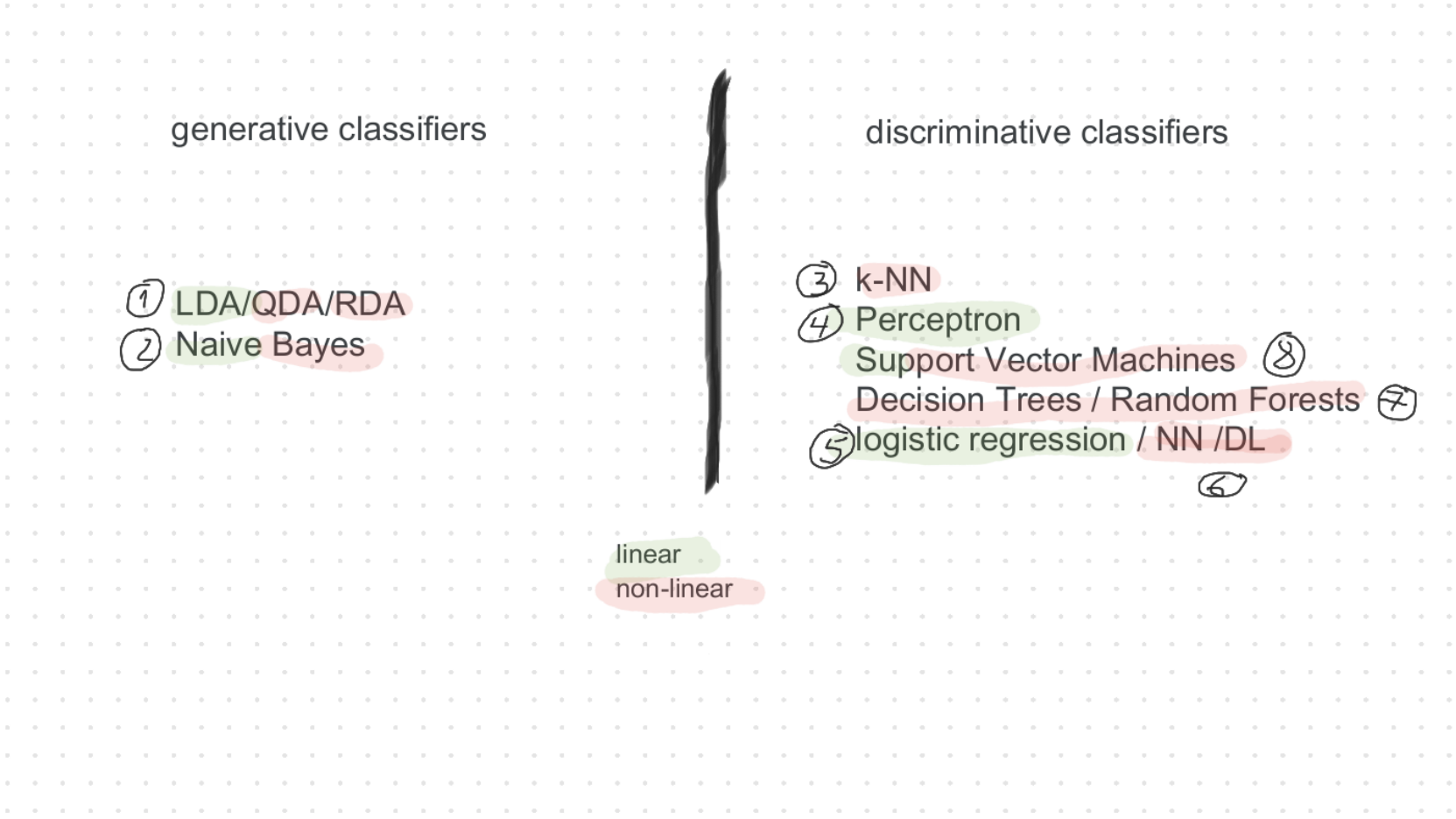
Machine Learning

FIB, Master in Data Science

Marta Arias, Computer Science @ UPC

Topic 2: Linear classifiers

Classifiers (linear or otherwise)



Classification setup

Decision theory for supervised learning

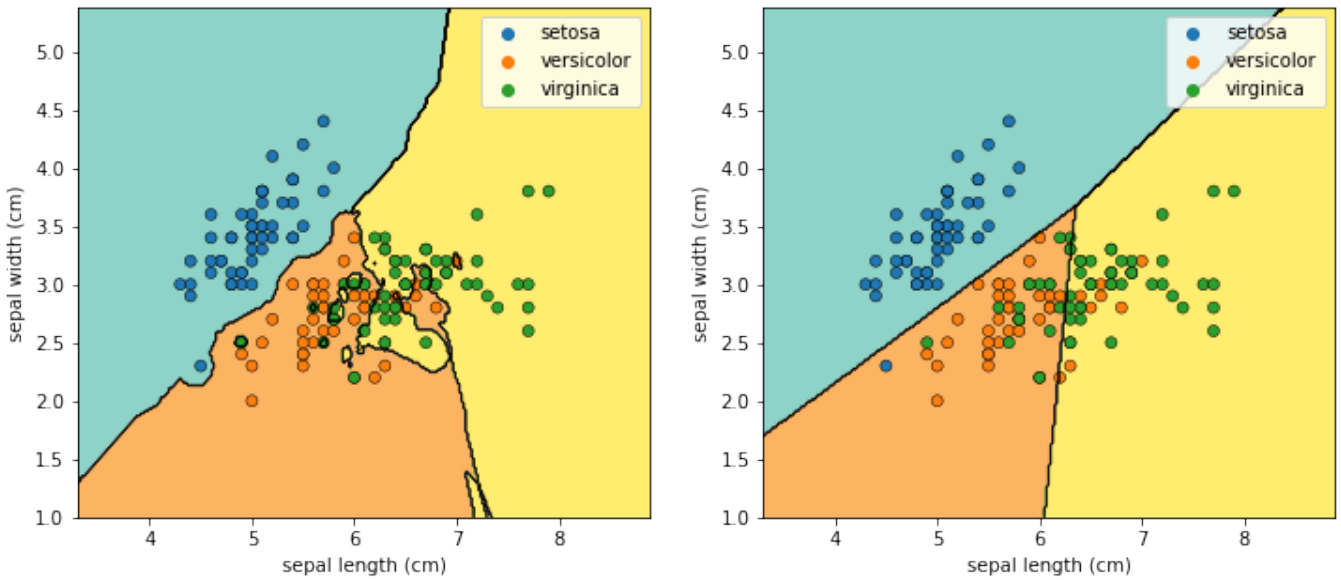
We depart from a finite labelled dataset $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathcal{Y}$ with $\mathcal{Y} = \{c_1, \ldots, c_K\}$. The set $\mathcal{Y}$ is the set of possible *labels* that examples $\mathbf{x}$ are associated with.

- In **binary classification** we have that $|\mathcal{Y}| = 2$, for example:
  - classify an email as spam/non-spam
  - classify an image as a cat/non-cat
  - predict tomorrow's weather as rainy/sunny
  - classify movie review as positive/negative
  - ...
- In **multi-class classification** we have that $|\mathcal{Y}| > 2$, for example:
  - classify news article as sports/politics/science/entertainment
  - predict tomorrow's weather as rainy/cloudy/sunny
  - ...

The aim of a good classifier is to predict the correct label for future unknown examples $\mathbf{x}$.

Useful terminology

- The **decision regions** are those regions of feature space where all points are assigned the same label
- The **decision boundaries** are those points in the frontier between decision regions
- **Linear classifiers** have decision boundaries represented by $d-1$-dimensional hyperplanes



Probabilistic classifiers, binary case

Many useful classifiers apart from predicting an input example's *class*, also provide a probabilistic prediction of how likely it is they belong to a certain class.

This is very useful since it allows to express *uncertainty* around a prediction

In binary classification, the target values $\mathcal{Y}$ are encoded as $y \in \{0, 1\}$ and predictions are given as a continuous value in the $[0, 1]$ range, i.e. $\hat{y} \in [0, 1]$:

- $\hat{y} = 0.9$ means that the probability that the example is of class "1" is 0.9, and of class "0" is 0.1

Probabilistic classifiers, multi-class case

In classification with $K > 2$ classes, it is common to use one-hot encoding, and so $y \in \{0, 1\}^K$

- $y = (0, 0, 1, 0)$ means that the example belongs to the *third* class
- $y = (1, 0, 0, 0)$ means that the example belongs to the *first* class

Predictions in this case are typically members of the $(K - 1)$-simplex:

$$\hat{y} = (\hat{y}_1, \ldots, \hat{y}_K)$$

with $0 \leq \hat{y}_k \leq 1$ for all k s.t. $1 \leq k \leq K$, and $\sum_k \hat{y}_k = 1$.

E.g. $\hat{y} = (0.6, 0.1, 0, 0.3)$ means that the classifier thinks the example belongs to class "1" with prob. 0.6, etc.

Decision boundary in probabilistic models

We assume that there is an unknown distribution over examples and target values (classes) that governs the datasets that we obtain. This is the *joint* distribution of examples and labels $p(\mathbf{x}, y)$ where $\mathbf{x} \in \mathbb{R}^d$ and $y \in \mathcal{Y}$.

On building a classifier we want to "minimize an expected loss", in the same way we did for regression (there, we used squared error as the measure of loss). Here, loss is slightly different since we have different types of targets (discrete classes). So a natural way of looking at this concept is through **loss** or **cost matrices**.

Using an example in the context of a medical screening test for cancer:

	is healthy ($y = 0$)	has cancer ($y = 1$)
predicted healthy ($\hat{y} = 0$)	0	50
predicted sick ($\hat{y} = 1$)	10	0

In this context, it is worse to leave a sick patient undetected (because the patient cannot undergo therapy) than scaring someone off unnecessarily (bad, too, but further tests can correct the wrong early diagnosis).

Decision boundary in probabilistic models, cont.

In this course we will focus however in the "0-1" loss mostly (wrong guess costs "1", right guess costs "0").

Now suppose you are given a new example $\mathbf{x}$ that we have to classify into one of $\mathcal{Y}$ classes. What we want is a "rule" that tells us how to choose a good $\hat{y} \in \mathcal{Y}$ for $\mathbf{x}$.

In general, we have r.v. X and Y with joint distribution $p(X, Y)$, where X is multi-dimensional input example and $Y \in \mathcal{Y}$ a label. We use the conditional distribution $p(Y|\mathbf{x}) := p(Y|X = \mathbf{x})$ in the following derivation. Here, we are computing the expected loss of assigning $\mathbf{x}$ to class label $c \in \mathcal{Y}$:

$$\begin{aligned}
\mathbb{E}_Y[L(Y, c)] &= \sum_{y \in \mathcal{Y}} L(y, c) p(Y = y|\mathbf{x}) \\
&= \sum_{y \neq c} p(Y = y|\mathbf{x}) \\
&= 1 - p(Y = c|\mathbf{x})
\end{aligned}$$

Expected loss, example

	true a	true b	true c
predicted a	0	50	25
predicted b	10	0	500
predicted c	15	10	0

$$\mathbb{E}_Y[L(Y, \text{pred. a})] = \sum_{y \in \{\text{a,b,c}\}} L(y, \text{pred. a}))p(Y = y|\mathbf{x})$$
$$= 0 + 50p(Y = \text{b}|\mathbf{x}) + 25p(Y = \text{c}|\mathbf{x})$$

$$\mathbb{E}_Y[L(Y, \text{pred. b})] = \sum_{y \in \{\text{a,b,c}\}} L(y, \text{pred. b}))p(Y = y|\mathbf{x})$$
$$= 10p(Y = \text{a}|\mathbf{x}) + 0 + 500p(Y = \text{c}|\mathbf{x})$$

$$\mathbb{E}_Y[L(Y, \text{pred. c})] = \sum_{y \in \{\text{a,b,c}\}} L(y, \text{pred. c}))p(Y = y|\mathbf{x})$$
$$= 15p(Y = \text{a}|\mathbf{x}) + 10p(Y = \text{b}|\mathbf{x}) + 0$$

Expected loss, example with 0-1 loss

	true a	true b	true c
predicted a	0	1	1
predicted b	1	0	1
predicted c	1	1	0

$$\begin{aligned}\mathbb{E}_Y[L(Y, \text{pred. a})] &= 0 + p(Y = \text{b}|\mathbf{x}) + p(Y = \text{c}|\mathbf{x}) \\ &= 1 - p(Y = \text{a}|\mathbf{x})\end{aligned}$$

$$\begin{aligned}\mathbb{E}_Y[L(Y, \text{pred. b})] &= 1p(Y = \text{a}|\mathbf{x}) + 0 + 1p(Y = \text{c}|\mathbf{x}) \\ &= 1 - p(Y = \text{b}|\mathbf{x})\end{aligned}$$

$$\begin{aligned}\mathbb{E}_Y[L(Y, \text{pred. c})] &= p(Y = \text{a}|\mathbf{x}) + p(Y = \text{b}|\mathbf{x}) + 0 \\ &= 1 - p(Y = \text{c}|\mathbf{x})\end{aligned}$$

Decision boundary in probabilistic models, cont.

To minimize expected loss we want to predict a class $\hat{y}$ for which $\mathbb{E}_Y[L(Y, \hat{y})]$ is minimized, so we want

$$\begin{aligned}\hat{y} &= \arg\min_y \mathbb{E}_Y[L(Y, y)] \\ &= \arg\min_y 1 - p(y|\mathbf{x}) \\ &= \arg\max_y p(y|\mathbf{x})\end{aligned}$$

This is called the **Bayes classifier**, and *it is optimal under 0-1 loss*. Its error is given by the sum over all possible $\mathbf{x}$, and it is called the **Bayes error rate**:

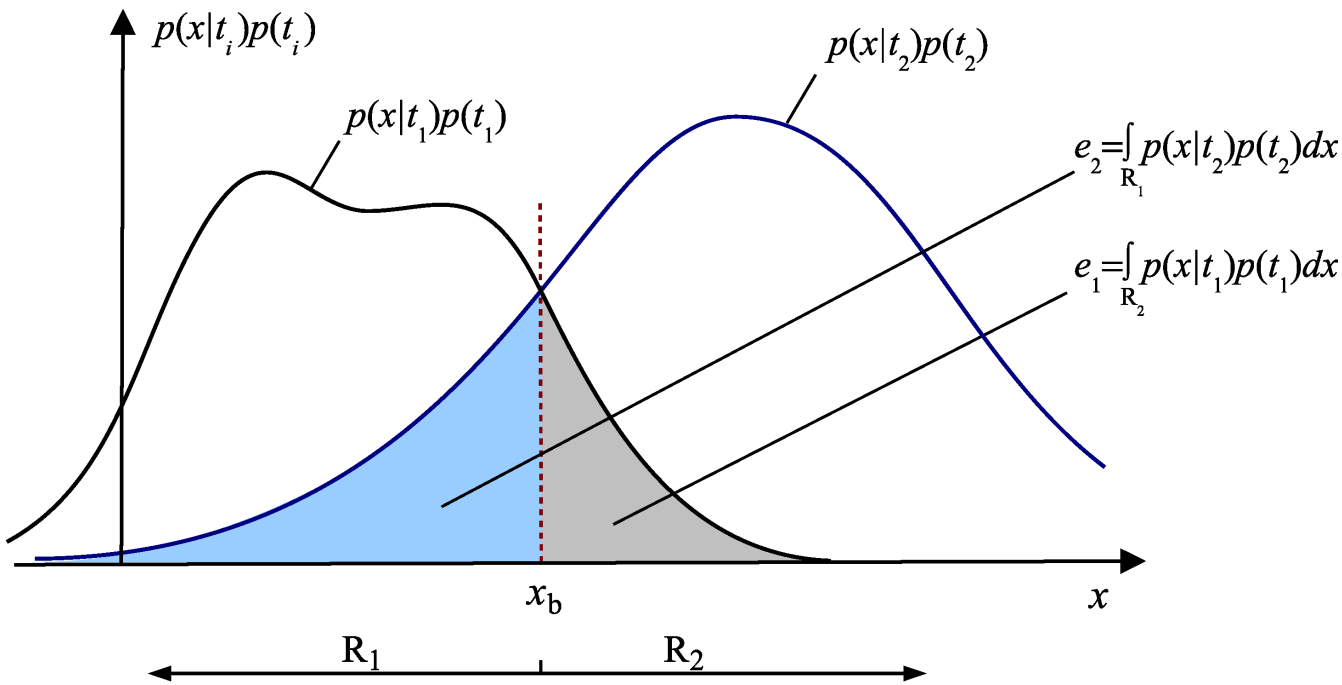
$$\begin{aligned}\text{Bayes error rate} = 1 - \mathbb{E}_X[p(\hat{y}|\mathbf{x})] &= 1 - \int_{\mathbf{x}} p(\hat{y}|\mathbf{x})p(\mathbf{x})d\mathbf{x} \\ &= 1 - \int_{\mathbf{x}} p(\mathbf{x}|\hat{y})p(\hat{y})d\mathbf{x}\end{aligned}$$

where $\hat{y} = \arg\max_y p(y|\mathbf{x})$

Decision boundary in probabilistic models, cont.

If we use the *Bayes classifier* to partition the input feature space into regions $\mathcal{R}_c$ where $c \in \mathcal{Y}$, then we can write the above error as follows:

$$\text{Bayes error rate} = 1 - \sum_c \int_{\mathbf{x} \in \mathcal{R}_c} p(\mathbf{x}|c)p(c)d\mathbf{x}$$



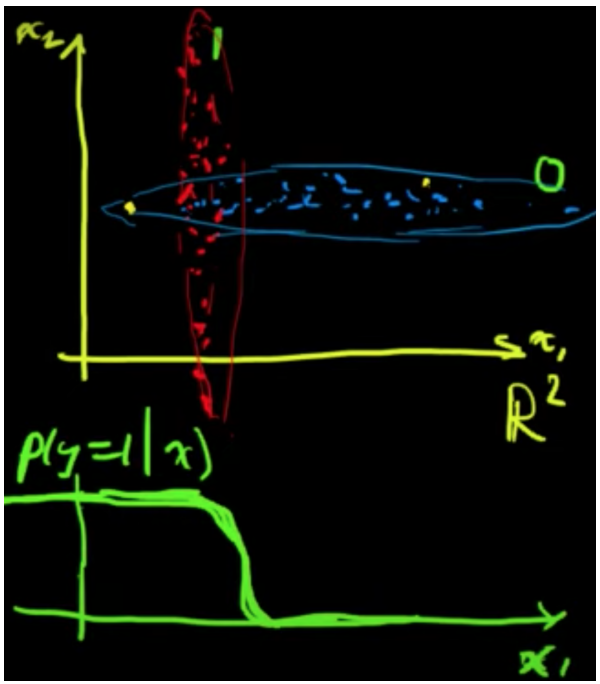
Decision boundary in probabilistic models, cont.

The *Bayes classifier* is optimal, however in practice of course we do not know what $p(y, \mathbf{x})$ is, so we cannot implement it exactly. Typically, Bayesian classifiers will estimate/learn $p(y|\mathbf{x})$ from data and will use these estimations instead, incurring in additional error.

But first ... two different ways of learning $p(y|\mathbf{x})$ ...

Discriminative vs. generative classifiers[1]

In Bayesian classifiers, the rule to classify a new example $\mathbf{x}$ is $\hat{y} := \arg\max_y p(y|\mathbf{x})$. We do not know the exact distribution and therefore the classifier's job is to *learn* it from a finite dataset.



There are two approaches for learning $p(y|\mathbf{x})$:

1. Discriminative:
   - directly learn $p(y|\mathbf{x})$
2. Generative:
   - learn $p(y|\mathbf{x})$ through Bayes rule $p(y|\mathbf{x}) \propto p(\mathbf{x}|y)p(y)$

[1] See this nice video explanation by Yoav Freund.

Generative classifiers

LDA/QDA/RDA and Naive Bayes

Discriminant analysis

Discriminant analysis is the result of implementing Bayes classifier under the assumption that **class-conditional distributions $p(\mathbf{x}|y)$ are gaussian**. If $\mathcal{Y} = \{c_1, \ldots, c_K\}$, then for all $1 \leq k \leq K$:

$$p(\mathbf{x}|y = c_k) \sim \mathcal{N}(\mu_k, \Sigma_k)$$

If we further assume that the prior distributions are $p(y = c_k) = \pi_k$ for all k, with $\sum_k \pi_k = 1$, then we define the **discriminant functions** $g_k(\mathbf{x})$ for each class c_k:

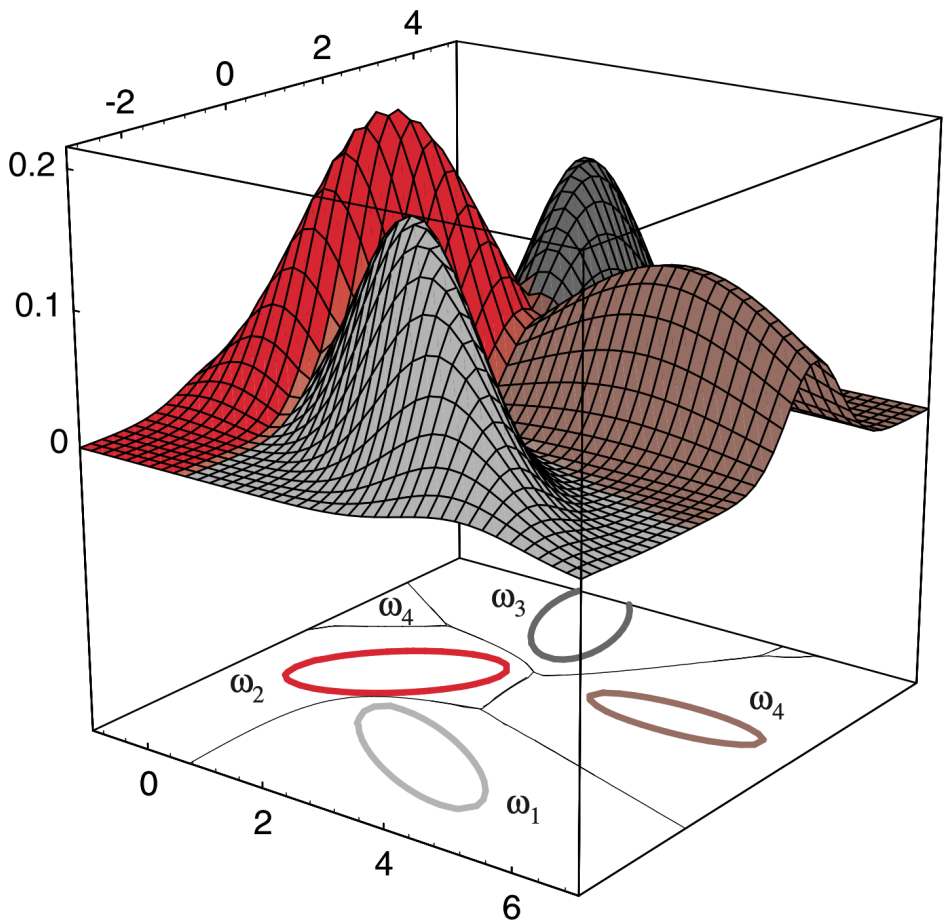
$$\begin{aligned} g_k(\mathbf{x}) &= \log\left[P(y = c_k)P(\mathbf{x}|y = c_k)\right] \\ &= \log \pi_k - \log(|2\pi\Sigma_k|^{\frac{1}{2}}) - \frac{1}{2}(\mathbf{x} - \mu_k)^T\Sigma_k^{-1}(\mathbf{x} - \mu_k) \\ &= \log \pi_k - \frac{1}{2}\left(\log|\Sigma_k| + (\mathbf{x} - \mu_k)^T\Sigma_k^{-1}(\mathbf{x} - \mu_k)\right) + const \end{aligned}$$

The function $g_k(\mathbf{x})$ is called a **quadratic discriminant function**, and the corresponding classifier is implemented by predict class $\hat{y} = \arg\max_k g_k(\mathbf{x})$ which corresponds to choosing the label with maximum probability a posteriori $P(y = c_k|\mathbf{x})$.

Discriminant analysis, the general case

QDA

The **decision boundaries** for this classifier are those regions such that $g_k(\mathbf{x}) = g_{k'}(\mathbf{x})$; they correspond to **hyper-quadrics** in feature space, and so this is a quadratic (non-linear) method.



Picture taken from Duda et al. book Pattern Classification. In fact, Chapter~2.6 contains the full details on this method.

Discriminant analysis with same covariance matrices

LDA

If we assume that the class-conditional densities share the **same covariance matrix** Σ, then we can simplify the corresponding discriminant functions to:

$$g_k(\mathbf{x}) = \log \pi_k + \mu_k^T \Sigma^{-1} \mathbf{x} - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k$$

These functions are **linear discriminant functions**; the **decision boundaries** for the resulting Bayesian classifier correspond to **hyperplanes** in feature space, and so this is a linear method.

Discriminant analysis, further assumptions

- If we further assume that $\Sigma = diag(\sigma_1^2, \ldots, \sigma_d^2)$ is **diagonal** (so input features are independent from each other), then:

$$g_k(\mathbf{x}) = \log \pi_k - \frac{1}{2} \sum_{j=1}^{d} \frac{(\mu_{kj} - x_j)^2}{\sigma_j^2}$$

- If Σ is an isotropic Gaussian, that is, $\Sigma = \sigma^2 I$, then

$$g_k(\mathbf{x}) = \log \pi_k - \frac{1}{2\sigma^2} \|\mu_k - \mathbf{x}\|^2$$

- Finally, if all priors are equal, that is, $\pi_k = \frac{1}{K}$, then

$$g_k(\mathbf{x}) = -\frac{1}{2} \|\mu_k - \mathbf{x}\|^2$$

Discriminant analysis, distance-based learning

In all cases, we have a **minimum-distance** classifier in $\mathbb{R}^d$:

- In the general QDA case (each class with its own covariance matrix), the classifier uses a different Mahalanobis distance from $\mathbf{x}$ to each class center μ_k
- In the LDA case where all covariance matrices are equal, the classifier uses the same Mahalanobis distance from $\mathbf{x}$ to each class center
- In the case where all covariance matrices are equal and diagonal, the classifier uses a weighted Euclidean distance
- In the case where all covariance matrices are equal and proportional to the identity matrix, the classifier uses the unweighted Euclidean distance

Discriminant analysis, implementation

In order to apply QDA or LDA or its simpler variants, we need to know the shapes and centers of each Gaussian representing the classes. A natural choice is to use MLE and estimates these from the available dataset $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$.

In the following, S_k is the subset of examples that belong to class c_k, i.e. $S_k = \{\mathbf{x}_i | y_i = c_k\}$, and $n_k = |S_k|$.

$$\hat{\pi}_k = \frac{n_k}{n} \qquad \hat{\mu}_k = \frac{1}{n_k} \sum_{\mathbf{x} \in S_k} \mathbf{x}$$

In QDA (different covariance matrices):

$$\hat{\Sigma}_k = \frac{1}{n_k - 1} \sum_{\mathbf{x} \in S_k} (\mathbf{x} - \mu_k)(\mathbf{x} - \mu_k)^T$$

In LDA (same covariance matrix):

$$\hat{\Sigma} = \frac{1}{n - n_k} \sum_{k=1}^{K} (n_k - 1)\hat{\Sigma}_k$$

Bayesian classifiers are **optimal** when the class-conditional densities and priors are known; the methods are well-principled, fast and reliable

For Gaussian classes, we get a quadratic classifier - QDA (if all covariance matrices are equal, a linear classifier - LDA); using a specific distance function corresponds to certain statistical assumptions:

- If the class-conditional densities are far from the assumptions, the model will be poor
- Even if the class-conditional densities are Gaussian, the parameters should be reliably estimated (particularly for QDA)
- Once we use sample statistics instead of population parameters, we loose optimality!

The question whether these assumptions hold can rarely be answered in practice; in most cases we are limited to posing and answering the question "does this classifier give satisfactory predictions or not?"

Regularized discriminant analysis (RDA)

When data is scarce, some problems may arise, e.g.

- If $d > n_k$ for some k, then **QDA cannot be applied** because $\hat{\Sigma}_k$ is going to be singular.
- If $d > n$ then **QDA nor LDA can be applied** because each $\hat{\Sigma}_k$ and $\hat{\Sigma}$ are singular.

Regularized discriminant analysis (RDA) computes covariances as:

$$\hat{\Sigma}_k(\alpha) = \alpha\hat{\Sigma}_k + (1-\alpha)\hat{\Sigma}$$

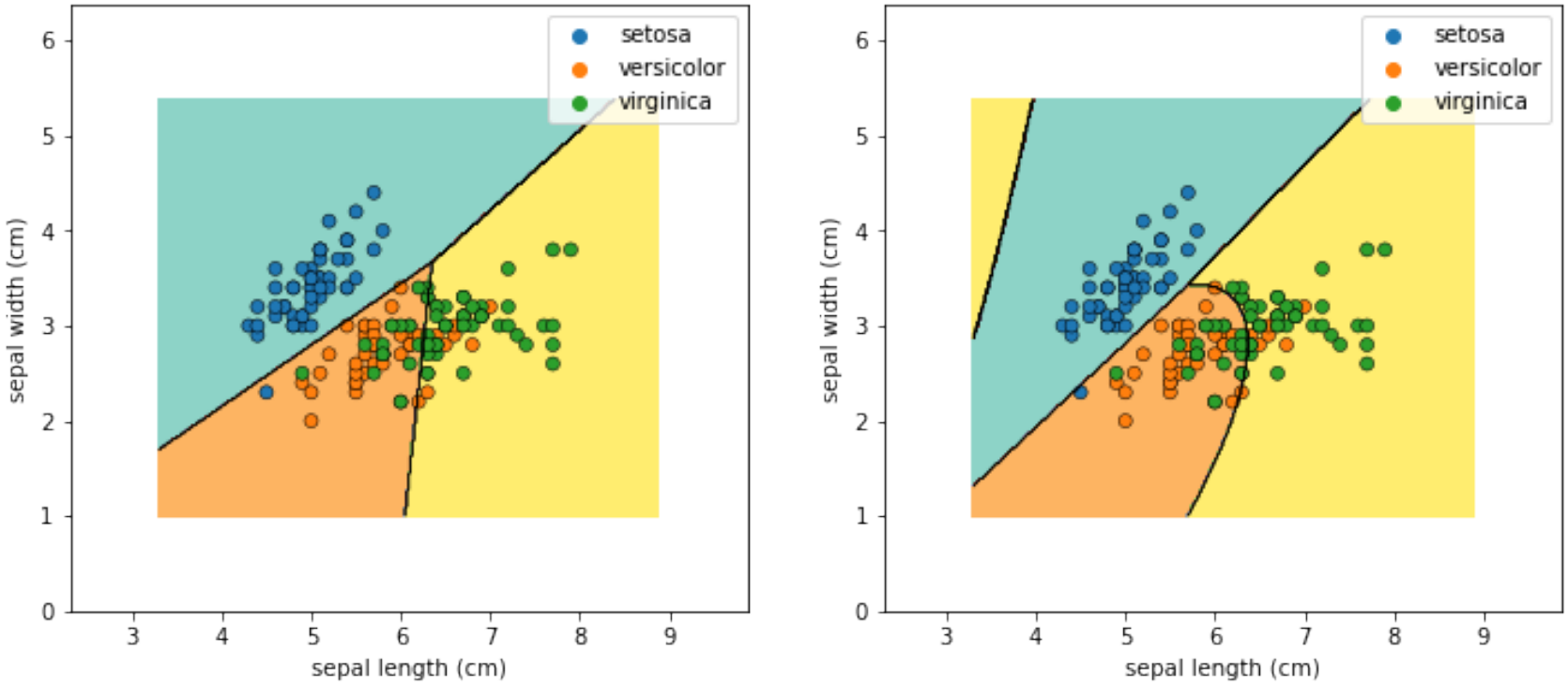
Here, $\alpha \in [0, 1]$ allows a continuum between QDA ($\alpha = 1$) and LDA ($\alpha = 0$)

Additionally, we may further regularize matrices by:

$$\hat{\Sigma}_k(\alpha, \gamma) = (1-\gamma)\hat{\Sigma}_k(\alpha) + \gamma\hat{\sigma}^2 I$$

where $\hat{\sigma}^2 = \frac{Tr[\hat{\Sigma}_k(\alpha)]}{d}$

Discriminant analysis, example with `iris` data



Naive Bayes

The Naive Bayes classifier is a Bayesian classifier that assumes that **features are pair-wise independent** in the class-conditional distribution:

$$p(\mathbf{x}|y) = \prod_{j=1}^{d} p(x_j|y)$$

This is in general not true, but it can be a good approximation for many cases. And it is certainly **practical** since it drastically reduces the amount of parameters we need to estimate.

To predict a class for input example $\mathbf{x}$ we choose as usual the one that maximizes:

$$g_k(\mathbf{x}) = \log \pi_k + \sum_{j=1}^{d} \log p(x_j|y = c_k)$$

So all we need to do is:

1. Estimate class priors as sample frequency ($\pi_k = \frac{n_k}{n}$)
2. Estimate class-conditional densities for each input feature independently

Categorical Naive Bayes

We can model **binary features** as Bernoulli distributions:

$$p(x|p) = p^x(1-p)^{1-x}$$

where $x \in \{0, 1\}$ and $p \in [0, 1]$ is the probability of the event happening.

For a binary feature, we need to estimate K parameters, one for each class, so:

$$p(x|y = c_k) = p_k^x(1-p_k)^{1-x}$$

Categorical Naive Bayes, cont.

If all our features are binary, then the discriminant functions become:

$$g_k(\mathbf{x}) = \log \pi_k + \sum_{j=1}^{d} \log p(x_j | y = c_k)$$

$$= \log \pi_k + \sum_{j=1}^{d} [x_j \log p_{kj} + (1 - x_j) \log(1 - p_{kj})]$$

where p_{kj} stands for the Bernoulli parameter for the j-th feature and k-th class. Notice this is a **linear function** of $\mathbf{x}$.

For categorical features with more than two values, the process is similar using a Categorical distribution:

$$g_k(\mathbf{x}) = \log \pi_k + \sum_{j=1}^{d} \sum_{v} [x_j = v] \log p_{kjv}$$

where $[exp]$ is 1 if its argument is true and 0 otherwise, and p_{kjv} is the Categorical parameter for value v for the j-th feature and k-th class.

Categorical Naive Bayes, cont.

To estimate these parameters, we use their sample frequencies in the data.

When data is scarce, 0-frequencies can be a problem, so **Laplace smoothing** is applied:

$$\hat{p}(v|y = c_k) = \frac{n_{kv} + p}{n_k + pV}$$

where $p \in \mathbb{R}^+$ is some "weight" assigned to the prior distribution of observing values v (typically is set to 1) and V is the number of modalities of the feature we are modelling. Here, n_k is the number of examples of class c_k in our training data and n_{kv} is the number of examples of class c_k that have v as their value.

Categorical Naive Bayes, example

Outlook	*Temperature*	*Humidity*	*Wind*	*Play ball*
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No

Use categorical Naive Bayes to predict the class for
$\mathbf{x}^* = (Sunny, Hot, Normal, Weak)^T$

(Answer should be "Yes" with probability 0.671)

Gaussian Naive Bayes

In case we have numerical features, it is common to assume them to be (univariate) Gaussian distributed and estimate their mean and variance using MLE.

If all features are assumed Gaussian, then Gaussian Naive Bayes is like QDA with diagonal covariance matrices.

Other common approaches to deal with numerical attributes would be:

- Discretize them and proceed as with categorical Naive Bayes, or
- Assume any other distribution and estimate its parameters from the training data

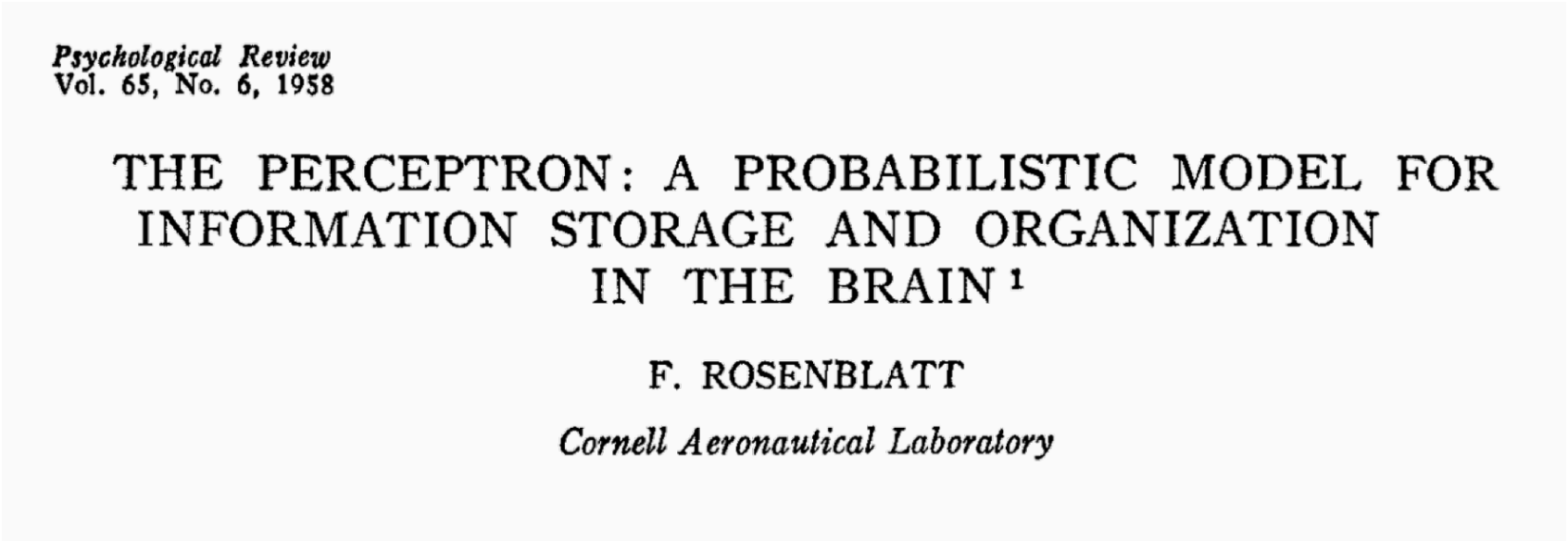
Note that if we have mixed type of features in our data, we can use a different distribution for each feature and then just add the log-likelihoods together.

Discriminative classifiers

Perceptron and Logistic regression

The good-old perceptron, an AI pioneer

A discriminative, non-probabilistic learning algorithm for linearly separable data



Psychological Review
Vol. 65, No. 6, 1958

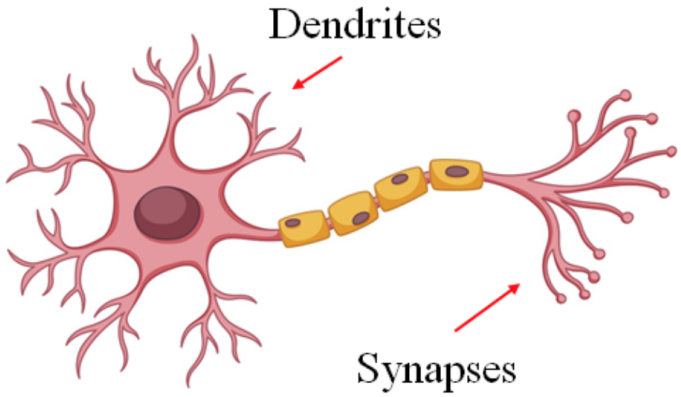
THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

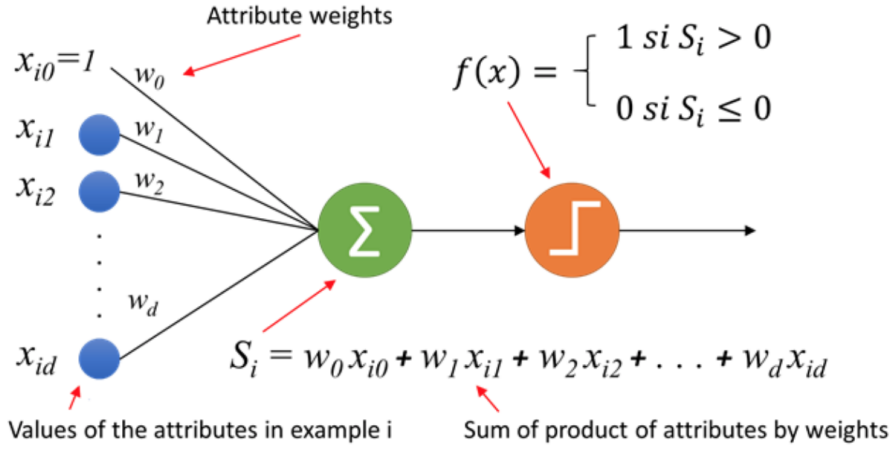
Cornell Aeronautical Laboratory

Figure 1: Rosenblatt's paper from 1958.

Perceptron model inspired by neurons



NEURON

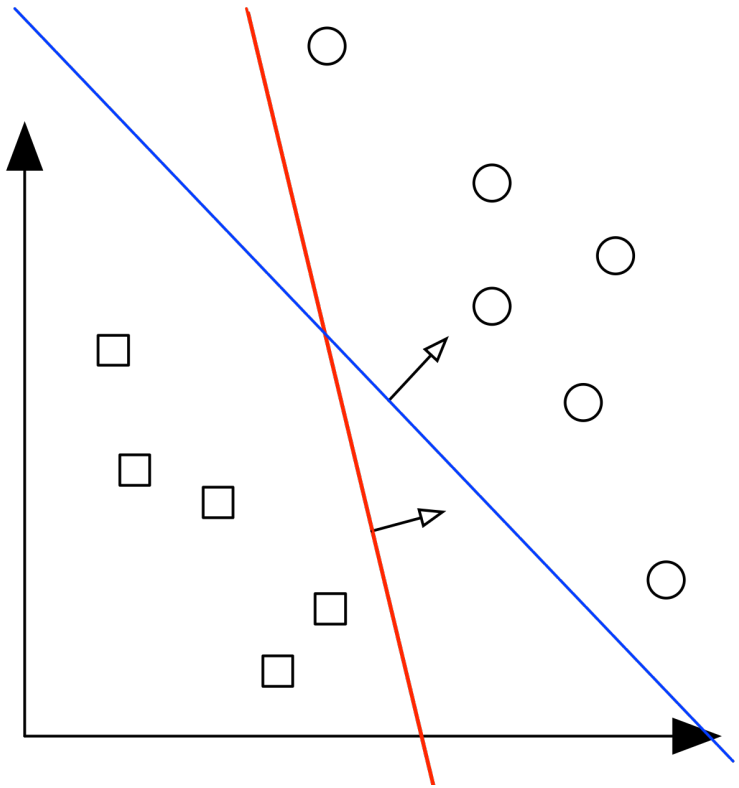


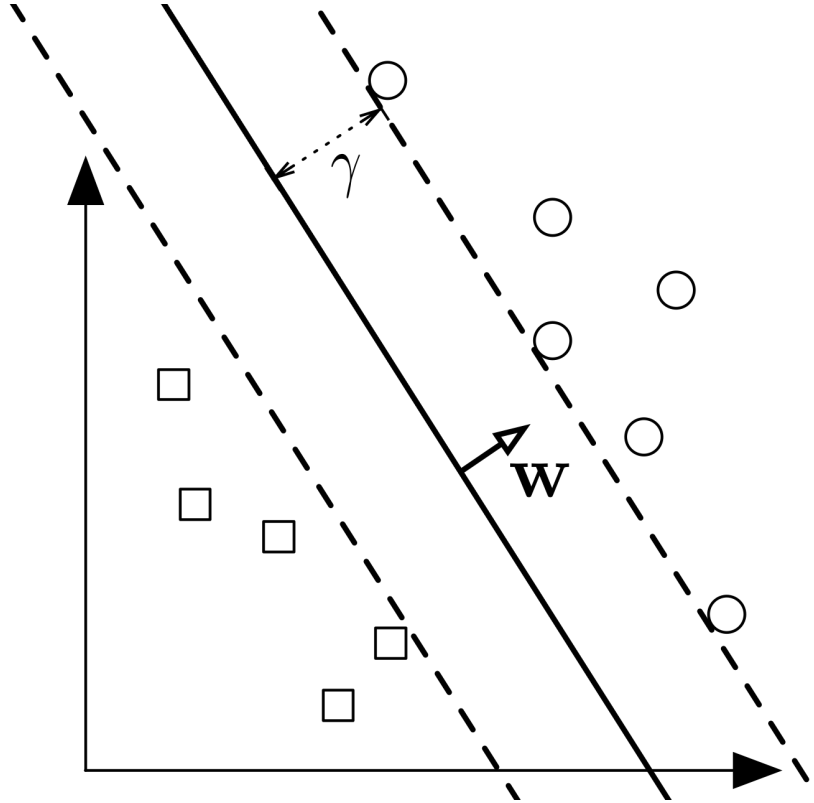
PERCEPTRON

$$y = sign(\mathbf{w}^T \mathbf{x}) = \begin{cases} +1 & \text{if } w_0 1 + w_1 x_1 + \cdots + w_d x_d > 0 \\ -1 & \text{otherwise} \end{cases}$$

The perceptron algorithm

Geometric interpretation of decision hyperplane





The perceptron algorithm

Finding optimal weights w

The *Perceptron* algorithm is an **on-line** algorithm.[2] Here, the training examples are *pairs* $(\mathbf{x}, y)$ with $\mathbf{x} \in \mathbb{R}^d$ and $y \in \{-1, 1\}$. Also, let's automatically scale all examples $\mathbf{x}$ to have (Euclidean) length 1, since this doesn't affect which side of the plane they are on.

1. Init weight vector $\mathbf{w} = 0$.
2. Given example $\mathbf{x}$, predict positive iff $\mathbf{w}^T\mathbf{x} > 0$.
3. On a mistake (i.e. $y \neq sign(\mathbf{w}^T\mathbf{x})$), update $\mathbf{w}$ as follows:
   - Mistake on positive: $\mathbf{w} = \mathbf{w} + \mathbf{x}$
   - Mistake on negative: $\mathbf{w} = \mathbf{w} - \mathbf{x}$

[2] which means that it receives one training example at a time, and updates its hypothesis (here, the weight vector $\mathbf{w}$) incrementally with each training example.

The perceptron algorithm, cont.

Let $\mathbf{w}$ be the value of the weight vector *before* and $\mathbf{w}'$ *after* the update.

- If a mistake on a positive $\mathbf{x}$ is made: $\mathbf{w'}^T\mathbf{x} = (\mathbf{w}+\mathbf{x})^T\mathbf{x} = \mathbf{w}^T\mathbf{x} + 1$.
- If a mistake on a negative $\mathbf{x}$ is made: $\mathbf{w'}^T\mathbf{x} = (\mathbf{w}-\mathbf{x})^T\mathbf{x} = \mathbf{w}^T\mathbf{x} - 1$.

So, in both cases we move closer (by 1) to the value we wanted.

Theorem. Let $\mathcal{D}$ be a sequence of labelled examples that are linearly separable.[3]. Assume that all $\mathbf{x}$ to have Euclidean length 1 and scale $\mathbf{w}$ to have length 1 as well. Then, the number of mistakes that Perceptron makes is at most $(\frac{1}{\gamma})^2$, where

$$\gamma = \min_{\mathbf{x}\in\mathcal{D}} \mathbf{w}^{*T}\mathbf{x}$$

The proof of the above theorem is described in many places; e.g. here

γ is often called the **margin** and it dictates how "easy" it is to separate both classes. More on this when we look at *Support Vector Machines*.

[3] Input data is linearly separable if there exists a weight vector $\mathbf{w}^*$ s.t. $\mathbf{w}^{*T}\mathbf{x} = y$ for all $(\mathbf{x}, y) \in \mathcal{D}$

The perceptron algorithm, example from Bishop's book

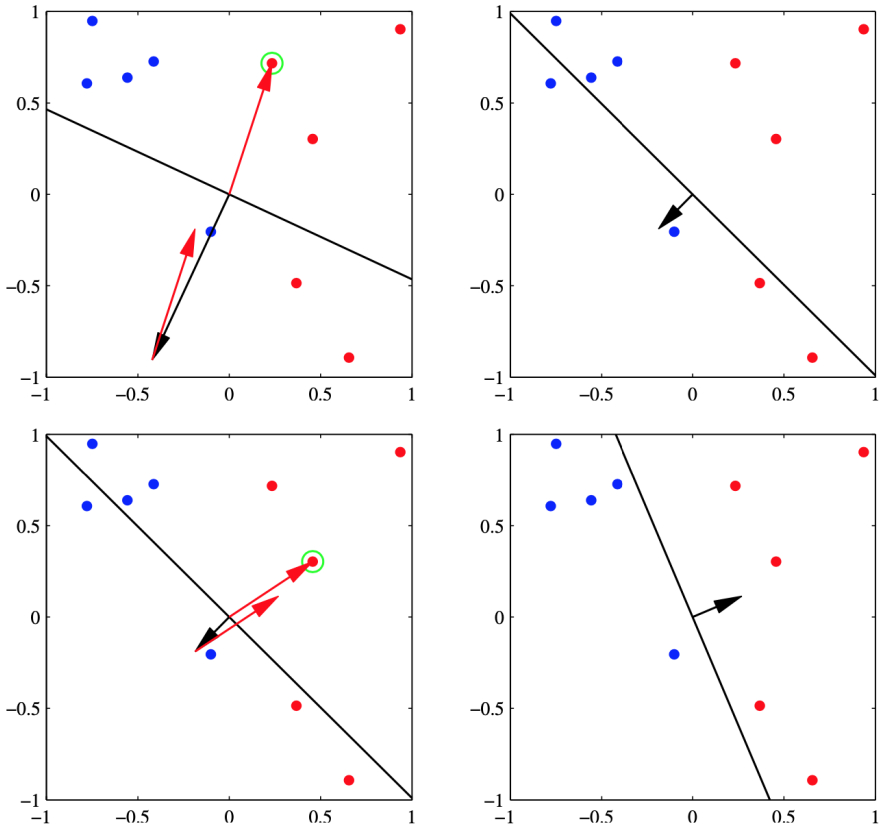


Figure 4.7 Illustration of the convergence of the perceptron learning algorithm, showing data points from two classes (red and blue) in a two-dimensional feature space (ϕ_1, ϕ_2). The top left plot shows the initial parameter vector $\mathbf{w}$ shown as a black arrow together with the corresponding decision boundary (black line), in which the arrow points towards the decision region which classified as belonging to the red class. The data point circled in green is misclassified and so its feature vector is added to the current weight vector, giving the new decision boundary shown in the top right plot. The bottom left plot shows the next misclassified point to be considered, indicated by the green circle, and its feature vector is again added to the weight vector giving the decision boundary shown in the bottom right plot for which all data points are correctly classified.

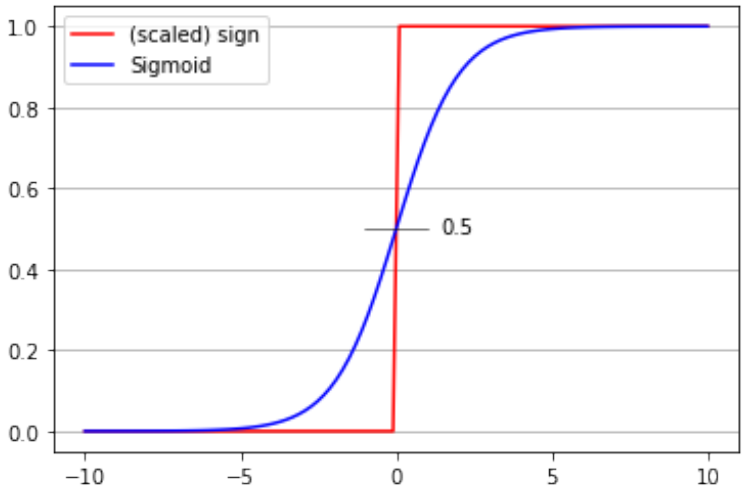
The Perceptron algorithm, remarks

What loss function is the Perceptron optimizing?

- **0-1 loss**, tries to fix *mistakes* (non-differentiable); there is no sense of big or small mistakes

The use of **sign()** in a prediction $sign(\mathbf{w}^T\mathbf{x})$ masks the magnitude of its input $\mathbf{w}^T\mathbf{x}$, which gives us a lot of information on "how far" $\mathbf{x}$ is from the separating hyperplane.

A better alternative is to use the **sigmoid** or **logistic** function, its "soft" version:

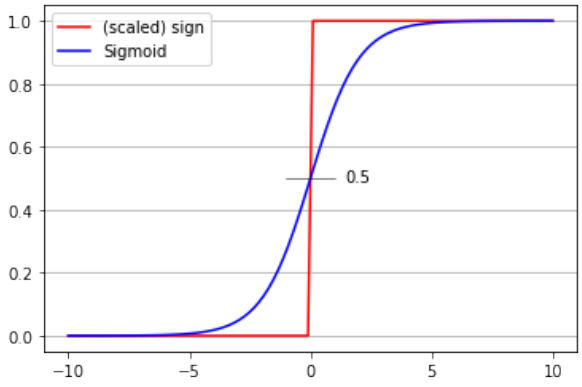


$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- maps $\mathbb{R}$ to $[0, 1]$ range so its output can be interpreted as a **probability**
- allows to reflect **uncertainty**

. . . which leads us to **logistic regression**

The logistic function



$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{e^z+1}$$

- has he following **symmetric** property:

$$\sigma(-z) = 1 - \sigma(z)$$

- it is **differentiable**:

$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$

- its inverse is the **logit function**

$$logit(p) = \log(\frac{p}{1-p})$$

Logistic regression

Let us start with the case of **binary classification**. Here, we are given labelled examples $(\mathbf{x}_i, y_i)$ such that $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{0, 1\}$. We associate the label $y_i = 1$ with a *positive example*. In *logistic regression* we model:

$$P(y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x})$$

where $\mathbf{w}$ are the weights (or coefficients) of a linear combination on *features* in $\mathbf{x}$.

Therefore, each $y_i \sim Ber(p_i)$ where $p_i = \sigma(\mathbf{w}^T \mathbf{x}_i)$.

Notice that **there is no assumption on how x is distributed**.

Bernoulli distribution

A binary random variable $Y \in \{0, 1\}$ is distributed according to a Bernoulli distribution with parameter p, then:

- $P(Y = 1; p) = p$ and $P(Y = 0; p) = 1 - p$

So, its *probability mass function* is:

$$P(y; p) = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{if } y = 0 \end{cases}$$

which may be more compactly written as

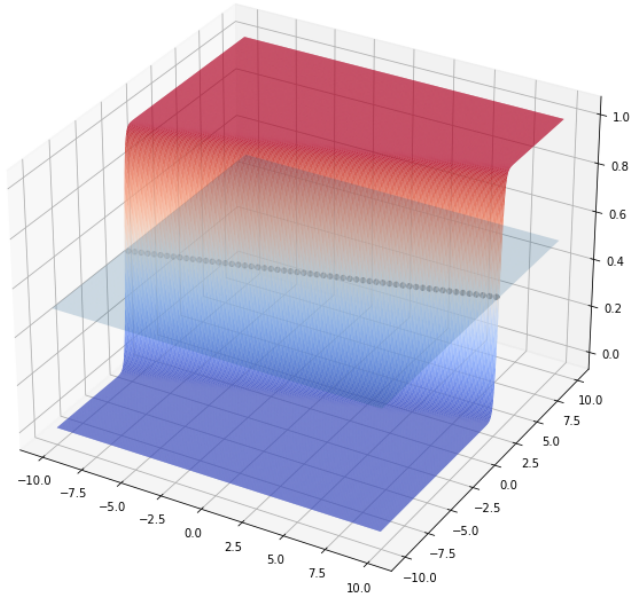
$$P(y; p) = p^y (1 - p)^{1-y}$$

Logistic regression, cont.

Why is logistic regression a linear classifier?

Well, we have determined that predictions are computed as $\hat{y} = \sigma(\mathbf{w}^T\mathbf{x})$, so $0 \leq \hat{y} \leq 1$ is taken as the probability of $\mathbf{x}$ being positive. A natural threshold value to use for hard class predictions is 0.5 however we may use any other fixed value if we want to be more or less conservative.

So, the prediction for $\mathbf{x}$ is positive iff $\sigma(\mathbf{w}^T\mathbf{x}) > 0.5$ iff $\mathbf{w}^T\mathbf{x} > 0$ which corresponds to a linear class boundary.



Logistic regression, cont.

Given an input dataset $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ with each $(\mathbf{x}_i, y_i)$ iid according to a Bernoulli distribution $y_i \sim Ber(\sigma(\mathbf{w}^T \mathbf{x}_i))$, we can use maximum likelihood to guide the search for suitable values for $\mathbf{w}$; in the following derivation our predictions are $\hat{y}_i = \sigma(\mathbf{w}^T \mathbf{x}_i)$.

$$
\begin{aligned}
\mathcal{L}(\mathbf{w}) &= \prod_{i=1}^{n} P(y_i | \mathbf{x}_i, \mathbf{w}) \\
&= \prod_{i=1}^{n} Ber(y_i | \hat{y}_i = \sigma(\mathbf{w}^T \mathbf{x}_i)) \\
&= \prod_{i=1}^{n} \hat{y}_i^{y_i} (1 - \hat{y}_i)^{(1-y_i)} \\
&= \prod_{i=1}^{n} \sigma(\mathbf{w}^T \mathbf{x}_i)^{y_i} (1 - \sigma(\mathbf{w}^T \mathbf{x}_i))^{(1-y_i)}
\end{aligned}
$$

Logistic regression, cont.

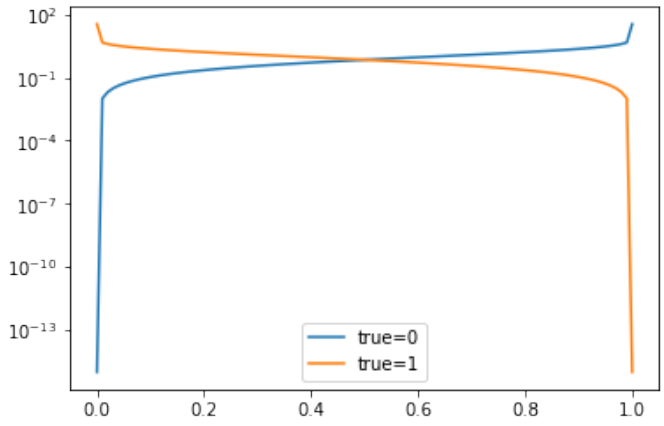
As usual, we maximize the *log-likelihood* although in this case it is more common to **minimize the negative log-likelihood**:

$$\begin{aligned} E(\mathbf{w}) := -\log \mathcal{L}(\mathbf{w}) &= -\sum_{i=1}^{n} \log P(y_i|\mathbf{x}_i, \mathbf{w}) \\ &= -\sum_{i=1}^{n} y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \end{aligned}$$

This last expression is known as **log loss** or **binary cross-entropy**, and it is a very common error measure used in classification.

Log-loss

Assuming Bernoulli targets in classification leads to log loss, a metric which makes a lot of sense; the more similar $\hat{y}$ is to y is, the closer it gets to 0:



$\hat{y}$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
y = 0	0.11	0.22	0.36	0.51	0.69	0.92	1.20	1.61	2.30
y = 1	2.30	1.61	1.20	0.92	0.69	0.51	0.36	0.22	0.11

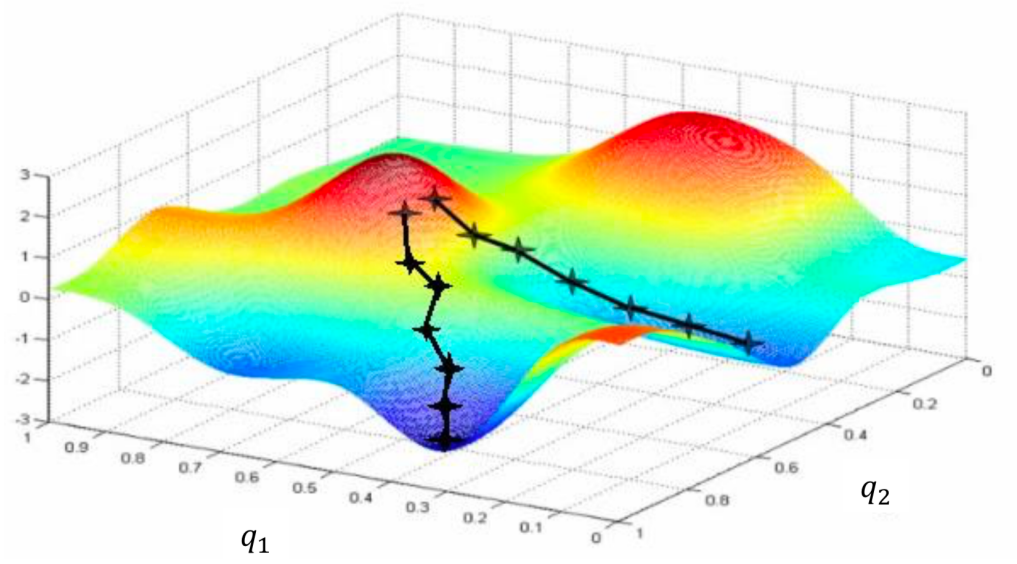
Logistic regression, cont.

Let us find the **gradient** of $E(\mathbf{w})$ w.r.t. $\mathbf{w}$:

$$
\begin{aligned}
\nabla E(\mathbf{w}) &= \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \\
&= -\sum_{i=1}^{n} \left[\frac{\partial (y_i \log \hat{y}_i)}{\partial \mathbf{w}} + \frac{\partial \left((1-y_i) \log(1-\hat{y}_i) \right)}{\partial \mathbf{w}} \right] \\
&= -\sum_{i=1}^{n} \left[\frac{\partial y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i)}{\partial \mathbf{w}} + \frac{(1-y_i) \log(1-\sigma(\mathbf{w}^T \mathbf{x}_i))}{\partial \mathbf{w}} \right] \\
&= \dots \\
&= \sum_i (y_i - \sigma(\mathbf{w}^T \mathbf{x}_i)) \mathbf{x}_i \\
&= \sum_i (y_i - \hat{y}_i) \mathbf{x}_i
\end{aligned}
$$

This time it is not possible to find a closed-form solution, and so we resort to iterative methods to finding a minimum of this function. The simplest is **gradient descent** which starts with random values for $\mathbf{w}$ and updates them following the gradient. We are in **optimization** land now.

Gradient descent



Gradient descent is a general optimization method for finding (local) minima of a **differentiable** function $E(\cdot)$. The idea is that if we follow the direction of the **negative gradient**, E will decrease the *fastest* (at least locally).

In general, we start with a random input $\mathbf{w}_0$, and using the update rule

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \gamma_n \nabla E(\mathbf{w}_k)$$

we get a sequence $\mathbf{w}_0, \mathbf{w}_1, \ldots$ such that $E(\mathbf{w}_0) \geq E(\mathbf{w}_1) \geq \ldots$ for small enough **learning rates** γ_n.

Gradient descent, general considerations

- often we use a single γ throughout the whole execution; it is not straightforward to choose it appropriately
  - if γ is too small then convergence may be **slow**
  - if γ is too large then we may not converge
- lots of learning rate strategies and heuristic exist (beyond scope of this course)
- **feature scaling** is important because the same learning rate is used for all features

Newton's algorithm, getting rid of learning rate

If we can afford to compute second derivatives (i.e. the *Hessian*) or $\mathbf{H}$:

$$\mathbf{H}(\mathbf{w}) := \nabla^2 E(\mathbf{w}) = \begin{pmatrix} \frac{\partial^2 E(\mathbf{w})}{\partial w_1^2} & \frac{\partial^2 E(\mathbf{w})}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_1 \partial w_d} \\ \frac{\partial^2 E(\mathbf{w})}{\partial w_2 \partial w_1} & \frac{\partial^2 E(\mathbf{w})}{\partial w_2^2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_2 \partial w_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E(\mathbf{w})}{\partial w_d \partial w_1} & \frac{\partial^2 E(\mathbf{w})}{\partial w_d \partial w_2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_d^2} \end{pmatrix}$$

Then we can use *Newton's algorithm* which is an algorithm for finding roots that we apply to the gradient. In our case, the updates are

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mathbf{H}(\mathbf{w}_k)^{-1} \nabla E(\mathbf{w}_k)$$

This update rule is derived by finding the minimum of the second-order (quadratic) Taylor series approximation for E around $\mathbf{w}_k$.

Notice **there is no learning rate** involved, however, at the expense of having to compute second-order derivatives (Hessian).

Back to logistic regression, Newton's algorithm: IRLS

The gradient of the negative log-likelihood for logistic regression is:

$$\nabla E(\mathbf{w}) = \sum_i (y_i - \hat{y}_i)\mathbf{x}_i = X^T(\mathbf{y} - \sigma(X\mathbf{w}))$$

The Hessian is:

$$\mathbf{H}(\mathbf{w}) = \sum_i \hat{y}_i(1 - \hat{y}_i)\mathbf{x}_i\mathbf{x}_i^T = X^T diag(\hat{y}_i(1 - \hat{y}_i))X$$

which leads to the **Iterated Reweighted Least Squares** (IRLS) algorithm.

Logistic regression, multi-class case (K>2)

This case is handled by having one separator $\mathbf{w}^{(k)}$ for each class $1 \leq k \leq K$. In this case, instead of a Bernoulli distribution for the targets y_i one uses a **Categorical** distribution. Each target y_i is represented with one-hot encoding $y_i = [0, \ldots, 0, 1, 0, \ldots, 0]$.

The likelihood in this setting is

$$\mathcal{L}(\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(K)}) = \prod_i \prod_k \hat{y}_{ik}^{y_{ik}}$$

and the negative log-likelihood is

$$E(\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(K)}) = -\sum_i \sum_k y_{ik} \log \hat{y}_{ik}$$

which is the **cross-entropy loss**.

To optimize this loss we do the same as in the binary case (use gradient descent or Newton's method).

Regularization

We can add $L1$ (lasso) or $L2$ (ridge) regularization on the *weights* to the cross-entropy loss to obtain regularized versions of logistic regression.

$$E_{lasso}(\mathbf{w}) = -\sum_i [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)] + \lambda \sum_{k=1}^{d} |w_k|$$

$$E_{ridge}(\mathbf{w}) = -\sum_i [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)] + \lambda \sum_{k=1}^{d} w_k^2$$