Chapter 2
Supervised Learning

2.1 Introduction

In a supervised learning task we are interested in finding a function that maps a set of given examples into a set of classes or categories. This function, called *classifier*, will be used later to classify new examples, which in general are different from the given ones. The process of modeling the classifier is called *training*, and the algorithm used during the training is called *learning algorithm*. The initial given examples used by the learning algorithm are called *training examples*. The training examples are usually provided by an external agent to the robot also called *tutor*. Once the classifier is learned, it should be tested on new unseen examples to evaluate its performance. These new examples are called *test examples*.

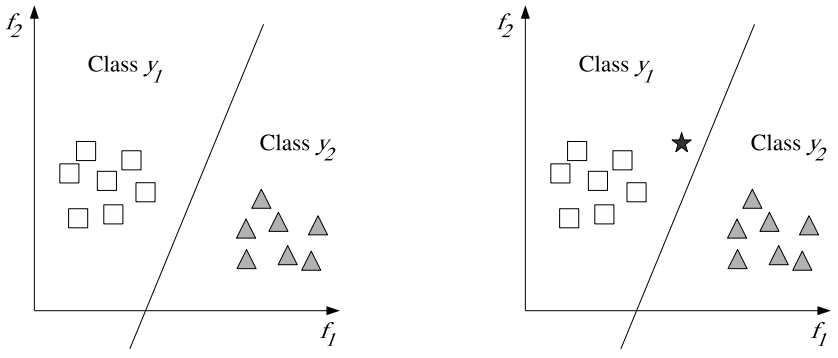
Before using the examples (both training and test) in the classification process, they are usually transformed into a *feature vector*. A feature vector is a set of values that represent some characteristics of the classes we want to learn. In the more general case, each example x is represented by a set of L features in the form

$$x = \{f_1, \ldots, f_L\},$$

In the problems presented in this book each feature will be represented by a real value, i.e. $f_l \in \mathbb{R}$.

In the case of a binary classification, the classifier needs to distinguish between two classes only. The most intuitive way of separating two classes is by creating a hyperplane that separates the examples of both classes in the feature space. An example is shown in Fig. 2.1. In this figure two classes y_1 and y_2 are represented in a bidimensional feature space. In the training process, a hyperplane—a line in 2D—is calculated in a way that separates the feature space into two parts. Each division contains the examples of one class. In the classification step, a new example is represented in the feature space. The new example is assigned the class corresponding to the division of the space in which the example is located.

Ó.M. Mozos: Semantic Labeling of Places with Mobile Robots, STAR 61, pp. 7–13.
springerlink.com



(a) Training step (b) Classification step

Fig. 2.1 (**a**) In the training step a line is calculated that separates the training examples of classes y_1 (*squares*) and y_2 (*triangles*). (**b**) In the classification step a new example (*star*) is classified according to the decision line. In this case it belongs to class y_1.

In general, the division of the feature space into different regions is done using functions that define different types of curves. These types of classifiers are called *discriminative*, since they just divide the feature space and discriminate the examples from the different classes.

The solution to the binary case can be used to learn one class only. To do this we divide the set of all possible examples in the world into two sets: a first set corresponding to the examples of the class we want to learn, and a second set containing the remaining examples. The examples of the learned class received the name *positive examples*, while the rest of the examples are called *negative examples*.

The learning of one class allows us to introduce some performance metrics for the classifiers. The evaluation of the performance is done using the set of test examples. These examples need to contain their corresponding class, since this value will be used in the metrics. After applying the learned classifier to the test set we can obtain the following values:

True positives (TP)	Test examples whose original class was positive and are classified as positive.
True negatives (TN)	Test examples whose original class was negative and are classified as negative.
False positives (FP)	Test examples whose original class was negative but are classified as positive.
False negatives (FN)	Test examples whose original class was positive but are classified as negative.

According to these values a good classifier will try to maximize the number of true classifications (TP and TN) while trying to minimize the number of false ones (FP and FN). These values are typically shown in a matrix form with two rows and two columns. This matrix is called *confusion matrix*. Moreover, these performance

values are the basics for further metrics such as lift charts, receiver operating characteristic (ROC) curves, or recall–precision curves [16].

An important phenomenon that should be avoided when training a classifier is *overfitting*. Overfitting occurs when a classifier does very well in the training set but performs poorly in the test set. This situation usually occurs because the learned function fits too much to the training examples, leaving a small margin to detect new examples if they are slightly far away in the feature space. The capability of a classifier to detect new examples, even if they are slightly different from the training set is called *generalization*. Opposite to overfitting, generalization is a desired behavior in a classifier.

A classifier is also called a hypothesis, since it is a guess of the class to which an example belongs. The hypotheses that are only slightly better than a random guess are called *weak hypotheses*. Similarly, the classifiers that are much better than a random guess are called strong hypotheses.

Further details on supervised learning can be found in [9, 14, 16].

2.2 Boosting

Boosting is a general method which attempts to improve the accuracy of a given learning algorithm [4, 8, 12]. This approach has its roots in the probably approximately correct (PAC) framework [15].

Kearns and Valiant [7, 6] were the first to pose the question of whether a weak learning algorithm can be combined into an accurate strong learning algorithm. Later, Shapire [11] demonstrated that any weak learning algorithm can be efficiently transformed or boosted into a strong learning algorithm.

The underlying idea of boosting is to combine a set of T weak hypotheses

$$\{h_1, h_2, \ldots, h_T\}$$

to form a strong hypothesis H such that the performance of the strong hypothesis is better than the performance of each of the single weak hypothesis h_t in the form

$$H(x) = \sum_{t=1}^{T} \alpha_t h_t(x) \,, \tag{2.1}$$

where α_t denotes the weight of hypothesis h_t. Both α_t and the hypothesis h_t are to be learned within the boosting procedure. The resulting strong hypothesis H has the form of a weighted majority vote classifier.

The boosting algorithm proceeds as follows. The algorithm is provided with a set of labeled training examples $(x_1, y_1), ..., (x_N, y_N)$, where y_n is the label associated with instance x_n. On each round $t = 1, \ldots, T$, the boosting algorithm devises a weight distribution D_t over the set of examples, and selects a weak classifier h_t with low error ε_t with respect to D_t. Thus, the distribution D_t specifies the relative importance of each example for the current round. After T rounds, the booster must combine the weak classifiers into a strong one. The key idea is to alter the distribution over

the training examples in a way that increases the weights of the harder elements, thus forcing the weak classifier to make less mistakes on these elements.

An important aspect related to boosting is overfitting. Large parts of the early literature explain that boosting would not overfit even when using a large number of rounds. However, simulations shown in [5, 10] indicate that data sets with high noise content could clearly show overfitting effects.

2.2.1 AdaBoost

The AdaBoost algorithm, introduced by Freund and Schapire [3], is one of the most popular boosting algorithms. Following the general idea of boosting, the AdaBoost algorithm takes as an input a training set of examples $(x_1, y_1), ..., (x_N, y_N)$, with y_n being the label associated with instance x_n. Since the algorithm is designed for binary classifications, the label for each example indicates whether it is positive $y_n = 1$ or negative $y_n = 0$. On each round $t = 1, \ldots, T$, AdaBoost calls a weak learning algorithm repeatedly to select a weak hypothesis h_t.

The AdaBoost algorithm differs from previous boosting algorithms [1, 2, 11] in that it needs no prior knowledge of the accuracies of the weak classifiers. During the boosting process, AdaBoost adapts to these accuracies and generates weighted majority hypotheses in which the weight of each weak hypothesis is a function of its accuracy.

The complete algorithm is described in Figure 2.2. In this algorithm, the distribution D indicates the importance of the examples at the beginning of the training process and it is controlled later by the iterative process. This distribution can be set initially as the uniform distribution so that $D_1(n) = 1/N$, meaning that all examples have the same importance at the beginning. On each round t, the algorithm maintains the normalized weight distribution $D_t(1), \ldots, D_t(N)$ over the training examples. The distribution D_t is fed to the weak learner which generates a classifier h_t that has a small error with respect to this distribution. The accuracy of the weak hypothesis h_t is measured by its error as

$$\varepsilon_t = \sum_{n=1}^{N} D_t(n) \left| h_t(x_n) - y_n \right| . \tag{2.2}$$

Notice that the error is measured with respect to the distribution D_t on which the weak learner was trained. In practice, the weak learning algorithm may be able to use the weights D_t on the training examples. Alternatively, when this is not possible, a subset of the training examples can be sampled according to D_t, and these resampled examples can be used to train the weak learner.

Using the new hypothesis h_t, the boosting algorithm generates the next weight vector D_{t+1}, and the process is repeated. After T iterations, the final strong hypothesis H is generated, combining the outputs of the T weak hypotheses using a weighted majority vote.

Freund and Schapire [3] proved that, for binary classification problems, the training error of the final hypothesis H generated by the AdaBoost algorithm is bounded by

$$\varepsilon \leq 2^T \prod_{t=1}^{T} \sqrt{\varepsilon_t (1-\varepsilon_t)} \leq \exp\left(-2 \sum_{t=1}^{T} \gamma^2\right), \tag{2.3}$$

where $\varepsilon_t = 1/2 - \gamma_t$ is the error of weak hypothesis h_t. Since a hypothesis that makes an entirely random guess has error 0.5, γ_t measures the accuracy of the weak hypothesis h_t relative to a random guess. This bound shows that the final training error drops exponentially if each of the weak hypotheses is better than a random guess.

Since the accuracy of the AdaBoost algorithm depends on the fact that the weak classifiers are better than a random guess, an alternative way to stop the iterative process of Fig. 2.2 consists of testing whether the selected weak classifier h_t has a classification error better than 0.5. If this is not the case, then the loop should

- Input:
  - Set of N labeled examples $(x_1, y_1), \ldots, (x_N, y_N)$ with $y_n = 1$ if the example x_n is positive, and $y_n = 0$ if the example x_n is negative
  - Distribution D over the N examples
  - Weak learner
  - Integer T specifying the number of iterations
- Initialize the weight vector $D_1(n)$ for $i = 1, \ldots, N$.
- For $t = 1, \ldots, T$
  1. Normalize the weight distribution
  $$D_t(n) = \frac{D_t(n)}{\sum_{i=1}^{N} D_t(i)}$$
  2. Train weak learner using distribution D_t and get back a hypothesis $h_t : \mathrm{X} \rightarrow \{0, 1\}$.
  3. Calculate the error for h_t as
  $$\varepsilon_t = \sum_{n=1}^{N} D_t(n) \left| h_t(x_n) - y_n \right|$$
  4. Set
  $$\beta_t = \frac{\varepsilon_t}{(1-\varepsilon_t)}$$
  5. Set the new weights
  $$D_{t+1}(n) = D_t(n) \beta_t^{1-|h_t(x_n)-y_n|}$$
- The final strong hypothesis is given by:
$$H(x) = \begin{cases} 1 \text{ if } \sum_{t=1}^{T} \left(\log \frac{1}{\beta_t}\right) h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \log \frac{1}{\beta_t} \\ 0 \text{ otherwise} \end{cases}$$

Fig. 2.2 The AdaBoost algorithm [3].

- Input:
  - Set of N labeled examples $(x_1, y_1), \ldots, (x_N, y_N)$ with $y_n = 1$ if the example x_n is positive, and $y_n = -1$ if the example x_n is negative
  - Weak learner
  - Integer T specifying the number of iterations
- Initialize the weights $D_1(n) = 1/N$ for $n = 1, \ldots, N$.
- For $t = 1, \ldots, T$
  1. Train the weak learner with distribution D_t and get back a hypothesis $h_t : \mathrm{X} \to [0, 1]$.
  2. Choose $\alpha_t \in \mathbb{R}$.
  3. Update:
  $$D_{t+1}(n) = \frac{D_t(n) \exp(-\alpha_t y_n h_t(x_n))}{Z_t}$$
  where Z_t is a normalization factor so that D_{t+1} is a distribution
- The final strong hypothesis is given by:
$$H(x) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Fig. 2.3 Generalized version of the AdaBoost algorithm [13].

be finished, and the final strong classifier H constructed using the weak classifiers selected so far.

2.2.2 Generalized AdaBoost

An alternative version to the original AdaBoost algorithm was introduced in [13]. This version, called generalized AdaBoost, presents several improvements. First, the output of the weak hypotheses can have any real value inside the range $[0, 1]$ rather than only two values. Second, in this version of the algorithm the different α_t, which correspond to the weights of the final weak hypotheses, are left unspecified. The complete algorithm is shown in Figure 2.3.

In [13] a possible choice for the different weights α_t is presented

$$\alpha_t = \frac{1}{2} \ln\left(\frac{1 + r_t}{1 - r_t}\right), \tag{2.4}$$

where r_t is chosen at each iteration so that its absolute value $|r_t|$ is maximized according to

$$r_t = \sum_{n=1}^{N} D_t(n) y_n h_t(x_n). \tag{2.5}$$

Several versions of the generalized algorithm together with different comparisons are presented in [13].

References

1. Freund, Y.: Boosting a weak learning algorithm by majority. In: COLT: Proceedings of the Workshop on Computational Learning Theory. Morgan Kaufmann, San Francisco (1990)
2. Freund, Y.: Data filtering and distribution modeling algorithms for machine learning. PhD thesis, University of California at Santa Cruz (1993)
3. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. In: Proceedings of the European Conference on Computational Learning Theory, pp. 23–37 (1995)
4. Freund, Y., Schapire, R.E.: A short introduction to boosting. Journal of Japanase Society for Artificial Intelligence 14(5), 771–780 (1999)
5. Grove, A.J., Schuurmans, D.: Boosting in the limit: Maximizing the margin of learned ensembles. In: Proceedings of the National Conference on Artificial Intelligence, pp. 692–699 (1998)
6. Kearns, M., Valiant, L.G.: Cryptographic limitations on learning boolean formulae and finite automata. Journal of the ACM 41(1), 67–95 (1994)
7. Kearns, M.J., Valiant, L.G.: Learning boolean formulae or finite automata is as hard as factoring. Technical Report TR-14-88, Harvard University Aiken Computation Laboratory (August 1988)
8. Meir, R., Rätsch, G.: An introduction to boosting and leveraging. In: Mendelson, S., Smola, A.J. (eds.) Advanced Lectures on Machine Learning. LNCS (LNAI), vol. 2600, pp. 118–183. Springer, Heidelberg (2003)
9. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
10. Rätsch, G., Onoda, T., Müller, K.R.: Soft margins for adaboost. Machine Learning 42(3), 287–320 (2001)
11. Schapire, R.E.: The strength of weak learnability. Machine Learning 5, 197–227 (1990)
12. Schapire, R.E.: The boosting approach to machine learning: An overview. In: MSRI Workshop on Nonlinear Estimation and Classification (2001)
13. Schapire, R.E., Singer, Y.: Improved boosting algorithms using confidence-rated predictions. Machine Learning 37(3), 297–336 (1999)
14. Theodoridis, S., Koutroumbas, K.: Pattern Recognition, 3rd edn. Academic Press, London (2006)
15. Valiant, L.G.: A theory of the learnable. Communications of the ACM 27(11), 1134–1142 (1984)
16. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques, 2nd edn. Morgan Kaufmann, San Francisco (2005)