Introduction to multivariate discrimination

Balázs Kégl

EPJ Web of Conferences **55**, 02001 (2013)
DOI: 10.1051/epjconf/20135502001

Introduction to multivariate discrimination

Balázs Kégl

[1]*LAL, Orsay (IN2P3)*

Abstract. Multivariate discrimination or classification is one of the best-studied problem in machine learning, with a plethora of well-tested and well-performing algorithms. There are also several good general textbooks [1–9] on the subject written to an average engineering, computer science, or statistics graduate student; most of them are also accessible for an average physics student with some background on computer science and statistics. Hence, instead of writing a generic introduction, we concentrate here on relating the subject to a practitioner experimental physicist. After a short introduction on the basic setup (Section 1) we delve into the practical issues of complexity regularization, model selection, and hyperparameter optimization (Section 2), since it is this step that makes high-complexity non-parametric fitting so different from low-dimensional parametric fitting. To emphasize that this issue is not restricted to classification, we illustrate the concept on a low-dimensional but non-parametric *regression* example (Section 2.1). Section 3 describes the common algorithmic-statistical formal framework that unifies the main families of multivariate classification algorithms. We explain here the large-margin principle that partly explains why these algorithms work. Section 4 is devoted to the description of the three main (families of) classification algorithms, neural networks, the support vector machine, and AdaBoost. We do not go into the algorithmic details; the goal is to give an overview on the form of the functions these methods learn and on the objective functions they optimize. Besides their technical description, we also make an attempt to put these algorithm into a socio-historical context. We then briefly describe some rather heterogeneous applications to illustrate the pattern recognition pipeline and to show how widespread the use of these methods is (Section 5). We conclude the chapter with three essentially open research problems that are either relevant to or even motivated by certain unorthodox applications of multivariate discrimination in experimental physics.

1 The supervised learning setup

The goal of supervised learning is to infer a function $g : \mathcal{X} \to C$ from a data set

$$\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \in (\mathcal{X} \times C)^n$$

comprised of pairs of *observations* $\mathbf{x}_i$ and *labels* y_i. The components $x^{(j)}$ of the d-dimensional *feature* or *observation* vector $\mathbf{x}$ are either real numbers or they come from an unordered set of cardinality $M^{(j)}$. In this latter case, without loss of generality, we will assume that $x^{(j)} \in \mathcal{I}^{(j)} = \{1, \ldots, M^{(j)}\}$.

When the label space C is real-valued (for example, the label represents the mass of a particle), the problem is known as *regression*, whereas when the labels come from a finite set of classes (for

example, $C = \{\text{signal}, \text{background}\}$), we are talking about *classification*. The quality of g on an arbitrary pair $(\mathbf{x}, y) \in \mathcal{X} \times C$ is measured by an *error* or *loss* function $L(g, (\mathbf{x}, y))$ that depends on the type of problem. In regression, the goal is to get $g(\mathbf{x})$ as close to y as possible, so the loss grows with the difference between $g(\mathbf{x})$ and y. The most widely used loss function in this case is the *quadratic* or *squared* loss

$$L_2(g, (\mathbf{x}, y)) = (g(\mathbf{x}) - y)^2.$$

In classification, typically there is no distance or similarity defined between the classes, so all we can measure is whether or not g predicts correctly the class y. The usual loss in this case is the *one-loss* or *zero-one loss*

$$L_{\mathbb{I}}(g, (\mathbf{x}, y)) = \mathbb{I}\{g(\mathbf{x}) \neq y\}, \tag{1}$$

where the indicator function $\mathbb{I}\{A\}$ is 1 if its argument A is true and 0 otherwise.

The goal of learning algorithms is not to *memorize* $\mathcal{D}$, rather to *generalize* from it. Indeed, it is rather trivial to construct a function g that has zero loss on every sample point $(\mathbf{x}_i, y_i)$ by explicitly setting $g(\mathbf{x}_i) \triangleq y_i$ on all sample points from $\mathcal{D}$, and, say, $g(\mathbf{x}_i) \triangleq 0$ everywhere else. We obviously have not *learned* anything from $\mathcal{D}$, we have simply memorized it. It is also clear that this function has very little chance to perform well on points not in the set $\mathcal{D}$ (unless 0 is a good prediction everywhere in which case the problem itself is trivial). To formalize this intuitive notion of generalization, it is usually assumed that all sample points (including those in $\mathcal{D}$) are independent samples from a fixed but unknown distribution $\mathfrak{D}$, and the goal is to minimize the *expected loss* (a.k.a., *risk* or *error*)

$$R(g) = \mathbb{E}_{(\mathbf{x},y)\sim\mathfrak{D}}\{L(g, (\mathbf{x}, y))\},$$

where $\mathbb{E}_{(\mathbf{x},y)\sim\mathfrak{D}}\{L\}$ denotes the expectation of L with respect to the random variable $(\mathbf{x}, y)$ drawn from the distribution $\mathfrak{D}$. When we *know* $\mathfrak{D}$, the *optimal* function is the one that minimizes the risk, that is,

$$g^* = \arg\min_g R(g).$$

With this notation, since the expectation of the indicator of an event is the probability of the event, the misclassification probability $\mathbb{P}\{g(\mathbf{x}) \neq y\}$ is the risk generated by the one-loss

$$R_{\mathbb{I}}(g) = \mathbb{E}_{(\mathbf{x},y)\sim\mathfrak{D}}\{L_{\mathbb{I}}(g, (\mathbf{x}, y))\} = \mathbb{P}\{g(\mathbf{x}) \neq y\},\ ^1 \tag{2}$$

and so it can be shown that the optimal classifier (the so-called *Bayes classifier*) is

$$g^*_{\mathbb{I}}(\mathbf{x}) = \arg\max_y \mathbb{P}\{y \mid \mathbf{x}\}.$$

In regression, the risk is the expectation of the squared error

$$R_2(g) = \mathbb{E}_{(\mathbf{x},y)\sim\mathfrak{D}}\left\{(g(\mathbf{x}) - y)^2\right\},$$

and it can be shown easily that the optimal regressor is the conditional expectation of y given $\mathbf{x}$, that is,

$$g^*_2(\mathbf{x}) = \mathbb{E}\{y \mid \mathbf{x}\}.$$

[1] For denoting the risk generated by a particular loss, we will use the same index: the risk generated by the loss $L_\bullet$ will be denoted by $R_\bullet$.

In practice, of course, the data distribution $\mathfrak{D}$ is unknown, and the goal is to get close to the performance of g^* by only using the sample $\mathcal{D}$. As our imaginary example demonstrates it, finding a function g that minimizes the *empirical risk* (or *training error*)

$$\widehat{R}(g) = \frac{1}{n}\sum_{i=1}^{n} L(g, (\mathbf{x}_i, y_i)) \tag{3}$$

is not necessarily a good idea. Nevertheless, it turns out that the estimator

$$\widehat{g}^* = \arg\min_{g\in\mathcal{G}} \widehat{R}(g) \tag{4}$$

can have good properties both in theory and in practice if the function class $\mathcal{G}$ is not too "rich" so the functions in $\mathcal{G}$ are not too complex. In fact, the *real* error $R(\widehat{g}^*)$ of the empirically best classifier $\widehat{g}^*$ is usually underestimated by the *training* error $\widehat{R}(\widehat{g}^*)$, but the bias can be controlled, and the minimization (4) can lead to a good solution if the complexity of the function class $\mathcal{G}$ is "small" compared to the number of data points n.

How to control the complexity of $\mathcal{G}$ and how to find an optimal function in $\mathcal{G}$ computationally efficiently are the two main subjects of the design of supervised learning algorithms. Finding $\widehat{g}^*$ in $\mathcal{G}$ is the subject of algorithmic design. The process is called *training* or *learning*, and the data set $\mathcal{D}$ on which $\widehat{R}(g)$ is minimized is the *training set*. Of course, once $\mathcal{D}$ is used for finding $\widehat{g}^*$, it is tainted from a statistical point of view: $\widehat{R}(\widehat{g}^*)$ is no longer an unbiased estimator of $R(\widehat{g}^*)$. *Overfitting* is the term that describes the situation when the risk $R(\widehat{g}^*)$ is significantly larger than the training error $\widehat{R}(\widehat{g}^*)$. To detect and to assess overfitting, $\widehat{g}^*$ has to be evaluated on a *test set* $\mathcal{D}'$, independent of $\mathcal{D}$.

1.1 Parametric fitting and the maximum likelihood method

The main subject of the chapter is non-parametric (model-less) multi-variate classification, and the setup of Section 1 is designed to formalize the theoretical framework for this approach. Nevertheless, it is worthy to note that "classical" low-dimensional fitting, be it maximum likelihood or least-square, also fits into this framework. These methods are treated in other chapters, so here we only sketch them to illuminate their relationship to the setup of Section 1.

When we know a generative probabilistic model, one of the possibilities of fitting a model is to maximize the likelihood $p(\mathbf{z})$. This can also be cast into the framework of this section with the usual loss function

$$L_P(\mathbf{z}) = -\log p(\mathbf{z}),$$

and with the risk that L_P implies

$$R_P(\mathbf{z}) = \mathbb{E}_{\mathbf{z}\sim\mathfrak{D}}\{-\log p(\mathbf{z})\}.$$

For example, when the data points $\mathbf{z}_i$ are triplets $(\mathbf{x}_i, y_i, \sigma_i)$ with the generative model of $y_i \sim \mathcal{N}(g(\mathbf{x}_i), \sigma_i)$,[2] maximum likelihood reduces to weighted least square ("chi square"), that is,

$$\arg\min_{g\in\mathcal{G}} \sum_{i=1}^{n} -\log p(\mathbf{x}_i, y_i, \sigma_i) = \arg\min_{g\in\mathcal{G}} \sum_{i=1}^{n} \frac{(g(\mathbf{x}_i) - y_i)^2}{\sigma_i^2}.$$

Although the least-square method is mostly used in parametric fitting when $\mathcal{G}$ is a low-complexity function class parametrized with a few parameters, neural networks and other model-less approaches can be used also with a (weighted) quadratic loss, and in this case all considerations about overfitting, complexity regularization, and hyperparameter optimization, usually associated with non-parametric *classification*, are also valid.

[2] $\mathcal{N}(\mu, \sigma)$ denotes the normal distribution with mean μ and standard deviation σ.

2 Complexity regularization (model selection) and hyperparameter optimization

The simplest way to control the complexity of the function class $\mathcal{G}$ is to fix it either explicitly (for example, by taking the set of all linear functions) or implicitly (for example, by taking the set of all functions that a neural network with N neurons and one hidden layer can represent). In *parametric* fitting, $\mathcal{G}$ is a set of functions parametrized with a low number of parameters, and the dimensionality d of the input space is usually low. Most importantly, the number of parameters is *fixed* independently of the size n of the training data set, so the overfitting issue only shows up in goodness-of-fit tests (for example, in setting the degrees of freedom of the chi-square distribution).

The situation is different in *non-parametric* fitting. Contrary to what its name suggests, non-parametric fitting means that we have a lot of parameters. This approach is used when we have little knowledge about the model that generated the observations, so we want to avoid the bias caused by fitting a misspecified rigid model. More importantly, we do not want to decide the complexity or the smoothness of the solution beforehand; we prefer that the data speak for itself.

Formally, we can define a (possibly infinite) *set* of function classes $\mathcal{G}_\theta$, parametrized by a vector θ of the so-called *hyperparameters*. Hyperparameters can take diverse forms. Most of them are related to the complexity of the class; the number of neurons in a neural network, the depth of decision trees, or the number of boosting iterations are typical examples. Other hyperparameters are penalty coefficients in a framework called *complexity regularization* [4] or *structured risk minimization* [9]. In this setup, instead of finding the empirical minimizer in $\mathcal{G}$, we *penalize* the complexity of the functions $g \in \mathcal{G}$ by an appropriate penalty $C(g)$, and find

$$\widehat{g}^*_\beta = \arg\min_{g\in\mathcal{G}} \widehat{R}(g) + \beta C(g), \tag{5}$$

where β is a penalty coefficient (a hyperparameter) which has to be tuned. The classical practices of *weight decay* or *early stopping* in neural networks fall in the category of structured risk minimization. Another typical example is the penalty coefficient (usually denoted by C) in support vector machines. A third family of hyperparameters are simple "technical" parameters, such as the learning rate in neural networks, or the number of features $x^{(j)}$ tested at each cut in a decision tree.

There is a vast literature on how to tune hyperparameters on the training set $\mathcal{D}$ itself. This so-called *model selection* problem has both Bayesian and frequentist solutions, but most of the time, at least in model-less non-parametric fitting, optimality results are only valid asymptotically (as training sample size $n \to \infty$), and so they only provide some ballpark hints in practice. The solution adopted in practice is called *validation*. In the simplest setup, we choose $\widehat{g}^*_\theta$ by minimizing the training error in each set $\mathcal{G}_\theta$

$$\widehat{g}^*_\theta = \arg\min_{g\in\mathcal{G}_\theta} \widehat{R}(g;\mathcal{D}), \tag{6}$$

then we choose the overall best predictor $\widehat{g}^*$ by minimizing the error on a held-out *validation* set $\mathcal{D}''$

$$\widehat{g}^* = \arg\min_{\widehat{g}^*_\theta} \widehat{R}(\widehat{g}^*_\theta;\mathcal{D}''),$$

where we explicitly added $\mathcal{D}$ and $\mathcal{D}''$ as an argument of the error to emphasize that the two minimizations use two different sets. Indeed, $\mathcal{D}''$ has to be independent of both the training set $\mathcal{D}$ and the eventual test set $\mathcal{D}'$ (used for estimating the performance of the final classifier). This procedure is known as *hyperparameter optimization* or *hyperparameter tuning* in machine learning.

The particularity of hyperparameter optimization is that the evaluation of the function $f(\theta) = \widehat{R}(\widehat{g}^*_\theta;\mathcal{D}'')$ is computationally expensive. Indeed, the evaluation of $f(\theta)$ requires the optimization of

$\widehat{g}_\theta \in \mathcal{G}_\theta$, that is, *training* $\widehat{g}_\theta$ on $\mathcal{D}$ and *testing* it on $\mathcal{D}''$, which can take hours or days, depending on the particular learning algorithm, the data size n, and the data dimensionality d. This feature relates hyperparameter optimization to *experimental design* where, say, we want to optimize a handful parameters of a detector by evaluating its performance using expensive simulations. When the number of hyperparameters is small (say, not more than 3), the usual procedure is simple *grid search*: we discretize the components of θ, and evaluate $f(\theta)$ for all members of the discrete set. For example, we train AdaBoost for $T \in \{10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10\,000\}$ iterations using decision trees with a number of leaves of $N \in \{2, 3, 5, 10, 20, 50, 100\}$, and then select the best (T, N) pair out of the 70 trained functions. When the number of hyperparameters exceeds three, this procedure rapidly becomes infeasible because the number of grid points blows up exponentially. Simple heuristics, such as Latin hypercube search, simple random search [10], or gradient search [11] can be applied, but the principled solution is *surrogate optimization*: we replace f by a, usually non-parametric, smooth function which we train iteratively on a sequence of evaluation points $(\theta_t, f(\theta_t))$. In each iteration t, a surrogate function $\widehat{f}_t(\theta)$ is regressed over the set $\{\theta_{t'}, f(\theta_{t'})\}_{t'=1}^{t}$, then a new evaluation point is selected based on $\widehat{f}_t(\theta)$. Gaussian process regression [12] is one of the most popular concrete choices for the regression method since it provides not only a regressor but also a conditional distribution $p\left(f(\theta) \mid \theta\right)$, and so it can be used with probabilistic criteria to select the next evaluation point θ_t. The best-known such criterion is *expected improvement*. The recent success of deep neural networks [13–15], with often tens of hyperparameters, generated a surge of papers on the subject [10, 16–19], showing that the jury is still out on what the best strategy is.

2.1 An example in low-dimensional fitting

To illustrate that overfitting and model selection do not only concern multi-variate classification, we start by a simple example using Gaussian process (GP) regression [12] on a one-dimensional fitting problem. The main ingredient of a GP is the kernel function $K(x, x')$ that defines the smoothness of the regressor functions $\widehat{g}(x)$. In this example we use the squared exponential kernel

$$K(x, x') = a \exp\left(-\frac{(x - x')^2}{w^2}\right),$$

where a and w are hyperparameters to be tuned. Without going into the details, the GP regression function is given in the form of

$$\widehat{g}(x) = \sum_{i=1}^{n} \alpha_i K(x, x_i).$$

The coefficient vector $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)$ is given by

$$\boldsymbol{\alpha} = (\mathbf{K} + \Sigma)^{-1}\mathbf{y},$$

where $\mathbf{K} = [K(x_i, x_j)]_{i,j=1}^{n}$ is the *Gram matrix* and $\mathbf{y} = (y_1, \ldots, y_n)$ is the label vector. The diagonal matrix Σ contains either the squared uncertainties σ_i^2 in the diagonal in case they are known, or a constant σ^2 which becomes a third hyperparameter that has to be tuned along with a and w.

There are both Bayesian and frequentist techniques to find the optimal hyperparameters a, w, and σ using a single training set $\mathcal{D}$. These methods work well in practice, so validation is not a common technique in GP regression. Nevertheless, for the sake of illustrating the general technique, we show on an example how validation works in this case. We generate 50 training and 50 validation points

(x_i, y_i) by first drawing x_i uniformly from the interval $[3, 100]$, and then drawing $y_i \sim \mathcal{N}(g(x_i), 0.4)$, where

$$g(x) = \sin\left(\sqrt{x}\right)$$

is our target regression function. Figure 1 depicts $g(x)$ in blue, and the training set $\mathcal{D}$ in red (the validation set $\mathcal{D}''$ is not shown). For the sake of simplicity, we fix $a = 10$ and $\sigma = 0.4$ and only tune the kernel width w. The role of w is to set the smoothness of the regressor function. We plot three cases in Figure 1. When $w = 10$, the estimated regressor $\widehat{g}(x)$ clearly overfits the data: it follows the training data too closely, achieving a root mean square error (RMSE) $\sqrt{R_2(g)} = 0.33$ on the training set. At the same time, on the validation set its RMSE is 0.47, giving a qualitative indication of overfitting. When $w = 80$, we are underfitting (oversmoothing) the data. The training and validation RMSEs are close (0.47 and 0.48, respectively), but they are both suboptimal. At the optimum $w = 40$, selected by the validation procedure, we can achieve a 0.39 training RMSE and a 0.38 validation RMSE. The validation RMSE slightly underestimates the expected RMSE since w was selected based on this sample, but this "hyper-overfitting" is negligible. In fact it is comparable to overfitting a single-parameter function in a parametric setup.

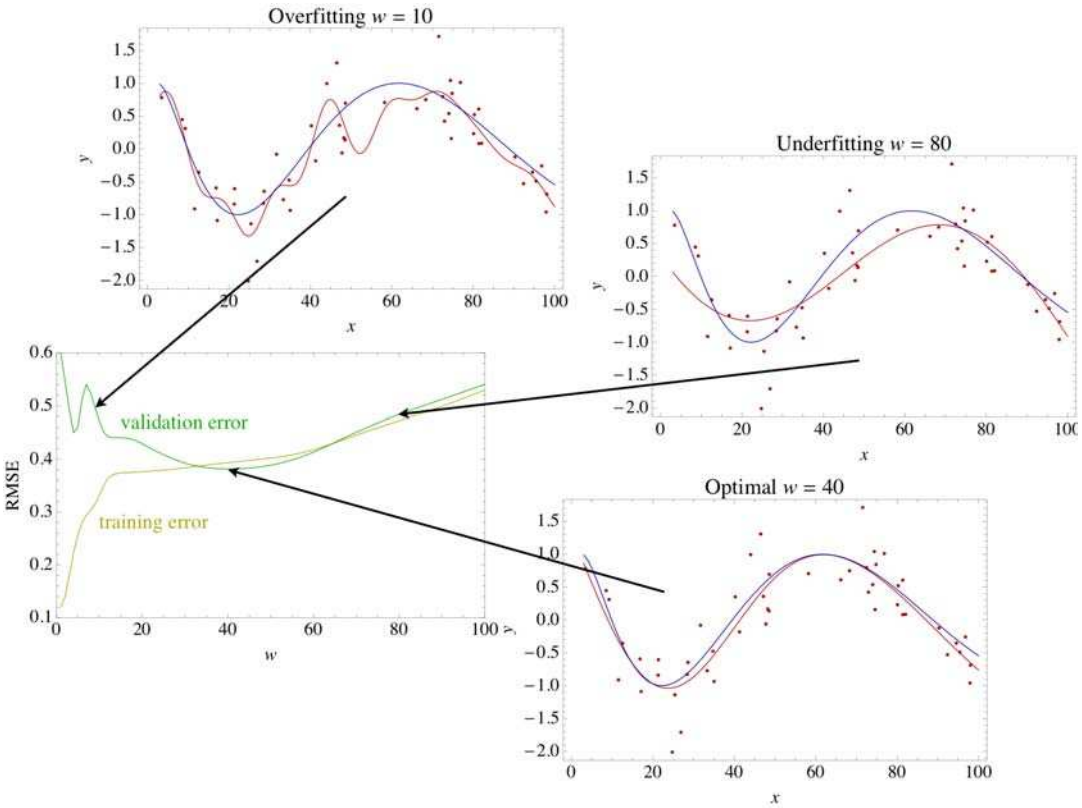


Figure 1. Over- and underfitting in non-parametric GP regression. The blue curve is the target regression function $g(x) = \sin\left(\sqrt{x}\right)$. The red dots are the training set: x_i is drawn uniformly from $[3, 100]$, and $y_i \sim \mathcal{N}(g(x_i), 0.4)$. Two of the GP hyperparameters are fixed to $a = 10$ and $\sigma = 0.4$, and the third hyperparamater w is tuned on a validation set, yielding an optimal value of $w = 40$.

3 Convex losses in binary classification

Finding the empirical minimizer $\widehat{g}^*$ (4), $\widehat{g}^*_\theta$ (6), or $\widehat{g}^*_\beta$ (5) can be algorithmically challenging even in relatively "simple" function sets $\mathcal{G}$. For example, finding the *linear* binary classifier that minimizes the training error $\widehat{R}_{\mathbb{I}}(g)$ is NP hard [20].[3] One way to make the learning problem tractable is to relax the minimization of $\widehat{R}_{\mathbb{I}}(g)$ by defining *convex losses* that upper bound the one-loss $L_{\mathbb{I}}(g,(\mathbf{x},y))$ (1).

To formalize this, we need to introduce some notions used in binary classification. Most of the learning algorithms do not directly output a $\{\pm 1\}$-valued classifier g, rather, they learn a real-valued *discriminant function* f, and obtain g by thresholding the output of $f(\mathbf{x})$ at 0, that is,

$$g(\mathbf{x}) = \begin{cases} +1 & \text{if } f(\mathbf{x}) > 0, \\ -1 & \text{if } f(\mathbf{x}) \leq 0. \end{cases}$$

Using the real-valued output of the discriminant function, the classifier can be evaluated on a finer scale by defining the (*functional*) *margin*

$$\rho = \rho(f,(\mathbf{x},y)) = f(\mathbf{x})y.$$

With this notation, the misclassification indicator (one-loss) of a discriminant function f on a sample point $(\mathbf{x},y)$ can be re-defined as a function of the margin ρ by

$$L_{\mathbb{I}}(f,(\mathbf{x},y)) = L_{\mathbb{I}}\Big(\rho(f,(\mathbf{x},y))\Big) = L_{\mathbb{I}}(\rho) = \mathbb{I}\{\rho < 0\}.$$

Besides the *sign* of ρ that represents the classification error, the *magnitude* of ρ is also important: it indicates the confidence or the robustness of the classification. Indeed, the combination of a large ρ with a Lipschitz-type (slope) penalty on f means that the *geometric* margin $\rho^{(\mathrm{g})}$ is also large, that is, $\mathbf{x}$ is far from the *decision border* (defined by the set of points for which $f(\mathbf{x}) = 0$, see Figure 2).

The idea behind large margin classification is to design loss functions that "push" the decision border away from the training points. The goal is not just to minimize the training zero-one error $R_{\mathbb{I}}(g)$ (2) but also to increase the margin of the correctly classified points. The common feature of these loss functions is that 1) they penalize larger errors (negative margins) more than smaller ones, and 2) they keep penalizing even correctly classified points especially if they are close to the decision border (their margin is close to zero). Formally, one can define smooth convex upper bounds of the margin-based one-loss $L_{\mathbb{I}}(\rho)$, and minimize the corresponding empirical risks instead of the training error $\widehat{R}_{\mathbb{I}}$. The most common losses, depicted in Figure 3, are the *exponential loss*

$$L_{\mathrm{EXP}}(\rho) = \exp(-\rho), \tag{7}$$

the *hinge loss*

$$L_{1+}(\rho) = \max(0, 1-\rho), \tag{8}$$

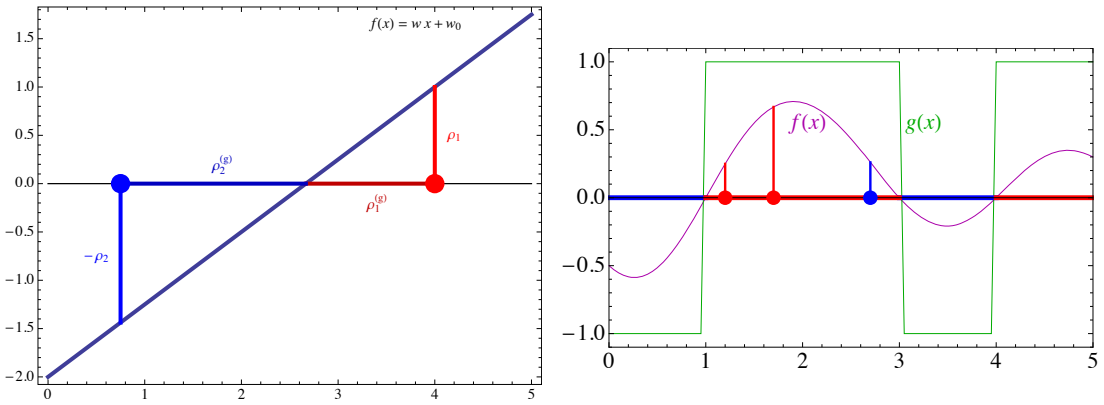
and the *logistic loss*

$$L_{\mathrm{LOG}}(\rho) = \log_2\big(\exp(-\rho)+1\big). \tag{9}$$

Given the margin-based loss $L_\bullet(\rho)$, the corresponding margin-based empirical risk can be defined as

$$\widehat{R}_\bullet(f) = \frac{1}{n}\sum_{i=1}^{n} L_\bullet(f(\mathbf{x}_i)y_i) \tag{10}$$

[3] http://en.wikipedia.org/wiki/NP_(complexity)



(a) (b)

Figure 2. Geometric and functional margins. The labels of the red and blue points are $y = +1$ and $y = -1$, respectively. (a) An upper bound ω on the slope w and a functional margin ρ implies a geometric margin (distance from the point $\mathbf{x} : f(\mathbf{x}) = 0$) of $\rho^{(g)} > \rho/\omega$. (b) The discriminant function $f(\mathbf{x})$ and the induced classifier $g(\mathbf{x})$. The blue point (with label $y = -1$) is misclassified and the two red points (with label $y = +1$) are correctly classified, but the second red point has a larger margin $f(x)y$ indicating that its classification is more robust.

Minimizing the margin-based empirical risks $\widehat{R}_{\text{EXP}}(f)$, $\widehat{R}_{1+}(f)$, or $\widehat{R}_{\text{LOG}}(f)$ have both algorithmic and statistical advantages. First, combining the minimization of these risks with Lipschitz-type penalties[4] often leads to convex optimization problems that can be solved with standard algorithms (e.g., linear, quadratic, or convex programming). Second, it can also be shown within the complexity regularization framework that the minimization of these penalized convex risks leads to large margin classifiers with good generalization properties, confirming the intuitive explanation of the previous paragraph.

4 Binary classification algorithms

The three most popular binary classification algorithms are the *multi-layer perceptron* or *neural network* (NN) [1, 21], the *support vector machine* (SVM) [9, 22, 23], and AdaBoost [24]. They have very different origins, and the algorithmic details also make them stand apart, nevertheless, they share some important concepts, and their practical success is explained, at least qualitatively, by the same theory on large margin classification.

An important pragmatic similarity is that they all learn *generalized linear* discriminant functions of the form

$$f(\mathbf{x}) = \sum_{t=1}^{T} \alpha^{(t)} h^{(t)}(\mathbf{x}). \tag{11}$$

In neural networks $h^{(t)}(\mathbf{x})$ is a *perceptron*, that is, a linear combination of the input features followed by a sigmoidal nonlinearity σ (such as arctan)

$$h^{(t)}(\mathbf{x}) = \sigma\left(w_0^{(t)} + \sum_{j=1}^{d} w_j^{(t)} x^{(j)}\right),$$

[4] often of the form of L_1 or L_2 penalties on the coefficient vector of a *generalized linear* model; see Section 4

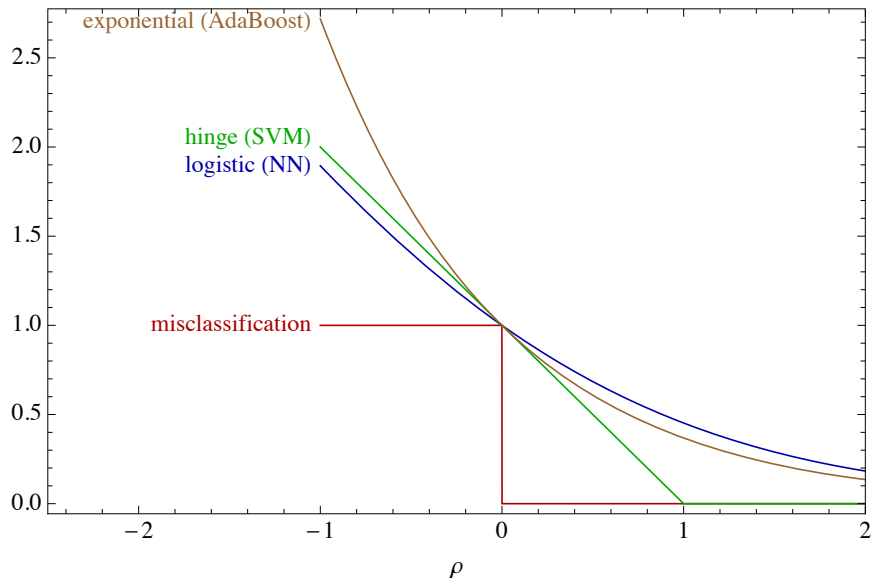


Figure 3. Convex margin losses used in binary classification.

where $x^{(j)}$ is the jth component of the d-dimensional input vector $\mathbf{x}$. The output weight vector $\boldsymbol{\alpha} = (\alpha^{(1)}, \ldots, \alpha^{(T)})$ and the $T \times (d+1)$-dimensional input weight matrix

$$\mathbf{W} = \begin{pmatrix} w_0^{(1)} & \ldots & w_d^{(1)} \\ \vdots & \ddots & \vdots \\ w_0^{(T)} & \ldots & w_d^{(T)} \end{pmatrix}$$

are learned using gradient descent optimization (called *back-propagation* in the NN literature). In the case of binary classification the neural network is usually trained to minimize the logistic loss (9). One of the advantages of NNs is their versatility: as long as the loss function is differentiable, it can be plugged into the algorithm. The differentiability condition imposed by the gradient descent optimization, on the other hand, constrains the loss function and the nonlinearities used in the hidden units: one could not use, say, a Heaviside-type nonlinearity or the one-loss.

The invention of the multilayer perceptron in 1986 [21] was a technological breakthrough: complex pattern recognition problems (e.g., handwritten character recognition) could suddenly be solved efficiently using neural networks. At the same time, the theory of machine learning at these early days was not yet developed to the point of being able to explain the principles behind algorithmic design. Most of the engineering techniques (such as *weight decay* or *early stopping*) were developed using a trial-and-error approach, and they were theoretically justified only much later within the complexity regularization and large margin framework [25]. Partly because of the "empirical" nature of neural network design, a common belief developed about the "user-unfriendliness" of neural networks. Added to this reputation was the uneasiness of having a non-convex optimization problem: back-propagation cannot be guaranteed to converge to a global minimum of the empirical risk. This is basically a no-issue in practice (especially on today's large problems), still, it is a common criticism from people who usually have never experimented with neural networks on real problems. As a consequence, neural networks were slightly over-shadowed in the 90s with the appearance of the sup-

port vector machine and AdaBoost. Today neural networks are, again, becoming more popular partly because of user-friendly program packages (e.g., [26]), partly due to the computational efficiency of the training algorithm (especially its stochastic version), and partly because of the recent success of unsupervised feature-learning techniques that use deep neural architectures to solve hard computer vision and language processing problems [13–15].

Support vector machines also learn generalized linear discriminant functions of the form (11) with

$$h^{(t)}(\mathbf{x}) = y_t K(\mathbf{x}_t, \mathbf{x}),$$

where $K(\mathbf{x}, \mathbf{x}')$ is a positive semidefinite *kernel* function that expresses the similarity between two observations $\mathbf{x}$ and $\mathbf{x}'$. $K(\mathbf{x}, \mathbf{x}')$ is usually monotonically decreasing with the distance between $\mathbf{x}$ and $\mathbf{x}'$. As in Gaussian process regression (Section 2.1), the most common choice for K is the squared exponential (a.k.a. Gaussian) kernel. The index t ranges over $t = 1, \ldots, n$ which means that each training point $(\mathbf{x}_t, y_t) \in \mathcal{D}$ contributes to f a kernel function centered at $\mathbf{x}_t$ with a sign equal to the label y_t, so the final discriminant function can be interpreted as a weighted *nearest neighbor* classifier where the weight comprises of the "similarity term" $K(\mathbf{x}_t, \mathbf{x})$ and the "importance term" $\alpha^{(t)}$.

The objective of training the SVM is to find the weight vector $\boldsymbol{\alpha} = (\alpha^{(1)}, \ldots, \alpha^{(T)})$ that minimizes the hinge loss (8) with a complexity penalty (the L_2 loss on the weights of features in the Hilbert space induced by $K(\mathbf{x}, \mathbf{x}')$). The objective function was explicitly designed based on the theory of large margin classification to ensure good generalization properties. A great advantage of the setup is that the objective is a quadratic function of $\boldsymbol{\alpha}$ with linear constraints $0 \leq \alpha^{(t)} \leq C$ (the so-called *box* constraints), so the global optimum can be found using quadratic programming. The result is sparse: only a subset of the training points in $\mathcal{D}$ have nonzero coefficients $\alpha^{(t)}$. These points are called *support vectors*, giving the name of the method.

The biggest disadvantage of the technique is its training time: naïve quadratic programming solvers run in super-quadratic time, and even with the most sophisticated tricks it is hard to beat the $O(n^2)$ barrier. The second disadvantage of the method is that in high-dimensional problems the number of support vectors is comparable to the size of the training set[5], so, when evaluating f at the test phase is a bottleneck (see Section 6.1), SVMs can be prohibitively slow. Third, unlike neural networks, SVMs are designed for binary classification, and the generic extensions for multi-class classification and regression do not reproduce the performance of binary SVM. Despite these disadvantages, the appearance of the support vector machine in the middle of the 90s revolutionized the technology of pattern recognition. Besides its remarkable generalization performance, the small number of hyperparameters[6] and the appearance of turn-key software packages[7] made SVMs the method of choice for a wide range of applications involving small-to-moderate size training sets. With the appearance of extra-large training sets in the last decade, training time and optimization became more important issues than generalization [27], so SVMs lost somewhat their dominant role.

AdaBoost learns a generalized linear discriminant function of the form (11) in an iterative fashion. It can be considered as constructing a neural network by adding one neuron at a time, but since back-propagation is no longer applied, there is no differentiability restriction on the *base classifiers* $h^{(t)}$. This opens the door to using domain-dependent features, making AdaBoost easy to adapt to a wide range of applications. The basic binary classification algorithm (Figure 4) consists of elementary steps that can be implemented by a first year computer science student in an hour. The only tricky step is

[5]This loss of sparsity is one of the manifestations of the phenomenon known as the *curse of dimensionality*.

[6]C in the bound of the αs and a length scale parameter of the kernel function $K(\mathbf{x}, \mathbf{x})$

[7]http://svmlight.joachims.org, http://www.csie.ntu.edu.tw/~cjlin/libsvm, http://www.torch.ch, http://www.loria.fr/~lauer/MSVMpack

the implementation of the base learner (line 3) whose goal is to return $h^{(t)}$ with a small *weighted* error

$$\widehat{R}_{\mathbb{I}}(h, \mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} w_i \mathbb{I}\{h(\mathbf{x}_i) \neq y_i\},$$

but most of the simple learning algorithms (e.g., decision trees, linear classifiers) can be easily adapted to this modified risk. The weights $\mathbf{w} = (w_1, \ldots, w_n)$ over the training points are initialized uniformly (line 1), and then updated in each iteration by a simple and intuitive rule (lines 7-10): if $h^{(t)}$ misclassifies $(\mathbf{x}_i, y_i)$, the weight w_i increases (line 8), whereas if $h^{(t)}$ correctly classifies $(\mathbf{x}_i, y_i)$, the weight w_i decreases (line 10). In this way subsequent base classifiers will concentrate on points that were "missed" by previous base classifiers. The coefficients $\alpha^{(t)}$ are also set analytically to the formula in line 5 which is monotonically decreasing with respect to the weighted error.

AdaBoost$(\mathcal{D}, \text{Base}(\cdot,\cdot), T)$

1 $\mathbf{w}^{(1)} \leftarrow (1/n, \ldots, 1/n)$ ▷ *initial weights*
2 **for** $t \leftarrow 1$ **to** T
3 $\quad h^{(t)} \leftarrow \text{Base}(\mathcal{D}, \mathbf{w}^{(t)})$ ▷ *base classifier*
4 $\quad \epsilon^{(t)} \leftarrow \sum_{i=1}^{n} w_i^{(t)} \mathbb{I}\left[h^{(t)}(\mathbf{x}_i) \neq y_i\right]$ ▷ *weighted error of the base classifier*
5 $\quad \alpha^{(t)} \leftarrow \frac{1}{2} \ln\left(\frac{1-\epsilon^{(t)}}{\epsilon^{(t)}}\right)$ ▷ *coefficient of the base classifier*
6 $\quad$ **for** $i \leftarrow 1$ **to** n ▷ *re-weighting the training points*
7 $\quad\quad$ **if** $h^{(t)}(\mathbf{x}_i) \neq y_i$ **then** ▷ *error*
8 $\quad\quad\quad w_i^{(t+1)} \leftarrow \frac{w_i^{(t)}}{2\epsilon^{(t)}}$ ▷ *weight increases*
9 $\quad\quad$ **else** ▷ *correct classification*
10 $\quad\quad\quad w_i^{(t+1)} \leftarrow \frac{w_i^{(t)}}{2(1-\epsilon^{(t)})}$ ▷ *weight decreases*
11 **return** $f^{(T)}(\cdot) = \sum_{t=1}^{T} \alpha^{(t)} h^{(t)}(\cdot)$ ▷ *weighted "vote" of base classifiers*

Figure 4. The pseudocode of binary AdaBoost. $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ is the training set, $\text{Base}(\cdot,\cdot)$ is the base learner, and T is the number of iterations.

This simple algorithm has several beautiful mathematical properties. First it can be shown [24] that if each base classifier $h^{(t)}$ is slightly better than a random guess (that is, every weighted error $\epsilon^{(t)} \leq \frac{1}{2} - \delta$ with $\delta > 0$), then the (unweighted) training error $\widehat{R}_{\mathbb{I}}$ of the final discriminant function $f^{(T)}$ becomes zero after at most

$$T = \left\lceil \frac{\log n}{2\delta^2} \right\rceil + 1$$

steps. The logarithmic dependence of T on the data size n implies that the technique is formally a *boosting* algorithm from which the second part of its name derives.[8] Second, it can be shown that

[8]The algorithm is also Ad*aptive* because the coefficients $\alpha^{(t)}$ depend on the errors $\epsilon^{(t)}$ of the base classifiers $h^{(t)}$.

the algorithm minimizes the exponential risk $\widehat{R}_{\text{EXP}}(f)$ (7) using a greedy functional gradient approach [28, 29]. In this alternative but equivalent formulation, in each iteration $h^{(t)}$ is selected to maximize the decrease (derivative) of $\widehat{R}_{\text{EXP}}(f^{(t-1)} + \alpha h^{(t)})$ at $\alpha = 0$, and then $\alpha^{(t)}$ is set to

$$\alpha^{(t)} = \arg\min_{\alpha} \widehat{R}_{\text{EXP}}(f^{(t-1)} + \alpha h^{(t)}).$$

Furthermore, for inseparable data sets (for which no linear combination of base classifiers achieves 0 training error), the exponential risk $\widehat{R}_{\text{EXP}}(f^{(T)})$ goes to the minimum achievable exponential risk as $T \to \infty$ [30]. It can also be proven [31] that for the *normalized* discriminant function

$$\widetilde{f}(\mathbf{x}) = \frac{\sum_{t=1}^{T} \alpha^{(t)} h^{(t)}(\mathbf{x})}{\sum_{t=1}^{T} \alpha^{(t)}},$$

the *margin-based* training error

$$\widehat{R}_{\mathbb{I}}^{(\theta)}(\widetilde{f}) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}\left\{\widetilde{f}(\mathbf{x}_i) y_i < \theta\right\}$$

also goes to zero exponentially fast for all $\theta < \widetilde{\rho}^*/2$, where

$$\widetilde{\rho}^* = \max_{\widetilde{\alpha}: \sum_{t=1}^{T} |\widetilde{\alpha}_t| = 1} \min_{i} \sum_{t=1}^{T} \widetilde{\alpha}^{(t)} h^{(t)}(\mathbf{x}_i) y_i$$

is the maximum achievable (normalized) minimum margin. This results shows that AdaBoost, similarly to the support vector machine, leads to a large margin classifier. It also explains the surprising experimental observation that the generalization error $R_{\mathbb{I}}(f)$ (estimated on a test set) often decreases even after the training error $\widehat{R}_{\mathbb{I}}(f)$ reaches 0.

AdaBoost arrived to the pattern recognition scene in the late 90s when support vector machines were in full bloom. AdaBoost has a lot of advantages over its main rival: it is fast (its time complexity is linear in n and T), it is an *any-time* algorithm[9], it has few hyperparameters, it is resistant to overfitting, and it can be directly extended to multi-class classification [32]. Due to these advantages, it rapidly became the method of choice of machine learning experts in certain types of problems where the natural setup is to define a large set of plausible features of which the final solution $f^{(T)}$ uses only a few (a so called *sparse* classifier). Arguably the most famous application is the first face detector running real-time on 30 frame-per-second video [33]. At the same time, AdaBoost is much less known among users with no computer science background than the support vector machine. The reason is paradoxically the simplicity of AdaBoost: since it is so easy to implement with a little background in programming, no machine learning expert took the effort to provide a turn-key software package easily usable for non-experts.[10] In fact it is not surprising that the only large non-computer-scientist community in which AdaBoost is much more popular than SVM is experimental physics: physicists, especially in high energy physics, are heavy computer-users with considerable programming skills. In the last five years AdaBoost (and other similar *ensemble* methods) seem to have been taking over the field of large-scale applications. In recent large-scale challenges [34, 35] the top entries are dominated by ensemble-based solutions, and SVM is almost non-existent in the most efficient approaches.

Although binary AdaBoost (Figure 4) is indeed simple to implement, multi-class AdaBoost has some nontrivial tricks. There are several multi-class extensions of the original binary algorithm, and

[9] It outputs a classifier even if it is stopped before convergence.

[10] We are attempting to improve the situation with our free multi-class program package available at http://multiboost.org.

it is not well-known, even among machine learning experts, that the original AdaBoost.M1 and AdaBoost.M2 algorithms [24] are largely suboptimal compared to AdaBoost.MH published a couple of years later [32].[11] On the other hand, the AdaBoost.MH paper [32] did not specify the implementation details of multi-class base learning, making the implementation non-trivial. For more information on multi-class AdaBoost.MH we refer the reader to the documentation of MultiBoost, available from the http://multiboost.org website.

5 Applications

In this section we illustrate the versatility of the abstract supervised learning model described in this chapter through presenting some of the machine learning applications we have worked on. It turns out that real-world applications are never so simple as just taking a turn-key implementation out of the box and running it. To make a system work, one needs both domain expertise and machine learning expertise, and so it often requires an interdisciplinary approach and considerable communication effort from experts of different backgrounds. Nevertheless, once the problem is reduced to the abstract setup described in Section 1, machine learning algorithms can be very efficient.

As the examples will show, classification or regression is only one step in the pattern recognition pipeline (Figure 5). Data collection is arguably the most costly step in most of the applications. In particular, harvesting the labels y_i usually involves human interaction. How to do this in a cost-effective way by, for example, using CAPTCHAs to obtain character labels [36] or making people label images by playing a game [37, 38], is itself a challenging research subject. On the other hand, in experimental physics simulators can be used to sample from the distribution $p(\mathbf{x} \mid y)$, so in these applications data collection is usually not an issue.

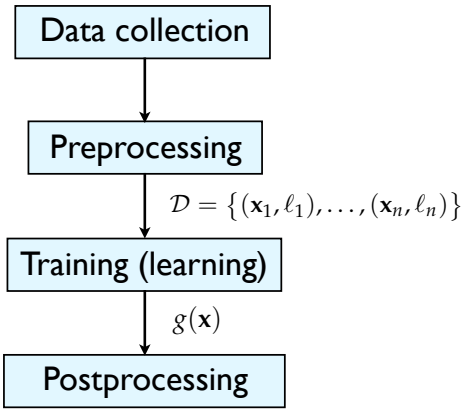


Figure 5. The pattern recognition pipeline.

The second step of the pipeline is preprocessing the data. There is a wide range of techniques that can be applied here, usually with the objective of simplifying the supervised learning task. First, if the observations are not already in a format of a real vector $\mathbf{x} \in \mathbb{R}^d$, *features* (column vectors $(x_1^{(j)}, \ldots, x_n^{(j)})$ of the data matrix) should be extracted from the raw data. The goal here is to find features that are plausibly correlated with the label y, so domain knowledge is almost always necessary

[11] The commonly used WEKA package (http://www.cs.waikato.ac.nz/ml/weka), for example, only contains a badly written implementation of AdaBoost.M1, effectively harming the reputation of boosting as a whole.

to carry out this step. Since learning algorithms usually deteriorate as the dimension of the input $\mathbf{x}$ increases (because of the so called *curse of dimensionality*), dimensionality reduction is often part of data preprocessing. Principal component analysis is the basic tool that is usually applied, but nonlinear manifold algorithms [39, 40] may also be useful if the training set is not too large. The output space C can also be transformed to massage the original problem into a setup that can be solved by standard machine learning tools. This can be done in an ad hoc way, but there exist also principled *reduction* techniques between machine learning problems (binary/multi-class classification, regression, even reinforcement learning) [41].

After training, postprocessing the results can also be an important step. If, for example, the original complex problem was reduced to an easy-to-solve setup, re-transforming the obtained labels and even re-calibrating the solution is often a good idea. In well-documented challenges of the last five years [34, 35, 42] we have also learned that the most competitive results are obtained when divers models trained using different algorithms are combined, so *model aggregation* techniques are also becoming part of the generic machine learning toolbox.

Sometimes it happens that an application poses a specific problem that can not be solved with existing tools. Solving such a problem and generalizing the solution to make it applicable for similar problems is one of the motivational engines behind the development of the machine learning domain.

5.1 Music classification, web page ranking, muon counting

In [43] we use AdaBoost for telling apart speech from music. The system starts by constructing the spectrogram on a set of recorded audio signals which constitute the observation vectors $\mathbf{x}$ in this case. AdaBoost is then run in a feature space inspired by image classification on the spectrogram "images". The output y of the system is binary, that is, $y \in C = \{\text{MUSIC}, \text{SPEECH}\}$. In [44] we stay within the music classification domain but we tackle a more difficult problem: finding the performing artist and the genre of a song based on the audio signal of the first [30]s of the song. The first module of the classifier pipeline is an elaborate signal processor that collects a vector of features for each [2]s segment and then aggregates them to constitute an observation vector $\mathbf{x}$ with about 800 components per song. We train two systems, one for finding the artist performing the song and another for predicting its genre. This application also stretches the limits of the classical multi-class (single-label) setup. It is plausible that a song belongs to several genres leading to a so-called *multi-label* classification. It may also be useful to train a unique system for predicting both the artist and the genre at the same time. This problem is known as *multi-task* learning.

The multi-variate regression problem can be illustrated by our recent work [45] in which we aim to predict the number of muons in a signal recorded in a water Cherenkov detector of the Pierre Auger experiment [46]. The pipeline, again, starts by extracting features that are plausibly correlated with the muon content of the signal. The 172 features are quite correlated, so first we apply principal component analysis to reduce the size of the observation vector $\mathbf{x}$ to 19. Then we train a neural network [1, 26] and convert its output into a point estimate and a pointwise error bar (uncertainty) of the number of muons.

In [47, 48] we use AdaBoost.MH for web page ranking. The input $\mathbf{x}$ of the classifier g is a set of features representing a search query and a document, and the label y represents the relevance of the document to the query. The goal is to rank the documents in order of their relevance. Of course, if the relevance of the document is correctly predicted, the implied ranking will also be good. Nevertheless, it is hard to formally derive a meaningful loss for each (document, query, relevance) triplet from the particular loss defined on rankings. The solution to this problem is a post-processing technique called *calibration*: instead of directly using the output of the classifier g, we send it through another regressor or ranker which is now trained to minimize the desired risk. Our concrete system also contains another

postprocessing module called *model aggregation*: instead of training one classifier g, we train a large number of classifiers by varying data preprocessing and algorithmic hyperparameters, and combine the results using a simple weighted voting scheme.

6 Three open research problems

In this section we briefly describe three open research problems that are either relevant to or even motivated by certain unorthodox applications of multivariate discrimination in experimental physics.

6.1 Trigger design: classification with test-time constraints

One of the recent applications of multivariate discriminants in high-energy physics is trigger design [49]. The goal of a trigger is to separate *signals* generated by a phenomenon to be detected or measured from *background*, which is an observed event that just happen to look like real signal because of random fluctuations or due to some uninteresting phenomena. The final goal of the analysis is to collect a large statistics of observations that, with high probability, were generated by the targeted phenomenon. Since the background is often several orders of magnitudes more frequent than the signal, part of the background/signal separation cannot be done offline. Due to either limited disk capacity or limited bandwidth, most of the raw signal has to be discarded by online triggers.

In the machine learning paradigm, a trigger is just a binary classifier with

$$C = \{\textsc{signal}, \textsc{background}\}.$$

There are several attributes that make the trigger a *special* binary classifier. First, it is extremely unbalanced: the probability of BACKGROUND is practically 1 in a lot of cases. This makes the classical setup of minimizing the error probability $R_{\mathbb{I}}(g)$ (2) inadequate: it is very hard to beat the trivial constant classifier $g(\mathbf{x}) \equiv \textsc{background}$ which has an error of $R_{\mathbb{I}}(g) = P(y = \textsc{signal})$. Indeed, the natural *gain* in trigger design is the *true positive* or *hit* indicator

$$G_{\text{TP}}(g, (\mathbf{x}, y)) = \mathbb{I}\{g(\mathbf{x}) = \textsc{signal} \mid y = \textsc{signal}\}.$$

Taking the complement of the true positive indicator

$$L_{\overline{\text{TP}}}(g, (\mathbf{x}, y)) = 1 - \mathbb{I}\{g(\mathbf{x}) = \textsc{signal} \mid y = \textsc{signal}\} = \mathbb{I}\{g(\mathbf{x}) = \textsc{background} \mid y = \textsc{signal}\}$$

as the loss, the implied risk is

$$R_{\overline{\text{TP}}}(g) = \mathbb{E}_{(\mathbf{x},y)\sim\mathfrak{D}}\{L_{\overline{\text{TP}}}(g, (\mathbf{x}, y))\} = \mathbb{P}\{g(\mathbf{x}) = \textsc{background} \mid y = \textsc{signal}\}.$$

Again, minimizing $R_{\overline{\text{TP}}}(g)$ is trivial by setting $g(\mathbf{x}) \equiv \textsc{signal}$ this time, so the goal is to minimize $R_{\overline{\text{TP}}}(g)$ with a *constraint* that the *false positive* rate

$$R_{\text{FP}}(g) = P(g(\mathbf{x}) = \textsc{signal} \mid y = \textsc{background})$$

is kept below a fixed level p_{FP}. As we mentioned above, in triggers we have $P(y = \textsc{signal}) \ll P(y = \textsc{background})$, so the false positive rate $R_{\text{FP}}(g)$ is approximately equal to the unconditional positive rate $P(g(\mathbf{x}) = \textsc{signal})$. In experimental physics terminology this means that a constraint is imposed on the *trigger rate*.

The second attribute that makes trigger design special is that we have strict computational constraints imposed on the evaluation of the classifier g on test samples $\mathbf{x}$. Typically, observations $\mathbf{x}$ arrive

at a given rate and $g(\mathbf{x})$ has to be run online. The designer can have some flexibility on the parallel handling of the incoming events, but computational resources are often limited. In certain cases [12] the electric consumption may also be limited and harsh conditions may require the use of robust hardware with low clock-rate.

Trigger design shares these two attributes (unbalanced classes and test-time constraints) with object detection in computer vision where machine learning has been applied widely. For example, when the goal is to detect faces in images, the probability of the signal class (face) is much lower then the probability of background (everything else). We have also computational constraints at test time if the goal is to detect faces *online* in video recordings with given frame rate and the detector hardware must fit into a compact camera. What makes trigger design slightly more challenging is the sometimes extremely low signal probability and the fact that the computational cost of each feature (components of $\mathbf{x}$) can vary in a large range. For example, the LHCb trigger [49, 51] can use "cheap" observables that can be evaluated fast to rapidly obtain a rough classifier. One can also almost reconstruct the collision event which can take up a large portion of the allotted time, but the resulting feature can be used reliably to discard background events.

This example shows why a natural answer to these challenges is to design *cascade* classifiers both in experimental physics and in object detection [33, 52–56]. A cascade classifier $g(\mathbf{x})$ is composed of a list of simpler binary classifiers $h_1(\mathbf{x}), \ldots, h_N(\mathbf{x})$ evaluated sequentially. For $j < N$, if $h_j(\mathbf{x})$ classifies the observation $\mathbf{x}$ negatively, the final classification of $g(\mathbf{x})$ is BACKGROUND, and if the output of $h_j(\mathbf{x})$ is positive, the observation is sent to the next $(j+1)$th *stage* (or *level* in the terminology of experimental physics). The classifier $h_N(\mathbf{x})$ at the last stage is the only one that can classify $\mathbf{x}$ as a SIGNAL. In computer vision, the stage classifiers $h_j(\mathbf{x})$ are usually learned using classification algorithms (most often AdaBoost). Some of the newest detection algorithms also attempt to learn the cascade structure automatically, nevertheless, manual experimentation is usually required to set hyperparameters (stagewise false positive/false negative rates, computational complexity of stage classifiers $h_j(\mathbf{x})$). In [57], we present a principled approach that can be used to automatically design test-time constrained classifiers. The automatic design also allows us to go beyond the cascade structure which is, although quite intuitive, an artificial constraint to keep the classifier structure simple and to accommodate manual tuning.

6.2 Learning to discover

The main application of multivariate discrimination in high-energy particle physics is, interestingly, not classification. The goal in these applications is to increase the sensitivity of counting-based statistical tests [58, 59]. Intuitively, the goal is to find regions of the feature space where the signal is present or where it is amplified with respect to its average abundance. Once the subregion is found, we claim the discovery of a novel phenomenon (particle) when the number of events in the region is significantly higher than that predicted by the pure background hypothesis. The simplest formal goal, motivated by a Poisson test, is to maximize

$$G(f) = \frac{s}{\sqrt{b}} \tag{12}$$

[12] For example, in JEM EUSO [50], where the detector will be installed on the International Space Station.

where

$$s = \sum_{i=1}^{n} \mathbb{I}\{f(\mathbf{x}_i) = \text{SIGNAL} \wedge y_i = \text{SIGNAL}\}$$
$$b = \sum_{i=1}^{n} \mathbb{I}\{f(\mathbf{x}_i) = \text{SIGNAL} \wedge y_i = \text{BACKGROUND}\}.$$

Although $G(f)$ has the right "flavor", it does not take into consideration the statistical fluctuations of the test when s and/or b are small, so other, more sophisticated, approximate criteria have also been derived [60]. Whereas all these criteria (including (12)) are clearly related to the classical classification error $R_{\mathbb{I}}(f)$, the two are not equivalent. A notable difference is that the *expectation* of G depends on the sample size n, whereas increasing sample size only decreases the *variance* of $R_{\mathbb{I}}$. Nevertheless, the standard practice is to learn a discriminant function $f : \mathbb{R}^d \to \mathbb{R}$ on $\mathcal{D}$ using standard classification methods that minimize the classification error R, and then use G only to optimize a threshold θ which defines the function g_θ by

$$g_\theta(\mathbf{x}) = \begin{cases} \text{SIGNAL} & \text{if } f(\mathbf{x}) > \theta \\ \text{BACKGROUND} & \text{otherwise.} \end{cases}$$

This way of using multivariate discriminants raises several interesting research questions. First, given a concrete test, what should the training criteria G be? Second, what is the relationship between G and $R_{\mathbb{I}}(f)$? Finally, how to adapt classical classification algorithms to the given criteria?

6.3 Deep learning for automatic feature construction

Finally let us raise a completely open research question. The current technology of multivariate discrimination allows us to classify objects that are represented by vectors $\mathbf{x} = (x^{(1)}, \ldots, x^{(d)})$ of fixed length d. The representation has to be "semantically aligned", that is, $x_1^{(j)}$ must be comparable to $x_2^{(j)}$ in two objects $\mathbf{x}_1$ and $\mathbf{x}_2$. At the same time, raw observation, be it pixels of an image or electronic channels in a detector event, seldom comes in this form. Today, the process of translating the raw observation into a semantically aligned vector of features is in the hands of the domain expert. One of the most important movements of the last decade in machine learning is the development of a family of techniques for building deep unsupervised neural architectures [13]. The goal is to construct representations in a mostly non-supervised way that can capture deep invariances in the raw data. Although the field is rather young, it already had a major impact on natural language processing [14] and computer vision [15]. Interesting and natural questions are whether these techniques can be adapted to scientific data, whether the invariances in, say, raw detector data are sufficient to construct representations that could be useful in the ultimate analysis, and whether automatic or semi-automatic feature extraction can improve on the manually constructed representations.

References

[1] C. M. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press, 1995.
[2] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.
[3] N. Cristianini and J. Shawe-Taylor, *Kernel methods for pattern recognition*, Cambridge University Press, 2004.
[4] L. Devroye, L. Györfi, and G. Lugosi, *A Probabilistic Theory of Pattern Recognition*, Springer, New York, 1996.

[5] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer-Verlag, 2009.

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

[7] Y. Freund and R.E. Schapire, *Boosting: Foundations and Algorithms*, MIT Press, Cambridge, MA, 2012.

[8] V. N. Vapnik, *The Nature of Statistical Learning Theory*, Springer-Verlag, New York, 1995.

[9] V. N. Vapnik, *Statistical Learning Theory*, Wiley, New York, 1998.

[10] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, 2012.

[11] Y. Bengio, "Gradient-based optimization of hyperparameters," *Neural Computation*, vol. 12, no. 8, pp. 1889–1900, 2000.

[12] C. E. Rasmussen and C. K. I. Williams, *http://www.gaussianprocess.org/gpmlGaussian Processes for Machine Learning*, MIT Press, 2006.

[13] Y. Bengio, A.C. Courville, and P. Vincent, "Unsupervised feature learning and deep learning: A review and new perspectives," *CoRR*, vol. abs/1206.5538, 2012.

[14] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research*, vol. 12, pp. 2493–2537, 2011.

[15] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*. 2012, vol. 25, MIT Press.

[16] J. Bergstra, R. Bardenet, B. Kégl, and Y. Bengio, "Algorithms for hyperparameter optimization," in *Advances in Neural Information Processing Systems (NIPS)*. The MIT Press, 2011, vol. 24.

[17] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *Advances in Neural Information Processing Systems*, 2012, vol. 25.

[18] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA: Automated selection and hyper-parameter optimization of classification algorithms," Tech. Rep., http://arxiv.org/abs/1208.3719, 2012.

[19] R. Bardenet, M. Brendel, B. Kégl, and M. Sebag, "Collaborative hyperparameter tuning," in *International Conference on Machine Learning (ICML)*, 2013.

[20] D. S. Johnson and F. P. Preparata, "The densest hemisphere problem," *Theoretical Computer Science*, vol. 6, pp. 93–107, 1978.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[22] B. Boser, I. Guyon, and V. Vapnik, "A training algorithm for optimal margin classifiers," in *Fifth Annual Workshop on Computational Learning Theory*, 1992, pp. 144–152.

[23] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[24] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, pp. 119–139, 1997.

[25] P. Bartlett, "The sample complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network," *IEEE Transactions on Information Theory*, vol. 44, no. 2, pp. 525–536, 1998.

[26] I. Nabney, *Netlab: Algorithms for Pattern Recognition*, Springer, 2002.

[27] L. Bottou and O. Bousquet, "The tradeoffs of large scale learning," in *Advances in Neural Information Processing Systems*, 2008, vol. 20, pp. 161–168.

[28] L. Mason, P. Bartlett, J. Baxter, and M. Frean, "Boosting algorithms as gradient descent," in *Advances in Neural Information Processing Systems*. 2000, vol. 12, pp. 512–518, The MIT Press.

[29] L. Mason, P. Bartlett, and J. Baxter, "Improved generalization through explicit optimization of margins," *Machine Learning*, vol. 38, no. 3, pp. 243–255, March 2000.

[30] M. Collins, R.E. Schapire, and Y. Singer, "Logistic regression, AdaBoost and Bregman distances," *Machine Learning*, vol. 48, pp. 253–285, 2002.

[31] R. E. Schapire, Y. Freund, P. Bartlett, and W. S. Lee, "Boosting the margin: a new explanation for the effectiveness of voting methods," *Annals of Statistics*, vol. 26, no. 5, pp. 1651–1686, 1998.

[32] R. E. Schapire and Y. Singer, "Improved boosting algorithms using confidence-rated predictions," *Machine Learning*, vol. 37, no. 3, pp. 297–336, 1999.

[33] P. Viola and M. Jones, "Robust real-time face detection," *International Journal of Computer Vision*, vol. 57, pp. 137–154, 2004.

[34] G. Dror, M. Boullé, I. Guyon, V. Lemaire, and D. Vogel, Eds., *Proceedings of KDD-Cup 2009 competition*, vol. 7 of *JMLR Workshop and Conference Proceedings*, 2009.

[35] Olivier Chapelle and Yi Chang, "Yahoo! Learning-to-Rank Challenge overview," in *Yahoo! Learning-to-Rank Challenge (JMLR W&CP)*, 2011, vol. 14, pp. 1–24.

[36] L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and Blum M., "reCAPTCHA: Human-based character recognition via web security measures," *Science*, vol. 321, no. 5895, pp. 1465–1468, 2008.

[37] L. Von Ahn and L. Dabbish, "Labeling images with a computer game," in *Conference on Human factors in computing systems (CHI04)*, 2004, pp. 319–326.

[38] L. Von Ahn, "Games with a purpose," *Computer*, vol. 39, no. 6, 2006.

[39] S. Roweis and Saul L. K., "Nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 290, pp. 2323–2326, 2000.

[40] J. B. Tenenbaum, V. de Silva, and Langford J. C., "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 290, pp. 2319–2323, 2000.

[41] A. Beygelzimer, J. Langford, and B. Zadrozny, *Performance Modeling and Engineering*, chapter Machine Learning Techniques–Reductions Between Prediction Quality Metrics, Springer, 2008.

[42] J. Bennett and S. Lanning, "The Netflix prize," in *KDDCup 2007*, 2007.

[43] N. Casagrande, D. Eck, and B. Kégl, "Geometry in sound: A speech/music audio classifier inspired by an image classifier," in *International Computer Music Conference*, Sept. 2005, vol. 17.

[44] J. Bergstra, N. Casagrande, D. Erhan, D. Eck, and B. Kégl, "Aggregate features and AdaBoost for music classification," *Machine Learning Journal*, vol. 65, no. 2/3, pp. 473–484, 2006.

[45] B. Kégl, R. Busa-Fekete, K. Louedec, R. Bardenet, X. Garrido, I.C. Mariş, D. Monnier-Ragaigne, S. Dagoret-Campagne, and M. Urban, "Reconstructing $N_{\mu 19}(1000)$," Tech. Rep. 2011-054, Auger Project Technical Note, 2011.

[46] Pierre Auger Collaboration, "Pierre Auger project design report," Tech. Rep., Pierre Auger Observatory, 1997.

[47] R. Busa-Fekete, B. Kégl, T. Éltető, and Gy. Szarvas, "Ranking by calibrated AdaBoost," in *Yahoo! Ranking Challenge 2010 (JMLR workshop and conference proceedings)*, 2011, vol. 14, pp. 37–48.

[48] R. Busa-Fekete, B. Kégl, T. Éltető, and Gy. Szarvas, "A robust ranking methodology based on diverse calibration of AdaBoost," in *European Conference on Machine Learning*, 2011, vol. 22, pp. 263–279.

[49] V. Gligorov and M. Williams, "Efficient, reliable and fast high-level triggering using a bonsai boosted decision tree," Tech. Rep., arXiv:1210.6861, 2012.

[50] Y. Takizawa, T. Ebisuzaki, Y. Kawasaki, M. Sato, M. E. Bertaina, H. Ohmori, Y. Takahashi, F. Kajino, M. Nagano, N. Sakaki, N. Inoue, H. Ikeda, Y. Arai, Y. Takahashi, T. Murakami, James H. Adams, and the JEM-EUSO Collaboration, "JEM-EUSO: Extreme Universe Space Observatory on JEM/ISS," *Nuclear Physics B - Proceedings Supplements*, vol. 166, pp. 72–76, 2007.

[51] V. Gligorov, "A single track HLT1 trigger," Tech. Rep. LHCb-PUB-2011-003, CERN, 2011.

[52] L. Bourdev and J. Brandt, "Robust object detection via soft cascade," in *Conference on Computer Vision and Pattern Recognition*. 2005, vol. 2, pp. 236–243, IEEE Computer Society.

[53] R. Xiao, L. Zhu, and H. J. Zhang, "Boosting chain learning for object detection," in *Ninth IEEE International Conference on Computer Vision*, 2003, vol. 9, pp. 709–715.

[54] J. Sochman and J. Matas, "WaldBoost – learning for time constrained sequential detection," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005, pp. 150–156.

[55] B. Póczos, Y. Abbasi-Yadkori, Cs. Szepesvári, R. Greiner, and N. Sturtevant, "Learning when to stop thinking and do something!," in *Proceedings of the 26th International Conference on Machine Learning*, 2009, pp. 825–832.

[56] M. Saberian and N. Vasconcelos, "Boosting classifier cascades," in *Advances in Neural Information Processing Systems 23*. 2010, pp. 2047–2055, MIT Press.

[57] D. Benbouzid, R. Busa-Fekete, and B. Kégl, "Fast classification using sparse decision DAGs," in *International Conference on Machine Learning*, June 2012, vol. 29.

[58] V. M. Abazov et al., "Observation of single top-quark production," *Physical Review Letters*, vol. 103, no. 9, 2009.

[59] Aaltonen, T. et. al, "Observation of electroweak single top-quark production," *Phys. Rev. Lett.*, vol. 103, pp. 092002, Aug 2009.

[60] G. Cowan, K. Cranmer, E. Gross, and O. Vitells, "Asymptotic formulae for likelihood-based tests of new physics," *The European Physical Journal C*, vol. 71, pp. 1–19, 2011.