive modeling is supervised pattern classification. It is a task of training a *model* based on labeled training data, such that the model can later be used to assign predefined class labels to new objects. Figure 14.1 shows this process.

Definition 14.1 (*Classifier*) We refer to a *classifier* as a system that typically has inputs vector of discrete and/or continuous feature values and outputs a single discrete value, the *class*. □

In that sense, for email filtering we use a classifier that classifies the email messages into "spam" and "no spam" classes. The input to this classifier may be a Boolean vector,

$$\mathbf{x} = (x_1, \dots, x_j, \dots, x_d), \tag{14.1}$$

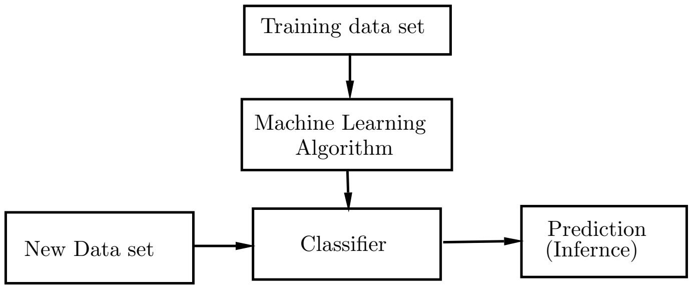


Fig. 14.1 A classifier's block diagram

where $x_1, \ldots, x_d$ corresponds to words from some standard dictionary, such that the element $x_j = 1$ (True) if the jth word of that dictionary is present in the email under consideration. If the jth word from the dictionary is absent in email, then $x_j = 0$. The aim is naturally to classify the emails based on these words. A learner (program) inputs a training set of examples, comprising $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,d})$, as an observed input, y_i as the supposed to be corresponding output, and also the classifier, which is supposed to be the class of this email. The test, that the learner program (i.e., classifier) has successfully learned is, whether this classifier produces the correct output y_t for future examples $\mathbf{x}_t$. That is, whether the spam filter has correctly classified the first-time seen email messages as spam or not spam, or some other classes as dictated by the classifier [6].

Though there are thousands of learning algorithms today, the learning can be expressed as comprising three parts: representation, evaluation, and optimization.

Representation

The classifier must be represented in some formal language. Choosing a representation amounts to choosing a set of classifiers with the capability to learn. The representing set is called the *hypothesis space* of the learner, i.e., it covers those features of the inputs to which we give importance. The classifier must be in the hypothesis space, otherwise it cannot learn.

Evaluation

The evaluation function, called *object function* or *scoring function* of a learning algorithm, should be able to distinguish between good classifiers from bad classifiers.

Optimization

This function is useful for searching a method for a classifier that helps in scoring the highest. Proper choice of optimization technique is key to the efficiency of the learner.

The learners can be divided into two major types: 1. whose representation comprises fixed size, e.g., *linear classifiers*, 2. those whose representation can grow with data, e.g., *decision trees*.

14.3 Support Vector Machines

Due to Support Vector Machines (SVMs), a new approach is introduced to machine learning; the method is based more on statistical foundations and less on biological plausibility. SVMs are trained by minimizing a convex cost function, which is a measure of the margin of correct classification. This margin, in the case of the perceptrons, is used for measuring the mistake complexity. In SVMs, the optimization of this margin is justified by a rigorous statistical analysis. The SVMs use binary classification, i.e., input is always split into two classes. This approach is proved useful in a wide range of applications, which vary from text classifications, e.g., "Is this article related to my search query?" to bioinformatics, e.g., "Do these micro-array profiles indicate cancerous cells?"

The SVMs were originally used for problems in binary classification, where it was required to distinguish an object belonging to one of the two categories. For a long time, people attempted to solve such problems by simply reducing them to binary classification problems. However, these reductions failed to exploit about the structure of the predicted objects. But SVMs are capable of exploiting the structure also, e.g., the conventional SVMs can be applied for parts-of-speech tagging applications, which requires resolving the sense of each word in a sentence using the context of the word, i.e., surrounding text.

The support vector machines are falling in the category of *supervised learning models* based on associative learning algorithms that analyze the data and recognize patterns, hence they have applications in classification and regression analysis. After having trained by some training examples, each belonging to one of the two categories, a SVM training algorithm builds a model that assigns new examples into one of the two categories. Thus, an SVM is a *non-probabilistic linear* classifier. They are powerful approaches to predictive modeling with success in a number of applications, which include handwritten digit and alphabet recognition, face detection, text categorization, etc.

An SVM is a model where examples are nothing but representation of points in space, that are mapped into two classes, such that the examples of the separate categories are divided by a clear gap that should be as wide as possible. Once the learning has taken place, any amount of new examples are then mapped into that same space and predicted to belong to a category depending on which side of the gap they fall on.

The SVMs fit in the context of classification where the attribute of the objects whose value is to be predicted, called *dependent attribute*, has two possible values: 0 and 1. The classification is performed on a *surface* (in three or more dimensions) in the space of predictor attributes, that separate the points with dependent attribute $= 0$, from those with dependent attribute $= 1$ (or it may be -1 and $+1$ respectively). An optimal separating surface is computed by maximizing the margin of separation as shown in Fig. 14.2, where, data point with 0 attribute are labeled by circle ($\circ$s) and those with attribute 1 are labeled as squares ($\Box$s). The figure shows a separable problem in a 2D space. The margin of separation is the distance between the boundary

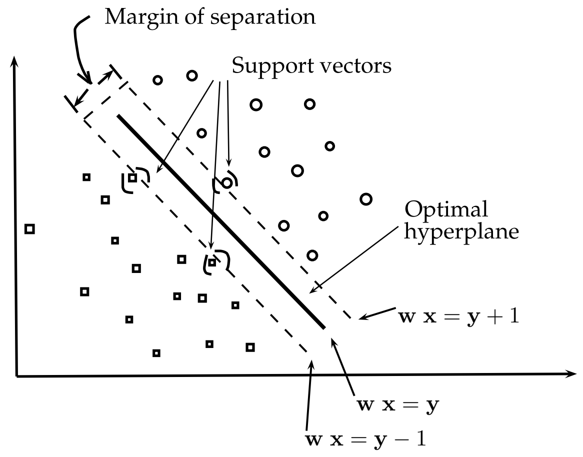

Margin of separation
Support vectors
Optimal hyperplane
$\mathbf{w}\,\mathbf{x} = \mathbf{y} + 1$
$\mathbf{w}\,\mathbf{x} = \mathbf{y}$
$\mathbf{w}\,\mathbf{x} = \mathbf{y} - 1$

Fig. 14.2 Classification through SVM

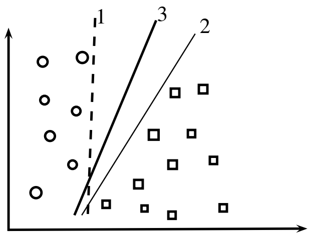


Fig. 14.3 A case of different margin in SVM

of points with dependent attribute = 0 and the boundary of those with dependent attribute = 1. The variables $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{w}'$ are vectors.

The margin (which is a hyperplane) is the measure of "safety" in separating the two sets of points, the larger is better. Figure 14.3 shows three separator lines between the set of data values of two categories, with separator line 3 producing the maximum margin. Computing the optimal separating surface, in a standard SVM formulation requires solving an optimization problem, which is quadratic in nature [2, 3].

14.3.1 Learning Pattern Recognition from Examples

The SVMs are based on *Analogical Learning*, which we discussed in the previous chapter. The Support Vector Machine (or Support Vector Network (SVN)) maps the input vectors into some high-dimensional feature space that we call $\mathbf{Z}$, through some nonlinear mapping, which is chosen a priori. A linear decision surface is constructed in this space, with special properties that ensure high generalization capability of the network. This approach gives rise to two problems: the first is conceptual in nature, while the other is technical. The conceptual problem is regarding how to find out a separating hyperplane that will generalize well in the presence of very high dimension of the feature space. The technical problem is about how to computationally treat such a high dimensionality space problem?

The conceptual part of this problem can be solved for the case of *optimal hyperplanes* for separable classes. An optimal hyperplane is a linear decision function with maximal margin between the vectors of two classes (see Fig. 14.2). To construct such a hyperplane, only a small amount of data is required, called *support vectors*, which determine the margin. The optimal hyperplanes should separate the training data without errors. The optimal hyperplane algorithm is described as follows.

The set of labeled training patterns,

$$(y_1, \mathbf{x}_1), \ldots, (y_\ell, \mathbf{x}_\ell), \quad y_i \in \{-1, 1\}, \tag{14.2}$$

are linearly separable if there exists a vector **w** and a scalar b (for bias) such that the following inequalities are valid for all elements of the training set given in Eq. (14.2).

$$\begin{aligned} \mathbf{w}.\mathbf{x}_i + b \geq 1 \quad &if \quad y_i = 1, \\ \mathbf{w}.\mathbf{x}_i + b \leq 1 \quad &if \quad y_i = -1. \end{aligned} \tag{14.3}$$

The inequalities of Eq. (14.3) can be rewritten as

$$y_i(\mathbf{w}.\mathbf{x_i} + b) \geq 1, \quad i = 1, \ldots, \ell. \tag{14.4}$$

The optimal hyperplane, which is the unique one separating the training data with a maximal margin, can be expressed by

$$\mathbf{w_0}.\mathbf{x} + b_0 = 0. \tag{14.5}$$

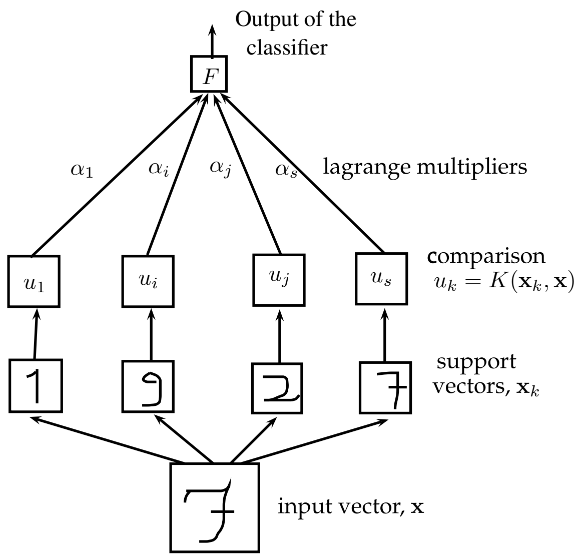
As an application of character recognition, Fig. 14.4 shows a classification of an unknown pattern (which appears like the decimal digit 7) using a support vector machine. A pattern in the input space is compared with support vectors $\mathbf{x}_k$, resulting in values u_1, u_i, u_j, u_s. The resulting values are nonlinearly transformed using *Lagrangian* multipliers. A linear function (F) of these transformed values determines the output of the classifier [5].

For the hyperplane expression (14.4), we use a standard optimization technique through which we construct the Lagrangian function,

$$L(\mathbf{w}, b, \Lambda) = \frac{1}{2}\mathbf{w}.\mathbf{w} - \sum_{i=1}^{\ell} \alpha_i [y_i(\mathbf{x}_i.\mathbf{w} + b) - 1], \tag{14.6}$$

where $\Lambda^T = (\alpha_1, \ldots, \alpha_\ell)$ is the vector of nonnegative Lagrange multipliers corresponding to the constraints.

Fig. 14.4 Classification of an unknown pattern



14.3.2 Maximum Margin Training Algorithm

As discussed above, in SVMs it is attempted to find out the support vectors with maximum margin. An algorithm for this purpose finds a decision function for n-dimension pattern vectors $\mathbf{x}$, which belong to either of the two classes, A or B. Input to the training algorithm is a set of k examples $\mathbf{x}_1$, ..., $\mathbf{x}_k$ with labels y_1, ..., y_k, respectively.

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_k, y_k) \tag{14.7}$$

$$\text{where} \quad \begin{cases} y_i = +1 & \text{if } \mathbf{x}_i \in \text{class } A \\ y_i = -1 & \text{if } \mathbf{x}_i \in \text{class } B. \end{cases} \tag{14.8}$$

Making use of these training examples during the learning phase, the SVM algorithm finds the decision functions $D(\mathbf{x})$, where $\mathbf{x}$ is an unknown pattern. Once the training phase is over, the class of unknown pattern is predicted using the following rule:

$$\begin{aligned} &\mathbf{x} \in A, \quad \text{if } D(\mathbf{x}) > 0 \\ &\mathbf{x} \in B, \quad \text{otherwise.} \end{aligned}$$

It is mandatory that the decision functions $D(\mathbf{x})$ are linear in their parameters, however they are not restricted to linear dependencies of $\mathbf{x}$. These functions can be expressed either in the direct space, or in dual space. The notation used for direct space is identical to the one used in perceptions, i.e.,

$$D(\mathbf{x}) = \sum_{i=1}^{k} w\varphi(\mathbf{x}_i) + b. \tag{14.9}$$

In the above equation, φ is a predefined functions of $\mathbf{x}$, and w and b are the adjustable parameters of the decision function. In the dual space, the decision functions are of the form,

$$D(\mathbf{x}) = \sum_{i=1}^{k} \alpha_i \, K(\mathbf{x}_i, \mathbf{x}) + b, \tag{14.10}$$

where coefficients α_i are the parameters to be adjusted, and $\mathbf{x}_i$ are the training patterns. The function K is a predefined kernel, for example, radial bias function or a potential function.

14.4 Predicting Structured Objects Using SVM

A potential drawback of SVMs and other statistical learning methods is that they treat the category structure as "flat"and do not consider the relationships between categories, which are commonly used for expressing the concepts as hierarchies or as taxonomies. The taxonomies or taxonomical structures offer clear advantages in supporting tasks like browsing, searching, or visualization. It is to realize that all the real-word concepts have complex hierarchical structures. For example, in browsing, the list of answers to a query can be taken as the children of the root (keyword). Further, each answer link would correspond to a web page, which may have many links, thus making a tree structure.

To better appreciate the above, consider the problem of natural language parsing as shown in Fig. 14.5, where the parser receives the input in the form of a sentence of natural language, and the output is a parser-tree as a decomposition of the sentence into its constituents. While support vector machines are used in NLP applications, such as Word Sense Disambiguation (WSD), parsing is not suited as a problem to be solved through SVMs for classification. This is because, in parsing, the output is not a classification of yes/no but a labeled tree, representing the structure of a sentence. So the question is, how can we take an SVM and learn predicting a parse-tree?

The question above arises not only for predicting the structured trees but also for a variety of other structures, like DNA sequence, or sequence ordering for image segments, etc.

Let us assume that there is multi-class SVM for document classification, where each document belongs to exactly one category. Let $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ be a set of n labeled training documents. Here, $\mathbf{x}_i \in \Re^d$ denotes a standard vector representation for the *i-th* training document, and there are total d number of such documents. Each label is y_i, where $y_i \in \mathcal{Y} \equiv \{1, \dots, k\}$, and k is the total number of categories.

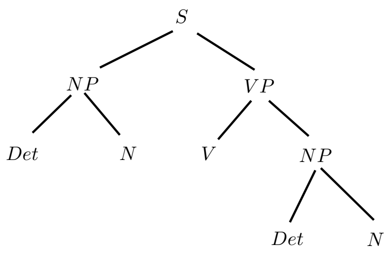


Fig. 14.5 Parse-tree for "The dog chased the thief"

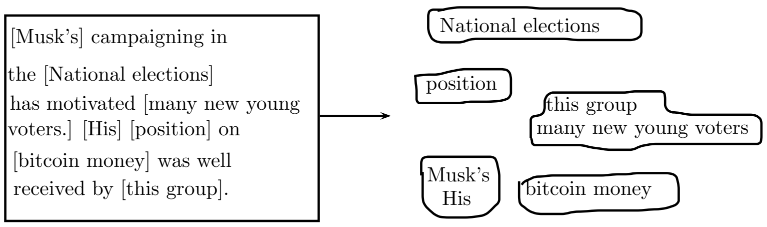

[Musk's] campaigning in the [National elections] has motivated [many new young voters.] [His] [position] on [bitcoin money] was well received by [this group].
National elections
position
this group
many new young voters
Musk's
His
bitcoin money

Fig. 14.6 Resolving the equivalent noun-phrase co-references

Let us assume that there is weight vector $\mathbf{w}_y$ for every class $1 \leq y \leq k$. We will refer to the stacked vector of all weights by $\mathbf{w} = (\mathbf{w}_1, \ldots, \mathbf{w}_k)$. The *structured output prediction* is the term used for this kind of prediction, where our objective is to learn a function $h : \mathscr{X} \rightarrow \mathscr{Y}$, that maps the inputs $\mathbf{x} \in \mathscr{X}$ to complex and structured output $y \in \mathscr{Y}$. In the structured output prediction, tasks range from predicting an equivalence relation, say, in noun-phrase co-reference resolution, to predicting a correct and well-formed machine translation.

Example 14.1 Resolving the equivalent noun-phrase co-references.

Consider a situation that a man called "Musk" contests in the elections and hence campaigns to seeks votes as a presidential candidate, and suggests election manifesto of promoting the bitcoin money if he wins the elections. Figure 14.6 shows an example of resolving noun-phrase *co-reference*, where there is an equivalence relation between a noun phrase (e.g., "Musk") and its co-reference "His", and similarly between "this group"and "many young voters".

The problems of this type exists elsewhere also, for example, in image segmentation, for determining an equivalence relation, say y, over a matrix of pixels $\mathbf{x}$, and the problem of web search to predict a document ranking y for a given query $\mathbf{x}$. The *structural SVMs* (SSVMs) are suitable for use, as well as to address these all, and a large range of prediction tasks with structured output. □

14.5 Working of Structural SVMs

How to map the input to some structured output? Basically, a task of prediction of a structure is similar to a task of *multi-class learning*. Each possible parse-tree $y \in \mathscr{Y}$ may correspond to one class, and classifying a new example $\mathbf{x}$ is nothing but predicting its correct *class* out of many possible classes, as shown in Fig. 14.7.

However, the problem is that there is a very large number of classes $\mathscr{Y}$. In case of parsing, the number of possible parse-trees are exponential as a function of the length of the sentence. This situation is similar for most other prediction problems, which output some structure. Hence, there is need for finding a *compact* representation of output spaces, which are large in size. In addition, a *single prediction* for an example is computationally challenging, this requires the enumeration of all the outputs. The third challenge is, how to distinguish between two wrong parse-trees, where one may be closer to the correct one. This, we call as *prediction error* [9, 11].
We can derive a structural SVM from a multi-class SVM, such that for each class y, the multi-class SVMs use a weight vector $\mathbf{w}_y$, and each input $\mathbf{x}$ has a score for each class y via the function,

$$f(\mathbf{x}, y) \equiv \mathbf{w}_y.\Phi(\mathbf{x}), \tag{14.11}$$

where Φ is a function that extracts the vector $\Phi(\mathbf{x})$ of binary or numeric features, from $\mathbf{x}$. Hence, every feature has additive-weighted influence in the modeled compatibility between inputs $\mathbf{x}$ and classes y. For the classification of $\mathbf{x}$, a prediction rule $h(\mathbf{x})$ chooses simply the highest-scoring class as the predicted output. This is expressed by

$$h(\mathbf{x}) \equiv argmax_{y\in\mathscr{Y}} f(\mathbf{x}, y). \tag{14.12}$$

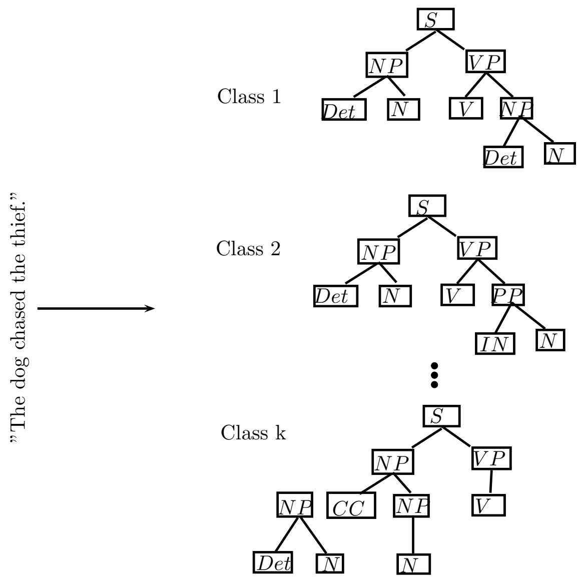


Fig. 14.7 Resolving to correct structure through *structural SVM*

This will result in a correct prediction of output y for input $\mathbf{x}$, provided that the weights $\mathbf{w} = (\mathbf{w}_1, \ldots, \mathbf{w}_k)$ are chosen such that the inequalities $f(\mathbf{x}, \bar{y}) < f(\mathbf{x}, y)$ hold true for all incorrect outputs $\bar{y} \neq y$. For a given training sample $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$, this leads directly to a margin (a hard-margin) formulation of the learning problem by requiring a fixed margin (= 1) separation of all training examples, while using the norm of $\mathbf{w}$ as a regularizer:

$$min_w \ \frac{1}{2}\|\mathbf{w}\|^2, \ \text{such that} f(\mathbf{x}_i, y_i) - f(\mathbf{x}_i, \bar{y}) \geq 1, \ (\forall i, \bar{y} \neq y_i). \tag{14.13}$$

For a k-class problem, the optimization problem has a total $n(k-1)$ inequalities, that are linear in $\mathbf{w}$. This is because, one can expand

$$f(\mathbf{x}_i, y_i) - f(\mathbf{x}_i, \bar{y}) = (\mathbf{w}_{yi} - \mathbf{w}_{\bar{y}}).\varPhi(\mathbf{x}_i), \tag{14.14}$$

which is a convex quadratic program.

The drawback of Eq. (14.13) for structured output is there is a generalization across the inputs $\mathbf{x}$, but output is without generalization due to a separate weight vector $\mathbf{w}_y$ for each class y. It is therefore not advisable to reduce the output prediction to multi-class classification, as the number of possible outputs are likely to become very large. This problem is solved using a new function $\varPsi(\mathbf{x}, y)$ in place of $\varPhi(\mathbf{x}, y)$ to extract features from input–output pairs. This new function is called as *joint feature map*. This will yield compatibility functions due to contributions from the combined properties of both inputs and outputs. Since the compatibility functions is defined via $f(\mathbf{x}, y) \equiv \mathbf{w}.\varPsi(\mathbf{x}, y)$, the number of parameters simply will be equal to the number of features extracted via $\varPsi$, and that may not depend on the number of classes $|\mathscr{Y}|$.

14.6 k-Nearest Neighbor Method

The nearest neighbor method is a *statistical* learning method, also called sometimes as *memory-based method*. The method is used for clustering of objects based on some similarity in their attributes. The k-nearest neighbor (k-NN) or simply nearest neighbor method is an *analogical* type of learning. Given a training set θ of N number of labeled patterns, each with n attributes, a nearest neighbor algorithm decides that some new pattern $\mathbf{x}$ belongs to the same category to which its closest neighbors in θ belong. In other words, a k-nearest neighbor algorithm assigns a new pattern, $\mathbf{x}$, to that category $\mathbf{x_i}$ to which the majority of its k-closest neighbors belong.

Example 14.2 Nearest Neighbor.

Consider there are five cars manufactured by company A, and for every car ten important attributes have been identified, like engine size, color, wheel size, fuel used, etc. Then, training set size is $N = 5$, and attribute size of each training set is $n = 10$. Let a new model is introduced by company B, and the nearest neighbors' size is taken as $k = 4$. In this situation, a car from the company A having four attributes common to the new car is, say model-Zeta. Therefore, Zeta being the nearest neighbor to the new model from company B, the new model is put in the class of model-Zeta. □

If k is relatively large, there are less chances of the decision going wrong due to noisy training pattern close to $\mathbf{x}$. However, the large values of k also reduce the sharpness or acuteness of this method. The k-NN method can be thought of as estimating the values of the probabilities of belongingness to class, given an input pattern $\mathbf{x}$.

The k-NN approach has been shown to be a useful non-parameteric technique for *regression* and *classification*. In both cases, the input comprises the k-closest neighbors of the vector. However, finding the k-NNs for a test sample, among N design samples, is a time-consuming process, particularly for large N. The reordering of these samples requires $N(N-1)/2$ pairwise distance computations, which is a time-consuming process for large values of N.

When used for object classification, the k-NN algorithm has many practical applications, e.g., in the areas of artificial intelligence, pattern recognition, statistics, cognitive psychology, vision analysis, and medicine, to name a few. The decision rule in k-NN provides a simple non-parameteric procedure for the assignment of a class label to the input pattern based on the class labels represented by the k-closest (for example, in terms of Euclidean distance) neighbors of the vector.

14.6.1 k-NN Search Algorithm

Let us assume that we are given N design samples, $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, each of which is n-dimensional sample. With this, it is required to compute the k-nearest neighbors ($k < n$) in a test sample $\mathbf{y}$, as measured by an appropriate distance function $d(\mathbf{y}, \mathbf{x}_i)$ [7].

There is no need of preprocessing of the labeled sample set in the nearest neighbor classifier. The nearest neighbor classification rule assigns an input sample vector $\mathbf{y}$ of unknown classification, to a class which is THE nearest neighbor to it (see Algorithm 14.1). The idea of the nearest neighbor can be extended to k-nearest neighbors with the vectors $\mathbf{y}$ being assigned to the class that is represented by a majority among the k-nearest neighbors. In the algorithm, d_k stands for distance from $\mathbf{y}$ to $\mathbf{x}_k$, for k-nearest neighbor.

When more than one neighbor is taken into account, it is likely that there may be a tie among the qualifying classes, which corresponds to maximum number of neighbors in the group of k-nearest neighbors. There is a simple way to handle this problem by restricting the possible values of k. For example, if there is two-class problem, and k is restricted to odd values only, no tie can occur.

Occurrence of a tie can be handled using the following approach. An unclassified (i.e., sample) vector is assigned to the class, of those labels that are tied, and for that class the sum of distances from the sample to each neighbor in the class is a minimum. There are chances that there may still remain a tie.

Algorithm 14.1 K-Nearest Neighbor Algorithm

```
1: Let $W = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ //set of $n$ labeled samples.
2: Input $\mathbf{y}$ //whose class is to be determined
3: Set $k$ to any value in 1...$n$
4: Let $S = \phi$ //initial set of k-nearest neighbors (for given $k$)
5: for $i$ = 1 to $n$ do
6:   compute distance $d_i(\mathbf{y}, \mathbf{x}_i)$ from $\mathbf{y}$ to $\mathbf{x}_i$
7:   if $(d_i \le d_k)$ then
8:     $S = S \cup \{\mathbf{x}_i\}$
9:   else
10:    if $\mathbf{x}_i$ is more close to $\mathbf{y}$ than some previous neighbor then
11:      Delete farthest in the $S$
12:      $S = S \cup \{\mathbf{x}_i\}$
13:    end if
14:  end if
15: end for
16: Determine class in $S$ that is in majority
17: if (there is tie) then
18:   Compute the sum of distances of neighbors in each class with the one having tie
19:   if (there is no tie) then
20:     Classify $\mathbf{y}$ into minimum found class
21:   end if
22: else
23:   Classify $\mathbf{y}$ into the class of majority
24: end if
25: End
```

For numerical attributes, the distance metric used in the nearest neighbor is *Euclidean distance*. Considering two patterns, $\mathbf{x}_1 = x_{1_1}, x_{1_2}, \dots, x_{1_m}$, and $\mathbf{x}_2 = x_{2_1}, x_{2_2}, \dots, x_{2_m}$, distance between two using Euclidean is given by

$$d(\mathbf{x}_1, \mathbf{x}_2) = \sqrt{\sum_{j=1}^{m} (x_{1_j} - x_{2_j})^2}. \tag{14.15}$$

To keep the spread of attribute values along each dimension approximately same, the above value of the distance is usually modified by scaling of the features. In that case the distance is represented by the following equation, where a_j^2 is the scale factor for the dimension j.

$$d(\mathbf{x_1}, \mathbf{x_2}) = \sqrt{\sum_{j=1}^{m} a_j^2 (x_{1_j} - x_{2_j})^2}. \tag{14.16}$$

Example 14.3 Finding the nearest neighbor with k patterns.

Figure 14.8 shows a problem of k-nearest neighbor classification, where $k = 6$. There are two patterns of each categories 1, 2, and 3, with numbers 1, 2, 3 marked after each pattern symbol, respectively. A collection of dots makes a pattern. For example,

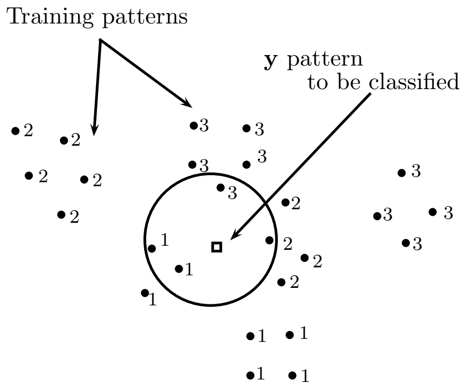


Fig. 14.8 k-nearest neighbor problem

in the two patterns of category 1, there are three and four dots, each followed by 1s. The pattern $\mathbf{y}$, represented by a box, is to be classified. In the circle enclosing, since the majority are in the category of 1, the new pattern $\mathbf{y}$ should be classified in category 1. □

14.7 Naive Bayes Classifiers

The Bayesian networks are powerful tools for decision-making and reasoning under uncertainty. These networks are specified by two components:

– *Graphical component*. It expresses uncertainty of causal relations, and consists of Directed Acyclic Graph (DAG) to represent causal relations and conditional probabilities of each node, given the parent of each. The vertices represent events and edges which are the relations between them.
– *Numerical component*. It quantifies different links in the DAG through conditional probability distribution of each node in the contexts of its parents.

Basically, the Bayes theorem permits optimal prediction of a *class* of new instances, given a vector $\mathbf{x} = \{x_1, ..., x_n\}$, of attribute's values. Since there is always insufficient training data to obtain accurate prediction of full joint probability distribution, the straight forward application of Bayes theorem is impracticable for machine learning. Therefore, there is need for making assumptions of independence to make the inference feasible.

The Bayesian classifier, called Naive Bayes, i.e., "straw man", approach takes this to the extreme when it is assumed that attributes are statistically independent, given the value of the class attribute. However, the above assumption never holds true, but still the Naive Bayes performs exceptionally well in most classification problems. Apart from this, the Naive Bayes approach is computationally efficient, as the training is linear in both the number of instances and attributes, and simple to implement [4].

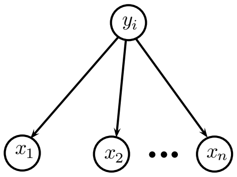


Fig. 14.9 Naive Bayes network structure

A Naive Bayes is composed of a DAG with one root node (called parent), which is an unobserved node representing *class* of each object to which the testing set belongs, and several leaf nodes, corresponding to observed nodes, with strong assumption of independence among these leaf nodes in the contexts of their common parent. The leaf nodes represent different *attribute* (*features*) specifying this object. Note that the Naive Bayes are composed of *two levels* only as shown in Fig. 14.9, whereas a Bayesian network may consist of many levels.

The decision of a Bayesian classifier is represented as a matrix of $P(x_j|y_k)$, which specifies the probability of occurrence of each feature value $(x_1, \dots, x_j, \dots, x_n)$ given each class y_k. To classify a new example having features among $x_1 \dots x_n$, we make use of Bayes theorem as

$$argmax_{y_i}\ P(y_i|\bigwedge x_j) = argmax_{y_i}\ \frac{P(\bigwedge x_j|y_i)P(y_i)}{\sum_k P(\bigwedge x_j|y_k)P(y_k)} \tag{14.17}$$

that computes $P(y_i|\bigwedge x_j)$, which is the probability of the example in class y_i given the features x_j. The subexpression $\bigwedge x_j$ denotes a conjunct of attribute values all occurring in an example. The summation is performed over N $(1 \le k \le N)$ classes, and the above probability is calculated for each class y_i, and then the class of the highest probability is selected. The probability $P(y_k)$ is estimated from the distribution of the training examples among classes. If independence of all attributes is assumed under a given context (i.e., class), then $P(\bigwedge x_j|y_k)$ can be calculated using

$$P(\bigwedge x_j|y_k) = \prod_j P(x_j|y_k). \tag{14.18}$$

The values $P(x_j|y_k)$ are calculated from the probability matrix. Here, x_j is the evidence on attribute nodes, which can be dispatched into pitches of n evidence of features, $x_1, \dots, x_j, \dots, x_n$. Because Naive Bayes is working on the assumption that these attributes are independent of each other (given the parent node), their combined probability is obtained by substituting Eq. (14.18) into (14.17), which results in,

$$\begin{aligned} argmax_{y_i}\ P(y_i|\bigwedge x_j) &= argmax_{y_i}\ \frac{\prod_j P(x_j|y_i)P(y_i)}{\sum_k [\prod_j P(x_j|y_k)P(y_k)]} \\ &= argmax_{y_i} \prod_j P(x_j|y_i)P(y_i). \end{aligned} \tag{14.19}$$

Note that, there is no need to explicitly compute the denominator $\sum_k [\prod_j P(x_j | y_k)P(y_k)]$, since it is common among all the computation being a normalization constant. In some practice, *log* is computed instead of a product, because probabilities involved can be very small.

The Bayesian learning method, as a classifier, builds the matrix $P(x_j|y_k)$ from training examples, by examining the frequency values in each class. It is possible to compute this matrix incrementally by incorporating one instance at a time. Alternatively, it can be constructed (i.e., without incremental approach), using all the data at a time.

Due to its simple structure, the Naive Bayes has many advantages. It is efficient, as the inference (i.e., classification) is achieved in a *linear time*. However, the Bayes network with general structure has complexity of *NP-complete*. Naive Bayes construction is incremental, in the sense that it can be easily updated, e.g., addition of new cases. The major problem with Naive Bayes is the assumption of strong independence relation, i.e., assumption of independence of features in the context of session class is not always true, and leads to negative influence on the inferred results.

Naive Bayes is most commonly used in text classification, where words are features, and presence/absence of a word can be used to determine the topic of a document.

Given N number of training examples, each with n number of attributes, complexity of generating probability matrix for Bayes classifier is $O(n.N)$. Hence, the classifier is substantially faster as the runtime is independent of the decision "rule"generated. Apart from this, the basic operation is performed only once.

14.8 Artificial Neural Networks

The science of machine learning is mostly experimental as there is a no universal learning algorithm existing yet. That is clear from the fact that, given a number of tasks, none can make a computer to learn every task well. A knowledge-acquisition algorithm is always required to be tested on learning tasks and data, that are specific to a given situation, and it is irrespective of whether it is recognizing a sunrise or it is doing a language translation. There is no method to prove that the given algorithm will be consistently better for all the situations. However, the human behavior apparently contradicts. We are fairly good at general learning abilities due to which we are able to master a number of tasks, like playing chess and playing cards. These arguments suggest and might serve as inspirations for building machines with some form of general intelligence. Therefore, the use of Artificial Neural Networks (ANN), which is a brain model, appears to be a logical justification for building intelligence systems [8].

The basic unit of the brain for performing the computation is a cell, called *neuron*; each one of them sends a signal to other neurons through very small gaps between the cells, called *synaptic clefts*. The property of any neuron of sending a signal through this gap, and the amplitude of the signal together, is called as *synaptic strength*. As a neuron learns enough, its synaptic strength increases, and in that situation, if it is stimulated by an electrical impulse, there are better chances that it would send messages to its neighboring neurons.

The current neural-based learning algorithms need close involvement of humans, for producing better results. Majority of these algorithms are based on *supervised learning*, where each training example is carried out using human-crafted labels, about what is being learned. Consider an example of a picture of sunrise associated with the caption: "Sunrise". In that instance, the goal of the learning algorithm is to take the photograph as an input, and produce output, the name of the object in the image, i.e., "sunrise". We know that the mathematical process of transforming an input to the output is a *function*. The synaptic strength, which is a numerical value, produces this function, which corresponds to the solution to the learning through ANN.

It is interesting to note that it can be achieved through rote learning also, but it would not be useful. In fact, when we want to teach to the algorithm "what the sunrise is", then to have the algorithm recognize any sunrise, even the one for which we have not trained! This is the ultimate goal of machine learning algorithm.

The Artificial Neural Networks (ANN) possess the following important properties:

Learning ability,
Massive parallelism,
Adaptability,
Fault tolerance,
Distributed representation and computation,
Generalization ability, and
Low energy consumption.

which make them candidate for many applications. Although the details of the proposals vary, the most common models for learning and computation take the *neuron* as the basic processing unit. Each processing unit is characterized by the following:

- an activity level to represent polarization state of a neuron,
- an output value to represent firing rate of the neuron,
- a set of input connections,
- *synapses* on the cell and its dendrite,
- a bias value to represent an internal resting level of a neuron, and
- a set of output connections to represent a neuron's *axonal* projections.

Each of these aspects of the unit are represented mathematically by real numbers.

Hence, each connection of a neuron has an associated weight, called *synaptic strength*, which influences the effect of the incoming input on the activity of the unit. This weight is either positive, called *excitatory* or, negative, called *inhibitory*.

The basic model of artificial neuron with binary threshold is shown in Fig. 14.10. The mathematical neuron computes the weight as the sum of its n number of input

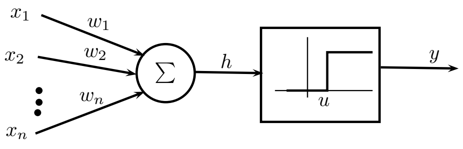


Fig. 14.10 Basic model of an artificial neuron

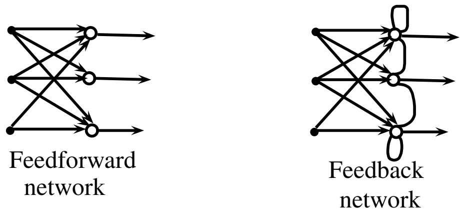


Fig. 14.11 Single-layer feedforward and feedback networks

signals $x_1, \ldots, x_n$, and the generated output is 1 if the sum is above some threshold value u, otherwise the output is zero. This can be represented by

$$y = \theta\big(\sum_{i=1}^{n} w_i x_i - u\big) \tag{14.20}$$

where $\theta(.)$ is called a unit step function at 0, and w_i is the *synapse* weight associated with the *ith* input. For the sake of simplicity, we consider the threshold u as another weight, $w_0 = -u$ attached to the neuron with a constant input $x_0 = 1$. A properly chosen weight allows a synchronous arrangement of such neurons to perform universal computations. There is a crude analogy of this neuron model to biological neurons as follows: the wires and interconnections model the *axons* and *dendrites*, respectively, in the biological neuron; connection weights in this model correspond to *synapses* in biological neuron; and threshold function approximates the activity in *soma*. However, this model is an overly simplified one of a true biological neuron.

The ANNs can be considered as weighted directed graphs, where artificial neurons act as nodes, and directed edges with weights are connections between neurons, and between outputs and inputs of neuron.

Based on the connection pattern an ANN can be classified as

1. *Feedforward networks*: In these, the direct graphs have no loops, and
2. *Recurrent feedback networks*: There are loops because of feedback connections.

Figure 14.11 shows the single-layer feedforward and feedback neural networks. The most common family of *feedforward networks* is the *multilayer perceptron*. The neurons in these networks are organized into layers with unidirectional connections between them. Usually, the feedforward networks are of a *static* type, as they produce only one set of output values instead of a sequence of values for a given input. These networks are without memory, as their output is independent of the previous states of the network.

The feedback networks are called *recurrent networks*, and are dynamic systems. Neurons' output patterns are computed when a new input pattern is presented to these networks. Due to feedback paths, the input to each neuron is then modified, which causes the network to enter a new state.

The problem of *learning* in neural networks is simply the problem of finding a set of connection strengths which allows the network to carry out the desired computation [10].

An ANN usually learns of the connection weights from the available training patterns, the performance of which gets improved over time through iterative updating of the weights. The ability of ANNs to automatically learn from examples makes them attractive and exciting. From a given collection of representative examples, the ANNs learn underlying rules as a form of input–output relationship, instead of following the rules specified by human experts. This simple factor is a major advantage of neural networks over traditional experts systems.

To understand this learning process, first there is need of a model of learning environment in which these neural networks operate. That means, we must know about what information is available to the network. This model is called the *learning paradigm*. The second requirement is to know about how these weights are updated. This means to know, as to which rule controls the updating process of the weight. In fact, a learning algorithm refers to some procedure which uses learning rules to adjust the weights of inputs.

All the three paradigms of learning exist in the neural networks, i.e., *supervised learning*, *unsupervised learning*, and *hybrid learning*. Supervised learning requires a teacher, where the network is provided with correct answers, i.e., output, for every input pattern. The weights are so determined that they allow the network to produce answers, which are as close as possible to known correct answers.

The unsupervised learning-based neural networks explores the underlying structure in the data, or correlations between patterns in the data, and organizes patterns into categories based on the correlations.

A hybrid learning ANN combines both the supervised and unsupervised approaches.

The learning rules in the neural networks are of four basic types. These are *error correction*, *Boltzmann*, *Hebbian*, and *Competitive learning*. Their basic principles are presented in the following.

14.8.1 Error-Correction Rules

A learning based on error-correction rules makes use of simple concepts. During the training, the input $x_1, \ldots, x_n$ is applied to the network, and the flows through the network generates a set of values in the units of output y. As the next step, the actual output y is compared with the desired target d. If the output and the target match, no change is made to weights. If there is no match, change is made to weights of some of the connections. The problem is to find out as which connections in the network were at fault that caused the error and resulting in a mismatch. Obviously, it is *supervised learning* paradigm. The principle used for error-correction learning rules is based on the error signal $(d - y)$, to modify the connection weights so as to gradually reduce the error magnitude.

The *perception-based learning* works on this principle of error correction. A *perceptron* comprises a single neuron with adjustable weights, w_i, $i = 1, \dots, n$, and a threshold value u, as shown in Fig. 14.10 (page no. 432). Given an input vector $\mathbf{x} = (x_1, \dots, x_n)^t$, where t is the iteration number, net input to the neuron is expressed as

$$v = \sum_{i=1}^{n} w_i x_i - u. \tag{14.21}$$

If $v > 0$ the output y of the perceptron is $+1$, otherwise it is 0. In a *classification* problem (of two classes, say A and B), the perceptron assigns an input pattern to class A if $y = 1$, and to class B if $y = 0$.

Algorithm 14.2 Perceptron learning Algorithm

```
1: Initialize weights $w_1, ..., w_n$, and threshold $u$ to some small random numbers,
2: Apply a small pattern input vector $(x_1, \dots, x_n)^t$ and evaluate the output of the neuron, as per
   equation (14.21),
3: Update each weight $w_i$ $(i = 1, ..., n)$ according to: $w_i(t + 1) = w_i(t) + \eta(d - y)x_i$.
```

Algorithm 14.2 is *backpropagation* learning algorithm, based on *error-correction principle*, where d is the desired output, t is the iteration number, and η $(0.0 < \eta < 1.0)$ is the gain, i.e., size of the step.

14.8.2 Boltzmann Learning

The learning in ANN based on Boltzmann machines has many properties: they are symmetric and recurrent (i.e., feedback) networks. It consists of binary units (+1 for "on", and −1 for "off"). The symmetric property means, weight on the connection from unit number i to unit j is identical to weight from unit j to unit i, formally, $w_{ij} = w_{ji}$. The neurons are also of two types: 1. *Visible neurons* are a subset of the entire set of neurons, and they interact with the environs, and 2. *Hidden neurons* are the remaining neurons, which do not interact. Each neuron is a stochastic unit, which generates and outputs (or it is a state) according to the Boltzmann distribution of statistical machines.

A Boltzmann machine also operates in one of two modes: 1. *Clamped* mode, where visible neurons are clamped onto a specific state determined by the environment; and 2. *Free-running* mode, in which both the visible and hidden neurons are allowed to operate freely.

The Boltzmann learning algorithm works to adjust the connection weights in such a way that the desired probability distribution is satisfied by the states of the visible units. According to the rule of Boltzmann learning, the change in the connection weight w_{ij} is given by

$$\Delta w_{ij} = \eta(\overline{\rho}_{ij} - \rho_{ij}), \tag{14.22}$$

where η is learning rate, $\overline{\rho}_{ij}$ and ρ_{ij} are correlations between the states of units i and j when the network operates in the clamped mode and free-running mode, respectively. The values of $\overline{\rho}_{ij}$ and ρ_{ij} are estimates using Monte Carlo experiments [8].

14.8.3 Hebbian Rule

The Hebbian learning rule specifies the magnitude of the weight by which the connection between two units is increased/decreased in proportion to the product of their activation. It builds on the Hebbs's learning rule, which states that the connections between two neurons might be strengthened if the corresponding neurons fire simultaneously. This rule works well as long as the input patterns are uncorrelated, however, this condition places serious limitations on the Hebbian learning rule. A Hebbian rule for ANN is described as

$$w_{ij}(t+1) = w_{ij}(t) + \eta y_j(t)\, x_i(t), \tag{14.23}$$

where x_i, y_j are the output values of neurons i and j, respectively. These are connected by the weight w_{ij}, and η represents the learning rate. An important property of this approach is that learning is carried out locally, that is, the change in synapse weight depends only on the activities of two neurons connected by it. This simplifies the implementation of the circuit.

A more powerful learning rule is the delta rule, that utilizes the discrepancy between the desired and actual output of each output unit to change the weights feeding into it.

14.8.4 Competitive Learning Rules

The competitive-learning units compete among themselves for activation, which is in contrast to the Hebbian learning, where multiple output units can be fired together. Hence, only one output unit is active at any given time. The biological neurons follow this type of learning.

The competitive learning often categorizes or clusters the input data, where similar patterns are grouped by the network and represented by a single unit. The grouping is carried out automatically, which is actually based on the correlations.

The simplest possible competitive learning network has a single layer of output units, as shown in Fig. 14.11, where each output unit j connects to all the input units x_is, through the weights w_{ij}, $i = 1, \dots, n$. Each output unit also connects to all other units via inhibitory weights, but has a self feedback with an excitatory weight. Due to the competition, only the unit i with the largest net input becomes the winner. This can be expressed by

$$\forall i \ \mathbf{w}_i^* \ \mathbf{x} \geq \mathbf{w}_i \ \mathbf{x}. \tag{14.24}$$

In this learning, only the weights of the winner unit gets updated.

14.8.5 Deep Learning

Beginning from 2005, deep learning—a neural net-based approach, which is driven for its inspiration from brain science—began to come into its own, and has now become a singular force propelling AI research forward.

The deep learning is concerned with simulation of ANNs that "learn" gradually, in the areas of image processing, speech recognition, and understanding, and even to make the decisions of their own. The basic technique relies on ANNs, which do not precisely mimic as to how the actual neurons are working. Instead of this they are based on the general principles of mathematics that allow them to learn from examples to recognize people or objects in a photograph, and translate the spoken language from one to another. The technologies based on deep learning have transformed the AI research, and have produced far accurate results in speech recognition, computer vision, natural language processing, and robotics.

To be successful in generalizing after observing a number of examples, deep learning network needs more than just the examples. For example, it depends on hypotheses about the data and assumptions about what can be a possible solution for a particular problem. A typical hypothesis that can be built into a system might conclude that if data input for a particular function in two situations are almost similar, the output should not change drastically. For example, on altering a few pixels in an image of a dog will not transform it into a picture of cat.

One type of a neural network that consists of hypotheses about images is called *convolutional* neural network. These networks when used in deep learning have many layers of neurons, which are organized in such a way that the output is less sensitive to the deviation from the original object, due to changes in the input image. For example, we will note the changes in a face when viewed from different angles, however we will still recognize it correctly. A well-trained deep learning network will also do a similar job.

The design of convolutional networks take their inspirations from multilayered structure of *visual cortex*—part of the brain that receives input from eyes. Too many layers of virtual neurons in a convolutional neural network are what that make the network "deep", and hence it is better able to learn about the world about it.

14.9 Instance-Based Learning

Instance-based learning (IBL) approaches uses supervised learning techniques. There are several variants of IBL, e.g., *exemplar-based learning*, *case-based reasoning*, and *memory-based learning*. Though all these methods emphasize somewhat different aspects, all these approaches are founded on the basic concept of an *instance* or a *case*, as a basis of knowledge representation and reasoning. The meaning of *case* here is, observation or example, or incident, which is a single experience, e.g., a pattern, along with its solution, is a problem of pattern recognition.

In general, a problem along with its solution is a case-based reasoning. To highlight the main properties of IBL, it is important to understand its difference with *model-based* learning. As a typical case, IBL methods learn by simply storing some of the observed examples. The processing of these inputs are differed until a prediction or some other query is actually requested. Later, predictions are derived by combining information from stored examples, in some way. Once the query is answered, the prediction and intermediate results are discarded.

In contrast, the *model-based* or *inductive-based* learning derive predictions in an indirect way as follows: as a first step, observed data is used in order to induce a model, say, a *decision tree* or a *regression function*. As a second step, the predictions are obtained using this model, which can also serve as the function of explaining.

Generally, the model-based algorithms, also called *eager algorithms*, carry higher complexity during the training phase than the instance-based algorithms. The latter are also called *lazy* algorithms, where learning is basically storing of selected algorithms. The lazy methods also need more storage requirements, in a linear order of the size of the input, and higher computational cost compared to deriving of a prediction.

The IBL algorithms make use of specific instances, instead of pre-compiled abstractions during the prediction. They also describe the probabilistic concepts, because they use *similarity* functions to yield graded matches between instances.

The IBL algorithms are derived from the nearest neighbor (NN) pattern classifiers, which also do not save and use only selected instances to generate the classification predictions. Thus, they are also called as edited NN algorithms. They also maintain the prefect consistency with the initial training set.

14.9.1 Learning Task

The instance-based learning makes use of the supervised approach, or learning from examples. The input is a sequence of instances, where each instance is a set of n attribute–value pair, thus creating an n-dimensional instance space. Only one of these attributes corresponds to a category attribute, and the other attributes are predictor attributes. A category is a set of all instances in the instance space which have the same value for their category attribute. For the sake of simplicity, it is assumed that categories are disjoint.

The main output of IBL algorithms is *concept* or *concept description*. The algorithm is a function that maps instances to categories, i.e., for a given instance from a instance space, it produces the classification—a predicted value for instance's category attribute.

An instance-based concept description comprises a set of stored instances and, some information about their past performances, e.g., the number of correct and incorrect classification predictions. The set of instances can change once each training instance has been processed. The concept descriptions are decided based on how the selected similarities and classification functions of the IBL algorithm make use of the current set of saved instances. In all the IBL algorithms, at least two of the following three components constitute these functions [1].

Function of Similarity

This function computes the similarity between two instances: one is a training instance, and the other is in the concept description. The similarities are represented in numeric values.

Classification Function

The input to this function are 1. similarity function's result, and 2. classification performance records of instances in the concept description, which produces output of the classification.

Concept Description Updater

An updater keeps records of classification performance and resolves as to which instance to include in the concept description. Inputs to the concept description updater are training instance, similarity results, classification results, and current concept description. The output of the updater is the modified concept description.

The first two functions in the above decide how the saved instances in the concept description can be used to predict the category attributes. Thus, concept description in IBL not only comprises a set of instances, but also these two functions.

There is an important difference between IBL algorithms and most other supervised learning methods: the IBL algorithms do not construct explicit abstractions, like decision trees or rules. Most of the learning algorithms produce generalizations using the instances, and use simple matching procedures to classify the instances when presented in future. This eliminates the need of storing rigid generalizations of concept descriptions for IBL algorithms.

14.9.2 IBL Algorithm

Algorithm 14.3 is the simple instance-based learning algorithm. The similarity function, $sim(x, y)$, used in the algorithm is expressed by

$$sim(x, y) = -\sqrt{\sum_{i=1}^{n} f(x_i, y_i)}. \tag{14.25}$$

To compute the similarity, the instances have n attributes. For numerical attributes, the relation used is

$$f(x_i, y_i) = (x_i, y_i)^2. \tag{14.26}$$

For attributes with Boolean and symbolic values, $f(x_i, y_i) = (x_i \neq y_i)$, i.e., f is false when $x_i \neq y_i$. The missing attribute values are taken as having maximum difference from the value present. If both are missing, then $f(x_i, y_i) = 1$. IBL Algorithm 14.3 is very similar to the nearest neighbor algorithm (see page no. 425) we have studied earlier. The only difference in the IBL algorithm is that it normalizes its attributes' ranges, processes instances incrementally, and has a simple policy that allows missing values of attributes [1].

It is important to note that this algorithm's concept description changes over time. In the k-NN algorithm, the classification function simply assigns classifications according to the nearest neighbor policy. In the IBL algorithm, we can find out instances in the instance space that will be classified by each of the stored instances. The term *CD* in the algorithm stands for *concept description*, and *TS* is the *training set*.

Algorithm 14.3 IBL algorithm

```
1: Initialize: $CD = \phi$
2: for every $x \in TS$ do
3:    for every $y \in CD$ do
4:       $SIM[y] = sim(x, y)$
5:    end for
6:    $y_{max} = \exists\, y \in CD$ with maximal $SIM[y]$
7:    if $class(x) = class(y_{max})$ then
8:       classification = True
9:    else
10:      Classification = False
11:   end if
12:   $CD = CD \cup \{x\}$
13: end for
14: End
```

14.10 Summary

Machine learning systems help the programs automatically learn from the data; it can be said to be a kind of a data mining. Machine learning is used in computer science and beyond, e.g., in search engines, spam filters, advertisements, credit scoring, fraud detection, stock trading, drug design, and in many other applications.

A learning algorithm can be expressed as comprising three parts:

1. *Representation*,
2. *Evaluation*, and
3. *Optimization*.

The popular approaches of statistical machine learning are *support vector machine*, *k-nearest neighbor* algorithm, *Naive Bayes*, and *instance-based learning*.

The support vector machines (SVMs) are *supervised learning models* which use associative learning algorithms, which can analyze the data and recognize patterns. Thus they have applications in classification and regression analysis.

An SVM is a model where examples are nothing but a representation of points in space, which are mapped to two classes, so that the examples of the separate categories are divided by a clear gap that is as wide as possible. The latter is to help in making clear distinctions between the categories. New examples are then mapped into that same space and predicted about their category based on which side of the gap they fall on. The support vector machine (or support vector network (SVN)) maps the input vectors into some high-dimensional feature space through some nonlinear mapping chosen a priori. A limitation of SVMs and other statistical learning methods is that the category structure is treated as "flat"and that they do not consider any relationships between categories, which are commonly expressed in concept *hierarchies* or *taxonomies*. The *structured SVMs* are capable of learning the taxonomical architectures.

The problem with taxonomical structures is that the number of classes are very large. For example, in parsing, the number of possible parse-trees is the exponential factor of the length of the sentence, and this scenario is similar for a majority of other problems that are designed to predict the output, which is structured in nature. Thus, there is need for exploring a more compact representation for these large output spaces.

The k- nearest neighbor method is a *statistical* method, where, given a training set θ of n labeled patterns, a nearest neighbor algorithm decides that some new pattern, $\mathbf{x}$, belongs to the same category, as do the closest neighbors in θ. The k-NN algorithms are used for the classification of objects, in many practical applications, e.g., in the areas of artificial intelligence, pattern recognition, statistics, cognitive psychology, vision analysis, and in medicines. Under many circumstances, the k-NN algorithm is used to perform the classification.

Bayes networks are powerful tools for decision-making and reasoning under uncertainty. These networks are specified using two components: 1. a *graphical component*, composed of directed acyclic graph (DAG) to represent causal relations, and 2. *numerical component,* consisting in a quantification of different links in the DAG by conditional probability distribution. The Bayesian classifier (Naive Bayes), i.e., "straw man" assumes that the attributes in the examples are statistically independent of each other, given the value of the class attribute, which makes it computationally efficient.

The most common models for *learning* and *computation* take the *neuron* as the basic processing unit. Each such processing unit has following characteristics:

1. *Activity level*. It is the state of polarization of a neuron.
2. *Output value*. It depends on the firing rate of the neuron.
3. *Input connections*. It is the collection of *synapses* on the cell and their dendrite.
4. *Bias value*. It is an internal resting level of the neuron.
5. *Output connections*. These are neuron's *axonal* projections.

The Artificial Neural Networks (ANNs) are weighted directed graphs, where nodes are treated as artificial neurons and directed edges (with weights) are connections between neurons, and also they act as connections between input and outputs. The problem of learning in neural networks is the problem of finding a set of connection strengths which can allow the network in future to carry out the desired computation. There exist all three paradigms for learning in a neural network: *supervised*, *unsupervised*, and *hybrid*.

Instance-based learning (IBL) approaches is supervised machine learning technique. Several variants of instance-based approaches have been devised, e.g., *memory-based learning*, *exemplar-based learning*, and *case-based reasoning*. The information provided by the stored examples is used in some way, to perform the predictions in the IBL. Once the query has been answered, the prediction itself and other intermediate results are discarded.

As opposed to IBL, the *model-based* or *inductive learning* methods derive predictions in an indirect way: in step one, the observed data is used in order to induce a model, say a *decision tree* or a *regression function*. Then in the second step, the predictions are obtained on the basis of this model, which can also serve other purposes like explaining or justifying the inferences.

Exercises

1. "The task of text categorization is to assign a given document to one of the categories out of a fixed set of categories. This is done on the basis text contents. The Naive Bayes model is often used for this purpose, where a query variable is the document category and the "effect" variables are presence/ absence of each word in the language. It is assumed that words occur independently in the documents, and their frequencies determine the document category." For this statement,
   a. Explain how such a models can be constructed, given a set of "training data" in the form of documents that have been already assigned to categories.
   b. Explain how to categorize a new document.
   c. Is the independence assumption reasonable? Justify your answer.
2. What linear or nonlinear function is used by an SVM for performing classification? How is an input vector $\mathbf{x}_i$ (instance) assigned to the positive or negative classes.

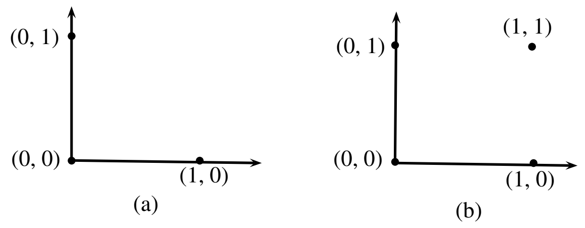


Fig. 14.12 Training data for SVMs

Table 14.1 Data set for machine learning

Department	Class	Age	Salary K$s
Sales	Programmar	$36\ldots40$	$51\ldots55$
Sales	Assistant	$31\ldots35$	$31\ldots35$
Sales	Assistant	$36\ldots40$	$36\ldots60$
Production	Assistant	$26\ldots30$	$51\ldots55$
Production	Programmar	$36\ldots40$	$71\ldots75$
Production	Assistant	$31\ldots35$	$51\ldots55$
Marketing	Programmar	$41\ldots45$	$51\ldots55$
Marketing	Assistant	$36\ldots40$	$46\ldots50$

3. Consider the SVM for the training data given in $\Re^2$, in Fig. 14.12a, b; find out the separating hyperplanes in both the cases.
4. Why is the Naive Bayes classification called Naive? What are the main goals behind this classification?
5. Consider the data given in Table 14.1,
   and use these to train a Naive Bayes classifier with designation attribute as the class label and all the remaining attributes regarded as input. Once you have your Naive Bayesian classifier, test the following unseen instances to find out the class:
   a. Marketing, $36\ldots41$, $51K\ldots55K$
   b. Sale, $36\ldots41$, $71K\ldots75K$

References

1. Aha DW et al (1991) Instance-based learning algorithms. Mach Learn 6:37–66
2. Boser EB et al (1992) A training algorithm for optimal margin classifiers. In: Proceedings of the fifth annual workshop on computational theory, COLT' 92. ACM, New York, pp 144–152
3. Bradley P (2002) Scaling mining algorithms. Commun ACM 45(8):38–43
4. Clark P, Niblett T (1989) The CN2 induction algorithm. Mach Learn 3:261–283
5. Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20:273–297
6. Domingos P (2012) A few useful things to know about machine learning. Commun ACM 55(10):78–87

7. Fukunaga K, Narendra PM (1975) A branch and bound algorithm for computing K-nearest neighbors. IEEE Trans Comput 750–753
8. Jain AK et al (1996) Artificial neural networks: a tutorial. Computer 3:31–46
9. Joachims T (2009) Predicting structured objects with support vector machines. Commun ACM 52(11):97–104
10. Rummelhart DE et al (1994) The basic ideas in neural networks. Commun ACM 37(3):86–92
11. Shawe-Taylor J (2009) Machine learning for complex predictions. Commun ACM 52(11):96–96