Chapter 8

Linear Classifiers

Suppose we are interested in building a linear classifer $f : \mathbb{R}^d \to \{-1, +1\}$. Linear classifiers try to find the relationship between inputs and outputs by constructing a linear function (a point, a line, a plane or a hyperplane) that splits $\mathbb{R}^d$ into two half-spaces. The two half-spaces act as decision regions for the positive and negative examples, respectively. Given a data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ consisting of positive and negative examples, there are many ways in which linear classifiers can be constructed. For example, a training algorithm may explicitly work to position the decision surface in order to separate positive and negative examples according to some problem-relevant criteria; e.g., it may try to minimize the fraction of examples on the incorrect side of the decision surface. Alternatively, the goal of the training algorithm may be to directly estimate the posterior distribution $p(y|\mathbf{x})$, in which case the algorithm is more likely to rely on the formal parameter estimation principles; e.g., it may maximize the likelihood. An example of a classifier with a linear decision surface is shown in Figure 8.1.

To simplify the formalism in the following sections, we will add a component $x_0 = 1$ to each input $(x_1, \ldots, x_d)$. This extends the input space to $\mathcal{X} = \mathbb{R}^{d+1}$ but, fortunately, it also leads us to a simplified notation in which the decision boundary in $\mathbb{R}^d$ can be written as $\mathbf{w}^\top \mathbf{x} = 0$, where $\mathbf{w} = (w_0, w_1, \ldots, w_d)$ is a set of weights and $\mathbf{x} = (x_0 = 1, x_1, \ldots, x_d)$ is any element of the input space. Nevertheless, we should remember that the actual inputs are d-dimensional.

Earlier in the introductory remarks, we presented a classifier as a function $f : \mathcal{X} \to \mathcal{Y}$ and have transformed the learning problem into approximating $p(y|\mathbf{x})$. In the case of

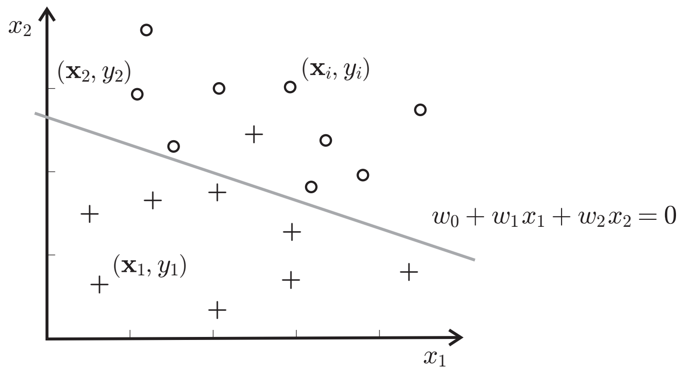

Figure 8.1: A data set in $\mathbb{R}^2$ consisting of nine positive and nine negative examples. The gray line represents a linear decision surface in $\mathbb{R}^2$. The decision surface does not perfectly separate positives from negatives.

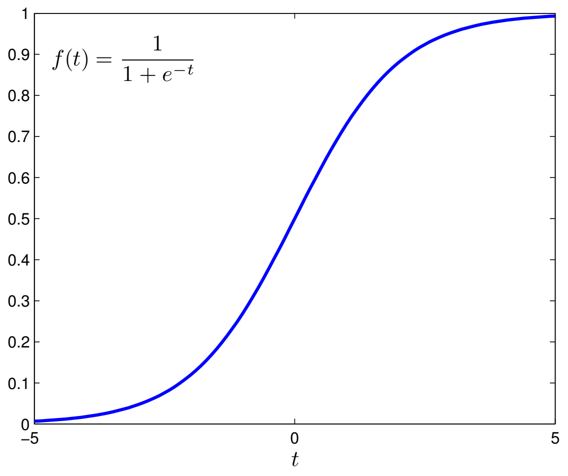

Figure 8.2: Sigmoid function in $[-5, 5]$ *interval.*

linear classifiers, our flexibility is restricted because our method must learn the posterior probabilities $p(y|\mathbf{x})$ and at the same time have a linear decision surface in $\mathbb{R}^d$. This, however, can be achieved if $p(y|\mathbf{x})$ is modeled as a monotonic function of $\mathbf{w}^\top\mathbf{x}$; e.g., $\tanh(\mathbf{w}^\top\mathbf{x})$ or $(1+e^{-\mathbf{w}^\top\mathbf{x}})^{-1}$. Of course, a model trained to learn posterior probabilities $p(y|\mathbf{x})$ can be seen as a "soft" predictor or a scoring function $s : \mathcal{X} \to [0, 1]$. Then, the conversion from s to f is a straightforward application of the maximum a posteriori principle: the predicted output is positive if $s(\mathbf{x}) \geq 0.5$ and negative if $s(\mathbf{x}) < 0.5$. More generally, the scoring function can be any mapping $s : \mathcal{X} \to \mathbb{R}$, with thresholding applied based on any particular value τ.

8.1 Logistic regression

Let us consider binary classification in $\mathbb{R}^d$, where $\mathcal{X} = \mathbb{R}^{d+1}$ and $\mathcal{Y} = \{0, 1\}$. The basic idea for many classification approaches is to hypothesize a closed-form representation for the posterior probability that the class label is positive and learn parameters $\mathbf{w}$ from data. In logistic regression, this relationship can be expressed as

$$P(Y = 1|\mathbf{x}, \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{w}^\top\mathbf{x}}}, \tag{8.1}$$

which is a monotonic function of $\mathbf{w}^\top\mathbf{x}$. Function $f(t) = \left(1 + e^{-t}\right)^{-1}$ is called the sigmoid function or the logistic function and is plotted in Figure 8.2. It allows us to map the input $\mathbf{w}^\top\mathbf{x} \in (-\infty, \infty)$ to $[0, 1]$ using a monotonic transformation.

8.1.1 Maximum conditional likelihood estimation

To frame the learning problem as parameter estimation, we will assume that the data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ is an i.i.d. sample from a fixed but unknown probability distribution $p(\mathbf{x}, y)$. Even more specifically, we will assume that the data generating process randomly draws a data point $\mathbf{x}$, a realization of the random vector $(X_0 = 1, X_1, \ldots, X_d)$, according to $p(\mathbf{x})$ and then sets its class label Y according to the Bernoulli distribution

$$p(y|\mathbf{x}) = \begin{cases} \left(\frac{1}{1+e^{-\boldsymbol{\omega}^\top \mathbf{x}}}\right)^y & \text{for } y = 1 \\ \left(1 - \frac{1}{1+e^{-\boldsymbol{\omega}^\top \mathbf{x}}}\right)^{1-y} & \text{for } y = 0 \end{cases} \tag{8.2}$$

where $\boldsymbol{\omega} = (\omega_0, \omega_1, \ldots, \omega_d)$ is a set of unknown coefficients we want to recover (or learn) from the observed data $\mathcal{D}$. Based on the principles of parameter estimation, we can estimate $\boldsymbol{\omega}$ by maximizing the conditional likelihood of the observed class labels $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ given the inputs $\mathbf{X} = (\mathbf{x}_1^\top, \mathbf{x}_2^\top, \ldots, \mathbf{x}_n^\top)$.

We shall first write the conditional likelihood function $p(\mathbf{y}|\mathbf{X}, \mathbf{w})$, or simply $l(\mathbf{w})$, as

$$l(\mathbf{w}) = \prod_{i=1}^{n} p(y_i|\mathbf{x}_i, \mathbf{w}). \tag{8.3}$$

This function can be thought of as the probability of observing a set of labels $\mathbf{y}$ given the set of data points $\mathbf{X}$ and the particular set of coeficients $\mathbf{w}$. However, compared to Eq. (8.2) where for a given $\mathbf{x}$ and $\boldsymbol{\omega}$

$$\int p(y|\mathbf{x}, \boldsymbol{\omega}) dy = 1,$$

here we have that for a given $\mathbf{x}$ and y

$$\int p(y|\mathbf{x}, \mathbf{w}) d\mathbf{w} \neq 1.$$

This means that the likelihood is not a probability distribution over the domain of $\mathbf{w}$. More formally, we define the parameter vector that maximizes the likelihood as

$$\begin{aligned} \mathbf{w}_{\text{ML}} &= \arg\max_{\mathbf{w}} \{l(\mathbf{w})\} \\ &= \arg\max_{\mathbf{w}} \left\{ \prod_{i=1}^{n} p(y_i|\mathbf{x}_i, \mathbf{w}) \right\}. \end{aligned} \tag{8.4}$$

By combining Eqs. (8.2-8.3) we can now express the likelihood function as

$$l(\mathbf{w}) = \prod_{i=1}^{n} \left(\frac{1}{1+e^{-\mathbf{w}^\top \mathbf{x}_i}}\right)^{y_i} \cdot \left(1 - \frac{1}{1+e^{-\mathbf{w}^\top \mathbf{x}_i}}\right)^{1-y_i}. \tag{8.5}$$

Note that maximizing Eq. (8.5) is equivalent to maximizing the log-likelihood function $ll(\mathbf{w}) = \log(l(\mathbf{w}))$

$$ll(\mathbf{w}) = \sum_{i=1}^{n} \left(y_i \cdot \log\left(\frac{1}{1+e^{-\mathbf{w}^\top \mathbf{x}_i}}\right) + (1 - y_i) \cdot \log\left(1 - \frac{1}{1+e^{-\mathbf{w}^\top \mathbf{x}_i}}\right)\right). \tag{8.6}$$

The negative of the log-likelihood from Eq. (8.6) is sometimes referred to as cross-entropy; thus, cross-entropy minimization is equivalent to the maximum likelihood method. To make everything more suitable for further steps, we will slightly rearrange Eq. (8.6) as

$$ll(\mathbf{w}) = \sum_{i=1}^{n} \left((y_i - 1)\, \mathbf{w}^\top \mathbf{x}_i + \log\left(\frac{1}{1+e^{-\mathbf{w}^\top \mathbf{x}_i}}\right)\right). \tag{8.7}$$

It does not take much effort to realize that there is no closed-form solution to $\nabla ll(\mathbf{w}) = \mathbf{0}$ (we did have this luxury in linear regression, but not here). Thus, we have to proceed with iterative optimization methods. That is, our goal is to calculate the gradient ($\nabla ll(\mathbf{w})$) and Hessian ($H_{ll(\mathbf{w})}$) in order to specify the update rule described by Newton-Raphson's method, as a function of inputs $\mathbf{X}$, class labels $\mathbf{y}$, and the current parameter vector. We can calculate the first and second partial derivatives of $ll(\mathbf{w})$ as follows

$$\begin{aligned}
\frac{\partial ll(\mathbf{w})}{\partial w_j} &= \sum_{i=1}^{n} \left((y_i - 1) \cdot x_{ij} - \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}_i}} \cdot e^{-\mathbf{w}^\top \mathbf{x}_i} \cdot (-x_{ij}) \right) \\
&= \sum_{i=1}^{n} x_{ij} \cdot \left(y_i - 1 + \frac{e^{-\mathbf{w}^\top \mathbf{x}_i}}{1 + e^{-\mathbf{w}^\top \mathbf{x}_i}} \right) \\
&= \sum_{i=1}^{n} x_{ij} \cdot \left(y_i - \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}_i}} \right) \\
&= \mathbf{f}_j^\top \left(\mathbf{y} - \mathbf{p} \right),
\end{aligned}$$

where $\mathbf{f}_j$ is the j-th column (feature) of data matrix $\mathbf{X}$, $\mathbf{y}$ is an n-dimensional column vector of class labels and $\mathbf{p}$ is an n-dimensional column vector of (estimated) posterior probabilities $p_i = P(Y_i = 1|\mathbf{x}_i, \mathbf{w})$, for $i = 1, ..., n$. Considering partial derivatives for every component of $\mathbf{w}$, we have

$$\nabla ll(\mathbf{w}) = \mathbf{X}^T \left(\mathbf{y} - \mathbf{p} \right). \tag{8.8}$$

This has worked well because $\mathbf{e} = \mathbf{y} - \mathbf{p}$ is the error vector and the zero error suggests that the gradient is also a zero vector. The second partial derivative of the log-likelihood function can be found as

$$\begin{aligned}
\frac{\partial^2 ll(\mathbf{w})}{\partial w_j \partial w_k} &= \sum_{i=1}^{n} x_{ij} \cdot \frac{e^{-\mathbf{w}^\top \mathbf{x}_i}}{\left(1 + e^{-\mathbf{w}^\top \mathbf{x}_i}\right)^2} \cdot (-x_{ik}) \\
&= -\sum_{i=1}^{n} x_{ij} \cdot \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}_i}} \cdot \left(1 - \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}_i}} \right) \cdot x_{ik} \\
&= -\mathbf{f}_j^\top \mathbf{P} \left(\mathbf{I} - \mathbf{P} \right) \mathbf{f}_k,
\end{aligned}$$

where $\mathbf{P}$ is an $n \times n$ diagonal matrix with $P_{ii} = p_i = P(Y_i = 1|\mathbf{x}_i, \mathbf{w})$ and $\mathbf{I}$ is an $n \times n$ identity matrix. The Hessian matrix $H_{ll(\mathbf{w})}$ can now be calculated as

$$H_{ll(\mathbf{w})} = -\mathbf{X}^T \mathbf{P} \left(\mathbf{I} - \mathbf{P} \right) \mathbf{X}. \tag{8.9}$$

This is a negative semi-definite matrix, which guarantees that the optimum we find will be a global maximum. Negative semi-definite Hessian corresponds to a concave likelihood function and has a global maximum (unique if negative definite). Substituting Eqs. (8.8-8.9) into Newton-Raphson's method results in the following weight update rule

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \left(\mathbf{X}^\top \mathbf{P}^{(t)} \left(\mathbf{I} - \mathbf{P}^{(t)} \right) \mathbf{X} \right)^{-1} \mathbf{X}^\top \left(\mathbf{y} - \mathbf{p}^{(t)} \right), \tag{8.10}$$

where the initial weights $\mathbf{w}^{(0)}$ can be calculated using the ordinary least squares regression as $\mathbf{w}^{(0)} = \left(\mathbf{X}^\top \mathbf{X} \right)^{-1} \mathbf{X}^\top \mathbf{y}$. Note that the second term on the right-hand side of Eq. (8.10) is calculated in each iteration t; thus we wrote $\mathbf{P}$ and $\mathbf{p}$ as $\mathbf{P}^{(t)}$ and $\mathbf{p}^{(t)}$, respectively, to indicate that $\mathbf{w}^{(t)}$ was used to calculate them. The computational complexity of this procedure is $O(d^3 + d^2 n)$ in each iteration, assuming $O(d^3)$ time for finding matrix inverses.

Weighted conditional likelihood function

In certain situations, it may be justified to allow for unequal importance of each data point. This modifies the conditional likelihood function from Eq. (8.5) to

$$l(\mathbf{w}) = \prod_{i=1}^{n} p_i^{c_i y_i} \cdot (1 - p_i)^{c_i(1-y_i)},$$

where $0 \leq c_i \leq 1$ is a cost for data point i. Taking that $\mathbf{C} = \operatorname{diag}(c_1, c_2, \ldots, c_n)$ we can now express the gradient of the log-likelihood as

$$\nabla ll(\mathbf{w}) = \mathbf{X}^\top \mathbf{C}(\mathbf{y} - \mathbf{p})$$

and the Hessian as

$$H_{ll(\mathbf{w})} = -\mathbf{X}^\top \mathbf{C}\mathbf{P}(\mathbf{I} - \mathbf{P})\mathbf{X}.$$

It is interesting to observe that the Hessian remains negative semi-definite. Thus, the update rule is expected to converge to a global maximum.

8.1.2 Minimizing Euclidean distance

Another approach that is frequently considered is minimization of the Euclidean distance between a vector of class labels $\mathbf{y}$ and a vector of model outputs $\mathbf{p} = (p_1, p_2, ..., p_n)$, where $p_i = P(Y_i = 1|\mathbf{x}_i, \mathbf{w})$. This is equivalent to minimizing the squared error function $E(\mathcal{D}, \mathbf{w})$ or $E(\mathbf{w})$ as

$$\begin{aligned} E(\mathbf{w}) &= \sum_{i=1}^{n} (y_i - p_i)^2 \\ &= \sum_{i=1}^{n} e_i^2, \end{aligned}$$

where $e_i = y_i - p_i$ is the error term that corresponds to a training data point $\mathbf{x}_i$. The minimization of $E(\mathbf{w})$ is formally expressed as

$$\begin{aligned} \mathbf{w}^* &= \arg\min_{\mathbf{w}} \{E(\mathbf{w})\} \\ &= \arg\min_{\mathbf{w}} \left\{ \sum_{i=1}^{n} (y_i - p_i)^2 \right\}. \end{aligned} \tag{8.11}$$

Similar to the maximum likelihood process, our goal will be to calculate the gradient vector and the Hessian of the error function. The partial derivatives of the error function can be calculated as follows

$$\begin{aligned}
\frac{\partial E(\mathbf{w})}{\partial w_j} &= \frac{\partial}{\partial w_j}\sum_{i=1}^{n} e_i^2 \\
&= \sum_{i=1}^{n} 2\cdot e_i \cdot \frac{\partial e_i}{\partial w_j} \\
&= 2\cdot\sum_{i=1}^{n}\left(y_i - \frac{1}{1+e^{-\mathbf{w}^\top\mathbf{x}_i}}\right)\cdot\frac{1}{(1+e^{-\mathbf{w}^\top\mathbf{x}_i})^2}\cdot e^{-\mathbf{w}^\top\mathbf{x}_i}\cdot(-x_{ij}) \\
&= -2\cdot\sum_{i=1}^{n} x_{ij}\cdot\frac{1}{1+e^{-\mathbf{w}^\top\mathbf{x}_i}}\cdot\left(1-\frac{1}{1+e^{-\mathbf{w}^\top\mathbf{x}_i}}\right)\cdot\left(y_i-\frac{1}{1+e^{-\mathbf{w}^\top\mathbf{x}_i}}\right) \\
&= -2\mathbf{f}_j^\top\mathbf{P}\left(\mathbf{I}-\mathbf{P}\right)\left(\mathbf{y}-\mathbf{p}\right).
\end{aligned}$$

This provides the gradient vector in the following form

$$\nabla E(\mathbf{w}) = -2\mathbf{X}^\top\mathbf{P}\left(\mathbf{I}-\mathbf{P}\right)\left(\mathbf{y}-\mathbf{p}\right).$$

Matrix $\mathbf{J}=\mathbf{P}\left(\mathbf{I}-\mathbf{P}\right)\mathbf{X}$ is referred to as Jacobian. In general, Jacobian is an $n\times d$ matrix calculated as

$$J_{E(\mathbf{w})} = \begin{bmatrix} \frac{\partial e_1}{\partial w_1} & \frac{\partial e_1}{\partial w_2} & \cdots & \frac{\partial e_1}{\partial w_d} \\ \vdots & \ddots & & \\ & & & \\ \frac{\partial e_n}{\partial w_1} & & & \frac{\partial e_n}{\partial w_d} \end{bmatrix}.$$

The second partial derivative of the error function can be found as

$$\begin{aligned}
\frac{\partial^2 E(\mathbf{w})}{\partial w_j \partial w_k} &= 2\cdot\sum_{i=1}^{n}\frac{\partial e_i}{\partial w_k}\cdot\frac{\partial e_i}{\partial w_j} + e_i\cdot\frac{\partial^2 e_i}{\partial w_j\partial w_k} \\
&= 2\cdot\sum_{i=1}^{n} x_{ij}\cdot\Big(p_i^2(1-p_i)^2 + p_i\cdot(1-p_i)\cdot(2p_i-1)\cdot(y_i-p_i)\Big)\cdot x_{ik}.
\end{aligned}$$

Thus, the Hessian can be computed as

$$\begin{aligned}
H_{E(\mathbf{w})} &= 2\mathbf{X}^\top\left(\mathbf{I}-\mathbf{P}\right)^\top\mathbf{P}^\top\mathbf{P}\left(\mathbf{I}-\mathbf{P}\right)\mathbf{X} + 2\mathbf{X}^\top\left(\mathbf{I}-\mathbf{P}\right)^\top\mathbf{P}^\top\mathbf{E}(2\mathbf{P}-\mathbf{I})\mathbf{X} \\
&= 2\mathbf{J}^\top\mathbf{J} + 2\mathbf{J}^\top\mathbf{E}(2\mathbf{P}-\mathbf{I})\mathbf{X},
\end{aligned}$$

where $\mathbf{P}=\text{diag}\left\{\mathbf{p}\right\}$, $\mathbf{E}=\text{diag}\left\{\mathbf{e}\right\}$ is a diagonal matrix containing elements $E_{ii}=e_i=y_i-p_i$ and $\mathbf{I}$ is an identity matrix. Thus, the process of minimizing the Euclidean distance between $\mathbf{y}$ and $\mathbf{p}$ results in the following update rule

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \left(\mathbf{J}^{(t)\top}\mathbf{J}^{(t)} + \mathbf{J}^{(t)\top}\mathbf{E}^{(t)}(2\mathbf{P}^{(t)}-\mathbf{I})\mathbf{X}\right)^{-1}\mathbf{X}^\top\mathbf{P}^{(t)}\left(\mathbf{I}-\mathbf{P}^{(t)}\right)\left(\mathbf{y}-\mathbf{p}^{(t)}\right),$$

where $\mathbf{w}^{(0)}$ can be calculated using ordinary least squares regression or assigned randomly.

An interesting problem here is that the Hessian is not guaranteed to be positive semi-definite. This suggests that $E(\mathbf{w})$ is not convex; i.e., it must have multiple minima with different values of the objective function. Finding a global optimum depends on how favorable the initial solution $\mathbf{w}^{(0)}$ is and how well the weight update step can escape local minima to find better ones. This difficulty can be mitigated when

$$\left|\frac{\partial e_i}{\partial w_j} \cdot \frac{\partial e_i}{\partial w_k}\right| \gg \left|e_i \cdot \frac{\partial^2 e_i}{\partial w_j \partial w_k}\right|$$

because the Hessian can be computed as

$$H_{E(\mathbf{w})} \approx 2\mathbf{J}^\top \mathbf{J}.$$

Such an approach is referred to as Gauss-Newton optimization. This Hessian is provably positive semi-definite, but the error function has effectively been changed.

Furthermore, assuming that $H_{E(\mathbf{w})} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix, results in a so-called gradient descent rule that is often used in unconstrained optimization. Gradient descent occasionally suffers from instability and is modified into the following update rule

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \eta \mathbf{X}^\top \mathbf{P}^{(t)} \left(\mathbf{I} - \mathbf{P}^{(t)}\right) \left(\mathbf{y} - \mathbf{p}^{(t)}\right),$$

where η is a positive constant smaller than one. The computational complexity necessary for each step of the gradient descent method is $O(dn^2)$.

8.1.3 Stochastic mode of optimization

We have derived that the weight update rule depends on the data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ and the predictions on all training examples using the weight vector from the current step. We will first rewrite the update rule of the gradient descent method as

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \Delta\mathbf{w}^{(t)},$$

where

$$\Delta\mathbf{w}^{(t)} = \eta \mathbf{X}^\top \mathbf{P}^{(t)} \left(\mathbf{I} - \mathbf{P}^{(t)}\right) \left(\mathbf{y} - \mathbf{p}^{(t)}\right).$$

To simplify the following steps, we will remove the designation of step t and rewrite $\Delta\mathbf{w}$ as

$$\Delta\mathbf{w} = \eta \cdot \begin{bmatrix} p_1(1-p_1)x_{11} & p_2(1-p_2)x_{21} & \cdots & p_n(1-p_n)x_{n1} \\ p_1(1-p_1)x_{12} & p_2(1-p_2)x_{22} & & \\ \vdots & & \ddots & \\ p_1(1-p_1)x_{1d} & & & p_n(1-p_n)x_{nd} \end{bmatrix} \cdot \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix}$$

From here, we can see that the weight update is simply a linear combination of training data points

$$\Delta\mathbf{w} = \eta \sum_{i=1}^{n} e_i p_i (1-p_i) \mathbf{x}_i,$$

This leads us to an interesting modification of the learning method. When the training data is large, it may be beneficial to update the weight vector after each data point is presented to the learning algorithm. That is, if data point $\mathbf{x}_i$ with its class label y_i is presented to the learner, we can modify the weight update to be

$$\Delta\mathbf{w} = \eta e_i p_i (1-p_i) \mathbf{x}_i.$$

A method where weights are updated after seeing each individual data point is referred to as *incremental* or *stochastic gradient descent* method. Alternatively, the training algorithm that utilizes all data points is frequently referred to as *batch* mode of training. The training algorithm can now be revised to randomly draw one data point at a time from $\mathcal{D}$ and then update the current weights using the previous equation. The algorithm stops when the weight vector converges.

Observe that both stochastic and batch modes have the following property: the influence of each data point on the weight update depends on how close the data point is to the separation hyperplane and whether it lies on the correct side of it. The points on the correct side but far away from the decision boundary have negligible influence on $\Delta\mathbf{w}$ (this is because $p_i(1-p_i) \approx 0$ and $e_i \approx 0$), the points on the incorrect side and far away from the decision boundary have a relatively larger influence (this is because $p_i(1-p_i) \approx 0$ and $|e_i| \approx 1$), whereas the points close to the decision boundary have the most individual influence (this is because $p_i(1-p_i) \approx 0.25$ and $|e_i| \approx 0.5$).

8.1.4 Predicting class labels

For a previously unseen data point $\mathbf{x}$ and a set of coefficients $\mathbf{w}^*$ found from Eq. (8.4) or Eq. (8.11), we simply calculate the posterior probability as

$$P(Y=1|\mathbf{x},\mathbf{w}^*) = \frac{1}{1+e^{-\mathbf{w}^{*\top}\cdot\mathbf{x}}}.$$

If $P(Y=1|\mathbf{x},\mathbf{w}^*) \geq 0.5$ we conclude that data point $\mathbf{x}$ should be labeled as positive ($\hat{y}=1$). Otherwise, if $P(Y=1|\mathbf{x},\mathbf{w}^*) < 0.5$, we label the data point as negative ($\hat{y}=0$). We can see this predictor as a simple mathematical function or as a computer code that outputs $P(Y=1|\mathbf{x},\mathbf{w}^*)$. Thus, the predictor maps a $(d+1)$-dimensional vector $\mathbf{x} = (x_0 = 1, x_1, \ldots, x_d)$ into a zero or one. Note that $P(Y=1|\mathbf{x},\mathbf{w}^*) \geq 0.5$ only when $\mathbf{w}^\top\mathbf{x} \geq 0$. Expression $\mathbf{w}^\top\mathbf{x} = 0$ represents equation of a hyperplane that separates positive and negative examples. Thus, logistic regression model is a linear classifier.

It is also interesting to discuss the relationship between the posterior probability $P(Y=1|\mathbf{x},\mathbf{w})$ and distance of a data point $\mathbf{x}$ from the decision surface determined by $\mathbf{w}$. It can be shown that the distance from a point $\mathbf{x}$ and hyperplane $\mathbf{w}^\top\mathbf{x} = 0$ can be expressed as

$$\mathrm{d} = \frac{w_0 + \sum_{j=1}^{d} w_j x_j}{\sqrt{\sum_{j=1}^{d} w_j^2}}.$$

We usually refer to d as signed distance because its sign determines on which side of the hyperplane the data point is found. This distance can also be expressed in a vector form as

$$\mathrm{d} = \frac{w_0 + \mathbf{w}_-^\top \mathbf{x}_-}{\|\mathbf{w}_-\|}$$

where $\mathbf{x}_- = (x_1, x_2, \ldots, x_d)$ and $\mathbf{w}_- = (w_1, w_2, \ldots, w_d)$. Thus, it follows straightforwardly that

$$P(Y=1|\mathbf{x},\mathbf{w}) = \frac{1}{1+e^{-\mathrm{d}\|\mathbf{w}_-\|}}.$$

8.1.5 Maximizing likelihood vs. minimizing Euclidean distance

The previous sections showed alternative ways of training logistic regression classifiers by maximizing likelihood and by minimizing Euclidean distance. Here we show that under certain conditions these approaches are equivalent. Unfortunately, the assumptions will not strictly apply to logistic regression and will become more important later when we consider more powerful models (neural networks) that generalize logistic regression.

Consider a process of learning a classification model that is trained to minimize the Euclidean distance between the class labels y and soft predictions $s(\mathbf{x}, \mathbf{w})$. We shall assume binary classification on real-numbered inputs; i.e., $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{0, 1\}$, and express the expectation of the squared error $e^2(\mathbf{w})$ as

$$\mathbb{E}\left[e^2(\mathbf{w})\right] = \int_{\mathcal{X}} \sum_{\mathcal{Y}} (y - s(\mathbf{x}, \mathbf{w}))^2 \, p(\mathbf{x}, y) d\mathbf{x},$$

where $p(\mathbf{x}, y)$ is the true but unknown distribution of the data. Because $s(\mathbf{x}, \mathbf{w}) \in [0, 1]$ and $\mathcal{Y} = \{0, 1\}$ we have

$$y - s(\mathbf{x}, \mathbf{w}) = \begin{cases} -s(\mathbf{x}, \mathbf{w}) & y = 0 \\ \\ 1 - s(\mathbf{x}, \mathbf{w}) & y = 1 \end{cases}$$

Using $p(\mathbf{x}, y) = p(y|\mathbf{x})p(\mathbf{x})$, we can now modify the expected squared error as

$$\mathbb{E}\left[e^2(\mathbf{w})\right] = \int_{\mathcal{X}} \left(s^2(\mathbf{x}, \mathbf{w})p(0|\mathbf{x}) + (1 - s(\mathbf{x}, \mathbf{w}))^2 p(1|\mathbf{x})\right) p(\mathbf{x}) d\mathbf{x}.$$

Assuming that $s(\mathbf{x}, \mathbf{w})$ is powerful enough to be independently optimized for each unit volume $d\mathbf{x}$, we observe that minimizing $\mathbb{E}\left[e^2(\mathbf{w})\right]$ corresponds to finding a model that minimizes

$$s^2(\mathbf{x}, \mathbf{w})p(0|\mathbf{x}) + (1 - s(\mathbf{x}, \mathbf{w}))^2 p(1|\mathbf{x})$$

for each $d\mathbf{x}$. Minimizing the expression above results in

$$2s(\mathbf{x}, \mathbf{w}^*)p(0|\mathbf{x}) - 2(1 - s(\mathbf{x}, \mathbf{w}^*))p(1|\mathbf{x}) = 0$$

from where we infer that $s(\mathbf{x}, \mathbf{w}^*) = p(1|\mathbf{x})$; i.e., the classifier that minimizes Euclidean distance is learning the posterior probability distribution and makes an optimal decision for any given data point $\mathbf{x}$. The expected squared error for such a predictor can be expressed as

$$\mathbb{E}\left[e^2(\mathbf{w}^*)\right] = \int_{\mathcal{X}} p(1|\mathbf{x}) \left(1 - p(1|\mathbf{x})\right) p(\mathbf{x}) d\mathbf{x},$$

which is the expected variance of the class posteriors over the input space $\mathcal{X}$.

Unfortunately, there are caveats in this argument. First, the model $s(\mathbf{x}, \mathbf{w})$ has to have enough flexibility to be able to learn the posterior probability of class 1 in $d\mathbf{x}$ independently of the rest of the input space. In addition, the data must be abundant to allow for such training and the optimization step must be able to find a global minimum (recall that the sum-of-squares objective function is not convex in classification). Linear classifiers are not flexible enough to be considered models that learn class posterior probabilities; they exhibit strong dependencies in predictions between faraway inputs. However, we will later see that classification models that are theoretically capable of learning the posterior distribution are neural networks.

Let us now look into maximizing likelihood. Consider again a process of learning a classification model that is trained to maximize the likelihood, where y are class labels and $s(\mathbf{x}, \mathbf{w})$ are soft predictions. We shall assume binary classification on real-numbered inputs; i.e., $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{0, 1\}$, and express the expectation of the cross-entropy loss function as

$$\mathbb{E}\left[e(\mathbf{w})\right] = -\int_{\mathcal{X}} \sum_{\mathcal{Y}} \left(y \log s(\mathbf{x}, \mathbf{w}) + (1-y) \log(1 - s(\mathbf{x}, \mathbf{w}))\right) p(\mathbf{x}, y) d\mathbf{x},$$

where $p(\mathbf{x}, y)$ is the true but unknown distribution of the data. Because $s(\mathbf{x}, \mathbf{w}) \in [0, 1]$ and $\mathcal{Y} = \{0, 1\}$ we have that

$$\mathbb{E}\left[e(\mathbf{w})\right] = \int_{\mathcal{X}} \left(\log(1 - s(\mathbf{x}, \mathbf{w})) p(0|\mathbf{x}) + \log s(\mathbf{x}, \mathbf{w}) p(1|\mathbf{x})\right) p(\mathbf{x}) d\mathbf{x}.$$

Assuming that $s(\mathbf{x}, \mathbf{w})$ is powerful enough to be independently optimized for each unit volume $d\mathbf{x}$, we observe that minimizing $\mathbb{E}\left[e(\mathbf{w})\right]$ corresponds to finding a model that minimizes

$$\log(1 - s(\mathbf{x}, \mathbf{w})) p(0|\mathbf{x}) + \log s(\mathbf{x}, \mathbf{w}) p(1|\mathbf{x})$$

for each $d\mathbf{x}$. Minimizing the expression above results in

$$-\frac{1}{1 - s(\mathbf{x}, \mathbf{w}^*)} p(0|\mathbf{x}) + \frac{1}{s(\mathbf{x}, \mathbf{w}^*)} p(1|\mathbf{x}) = 0$$

from where we infer that $s(\mathbf{x}, \mathbf{w}^*) = p(1|\mathbf{x})$; i.e., the classifier that minimizes cross-entropy is learning the posterior probability distribution and makes an optimal decision for any given data point $\mathbf{x}$. The expected cross-entropy for such a predictor can be expressed as

$$\mathbb{E}\left[e(\mathbf{w}^*)\right] = -\int_{\mathcal{X}} \left(p(0|\mathbf{x}) \log p(0|\mathbf{x}) + p(1|\mathbf{x}) \log p(1|\mathbf{x})\right) p(\mathbf{x}) d\mathbf{x},$$

which is the expected differential entropy of the class posteriors over the input space $\mathcal{X}$.

8.2 Perceptron

As before, we will consider binary linear classifiers in $\mathbb{R}^d$, where for convenience we will take $\mathcal{X} = \{1\} \times \mathbb{R}^d$ and $\mathcal{Y} = \{-1, +1\}$. However, in contrast to the logistic regression approach where we used the logistic (sigmoid) function to model the posterior probability $p(y|\mathbf{x})$, here we are interested in a simpler task of directly finding a linear decision surface $\mathbf{w}^\top \mathbf{x} = 0$ (line, plane, hyperplane) that separates positive and negative examples available in the training set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$.

The perceptron is a simple machine or function $f : \mathcal{X} \to \mathcal{Y}$ defined as

$$f(\mathbf{x}) = \begin{cases} +1 & \mathbf{w}^\top \mathbf{x} \geq 0 \\ \\ -1 & \mathbf{w}^\top \mathbf{x} < 0 \end{cases}, \tag{8.12}$$

where, as before, $\mathbf{w} = (w_0, w_1, \ldots, w_d) \in \mathbb{R}^{d+1}$ is a vector of weights we seek to determine through training and $\mathbf{x} = (x_0 = 1, x_1, \ldots, x_d)$ is a data point from the input space $\mathcal{X}$.

Because $f(\mathbf{x})$ is non-differentiable, we cannot use differential calculus to optimize a predefined error or cost function. Instead, we will propose a weight update rule and then demonstrate that the training algorithm finds a good solution under certain assumptions.

First, we will assume that the data points from $\mathcal{D}$ are presented to the learning algorithm one at a time and the weights will be updated in the stochastic (incremental) mode. If the data set is infinitely large (say, we consider a data stream), the examples are presented to the learner, the weight update is made (if needed) and the example is deleted. On the other hand, if the data set is finite the learning algorithm can loop over the data set; i.e., once all n points have been presented to the learner, we start from the beginning. Second, and this is a strong assumption, we will consider that the positive and negative examples in $\mathcal{D}$ are linearly separable. Finally, for simplicity of downstream analysis, we will assume that the initial weight vector $\mathbf{w}^{(0)} = \mathbf{0}$.

For each data point $\mathbf{x}$ drawn from the data set at step t we have to consider two cases: (i) $\mathbf{x}$ is correctly classified and (ii) $\mathbf{x}$ is incorrectly classified using the decision surface determined by the current set of weights $\mathbf{w}^{(t)}$. If $\mathbf{x}$ is correctly classified, we do not need to update the current set of weights; i.e., $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)}$. On the other hand, if $\mathbf{x}$ is incorrectly classified, there are two possibilities: (i) if $\mathbf{x}$ was classified as negative (and its true class label is positive) we will update the weights using $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \mathbf{x}$; ($ii$) if $\mathbf{x}$ was classified as positive (and its true class label is negative), we will update the weight vector using $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \mathbf{x}$.

Why are those decisions good? Consider the first scenario of incorrect classification (underclassification); that is, when $\mathbf{w}^{(t)\top}\mathbf{x} < 0$. We will investigate the updated inner product between the weight vector and input $\mathbf{x}$ used in Eq. (8.12); that is

$$\begin{aligned}\mathbf{w}^{(t+1)\top}\mathbf{x} &= \mathbf{w}^{(t)\top}\mathbf{x} + \mathbf{x}^\top\mathbf{x} \\ &= \mathbf{w}^{(t)\top}\mathbf{x} + ||\mathbf{x}||^2 \\ &\geq \mathbf{w}^{(t)\top}\mathbf{x},\end{aligned}$$

and, therefore, the weight update rule is moving the inner product $\mathbf{w}^{(t)\top}\mathbf{x}$, and with it the prediction $f(\mathbf{x})$, in the right direction. A similar equation can be constructed for the second case of incorrect classification (overclassification). Our hypothesis is that if the algorithm always moves the decision boundary in the right direction, it will eventually find a hyperplane that correctly separates positive examples from negative examples.

The update rules for correct and incorrect classification (both underclassification and overclassification) can be combined as follows

$$\mathbf{w}^{(t+1)} = \begin{cases} \mathbf{w}^{(t)} & \mathbf{x} \text{ is correctly classified} \\ \\ \mathbf{w}^{(t)} + y\mathbf{x} & \mathbf{x} \text{ is incorrectly classified} \end{cases}, \tag{8.13}$$

where y is the class label of a data point $\mathbf{x}$. We refer to the weight update rule from Eq. (8.13) as the *perceptron training rule.* The entire perceptron training algorithm is presented in Algorithm 5.

We will now prove that, if the data points in $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ are linearly separable, the perceptron training algorithm converges and finds a separation hyperplane in a finite

Algorithm 5: Perceptron training algorithm. The algorithm loops over the training data $\mathcal{D}$ until either the weight vector is unchanged for a pre-specified number of steps or the maximum number of steps is exceeded.

```
Input:
    Training data: $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, $\mathcal{X} = \{1\} \times \mathbb{R}^d$ and $\mathcal{Y} = \{-1, +1\}$
    Learning parameter: $\eta \in (0, 1]$
    Termination criteria; e.g., the maximum number of steps

Initialization:
    $\mathbf{w} \leftarrow \mathbf{0}$

Weight learning:
    repeat until termination criteria are satisfied
        draw the next labeled example $(\mathbf{x}, y)$ from $\mathcal{D}$
        if  $(\mathbf{w}^\top \mathbf{x} \geq 0 \ \wedge \ y = -1) \ \vee \ (\mathbf{w}^\top \mathbf{x} < 0 \ \wedge \ y = +1)$
            $\mathbf{w} \leftarrow \mathbf{w} + \eta y \mathbf{x}$
        end
    end

Output:
    Weight vector $\mathbf{w} \in \mathbb{R}^{d+1}$
```

number of steps. Alternatively, if the data set is infinitely large, the number of incorrect classifications during training will be finite. To do this, we will turn the entire data set into a set of positive examples $\mathcal{D}^+$ by using $\mathbf{x}_i \leftarrow -\mathbf{x}_i$ and $y_i \leftarrow -y_i$, whenever the original class label is negative. This does not affect the weight update because the product $y_i \mathbf{x}_i$ is unchanged, but it does simplify the weight update rule to $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \mathbf{x}$ for the learning from $\mathcal{D}^+$.

The assumption that data points are linearly separable corresponds to the fact that there exists at least one solution vector, say $\mathbf{w}_0$, as well as a positive constant ε such that $\mathbf{w}_0^T \mathbf{x} > \varepsilon$ for $\forall \mathbf{x} \in \mathcal{D}^+$. Without loss of generality, we will also assume that $\mathbf{w}_0$ has unit length; i.e., $||\mathbf{w}_0|| = 1$. We will now look at the weight vector updated after the ℓ-th misclassified example, $\mathbf{w}^{(\ell)}$, and calculate the cosine of the angle between $\mathbf{w}_0$ and $\mathbf{w}^{(\ell)}$ as

$$\cos(\mathbf{w}_0, \mathbf{w}^{(\ell)}) = \frac{\mathbf{w}_0^\top \mathbf{w}^{(\ell)}}{||\mathbf{w}_0|| \cdot ||\mathbf{w}^{(\ell)}||}. \tag{8.14}$$

We will first investigate the growth of the numerator and denominator with the number of updates ℓ. Considering that $\mathbf{x}^{(\ell)}$ is the ℓ-th misclassified example, we first look at the numerator

$$\begin{aligned}
\mathbf{w}_0^\top \mathbf{w}^{(\ell)} &= \mathbf{w}_0^\top \left(\mathbf{w}^{(\ell-1)} + \mathbf{x}^{(\ell)} \right) \\
&= \mathbf{w}_0^\top \mathbf{w}^{(\ell-1)} + \mathbf{w}_0^\top \mathbf{x}^{(\ell)} \\
&> \mathbf{w}_0^\top \mathbf{w}^{(\ell-1)} + \varepsilon,
\end{aligned}$$

given that $\mathbf{w}_0^\top \mathbf{x} > \varepsilon$ for $\forall \mathbf{x} \in \mathcal{D}^+$. By repeating this approach recursively for $\mathbf{w}^{(\ell-1)}$, $\mathbf{w}^{(\ell-2)}$, ..., $\mathbf{w}^{(1)}$ and taking that $\mathbf{w}^{(0)} = \mathbf{0}$, we see that

$$\mathbf{w}_0^\top \mathbf{w}^{(\ell)} > \ell\varepsilon. \tag{8.15}$$

Next, we will look at the norm of $\mathbf{w}^{(\ell)}$

$$\begin{aligned} ||\mathbf{w}^{(\ell)}||^2 &= \left(\mathbf{w}^{(\ell-1)} + \mathbf{x}^{(\ell)}\right)^\top \left(\mathbf{w}^{(\ell-1)} + \mathbf{x}^{(\ell)}\right) \\ &= ||\mathbf{w}^{(\ell-1)}||^2 + 2\mathbf{w}^{(\ell-1)\top}\mathbf{x}^{(\ell)} + ||\mathbf{x}^{(\ell)}||^2 \\ &< ||\mathbf{w}^{(\ell-1)}||^2 + ||\mathbf{x}^{(\ell)}||^2, \end{aligned}$$

where we used that $\mathbf{w}^{(\ell-1)\top}\mathbf{x}^{(\ell)} < 0$ because the (positive) data point was misclassified. From here, we see that

$$\begin{aligned} ||\mathbf{w}^{(\ell)}||^2 &< \sum_{l=1}^{\ell} ||\mathbf{x}^{(\ell)}||^2 \\ &\leq \ell M^2, \end{aligned} \tag{8.16}$$

where

$$M = \max_{\mathbf{x}_i \in \mathcal{D}^+} ||\mathbf{x}_i||.$$

We introduced the constant M to indicate that the norm of the input vectors is bounded. It is now important to make the following observations.

1. Eq. (8.15) shows that the numerator in Eq. (8.14) grows at least linearly with ℓ
2. Eq. (8.16) shows that the denominator in Eq. (8.14) grows at most linearly with $\sqrt{\ell}$.

However, the two conditions become incompatible as $\ell \to \infty$ because the cosine function cannot be larger than 1. Therefore, we conclude that there must exist some number of updates, say $\ell_{\max}$, for which

$$\ell_{\max}\varepsilon < \mathbf{w}^\top \mathbf{w}^{(\ell_{\max})} \leq ||\mathbf{w}^{(\ell_{\max})}|| < \sqrt{\ell_{\max} M^2}.$$

This, in turn, leads to the upper limit on the number of updates

$$\ell_{\max} < \frac{M^2}{\varepsilon^2}.$$

Under the aforementioned assumptions, the linear decision boundary (i.e., the underlying concept) will be learned after a finite number of $\ell_{\max}$ updates, which guarantees the convergence of the perceptron training algorithm. Therefore, the perceptron training algorithm will find a separating hyperplane.

In practice, it is useful to modify the update rule to

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \eta y \mathbf{x},$$

where $\eta \in (0, 1]$ in order to avoid rapid movements of the separating hyperplane during training. The convergence of the perceptron training algorithm can be readily proved when $\eta \in (0, 1]$.

Problems with the perceptron training algorithm

While interesting in the context of learning theory, the perceptron training algorithm has a strong assumption on linear separability of the data, that is necessary for the algorithm to converge. If the data are not linearly separable, the algorithm never stops and the cycles in weight updates can be hard to detect. Therefore, a different learning algorithm is needed to correct for this deficiency. We discuss this next.

8.2.1 The Pocket algorithm

To overcome the problem with linear separability of the data, the perceptron training algorithm needs to be modified. One of the best approaches to that problem is the Pocket algorithm. The idea is simple: execute the perceptron training algorithm, but count the number of consecutive examples that are classified correctly after each weight update. Whenever the longest count is achieved (at the next necessary weight update), keep the best performing set of weights "in the pocket". Over time, the set of weights with the smallest misclassification error will remain in the pocket with high probability. In fact, it can be shown that in the limit the Pocket algorithm indeed minimizes the misclassification rate. The training procedure is presented in Algorithm 6.

8.2.2 Representational power of the perceptron

The perceptron is a linear classifier in that it learns a decision boundary $\mathbf{w}^\top \mathbf{x} = 0$ which separates the input space $\mathcal{X} = \mathbb{R}^d$ into two half-spaces. While this is relatively straightforward, let us also take a look at a special case when the input examples lie in the corners of a d-dimensional hypercube; i.e., $\mathcal{X} = \{-1, +1\}^d$. This case is important in computational learning theory and will also help us to understand the inductive bias of the perceptron compared to that of, say, decision trees.

We can immediately recognize that the perceptron cannot either represent or learn all binary functions. Using $d = 2$, we can see that regardless of the data set for training, the perceptron cannot represent an exclusive-or (XOR) function

$$f_{\mathrm{XOR}}(\boldsymbol{x}) = \begin{cases} +1 & \text{for } x_1 \neq x_2 \\ -1 & \text{for } x_1 = x_2 \end{cases}$$

The XOR function can be seen in d-dimensional space as a parity function that outputs $+1$ if the number of 1's is odd and -1 if the number of ones in $\boldsymbol{x}$ is even. However, perceptron can learn an important class of functions that detect when at least m-out-of-d input features are $+1$. Looking at the linear combination of weights

$$w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_d x_d \geq 0$$

we can see that setting weights $w_1 = w_2 = \ldots = w_d = 1$ and $w_0 = -m$ (when $\mathcal{X} = \{0, 1\}^d$) and $w_0 = d - 2m$ (when $\mathcal{X} = \{-1, +1\}^d$) will achieve the desired effect of $w_0 + \sum_{j=1}^{d} w_j x_j \geq 0$.

8.2.3 Kernelizing perceptrons

The weight vector found by the perceptron training algorithm is obtained by adding or subtracting data points from $\mathcal{D}$. This leads us to the following expression for the weight

Algorithm 6: Pocket algorithm. The algorithm loops over the training data $\mathcal{D}$ until either $\mathbf{w}_{\text{pocket}}$ is unchanged for a pre-specified number of steps or the maximum number of steps is exceeded.

```
Input:
    Training data: $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$, $\mathcal{X} = \{1\} \times \mathbb{R}^d$ and $\mathcal{Y} = \{-1, +1\}$
    Learning parameter: $\eta \in (0, 1]$
    Termination criteria; e.g., the maximum number of steps

Initialization:
    $\mathbf{w} \leftarrow \mathbf{0}$
    $\mathbf{w}_{\text{pocket}} \leftarrow \mathbf{0}$
    $\text{run} \leftarrow 0$
    $\text{run}_{\text{pocket}} \leftarrow 0$

Weight learning:
    repeat until termination criteria are satisfied
        draw the next labeled example $(\mathbf{x}, y)$ from $\mathcal{D}$
        if  $(\mathbf{w}^\top \mathbf{x} \geq 0 \wedge y = -1) \vee (\mathbf{w}^\top \mathbf{x} < 0 \wedge y = +1)$
            if  $\text{run} > \text{run}_{\text{pocket}}$
                $\mathbf{w}_{\text{pocket}} \leftarrow \mathbf{w}$
                $\text{run}_{\text{pocket}} \leftarrow \text{run}$
            end
            $\mathbf{w} \leftarrow \mathbf{w} + \eta y \mathbf{x}$
            $\text{run} \leftarrow 0$
        else
            $\text{run} \leftarrow \text{run} + 1$
        end
    end

Output:
    Weight vector $\mathbf{w}_{\text{pocket}} \in \mathbb{R}^{d+1}$
```

vector $\mathbf{w}$

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i,$$

where α_i is the number of times example $\mathbf{x}_i$ was misclassified during training. Given that the classification of the data point $\mathbf{x}$ is made based on the linear combination with the weight vector $\mathbf{w}$, this leads us to the following expression

$$\begin{aligned} \mathbf{w}^\top \mathbf{x} &= \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^\top \mathbf{x} \\ &= \sum_{i=1}^{n} \alpha_i y_i k(\mathbf{x}_i, \mathbf{x}), \end{aligned} \tag{8.17}$$

where $k(\mathbf{x}_i, \mathbf{x}_j)$ is an inner product of data points i and j. This formulation, however, gives us an opportunity to exploit a richer set of similarity functions $k(\cdot, \cdot)$ that will allow us to find non-linear decision surfaces as long as $k(\cdot, \cdot)$ can be seen as an inner product of data points in a transformed vector space; e.g., $k(\mathbf{x}_i, \mathbf{x}_j) = \boldsymbol{\phi}(\mathbf{x}_i)^\top \boldsymbol{\phi}(\mathbf{x}_j)$. This will also allow classification on complex objects such as sequences or graphs, where $\boldsymbol{\phi}$ can be interpreted as an embedding function.

The price to pay for kernelizing the perceptron is that a set of misclassified examples must be maintained together with the number of times they were misclassified. In addition, the speed of training and classification is impacted because Eq. (8.17) must be recomputed for each prediction.

8.3 Naive Bayes classifiers as linear models

Naive Bayes binary classifier is generally referred to as a soft non-linear classifier because it can learn non-linear concepts but is not a universal approximator. In some cases, however, this classifier is guaranteed to find linear decision boundaries. We here investigate such situations by first considering a simple case of binary features ($\mathcal{X} = \{0, 1\}^d$) and then broadening it to the class of distributions from the exponential family ($\mathcal{X} = \mathbb{R}^d$). We only consider binary classification; that is, $\mathcal{Y} = \{0, 1\}$.

8.3.1 Linear classification boundary for binary features

Naive Bayes classifier with binary features and two classes is a linear classifier. This is somewhat surprising, as this generative approach looks very different from what we did before. To see why this is the case, notice that the classifier will make a positive decision when

$$P(Y = 1|\boldsymbol{x}) \geq P(Y = 0|\boldsymbol{x})$$

that is, when

$$p(\boldsymbol{x}|Y = 1)p(Y = 1) \geq p(\boldsymbol{x}|Y = 0)p(Y = 0)$$

We will shorten this notation using $p(Y=0)=p(0)$, $p(\boldsymbol{x}|Y=0)=p(\boldsymbol{x}|0)$, etc. Using the naive Bayes assumption from Eq. (4.1), we now have

$$p(1)\prod_{j=1}^{d} p(x_j|1) \geq p(0)\prod_{j=1}^{d} p(x_j|0),$$

which, after applying a logarithm, becomes

$$\log p(1) + \sum_{j=1}^{d} \log p(x_j|1) \geq \log p(0) + \sum_{j=1}^{d} \log p(x_j|0).$$

Let us now investigate class-conditional probabilities $p(x_j|y)$ when $y \in \{0,1\}$. Recall that each feature is Bernoulli distributed; i.e.,

$$p(x_j|1) = p_{j,1}^{x_j}(1-p_{j,1})^{1-x_j}$$

and

$$p(x_j|0) = p_{j,0}^{x_j}(1-p_{j,0})^{1-x_j},$$

where parameters $p_{j,c}$ are estimated from the training set. Taking $p(Y=c)=p_c$, we have

$$\sum_{j=1}^{d} x_j \log \frac{p_{j,1}(1-p_{j,0})}{(1-p_{j,1})p_{j,0}} + \sum_{j=1}^{d} \log \frac{1-p_{j,1}}{1-p_{j,0}} + \log \frac{p_1}{p_0} \geq 0.$$

We can write the previous expression as

$$w_0 + \sum_{j=1}^{d} w_j x_j \geq 0,$$

where

$$\begin{aligned} w_0 &= \log \frac{p_1}{p_0} + \sum_{j=1}^{d} \log \frac{1-p_{j,1}}{1-p_{j,0}} \\ w_j &= \log \frac{p_{j,1}(1-p_{j,0})}{(1-p_{j,1})p_{j,0}} \qquad j \in \{1,2,\ldots,d\}. \end{aligned}$$

Therefore, in the case of binary features, naive Bayes is a linear classifier.

8.3.2 Linear classification boundary for continuous features

Let us now consider binary classification on a d-dimensional feature space of real numbers; i.e., $\mathcal{X}=\mathbb{R}^d$ and $\mathcal{Y}=\{0,1\}$. Suppose that the class-conditional distribution for every feature j comes from an exponential family and that the same distribution from the exponential family is used to model each class-conditional distribution for that feature. Recall that the exponential family distributions have the following canonical form

$$p(x_j|\boldsymbol{\theta}_{jc}) = \exp\left(\boldsymbol{\theta}_{jc}^T \boldsymbol{t}(x_j) - a(\boldsymbol{\theta}_{jc}) + b(x_j)\right),$$

where $\boldsymbol{\theta}_{jc}$ indicates different parameters for each feature j and class value $c \in \{0, 1\}$. Let us now express the posterior probability $P(Y = 1|\boldsymbol{x})$ as

$$\begin{aligned} p(1|\boldsymbol{x}) &= \frac{p(\boldsymbol{x}|1)p(1)}{p(\boldsymbol{x}|0)p(0) + p(\boldsymbol{x}|1)p(1)} \\ &= \frac{1}{1 + \frac{p(\boldsymbol{x}|0)p(0)}{p(\boldsymbol{x}|1)p(1)}} \\ &= \frac{1}{1 + \exp\left(-\ln \frac{p(\boldsymbol{x}|1)p(1)}{p(\boldsymbol{x}|0)p(0)}\right)} \\ & \frac{1}{1 + \exp\left(-\gamma(\boldsymbol{x})\right)} \end{aligned}$$

We can now separately look at the function $\gamma(\boldsymbol{x})$ and under what conditions it can be expressed as a linear combination of feature values. By applying the naive Bayes assumption from Eq. (4.1) we get

$$\begin{aligned} \gamma(\boldsymbol{x}) &= \ln \frac{p(1)}{p(0)} + \sum_{j=1}^{d} \Big(\boldsymbol{\theta}_{j1}^T \boldsymbol{t}(x_j) - a(\boldsymbol{\theta}_{j1}) + b(x_j) - \boldsymbol{\theta}_{j0}^T \boldsymbol{t}(x_j) - a(\boldsymbol{\theta}_{j0}) + b(x_j)\Big) \\ &= \gamma_0(\boldsymbol{x}) + \sum_{j=1}^{d} \gamma_j(\boldsymbol{x}) \end{aligned}$$

where $\gamma_0(\boldsymbol{x})$ does not depend on $\boldsymbol{x}$. To proceed from here, we will look at two different distributions for x_j and then infer broader conclusions. We easily see that we can consider every component $\gamma_j(\boldsymbol{x})$ separately. If we can show that $\gamma_j(\boldsymbol{x}) = w_{j0} + w_j x_j$ we can conclude that the entire expression is a linear combination of feature values.

Let us first assume a Poisson distributed class-conditional distributions for a feature j. Using exponential family notation, we can re-write the Poisson distribution as

$$p(x|\lambda) = \exp(x \log \lambda - \lambda - \log x!),$$

with $\boldsymbol{\theta} = \log \lambda$, $\boldsymbol{t}(x) = x$, $a(\boldsymbol{\theta}) = \lambda$, and $b(x) = -\log x!$, and observe that

$$\begin{aligned} \gamma_j(\boldsymbol{x}) &= x_j \ln \lambda_{j1} - \lambda_{j1} - \ln x_j! - x_j \ln \lambda_{j0} + \lambda_{j0} + \ln x_j! \\ &= w_{j0} + w_j x_j \end{aligned}$$

where $w_{j0} = \lambda_{j0} - \lambda_{j1}$ and $w_j = \ln \lambda_{j1}/\lambda_{j0}$. If we now consider a mixture of Poisson distributions for every feature, we get

$$\gamma(\boldsymbol{x}) = w_0 + \sum_{j=1}^{d} w_j x_j,$$

where

$$w_0 = \ln \frac{p(1)}{p(0)} + \sum_{j=1}^{d} (\lambda_{j0} - \lambda_{j1})$$

and

$$w_j = \ln \lambda_{j1} - \ln \lambda_{j0}.$$

Therefore, the Poisson distributed features lead to a linear classification model as long as both class-conditional distributions for each feature are separate Poisson distributions.

The case of Gaussian features is a bit more complex. Using the exponential family re-write for a Gaussian distribution

$$p(x|\mu, \sigma) = \exp\left(-\frac{x^2}{2\sigma^2} + x\frac{\mu}{\sigma^2} - \frac{\mu^2}{2\sigma^2} - \frac{1}{2}\ln(2\pi\sigma^2)\right)$$

we obtain that

$$\begin{aligned}\gamma_j(\boldsymbol{x}) &= -\frac{x_j^2}{2\sigma_{j1}^2} + x_j\frac{\mu_{j1}^2}{\sigma_{j1}^2} - \frac{\mu_{j1}^2}{\sigma_{j1}^2} - \frac{\ln(2\pi\sigma_{j1}^2)}{2} + \frac{x_j^2}{2\sigma_{j0}^2} - x_j\frac{\mu_{j0}^2}{\sigma_{j0}^2} + \frac{\mu_{j0}^2}{\sigma_{j0}^2} + \frac{\ln(2\pi\sigma_{j0}^2)}{2} \\ &\neq w_{j0} + w_j x_j\end{aligned}$$

because of the squared x_j's. However, we can see that the quadratic terms cancel each other out if $\sigma_{j1} = \sigma_{j0}$. Therefore, the case of Gaussian features will also result in a linear combination if the class-conditional Gaussians for the same feature are modeled with equal variance.

This analysis suggests that there exists a broad range of conditions that result in linear classification models when naive Bayes assumptions are applied. These include that the distribution of each feature is considered to be a two-component mixture of the distributions from the same group within exponential family, sometimes with additional constraints as with the Gaussians above. Notice that the Bernoulli distribution from the previous section is itself a member of exponential family. Nevertheless, considering its case separately shows different ways in which our conclusions can be reached.

8.4 Multinomial logistic regression

Now let us consider discriminative multiclass classification, where $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{1, 2, \ldots, k\}$. This setting arises naturally in machine learning, where there is often more than two categories. For example, if we want to predict the blood type (A, B, AB and O) of an individual, then we have four classes. Here we discuss multiclass classification where we only want to label a datapoint with one class out of k. In other settings, one might want to label a datapoint with multiple classes; this is briefly mentioned at the end of this section.

We can nicely generalize to this setting using the idea of multinomials and the corresponding link function, as with the other generalized linear models. The multinomial distribution is a member of the exponential family. We can write

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{y_1! \ldots y_k!} p(y_1 = 1|\mathbf{x})^{y_1} \ldots, p(y_k = 1|\mathbf{x})^{y_k} \tag{8.18}$$

where the usual numerator $n! = 1$ because $n = \sum_{j=1}^{k} y_j = 1$ since we can only have one class value. As with logistic regression, we can parametrize

$$p(y_j = 1|\mathbf{x}) = \sigma(\mathbf{x}^\top \mathbf{w}_j) = (1 + \exp(-\mathbf{x}^\top \mathbf{w}))^{-1}.$$

However, we must also ensure that $\sum_{j=1}^{k} p(y_j = 1|\mathbf{x}) = 1$. To do so, we "pivot" around the final class, $p(y_k = 1|\mathbf{x}) = 1 - \sum_{j=1}^{k-1} p(y_j = 1|\mathbf{x})$ and only explicitly learn $\mathbf{w}_1, \ldots, \mathbf{w}_{k-1}$. Note that these models are not learned independently, because they are tied by the probability for the last class. The parameters can be represented as a matrix $\mathbf{W} \in \mathbb{R}^{d \times k}$ where $\mathbf{W} = [\mathbf{w}_1, \dots, \mathbf{w}_k]$ is composed of k weight vectors with $\mathbf{w}_k = \mathbf{0}$. We will see why we fix $\mathbf{w}_k = \mathbf{0}$.

The transfer (inverse of the link) for this setting is the softmax transfer

$$\begin{aligned}\operatorname{softmax}(\mathbf{x}^\top \mathbf{W}) &= \left[\frac{\exp(\mathbf{x}^\top \mathbf{w}_1)}{\sum_{j=1}^k \exp(\mathbf{x}^\top \mathbf{w}_j)}, \dots, \frac{\exp(\mathbf{x}^\top \mathbf{w}_k)}{\sum_{j=1}^k \exp(\mathbf{x}^\top \mathbf{w}_j)}\right] \\ &= \left[\frac{\exp(\mathbf{x}^\top \mathbf{w}_1)}{\mathbf{1}^\top \exp(\mathbf{x}^\top \mathbf{W})}, \dots, \frac{\exp(\mathbf{x}^\top \mathbf{w}_k)}{\mathbf{1}^\top \exp(\mathbf{x}^\top \mathbf{W})}\right]\end{aligned}$$

and the prediction is $\operatorname{softmax}(\mathbf{x}) = \hat{\mathbf{y}} \in [0,1]^k$, which gives the probability in each entry of being labeled as that class, where $\hat{\mathbf{y}}^\top \mathbf{1} = 1$ signifying that the probabilities sum to 1. Note that this model encompasses the binary setting for logistic regression, because $\sigma(\mathbf{x}^\top \mathbf{w}) = (1 + \exp(-\mathbf{x}^\top \mathbf{w}))^{-1} = \frac{\exp(\mathbf{x}^\top \mathbf{w})}{1+\exp(\mathbf{x}^\top \mathbf{w})}$. The weights for multinomial logistic regression with two classes are then $\mathbf{W} = [\mathbf{w},\ \mathbf{0}]$ giving

$$\begin{aligned}p(y = 0|\mathbf{x}) &= \frac{\exp(\mathbf{x}^\top \mathbf{w})}{\mathbf{1}^\top \exp(\mathbf{x}^\top \mathbf{W})} \\ &= \frac{\exp(\mathbf{x}^\top \mathbf{w})}{\exp(\mathbf{x}^\top \mathbf{w}) + \exp(\mathbf{x}^\top \mathbf{0})} \\ &= \frac{\exp(\mathbf{x}^\top \mathbf{w})}{\exp(\mathbf{x}^\top \mathbf{w}) + 1} \\ &= \sigma(\mathbf{x}^\top \mathbf{w}).\end{aligned}$$

Similarly, for $k > 2$, by fixing $\mathbf{w}_k = \mathbf{0}$, the other weights $\mathbf{w}_1, \dots, \mathbf{w}_{k-1}$ are learned to ensure that $p(y = k|\mathbf{x}) = \frac{\exp(\mathbf{x}^\top \mathbf{w}_k)}{\mathbf{1}^\top \exp(\mathbf{x}^\top \mathbf{W})} = \frac{1}{1+\sum_{j=1}^{k-1} \exp(\mathbf{x}^\top \mathbf{w}_j)}$ and that $\sum_{j=1}^k p(y = j|\mathbf{x}) = 1$.

With the parameters of the model parameterized by $\mathbf{W}$ and the softmax transfer, we can determine the maximum likelihood formulation. By plugging in the parameterization into Equation (8.18), taking the negative log of that likelihood and dropping constants, we arrive at the following loss for samples $(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_n, \mathbf{y}_n)$

$$\min_{\mathbf{W} \in \mathbb{R}^{d \times k} : \mathbf{W}_{:k} = \mathbf{0}} \sum_{i=1}^n \log\left(\mathbf{1}^\top \exp(\mathbf{x}_i^\top \mathbf{W})\right) - \mathbf{x}_i^\top \mathbf{W} \mathbf{y}_i$$

with gradient

$$\nabla \sum_{i=1}^n \left(\log\left(\mathbf{1}^\top \exp(\mathbf{x}_i^\top \mathbf{W})\right) - \mathbf{x}_i^\top \mathbf{W} \mathbf{y}_i\right) = \sum_{i=1}^n \frac{\exp(\mathbf{x}_i^\top \mathbf{W})^\top \mathbf{x}_i^\top}{\mathbf{1}^\top \exp(\mathbf{x}_i^\top \mathbf{W})} - \mathbf{x}_i \mathbf{y}_i^\top.$$

As before, we do not have a closed form solution for this gradient, and will use iterative methods to solve for $\mathbf{W}$. Note that here, unlike previous methods, we have a constraint on part of the variable. However, this was solely written this way for convenience. We do not optimize $\mathbf{W}_{:k}$, as it is fixed at zero; one can rewrite this minimization and gradient to only apply to the $W_{:(1:k-1)}$. This corresponds to initializing $\mathbf{W}_{:k} = \mathbf{0}$, and then only using the first $k-1$ columns of the gradient in the update to $W_{:(1:k-1)}$.

The final prediction $\text{softmax}(\mathbf{x}^\top \mathbf{W}) \in [0,1]$ gives the probabilities of being in a class. As with logistic regression, to pick one class, the highest probability value is chosen. For example, with $k = 4$, we might predict $[0.1 \;\; 0.2 \;\; 0.6 \;\; 0.1]$ and so decide to classify the point into class 3.

Remark about overlapping classes: If you want to predict multiple classes for a datapoint $\mathbf{x}$, then a common strategy is to learn separate binary predictors for each class. Each predictor is queried separately, and a datapoint will label each class as 0 or 1, with potentially more than one class having a 1. Above, we examined the case where the datapoint was exclusively in one of the provided classes, by setting $n = 1$ in the multinomial.