POLIMI
DATA SCIENTISTS

Machine Learning

Course Notes

Edited by:

Marco Varrone

Credits

The following notes have been written by the Polimi Data Scientists student association by combining Prof. Restelli's lectures and slides with content from the following books:

- Bishop, *"Pattern Recognition and Machine Learning"*, Springer, 2006
- Sutton and Barto, *"Reinforcement Learning: an Introduction"*, MIT Press, 1998
- Mitchell, *"Machine Learning"*, McGraw Hill, 1997

Contents

Chapter 1

Introduction: Machine Learning Models

1.1 Supervised Learning

- Goal
  - Estimating the unknown model that maps known inputs to known outputs
  - Training set: $\mathcal{D} = \{\langle x, t \rangle\} \Rightarrow t = f(x)$
- Problems
  - Classification
  - Regression
  - Probability estimation
- Techniques
  - Artificial Neural Networks
  - Support Vector Machines
  - Decision trees
  - ...

1.2 Unsupervised Learning

- Goal
  - Learning a more efficient representation of a set on unknown inputs
  - Training set: $\mathcal{D} = \{x\} \Rightarrow ? = f(x)$
- Problems
  - Compression
  - Clustering
- Techniques
  - K-means
  - Self-organizing maps
  - Principal Component Analysis
  - ...

1.3 Reinforcement Learning

- Goal
  - Learning the optimal policy
  - Training set: $\mathcal{D} = \{\langle x, u, x', r\rangle\} \Rightarrow \pi * (x) = argmax_u\{Q * (x, u)\}$ where *Q*(x,u)* must be estimated
- Problems
  - Markov Decision Process (MDP)
  - Partially Observable MDP (POMDP)
  - Stochastic Games (SG)
- Techniques
  - Q-learning
  - SARSA
  - Fitted Q-iteration
  - ...

Part I

Supervised Learning

Chapter 2

Introduction

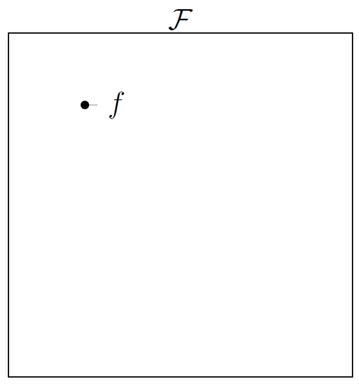
Supervised learning is the largest, most mature and most widely used sub-field of machine learning.
Given training data set including desired outputs $\mathcal{D} = \{\langle x, t \rangle\}$ from some unknown function f.
Find a good approximation of f that generalizes well on test data.
Input variables x are also called features or attributes.
Output variables t are also called targets or labels.

- If t is discrete: classification
- if t is continuous: regression
- if t is the probability of x: probability estimation, which is different from regression because of the constraints imposed by probability.

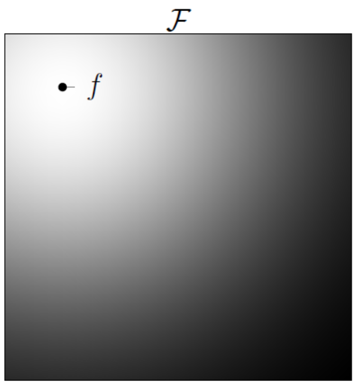
Supervised learning is based on an ill-posed problem, because given the training set we are not interested on the performance on the training set, but we have to optimize and objective function which is not known.

2.1 Overview of Supervised Learning

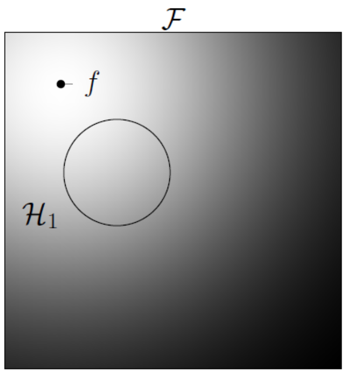
We want to approximate f given the dataset $\mathcal{D}$
The steps are:



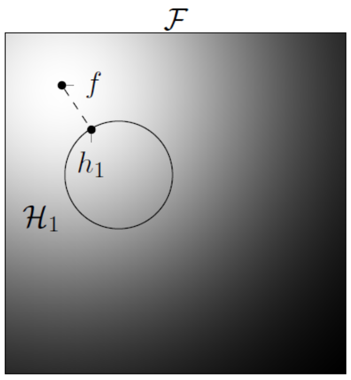
1. Define a **loss function** $\boldsymbol{L}$, which is a metric of how much a f is far from the target



2. Choose some **hypothesis space** $\mathcal{H}$, which is the subspace of all the possible functions that we are interested in.

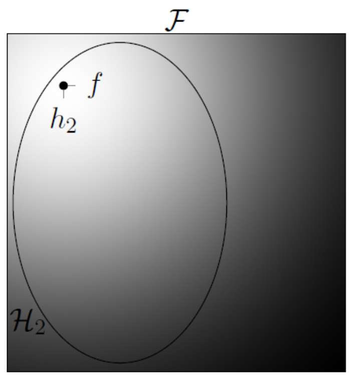


3. Optimize to find and **approximate model** $\boldsymbol{h}$ that minimizes the loss with respect to the target function.

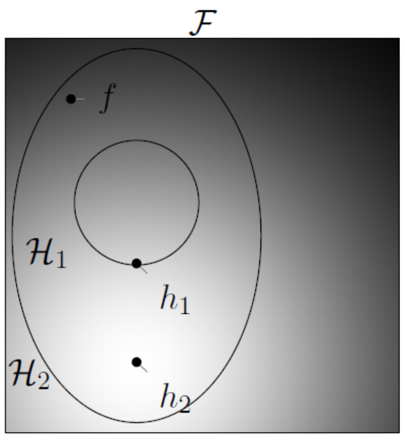


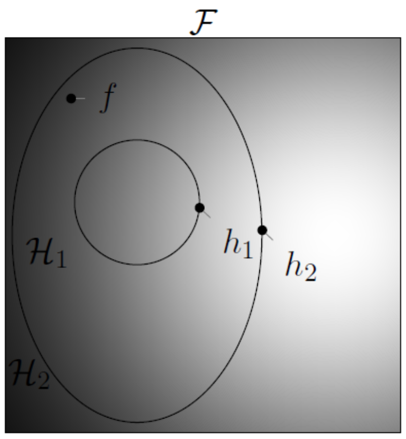
What happens if we enlarge the hypothesis space?

The loss of the best hypothesis will not increase and probably decrease.
So the hypothesis space can be enlarged until the true model is inside of it, obtaining a loss value equal to zero.



If we don't know f but we only have a few samples taken from it, the loss function is based not based on the function anymore, but on the samples.
So, the optimal function h_2 based on the new loss function may be far from the true model and be farther as long as we enlarge the hypothesis space.
If there are few samples, the loss function may be very noisy, and the more we enlarge the hypothesis space, the more we are sensitive to the noise.
With infinite samples we can enlarge the hypothesis space without problems, because it's like knowing f (no uncertainty).





Chapter 3

Linear Regression

3.1 Linear Regression

The goal of regression is to learn a mapping from D-dimensional vector x of input variables to a continuous output t. The simplest form of linear regression models are linear functions of the input variables.

$$y(\mathbf{b}, \mathbf{w}) = w_0 + \sum_{j=1}^{D-1} w_j x_j = \mathbf{w}^T \mathbf{x}$$

where $\mathbf{x} = (1, x_1, ..., x_{D-1})$ and w_0 is called offset or **bias** parameter.

3.1.1 The loss function

The **loss** (error) function is a measure to quantify the performance on a task $L(t, y(\mathbf{x}))$.
The **average**, or expected, loss is given by

$$\mathbb{E}[L] = \int\int L(t, y(\mathbf{x}))p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t$$

A common choice is the **squared loss function**

$$\mathbb{E}[L] = \int\int (t - y(\mathbf{x}))^2 p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t$$

*p(**x**,t)* is the probability distribution under which we want to optimize the model. It's the distribution over the future (test) samples which is usually equal to the distribution of the train samples. So, if *p(x,t)* is known, also the distribution of the true model is known.
Our goal is to choose y($\mathbf{x}$) so as to minimize $\mathbb{E}[L]$. Hence, the optimal solution is the conditional average

$$y(\mathbf{x}) = \int tp(t|\mathbf{x})\mathrm{d}t = \mathbb{E}[t|\mathbf{x}]$$

The squared loss is not the only possible choice of loss function for regression. An example is the simple generalization of the squared loss, called **Minkowski** loss

$$\mathbb{E}[L] = \int\int (t - y(\mathbf{x}))^q p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t$$

The minimum of $\mathbb{E}[L]$ is given by:

- the conditional mean for $q = 2$
- the conditional median[1] for $q = 1$
- the conditional mode[2] for $q \to 0$

Note that in a Gaussian distribution mean=median=mode.

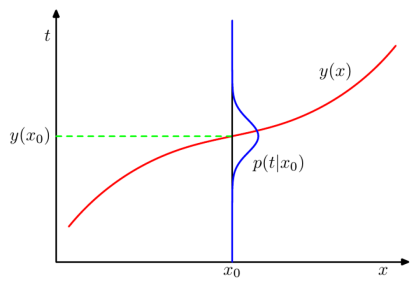


Figure 3.1: *y(x)* is given by the mean of the conditional distribution *p(t—x)*

[1]The value separating the higher half of the samples from the lower half
[2]The most frequent sample

3.2 Basis functions

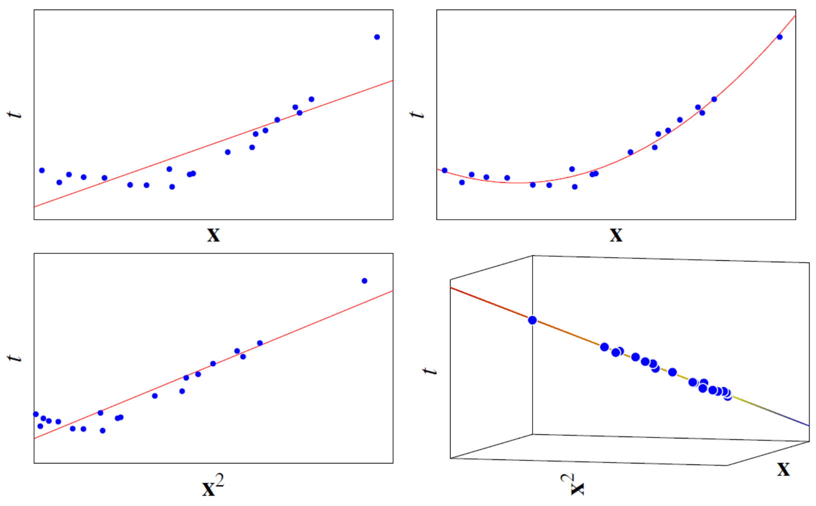
We extend the class of models by considering linear combinations of fixed nonlinear functions (**basis functions**) of the input variables

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x})$$

where $\phi(\mathbf{x}) = (1, \phi_1(\mathbf{x}), ..., \phi_{M-1}(\mathbf{x}))^T$.
There are multiple examples of basis functions:

- Polynomial $\phi_j(x) = x^j$.
  One limitation is that they are global functions of the input variable, so that changes in one region of input space affect all other regions.
- Gaussian $\phi_j(x) = exp\left(-\frac{(x-\mu_j)^2}{2\sigma^2}\right)$.
  They are not required to have a probabilistic interpretation, and in particular the normalization coefficient is unimportant because these basis functions will be multiplied by w_j.
- Sigmoidal $\phi_j(x) = \frac{1}{1+exp\left(\frac{\mu_j - x}{\sigma}\right)}$

If there is no linear relationship between the target and the input, it is possible to change the space through the basis function to obtain a linear model that can learn nonlinear functions of the input variable.



In this way the model is linear in the new feature space but nonlinear in the original space.

3.2.1 Discriminative vs Generative

- **Direct approach**
  - Find a regression function y($\mathbf{x}$) directly from the training data by searching in the space of the model (changing the parameters)
- **Discriminative approach**
  - Model directly the conditional density $p(t|\mathbf{x})$
  - Marginalize to find the conditional mean $\mathbb{E}[t|\mathbf{x}] = \int tp(t|\mathbf{x})\mathrm{d}t$
- **Generative approach**: useful for augmenting data because it can generate new samples (e.g. new input) and then ask someone to label new data
  - Model the joint density $p(\mathbf{x}, t) = p(\mathbf{x}|t)p(t)$
  - Infer conditional density $p(t|\mathbf{x}) = \frac{p(\mathbf{x},t)}{p(\mathbf{x})}$
  - Marginalize to find the conditional mean $\mathbb{E}[t|\mathbf{x}] = \int tp(t|\mathbf{x})\mathrm{d}t$

3.3 Direct approaches

3.3.1 Minimizing Least Squares

Given a data set with N samples, let us consider the following loss function

$$L(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(x_n, \mathbf{w}) - t_n)^2$$

This is (half) the **residual sum of squares** (**RSS**), also known as sum of squares error (**SSE**). It can also be written as the sum of the ℓ_2-norm[3] of the vector of the residual errors

$$RSS(\mathbf{w}) = ||\epsilon||_2^2 = \sum^{N} \epsilon_i^2$$

Let's write RSS in matrix form with $\Phi = (\phi(\mathbf{x}_1), ..., \phi(\mathbf{x}_N))^T$ and $\mathbf{t} = (t_1, ..., t_N)^T$
The residual $\epsilon = \mathbf{t} - \Phi\mathbf{w}$, of size $N \times 1 - (N \times M)(M \times 1) = N \times 1$
Given that $||\epsilon||_2^2 = \sum_i \epsilon_i^2 = \epsilon^T\epsilon$

$$L(\mathbf{w}) = \frac{1}{2}RSS(\mathbf{w}) = \frac{1}{2}(\mathbf{t} - \Phi\mathbf{w})^T(\mathbf{t} - \Phi\mathbf{w})$$

To obtain the minimum we need the point with:

- gradient $= 0$
- curvature has all eigenvalues > 0

We compute the first and second derivative

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = -\Phi^T(\mathbf{t} - \Phi\mathbf{w}); \quad \frac{\partial^2 L(\mathbf{w})}{\partial \mathbf{w}\partial \mathbf{w}^T} = \Phi^T\Phi$$

We can observe that, assuming $\Phi^T\Phi$ nonsingular, it is symmetric and positive semi-definite, so all the eigenvalues are ≥ 0. If some eigenvalues are $= 0$ we have infinite solutions.

$$-\Phi^T(\mathbf{t} - \Phi\mathbf{w}) = 0 \Rightarrow (\Phi^T\Phi)^{-1}\Phi^T\Phi\mathbf{w} = (\Phi^T\Phi)^{-1}\Phi^T t \Rightarrow \Phi^T\Phi w = \Phi^T t$$

And we obtain

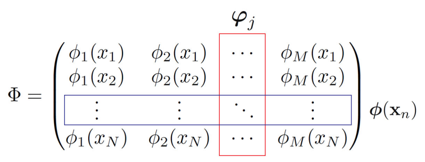
$$\hat{\mathbf{w}}_{OLS} = (\Phi^T\Phi)^{-1}\Phi^T t$$

It is not possible to perform the computation in two cases:

- we have less samples than features, so $\Phi^T\Phi$ is not with full rank, so we have infinite solutions
- some features are linear combination of others, so we have a singular matrix

3.3.2 Geometric interpretation

We consider an N-dimensional space whose axes are given by the t_n, so that $\mathbf{t} = (t_1, ..., t_N)$ is a vector in this space. Each basis function $\phi(x_n)$ can be also represented as a vector in this space.
Note that as we can see from Figure 3.2, φ_j corresponds to the j^{th} column of Φ, whereas $\phi(x_n)$ corresponds to the n^{th} row of Φ.
If the number M of basis functions is smaller than N, than the M vectors $\phi(x_n)$ will span a linear subspace $\mathcal{S}$ of dimensionality M.
Define $\hat{\mathbf{t}}$ as the N-dimensional vector whose n^{th} element is given by $y(x_n, \mathbf{w})$. Because $\hat{\mathbf{t}}$ is an arbitrary linear combination $\varphi_1, ..., \varphi_M$, it lies in an M-dimensional subspace $\mathcal{S}$.


φ_j
$\Phi =$
$\phi_1(x_1)$ $\phi_2(x_1)$ $\cdots$ $\phi_M(x_1)$
$\phi_1(x_2)$ $\phi_2(x_2)$ $\cdots$ $\phi_M(x_2)$
$\vdots$ $\vdots$ $\ddots$ $\vdots$
$\phi_1(x_N)$ $\phi_2(x_N)$ $\cdots$ $\phi_M(x_N)$
$\phi(\mathbf{x}_n)$
Figure 3.2: Graphical representation of $\phi_(x_n)$ and φ_j

The residual sum of squares is then equal to the squared Euclidean distance between $\hat{\mathbf{t}}$ and $\mathbf{t}$. Thus the least-squares solution for $\mathbf{w}$ corresponds to that choice of $\hat{\mathbf{t}}$ that lies in the subspace $\mathcal{S}$ and that is closest to $\mathbf{t}$. This solution corresponds to the orthogonal projection of $\mathbf{t}$ onto the subspace $\mathcal{S}$.

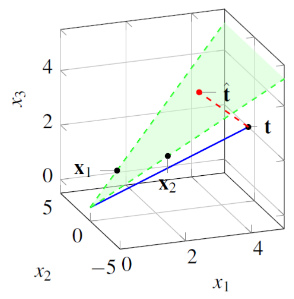
$$\hat{\mathbf{t}} = \Phi\hat{\mathbf{w}} = \Phi(\Phi^T\Phi)^{-1}\Phi^T\mathbf{t}$$

where $H = \Phi(\Phi^T\Phi)^{-1}\Phi^T$ is called hat matrix.

[3] $||x||_p = \left(\sum_{i\in\mathbb{N}} |x_i|^p\right)^{1/p}$

Assume $N = 3$ and $M = D = 2$

$$\mathbf{\Phi} = \mathbf{X} = \begin{pmatrix} 1 & 2 \\ 1 & -2 \\ 1 & 2 \end{pmatrix} \quad \mathbf{t} = \begin{pmatrix} 5 \\ 1 \\ 2 \end{pmatrix} \quad \hat{\mathbf{t}} = \begin{pmatrix} 3.5 \\ 1 \\ 3.5 \end{pmatrix}$$



3.3.3 Gradient Optimization

A closed-form solution can be impractical with large data sets. It may be worthwhile to use a sequential (**online**) algorithm, in which the gradient is computed one sample at a time. This algorithm can be applied also when data observations are arriving in a continuous stream, and predictions must be made before all of the data points are seen.
For this purpose we can apply the **stochastic gradient descent**.
If the loss function can be expressed as a sum over samples ($L(\mathbf{x}) = \sum_n L(x_n)$) the algorithm updates $\mathbf{w}$ as follows:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha^{(k)} \nabla L(X_n)$$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha^{(k)} (\mathbf{w}^{(k)^T} \phi(\mathbf{x}_n) - t_n) \phi(\mathbf{x}_n)$$

where k is the iteration and α is the **learning rate**.
For convergence the learning rate has to satisfy:

$$\sum_{k=0}^{\infty} \frac{1}{\alpha^{(k)}} = +\infty \qquad \text{(learning converges quickly enough)}$$

$$\sum_{k=0}^{\infty} \frac{1}{\alpha^{(k)2}} < +\infty \qquad \text{(learning converges not too fast)}$$

A possible example of learning rate is $\frac{1}{k}$.

3.4 Discriminative approaches

3.4.1 Maximum Likelihood (ML)

We assume that the target variable t is given by a deterministic function y with a **Gaussian noise** $\epsilon \sim \mathcal{N}(0, \sigma^2)$:

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon$$

So, $t \sim \mathcal{N}(y(\mathbf{x}, \mathbf{w}), \sigma^2)$.
Given N samples independent and identically distributed (i.i.d), with inputs $\mathbf{X} = \mathbf{x}_1, ..., \mathbf{x}_N$ and outputs $\mathbf{t} = (t_1, ..., t_N)^T$, the likelihood function is

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}(t_n|\mathbf{w}^T \phi(\mathbf{x}_n), \sigma^2) =$$

$$= \prod_{n=1}^{N} \frac{1}{2\pi\sigma^2} e^{-\frac{(t - y(\mathbf{x}, \mathbf{w}))^2}{2\sigma^2}}$$

Now we need to use $\mathbf{w}$ to approximate the mean of the Gaussian, because as seen in section 3.1.1, with a squared loss function, the optimal prediction of a new value $\mathbf{x}$ is equal to the conditional mean of the

target variable. This can be done by finding the **maximum likelihood**, which is the probability of the target values given the assumed model.
We can consider the **log-likelihood function**, which has the maximum value at the same point.

$$\begin{aligned}\ell(\mathbf{w}) = \ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) &= \sum_{n=1}^{N} \ln p(t_n|\mathbf{x}_n, \mathbf{w}, \sigma^2) \\ &= -\frac{N}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}RSS(\mathbf{w}) = -\frac{N}{2}\ln(2\pi\sigma^2) - \frac{1}{4\sigma^2}(\mathbf{t} - \mathbf{w}\Phi)^T(\mathbf{t} - \mathbf{w}\Phi)\end{aligned}$$

Note that the first term does not depend on $\mathbf{w}$, so it will disappear when we compute the gradient; the second term is basically the loss function, confirming that maximizing the likelihood is equivalent to minimizing the loss function.

$$\nabla \ln \ell(\mathbf{W}) = -\Phi^T(\mathbf{t} - \Phi\mathbf{w}) = -\Phi^T\mathbf{t} + \Phi^T\Phi\mathbf{w} = 0$$
$$\mathbf{w}_{ML} = (\Phi^T\Phi)^{-1}\Phi^T\mathbf{t}$$

Where $(\Phi^T\Phi)^{-1}\Phi^T$ is a generalization of the inverse of a matrix for non-squared matrices (*Moore-Penrose pseudo-inverse*).

3.4.2 Variance of the parameters

We assume that:

- the observations t_i are uncorrelated and have constant variance σ^2
- the x_i are fixed (non random)

The variance-covariance matrix[4] of the least-squares estimate is:

$$Var(\hat{\mathbf{w}}_{OLS}) = (\Phi^T\Phi)^{-1}\sigma^2$$

Usually, the variance σ^2 is estimated by

$$\hat{\sigma}^2 = \frac{1}{N-M-1}\sum_{n=1}^{N}(t_n - \hat{\mathbf{w}}^T\phi(\mathbf{x}_n))^2$$

Assuming that the model is linear in the features $\phi_1(), ..., \phi_M()$ and that the noise is additive and Gaussian

$$\hat{\mathbf{w}} \sim \mathcal{N}(\mathbf{w}, (\Phi^T\Phi)^{-1}\sigma^2) \qquad (N-M-1)\hat{\sigma}^2 \sim \sigma^2\chi^2_{N-M-1}$$

Such properties can be used to form **test hypothesis** and **confidence intervals**.

Theorem (Gauss-Markov)
*The least squares estimate of **w** has the **smallest variance** among all linear **unbiased** estimates.*

It follows that the least squares estimator has the lowest MSE[5] of all linear estimators with no bias (in which expected value = true value)
However, there may exist a biased estimator with smaller MSE.

3.4.3 Multiple Outputs

If we want to predict $K > 1$ target variables, denoted collectively by the target vector $\mathbf{t}$, we could introduce a different set of basis functions for each component of $\mathbf{t}$, leading to multiple, independent regression problems.
Usually, a single set of basis functions is considered:

$$\underbrace{\hat{\mathbf{W}}_{ML}}_{M\times K} = (\Phi^t\Phi)^{-1}\Phi^T \underbrace{\mathbf{T}}_{N\times K}$$

For each output t_k we have

$$\hat{\mathbf{w}} = (\Phi^T\Phi)^{-1}\Phi^T\mathbf{t}_k$$

where $\mathbf{t}_k$ is an N-dimensional column vector. Thus the solution decouples between the different outputs, and we need only to compute a single pseudo-inverse matrix, which is shared by all of the vectors $\mathbf{w}_k$.

[4]It has variance on the diagonal and covariance on the rest of the matrix
[5]Mean Squared Error: $\frac{\sum_n (y(x_n, \mathbf{w}) - t_n)^2}{N}$

3.5 Regularization

3.5.1 Under-fitting vs Over-fitting

A model with low complexity is usually not capable of fitting the data and so representing appropriately the true model (**under-fitting**).
A model with high complexity (e.g. high-order polynomials) gives excellent fit over the training data, but a poor representation of the true function (**over-fitting**).
The goal is to achieve a good generalization and we obtain some quantitative insight about that by reserving a part of the set for testing. By calculating the error also for the test set it is possible to evaluate the generalization. The **root-mean-square error** (**RMS**) is used:

$$E_{RMS} = \sqrt{\frac{2 * RSS(\hat{\mathbf{w}})}{N}}$$

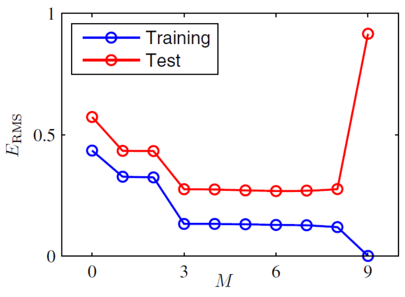
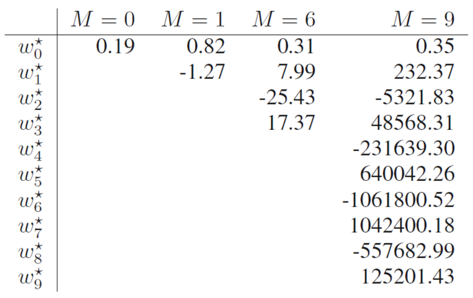


Figure 3.3: E_{RMS} on training and test sets

In which the division by N allows to compare sets of different sizes and the square root ensures that the error is based on the same scale as the target variable.
As we can see from Figure 3.3, from a specific value of the model complexity, even if the training error continues decreasing, ato some point the test error starts to increase. That's because there is over-fitting and the model has huge values for the parameters in order to try to fit all the points of the training set, as showed in Figure 3.4.



	$M=0$	$M=1$	$M=6$	$M=9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

Figure 3.4: Table of the coefficients $\mathbf{w}^*$ for polynomials of various order.

For a given model complexity, the over-fitting problem becomes less severe as the size of the data set increases. In other words, the larger the data set, the more complex (and flexible) the model we can afford to fit the data.

3.5.2 Ridge regression

Regularization involves adding a penalty term to the error function to discourage the coefficients from reaching large values and so, to make the curve smoother.

$$L(\mathbf{w}) = L_D(\mathbf{w}) + \lambda L_W(\mathbf{w})$$

Where $L_D(\mathbf{w})$ is the empirical loss on data (e.g. RSS) and $L_W(\mathbf{w})$ is the measure of the size of the parameters (the model complexity).
By taking $L_W(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} = \frac{1}{2}||\mathbf{w}||_2^2$ we obtain the **ridge regression** (or weight decay)

$$L(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(t_i - \mathbf{w}^T\phi(\mathbf{x}_i))^2 + \frac{\lambda}{2}||\mathbf{w}||_2^2$$

Note that if there are a lot of samples, the regularization is useless and it may even worsen the performance of the model.
The loss function is still quadratic in $\mathbf{w}$:

$$\hat{\mathbf{w}}_{ridge} = (\lambda\mathbf{I} + \Phi^T\Phi)^{-1}\Phi^T t$$

The matrix $\lambda\mathbf{I} + \Phi^T\Phi$ is always non-singular, because of the fact that $\Phi^T\Phi$ is semidefinite (eigenvalues ≥ 0) and so the whole matrix has all eigenvalues $\geq \lambda + 0 = \lambda$.

3.5.3 Lasso regression

Another popular regularization method is **lasso**

$$L(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(t_i - \mathbf{w}^T\phi(\mathbf{x}_i))^2 + \frac{\lambda}{2}||\mathbf{w}||_1$$

where $||\mathbf{w}||_1 = \sum_{j=1}^{M}|w_j|$
Differently from ridge, lasso is **nonlinear** in t_i and no closed-form solution exists (quadratic programming could be used). Nonetheless, it has the property that if λ is sufficiently large, some of the coefficients are driven to zero, leading to a **sparse model**. For this reason it can be also used for **feature selection**, excluding the ones that have coefficient equal to zero.

3.6 Bayesian Linear Regression

3.6.1 Bayesian approach

In the previous examples of linear regression we have viewed probabilities in terms of the frequencies of random, repeatable events (**frequentist** approach). In some cases it's not possible to repeat multiple times some events to define a notion of probability. In such circumstances, the solution is to quantify our expression of uncertainty and make precise revisions of the uncertainty after acquiring new evidence. This is the **Bayesian** approach, which can be divided in steps:

1. Formulate the knowledge about the world in a probabilistic way:
    - Define the model that expresses our knowledge quantitatively .
    - The model will have some unknown parameters.
    - Capture our assumptions about unknown parameters by specifying the prior distribution *p(**w**)* over those parameters before seeing the data.
2. Observe the data, whose effect is expressed through the conditional probability $p(\mathcal{D}|\mathbf{w})$. It can be viewed as a function of the parameter vector (**likelihood function**).
3. Compute the posterior probability distribution $p(\mathbf{w}|\mathcal{D})$ for the parameters, given the observed data. It is the uncertainty in $\mathbf{w}$ *after* $\mathcal{D}$ is observed.
4. Use the posterior distribution to:
    - Make predictions by averaging over the posterior distribution.
    - Examine/Account for uncertainty in the parameter values.
    - Make decisions by minimizing the expected posterior loss.

Note that this new approach is not affected by the problem of over-fitting.

3.6.2 Posterior Distribution

The posterior distribution for the model parameters can be found by combining the prior with the likelihood for the parameters given the data.
This is accomplished by using **Bayes' Rule**:

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{P(\mathcal{D})}$$

where $P(\mathcal{D})$ is the marginal likelihood (normalizing constant): $P(\mathcal{D}) = \int p(\mathcal{D}|\mathbf{w})P(\mathbf{w})d\mathbf{w}$ The Bayes' theorem can be expressed in words in multiple ways:

$$P(parameters|data) = \frac{P(data|parameters)P(parameters)}{P(data)}$$

or

$$\textit{posterior} \propto \textit{likelihood} \cdot \textit{prior}$$

Where all of these quantities are viewed as functions of $\mathbf{w}$.
The aim is to obtain the most probable value of $\mathbf{w}$ given the data (**maximum a posteriori** or **MAP**), which is the mode of the posterior.

If new data is available, it is possible to use the posterior value as prior to compute a new posterior. This sequential approach of learning depends only on the assumption of i.i.d[6] data. It can be useful in real-time learning scenarios o in case of large datasets, because they do not require the whole dataset to be stored or loaded into memory.
It is important that the prior and posterior have the same distribution. This fact leads to introducing the concept of **conjugate priors**: for a given probability distribution, we can seek a prior that is conjugate to the likelihood function, so that the posterior has the same distribution as the prior. For example the prior of a Gaussian is a Gaussian, and the prior of a Beta is a Bernoulli (so it gives another Beta as posterior).

3.6.3 Predictive Distribution

Prediction for a new data point $\mathbf{x}^*$ (given the training dataset $\mathcal{D}$) can be done by integrating over the posterior distribution:

$$p(\mathbf{x}^*|\mathcal{D}) = \int p(\mathbf{x}^*|\mathbf{w}, \mathcal{D}) p(\mathbf{w}|\mathcal{D}) \mathrm{d}\mathbf{w} = \mathbb{E}[p(\mathbf{x}^*|\mathbf{w}, \mathcal{D})]$$

which is sometimes called **predictive distribution**.
Note that computing the predictive distribution requires knowledge of the posterior distribution, which is usually intractable.

3.6.4 Bayesian Linear Regression

In the Bayesian approach the parameters of the model are considered as drawn from some distribution. Assuming a Gaussian likelihood model, the conjugate prior is Gaussian too

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{w}_0, \mathbf{S}_0)$$

Given the data $\mathcal{D}$, the posterior is still Gaussian:

$$p(\mathbf{w}|\mathbf{t}, \Phi, \sigma^2) \propto \mathcal{N}(\mathbf{w}|\mathbf{w}_0, \mathbf{S}_0)\mathcal{N}(\mathbf{t}|\Phi\mathbf{w}, \sigma^2\mathbf{I}_N) = \mathcal{N}(\mathbf{w}|\mathbf{w}_N, \mathbf{S}_N)$$

where

$$\mathbf{w}_N = \mathbf{S}_N \left(\underbrace{\mathbf{S}_0^{-1}\mathbf{w}_0}_{\text{from prior}} + \underbrace{\frac{\Phi^T\mathbf{t}}{\sigma^2}}_{\text{from data}} \right)$$

$$\mathbf{S}_N^{-1} = \mathbf{S}_0^{-1} + \frac{\Phi^T\Phi}{\sigma^2}$$

In a Gaussian distribution the mode coincides with the mean. It follows that $\mathbf{w}_N$ is the MAP estimator.
In many cases we may have little idea of what form the distribution should take. We may then seek a form of prior distribution, called a **noninformative prior**, which is intended to have as little influence on the posterior distribution as possible.
In this case the value $\mathbf{S}_0 \to \infty$ and so

$$\mathbf{S}_N^{-1} \to 0 + \frac{\Phi^T\Phi}{\sigma^2} \quad \Rightarrow \quad \mathbf{S}_N = \sigma^2(\Phi^T\Phi)^{-1} \quad \Rightarrow \mathbf{w}_N = \sigma^2(\Phi^T\Phi)^{-1}\frac{\Phi^T\mathbf{t}}{\sigma^2} = (\Phi^T\Phi)^{-1}\Phi^T\mathbf{t}$$

so, $\mathbf{w}_N$ reduces to the ML estimator. If $\mathbf{w}_0 = 0$ and $\mathbf{S}_0 = \tau^2\mathbf{I}$, then $\mathbf{w}_N$ reduces to the ridge estimate, where $\lambda = \frac{\sigma^2}{\tau^2}$

[6]independent and identically distributed

3.6.5 Posterior Predictive Distribution

We are interested in the **posterior predictive distribution**, which is the distribution over the output variable obtained taking into account all the models, each one with its density.

$$p(t|\mathbf{x},\mathcal{D},\sigma^2) = \int \mathcal{N}(t|\mathbf{w}^T\phi(\mathbf{x}),\sigma^2)\overbrace{\mathcal{N}(\mathbf{w}|\mathbf{w}_N,\mathbf{S}_N)}^{\substack{\text{probability}\\ \text{of the model}}}\,\mathrm{d}\mathbf{w}$$
$$= \mathcal{N}(t|\mathbf{w}_N^T\phi(\mathbf{x}),\sigma_N^2(\mathbf{x}))$$

where

$$\sigma_N^2(\mathbf{x}) = \underbrace{\sigma^2}_{\substack{\text{noise in the}\\ \text{target values}}} + \underbrace{\phi(\mathbf{x})^T\mathbf{S}_N\phi(\mathbf{x})}_{\substack{\text{uncertainty associated}\\ \text{with parameter values}}}$$

In the limit, as $N \to \infty$, the second term goes to zero.
The variance of the predictive distribution arises only from the additive noise governed by parameter σ.

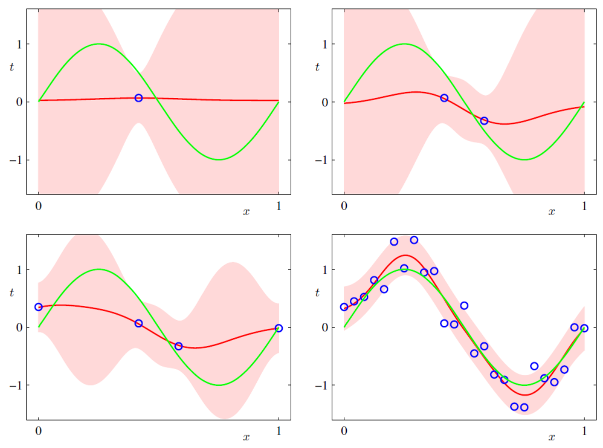


Figure 3.5: Example of the predictive distribution. The pink area will never go to zero for the intrinsic noise in the samples, even asymptotically

3.7 Notable indices

- Residual Sum of Squares $RSS(w) = \sum_n(\hat{t}_n - t_n)^2$, representing how much of the prediction differs from the true value.

- Coefficient of determination $R^2 = 1 - \frac{RSS(w)}{\sum_n(\bar{t}-t_n)^2}$ representing the fraction of the variance of the data explained by the model (i.e. how much better we are doing w.r.t just using the mean of the target $\bar{t}$). In space with a single feature this is equal to the correlation coefficient between the input and the output.[7]

- Degrees of Freedom $dfe = N - M$ representing how much our model is flexible in fitting the data. The rule of thumb is that *dfe* is enough if it is equal to 10^c where c is the number of parameters.

- Adjusted coefficient of determination $R^2_{adj} = R^2\frac{N}{dfe}$, which is the coefficient of determination corrected w.r.t. how much flexibility the model has.

- Root Mean Square Error $RMSE = \sqrt{\frac{RSS(w)}{N}}$, representing approximately how much error we get on a predicted data over the training set (i.e. a normalized version of the *RSS*.

[7] $\bar{t} = \frac{\sum_n t_n}{N}$

3.8 Modeling Challenges

A challenge is to specify a suitable model and suitable prior distributions:

- a suitable model should admit all the possibilities that thought to be all likely
- a suitable prior should avoid giving zero or very small probabilities to possible events, but should also avoid spreading out the probability over all possibilities

To avoid uninformative priors, we may need to model dependencies between parameters
One strategy is to introduce latent variables into the model and hyperparameters into the prior.

Chapter 4

Linear Classification

4.1 Introduction

The goal of classification is to assign an input $\mathbf{x}$ into one of K discrete classes C_k, where $k = 1, ..., K$. In linear classification, the input space is divided into decision regions whose boundaries are called **decision surfaces**.
If in linear regression models

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

are linear in the parameters, in linear classification models

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

are not necessarily linear in the features because the function f is nonlinear. In linear classification models are linear in the decision surfaces, which correspond to $y(\mathbf{x}, \mathbf{w}) = const$. In other words, there must be used nonlinear functions in the parameters to classify the input. Note that, as for regression, there can be used **fixed nonlinear basis functions**.

4.1.1 Notation

- In two-class problems, the binary target value $t \in \{0, 1\}$. t can be interpreted as the probability of the input to belong to the positive class.

- If there are K classes, t is a vector of length K and contains a single 1 for the correct class and 0 elsewhere (**1-of-K encoding**). For example, if there are 5 classes, a target vector is

$$t = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

  and the input value belongs to the fourth class.

4.1.2 Fixed Basis Functions

Being the decision surface linear, the input space must be linear separable. If it is not the case, the main idea is to make a **fixed non linear transformation** of the input space, using a vector of basis functions $\phi(\mathbf{x})$. In this way, decision boundaries will be linear in the **feature space**, but would correspond to non linear boundaries in the original input space.

In Figure 4.1, it can be seen that in the input space (left) the two classes are not linearly separable, but applying Gaussian basis functions it has been found a feature space (right) in which the two classes are linear separable.

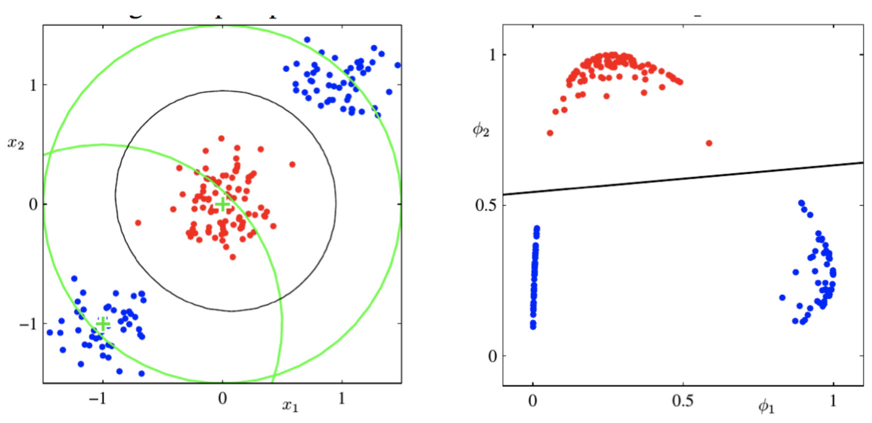


Figure 4.1

4.1.3 Approaches to Classification

- **Discriminant function** (or direct approach): build a function that directly maps each input to a specific class.
- **Probabilistic approach**: model the conditional probability distribution $p(C_k|\mathbf{x})$, that is the probability of the input $\mathbf{x}$ to belong to C_k.
  - **Probabilistic discriminative approach**: Model $p(C_k|\mathbf{x})$ directly, for instance using parametric models (e.g., logistic regression).
  - **Probabilistic generative approach**: Model $p(\mathbf{x}|C_k)$, that is the class conditional density, and $p(C_k)$, the probability to belong to a given class, and then using the **Bayes' rule**:

$$P(C_k|\mathbf{x}) = \frac{p(\mathbf{x}|C_k)p(C_k)}{p(\mathbf{x})}$$

4.2 Discriminant function (Direct approach)

4.2.1 Considerations

Let's assume a **two-class** problem. Considering the following model

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

a possible approach can be to assign $\mathbf{x}$ to C_1 if $y(\mathbf{x}) \geq 0$ and C_2 otherwise. This means that the decision boundary is $y(\mathbf{x}) = 0$ and it is a $(D-1)$-dimensional hyperplane within the D-dimensional input space.

Direction of the boundary

Let's consider two points on the decision surface:

$$y(\mathbf{x}_A) = y(\mathbf{x}_B) = 0$$

$$\begin{cases} \mathbf{x}_A^T\mathbf{w} + w_0 = 0 \\ \mathbf{x}_B^T\mathbf{w} + w_0 = 0 \end{cases} \tag{4.1}$$

$$(\mathbf{x}_A - \mathbf{x}_B)^T\mathbf{w} = 0$$

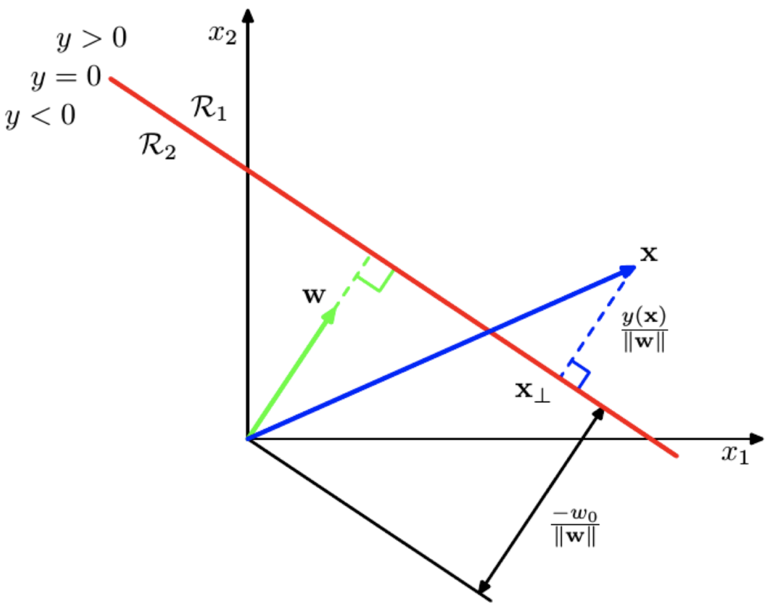
The last equation has a particular meaning: $\mathbf{w}$ is **orthogonal** to the decision surface (identified by the difference vector between $\mathbf{x}_A$ and $\mathbf{x}_B$). Remember, indeed, that if the scalar product between two vectors is null, then the two vectors are perpendicular.
This means that the parameter vector $\mathbf{w}$ changes the orientation of the decision surface.

Location of the decision surface

Let's consider an input $\mathbf{x}$ on the decision surface. As already said, it means that $y(\mathbf{x}) = 0$. Applying the distance between the decision surface and the origin

$$d(\mathbf{0}, y(\mathbf{x})) = \frac{|\mathbf{0} \cdot y(\mathbf{x})|}{||\mathbf{w}||} = \frac{|\sum_{i=1}^{D} 0 \cdot w_i + w_0|}{||\mathbf{w}||} = \frac{|w_0|}{||\mathbf{w}||}$$

and hence w_0 determines the **translation** of the separating hyperplane with respect to the origin.



In the figure above:

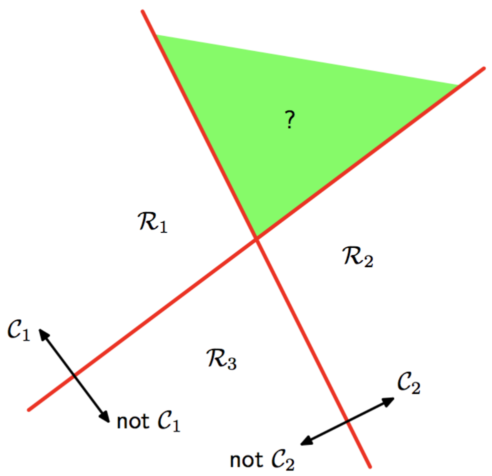
- in red the decision surface
- in green the weight vector
- in blue the input

Note that the parameter vector $\mathbf{x}$ is orthogonal to the decision surface and that the distance, with sign, between the input vector $\mathbf{x}$ and the separating hyperplane is given by $\frac{y(\mathbf{x})}{||\mathbf{w}||}$

Multiple classes classification

Let's focus now on **multiple classes**: $K > 2$ classes. There are different approaches in handling multiple classes.

- **One-versus-the-rest**: modeling $K-1$ different classifiers, each of which solves a two class problem

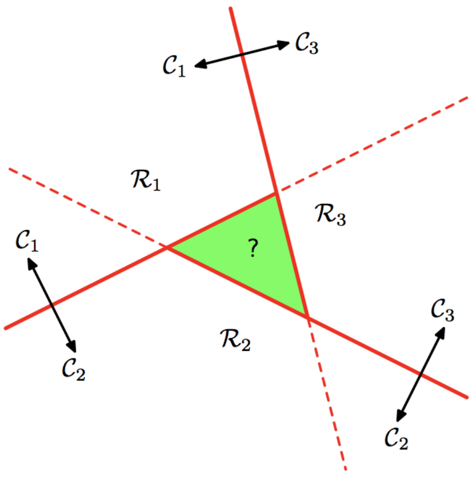


For example, considering the classes C_1 and C_2, there will be 4 different regions:

- R_1: the input is a cat and not a dog
- R_2: the input is a dog and not a cat
- R_3: the input is neither a cat nor a dog
- R_4: the input is a cat and a dog

If the classes are mutually exclusive, there is obviously a problem in R_4.

- **One-versus-one**: modeling $\frac{K(K-1)}{2}$ binary classifiers



Note that also in this case, there is a problem of ambiguity.

- Using K **linear discriminant functions** of the form:

$$y_k(\mathbf{x}) = \mathbf{x}^T \mathbf{w}_k + w_{k0}$$

where $k = 1, ..., K$ and $\mathbf{x}$ will be assigned to the class C_k such that $y_k(\mathbf{x}) > y_j(\mathbf{x}) \quad \forall j \neq k$. This means that the decision boundary between two generic classes C_k and C_j is the hyperplane identified by

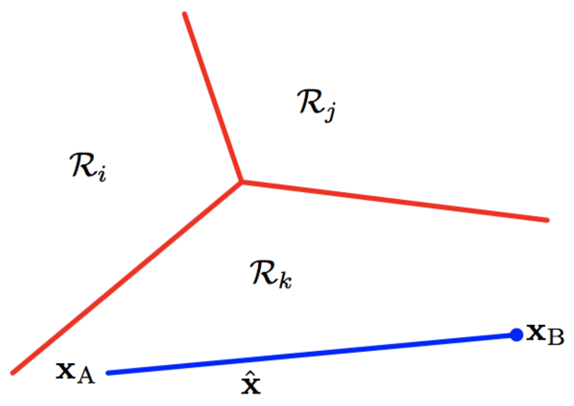
$$y_k(\mathbf{x}) = y_j(\mathbf{x})$$

and hence corresponds to

$$(w_k - w_j)^T \mathbf{x} + (w_{k0} - w_{j0}) = 0.$$

The decision regions of such discriminants have the same form of the decision boundary discussed in the two-class approach and so analogous geometrical properties hold.

An important property of this type of classifiers is that the decision regions are **singly connected** and **convex** (as it can be seen in the figure below).



Proof of the singular connection and convex property

Let's consider two points in R_k: $\mathbf{x}_A$ and $\mathbf{x}_B$. Any point $\hat{\mathbf{x}}$ that lies on the line connecting $\mathbf{x}_A$ and $\mathbf{x}_B$ can be expressed by the linear combination of these two points:

$$\hat{\mathbf{x}} = \lambda \mathbf{x}_A + (1 - \lambda)\mathbf{x}_B$$

with $0 \leq \lambda \leq 1$. But, from the linearity of the discriminant functions, it follows that

$$y_k(\hat{\mathbf{x}}) = \lambda y_k(\mathbf{x}_A) + (1 - \lambda) y_k(\mathbf{x}_B).$$

Taking in consideration now that $\mathbf{x}_A$ and $\mathbf{x}_B$ lie inside R_k, it follows that

$$y_k(\mathbf{x}_A) > y_j(\mathbf{x}_A)$$

$\forall j \neq k$ and hence

$$y_k(\hat{\mathbf{x}}) > y_j(\hat{\mathbf{x}})$$

and for this reason also $\hat{\mathbf{x}}$ lies inside R_k.
In this way, it has been proven the convexity of this type of decision surfaces.

4.2.2 Least squares for Classification

Here it will be applied least squares to find a solution for classification. As it will explained below, this is not a convenient way to find an optimal classifier. The reason why it is taken in consideration is only to try to use the same formalism to classification problems.
Considering a general classification problem with K classes using 1-of-K encoding scheme for the target vector $\mathbf{t}$. This means that each class is described by its own linear model

$$y_k(\mathbf{x}) = \mathbf{x}^T \mathbf{w}_k + w_{k0}$$

where $k = 1, ..., K$. In an alternative way, using vector notation to group all the classes

$$\mathbf{y}(\mathbf{x}) = \tilde{\mathbf{W}}^T \tilde{\mathbf{x}}$$

where

$$\tilde{\mathbf{W}} = \begin{bmatrix} w_{0,1} \dots w_{0,K} \\ \vdots \ddots \vdots \\ w_{D,1} \dots w_{D,K} \end{bmatrix}$$

is a $(D+1) \times K$ matrix (each column is a weight vector of different classifier) and $\tilde{\mathbf{x}} = \begin{bmatrix} 1 \\ x_0 \\ \vdots \\ x_D \end{bmatrix}$

The next step is to find the optimal weight matrix $\tilde{\mathbf{W}}$.
Given a dataset $\mathcal{D} = \{\mathbf{x}_i, t_i\}$, where $i = 1, ..., N$ and considering the loss function

$$E_D(\tilde{\mathbf{W}}) = \frac{1}{2} Tr\{(\tilde{\mathbf{X}}\tilde{\mathbf{W}} - \mathbf{T})^T (\tilde{\mathbf{X}}\tilde{\mathbf{W}} - \mathbf{T})\}$$

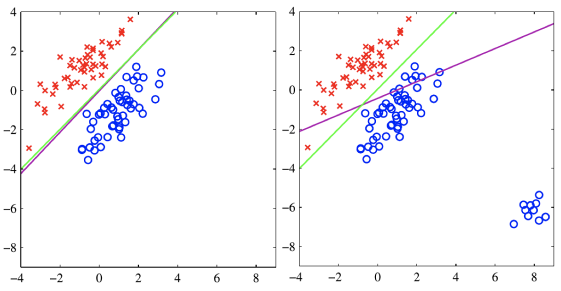
where $Tr\{\}$ means the trace of the matrix

$$\tilde{\mathbf{X}} = \begin{bmatrix} \tilde{\mathbf{x}}_1^T \\ \dots \\ \tilde{\mathbf{x}}_N^T \end{bmatrix}$$

Minimizing least squares will lead to the already known closed-form solution

$$\tilde{\mathbf{W}} = (\tilde{\mathbf{X}}^T \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^T \mathbf{T}$$

Problem with least squares Approach



In the two figures above, it can be seen the main problem using least squares. The left plot shows data from two classes, denoted by red crosses and blue circles, together with the decision boundary found by least squares (magenta curve) and also by the logistic regression model (green curve). The right-hand plot shows the corresponding results obtained when extra data points are added at the bottom right of the diagram, showing that least squares is highly sensitive to **outliers**, unlike e.g., logistic regression. The solution may also fail due to the assumption of a Gaussian conditional distribution that is not satisfied by the binary target vectors.

4.2.3 Perceptron

The **perceptron** is another example of **linear discriminant model**. It is, unlike least squares that has a closed form solution, an **online** linear classification algorithm.

It corresponds to a two-class model:

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

where f is a step function

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

This means that target values are +1 for C_1 and -1 for C_2.

The algorithm finds the separating hyperplane by minimizing the distance of **misclassified points** to the decision boundary. For this reason, the loss function to be minimized is

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

where $\mathcal{M}$ is the set of misclassified points.

The minus in the loss function is a trick to make it always positive because, in that summation (that includes **only** misclassified points), $\mathbf{w}^T \phi(\mathbf{x}_n) t_n$ is always negative. Note that $\mathbf{w}^T \phi(\mathbf{x}_n)$ is proportional to the distance to the decision boundary and, for this reason, it is a quantitative value of how much the solution found is miscalculating the current misclassified point.

Minimization is performed using **stochastic gradient descent**:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha \nabla L_P(\mathbf{w}) = \mathbf{w}^{(k)} + \alpha\phi(\mathbf{x}_n)t_n$$

Since the hyperplane's position doesn't change based on the scaling factor, the learning rate α can be set to 1.

The algorithm

```
Input:data set $\mathbf{x}_n \in \mathbb{R}^D$,
$t_n \in \{-1, +1\}$, for $n = 1 : N$
Initialize $\mathbf{w}_0$
$k \leftarrow 0$
repeat
    $k \leftarrow k + 1$
    $n \leftarrow k \bmod N$
    if $\hat{t}_n \neq t_n$ then
        $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + \phi(\mathbf{x}_n)t_n$
    end if
until convergence
```

Perceptron Convergence Theorem

The effect of a single update is to reduce the error due to the misclassified point. But this does not imply that the overall loss is reduced at each stage, we could even increase. For this reason, one could ask if the algorithm is able to find an optimal solution. The following theorem helps:

Theorem
If the training dataset is ***linearly separable*** *in the feature space Φ, then the perceptron learning algorithm is guaranteed to find an* ***exact solution*** *in a* ***finite number of steps****.*

Of course, the number of steps might be substantial and for this reason it is impossible to distinguish between **nonseparable** problems and **slowly converging** ones (it is a semidecidable problem).
Furthermore, if multiple solutions exist, the one found depends on the initialization of the parameters $\mathbf{w}$ and the order of presentation of the data points.

4.3 Probabilistic Discriminative Approach

Let's now move on a probabilistic approach: logistic regression. Although the name might confuse, please note that it is a classification algorithm.

4.3.1 Logistic Regression

Considering a problem of two-class classification, in logistic regression the posterior probability of class C_1 can be written as a **logistic sigmoid function**

$$p(C_1|\phi) = \frac{1}{1 + e^{-\mathbf{w}^T\phi}} = \sigma(\mathbf{w}^T\phi)$$

and $p(C_2|\phi) = 1 - p(C_1|\phi)$. Note that the term 'sigmoid' means S-shaped.

4.3.2 Maximum Likelihood for logistic regression

Given a dataset $\mathcal{D} = \{\mathbf{x}_n, t_n\}, t_n \in \{0, 1\}$, applying ML approach means to maximize the probability of getting the right label:

$$P(\mathbf{t}|\mathbf{X}, \mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n}(1 - y_n)^{1-t_n}, \quad y_n = \sigma(\mathbf{w}^T \phi_n)$$

Taking the negative log of the likelihood, the **cross-entropy error function** can be defined and it has to be minimized:

$$L(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}) = -\sum_{n=1}^{N}(t_n \ln y_n + (1 - t_n) \ln(1 - y_n)) = \sum_{n=1}^{N} L_n$$

Differentiating and using the chain rule:

$$\frac{\partial L_n}{\partial y_n} = \frac{y_n - t_n}{y_n(1 - y_n)}, \qquad \frac{\partial y_n}{\partial \mathbf{w}} = y_n(1 - y_n)\phi_n, \qquad \frac{\partial L_n}{\partial \mathbf{w}} = \frac{\partial L_n}{\partial y_n}\frac{\partial y_n}{\partial \mathbf{w}} = (y_n - t_n)\phi_n$$

The gradient of the loss function is

$$\nabla L(\mathbf{w}) = \sum_{n=1}^{N}(y_n - t_n)\phi_n$$

It has the same form as the gradient of the sum-of-squares error function for linear regression.
But in this case y is not a linear function of $\mathbf{w}$ and so, there is no closed form solution. The error function is **convex** (only one optimum) and can be optimized by standard gradient-based optimization techniques. It is, hence, easy to adapt to the online learning setting.

4.3.3 Multiclass Logistic Regression

For the multiclass case, the posterior probabilities can be represented by a softmax transformation of linear functions of feature variables

$$p(C_k|\phi) = y_k(\phi) = \frac{e^{\mathbf{w}_k^T \phi}}{\sum_j e^{\mathbf{w}_j^T \phi}}$$

$\phi(\mathbf{x})$ has been abbreviated with ϕ for simplicity.
Maximum likelihood is used to directly determine the parameters

$$p(\mathbf{T}|\Phi, \mathbf{w}_1, ..., \mathbf{w}_K) = \prod_{n=1}^{N} \underbrace{\left(\prod_{k=1}^{K} p(C_k|\phi_n)^{t_{nk}}\right)}_{\text{Term for correct class}} = \prod_{n=1}^{N}\left(\prod_{k=1}^{K} y_{nk}^{t_{nk}}\right)$$

where $y_{nk} = p(C_k|\phi_n) = \frac{e^{\mathbf{w}_k^T \phi_n}}{\sum_j e^{\mathbf{w}_j^T \phi_n}}$
The cross-entropy function is

$$L(\mathbf{w}_1, ..., \mathbf{w}_K) = -\ln p(\mathbf{T}|\Phi, \mathbf{w}_1, ..., \mathbf{w}_K) = -\sum_{n=1}^{N}\left(\sum_{k=1}^{K} t_{nk} \ln y_{nk}\right)$$

Taking the gradient

$$\nabla L_{\mathbf{w}_j}(\mathbf{w}_1, ..., \mathbf{w}_k) = \sum_{n=1}^{N}(y_{nj} - t_{nj})\phi_n$$

4.4 Probabilistic Generative Approach

Generative models have the purpose of modeling the joint probability density function of the couple input/output $p(C_k, \mathbf{x})$, which allows to generate also new data from what has been learned.

4.4.1 Naive Bayes

This method is based on the assumption that each input is conditionally (w.r.t the class) independent from each other. By considering Bays rule:

$$\begin{aligned}
&p(C_k|\mathbf{x}) = \frac{p(C_k)p(\mathbf{x}|C_k)}{p(\mathbf{x})} \propto p(x1, ..., x_M, C_k) \\
&= p(x_1|x_2, ..., x_M, C_k)p(x_2, ..., x_M, C_k) \\
&= p(x_1|x_2, ..., x_M, C_k)p(x_2|x_3, ..., x_M, C_k)p(x_3, ..., x_M, C_k) \\
&= p(x_1|x_2, ..., x_M, C_k)p(x_2|x_3, ..., x_M, C_k)...p(x_{M-1}|x_M, C_k)p(x_M|C_k)p(C_k) \\
&= p(C_k)\prod_{j=1}^{M} p(x_j|C_k)
\end{aligned}$$

The decision function, which maximizes the MAP probability, is the following:

$$y(\mathbf{x}) = arg\max_k p(C_k)\prod_{j=1}^{M} p(x_j|C_k)$$

In case of continuous variable, one of the usual assumption is to use Gaussian distributions for each variable $p(x_j|C_k) = \mathcal{N}(x_j; \mu_{jk}, \sigma^2_{jk})$ and a uniform prior $p(C_k) = \frac{1}{K}$

4.5 Non-parametric methods

IN **non-parametric methods**, differently from parametric ones, the training set cannot be discarded after training because it will be used also during the prediction phase.

4.5.1 K-nearest neighbor

Specifically for **K-NN** the training phase is practically absent but the prediction phase is quite slow.
For each sample to predict, the closest k samples are selected and the label belonging to the majority of them is assigned to the new sample. If k is even, a policy for breaking the ties has to be chosen, e.g. randomly.
The concept of *closest* requires the definition of a similarity measure, which is not always trivial but that has the advantage of the possibility to use K-NN also for objects (such as graphs) for which a similarity can be defined.
It is affected by the **curse of dimensionality**, which means that having a very high number of dimensions will decrease the performance of the predictor. The curse is caused by the fact that with high dimensions, all the points tend to have the same distance from one to another.
The choice of the k parameter is very important for the performance of the algorithm and it can be chosen through cross-validation. A very low k will have high variance and low bias, while a high k will have a low variance but high bias.

4.6 Performance measures

To evaluate the performance of a method we need to consider the **confusion matrix**, which says the number of points which have been correctly classified and those which have been misclassified.

	Actual Class: 1	Actual Class: 0
Predicted Class: 1	*tp*	*fp*
Predicted Class: 0	*fn*	*tn*

By basing on this matrix the following metrics can be evaluated:

- Accuracy: $Acc = \frac{tp+tn}{N}$ fraction of the samples correctly classified in the dataset
- Precision: $Pre = \frac{tp}{tp+fp}$ fraction of samples correctly classified in the positive class among the ones classified in the positive class

- Recall: $Rec = \frac{tp}{tp+fn}$ fraction of samples correctly classified in the positive class among the ones belonging to the positive class
- F1 score: $F1 = \frac{2 \cdot Pre \cdot Rec}{Pre+Rec}$ harmonic mean of the precision and recall

Chapter 5

Bias-Variance trade-off and model selection

5.1 No Free Lunch Theorems

Define $Acc_G(L)$ as the generalization accuracy of the learner L, which is the accuracy of L on non-training samples.
$\mathcal{F}$ is the set of all possible concepts, $y = f(x)$, that are the true models we want to learn.

Theorem
For any learner L, $\frac{1}{|\mathcal{F}|}\sum_{\mathcal{F}} Acc_G(L) = \frac{1}{2}$
(given any distribution $\mathcal{P}$ over $\mathbf{x}$ and the training set size N)

This means that for any learner, *on average* of all the models, has the same accuracy of random sampling, so, what it is learned from the seen data is useless for predicting new data.
But this does not invalidate the entire field of Machine Learning because it is usually based on the assumption that not all models are equally likely, because the seen data is related to the unseen data and so the distribution over models is restricted after seeing the data.

Corollary
For any two learners L_1, L_2:
if $\exists$learning problems s.t. $Acc_G(L_1) > Acc_G(L_2)$
then $\exists$learning problems s.t. $Acc_G(L_2) > Acc_G(L_1)$

This means that there is no best method for solving all the machine learning problems, because every method is based on some assumption and it can perform better only on some subset of models.

5.2 Bias-Variance Tradeoff

5.2.1 Bias-Variance decomposition

The introduction of regularization terms can control over-fitting for models with many parameters, but this raises the question of how to determine a suitable value for the regularization coefficient λ.
Assume that we have a data set $\mathcal{D}$ with N samples obtained by a function $t_i = f(\mathbf{x}_i) + \epsilon$ with $\mathbb{E}[\epsilon] = 0$ and $Var[\epsilon] = \sigma^2$.
We want to find a model $y(\mathbf{x})$ that approximates f as well as possible. Consider the **expected square**

error on an unseen sample $\mathbf{x}$.

$$\begin{aligned}
\mathbb{E}[(t-y(\mathbf{x}))^2] &= \mathbb{E}[t^2 + y(\mathbf{x})^2 - 2ty(\mathbf{x})] \\
&= \mathbb{E}[t^2] + \mathbb{E}[y(\mathbf{x})^2] - \mathbb{E}[2ty(\mathbf{x})] \\
&= \mathbb{E}[t^2] \pm \mathbb{E}[t]^2 + \mathbb{E}[y(\mathbf{x})^2] \pm \mathbb{E}[y(\mathbf{x})]^2 - 2f(\mathbf{x})\mathbb{E}[y(\mathbf{x})] \\
&= Var[t] + \mathbb{E}[t]^2 + Var[y(\mathbf{x})] + \mathbb{E}[y(\mathbf{x})]^2 - 2f(\mathbf{x})\mathbb{E}[y(\mathbf{x})] \\
&= Var[t] + Var[y(\mathbf{x})] + (f(\mathbf{x}) - \mathbb{E}[y(\mathbf{x})])^2 \\
&= \underbrace{Var[t]}_{\sigma^2} + \underbrace{Var[y(\mathbf{x})]}_{\text{Variance}} + \underbrace{\mathbb{E}[f(\mathbf{x}) - y(\mathbf{x})]^2}_{\text{Bias}^2}
\end{aligned}$$

Note that t is stochastic and $y(\mathbf{x})$ is deterministic but can be considered stochastic because data has noise.

5.2.2 Bias

If the data set $\mathcal{D}$ is sampled multiple times you expect to learn a different $y(\mathbf{x})$.
The expected hypothesis is $\mathbb{E}[y(\mathbf{x})]$.
The **bias** is the difference between the average prediction over all data sets and the true function:

$$\int (f(\mathbf{x}) - \mathbb{E}[y(\mathbf{x})])^2 p(\mathbf{x}) \mathrm{d}\mathbf{x}$$

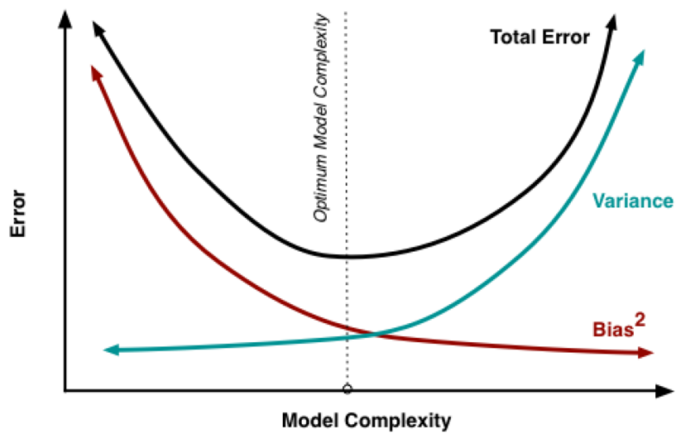
A high bias means that even with a lot of samples it is not possible to learn the true model (under-fitting).
It decreases with more complex models.

5.2.3 Variance

It measures how much the solutions for individual data sets vary around their average:

$$\int \mathbb{E}[(y(\mathbf{x}) - \bar{y}(\mathbf{x}))^2] p(\mathbf{x}) \mathrm{d}\mathbf{x}$$

where $\bar{y}(\mathbf{x}) = \mathbb{E}[y(\mathbf{x})]$. A high variance means that the model depends highly on noise (over-fitting) and so, its solutions vary a lot depending on the particular choice of the data set.
It can decrease by using simpler models or with more samples and the latter does not increase the bias (but obviously it is not always possible to have more samples).



5.2.4 Prediction Error

The training error is not necessarily a good measure, because we care about the error over all the input points.

Regression:

$$L_{true} = \int (f(\mathbf{x}) - y(\mathbf{x}))^2 p(\mathbf{x}) \mathrm{d}\mathbf{x}$$

Classification:

$$L_{true} = \int I(f(\mathbf{x}) \neq t(\mathbf{x})) p(\mathbf{x}) \mathrm{d}\mathbf{x}$$

The **training error** is an optimistically bias estimate of the **prediction error**.
For this reason the data set is usually split into **test set** and **training set**. The training data is used to optimize the parameters and the test data is used to evaluate the prediction error

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (t_n - y(\mathbf{x}_n))^2$$

If the data set is independent it is not biased and so, it must not be used for learning or to choose the model complexity.
Now different techniques used to manage the bias-variance tradeoff will be described

5.3 Model selection

The **curse of dimensionality** is related to the exponential increase in volume associated with adding extra dimensions to the input space, leading to a decrease of power of the samples.
Working with high-dimensional data is difficult:

- large variance (over-fitting)
- needs many samples: N^d
- high computational cost

If we can't solve a problem with a few features, adding more features seems like a good idea. However if the number of samples stays the same, the method with more features is likely to perform worse.

5.3.1 Feature Selection

Best subset selection

It identifies a subset of input features that are the most related to the output.
The best subset selection method is to try all the possible combinations of features.
Let $\mathcal{M}_0$ denote the **null model**, which contains no input features (it simply predicts the mean).
For $k = 1, 2, ...M$

1. Fit all $\binom{M}{k}$ models that contain exactly k-features
2. Pick the best among these models and call it $\mathcal{M}_k$. The quality is assessed based on the training error.

Then select a single best model from among $\mathcal{M}_0, ..., \mathcal{M}_M$ using some criterion (cross-validation, AIC, BIC,...).
This approach has problems of high computational cost and over-fitting when M is large.
Three **metaheuristics** are used:

- **Filter**: rank the features (ignoring the classifier) and select the best ones
- **Embedded**: the learning algorithm exploits its own variable selection technique (Lasso, ...)
- **Wrapper**: evaluate only some subsets
  - **Forward step-wise selection**: starts from an empty model and adds features one at a time
  - **Backward step-wise selection**: starts with all the features and removes the least useful features one at a time

The model containing all the features will always have the smallest training error. We wish to choose a model with low test error and there are two approaches to estimate it:

- Direct estimation using a validation approach
- Making an adjustment to the training error to account for model complexity

Direct estimation

A possible way is to randomly split the data set into training set, validation set and test set and use the validation set to tune the learning algorithm. But if the validation set is not big enough there is a chance of over-fitting over the validation set.
To prevent over-fitting cross-validation can be used.
Leave-One-Out Cross Validation (LOO) uses a validation set $\{n\}$ with 1 example extracted from the data set $\mathcal{D}$ and learns the model with $\mathcal{D} \setminus \{n\}$ data set.
The estimation error is based on the performance over only one point (which is very noisy) but this process is repeated for all the N points of the data set and the error is averaged.

$$L_{LOO} = \frac{1}{N} \sum_{n=1}^{N} (t_n - y_{\mathcal{D}\setminus\{n\}}(\mathbf{x}_n))^2$$

LOO is almost unbiased and it is slightly pessimistic because the training is over N - 1 instead of N, but unfortunately the computational cost is high.
***k*-fold Cross Validation** randomly divides the training data into k equal parts: $\mathcal{D}_1, ..., \mathcal{D}_k$ and for each i:

- Learns the model $y_{\mathcal{D}\setminus\mathcal{D}_i}$
- Estimates the error of $y_{\mathcal{D}\setminus\mathcal{D}_i}$ on validation set $\mathcal{D}_i$

$$L_{\mathcal{D}_i} = \frac{k}{N} \sum_{(\mathbf{x}_n, t_n) \in \mathcal{D}_i} (t_n - y_{\mathcal{D}\setminus\mathcal{D}_i}(\mathbf{x}_n))^2$$

- All the k errors are averaged

$$L_{k-fold} = \frac{1}{k} \sum_{i=1}^{k} L_{\mathcal{D}_i}$$

Usually $k = 10$ is used.
Using cross-validation for tuning the model parameters means selecting a grid of parameter values an compute the cross-validation error rate for each of them. Then select the values for which the cross-validation error is the smallest and finally re-fit the model using all the samples and the selected values for the tuning parameters.
k-fold Cross validation is much faster than LOO but it is more (pessimistically) biased.

Adjustment Techniques

These techniques are usually used when we have a small dataset and a complex model

- $C_p = \frac{1}{N}(RSS + 2d\tilde{\sigma}^2)$ where d is the total number of parameters, $\tilde{\sigma}^2$ is an estimate of the variance of the noise ϵ
- AIC $= -2 \log L + 2d$ where L is the maximized value of the likelihood function for the estimated model
- BIC$= \frac{1}{N}(RSS + \log(N)d\tilde{\sigma}^2)$ it replaces the $2d\tilde{\sigma}^2$ of C_p with $\log(N)d\tilde{\sigma}^2$ term. Since $\log N > 2$ for any $n > 7$, BIC select smaller models
- $AdjustedR^2 = 1 - \frac{RSS/(N-d-1)}{TSS/(N-1)}$ where TSS is the total sum of squares $\sum_{n=1}^{N}(y_n - \bar{y})^2$. Differently from the other criteria, here a large value indicates a model with a small test error.

5.3.2 Regularization

Ridge regression and Lasso are methods for shrinking the parameters towards zero and the shrinking coefficient estimates can significantly reduce the variance. So, because of the penalization, even with different data, the models will be more similar.
But, at the same time, regularization increases bias, because the penalization in the loss function modifies the objective of the optimization and so, even with infinite samples, it would be impossible to obtain a perfect solution. That's the reason why regularization should not be used when a lot of samples are available.

5.3.3 Dimension reduction

Differently from the previous approaches, dimension reduction methods transform the original features and then the model is learned on the transformed variables.
It is an unsupervised learning approach.

Principal Component Analysis (PCA)
It is based on the principle of projecting the data onto the input subspace which accounts for most of the variance:

- Find a line such that when the data is projected onto that line, it has the maximum variance.
- Find a new line, orthogonal to the first one, that has maximum projected variance.
- Repeat until m lines have been identified and project the points in the data set on these lines.

The precise steps of PCA are the following:

- Compute the mean of the data
$$\bar{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n$$
- Bring the data to zero-mean (by subtracting $\bar{\mathbf{x}}$)
- Compute the covariance matrix $\mathbf{S} = \mathbf{X}^T\mathbf{X} = \frac{1}{N-1}\sum_{n=1}^{N}(\mathbf{x}_n - \bar{\mathbf{x}})(\mathbf{x}_n - \bar{\mathbf{x}})^T$
  - Eigenvector $\mathbf{e}_1$ with largest eigenvalue λ_1 is the first principal component.
  - Eigenvector $\mathbf{e}_k$ with k^{th} largest eigenvalue λ_k is the k^{th} principal component.
  - $\lambda_k / \sum_i \lambda_i$ it the proportion of variance captured by the k^{th} principal component.

The full set of PCs comprises a new orthogonal basis for feature space, whose axes are aligned with the maximum variances of the original data and that are a linear combination of the original components.
The projection of the original data onto the first k PCs ($\mathbf{E}_k = (\mathbf{e}_1, ..., \mathbf{e}_k)$) gives a reduced dimensionality representation of the data
$$\mathbf{X}' = \mathbf{X}\mathbf{E}_k$$
Transforming the reduced dimensionality projection back into the original spaces gives a reduced dimensionality reconstruction of the data, that will have some error. This error can be small and often acceptable given the other benefits of dimensionality reduction.
PCA has multiple benefits:

- Helps to reduce the computational complexity
- Can help supervised learning, because reduced dimensions allow simpler hypothesis spaces and less risk of overfitting
- Can be used for noise reduction

But also some drawbacks:

- Fails when data consists of multiple clusters
- The directions of greatest variance may not be the most informative

- Computational problems with many dimensions
- PCA computes linear combination of features, but data often lies on a nonlinear manifold. Suppose that the data is distributed on two dimensions as a circumference: it can be actually represented by one dimension, but PCA is not able to capture it.

5.4 Model Ensembles

We now consider meta-algorithms, which, instead of learning one model, learn several and combine them.

5.4.1 Bagging

This algorithm reduces the variance without increasing the bias, but the variance needs to be high initially in order to see considerable improvements.
It is based on the principle that averaging over multiple models reduces the variance:

$$Var(\bar{X}) = \frac{Var(X)}{N}$$

Actually this is not the real reduction because it is based on the assumption that the models are independent.
To be able to train multiple models from a unique train set, **bootstrapping** is performed: generate B bootstrap samples of the training data by random sampling with replacement.
Then train a model for each bootstrap sample and the prediction will be determined through majority vote in case of classification or through average of the predicted values in case of regression.
It improves the performance for unstable learners which vary significantly with small changes in the data set.
Note that Bagging is relatively easy to parallelize because all the models work on the same problem independently.

5.4.2 Boosting

The aim of boosting is to reduce the bias without increasing the variance (too much).
It works by sequentially training **weak learners**, learners that have a performance that on **any** train set is slightly better than chance prediction (high bias).
The steps are the following:

1. Weight all the train samples equally
2. Train a weak model on the train set
3. Compute the error of the model on the train set
4. Increase the weights on the train cases where the model gets wrong
5. Train a new model on re-weighted train set, so that the model is more concerned on misclassified points.
6. Repeat from 3. until tired
7. The final model is composed by the weighted prediction of each model

Boosting might hurt the performance on noisy datasets, while Bagging doesn't have this problem.

Chapter 6

PAC-Learning and VC-Dimension

6.1 PAC-Learning

Overfitting happens because:

- with a large hypothesis space the training error is a bad estimate of the prediction error, hence we would like to infer something about the generalization error from the training samples.
- the learner doesn't have access to enough samples, hence we would like to estimate how many samples are enough.

This cannot be performed by measuring the bias and the variance, but we can bound them.

6.1.1 The problem setting

We are focusing on classification, so, given:

- Set of instances X.
- Set of hypotheses H (finite).
- Set of possible target concepts C. Each concept c corresponds to a boolean function $c : \mathbf{X} \to \{0, 1\}$ which can be viewed as belonging to a certain class or not.
- Training instances generated by a fixed, unknown probability distribution $\mathcal{P}$ over $\mathbf{X}$.

The learner observes a sequence $\mathcal{D}$ of training examples $\langle x, c(x)\rangle$, for some target concept $c \in C$ and it must output a hypothesis h estimating c.
h is evaluated by its performance on subsequent instances drawn according to $\mathcal{P}$

$$L_{true} = Pr_{x\in\mathcal{P}}[c(x) \neq h(x)]$$

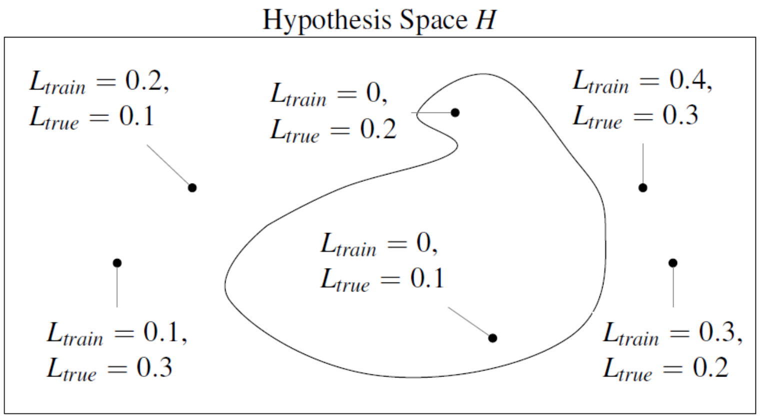
We want to bound L_{true} given L_{train}, which is the percentage of misclassified training instances.

6.1.2 Version Spaces

The version space $VS_{H,\mathcal{D}}$ is the subset of hypothesis in H consistent with training data $\mathcal{D}$ ($L_{train} = 0$). How likely is the learner to pick a bad hypothesis?

Theorem
If the hypothesis space H is finite and $\mathcal{D}$ is a sequence of $N \geq 1$ independent random examples of some target concept c, then for any $0 \leq \epsilon \leq 1$, the probability that $VS_{H,\mathcal{D}}$ contains a hypothesis error greater that ϵ is less than $|H|e^{-\epsilon N}$:

$$Pr(\exists h \in H : L_{train} = 0 \land L_{true} \geq \epsilon) \leq |H|e^{-\epsilon N}$$


Hypothesis Space H
$L_{train} = 0.2$, $L_{true} = 0.1$
$L_{train} = 0$, $L_{true} = 0.2$
$L_{train} = 0.4$, $L_{true} = 0.3$
$L_{train} = 0$, $L_{true} = 0.1$
$L_{train} = 0.1$, $L_{true} = 0.3$
$L_{train} = 0.3$, $L_{true} = 0.2$

We want this probability to be at most δ

$$|H|e^{-\epsilon N} \leq \delta$$

We can compute N based on the other variables

$$N \geq \frac{1}{\epsilon}\left(\ln|H| + \ln\left(\frac{1}{\delta}\right)\right)$$

Note that if we consider M boolean features, there are $|C| = 2^M$ distinct concepts and hence $|H| = 2^{2^M}$. This means that even if we have a logarithmic dependency on $|H|$, it is still exponential w.r.t. M.
Consider a class C of possible target concepts defined over a set of instances X and a learner L using hypothesis space H.

Definition
C is **PAC-learnable** if there exists and algorithm L such that for every $c \in C$, for any distribution $\mathcal{P}$, for any ϵ such that $0 \leq \epsilon < \frac{1}{2}$ and δ such that $0 \leq \delta < 1$, with probability at least $1 - \delta$, outputs an hypothesis $h \in H$ such that $L_{true}(h) \leq \epsilon$ using a number of samples that is polynomial of $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.

Definition
C is **efficiently PAC-learnable** by learner L using H if and only if for every $c \in C$, for any distribution $\mathcal{P}$, for any ϵ such that $0 \leq \epsilon < \frac{1}{2}$ and δ such that $0 \leq \delta < \frac{1}{2}$, with probability at least $1 - \delta$, L outputs an hypothesis $h \in H$ such that $L_{true}(h) \leq \epsilon$, in time that is polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, M and *size(c)*.

6.1.3 Agnostic Learning

Usually the train error is not equal to zero, so the VS is empty.
In this case there is the need of bounding the gap between train and true errors.

$$L_{true}(h) \leq L_{train}(h) + \epsilon$$

Using the **Hoeffding bound**: for N i.i.d. coin flips $X_1, ..., X_N$, where $X_i \in \{0,1\}$ and $0 < \epsilon < 1$, we define the empirical mean $\bar{X} = \frac{1}{N}(X_1 + ... + X_N)$, obtaining the following bound:

$$Pr(\mathbb{E}[\bar{X}] - \bar{X} > \epsilon) \leq e^{-2N\epsilon^2}$$

Theorem Given a hypothesis space H finite, a dataset $\mathcal{D}$ with N i.i.d. samples and $0 < \epsilon < 1$:

$$Pr(\exists h \in H : L_{true}(h) - L_{train}(h) > \epsilon) \leq |H|e^{-2N\epsilon^2}$$

Now we can say that

$$L_{true}(h) \leq \underbrace{L_{train}(h)}_{\text{Bias}} + \underbrace{\sqrt{\frac{\ln|H| + \ln\frac{1}{\delta}}{2N}}}_{\text{Variance}}$$

For large $|H|$

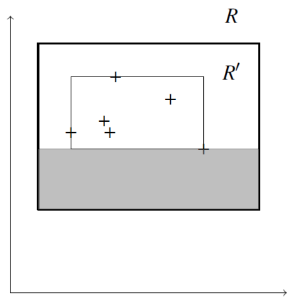
- low bias (assuming we can find a good h)
- high variance (because bound is loser)

For small $|H|$

- high bias
- low variance (tighter bound)

The bound can be used as a model selection method like cross-validation, with the benefit of using all the data but the drawback of having a pessimistic bound and that can be applied only if $|H|$ is finite.

Consider for example the case in which we want to learn an unknown target axis-aligned rectangle R and we have randomly drawn samples with a label that indicate whether the point is contained or not in R.
Consider the hypothesis corresponding to the tightest rectangle *R'* around positive samples. The error region is the difference between R and *R'*, that can be seen as the union of four rectangular regions.
In each of these regions we want an error less than $\epsilon/4$. When N samples are drawn, a bad event is when the probability of all the N samples of being outside this region is at most $(1-\epsilon/4)^N$.
The same holds for the other three regions, and so by union bound we get $4(1-\epsilon/4)^N$ and we want that the probability of a bad event is less than δ.



$$4(1-\epsilon/4)^N \leq \delta$$

By exploiting the inequality $(1-x) \leq e^{-x}$, we get:

$$N \geq (4/\epsilon)\ln(4/\delta)$$

6.2 VC-Dimension

In a continuous hypothesis space, in which $|H| = \infty$, it is not possible to use the conclusions derived from PAC-learning.
We need to get a bound error as a function of the number of points that can be completely labeled.

Definition
A **dichotomy** of a set S is a partition of S into two disjoint subsets.

For example, a positive and negative classes dichotomy is any attribution of positive/negative class over the instances.

Definition
A set of instances S is shattered by hypothesis space H if and only if for every dichotomy of S there exists some hypothesis in H consistent with this dichotomy.

Definition
The **VC dimension**, *VC(H)*, of hypothesis space H defined over instance space X is the size of the largest finite subset of X shattered by H. If arbitrarily large finite sets of X can be shattered by H, then $VC(H) \equiv \infty$.

The rule of thumb is that usually the number of parameters in the model matches the maximum number of points. But there are problems in which the number of parameters is infinite and the VC-dimension is finite or we have an hypothesis space with 1 parameter and infinite VC-dimension (e.g. a *sin* function with 2 parameters, phase and frequency, but $VC(H) = \infty$).
Based on VC-dimension the number of randomly drawn examples guaranteeing an error of at most ϵ with probability at least $(1-\delta)$ is:

$$N \geq \frac{1}{\epsilon}\left(4\log_2\left(\frac{2}{\delta}\right) + 8VC(H)\log_2\left(\frac{13}{\epsilon}\right)\right)$$

From that we can define the PAC bound using VC-dimension

$$L_{true}(h) \leq L_{train}(h) + \sqrt{\frac{VC(H)\left(\ln \frac{2N}{VC(H)} + 1\right) + \ln \frac{4}{\delta}}{N}}$$

This can be used to minimize the empirical error and the risk introduced by considering a large hypothesis space.
To prove that an hypothesis space H has $VC(H) = k$ we need to verify that $VC(H) \geq k$ by finding a shatterable instance, and to verify that $VC < k + 1$ by showing that it is not possible to find any shatterable instances.
For example, if we need to prove that $VC(H) < 4$ in a 2-D space we need to prove that the following instance cases cannot be shattered:

- 4 points aligned
- 3 points aligned and the fourth in an arbitrary position
- 4 points on a convex hull
- 3 points on a convex hull (triangle) and one point inside the hull

For the second part it is also possible to prove it by showing that a more flexible hypothesis space *H'* has $VC(H') = k$.

VC-dimension has the following properties:

Theorem
The VC dimension of a hypothesis space $|H| < \infty$ is bounded from above

$$VC(H) \leq \log_2(|H|)$$

Theorem
Concept class C with $VC(C) = \infty$ is not PAC-learnable

Chapter 7

Kernel Methods

Kernel methods are non-parametric and memory-based (eh. K-NN). It means that:

- The training data points, or a subset of them, are kept and used also during the prediction phase.
- They are fast to train and slow to predict.
- They require a metric of similarity to be defined, between two vectors of the input space.

Many linear parametric models can be re-cast into equivalent dual representations where predictions are based on a kernel function evaluated at training points.
The kernel function is given by:

$$k(x, x') = \phi(\mathbf{x})^T \phi(\mathbf{x}')$$

where $\phi(\mathbf{x})$ is a basis function.
It is a symmetric function of its arguments

$$k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$$

and it can be interpreted as similarity between **x** and **x**'.
The simplest case is for **identity mapping** in feature space: $\phi(\mathbf{x}) = \mathbf{x}$, in which case $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T\mathbf{x}'$, called **linear kernel**.
The concept of kernel allows to build interesting extensions of known algorithms by making use of the **kernel trick**.
The basic idea is that if an input vector **x** appears only in the form of scalar products, then it is possible to replace that scalar product with some choice of kernel.
Many kernel functions have the property of being a function only of the difference between the arguments

$$k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}')$$

They are called **stationary** because they are invariant to translation in input space.
An even more specific class are the **homogeneous kernels**, also known as **radial basis functions**, which depend only on the magnitude of the distance between the arguments.

$$k(\mathbf{x}, \mathbf{x}') = k(||\mathbf{x} - \mathbf{x}'||)$$

7.1 Dual representation

Many linear models for regression and classification can be reformulated in terms of dual representation in which the kernel function arises naturally.
Consider a linear regression model in which the parameters are obtained by minimizing the regularized sum-of-squares error function

$$L_{\mathbf{w}} = \frac{1}{2}\sum_{n=1}^{N}(\mathbf{w}^T\phi(\mathbf{x}_n) - t_n)^2 + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

Setting the gradient of $L_{\mathbf{w}}$ with respect to $\mathbf{w}$ equal to zero:

$$\mathbf{w} = -\frac{1}{\lambda}\sum_{n=1}^{N}(\mathbf{w}^T\phi(\mathbf{x}_n) - t_n)\phi(\mathbf{x}_n) = \sum_{n=1}^{N} a_n\phi(\mathbf{x}_n) = \Phi^T\mathbf{a}$$

The coefficients a_n are functions of $\mathbf{w}$: $a_n = -\frac{1}{\lambda}(\mathbf{w}^T\phi(\mathbf{x}_n) - t_n)$
Define the **Gram matrix** $K = \phi \times \phi^T$ an $N \times N$ matrix, with elements

$$K_{nm} = \phi(\mathbf{x}_n)^T\phi(\mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m)$$

Given N vectors, the Gram matrix is th matrix of all inner products

$$K = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \dots & k(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & \dots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}$$

Note that K is a matrix of similarities of pairs of samples (is symmetric).
Substituting $\mathbf{w} = \Phi^T\mathbf{a}$ into $L_{\mathbf{w}}$ gives

$$L_{\mathbf{w}} = \frac{1}{2}\mathbf{a}^T\Phi\Phi^T\Phi\Phi^T\mathbf{a} - \mathbf{a}^T\Phi\Phi^T\mathbf{t} + \frac{1}{2}\mathbf{t}^T\mathbf{t} + \frac{\lambda}{2}\mathbf{a}^T\Phi\Phi^T\mathbf{a}$$

where $\mathbf{t} = (t_1, ..., t_N)^T$
The sum of squares error function is written in terms of Gram matrix as

$$L_{\mathbf{a}} = \frac{1}{2}\mathbf{a}^T KK\mathbf{a} - \mathbf{a}^T K\mathbf{t} + \frac{1}{2}\mathbf{t}^T\mathbf{t} + \frac{\lambda}{2}\mathbf{a}^T K\mathbf{a}$$

Solving for $\mathbf{a}$ by combining $\mathbf{w} = \Phi^T\mathbf{a}$ and $a_n = -1/\lambda(\mathbf{w}^T\phi(\mathbf{x}_n) - t_n)$

$$\mathbf{a} = (K + \lambda\mathbf{I}_N)^{-1}\mathbf{t}$$

The solution for $\mathbf{a}$ can be expressed as a linear combination of elements of $\phi(\mathbf{x})$ whose coefficients are entirely in terms of kernel $k(\mathbf{x}, \mathbf{x}')$ from which we can recover the original formulation in terms of parameters $\mathbf{w}$.

For the prediction for a new input $\mathbf{x}$ we can substitute back $\mathbf{a}$ into the linear regression model

$$y(\mathbf{x}) = \mathbf{w}^T\phi(\mathbf{x}) = a^T\Phi\phi(\mathbf{x}) = \mathbf{k}(\mathbf{x})^T(K + \lambda\mathbf{I}_N)^{-1}\mathbf{t}$$

where $\mathbf{k}(\mathbf{x})$ has elements $k_n(\mathbf{x}) = k(\mathbf{x}_n, \mathbf{x})$, that means how much each sample is similar to the query vector $\mathbf{x}$.
Prediction is a linear combination of the target values from the training set.
The advantages of the dual representation is that the solution for $\mathbf{a}$ is expressed entirely in terms of kernel function. Once we get $\mathbf{a}$ we can recover $\mathbf{w}$ as linear combination of elements of $\phi(\mathbf{x})$ using $\mathbf{w} = \Phi^T\mathbf{a}$.
In the parametric version, the solution is $\mathbf{w}_{ML} = (\Phi^T\Phi)^{-1}\Phi^T\mathbf{t}$.
So, instead of inverting an $M \times M$ matrix we are inverting an $N \times N$ matrix, which apparently seems a disadvantage. But in this case we can work with the kernel function and therefore

- avoid working with a feature vector $\phi(\mathbf{x})$
- build a feature space with high dimensionality (even infinite)
- leverage the possibility for kernels to be defined not only over simply vectors of real numbers, but also over objects (for which a similarity measure can be defined).

7.2 Constructing kernels

To exploit the kernel trick valid kernel functions are needed.
The first method is to choose a feature space mapping $\phi(\mathbf{x})$ and use it to find the corresponding kernel.
In case of one-dimensional input space

$$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T\phi(\mathbf{x}') = \sum_{i=1}^{M}\phi_i(\mathbf{x})\phi_i(\mathbf{x}')$$

The second method is to construct kernel functions directly. In this case, we must ensure that the function we choose is a valid kernel, so, it corresponds to a scalar product in some (perhaps infinite dimensional) feature space.
For example, consider the kernel function $k(x,z) = (x^T z)^2$, in two dimensional space

$$k(x,z) = (\mathbf{x}^T z)^2 = (x_1 z_1 + x_2 z_2)^2 = x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2 = (x_1^2, \sqrt{2}x_1 x_2, x_2^2)(z_1^2, \sqrt{2}z_1 z_2, z_2^2)^T = \phi(\mathbf{x})^T \phi(\mathbf{z})$$

It is possible to test whether a function is a valid kernel without having to construct the basis function explicitly.
The necessary and sufficient condition for a function $k(\mathbf{x},\mathbf{x}')$ to be a kernel is that the Gram matrix K is positive semi-definite for all possible choices of the set $\{\mathbf{x}_n\}$. It means that $\mathbf{x}^T K \mathbf{x} \geq 0$ for non-zero vectors $\mathbf{x}$ with real entries, i.e. $\sum_n \sum_m K_{n,m} \mathbf{x}_n \mathbf{x}_m \geq 0$ for any real number $\mathbf{x}_n, \mathbf{x}_m$.

Theorem (Mercer's)
Any continuous, symmetric, positive semi-definite kernel function $k(x,y)$ can be expressed as a dot product in a high-dimensional space.

Given valid kernels $k_1(\mathbf{x},\mathbf{x}')$ and $k_2(\mathbf{x},\mathbf{x}')$ the following new kernels will be valid:

1. $k(\mathbf{x},\mathbf{x}') = ck_1(\mathbf{x},\mathbf{x}')$
2. $k(\mathbf{x},\mathbf{x}') = f(\mathbf{x})k_1(\mathbf{x},\mathbf{x}')f(\mathbf{x})$
3. $k(\mathbf{x},\mathbf{x}') = q(k_1(\mathbf{x},\mathbf{x}'))$ where *q()* is a polynomial with non-negative coefficients
4. $k(\mathbf{x},\mathbf{x}') = e^{k_1(\mathbf{x},\mathbf{x}')}$
5. $k(\mathbf{x},\mathbf{x}') = k_1(\mathbf{x},\mathbf{x}') + k_2(\mathbf{x},\mathbf{x}')$
6. $k(\mathbf{x},\mathbf{x}') = k_1(\mathbf{x},\mathbf{x}')k_2(\mathbf{x},\mathbf{x}')$
7. $k(\mathbf{x},\mathbf{x}') = k_3(\phi(\mathbf{x}),\phi(\mathbf{x}'))$ where $\phi(\mathbf{x})$ is a function from $\mathbf{x}$ to $\mathbb{R}^M$
8. $k(\mathbf{x},\mathbf{x}') = \mathbf{x}^T A \mathbf{x}'$ where A is a symmetric positive semidefinite matrix
9. $k(\mathbf{x},\mathbf{x}') = k_a(\mathbf{x}_a,\mathbf{x}'_a) + k_b(\mathbf{x}_b,\mathbf{x}'_b)$ where x_a and x_b are variables with $\mathbf{x} = (x_a, x_b)$
10. $k(\mathbf{x},\mathbf{x}') = k_a(\mathbf{x}_a,\mathbf{x}'_a)k_b(\mathbf{x}_b,\mathbf{x}'_b)$

A commonly used kernel is the Gaussian kernel:

$$k(\mathbf{x},\mathbf{x}') = e^{-\frac{||\mathbf{x}-\mathbf{x}'||^2}{2\sigma^2}}$$

It is seen as a valid kernel by expanding the square

$$||\mathbf{x}-\mathbf{x}'||^2 = (\mathbf{x}-\mathbf{x}')^T(\mathbf{x}-\mathbf{x}') = \mathbf{x}^T\mathbf{x} + \mathbf{x}'^T\mathbf{x}' - 2\mathbf{x}^T\mathbf{x}'$$

To give

$$k(\mathbf{x},\mathbf{x}') = e^{-\frac{\mathbf{x}^T\mathbf{x}}{2\sigma^2}} e^{-\frac{\mathbf{x}^T\mathbf{x}'}{\sigma^2}} e^{-\frac{\mathbf{x}'^T\mathbf{x}'}{2\sigma^2}}$$

From kernel construction rules 2 and 4, together with validity of linear kernel.

Chapter 8

Support Vector Machines

A **Support Vector Machine** (**SVM**) one of the best methods for classification and it is characterized by:

- A subset of training examples $\mathbf{x}$ (the **support vectors**)
- A vector of weights for them $\mathbf{a}$
- A similarity function $k(x, x')$ (the kernel)

The class prediction for a new example x_q with $t_i \in \{-1, 1\}$

$$f(x_q) = sign\left(\sum_{m\in\mathcal{S}} \alpha_m t_m k(x_q, x_m) + b\right)$$

where $\mathcal{S}$ is the set of indices of the support vectors. It is, hence, a smart way of doing instance-based learning.
It can be seen as a generalization of the perceptron:

$$f(\mathbf{x}_q) = sign\left[\sum_{j=1}^{M} w_j \phi_j(\mathbf{x}_q)\right]$$

but

$$w_j = \sum_{n=1}^{N} \alpha_n t_n \phi_j(\mathbf{x}_n)$$

so

$$f(\mathbf{x}_q) = sign\left[\sum_{j=1}^{M}\left(\sum_{n=1}^{N} \alpha_n t_n \phi_j(\mathbf{x}_n)\right)\phi_j(\mathbf{x}_q)\right] = sign\left[\sum_{n=1}^{N} \alpha_n t_n (\phi(\mathbf{x}_q)\phi(\mathbf{x}_n))\right]$$

The sum over the features has been rewritten as a sum over the samples.
From the perceptron, by replacing the dot product with an arbitrary similarity function a more powerful learner is obtained.

8.1 Learning SVMs

If there are multiple solutions to perfectly classify the training data set, the one that will give the smallest generalization error should be found. The support vector machine approaches this problem through the concept of **margin**, which is defined to be the smallest distance between the separating hyperplane and any of the samples.
The aim is to maximize the margin to be more robust to noise.
We assume that an hyperplane which correctly classifies the samples exists, with equation $y(\mathbf{x}_n) =$

$\mathbf{w}^T\phi(\mathbf{x}_n) + b$, where b is equivalent to the intercept w_0.
Thus, $t_n y(\mathbf{x}_n) > 0$ for all n and the distance of a point $\mathbf{x}_n$ to the decision surface is given by

$$\frac{t_n y(\mathbf{x}_n)}{||\mathbf{w}||} = \frac{t_n(\mathbf{w}^T\phi(\mathbf{x}_n) + b)}{||\mathbf{w}||}$$

The maximum margin solution is found by solving

$$\mathbf{w}^* = arg \max_{\mathbf{w},b} \left(\frac{1}{||\mathbf{w}||} \min_n (t_n(\mathbf{w}^T\phi(\mathbf{x}_n) + b)) \right)$$

The direct solution is complex and so we need to consider an equivalent problem, easier to solve.
If we make the rescaling $\mathbf{w} \to c\mathbf{w}$ and $b \to cb$ then the distance from any point $\mathbf{x}_n$ to the separating hyperplane is unchanged (**scale invariant**). Thus we can fix the margin to 1:

$$t_n(\mathbf{w}^T\phi(\mathbf{x}_n) + b) = 1$$

Now that the margin has been fixed we try to maximize $\frac{1}{||\mathbf{w}||}$ which is equivalent to minimizing $||\mathbf{w}||^2$

$$\begin{aligned} \textbf{Minimize} \quad & \frac{1}{2}||\mathbf{w}||^2 \\ \textbf{Subject to} \quad & t_n(\mathbf{w}^T\phi(\mathbf{x}_n) + b) \geq 1, \quad \text{for all } n \end{aligned}$$

This is an example of quadratic programming problem in which we re trying to minimize a quadratic function subject to a set of linear inequality constraints.
In a constraint optimization problem as the following:

$$\begin{aligned} \textbf{Minimize} \quad & f(\mathbf{w}) \\ \textbf{Subject to} \quad & h_i(\mathbf{w}) = 0, \quad \text{for } i = 1, 2, ... \end{aligned}$$

in which f and h_i are quadratic, at the solution $\mathbf{w}^*$, $\nabla f(\mathbf{w}*)$ must lie in subspace spanned by $\{\nabla h_i(\mathbf{w}^*) : i = 1, 2, ...\}$
Lagrangian function:

$$L(\mathbf{w}, \lambda) = f(\mathbf{w}) + \sum_i \lambda_i h_i(\mathbf{w})$$

The λ_is are the **Lagrange multipliers** and we need to solve $\nabla L(\mathbf{w}^*, \lambda^*) = 0$.
In case of inequality constraints:

$$\begin{aligned} \textbf{Minimize} \quad & f(\mathbf{w}) \\ \textbf{Subject to} \quad & g_i(\mathbf{w}) \leq 0, \quad \text{for } i = 1, 2, ... \\ & h_i(\mathbf{w}) = 0, \quad \text{for } i = 1, 2, ... \end{aligned}$$

The Lagrange multipliers for the inequalities are α_i. The **KKT conditions** (necessary conditions) are:

$$\begin{aligned} \nabla L(\mathbf{w}^*, \alpha^*, \lambda^*) &= 0 \\ h_i(\mathbf{w}*) &= 0 \\ g_i(\mathbf{w}^*) &\leq 0 \\ \alpha_i^* &\geq 0 \\ \alpha_i^* g_i(\mathbf{w}^*) &= 0 \end{aligned}$$

Note that either a constraint is active ($g_i(\mathbf{w}*) = 0$), and hence the point is a support vector, or its multiplier is zero ($\alpha_i^* = 0$).
Furthermore, if a new sample is obtained, the SVM has to be retrained only if it is inside the margin or on on the boundary, or if it was misclassified. In any other case the SVM doesn't need to be retrained because it would not change.

8.1.1 Primal and Dual problems

The problem over $\mathbf{w}$(weights over the features) is the primal.
After solving the equations for $\mathbf{w}$ ans substituting, the resulting problem λ (weights over the instances) is the dual, which is easier to solve instead of the primal.
Consider the Lagrangian function

$$L(\mathbf{w}, b, \alpha) = \frac{1}{2}||\mathbf{w}||_2^2 - \sum_{n=1}^{N} \alpha_n (t_n(\mathbf{w}^T \phi(\mathbf{x}_n) + b) - 1)$$

Putting the gradient w.r.t $\mathbf{w}$ and b to zero we get

$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n t_n \phi(\mathbf{x}_n), \qquad 0 = \sum_{n=1}^{N} \alpha_n t_n$$

The Lagrangian can be rewritten as follows

$$\begin{aligned} \textbf{Maximize} \quad & \tilde{L}(\alpha) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m) \\ \textbf{Subject to} \quad & \alpha_n \geq 0, \quad \text{for } n = 1, ..., N \\ & \sum_{n=1}^{N} \alpha_n t_n = 0 \end{aligned}$$

8.1.2 SVM prediction

The classification of new points with the train model is

$$y(\mathbf{x}) = sign \left(\sum_{n=1}^{N} \alpha_n t_n k(\mathbf{x}, \mathbf{x}_n) + b \right)$$

where $b = \frac{1}{N_\mathcal{S}} \sum_{n \in \mathcal{S}} \left(t_n - \sum_{m \in \mathcal{S}} \alpha_m t_m k(\mathbf{x}_n, \mathbf{x}_m) \right)$
Notice that $N_\mathcal{S}$ (the number of support vectors) is usually much smaller than N.
When the number of dimensions increases, the number of support vectors increases too, and the percentage starts to become significant (**curse of dimensionality**).
Scalability becomes an issue and this affects the generalization guarantees, which is the expected error of SVM predictions.

8.1.3 Solution techniques

It is possible to use generic quadratic programming solvers but the number of constraints depends on the number of samples, hence, specialized optimization algorithms are often used.
One example is **Sequential Minimal Optimization** (**SMO**) which updates one α_i at a time, but in this way violates the constraints $\sum_n \alpha_n t_n = 0$. It works as follows:

1. Find an example x_i that violates the KKT conditions
2. Select a second example x_j heuristically
3. Jointly optimize α_i and α_j

8.2 Handling noisy data

Now consider the case in which it is not possible to have a perfect classifier.
We have to introduce some slack variables ξ_i that allow to violate the margin constraint, but at a cost of a penalty C.

$$\begin{aligned} \textbf{Minimize} \quad & ||\mathbf{w}||_2^2 + C \sum_i \xi_i \\ \textbf{Subject to} \quad & t_i(\mathbf{w}^T x_i + b) \geq 1 - \xi_i, && \text{for all } i \\ & \xi_i \geq 0, && \text{for all } i \end{aligned}$$

Note that C is a coefficient that allows to tradeoff bias and variance, because high C means a complex solution, while a low C leads the boundary to be smoother. So, it can be chosen by cross validation.

8.2.1 Dual representation

$$\begin{aligned}
\textbf{Maximize} \quad & \tilde{L}(\alpha) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m) \\
\textbf{Subject to} \quad & 0 \leq \alpha_n \leq C, \quad \text{for } n = 1, ..., N \\
& \sum_{n=1}^{N} \alpha_n t_n = 0
\end{aligned}$$

The support vectors are points associated to $\alpha_n > 0$

- if $\alpha_n < C$ the point lies on the margin
- if $\alpha_n = C$ the point lies inside the margin and it can be either correctly classified ($\xi_i \leq 1$) or misclassified ($\xi_i > 1$)

8.3 Bounds

It is possible to define multiple types of bound on the true error, for example:

- **Margin bound**: a bound on the VC dimension which decreases with the margin. The larger the margin, the less the variance and so, the less the VC dimension. Unfortunately the bound is quite pessimistic
- **Leave-one-out bound**:
$$L_h = \frac{\mathbb{E}[\text{Number of support vectors}]}{N}$$
It is easier to compute and is related to the fact that fewer support vectors mean a stronger solution. It has the advantage of not needing to run the SVM multiple times.

Part II

Reinforcement Learning

Introduction

Reinforcement Learning is learning what actions to perform in order to maximize the cumulative rewards, which is usually a number. The learner must discover which actions yield the most reward by trying them and these actions may have long-term consequences, affecting the next situations and, through that, all the subsequent rewards. For this reason it may better to sacrifice an immediate reward to gain more long-term reward.

Agent-Environment Interface

The learner and decision maker is called the agent. The entity it interacts with is called environment. Usually it can be considered as anything the agent cannot control arbitrarily.
At each step t, the agent:

- Executes action a_t
- Receives observation o_t
- Receives scalar reward r_t

At each step t, the environment:

- Receives action a_t
- Emits observation o_t
- Emits scalar reward r_t

The history is the sequence of observations, action, rewards

$$h_t = a_1, o_1, r_1, ..., a_t, o_t, r_t$$

It influences the next action to be chosen by the agent and the observations and rewards that the environment will emit. The state is all the information that is needed to solve the sequential optimization problem, which is different by the observations, which represent what the agent is able to perceive through its sensors. So, the agent state at time step t is

$$s_t^a = f(a_1, o_1, r_1, ..., a_t, o_t, r_t) = f(h_t)$$

The environment state s_t^e is the environment's private representation, based on which the next observation/reward is produced. It is usually not visible to the agent (e.g. some card games), but even if it is, it may contain irrelevant information.
In a fully observable environment the agent directly observers the environment state ($o_t = s_t^a = s_t^e$)
Given the previous information on Reinforcement Learning, we can state that it is useful when:

- The dynamics of the environment are unknown or difficult to be modeled ($\neq$ automatic control theory)
- The model of the environment is too complex to be solved exactly, so that approximate solutions are needed

Example 1: Rubik's Cube

- State space: ~ 4.33 x 10^{19}
- Actions: 12 for each state
- State transitions: deterministic
- Rewards: -1 for each step
- Undiscounted (same reward for each step)
- Single-agent

Example 2: Blackjack

- State space: $\sim$ 800 if the sum of the cards is considered, $\sim 104,000$ if the actual combination of the cards is considered
- Actions: from 2 to 4 according to the state
- State transitions: stochastic
- Rewards: 0 for each step, {-2, -1, 0, 1, 1.5, 2} at the end
- Undiscounted
- Single agent: the dealer is deterministic and pays attention only to its cards

Example 3: Pole balancing

- State space: four continuous state variables x (position), $\dot{x}$ (speed), θ (pole's angle), $\dot{\theta}$ (pole's angular speed)
- Actions: 2 actions {-N, N}
- State transitions: deterministic (stochastic if noise in the sensors)
- Rewards: 0 in the goal region, -1 outside goal region, -100 outside feasible region
- Single-agent

Example 4: Robot navigation

- State space: robot coordinates
- Actions: moving actions
- State transitions: stochastic
- Rewards: -1 for each step
- Undiscounted/discounted
- Single-agent

Often the state cannot be observed (partially observable): shift from state *information* to *belief* $\Rightarrow$ from binary state space to continuous one (probability distribution)

Example 5: Web banner advertising

- State space: single state or multiple states (if some information about the user are known)
- Actions: one for each banner
- No dynamics: the choice of the banner doesn't influence the next user and it is not possible to influence the distribution of the states (the samples regarding the banners).
- Rewards: probability of click * cost per click

Example of trade-off *exploration* vs *exploitation*: the agent has to try new actions to make better action selections in the future but at the same time it has to select the actions that it knows will give the better rewards (but in this case it will lose some opportunities because the agent will discard some banners after too few samples to apply ones that it thinks are better).

Example 6: Chess

- State space: $\sim 10^{47}$
- Actions: from 0 to 218
- State transitions: deterministic/stochastic depending on the opponent
- Rewards: 0 each step, {-1, 0, 1} at the end
- Undiscounted
- Single-agent/Multi-agent: depends respectively on whether the opponent's policy is stationary or it changes based on the actions of the agent

With these rewards the prize is extremely delayed, so it is harder to find the causality between the actions and the outcome of the match. But on the other side, if rewards are given during the game, a bias may be introduced, and it may be a wrong bias. So, if achieving subgoals were rewarded, the agent might find a way to reach them without achieving the real goal.

Example 7: Texas Hold'em

- State space: huge and not observable
- Actions: fold, call and raise
- State transitions: deterministic/stochastic depending on the opponent
- Rewards: 0 each step, {-1, 0, 1} at the end
- Undiscounted
- Partially observable

If the opponent plays randomly, even if it's possible to see her cards, it is not possible to extract useful information from them, so the problem can be considered as fully-observable.

Chapter 9

Markov Decision Processes

Markov assumption: "the future is independent of the past given the present"

Definition
A stochastic process X_t is said to be **Markovian** if and only if

$$\mathbb{P}(X_{t+1} = j | X_t = i, X_{t-1} = k_{t-1}, ..., X_1 = k_1, X_0 = k_0) = \mathbb{P}(X_{t+1} = j | X_t = i)$$

For these processes the state captures all the information from history and once it is known, the history is not needed anymore. The state is a sufficient statistic for the future, so the present value has all the information to calculate the probability of the next value. Note that this assumption doesn't hold for partially-observable problems.
The conditional probabilities are transition probabilities and if they are stationary, we can write

$$p_{ij} = \mathbb{P}(X_{t+1} = j | X_t = i) = \mathbb{P}(X_t = j | X_0 = i)$$

Hence, it is possible to build a state transition matrix of dimension $\#states \times \#states$.

9.1 Markov Decision Processes

MDPs are a classical formalization of sequential decision making, where actions influence not just the immediate reward, but also subsequent states, and through those future rewards. Thus they need to tradeoff between immediate and delayed reward.
A Markov Decision Process is a Markov reward process with decisions (actions). It models an environment in which all states are Markovian and time is divided into stages. MDPs are applicable to problems with the following characteristics:

- finite actions
- finite state space
- single agent
- deterministic/stochastic

Definition
A Markov Process is a tuple $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu \rangle$

- $\mathcal{S}$ is a (finite) set of states
- $\mathcal{A}$ is a (finite) set of actions. $\mathcal{A}$ could be $f(\mathcal{S})$ because the actions may depend on the current state
- P is a state transition probability matrix, $P(s'|s, a)$. It is a rectangular matrix of dimension $(|\mathcal{S}| + |\mathcal{A}|) \times |\mathcal{S}|$

- R is a reward function, $R(s, a) = \mathbb{E}[r|s, a]$
- γ is a discount factor, $\gamma \in [0, 1]$
- a set of initial probabilities $\mu_i^0 = P(X_0 = i)$ for all i

9.2 Goals, rewards and return

According to the **Sutton hypothesis**, all of what we mean by goals and purposes can be well thought of as the maximization of the cumulative sum of a received scalar signal (a **reward**).
That is not always true, but it is so simple and flexible that we have to disprove it before considering anything more complicated.
Note that the reward signal is the way of communicating to the agent *what* to achieve, not *how* to achieve it.
The same goal can be specified by (infinite) different reward functions and a goal must e outside the agent's direct control. The agent must be able to measure success explicitly and frequently.
A problem is characterized by a specific time horizon, which can be either:

- **finite**: finite and fixed number of steps. In this case the horizon reduces at each step, and so every time there is a different optimization problem (non-stationary policy).
- **indefinite**: until some stopping criteria is met (**absorbing states**), e.g. Blackjack. It can last arbitrarily long but it will eventually termiante.
- **infinite**: e.g. pole balancing (ideally)

There are multiple available choices of cumulative reward:

- total reward: it may diverge, so it is not used in problems with infinite horizon.
$$V = \sum_{i=1}^{\infty} r_i$$
- average reward:
$$V = \lim_{n \to \infty} \frac{r_1 + ... + r_n}{n}$$
- discounted reward:
$$V = \sum_{i=1}^{\infty} \gamma^{i-1} r_i$$

If the values of r_i are bounded ($|r_i| \leq R$) for all i:

$$V = \sum_{i=1}^{\infty} \gamma^{i-1} r_i \leq R \sum_{i=1}^{\infty} \gamma^{i-1} = \frac{R}{1-\gamma}$$

And hence, it cannot diverge.
The **discount factor** $\gamma \in [0, 1)$ is how much value the reward will lose in 1 time-step. It depends on the definition of the problem, because it regulates how much the agent pays attention to the future. It is often mistakenly used as a trick to find a (faster) suboptimal solution.

- γ close to 0 leads to "myopic" evaluation
- γ close to 1 leads to "far-sighted" evaluation, but more difficult to compute.

γ can be also interpreted as the probability that the process will go on.

Definition
The **return** v_t is the total discounted reward from time-step t.

$$v_t = r_{t+1} + \gamma r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

The value of receiving reward r after $k + 1$ time-steps is $\gamma^k r$.

9.3 Policies and value functions

A **policy** is a mapping, at any given point in time, from states to probabilities of selecting each possible action. It decides which action the agents selects, defining its behavior.
Policies can be:

- Markovian $\subseteq$ History-dependent
- Deterministic $\subseteq$ Stochastic
- Stationary $\subseteq$ Non-stationary

The sets in the left-hand side are more specific than the ones on the right-hand side. Any MDP has at least one optimal markovian, deterministic, stationary policy.

Definition
A policy π is a distribution over actions given the state:

$$\pi(a|s) = \mathbb{P}[a|s]$$

MDP policies depend on the current state (not the history) i.e. they are stationary.
Given an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu \rangle$ and a policy π

- The state sequence $s_1, s_2, ...$ is a **Markov process** $\langle \mathcal{S}, P^\pi, \mu \rangle$
- The state and reward sequence $s_1, r_2, s_2, ...$ is a Markov reward process (**Markov Chain**) $\langle \mathcal{S}, P^\pi, R^\pi, \gamma, \mu \rangle$, where
$$P^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) P(s'|s,a) \qquad R^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) R(s,a)$$

P^π is the probability of going from s to s' following policy π . By averaging over it, the actions disappear, obtaining a matrix with dimension $|\mathcal{S}| \times |\mathcal{S}|$.
R^π is the expected reward in one step obtainable from state s, following policy π.

Given a policy π it is possible to define the utility of each state: **Policy Evaluation**.

Definition
The **state-value function** $V^\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π.

$$V^\pi(s) = \mathbb{E}_\pi[v_t | s_t = s]$$

For control purposes, rather than the values of each state, it is easier to consider the value of each action in each state.

Definition
The **action-value function** $Q^\pi(s,a)$ is the expected return starting from state s, taking action a, and then following policy π.

$$Q^\pi(s,a) = \mathbb{E}_\pi[v_t | s_t = s, a_t = a]$$

The expected value is used because the next states are not deterministic.

9.3.1 Bellman Expectation Equation

The state-value function can again be decomposed into immediate reward plus discounted value of successor state

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s') \right) \end{aligned}$$

The action-value function can similarly be decomposed

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_\pi[r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a] \\ &= R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s') \\ &= R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s', a') \end{aligned}$$

Note that by definition $V^\pi(s') \doteq \sum_{a \in A} \pi(a'|s') Q^\pi(s', a')$ and that in Q^π function no sum is present because the action is fixed.
The **Bellman expectation equation** can be expressed concisely using the induced MRP (Markov Reward Process)

$$V^\pi = R^\pi + \gamma P^\pi V^\pi$$

Note that this equation can be expressed as a system of linear equations, with both the number of unknowns and the number of equations equal to the number of states. So, if the equations are all linearly independent, the system is solvable in closed form, with direct solution

$$V^\pi = (I - \gamma P^\pi)^{-1} R^\pi$$

P^π is a stochastic matrix, so any row sums to 1. Hence, it always have at least one eigenvalue = 1 and all eigenvalues are ≤ 1 . For this reason the maximum eigenvalue of $\gamma P^\pi = 1 * \gamma = \gamma$. Finally we can say that the eigenvalues of $I - \gamma P^\pi$ are $\geq 1 - \gamma$. This means that:

- if $\gamma \leq 1$ the matrix $I - \gamma P^\pi$ is always **not singular**.
- if $\gamma = 1$ the matrix $I - \gamma P^\pi$ is always **singular**.

9.3.2 Bellman operator

Definition
The **Bellman operator** for V^π is defined as $T^\pi : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$ (maps value functions to value functions):

$$(T^\pi V^\pi)(s) = \sum_{a \in A} \pi(a|s) \left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^\pi(s') \right)$$

Using the Bellman operator, Bellman expectation equation can be compactly written as:

$$T^\pi V^\pi = V^\pi$$

The operation of updating the value function applying the Bellman operator is called **backup**.
V^π is the **unique fixed point** of the Bellman operator, so any other input will result in a vector different from V^π.
If $0 < \gamma < 1$ then T^π is a **contraction** w.r.t the maximum norm[1], hence, applying it ∞ times, V^π will be obtained.
The same is valid for the Bellman operator for Q^π, which is defined as $T^\pi : \mathbb{R}^{|\mathcal{S}| \times |\mathcal{A}|} \to \mathbb{R}^{|\mathcal{S}| \times |\mathcal{A}|}$ (maps action-value functions to action-value functions):

$$(T^\pi Q^\pi)(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \sum_{a' \in A} \pi(a'|s') Q^\pi(s', a')$$

The properties of Bellman Operators are:

- **Monotonicity**: if $f_1 \leq f_2$ component-wise

$$T^\pi f_1 \leq T^\pi f_2 \quad , \quad T^* f_1 \leq T^* f_2$$

- **Max-Norm Contraction**: for two vectors f_1 and f_2

$$||T^\pi f_1 - T^\pi f_2||_\infty \leq \gamma ||f_1 - f_2||_\infty$$

$$||T^* f_1 - T^* f_2||_\infty \leq \gamma ||f_1 - f_2||_\infty$$

[1] max absolute value of the components

- V^π is the unique fixed point of T^π
- V^* is the unique fixed point of T^*
- For any vector $f \in \mathbb{R}^{|\mathcal{S}|}$ and any policy π, we have

$$\lim_{k\to\infty} (T^\pi)^k f = V^\pi \quad , \quad \lim_{k\to\infty} (T^*)^k f = V^*$$

Definition
The **optimal state-value function** *V*(s)* is the maximum value function over all policies

$$V^*(s) = \max_\pi V^\pi(s)$$

Definition
The **optimal action-value function** *Q*(s,a)* is the maximum action-value function over all policies

$$Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

The optimal value function specifies the best possible performance in the MDP. The goal is to find a policy that optimizes the value in all the states (or state-actions) and it is known that a single deterministic, markovian and stationary policy that achieves in all the states the optimal performance exists.

9.3.3 Optimal policy

Value functions define a partial ordering over policies

$$\pi \geq \pi' \text{ if } V^\pi(s) \geq V^{\pi'}(s), \quad \forall s \in \mathcal{S}$$

Theorem For any Markov Decision Process

- There exists an optimal policy π^* that is better than or equal to all other policies, $\pi^* \geq \pi, \quad \forall\pi$
- All optimal policies achieve the optimal value function, $V^{\pi^*}(s) = V^*(s)$
- All optimal policies achieve the optimal action-value function, $Q^{\pi^*}(s,a) = Q^*(s,a)$
- There is always a deterministic optimal policy for any MDP

A deterministic optimal policy can be found by maximizing over $Q^*(s,a)$

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = arg\max_{a\in\mathcal{A}} Q^*(s,a) \\ 0 & \text{otherwise} \end{cases}$$

9.3.4 Bellman Optimality Equation

The **Bellman Optimality Equation** for *V** is:

$$\begin{aligned} V^*(s) &= \max_{a\in\mathcal{A}} Q^*(s,a) \\ &= \max_{a\in\mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V^*(s') \right\} \end{aligned}$$

The Bellman Optimality Equation for *Q** is:

$$\begin{aligned} Q^*(s,a) &= R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V^*(s') \\ &= R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a) \max_{a'\in\mathcal{A}} Q^*(s',a') \end{aligned}$$

Th Bellman optimality equation is not linear, so a closed form solution for the general case does not exist.
There are many iterative solution methods:

- Dynamic Programming
  - Policy Iteration
  - Value Iteration
- Linear Programming
- Reinforcement Learning
  - Q-learning
  - SARSA

Chapter 10

Solving Markov Decision Processes

10.1 Policy Search

Solving an MDP means finding an optimal policy.
A naive approach consists of:

- enumerating all the deterministic Markov policies
- evaluating each policy
- return the best one

The number of policies is exponential: $|\mathcal{A}|^{|\mathcal{S}|}$

10.2 Dynamic Programming

The term **Dynamic Programming** (**DP**) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP. It is a method for solving complex problems by recursively solving subproblems and then combining their solutions.
Dynamic Programming is a very general solution method for problems which have two properties:

- **Optimal substructure**
  - Principle of optimality applies
  - Optimal solutions can be decomposed into subproblems (in MDP is given by the Bellman equation)
- **Overlapping subproblems**
  - Subproblems recur many times
  - Solutions can be cached and reused (in MDP is given by the value function)

Dynamic Programming assumes full knowledge of the MDP, of which both the state transition model and the reward function are known. It is used for planning in an MDP.
Prediction

- Input: MDP $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu \rangle$ and policy π (i.e. MRP $\langle \mathcal{S}, P^\pi, R^\pi, \gamma, \mu \rangle$)
- Output: value function V^π

Control

- Input: MDP $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu \rangle$
- Output: optimal value function V^* and optimal policy π^*

For finite-horizon MDPs it is possible to use backward induction:
Backward recursion

$$V_k{}^*(s) = \max_{a \in \mathcal{A}_k} \left\{ R_k(s,a) + \sum_{s' \in \mathcal{S}_{k+1}} P_k(s'|s,a) V^*_{k+1}(s') \right\}, \quad k = N-1, ..., 0$$

$V^*_k(s)$ is the value function for each time step k (non-stationary policy) and $V^*_N = 0$.
Optimal policy

$$\pi^*_k \in arg \max_{a \in \mathcal{A}_k} \left\{ R_k(s,a) + \sum_{s' \in \mathcal{S}_{k+1}} P_k(s'|s,a) V^*_{k+1}(s') \right\}, k = 0, ..., N-1$$

With a cost of $N|\mathcal{S}||\mathcal{A}|$ vs $|\mathcal{A}|^{N|\mathcal{S}|}$ of brute force policy search, where N is the number of steps.
From now on we will focus on infinite-horizon MDPs.

10.2.1 Policy Iteration

The policy iteration method can be decoupled into:

- **policy evaluation**: we compute the state-value function V^π for a given policy π
- **policy improvement**: we change the policy according to the newly estimated values

Policy evaluation

An iterative policy evaluation is performed by applying the Bellman expectation backup an infinite number of times. A full policy-evaluation backup:

$$V_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V_k(s') \right]$$

Applying a backup operation to each state is called **sweep**. Using synchronous backups: at each iteration *k + 1*, for all states[1] $s \in \mathcal{S}$, update $V_{k+1}(s)$ from $V_k(s')$.
After few iterations even if the optimal value function is not determined, the optimal policy has usually already converged, because it depends on the shape of V, not on its absolute value. So, instead of using the closed form solution, which is expensive, applying the iteration for a few steps allows to have a bad approximation of the value function, but a good estimation of the policy.

Policy improvement

For a given states s, would it be better to do an action $a \neq \pi(s)$?
We can improve the policy by acting greedily

$$\pi'(s) = arg \max_{a \in \mathcal{A}} Q^\pi(s,a)$$

This improves the value from any state s over one step

$$Q^\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} Q^\pi(s,a) \geq Q^\pi(s, \pi(s)) = V^\pi(s)$$

Theorem: Policy improvement theorem
Let π and π' be any pair of deterministic policies such that

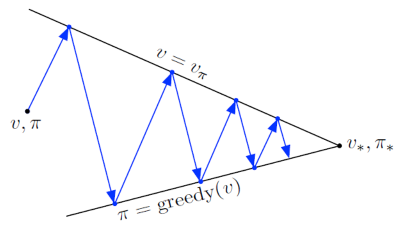
$$Q^\pi(s, \pi'(s)) \geq V^\pi(s) \quad , \quad \forall s \in \mathcal{S}$$

Then the policy π' must be as good as, or better than π

$$V^{\pi'}(s) \geq V^\pi(s) \quad , \quad \forall s \in \mathcal{S}$$

[1]A possible speedup may obtained by updating only the states that vary a lot

Proof



$$\begin{aligned} V^\pi(s) &\le Q^\pi(s, \pi'(s)) = \mathbb{E}_{\pi'}[r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s] \\ &\le \mathbb{E}_{\pi'}[r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s] \\ &\le \mathbb{E}_{\pi'}[r_{t+1} + \gamma r_{t+2} + \gamma^2 Q^\pi(s_{t+2}, \pi'(s_{t+2})) | s_t = s] \\ &\le \mathbb{E}_{\pi'}[r_{t+1} + \gamma r_{t+2} + \ldots | s_t = s] = V^{\pi'}(s) \end{aligned}$$

If the improvement stops ($V^{\pi'} = V^\pi$):

$$Q^\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} Q^\pi(s, a) = Q^\pi(s, \pi(s)) = V^\pi(s)$$

But this is the Bellman optimality equation, therefore $V^\pi(s) = V^{\pi'}(s) = V^*(s) \quad \forall s \in \mathcal{S}$, so π is an optimal policy.
It is possible to introduce a stopping condition, e.g. ϵ-convergence of the value function or stop after k iterations

10.2.2 Value Iteration

In this case the problem of finding the optimal policy π is solved by iteratively applying the Bellman optimality equation, without any explicit policy. In fact, intermediate value functions may not correspond to any policy.
Define the max-norm: $||V||_\infty = \max_{s \in \mathcal{S}} |V(s)|$

Theorem
Value Iteration converges to the optimal state-value function $\lim_{x \to \infty} V_k = V^*$
Proof

$$||V_{k+1} - V^*||_\infty = ||T^* V_k - T^* V^*||_\infty \le \gamma ||V_k - V^*||_\infty \le \ldots \le \gamma^{k+1} ||V_0 - V^*||_\infty \to \infty$$

Theorem

$$||V_{i+1} - V_i||_\infty < \epsilon \Rightarrow ||V_{i+1} - V^*||_\infty < \frac{2\epsilon\gamma}{1 - \gamma}$$

10.2.3 Synchronous Dynamic Programming Algorithms

Problem	**Bellman Equation**	**Algorithm**
Prediction	Bellman Expectation Equation	Policy Evaluation (Iterative)
Control	Bellman Expectation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

Finding the optimal policy is polynomial in the number of states for fixed-discounted MDPs, but it often grows exponentially with the number of state variables (**curse of dimensionality**). Asynchronous DP can be applied to larger problems, and appropriate for parallel computation.
Value Iteration: $O(|\mathcal{S}|^2 |\mathcal{A}|)$ for each iteration.
Policy Iteration: cost of policy evaluation + cost of policy iteration
Each iteration of PI is computationally more expensive than each iteration of VI, but PI typically requires fewer iterations to converge.

10.3 Infinite Horizon Linear Programming

Recall, at value iteration convergence we have

$$\forall s \in \mathcal{S}: \quad V^*(s) = \max_{a \in \mathcal{A}} \left\{ R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^*(s') \right\}$$

LP formulation to find V^*:

$$\begin{aligned}
&\min_V \sum_{s \in \mathcal{S}} \mu(s) V(s) \\
&\text{s.t.} \quad V(s) \geq R(s,a) + \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s'), \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}
\end{aligned}$$

$|\mathcal{S}|$ variables and $|\mathcal{S}||\mathcal{A}|$ constraints.

Theorem
V^* is the solution of the above LP.

Proof
Let T^* be the optimal Bellman operator, then the LP can be written as:

$$\begin{aligned}
&\min_V \mu^T V \\
&\text{s.t.} \quad V \geq T^*(V)
\end{aligned}$$

1. Monotonicity property: if $U \geq V$ then $T^*(U) \geq T^*(V)$
2. Hence, if $V \geq T^*(V)$ then $T^*(V) \geq T^*(T^*(V))$, and by repeated application, $V \geq T^*(V) \geq T^{*2} \geq T^{*3}(V) \geq ... \geq T^{*\infty}(V) = V^*$
3. Any feasible solution of the LP must satisfy $V \geq T^*(V)$, and hence must satisfy $V \geq V^*$
4. Hence, assuming all entries μ are positive, V^* is the optimal solution for the LP

10.3.1 Dual Linear Program

$$\begin{aligned}
&\max_\lambda \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \lambda(s,a) R(s,a) \\
&\text{s.t.} \quad \sum_{a' \in \mathcal{A}} \lambda(s', a') = \mu(s) + \gamma \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \lambda(s,a) P(s'|s,a), \quad \forall s' \in \mathcal{S} \\
&\qquad \lambda(s,a) \geq 0, \forall s \in \mathcal{S}, a \in \mathcal{A}
\end{aligned}$$

where $\lambda(s,a) = \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(s_t = s, a_t = a)$
The first equation maximizes the expected discounted sum of rewards
Optimal policy: $\pi^*(s) = arg \max_{a \in \mathcal{A}} \lambda(s,a)$
LP worst-case convergence guarantees are better than those of DP methods, but LP methods become impractical at much smaller number of states than DP methods do.

Chapter 11

RL in finite MDPs

11.1 Monte Carlo Methods

Now, we do not assume complete knowledge of the environment (**model-free**), so, the MDP is unknown but it is possible to interact with it.
We can distinguish between:

- **Model-free Prediction**: estimate the value function of an unknown MDP (MDP + policy)
- **Model-free Control**: optimize the value function of an unknown MDP

Monte Carlo (MC) methods can be used in this situation, because they can learn directly from episodes of experience - sample sequences of states, action and rewards from actual or simulated interaction with the environment.
We assume that experience is divided into episodes, and that all episodes eventually terminate no matter what actions are selected. MC learns from **complete** episodes, so value estimates and policies are changed only on the completion of an episode. This limits the application of MC to only **episodic** MDPs, in which all episodes must terminate. An important fact about MC methods is that the estimates of each state are **independent**. The estimate for one state does not build upon the estimate of any other state, as in the case of DP (**no bootstrap**).
MC can be used for prediction:

- Input: Episodes of experience $\{s_1, a_1, r_2, ..., s_T\}$ generated by following policy π in given MDP
- or: Episodes of experience $\{s_1, a_1, r_2, ..., s_T\}$ generated by MRP
- Output: Value function V^π

Or for control:

- Input: Episodes of experience $\{s_1, a_1, r_2, ..., s_T\}$ in given MDP
- Output: Optimal value function V^*
- Output: Optimal policy π^*

11.2 Model-free Prediction

11.2.1 Monte Carlo Prediction

A way to estimate the state-value function for a given policy from experience is simply by averaging the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value.
Let X be a random variable with mean $\mu = \mathbb{E}[x]$ and variance $\sigma^2 = Var[X]$. Let $x_i \sim X, i = 1, ..., n$ be n i.i.d realizations of X.
Empirical mean of X:

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} x_i$$

We have $\mathbb{E}[\hat{\mu}_n] = \mu$, $Var[\hat{\mu}_n] = \frac{Var[X]}{n}$
Recall,

- Weak law of large numbers: $\hat{\mu}_n \xrightarrow{a.s.} \mu(\lim_{n\to\infty} \mathbb{P}(|\hat{\mu}_n - \mu| > \epsilon) = 0)$
- Strong law of large numbers: $\hat{\mu}_n \xrightarrow{P} \mu(\mathbb{P}(\lim_{n\to\infty} \hat{\mu}_n = \mu) = 1)$

So, Monte Carlo policy evaluation uses empirical mean return instead of expected return and it can be computed with two different approaches:

- **First-visit MC**: average returns only for the first time s is visited (**unbiased** estimator) in an episode
- **Every-visit MC**: average returns for every time s is visited (**biased** but **consistent**[1] estimator)

The mean $\hat{\mu}_1, \hat{\mu}_2, ...$ of a sequence $x_1, x_2, ...$ can be computed incrementally

$$\begin{aligned}
\hat{\mu}_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\
&= \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) \\
&= \frac{1}{k}(x_k + (k-1)\hat{\mu}_{k-1}) \\
&= \hat{\mu}_{k-1} + \frac{1}{k}(x_k - \hat{\mu}_{k-1})
\end{aligned}$$

Update $V(s)$ incrementally after an episode. For each state s_t with return v_t

$$N(s_t) \leftarrow N(s_t) + 1$$

$$V(s_t) \leftarrow V(s_t) + \frac{1}{N(s_t)}(v_t - V(s_t))$$

where $N(s_t)$ is the number of times s_t has been visited.
In non-stationary problems, it is useful to track a running mean, i.e. to forget old episodes (but it may not converge to the true mean anymore)

$$V(s_t) \leftarrow V(s_t) + \alpha(v_t - V(s_t))$$

Note that the time required to estimate one state does not depend on the total number of states.

11.2.2 Stochastic approximation

Let X be a random variable in [0, 1] with mean $\mu = \mathbb{E}[X]$. Let $x_i \sim X, i = 1, ..., n$ be n i.i.d realizations of X.
Consider the estimator (**exponential average**)

$$\mu_i = (1 - \alpha_i)\mu_{i-1} + \alpha_i x_i$$

with $\mu_1 = x_1$ and α_1 are step-size parameters or learning rates.
If $\sum_{i\geq 0} \alpha_1 = \infty$ and $\sum_{i\geq 0} \alpha_i^2 < \infty$, then $\hat{\mu}_n \xrightarrow{a.s.} \mu$, i.e. the estimator $\hat{\mu}_n$ is consistent.

11.3 Temporal Difference Learning

Similarly to MC, **Temporal Difference** (**TD**) methods learn directly from episodes of experience and are model-free too. The main difference is that TD learns from incomplete episodes, so, it relies on previous estimates (**bootstrapping**) and it updates a guess towards a guess.
The goal is to learn V^π online from experience under policy π.

[1] The estimate eventually converges in probability to the real value

The simplest temporal-difference learning algorithm is TD(0). It updates $V(s_t)$ towards the estimated return $r_{t+1} + \gamma V(s_{t+1})$

$$V(s_t) \leftarrow V(s_t) + \alpha(\overbrace{\underbrace{r_{t+1} + \gamma V(s_{t+1})}_{\text{TD target}} - V(s_t)}^{\text{TD error } \delta_t})$$

11.3.1 Comparison between MC and TD

TD can learn online after every step, before knowing the final outcome, while MC must wait until the end of the episode before return is known.
Furthermore, TD can learn even without the final outcome at all, as in incomplete sequences of in continuing (non-terminating) environments. MC can only learn from complete sequences and from episodic envornments.
MC has high variance, zero bias:

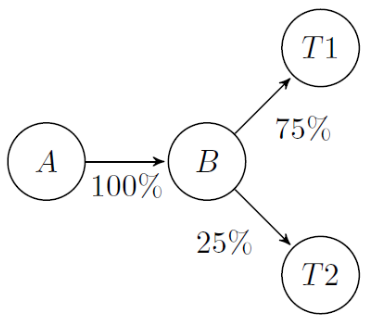
- Good convergence properties
- Works well with function approximation
- Not very sensitive to initial value
- Very simple to understand and use

TD has low variance, some bias:

- Usually more efficient than MC
- TD(0) converges to $V^{\pi}(s)$
- Problem with function approximation
- More sensitive to initial values

Suppose there is available only a finite amount of experience (in terms of episodes or time steps). In this case, a common approach is to present the experience repeatedly until the method converges. Given an approximate V, the increments are computed for every time step t at which a nonterminal state is visited, but the value function is changed only once, by the sum of all the increments.
Then all the available experience is processed again with the new value function, and so on, until it converges. This process is called **batch updating**.
Under batch updating, TD(0) converges deterministically to a single answer independent of α, as long as it is chosen to be sufficiently small. The MC constant α method also converges under the same conditions, but to a different answer. Let's make an example.
Suppose there are two states A and B and the following 8 episodes are observed: The optimal value for

$A, 0, B, 0$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 0$



V(B) is $\frac{3}{4}$, because six out of the eight times in state B the process terminated immediately with return 1, and the rest of the times with 0.
The optimal value for the estimate *V(A)* can be computed in two ways:

- Observe that 100% of the times the process was in state A it traversed immediately to B (with a reward of 0), so A must have value $\frac{3}{4}$ as well. The Markov process for this first approach is shown above and that's the answer that batch T(0) gives.

$$\hat{P}(s'|s,a) = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T} 1(s_t^k, a_t^k, s_{t+1}^k = s, a, s')$$

$$\hat{R}(s,a) = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T} 1(s_t^k, a_t^k = s, a) r_t^k$$

- Observe that we have seen A once and the return that followed it was 0; we therefore estimate *V(A)* as 0. This is the answer that batch Monte Carlo methods give.

$$\sum_{k=1}^{K} \sum_{t=1}^{T} (v_t^k - V(s_t^k))^2$$

Notice that the second answer gives minimum squared error on the training data (actually gives zero). So, MC best fits to the observed returns. But we expect that the first answer till produce lower error on future data. Finally we can say that TD, differently from MC, exploits the Markov property, so it is usually more efficient in Markov environments, while MC is more efficient in non-markovian environments.

11.4 n-step TD Prediction

Bootstrapping works best if it is over a length of time in which a significant and recognizable state change as occurred. n-step methods enable bootstrapping to occur over multiple steps and methods in which the temporal difference extends over n steps are called ***n*-step TD methods**.
Consider the following n-step returns from $n = 1, 2, ..., \infty$:

$$\begin{aligned}
&n = 1 \quad (TD) && v_t^{(1)} = r_{t+1} + \gamma V(s_{t+1}) \\
&n = 2 && v_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2}) \\
&\vdots \\
&n = \infty \quad (MC) && v_t^{(\infty)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{T-1} r_T
\end{aligned}$$

Define the n-step return:

$$v_t^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n})$$

The n-step temporal-difference learning is:

$$V(s_t) \leftarrow V(s_t) + \alpha(v_t^{(n)} - V(s_t))$$

It is also possible to average n-step returns over different n, to combine information from two different time-steps.
The **λ-return** v_t^λ combines all n-step return $v_t^{(n)}$, each weighted proportional to λ^{n-1}, and is normalized by a factor of $1 - \lambda$ to ensure that the weights sum to 1.

$$v_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} v_t^{(n)}$$

The approach considered so far is what is called, the theoretical, or *forward*, view of a learning algorithm. For each state visited, we look forward in time to all the future rewards and decide how best to combine them. In this case the episodes need to be complete as in MC.
To overcome this problem the backward view **TD(λ)** is introduced, improving over the off-line λ-return algorithm in three ways.

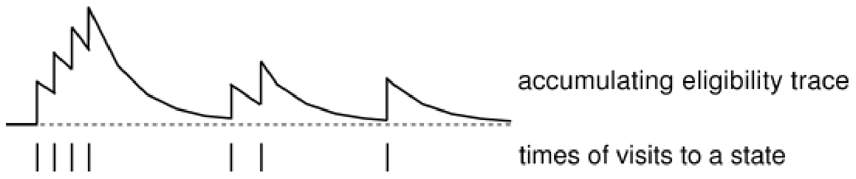
- It updates the weight vector on every step of an episode rather than only at the end, and thus its estimates may be better sooner.

- Its computations are equally distributed in time rather than all at the end of the episode.
- It can be applied to continuing or incomplete episode problems rather than just complete episode problems.

The **eligibility trace** is a short-term memory vector $\mathbf{e}_t \in \mathbb{R}^d$. When a state is visited, the corresponding component of $\mathbf{e}_t$ is bumped up and than begins to fade away. So, for a given state s, the update of $V(s)$ will occur in proportion to the TD-error δ_t and to the eligibility trace $e_t(s)$. The **trace-decay** parameter λ determines the rate at which the trace falls.

$$\begin{aligned} e_0(s) &= 0 \\ e_t(s) &= \gamma\lambda e_{t-1}(s) + \mathbf{1}(s = s_t) \\ V(s) &\leftarrow V(s) + \alpha\delta_t e_t(s) \end{aligned}$$

The eligibility trace keeps track of which states have contributed, positively or negatively, to recent state valuations, by combining both **frequency heuristics** (assign credit to the most frequent states) and **recency heuristics** (assign credit to the most recent states). When $\lambda = 0$, only the current state is



updated

$$\begin{aligned} e_t(s) &= \mathbf{1}(s = s_t) \\ V(s) &\leftarrow V(s) + \alpha\delta_t e_t(s) \end{aligned}$$

Which is exactly equivalent to TD(0) update

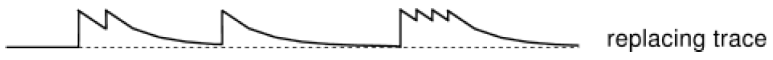
$$V(s_t) \leftarrow V(s_t) + \alpha\delta_t$$

When $\lambda = 1$, the sum of TD errors telescopes into MC error.

$$\begin{aligned} &\delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + ... + \gamma^{T-t}\delta_{T-1} \\ &= r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \\ &+ \gamma r_{t+2} + \gamma^2 V(s_{t+2}) - \gamma V(s_{t+1}) \\ &+ \gamma^2 r_{t+3} + \gamma^3 V(s_{t+3}) - \gamma^2 V(s_{t+2}) \\ &\quad\vdots \\ &+ \gamma^{T-1} r_{t+T} + \gamma^T V(s_{t+T} - \gamma^{T-1} V(s_{t+T-1}) \\ &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{T-1} r_{t+T} - V(s_t) \\ &= v_t - V(s_t) \end{aligned}$$

TD(1) is roughly equivalent to every-visit Monte Carlo. The error is accumulated online, step-by-step but the value function is only updated offline at the end of the episode.
Using accumulating traces, frequently visited states can have eligibilities grater that 1, causing problems of convergence. For this reason **replacing traces** can be introduced: when a state is visited, instead of adding 1, the trace is set to 1.

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ 1 & \text{if } s = s_t \end{cases}$$



11.5 Model-Free Control

Model-free control can solve problems in which the MDP model is unknown, but experience can be sampled, or the MDP model is known but it is too big to use, except by samples.
All learning control methods face a dilemma: they seek to learn action values for an optimal behavior by acting in a sub-optimal way in order to explore all actions. How can they learn about the optimal policy while behaving according to an exploratory policy? Two kinds of learning can be distinguished:

- **On-policy learning**: learn about policy π from experience sampled from π. So, it learn action values for a near-optimal policy that still explores.
- **Off-policy learning**: learn about policy π from experience sampled from $\bar{\pi}$. So the **target policy** is different from the one used to interact with the environment (**behavioral policy**)

11.5.1 On-Policy Learning

Monte Carlo Control

In model-free control, the first step (evaluation) of policy iteration is performed over *Q(s,a)* instead of *V(s)* because the former does not require the model of the MDP. Hence, instead of:

$$\pi'(s) = arg \max_{a \in \mathcal{A}} \{R(s,a) + P(s'|s,a)V(s')\}$$

we use:

$$\pi'(s) = arg \max_{a \in \mathcal{A}} Q(s,a)$$

Considering an example in which the first reward given to an action is, just by chance, equal to zero. It is probable that most of the future rewards for the other actions will be higher and so their value functions will be higher as well. By applying a pure greedy policy the first action will not be explored anymore.
To prevent this situation, one of the solutions to ensure continual exploration is to use ϵ-greedy exploration, in which all m actions are tried with non-zero probability: with probability 1 - ϵ the greedy action is chosen; with probability ϵ a random action is chosen.

$$\pi(s,a) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon & \text{if } a^* = arg \max_{a \in \mathcal{A}} Q(s,a) \\ \frac{\epsilon}{m} & \text{otherwise} \end{cases}$$

Theorem
For any ϵ-greedy policy π, the ϵ-greedy policy π' with respect to Q^π is an improvement

$$\begin{aligned} Q^\pi(s,\pi'(s)) &= \sum_{a \in \mathcal{A}} \pi'(a|s) Q^\pi(s,a) \\ &= \frac{\epsilon}{m} \sum_{a \in \mathcal{A}} Q^\pi(s,a) + (1-\epsilon) \max_{a \in \mathcal{A}} Q^\pi(s,a) \\ &\geq \frac{\epsilon}{m} \sum_{a \in \mathcal{A}} Q^\pi(s,a) + (1-\epsilon) \sum_{a \in \mathcal{A}} \frac{\pi(a|s) - \frac{\epsilon}{m}}{1-\epsilon} Q^\pi(s,a) \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) Q^\pi(s,a) = V^\pi(s) \end{aligned}$$

Therefore from policy improvement theorem, $V^{\pi'(s)} \geq V^\pi(s)$.
Learning policies are labeled as **GLIE** (Greedy in the Limit of Infinite Exploration) if they satisfy the following conditions:

- All state-action pairs are explored infinitely many times

$$\lim_{k \to \infty} N_k(s,a) = \infty$$

- The policy converges on a greedy policy

$$\lim_{k\to\infty} \pi_k(a|s) = \mathbf{1}(a = arg\max_{a'\in\mathcal{A}} Q_k(s', a'))$$

Sample k^{th} episode using $\pi : \{s_1, a_1, r_2, ..., s_T\} \sim \pi$
For each state s_t and a_t in the episode,

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{1}{N(s_t, a_t)}(v_t - Q(s_t, a_t))$$

Improve the policy based on new action-value function

$$\epsilon \leftarrow \frac{1}{k}$$

$$\pi \leftarrow \epsilon\text{-greedy}(Q)$$

Theorem
GLIE Monte Carlo control converges to the optimal action-value function, $Q(s, a) \to Q^*(s, a)$

The are three main time scales:

- Behavioral time scale $\frac{1}{1-\gamma}$ (discount factor)
- Sampling in the estimation of the Q-function α (learning rate)
- Exploration ϵ (e.g. for ϵ-greedy strategy)

SARSA algorithm

An On-Policy learning algorithm is **SARSA** (which stands for State, Action, Reward, State, Action). It uses the following update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$$

This update is done after every transition from a nonterminal state *s*. If *s'* is terminal, then *Q(s', a')* is zero.

Theorem
SARSA converges to the optimal action-value function, $Q(s, a) \to Q^*(s, a)$, under the following conditions:

- *GLIE sequence of policies $\pi_t(s, a)$*
- *Robbins-Monro sequence of step-sizes α_t*

$$\sum_{t=1}^{\infty} \alpha_t = \infty$$

$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

A **SARSA(λ)** algorithm can be formalized by introducing the eligibility trace in the update rule as done for the MC algorithm.

11.5.2 Off-Policy Learning

Having two distinct policies is important to be able to:

- learn from observing humans or other agents
- re-use experience generated from old policies
- learn about optimal policy while following an exploratory policy
- learn about multiple policies while following one policy

Almost all off-policy methods make use of **importance sampling**, a general technique to estimate the expectation of a different distribution w.r.t. the distribution used to draw samples.

$$\begin{aligned}\mathbb{E}_{x\sim P}[f(x)] &= \sum P(x)f(x)\\ &= \sum Q(x)\frac{P(x)}{Q(x)}f(x)\\ &= \mathbb{E}_{x\sim Q}\left[\frac{P(s)}{Q(x)}f(x)\right]\end{aligned}$$

For Monte Carlo algorithm the following process is performed:

1. The returns generated from $\bar{\pi}$ are used to evaluate π.
2. The return v_t is weighted according to the similarity between policies.
3. Multiply the importance sampling corrections along the whole episode
$$v_t^{\mu} = \frac{\pi(a_t|s_t)\pi(a_{t+1}|s_{t+1})}{\bar{\pi}(a_t|s_t)\bar{\pi}(a_{t+1}|s_{t+1})}\cdots\frac{\pi(a_T|s_T)}{\bar{\pi}(a_T|s_T)}v_t$$
4. Update value towards the corrected return
$$Q(s_t,a_t)\leftarrow Q(s_t,a_t)+\alpha(v_t-Q(s_t,a_t))$$

Note that it is not possible to use it if $\bar{\pi}$ is zero where π is non-zero and that importance sampling can dramatically increase the variance.

Importance sampling can be applied also with SARSA, by using TD targets generated from π to evaluate $\bar{\pi}$.
The TD target $r+\gamma Q(s',a')$ is weighted according to the similarity between policies and in this case only a single importance sampling correction is needed:

$$Q(s_t,a_t)\leftarrow Q_{s_t,a_t}+\alpha\left(r_{t+1}+\gamma\frac{\pi(a|s)}{\bar{\pi}(a|s)}Q(s_{t+1},a_{t+1})-Q(s_t,a_t)\right)$$

Compared to Monte Carlo importance sampling the variance is much lower and policies only need to be similar over a singe step.

Q-Learning

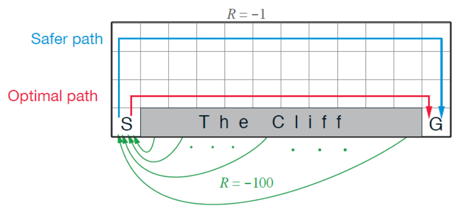
One of the early breakthroughs in reinforcement learning was the development of and off-policy TD control algorithm known as **Q-learning**, whose update rule is

$$Q(s,a)\leftarrow Q(s,a)+\alpha(r+\gamma\max_{a'\in\mathcal{A}}Q(s',a')-Q(s,a))$$

In this case, the algorithm learns about the optimal policy $\pi=\pi^*$ from experience sampled from the behavior policy $\bar{\pi}$.
The learned action-value function Q directly approximates Q^*, the optimal one.
The behavior policy can depend on *Q(s,a)*: e.g. $\bar{\pi}$ could be ϵ-greedy with respect to *Q(s,a)* or as

$Q(s,a) \to Q^*(s,a)$, the behavior policy improves.

To understand the differences between SARSA and Q-learning, and hence between on-policy and off-policy learning, the Cliffwalking problem is considered.
It is the usual gridworld with reward -1 on all transitions except those into the region marked "The Cliff", in which the reward is -100 and the agent is moved back to the start.



After an initial transient, Q-learning learns values for the optimal policy, which travels right along the edge of the cliff. This results in occasionally falling into the cliff because of the ϵ-greedy action selection. SARSA, on the other hand, takes the action selection into account and learns the longer but safer path which is far from the cliff. This results in a sub-optimal policy but with a better online performance with $\epsilon \neq 0$. But when $\epsilon \to 0$ gradually, they both converge to optimal.
It is possible to extend Q-learning with eligibility traces by marking every state-action pair as eligible and backup over non-greedy policy.
Watkins:

- Zero out the eligibility trace after a non-greedy action
- Do max when backing up at first non-greedy choice

The disadvantage of Watkins' method is that early in learning, the eligibility trace will be zeroed out frequently, resulting in little advantages. A possible solution, but difficult to implement, is the one described by **Peng**:

- Backup max action except at the end
- Never cut traces

Full Backup (DP)	**Sample backup (TD)**
Iterative Policy Evaluation $V(s) \leftarrow \mathbb{E}_\pi[r + \gamma V(s')\mid s]$	TD Learning $V(s) \overset{\alpha}{\leftarrow} r + \gamma V(s')$
Q–Policy Iteration $Q(s,a) \leftarrow \mathbb{E}_\pi[r + \gamma Q(s',a')\mid s,a]$	SARSA $Q(s,a) \overset{\alpha}{\leftarrow} r + \gamma Q(s',a')$
Q–Value Iteration $Q(s,a) \leftarrow \mathbb{E}_\pi[r + \gamma \max_{a' \in \mathcal{A}} Q(s',a')\mid s,a]$	Q–learning $Q(s,a) \overset{\alpha}{\leftarrow} r + \gamma \max_{a' \in \mathcal{A}} Q(s',a')$

Chapter 12

Multi-Armed Bandit

12.1 Introduction

During learning one of the main problems is that the value of each action $Q(a|s)$ is not precisely known. Online decision making leads us to face a fundamental choice:

- **Exploration**: gather more information from unexplored/less explored options
- **Exploitation**: select the option we consider to be the best one so far

Depending on how much we are far-sighted we might make some sacrifice in the short-term to gain more in the future.
Furthermore, in an infinite time horizon perspective we also want to gather enough information to find the best overall decision with high probability.
Two common approaches to perform exploration are the ϵ-greedy policy (which does not achieve the optimal policy) and the use of Softmax (the Boltzmann distribution):

$$\pi(a_i|s) = \frac{e^{\frac{\hat{Q}(a_i|s)}{\tau}}}{\sum_{a \in \mathcal{A}} e^{\frac{\hat{Q}(a|s)}{\tau}}}$$

Where τ is a temperature parameter which decreases over time. Softmax weights the actions according to its estimated value $\hat{Q}(a|s)$, so that there is a high probability of choosing the actions with high reward.

12.2 Multi-Armed Bandit

A **Multi-Armed Bandit** problem is the one in which you are faced repeatedly with a choice among multiple actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the selected action. The objective is to maximize the expected total reward over some finite time period. We can see the Multi-Armed Bandit setting as a specific case of an MDP:

- $\mathcal{S}$ is a set of states $\rightarrow$ single state $\mathcal{S} = \{s\}$. This means that the expectation of reward will change over time only stochastically.
- $\mathcal{A}$ is a set of actions $\rightarrow$ arms $\mathcal{A} = \{a_1, ..., a_N\}$ (finite)
- P is a state transition probability matrix $\rightarrow P(s|a_i, s) = 1, \forall a_i$. There is only one state, so, the probability of staying in the same state is always 1.
- R is a reward function $\rightarrow R(s, a_i) = R(a_i)$
- γ is a discount factor $\rightarrow \gamma = 1$ because we have a finite time horizon and so there is no chance of getting infinite rewards.
- μ^0 is a set of initial probabilities $\rightarrow \mu^0(s) = 1$

The only thing needed to have the full definition of the problem is the characterization of the reward, which can be:

- Deterministic (trivial)
- Stochastic: the reward is sampled from a random variable
- Adversarial: the opponent can choose the reward

12.3 Stochastic MAB

A Multi-Armed Bandit setting is a tuple $\langle \mathcal{A}, \mathcal{R} \rangle$:

- $\mathcal{A}$ is a set of N possible arms (choices)
- $\mathcal{R}$ is a set of unknown distributions $r_{i,t} \sim \mathcal{R}(a_i)$ and $\mathbb{E}[\mathcal{R}(a_i)] = R(a_i)$. For simplicity assume $\mathcal{R}(a_i) \in [0, 1]$

The process considered is the following:

1. At each time step t the agent selects a single arm a_{i_t}
2. The environment generates a reward $r_{i_t,t}$
3. The agent updates her information by means of a history h_t

12.3.1 Regret

The final aim of the agent is to maximize the cumulative reward over a given time horizon T:

$$\sum_{t=1}^{T} r_{i_t,t}$$

and possibly converge to the option with largest expected reward if $T \to \infty$.
Assume to know the best arm a^*. Its expected reward is $R^* = R(a) = \max_{a \in \mathcal{A}} \mathbb{E}[\mathcal{R}(a)]$.
At a given time step t, the action a_{i_t} is selected, the following loss occurs:

$$\mathcal{R}(a^*) - \mathcal{R}(a_{i_t})$$

On average the algorithm loses:

$$\mathbb{E}[\mathcal{R}(a^*) - \mathcal{R}(a_{i_t})] = R^* - R(a_{i_t})$$

It is desired to minimize the expected regret suffered over a finite time horizon of T rounds, called **Expected Pseudo Regret**

$$L_T = TR^* - \mathbb{E}\left[\sum_{t=1}^{T} R(a_{i_t})\right]$$

The expected value is taken w.r.t. to the stochasticity of the reward function and the randomness of the used algorithm.
Note that the maximization of the cumulative reward is equivalent to the minimization of the cumulative regret.
Another way of reformulating the cumulative regret is by defining the average difference in reward between the generic arm a_i and the optimal one a^* as $\Delta_i := R^* - R(a_i)$.
Then define the number of times an arm a_i has been pulled after a total of t time steps as $N_t(a_i)$

$$\begin{aligned} L_T &= TR^* - \mathbb{E}\left[\sum_{t=1}^{T} R(a_{i_t})\right] \\ &= \mathbb{E}\left[\sum_{t=1}^{T} R^* - R(a_{i_t})\right] \\ &= \sum_{a \in \mathcal{A}} \mathbb{E}[N_t(a_i)](R^* - R(a_i)) = \sum_{a \in \mathcal{A}} \mathbb{E}[N_T(a_i)]\Delta_i \end{aligned}$$

i.e. we want to minimize the number of times a suboptimal arm is selected.

12.3.2 Lower Bound

The definition of regret in terms of Δ_i implies that any algorithm performance is determined by the similarity among arms. The more the arms are similar, the more the problem is difficult.

Theorem: MAB lower bound
Given a MAB stochastic problem, any algorithm satisfies:

$$\lim_{T\to\infty} L_T \geq \log T \sum_{a_i|\Delta_i>0} \frac{\Delta_i}{KL(\mathcal{R}(a_i), \mathcal{R}(a^*))}$$

where $KL(\mathcal{R}(a_i), \mathcal{R}(a^*))$ is the Kullback-Leibler divergence between the two distributions $\mathcal{R}(a_i)$ and $\mathcal{R}(a^*)$

12.3.3 Pure Exploitation algorithm

A pure exploitation algorithm can be formulated, which always selects the action s.t. $a_{i_t} = arg\max_a \hat{R}_t(a)$ where the expected reward for an arm is:

$$\hat{R}_t(a_i) = \frac{1}{N_t(a_i)} \sum_{j=1}^{t} r_{i,j} \mathbf{1}\{a_i = a_{i_j}\}$$

This approach might not converge to the optimal action and the uncertainty corresponding to the $\hat{R}_t(a)$ is not considered.
There are two formulations to face this problem:

- Frequentist: $R(a_1), ..., R(a_N)$ are unknown parameters and a policy selects at each time step an arm based on the observation history.
- Bayesian: $R(a_1), ..., R(a_N)$ are random variables with prior distributions $f_1, ..., f_N$ and a policy selects at each time step an arm based on the observation history and on the provided priors.

The more we are uncertain on a specific choice, the more we want the algorithm to explore that option. We might lose some value in the current round, but it might turn out that the explored action is the best one.

12.3.4 Upper Confidence Bound Approach

Instead of using the empiric estimate we consider an upper bound $U(a_i)$ over the expected value $R(a_i)$. More formally, we need to compute an upper bound

$$U(a_i) := \hat{R}_t(a_i) + B_t(a_i) \geq R(a_i)$$

with high probability (e.g. more than 1 - $\delta, \delta \in (0, 1)$).
The bound length $B_t(a_i)$ depends on how much information we have on the arm, which is the number of times that arm has been pulled so far $N_t(a_i)$.
In order to set the upper bound we resort to the Hoeffding Bound: Let $X_1, ..., X_t$ be i.i.d random variables with support in [0, 1] and identical mean $\mathbb{E}[X_i] =: X$ and let $\bar{X}_t = \frac{\sum_{i=1}^t X_i}{t}$ be the sample mean. Then

$$P(X > \bar{X}_t + u) \leq e^{-2tu^2}$$

It is now possible to apply this inequality to the upper bounds corresponding to each arm:

$$P\left(R(a_i) > \hat{R}_t(a_i) + B_t(a_i)\right) \leq e^{-2N_t(a_i)B_t(a_i)^2}$$

To compute the upper bound:

- Pick a probability p that the real value exceeds the bound

$$e^{-2N_t(a_i)B_t(a_i)^2} = p_t$$

- Solve to find $B_t(a_i)$

$$B_t(a_i) = \sqrt{\frac{-\log p_t}{2N_t(a_i)}}$$

- Reduce the value of p over time, e.g. $p_t = t^{-4}$

$$B_t(a_i) = \sqrt{\frac{2\log t}{N_t(a_i)}}$$

- Ensure to select the optimal action as the number of samples increases

$$\lim_{t\to\infty} B_t(a_i) = 0 \Rightarrow \lim_{t\to\infty} U_t(a_i) = R(a_i)$$

12.3.5 UCB1

By exploiting the upper bound it is possible formulate the UCB1 algorithm, that for each time step:

- Computes $\hat{R}_t(a_i), \quad \forall a_i$
- Computes $B_t(a_i), \quad \forall a_i$
- Plays arm $a_{i_t} = arg\max_{a_i\in\mathcal{A}}\left(\hat{R}_t(a_i) + B_t(a_i)\right)$

Theorem: UCB1 upper bound
At finite time T, the expected total regret of the UCB1 algorithm applied to a stochastic MAB problem is:

$$L_T \leq 8\log T \sum_{i|\Delta_i>0} \frac{1}{\Delta_i} + \left(1+\frac{\pi^2}{3}\right)\sum_{i|\Delta_i>0}\Delta_i$$

12.3.6 Thompson Sampling

Thompson Sampling is a general Bayesian methodology for online learning:

1. Consider a Bayesian prior for each arm $f_1, ..., f_N$ as a starting point
2. At each round t, sample from each one of the distributions $\hat{r}_1, ..., \hat{r}_N$
3. Pull the arm a_{i_t} with the highest sampled value $i_t = arg\max_i \hat{r}_i$
4. Update the prior incorporating the new information

For example for Bernoulli rewards the prior conjugate distribution is $Beta(\alpha, \beta)$. Start from a prior $f_i(0) = Beta(1, 1)$ (uniform prior) for each arm a_i and apply the incremental update formula for the pulled arm a_i:

- In the case of a success occurs $f_i(t+1) = Beta(\alpha_t + 1, \beta_t)$
- In the case of a failure occurs $f_i(t+1) = Beta(\alpha_t, \beta_t + 1)$

Theorem: TS upper bound
At time T, the expected total regret of Thompson Sampling algorithm applied to a stochastic MAB problem is:

$$L_T \leq O\left(\sum_{i|\Delta_i>0}\frac{\Delta_i}{KL(\mathcal{R}(a_i), \mathcal{R}(a^*))}(\log T + \log\log T)\right)$$

12.4 Adversarial MAB

A Multi-Armed Bandit Adversary setting is a tuple $\langle \mathcal{A}, \mathcal{R} \rangle$

- $\mathcal{A}$ is a set of N possible arms
- $\mathcal{R}$ is a reward vector $r_{i,t}$ decided by an adversary player at each turn

At each time step t the agent selects a single arm a_{i_t} and at the same time the adversary chooses the rewards $r_{i,t}, \forall i$. Then the agent receives the reward corresponding to that arm.
The final objective is to maximize the cumulative reward over a time horizon T:

$$\sum_{t=1}^{T} r_{i_t,t}$$

12.4.1 Regret

We cannot compare the cumulated regret we gain with the optimal one. Moreover, the fact that there is an adversary choosing the regret does not allow to use deterministic algorithms.
We can define the **weak regret** as the loss of reward compared with the the best *constant* action

$$L_T = \max_i \sum_{t=1}^{T} r_{i,t} - \sum_{t=1}^{T} r_{i_t,t}$$

12.4.2 Lower bound

Theorem: minimax lower bound Let *sup* be the supremum over all distribution of rewards such that, for $i \in \{1, ..., N\}$ the rewards $r_{i,1}, ..., r_{i,2}, ..., r_{i,j} \in 0, 1$ are i.i.d, and let *inf* be the infimum over all forecasters. Then:

$$\inf \sup \mathbb{E}[L_T] \geq \frac{1}{20}\sqrt{TN}$$

where the expectation is taken with respect to both the random generation rewards and the internal randomization of the forecaster.
Note that is not really possible to compare it with the stochastic version, but it points out that this problem is more difficult.

12.4.3 EXP3

This algorithm is a variation of the Softmax algorithm.
The probability of choosing an arm is

$$\pi_t(a_i) = (1-\beta)\frac{w_t(a_i)}{\sum_j w_t(a_j)} + \frac{\beta}{N}$$

where

$$w_{t+1}(a_i) = \begin{cases} w_t(a)e^{-\eta \frac{r_{i,t}}{\pi_t(a_i)}} & \text{if } a_i \text{has been pulled} \\ w_t(a_i) & \text{otherwise} \end{cases}$$

Theorem: EXP3 upper bound At time T, the expected total regret of EXP3 algorithm applied to an adversarial MAB problem with $\beta = \eta = \sqrt{\frac{N \log N}{(e-1)T}}$ is:

$$\mathbb{E}[L_T] \leq O(\sqrt{TN \log N})$$

where the expectation is taken with respect to both the random generation rewards and the internal randomization of the forecaster.