Advanced Machine Learning – Lecture 4: Classification

Charalampos Giannakis

Linear Models for Classification

The goal in classification is to take an input vector $\mathbf{x}$ and assign it to one of K discrete classes. We will focus on **linear models** for classification, meaning the decision boundaries between classes will be linear functions of the input (i.e. defined by hyperplanes in the input space). In an input space of dimensionality D, a hyperplane has dimension $D-1$. Thus, a linear classifier separates classes by $(D-1)$-dimensional hyperplanes in the D-dimensional input space.

A basic starting point is to use a linear function similar to linear regression:

$$y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0 \ ,$$

where $\mathbf{w}$ is a weight vector and w_0 is a bias (intercept). To turn this linear combination into a discrete class prediction, we apply a nonlinear **activation function** $f(\cdot)$ to $y(\mathbf{x})$. For example, in a two-class setting we might take f to be a step function (also called the Heaviside or sign function), which outputs one class if $y(\mathbf{x}) \geq 0$ and the other class if $y(\mathbf{x}) < 0$. With such an activation, the model

$$y(\mathbf{x}) = f(\mathbf{w}^T\mathbf{x} + w_0)$$

produces discrete class labels. However, note that due to the non-linear activation $f(\cdot)$, the overall classifier is **not linear in its parameters** (even though $y(\mathbf{x})$ is a linear function of $\mathbf{w}$). This means training (fitting parameters) will generally require non-linear optimization.

Two-Class Linear Discriminant Functions

Simplest case – two classes: Consider first a binary classification problem (two classes). We use a linear discriminant function $y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$, with $\mathbf{w}$ as the weight vector and w_0 as the bias term. The decision rule is to predict class 1 if $y(\mathbf{x}) \geq 0$, and class 2 if $y(\mathbf{x}) < 0$ (assuming a step activation as above). The **decision boundary** is thus the set of points where $y(\mathbf{x}) = 0$, i.e. all $\mathbf{x}$ satisfying

$$\mathbf{w}^T\mathbf{x} + w_0 = 0 \ .$$

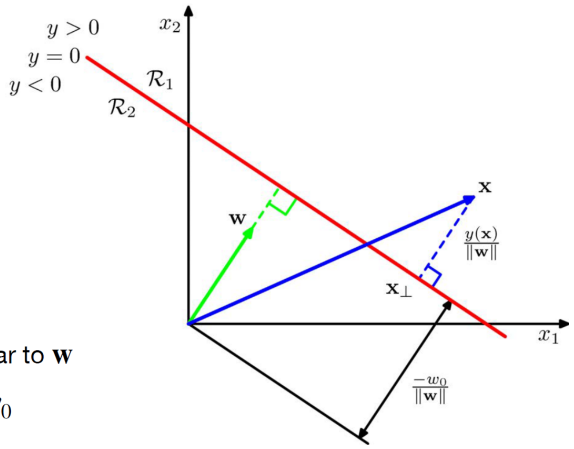
This equation defines a hyperplane in the input space. Any input on one side of the hyperplane will yield $y(\mathbf{x}) > 0$ (and be classified as class 1), while any input on the other side yields $y(\mathbf{x}) < 0$ (class 2).

Now, consider two points $\mathbf{x}_a$ and $\mathbf{x}_b$ that both lie on the decision surface (boundary), so $y(\mathbf{x}_a) = 0$ and $y(\mathbf{x}_b) = 0$. Subtracting the equations $\mathbf{w}^T\mathbf{x}_a + w_0 = 0$ and $\mathbf{w}^T\mathbf{x}_b + w_0 = 0$, we obtain:

$$\mathbf{w}^T(\mathbf{x}_a - \mathbf{x}_b) = 0 \ .$$

This implies that $\mathbf{w}$ is **orthogonal** (perpendicular) to the difference $(\mathbf{x}_a - \mathbf{x}_b)$ of any two points on the decision boundary. Geometrically, this means $\mathbf{w}$ is a normal vector to the decision hyperplane. In other words, the weight vector $\mathbf{w}$ points in a direction perpendicular to the decision surface.

If a point $\mathbf{x}$ happens to lie on the decision boundary, then $\mathbf{w}^T\mathbf{x} + w_0 = 0$. We can solve this for w_0 to express the bias in terms of a particular boundary point: $w_0 = -\mathbf{w}^T\mathbf{x}$. More generally, the distance from the origin to the hyperplane can be derived by normalizing this equation. For any point on the hyperplane, the quantity $\frac{\mathbf{w}^T\mathbf{x}}{\|\mathbf{w}\|}$ equals $-\frac{w_0}{\|\mathbf{w}\|}$. This value is the (signed) perpendicular distance from the origin to the hyperplane. Thus, the magnitude of w_0 relative to $\|\mathbf{w}\|$ controls the displacement of the decision boundary from the origin along the direction of $\mathbf{w}$.



- Decision surface is perpendicular to $\mathbf{w}$
- Displacement is controlled by w_0

Figure illustrates a two-class linearly separable case.

Here $\mathbf{w}$ is orthogonal to the decision line, and shifting w_0 moves the line parallel to itself.

Geometry of Linear Discriminants

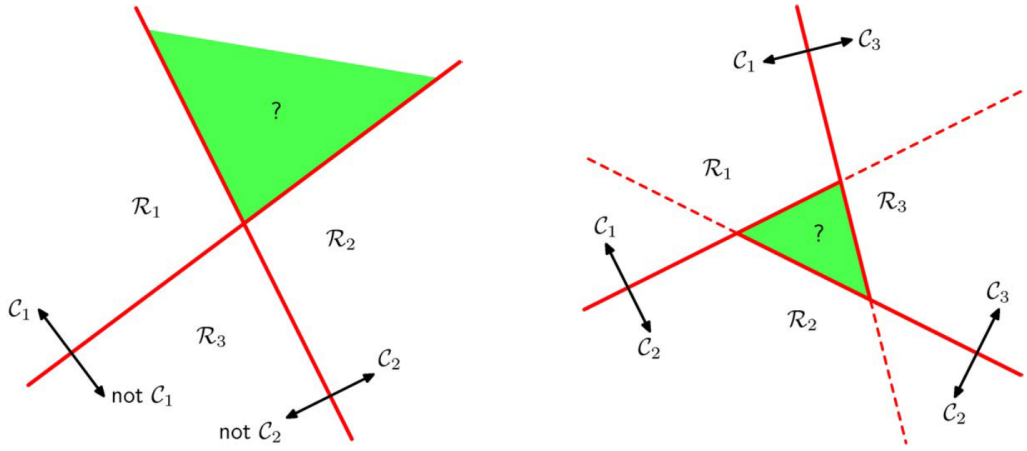
From the above analysis, we note two key geometric facts about a linear discriminant $y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$:

- The decision surface (the hyperplane defined by $\mathbf{w}^T\mathbf{x} + w_0 = 0$) is **perpendicular to** the weight vector $\mathbf{w}$.
- The **offset** (position) of this hyperplane is controlled by the bias w_0. Increasing or decreasing w_0 shifts the hyperplane along the direction of $\mathbf{w}$ without changing its orientation.

Figure above demonstrates this geometry: the vector $\mathbf{w}$ is normal to the hyperplane, and the intercept $-\frac{w_0}{\|\mathbf{w}\|}$ determines how far the hyperplane is from the origin.

Multiple Classes and K-Class Discriminants

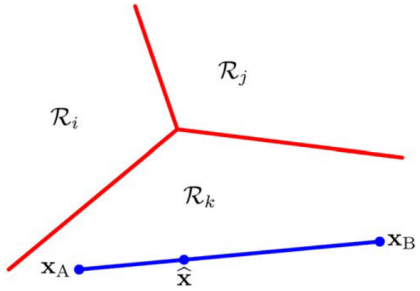
The binary linear classifier can be generalized to K classes (with $K > 2$). A naïve approach would be to combine multiple two-class classifiers (for example, one vs. rest or one vs. one schemes). However, using multiple binary classifiers to handle K classes is **not generally a good idea** because it can lead to ambiguous regions where the classifiers' decisions conflict.



A better approach is to construct a single classifier that directly handles K classes by using **multiple discriminant functions**. Specifically, we can introduce K linear functions (one per class):

$$y_k(\mathbf{x}) = \mathbf{w}_k^T\mathbf{x} + w_{k0}, \qquad k = 1, 2, \ldots, K \ ,$$

where each class C_k has its own weight vector $\mathbf{w}_k$ and bias w_{k0}. Given an input $\mathbf{x}$, the classifier computes all K discriminant scores $\{y_1(\mathbf{x}), \ldots, y_K(\mathbf{x})\}$ and then assigns $\mathbf{x}$ to the class with the largest score. In other words, $\mathbf{x}$ is predicted to belong to class C_k if $y_k(\mathbf{x}) > y_j(\mathbf{x})$ for all $j \neq k$.



What do the decision boundaries look like in this case? The boundary between class C_k and class C_j is the set of points where the two discriminants are equal: $y_k(\mathbf{x}) = y_j(\mathbf{x})$. Setting $\mathbf{w}_k^T\mathbf{x} + w_{k0} = \mathbf{w}_j^T\mathbf{x} + w_{j0}$ and rearranging yields:

$$(\mathbf{w}_k - \mathbf{w}_j)^T\mathbf{x} + (w_{k0} - w_{j0}) = 0 \ ,$$

which indeed defines a hyperplane (linear boundary) between classes C_k and C_j. This hyperplane's normal vector is $(\mathbf{w}_k - \mathbf{w}_j)$.

An attractive property of using K linear discriminant functions is that each class's **decision region** (the set of $\mathbf{x}$ classified as C_k) will be a single connected convex region in input space. This is because all decision boundaries are linear. Convex, connected decision regions make the classifier's behavior easier to interpret and analyze.

(If one tried to patch together multiple 2-class classifiers, the decision regions could become fragmented or inconsistent. The single K-class discriminant approach avoids that issue.)

The Perceptron Algorithm

One of the earliest learning algorithms for a linear classifier is the **Perceptron**, introduced by Frank Rosenblatt in 1962. The perceptron uses a linear model combined with a hard threshold activation. In its basic form, the perceptron can be described as:

$$y(\mathbf{x}) = f(\mathbf{w}^T\phi(\mathbf{x})),$$

where $\phi(\mathbf{x})$ is a fixed feature mapping of the input (in the simplest case $\phi(\mathbf{x}) = \mathbf{x}$ with an added constant 1 for the bias), and $f(a)$ is a step function:

$$f(a) = \begin{cases} +1, & a \geq 0, \\ -1, & a < 0 \ , \end{cases}$$

assuming we encode the two classes as $+1$ and -1 (an alternative common encoding is 0/1, but here we'll use $\{-1, +1\}$ for convenience in algebra). In this formulation, $y(\mathbf{x})$ will output either +1 or -1 as the predicted class label.

Training the perceptron: Let our training set be $\{(\mathbf{x}_n, t_n)\}_{n=1}^N$ where $t_n \in \{-1, +1\}$ is the target class of example n. The perceptron learning rule is based on a criterion that tries to minimize the number of misclassified examples. Define the set of misclassified examples (with respect to the current weight vector $\mathbf{w}$) as

$$\mathcal{M} = \{n \mid t_n \, \mathbf{w}^T \phi(\mathbf{x}_n) < 0\} \ ,$$

i.e. those indices for which the perceptron's prediction is wrong (since t_n and $\mathbf{w}^T\phi(x_n)$ have opposite signs). The perceptron criterion can be written as an error function that sums the negative margins of misclassified points:

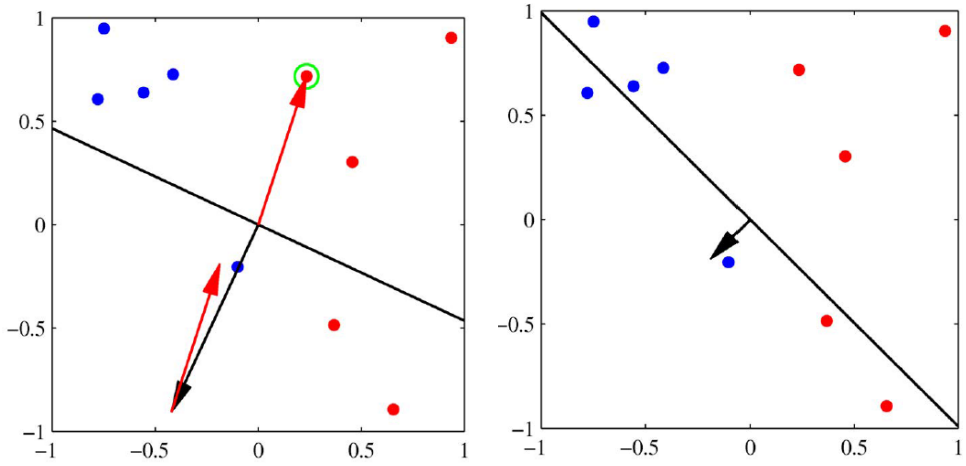
$$E_P(\mathbf{w}) = -\sum_{n \in \mathcal{M}} t_n \, \mathbf{w}^T \phi(\mathbf{x}_n) \ .$$

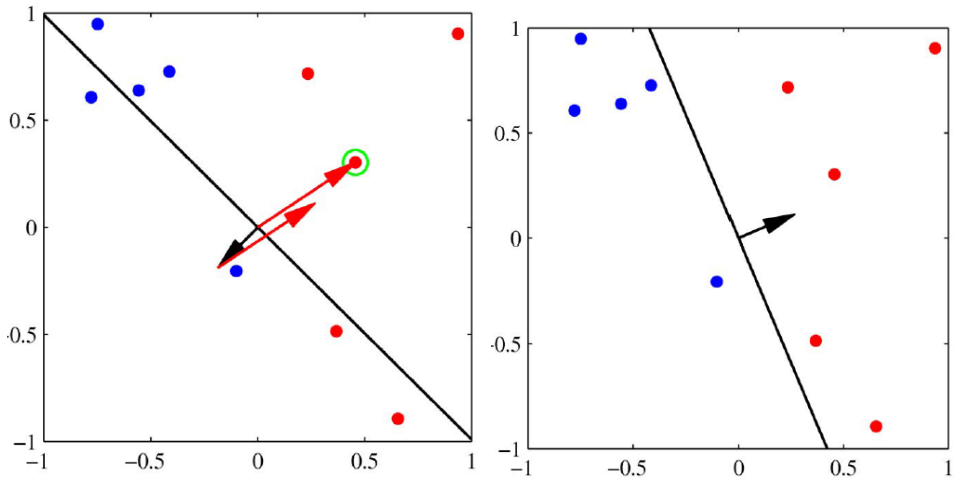
This error is zero if all points are correctly classified (in which case $\mathcal{M}$ is empty). If a point is misclassified, $t_n \mathbf{w}^T\phi(x_n)$ is negative, so each misclassified point contributes a positive amount to E_P.

Minimizing E_P directly is difficult because it's a piecewise-linear and non-differentiable function of $\mathbf{w}$ (the set $\mathcal{M}$ changes discontinuously as $\mathbf{w}$ changes). However, we can derive a simple **stochastic gradient descent** algorithm. The key observation is that a misclassified point has $t_n \mathbf{w}^T\phi(x_n) < 0$, so in order to reduce the error, we want to adjust $\mathbf{w}$ to make $t_n \mathbf{w}^T\phi(x_n)$ larger (closer to positive). A stochastic gradient step on E_P for a single misclassified example n gives the update:

$$\mathbf{w}^{r+1} = \mathbf{w}^r - \eta \nabla E_P(\mathbf{w}) = \mathbf{w}^r + \eta \, t_n \, \phi(\mathbf{x}_n) \ ,$$

where η is a learning rate (a positive constant). Intuitively, if $t_n = +1$ but the prediction $\mathbf{w}^T\phi(x_n)$ was negative (too low), we add $\phi(x_n)$ to $\mathbf{w}$ to increase the output. If $t_n = -1$ but $\mathbf{w}^T\phi(x_n)$ was positive, we subtract $\phi(x_n)$ from $\mathbf{w}$. This update pushes the decision boundary in the direction that corrects the mistake. One can show that this update is equivalent to performing gradient descent on E_P for that example.





Typically we iterate over training examples, applying the above update whenever we encounter an example that is misclassified, until no errors remain or a maximum number of iterations is reached. Because the update does not depend on $\mathbf{w}$ in a complex way (it's just addition of a vector, and η can even be set to 1 for simplicity), the perceptron update is straightforward.

Convergence: The perceptron convergence theorem states that if the training data is linearly separable (i.e., there exists *some* $\mathbf{w}$ that classifies all points correctly), then the perceptron algorithm will eventually find a solution that perfectly classifies the training data. In fact, one can bound the number of updates it will make before convergence (it will be finite). However, if the data is not linearly separable, the perceptron will not converge; it will continue to loop through the data making updates indefinitely.

Historical note: The perceptron was initially greeted with enthusiasm, but Minsky and Papert (1969) famously showed limitations of the perceptron (specifically, a single-layer perceptron cannot learn certain functions like XOR). Their analysis dampened research in neural networks for a number of years. It was later realized that multi-layer perceptrons (with non-linear hidden layers) overcome these limitations, leading to the development of modern neural networks.

Probabilistic Generative Models for Classification

Thus far we have discussed linear discriminants from a mostly geometric or algorithmic perspective. We now take a probabilistic approach. In a **generative model** for classification, we model how data in each class is generated by specifying class-conditional probability density functions $p(\mathbf{x}|C_k)$ and class prior probabilities $p(C_k)$. Given these, we can use Bayes' theorem to compute the posterior probability of class membership for a new point $\mathbf{x}$, and then choose the class with highest posterior.

Two-Class Posteriors and the Logistic Sigmoid

For simplicity, first consider the case of two classes C_1 and C_2. From the definition of conditional probability:

$$p(C_1|\mathbf{x}) = \frac{p(\mathbf{x}|C_1)\,p(C_1)}{p(\mathbf{x}|C_1)p(C_1) + p(\mathbf{x}|C_2)p(C_2)} \ ,$$

using $p(C_2|\mathbf{x}) = 1 - p(C_1|\mathbf{x})$ implicitly in the denominator. We can rearrange this expression into a more illuminating form. Define the log-odds (logit) a for class C_1 as:

$$a \ = \ \ln \frac{p(\mathbf{x}|C_1)\,p(C_1)}{p(\mathbf{x}|C_2)\,p(C_2)} \ .$$

Then we can write the posterior as:

$$p(C_1|\mathbf{x}) \ = \ \frac{1}{1+\exp(-a)} \ .$$

The right-hand side is the familiar **logistic sigmoid** function of a, often denoted $\sigma(a)$. In fact:

$$p(C_1|\mathbf{x}) = \sigma(a) \ , \qquad \text{where } \ \sigma(z) = \frac{1}{1+e^{-z}} \ .$$

The inverse of the sigmoid (the logit function) is $\sigma^{-1}(p) = \ln \frac{p}{1-p}$, consistent with our definition of a as a log-odds.

Crucially, a is a linear function of $\mathbf{x}$ **if** the class-conditional densities $p(\mathbf{x}|C_k)$ belong to an exponential family. For example, if $p(\mathbf{x}|C_k)$ are Gaussian with shared covariance (as we will detail shortly), then a will turn out linear in x. In such cases, $p(C_1|\mathbf{x})$ becomes a sigmoid of a linear function, i.e. a logistic regression model. This links generative and discriminative viewpoints: the logistic regression can be derived from assuming a certain generative model.

(We have derived $p(C_1|\mathbf{x})$ as a sigmoid; of course $p(C2|\mathbf{x}) = 1 - p(C_1|\mathbf{x}) = \sigma(-a)$.)

For **multiple classes** ($K > 2$), we generalize the logistic sigmoid to the **softmax function**. Define for each class C_k a score

$$a_k = \ln(p(\mathbf{x}|C_k)\,p(C_k)) \ .$$

These are like unnormalized log-posteriors (log probabilities up to an additive constant). The posterior for class C_k is given by normalizing the exponentials of these scores:

$$p(C_k|\mathbf{x}) = \frac{\exp(a_k)}{\sum_{j=1}^{K} \exp(a_j)} \ .$$

This is the softmax. It can be viewed as a "soft" version of taking a maximum: whichever a_k is largest will correspond to the largest posterior, but all classes get a positive probability. The softmax ensures $\sum_k p(C_k|\mathbf{x}) = 1$ and $0 < p(C_k|\mathbf{x}) < 1$. Different functional forms of $p(\mathbf{x}|C_k)$ will yield different discriminant functions $a_k(\mathbf{x})$ and thus different decision boundaries (linear or nonlinear).

Gaussian Generative Model (Continuous Inputs)

To make this concrete, consider a common assumption: **Gaussian class-conditional densities with shared covariance**. Suppose $\mathbf{x} \in \mathbb{R}^D$ and for each class C_k,

$$p(\mathbf{x}|C_k) = \mathcal{N}(\mathbf{x} \mid \mu_k, \Sigma) \ ,$$

a Gaussian with mean μ_k and a covariance matrix Σ that is the same for all classes. Also let the class priors be $p(C_k)$. In the two-class case ($k = 1, 2$), the log-odds a becomes:

$$a = \ln \frac{\mathcal{N}(\mathbf{x}|\mu_1, \Sigma)\, p(C_1)}{\mathcal{N}(\mathbf{x}|\mu_2, \Sigma)\, p(C_2)} \ .$$

We can expand this using the Gaussian density formula. The Gaussian density in D dimensions is:

$$\mathcal{N}(\mathbf{x}|\mu, \Sigma) = \frac{1}{(2\pi)^{D/2}|\Sigma|^{1/2}} \exp\Big\{ -\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\Big\} \ .$$

Taking the ratio inside the log, the normalizing constants (the $(2\pi)^{-D/2}|\Sigma|^{-1/2}$ terms) cancel out since Σ is shared. We get:

$$a = \ln p(C_1) - \ln p(C_2) \ - \ \frac{1}{2}(\mathbf{x}-\mu_1)^T\Sigma^{-1}(\mathbf{x}-\mu_1) \ + \ \frac{1}{2}(\mathbf{x}-\mu_2)^T\Sigma^{-1}(\mathbf{x}-\mu_2) \ .$$

This quadratic expression in $\mathbf{x}$ actually simplifies to a **linear** function of $\mathbf{x}$. After expanding the quadratic terms, many cancel out, leaving:

$$a = \mathbf{w}^T\mathbf{x} + w_0 \ ,$$

where

$$\mathbf{w} = \Sigma^{-1}(\mu_1 - \mu_2) \ , \qquad w_0 = -\frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \ln\frac{p(C_1)}{p(C_2)} \ .$$

Thus $p(C_1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0)$ is a logistic-regression classifier. We see that **when Gaussian class distributions share the same covariance, the optimal decision boundary is linear**. In contrast, if each class had its own covariance matrix Σ_k, the log-odds $a(\mathbf{x})$ would generally be a quadratic function of $\mathbf{x}$, leading to **quadratic decision boundaries**.

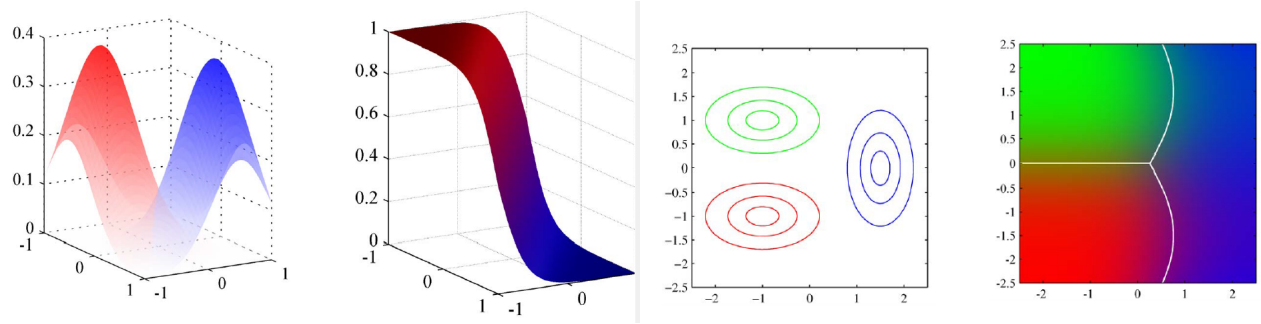


Figure illustrates this: with shared covariance, the decision boundary between two Gaussian clusters is a straight line (hyperplane in higher dimensions). If the covariances differ, the boundary becomes a curved quadratic surface.

Maximum Likelihood Parameter Estimation

In a generative approach, we often need to **fit the parameters** of our model (e.g., the class means, covariances, and priors for a Gaussian model) from data. This is typically done by maximum likelihood estimation (MLE).

For example, in the two-class Gaussian scenario above, let's denote by N_1 the number of training points belonging to class C_1 and N_2 the number belonging to C_2 (with $N_1 + N_2 = N$). We can introduce binary labels $t_n \in \{0, 1\}$ for each point, where $t_n = 1$ if $\mathbf{x}_n$ is class C_1 and $t_n = 0$ if it's class C_2. The likelihood of the data (jointly with labels) given parameters $\{q, \mu_1, \mu_2, \Sigma\}$ (where $q = p(C_1)$ and $p(C_2) = 1 - q$) is:

$$p(\mathbf{t}, \mathbf{X}|q, \mu_1, \mu_2, \Sigma) = \prod_{n=1}^{N} \left[\, q \, \mathcal{N}(\mathbf{x}_n|\mu_1, \Sigma) \, \right]^{t_n} \left[\, (1-q) \, \mathcal{N}(\mathbf{x}_n|\mu_2, \Sigma) \, \right]^{(1-t_n)} .$$

Taking the log of the likelihood, we can separate terms. The terms involving q are:

$$\sum_{n=1}^{N} \left[t_n \ln q + (1 - t_n) \ln(1 - q) \right] ,$$

which is maximized w.rt. q by $q = \frac{1}{N} \sum_{n=1}^{N} t_n = \frac{N_1}{N}$ (so the MLE for the prior is just the empirical class proportion). The terms involving μ_1 appear in

$$\sum_{n=1}^{N} t_n \ln \mathcal{N}(\mathbf{x}_n|\mu_1, \Sigma) ,$$

which, ignoring constants, simplifies to

$$-\frac{1}{2} \sum_{n=1}^{N} t_n (\mathbf{x}_n - \mu_1)^T \Sigma^{-1} (\mathbf{x}_n - \mu_1) .$$

Maximizing this with respect to μ_1 yields

$$\mu_1 = \frac{\sum_{n=1}^{N} t_n \mathbf{x}_n}{\sum_{n=1}^{N} t_n} = \frac{1}{N_1} \sum_{n=1}^{N} t_n \mathbf{x}_n ,$$

i.e. the MLE for the mean is the sample mean of class 1 points. Similarly, μ_2 is the sample mean of class 2 points. The shared covariance MLE turns out to be the weighted average of the class covariances (pooled covariance).

In summary, modeling $p(\mathbf{x}|C_k)$ with Gaussian distributions and fitting by MLE yields a classifier that is very similar to logistic regression, but with potentially many more parameters if D is large (due to the covariance matrix). This leads to our next topic.

Logistic Regression (Discriminative Approach)

Instead of modeling class densities, **logistic regression** directly models the posterior probabilities of classes given $\mathbf{x}$, using a linear function passed through a logistic (sigmoid or softmax) non-linearity. It is a **discriminative model** (focused on $p(\text{class}|\mathbf{x})$ rather than $p(\mathbf{x}|\text{class})$). We already encountered logistic regression in the generative context; now we consider it as a standalone approach.

For two classes, logistic regression assumes:

$$p(C_1|\phi(\mathbf{x})) = y(\mathbf{x}) = \sigma(\mathbf{w}^T\phi(\mathbf{x})) \ ,$$

$$p(C_2|\mathbf{x}) = 1 - y(\mathbf{x}) = \sigma(-\mathbf{w}^T\phi(\mathbf{x})) \ ,$$

$$\frac{\partial\sigma(a)}{\partial a} = \sigma(a)(1-\sigma(a))$$

where $\sigma(z) = 1/(1+e^{-z})$ as before. Here $\phi(\mathbf{x})$ is a chosen feature vector (which could simply be $[1, x_1, x_2, \ldots, x_D]^T$ including a constant 1 for bias). The parameters to learn are $\mathbf{w}$.

More compact than maximum likelihood fitting of Gaussians. For $\mathbf{M}$ parameters, Gaussian model uses $\mathbf{2M}$ parameters for the means, and $\frac{1}{2}\mathbf{M}(\mathbf{M}-1)$ parameters for the shared covariance matrix

We typically fit $\mathbf{w}$ by **maximum likelihood**. Given a set of independent training examples, the likelihood (conditional on inputs) is:

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} [y_n]^{t_n} \, [\, 1 - y_n \,]^{\, 1-t_n} \ ,$$

where $y_n = \sigma(\mathbf{w}^T\phi(x_n))$ and $t_n \in \{0, 1\}$ is the class label. This is the product of Bernoulli probabilities for each data point. It is more convenient to maximize the log-likelihood:

$$\ln L(\mathbf{w}) = \sum_{n=1}^{N} \Big\{ \, t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \Big\} \ .$$

Negative log-likelihood (NLL) and cross-entropy: We often minimize the negative log-likelihood, which in this case is:

$$E(\mathbf{w}) = -\ln L(\mathbf{w}) = -\sum_{n=1}^{N} \Big[\, t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \Big] \ .$$

This $E(\mathbf{w})$ is exactly the **cross-entropy** error function for two-class classification. It measures the discrepancy between the predicted probabilities y_n and the true labels t_n. (If we were to use $t_n \in \{-1, 1\}$, a similar error function can be formulated, but the 0-1 encoding is more common for logistic regression.)

We can find the gradient of the negative log-likelihood with respect to $\mathbf{w}$:

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} (y_n - t_n)\, \phi(\mathbf{x}_n) \ .$$

Derivation: $\partial E / \partial \mathbf{w} = -\sum_n [t_n \frac{\partial \ln y_n}{\partial \mathbf{w}} + (1 - t_n) \frac{\partial \ln(1-y_n)}{\partial \mathbf{w}}]$. Using $\frac{\partial \ln y_n}{\partial \mathbf{w}} = (1 - y_n)\, \phi(x_n)$ and $\frac{\partial \ln(1-y_n)}{\partial \mathbf{w}} = -y_n\, \phi(x_n)$ (these come from the derivative of the sigmoid), the terms combine to $(y_n - t_n)\phi(x_n)$.

Setting the gradient to zero gives the **normal equations** for logistic regression:

$$\sum_{n=1}^{N} (y_n - t_n)\, \phi(x_n) = 0 \ .$$

However, unlike linear regression (where such normal equations could be solved in closed form), here y_n depends on $\mathbf{w}$ in a nonlinear way, so we cannot solve for $\mathbf{w}$ analytically. Instead, we must use **iterative numerical optimization** to find the $\mathbf{w}$ that minimizes $E(\mathbf{w})$.

A common choice is to use **Newton-Raphson** method, which in this context leads to the **Iteratively Reweighted Least Squares (IRLS)** algorithm. Newton-Raphson updates have the form:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - H^{-1} \nabla E(\mathbf{w}_{\text{old}}) \ ,$$

where H is the Hessian matrix (second derivative matrix) of $E(\mathbf{w})$. For the logistic regression error, one can show that the Hessian is:

$$H = \sum_{n=1}^{N} y_n (1 - y_n)\, \phi(x_n) \phi(x_n)^T = \Phi^T R \Phi \ ,$$

where we stack the feature vectors $\phi(x_n)$ into a design matrix Φ, and R is an $N \times N$ diagonal matrix with $R_{nn} = y_n(1 - y_n)$. The gradient can be written as $\nabla E = \Phi^T (y - t)$ where y and t are the vectors of predictions and targets respectively. Plugging these into the Newton update:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - (\Phi^T R \Phi)^{-1} \Phi^T (y - t) \ .$$

This update can be derived as solving a weighted least squares problem in each iteration. In fact, one can define a *working variable* (sometimes called the adjusted response)

$$\mathbf{z} = \Phi \mathbf{w}_{\text{old}} - R^{-1} (y - t) \ ,$$

and then show the above update is equivalent to fitting a linear regression model $\Phi w \approx z$ with weight matrix R. Each Newton step re-estimates $\mathbf{w}$, and then y, R, and z are recomputed, and the process repeats until convergence. This procedure is the IRLS algorithm for logistic regression.

For comparison, if we apply Newton's method to the sum-of-squares error for linear regression, the Hessian is constant (independent of w) and equal to $\Phi^T \Phi$, so Newton-Raphson jumps to

the solution in one step (since setting the gradient to zero gives $\Phi^T\Phi\, w = \Phi^T t$ which is solved by $w = (\Phi^T\Phi)^{-1}\Phi^T t$). In logistic regression, the effective "normal equations" $\Phi^T R\Phi\, w = \Phi^T Rz$ depend on w through R and z, so we must iterate.

In summary, logistic regression provides a more direct route to classification than generative models, with typically fewer parameters when D is large (no covariance matrix to estimate). Its training involves optimizing a convex objective (cross-entropy), which can be done efficiently with algorithms like gradient descent or Newton-Raphson/IRLS. The result is a linear decision boundary (if using the raw inputs or fixed basis functions $\phi(x)$) similar in form to the ones discussed earlier, but arrived at from a discriminative learning perspective rather than an assumption of underlying generative distributions.