# Machine Learning

## Introduction

- **Supervised learning**: $\mathcal{D} = \{(x, t)\} \Rightarrow t = f(x)$
  Learn from data a model (function $f$) that maps known inputs to known outputs.
  We believe that there exists the true function $f$. The aim is to find the best approximation given the data.
  - Define a **loss function** $\mathcal{L}$ *(function that tells how good the current approximation is)*
  - Choose the **hypothesis space** $\mathcal{H}$ *(family of functions that we'll inspect to find the best approximation)*
  - Find in $\mathcal{H}$ an approximation $h$ of $f$ that minimizes $\mathcal{L}$

  If $f \notin \mathcal{H}$ there will be an irreducible error. However, it's not always a good choice to enlarge $\mathcal{H}$ because we rely on a loss function which may not be the optimal (if we can design the optimal loss function then we already know the function $f$), hence we may end in a worse approximation of $f$.

  The **direct method** aims ~~is~~ to learn directly an approximation of $f$ from $\mathcal{D}$, with no probablisitc modelling (a parametrized function is defined and the parameters are optimized based on the collected data). Alternatively, the **discriminative approach** aims to model the conditional density $p(t|x)$, i.e. the probability of getting some target given some input. Then it marginalizes to find the conditional mean $\mathbb{E}[t|x] = \int t \cdot p(t|x)dx$, which represents the most likely value of $t$ given $x$. Instead, the **generative approach** aims to model the joint density $p(x, t)$, i.e. the probability of observing jointly the input and its target. From $p(x, t)$ it's possible to infer $p(t|x)$ and then $\mathbb{E}[t|x]$, which has the same meaning as before. The last approach is more complex, however it allows to do more (*e.g.* generate data).

  Generally, the elements of supervised learning algorithms are:

  - **representation**: choose the hypothesis space $\mathcal{H}$ *(how to represent the approximation function)*
  - **evaluation**: loss function $\mathcal{L}$ *(how to evaluate the approximation)*
  - **optimization**: search for the optimum $h$ given $\mathcal{H}$ and $\mathcal{L}$

- **Unsupervised learning**: $\mathcal{D} = \{x\} \Rightarrow f(x)$
  Learn previously unknown patterns and efficient data representation.
- **Reinforcement learning**: $\mathcal{D} = \{(x, a, x', r)\} \Rightarrow \pi^*(x) = \arg\max_a Q^*(x, a)$
  Given (a set of) $x$ current situation, $a$ action, $x'$ situation after the action and $r$ reward, learn the optimal policy function $\pi^*(x)$, i.e. the best action to do when the situation is $x$, based on $Q^*(x, a)$, function that provides the expected reward in the long term given the state $x$ and the action $a$.

## Linear Regression

Learn an approximation of the function $f(x)$ that maps input $x$ to a continuous output $t$ from a dataset $\mathcal{D}$.

- **How do we model $f$?**
  In linear regression $f(x)$ is a linear function, hence we can solve the optimization problem analytically.

  $y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{D-1} w_j x_j = \mathbf{w}^T \mathbf{x}$ $\quad\quad$ $\mathbf{x} = (1, x_1, .., x_{D-1})$.

  The hypothesis space $\mathcal{H}$ is the space of all the possible values of the weights $\mathbf{w} = (w_0, w_1, .., w_{D-1})$. One point in $\mathcal{H}$ represents a model. The dimension of the space is $D$, we have to consider the constant term.
- **How do we evaluate the approximation?**
  A convenient loss function is the sum of squared errors (SSE), also known as residual sum of squares (RSS):

  $L(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} (y(\mathbf{x}_n, \mathbf{w}) - t_n)^2$ $\quad$ where: $\quad \sum_{n=1}^{N} (y(\mathbf{x}_n, \mathbf{w}) - t_n)^2 = \sum_{n=1}^{N} \epsilon^2 = ||\epsilon||_2^2 := RSS(\mathbf{w})$
- **How can we improve?**
  The regression model must be linear in the parameters (weights), thus we can define a model using non-linear basis functions $\phi_j(\mathbf{x})$. The basis functions are non-linear transformations of the original input.

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

  To find the right basis functions we can exploit some previous knowledge about the problem or we can try to apply some standard basis functions (polynomial, gaussian, sigmoidal). The parameters regarding these functions (grade of the polynomial, $\mu_i$, $\sigma_i$) are design choices. The final choice is made comparing the perfomances.
- **How do we find the best w?**
  Ordinary Least Squares (OLS) is a closed-form optimization of the RSS:

  $L(\mathbf{w}) = \frac{1}{2} RSS(\mathbf{w}) = \frac{1}{2} ||\epsilon||_2^2 = \frac{1}{2} \epsilon^T \epsilon = \frac{1}{2} (\mathbf{t} - \mathbf{\Phi w})^T (\mathbf{t} - \mathbf{\Phi w})$

  where each row of $\mathbf{\Phi}$ corresponds to a sample, *e.g.* the row of the $j$-th sample is $(1, \phi_1(\mathbf{x}_j), .., \phi_{M-1}(\mathbf{x}_j))$.
  Since $L(\mathbf{w})$ is quadratic w.r.t. $\mathbf{w}$ we can compute the first and second derivative to find the optimal $\mathbf{w}$:

$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = -\mathbf{\Phi}^T(\mathbf{t} - \mathbf{\Phi}\mathbf{w}), \qquad \frac{\partial^2 L(\mathbf{w})}{\partial \mathbf{w}\mathbf{w}^t} = \mathbf{\Phi}^T\mathbf{\Phi} \qquad \Rightarrow \qquad \hat{\mathbf{w}}_{OLS} = (\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\mathbf{t}$

To perform the inversion the matrix $\mathbf{\Phi}$ cannot have 0 eigenvalues (no linear dependence among features). Hence, the output $\hat{\mathbf{t}}$ of the model is:

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

which is a linear combinations of the columns of the matrix $\mathbf{\Phi}$. The geometrical meaning of OLS is: the columns of the matrix $\mathbf{\Phi}$ generate a space on which we project the true target vector $\mathbf{t}$, obtaining $\hat{\mathbf{t}}$. $\hat{\mathbf{t}}$ is the best solution given the features. We can't help with dependencies on features which are not given or with noise.

However, if we don't have all the data in advance or if the dataset is too large to perform OLS, it is convenient to use an online learning approach (sequential learning): Least Mean Square (LMS).

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

where $\alpha$ is the learning rate: $\sum_{k=1}^{\infty}\alpha^{(k)} = +\infty$ (not too fast), $\sum_{k=1}^{\infty}\alpha^{(k)^2} < +\infty$ (not too slow).

- **Regularization**

As we add parameters (complexity) to the model we can see that their values increase in absolute value. This is because a complex model is highly-changing and so there is a need for something that changes a lot with the input → highly-valued weights. To keep the function smooth we can add a penalty term to the loss function that has the role of keeping weights as small as possible:

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

where $L_D(\mathbf{w})$ is the usual loss function, $L_W(\mathbf{w})$ accounts for the model complexity and $\lambda$ is the regularization coefficient (the higher the $\lambda$, the more important will be to keep the weights small).

Ridge regression: $L(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(\mathbf{x}_n, \mathbf{w}) - t_n)^2 + \frac{\lambda}{2}||\mathbf{w}||_2^2 \quad \Rightarrow \quad \hat{\mathbf{w}}_{ridge} = (\lambda\mathbf{I} + \mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\mathbf{t}$

Lasso regression: $L(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(\mathbf{x}_n, \mathbf{w}) - t_n)^2 + \frac{\lambda}{2}||\mathbf{w}||_1^2 \quad \Rightarrow \quad$ no closed form optimization

- **Probabilistic approach: Maximum Likelihood (ML)**

We can deal with regression in a probabilistic way: define a probabilistic model that maps inputs to outputs which includes some unknown parameters, then model the likelihood (probability of observed data given a set of parameters) and estimate the parameters by maximizing the likelihood:

$$\mathbf{w}_{ML} = \arg\max_{\mathbf{w}} p(\mathcal{D}|\mathbf{w})$$

In the case of linear regression we assume a linear model for $y(\mathbf{x}, \mathbf{w})$ and a gaussian noise:

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon = \mathbf{w}^T\phi(\mathbf{x}) + \epsilon \qquad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

Given a dataset of $N$ samples and considering all the points independent among each other we obtain:

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

by maximizing the logarithm of this quantity we end up with $\mathbf{w}_{ML} = \mathbf{w}_{OLS}$, which means that when we apply OLS we implictly do the maximum likelihood approach with assumption of linearity and $\epsilon \sim \mathcal{N}(0, \sigma^2)$.
This procedure doesn't provide any tool to get an estimate of the uncertainty level that we have on the estimates.

- **Probabilistic approach: Bayesian Linear Regression**

Another probabilistic approach is the bayesian one: define a model with some unknown parameters, capture the assumptions with the prior distribution over these parameters (before seeing the data), observe the data and use them to compute the posterior probability distributions for the parameters as:

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

Given $\mathcal{D}$ the most probable value of $\mathbf{w}$ is the mode of the posterior, i.e. the maximum a posteriori (MAP).

In the case of bayesian linear regression we assume a gaussian prior and likelihood and so, also the posterior:

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

where $\mathbf{w}_N = \mathbf{S}_N(\mathbf{S}_0^{-1}\mathbf{w}_0 + \frac{\mathbf{\Phi}^T\mathbf{t}}{\sigma^2})$ and $\mathbf{S}_N^{-1} = \mathbf{S}_0^{-1} + \frac{\mathbf{\Phi}^T\mathbf{\Phi}}{\sigma^2}$. The most neutral choice is $\mathbf{w}_0 = \mathbf{0}$ and $\mathbf{S}_0 = \sigma^2\mathbf{I}$. To represent no previous knowledge we put $\mathbf{w}_0 = \mathbf{0}$ and $\mathbf{S}_0 =$ infinity (infinitely uncertain), getting back to $\mathbf{w}_{OLS}$.
With the bayesian approach, unlikely in OLS, we get two estimates: $\mathbf{w}_N$ and $\mathbf{S}_N$, which means that we have also an estimate of the uncertainty of the weights vector. Lastly, bayesian regression allows also to account for regularization, which can be seen as specific case for which we make specific decisions for the prior.

With a bayesian framework it is possible to compute the probability distribution of the target for a new sample $\mathbf{x}^*$ by integrating over the posterior distribution:

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

In the case of bayesian linear regression we can obtain an analitical predictive version:

$$p(t|\mathbf{x}, \mathcal{D}, \sigma^2) = \mathcal{N}(t|\mathbf{w}_N^T\phi(\mathbf{x}), \sigma_N^2(\mathbf{x}))) \quad \text{where:} \quad \sigma_N^2(\mathbf{x}) = \sigma^2 + \phi(\mathbf{x})^T\mathbf{S}_N\phi(\mathbf{x})$$

# Linear Classification

Learn an approximation of the function $f(x)$ that maps input $x$ to a discrete class $C_k$ from a dataset $\mathcal{D}$.

- **How do we model $f$?**
  In linear classification we introduce generalized models: $f(x)$ partitions the input space into decision regions whose boundaries are called decision boundaries/surfaces. The decision boundaries/surfaces are linear functions of $\mathbf{x}$ and $\mathbf{w}$, they corresponds to $\mathbf{w}^T\mathbf{x} + w_0 = const$. The overall function $f(x)$ is modeled as:
  $f(\mathbf{x}, \mathbf{w}) = f(w_0 + \sum_{j=1}^{D-1} w_j x_j) = f(w_0 + \mathbf{w}^T\mathbf{x})$ ← generalized linear model
  As in linear regression, we can extend models by using a fixed set of basis function $\phi_j(\mathbf{x})$. In this way we apply a non-linear transformation to map the input space into a feature space. As results, decision boundaries that are linear in the feature space correspond to nonlinear decision boundaries in the input space.
- **How do we encode (mathematically) $C_k$?**
  A common encoding for two-classes problems is a binary encoding: $t \in \{0, 1\}$. This encoding gives a clear interpretation of $f(x)$, which is the probability of belonging to the class labeled with 1. An alternative is to consider $t \in \{-1, +1\}$. If the problem has $K$ classes a typical choice is 1-of-$K$ encoding, i.e. one-hot-encoding.
- **Least Squares for Classification**
  Consider a $K$-classes problem. Using one-hot-encoding we can model each class with a linear function:
  $y_k(\mathbf{x}) = \mathbf{w}_k^T\mathbf{x} + w_{k0} \quad \Rightarrow \quad \mathbf{y}(\mathbf{x}) = \tilde{\mathbf{W}}^T\tilde{\mathbf{x}}$
  Applying least squares to find the optimal value of $\tilde{\mathbf{W}}$ we obtain:
  $\tilde{\mathbf{W}} = (\tilde{\mathbf{X}}^T\tilde{\mathbf{X}})^{-1}\tilde{\mathbf{X}}^T\mathbf{T}$
  where $\tilde{\mathbf{X}}$ is the matrix whose $j$-th row is $\tilde{\mathbf{x}}_j^T$ and $\mathbf{T}$ is the matrix whose $j$-th row is $\mathbf{t}_j^T$.
  A new sample is mapped to class $C_k$ if $t_k > t_j \ \forall j \neq k$, where $t_k = \tilde{\mathbf{w}}_k^T\tilde{\mathbf{x}}_k$.
  The main problems with this method is that it's extremely sensitive to outliers and that we're assuming gaussian noise for the OLS computation (while in labeling there is no noise).
- **Perceptron**
  The perceptron is a linear discriminant model for two-classes problem where encoding is $\{-1, +1\}$.
  $$f(\mathbf{x}, \mathbf{w}) = \begin{cases} +1 & \text{if } \mathbf{w}^T\phi(\mathbf{x}) \geq 0 \\ -1 & \text{if else} \end{cases}$$
  The perceptron algorithm aims at finding a decision surface (separating hyperplane) by minimizing the distance of misclassified samples to the boundary. The loss function is the distance since it's a continuous function.
  $L_P(\mathbf{w}) = -\sum_{n \in \mathcal{M}} \mathbf{w}^T\phi(\mathbf{x}_n)\, t_n$
  where $\mathcal{M}$ is the set of misclassified points. The distance to the boundary is $\propto \mathbf{w}^T\phi(\mathbf{x}_n)$, we multiply it by $t_n$ to have always $\mathbf{w}^T\phi(\mathbf{x}_n)\, t_n < 0$ and then we add "$-$" to have something overall positive. The loss function can be minimized using stochastic gradient descent:
  $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \alpha\, \phi(\mathbf{x}_n)\, t_n$ $= \mathbf{w}^{(k)} - \alpha \nabla L_p(\mathbf{w})$
  The perceptron algorithm is:
  - Given $\mathcal{D} = \{\mathbf{x}_n, t_n\}$ initialize $\mathbf{w}^{(0)}$, $k = 0$
  - Repeat until convergence :
    1. $k \leftarrow k + 1$
    2. $n \leftarrow k \mod N$
    3. If $(\hat{t}_n \neq t_n) \Rightarrow \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \phi(\mathbf{x}_n)\, t_n$

  At the end, if the training data is linearly separable in the feature space $\mathbf{\Phi}$, then the perceptron learning algorithm is guaranteed to find an exact solution in a finite number of steps.
- **Probabilistic approach: Logistic Regression**
  We can model the conditional probability of a class given the input using (as assumption) a sigmoidal function:
  $p(C_1|\phi) = \frac{1}{1+exp(-\mathbf{w}^T\phi(\mathbf{x}))} = \sigma(\mathbf{w}^T\phi(\mathbf{x}))$
  This model is known as logistic regression and its meaning is the probability of the point $\phi(\mathbf{x})$ belonging to the class $C_1$. Since this model is used for two-classes problems, the probability of belonging to $C_2$ is given by $1 - C_1$. The goal is to optimize $\mathbf{w}$ in order to make this probability distribution fit as good as possible the observed data. In order to do so, we go through the likelihood function (we want to maximize the likelihood):
  $p(\mathbf{t}|\mathbf{X}, \mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n}(1 - y_n)^{1-t_n}$ where: $y_n = \sigma(\mathbf{w}^T\phi(\mathbf{x}_n))$
  A convenient loss function to minimize is the negative log-likelihood, known as cross-entropy error function:
  $L(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}) = -\sum_{n=1}^{N}(t_n \ln y_n + (1 - t_n)\ln(1 - y_n))$
  Due to the non-linearity there is no closed-form solution. However, the error function is convex and gradient-based optimization can be applied (also in online learning settings):
  $\nabla L(\mathbf{w}) = \sum_{n=1}^{N}(y_n - t_n)\,\phi(\mathbf{x}_n)$

In multiclass problems we can apply the multiclass logistic regression. In this case $p(C_k|\phi)$ is modeled by a softmax trasformation of the output of $K$ linear functions (one for each class):

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

Assuming one-hot-encoding we can obtain the cross-entropy error function and its gradient.

## Model Evaluation, Selection and Ensambles

- **Bias-Variance**
  Should we always choose the model with the lowest loss function on $\mathcal{D}$? No, the loss function may be biased towards training data. The bias-variance is a framework to analyze the performance of models.

  Suppose that data come from the (true) model $t_i = f(\mathbf{x}_i) + \epsilon$, with $\mathbb{E}[\epsilon] = 0$ and $Var(\epsilon) = \sigma^2$, and that we create a model $\hat{t}_i = y(\mathbf{x}_i)$ learned from $\mathcal{D} = \{\mathbf{x}_i, t_i\}$. Then, the expected square error is:

  $$\mathbb{E}[(t - y(\mathbf{x}))^2] = Var(t) + Var(y(\mathbf{x})) + \mathbb{E}[f(\mathbf{x}) - y(\mathbf{x})]^2$$

  where $Var(t)$ is an irreducible error (variance of the noise), $Var(y(\mathbf{x}))$ is the variance of the model and the last term is the $bias^2$. The variance of the model is how spread the expected error is w.r.t. all the possible dataset that we can use to learn the model (if we learn the model on $N$ different datasets, how different will they be? low variance → similar models). The bias of a model is the difference between the truth ($f$) and what we expect to learn ($\mathbb{E}[y(\mathbf{x})]$). Ideally we would like to have low bias and low variance, in practice it's a trade-off.

- **Dealing with Bias-Variance trade-off**
  Regularization is a way to deal with the bias-variance trade-off: by adding the $\lambda$ term and by increasing the value of $\lambda$ we force the model to remain smooth (low variance). Other techniques to lower the variance are:

  → *Feature selection*: a simple idea is to perform an exhaustive search (comparison in performances of all the possible combinations of the features), however it may be computationally infeasible. Other approaches are:

  - *filter*: features are ranked independetly based on how much information they can bring w.r.t. the target
  - *embedded*: feature selection is performed as a step of the machine learning approach used (*e.g.* lasso)
  - *wrapper*: still an attempt to do the best subset selection among the possibles, but a greedy search algorithm is applied to reduce the combinations that are analyzed to avoid the exhaustive search (two examples of this approach are backward and forward stepwise selection)

  → *Dimensionality reduction*: the aim is to reduce the dimension of the input space but, differently from feature selection, all the features are used since they're mapped into a lower-dim space. Moreover, dimensionality reduction methods are unsupervised. A dimensionality reduction technique is PCA:

  1. center the points and (if needed) apply normalization
  2. compute the covariance matrix, its eigenvectors and eigenvalues
  3. pick $k < d$ eigenvectors with highest eigenvalues
  4. project the data on the selected eigenvectors

  PCA is efficient, however it may fail if there are features that explain a little amount of variance but are very important for the explaination of the target. It may also fail if data are distributed in multiple clusters.

- **Model Ensambles**
  We can reduce bias/variance using ensamble methods (learn several models and combine them), more precisely bagging for variance reduction and boosting for bias reduction:

  → *Bagging* (bootstrap aggregation) (↓ variance, ✗ stable learners)

  1. Consider a dataset of $\mathcal{D}$ touples
  2. At iteration $i$ sample with raplcement from $\mathcal{D}$ a training set $\mathcal{D}_i$ of $d$ touples (bootstrap)
  3. Learn a model $\mathcal{M}_i$ for each training set $\mathcal{D}_i$
  4. Return the target prediction for each model $\mathcal{M}_i$
  5. As final result return the majority voting in case of classification or the average in case of regression

  → *Boosting* (focus on misclassified points) – *Adaboost* (classification with two classes $(-1/+1)$) (↓ bias, ✓ stable learners)

  1. Assign uniform weights to each training sample
  2. At iteration $i$ learn a weak classifier $h_i$
  3. Compute the error $\epsilon_i = \sum_j w_j \mathbb{I}_{\{h_i(x_j) \neq y_j\}}$ and compute $\alpha_i = \frac{1}{2} \ln(\frac{1-\epsilon_i}{\epsilon_i})$
  4. Update (and then normalize) the weights: $w_{i+1} = w_i e^{-\alpha_i}$ correctly classified
     $= w_i e^{\alpha_i}$ incorrectly classified
  5. The final model is: $f(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x)) \in \{-1, +1\}$

- **Model Evaluation**
  To properly evaluate the performance of a model we typically use $k$-fold cross validation: randomly split the training data $\mathcal{D}$ into $k$ folds, for each fold $\mathcal{D}_i$ train the model on $\mathcal{D}\backslash\mathcal{D}_i$ and compute the error on $\mathcal{D}_i$.

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

  Finally, estimate the prediction error as the average error computed:

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

  After the whole procedure, we end up with $k$ different models, what do we do? We can choose one of them or combine them together to obtain an ensamble. However, a good and common approach is to use this procedure only to fix the hyperparameters, then we can use the "train + validation" approach to train the decided model.

  Otherwise, to compare models we can use some complexity-adjusted model evaluation indeces (Mallow's $C_P$, Akaike Information Criteria, Bayesian Information Criteria, ~~Bayesian Information Criteria,~~ adjusted $R^2$). The indeces adjust the training error s.t. it takes into account also the complexity of the models.

# Kernel Methods

- **Kernel functions**
  Sometimes there are non-linear patterns in the data and linear models are not rich enough. Kernel methods allow to make linear models work in non-linear settings by mapping data to higher dimensions, where patterns become linear. Kernel methods don't require to explicitly compute the feature mapping.

  A kernel function is defined as the scalar product between the feature vectors of two data samples:

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

  We can interpret it as similarity measure between the two samples. It is possible to rework the representation of linear models to replace all the terms that involve $\phi(\mathbf{x})$ with other terms that involve only $k(\mathbf{x},\cdot)$. Thus, the output of a linear model can be computed only on the basis of the similarities between samples (computed with the kernel function). This approach is called kernel trick.

- **Kernel Ridge Regression**
  Let's consider the ridge regression loss function:

  $L(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(\mathbf{w}^T\phi(\mathbf{x}_n) - t_n)^2 + \frac{\lambda}{2}||\mathbf{w}||_2^2 = \frac{1}{2}(\mathbf{t}-\mathbf{\Phi w})^T(\mathbf{t}-\mathbf{\Phi w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$

  We set to zero the gradient of $L$ w.r.t. $\mathbf{w}$:

  $\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \lambda\mathbf{w} - \mathbf{\Phi}^T(\mathbf{t}-\mathbf{\Phi w}) = 0$

  and, instead of solving for $\mathbf{w}$, we do a change of variable:

  $\mathbf{w} = \mathbf{\Phi}^T\lambda^{-1}(\mathbf{t}-\mathbf{\Phi w}) := \mathbf{\Phi}^T\mathbf{a}$ where: $\mathbf{a} = \lambda^{-1}(\mathbf{t}-\mathbf{\Phi w})$

  Substituting every $\mathbf{w}$ in the expression of the gradient of $L$ equal to zero we obtian:

  $\mathbf{a} = (\mathbf{K}+\lambda\mathbf{I})^{-1}\mathbf{t}$ where: $\mathbf{K} = \mathbf{\Phi\Phi}^T$

  The new matrix, $\mathbf{K}$, is called Gram matrix and it represents the similarities between each pair of samples in the training data. Its structure is:

  $\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1,\mathbf{x}_1) & .. & k(\mathbf{x}_1,\mathbf{x}_N) \\ .. & .. & .. \\ k(\mathbf{x}_N,\mathbf{x}_1) & .. & k(\mathbf{x}_N,\mathbf{x}_N) \end{bmatrix}$ where: $k(\mathbf{x}_i,\mathbf{x}_j) = \phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j)$

  Due to this reprsentation we can formulate the dual representation for the regression model:

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

  where $\mathbf{k}(\mathbf{x})^T = (k(\mathbf{x}_1,\mathbf{x}),..,k(\mathbf{x}_N,\mathbf{x}))$, which means that the prediction is computed as the linear combination of the target values of the samples in the training set.

  The dual representation requires to compute the inverse of $(\mathbf{K}+\lambda\mathbf{I}_N)$, which is convenient when $M$ (number of $\phi_j$) is very large. All the process doesn't require to explicitly compute $\phi$'s and this is good: the similarity between data samples is generally both less expensive to compute and easier to design than computing $\phi$'s.

- **Design of Kernels**
  To avoid to explicitly compute the feature vectors, we can either design a kernel function from scratch or design from existing kernel functions by applying a set of rules. In any case, we must be sure that the designed kernel function is valid (which means that it corresponds to a scalar product into a feature space). The *Mercer theorem* states that any continuous, symmetric, positive semi-definite kernel function can be expressed as a dot product in a high-dim space. A necessary and sufficient condition for a kernel to be valid is that its Gram matrix $\mathbf{K}$ has to be positive semi-definite for all possible choices of $\mathcal{D}$, i.e. $\mathbf{x}^T\mathbf{K}\mathbf{x} > 0$ for any non zero real vector $\mathbf{x}$.

  Some commonly used kernels are:

  - *gaussian kernel*: $k(\mathbf{x},\mathbf{x}') = exp(-||\mathbf{x}-\mathbf{x}'||^2/2\sigma^2)$, corresponding to the dot product of two $\infty$-dim $\phi$'s
  - *symbolic data*: $k(A_1,A_2) = 2^{|A_1\cap A_2|}$ where $A_1$ and $A_2$ are sets
  - *kernel based on probability distributions*: $k(\mathbf{x},\mathbf{x}') = p(\mathbf{x})\,p(\mathbf{x}')$

- **Kernel $k$-nn Regression**

  $k$-nn can be applied to solve a regression problem by averaging the $k$ nearest samples in the training data:

  $$\hat{f}(\mathbf{x}) = \frac{1}{k} \sum_{\mathbf{x}_i \in N_k(\mathbf{x})} t_i$$

  This procedure is very sensible to noise (moving a little bit makes the function change a lot). To mitigate the noise effect we can introduce a kernel function. The *Nadaraya-Watson model*, or kernel regression, deal with this issue by using a kernel function to compute a weighted average of samples:

  $$\hat{f}(\mathbf{x}) = \frac{\sum_{n=1}^{N} k(\mathbf{x}, \mathbf{x}_n)\, t_n}{\sum_{n=1}^{N} k(\mathbf{x}, \mathbf{x}_n)}$$

- **Gaussian Process**

  We can apply kernel methods to bayesian regression, obtaining a gaussian process. We start from the same assumptions used in bayesian linear regression: linear model with a prior distribution over the weights

  $$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \mathbf{x}$$
  $$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|0, \tau^2 \mathbf{I})$$

  Instead of modeling the distribution of the weights we try to model the outputs of the regression function ($y$'s):

  $$\mathbf{y} = \mathbf{\Phi w} \quad \Rightarrow \quad p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|\mu, \mathbf{S})$$

  where:

  $$\mu = \mathbb{E}[\mathbf{y}] = \mathbf{\Phi}\mathbb{E}[\mathbf{w}] = \mathbf{0}$$
  $$\mathbf{S} = Cov(\mathbf{y}) = \mathbb{E}[\mathbf{y}\mathbf{y}^T] = \mathbf{\Phi}\mathbb{E}[\mathbf{w}\mathbf{w}^T]\mathbf{\Phi}^T = \tau^2 \mathbf{\Phi}\mathbf{\Phi}^T = \mathbf{K} \quad \text{where:} \quad \mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) = \tau^2 \phi(\mathbf{x}_i, \mathbf{x}_j)$$

  This gives a probabilistic interpretation of the kernel function as:

  $$k(\mathbf{x}_i, \mathbf{x}_j) = \mathbb{E}[y(\mathbf{x}_i) y(\mathbf{x}_j)]$$

  the kernel function computed between two samples represents how much the outputs of the model are correlated in the two points. The kernel shapes how correlated are the outputs of the model for each pair of points.

  Some commonly used kernels are:

  - *gaussian kernel*: $k(\mathbf{x}, \mathbf{x}') = exp(-||\mathbf{x} - \mathbf{x}'||^2 / 2\sigma^2)$
  - *exponential kernel*: $k(\mathbf{x}, \mathbf{x}') = exp(-\theta|\mathbf{x} - \mathbf{x}'|)$

  In order to apply Gaussian process models to the problem of regression, we need to take account of the noise on the observed data:

  $$t_n = y(\mathbf{x}_n) + \epsilon_n$$

  Assuming the noise gaussian and independent on each point, the joint distribution of the targets will be:

  $$p(\mathbf{t}|\mathbf{y}) = \mathcal{N}(\mathbf{t}|\mathbf{y}, \sigma^2 \mathbf{I})$$

  Thus, we can compute the marginal distribution $p(\mathbf{t})$ (given $p(\mathbf{t}|\mathbf{y}) = \mathcal{N}(\mathbf{t}|\mathbf{y}, \sigma^2\mathbf{I})$ and $p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K})$):

  $$p(\mathbf{t}) = \int p(\mathbf{t}|\mathbf{y}) p(\mathbf{y}) d\mathbf{y} = \mathcal{N}(\mathbf{t}|\mathbf{0}, \mathbf{C}) \quad \text{where:} \quad \mathbf{C} = \mathbf{K} + \sigma^2 \mathbf{I} \quad \text{i.e.} \quad C(\mathbf{x}_i, \mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j) + \sigma^2 \delta_{ij}$$

  To make prediction, given $D = \{(\mathbf{x}_1, t_1), .., (\mathbf{x}_N, t_N)\}$ and $\mathbf{x}_{N+1}$, we frist model the joint distribution $p(\mathbf{t}_{N+1})$ and then the conditional:

  $$p(t_{N+1}|\mathbf{t}_N) = \mathcal{N}(m, \sigma^2)$$

  where:

  $$m(\mathbf{x}_{N+1}) = \mathbf{k}^T \mathbf{C}_N^{-1} \mathbf{t}_N \quad \text{with:} \quad [\mathbf{k}]_j = k(\mathbf{x}_j, \mathbf{x}_{N+1})$$
  $$\sigma^2(\mathbf{x}_{N+1}) = c - \mathbf{k}^t \mathbf{C}_N^{-1} \mathbf{k} \quad \text{with:} \quad c = k(\mathbf{x}_{N+1}, \mathbf{x}_{N+1}) + \sigma^2$$

# Support Vector Machines

- **Separable problems**

In the linear classification settings we characterized a separating hyperplane with the equation $\mathbf{w}^T\phi(\mathbf{x})+b=0$. The distance of a point $\mathbf{x}_n$ from the separating boundary is given by: $y(\mathbf{x}_n)/||\mathbf{w}||_2 = (\mathbf{w}^T\phi(\mathbf{x}_n)+b)/||\mathbf{w}||_2$, however, since it's not a proper distance (it can be $<0$), we multiply it by the target $t_n$ (if the distance is $<0$ then the corresponding target is $-1$). In this way we can define the margin as:

$$margin = \min_n \frac{t_n\,(\mathbf{w}^T\phi(\mathbf{x}_n)+b)}{||\mathbf{w}||_2}$$

The optimal separating hyperplane is the one that has the maximum margin, so the one corresponding to :

$$\arg\max_{\mathbf{w},b}\{\min_n \frac{t_n\,(\mathbf{w}^T\phi(\mathbf{x}_n)+b)}{||\mathbf{w}||_2}\}$$

This optimization problem results difficult to be solved, so we formulate an equivalent constrained optimization problem. Among all the possible separating hyperplanes, we consider only those which we can express as:

$$t_n\,(\mathbf{w}^T\phi(\mathbf{x}_n)+b)=1 \qquad \forall\,\mathbf{x}_n\in\mathcal{S}=\{\text{support vectors}\}$$

where the support vectors are all the points that lay on the margin. In this way the margin is equivalent to $1/||\mathbf{w}||_2$ and so, to maximalize the margin we have to minimize $||\mathbf{w}||_2$. The overall constrained optimization problem can be written as:

$$\begin{aligned}&\text{Minimize:} && \tfrac{1}{2}||\mathbf{w}||_2^2\\ &\text{Subject to:} && t_n\,(\mathbf{w}^T\phi(\mathbf{x}_n)+b)\geq 1 \qquad \forall n\end{aligned}$$

This formulation contains $\phi(\cdot)$, which we would like to avoid. We can introduce a kernel function and we can derive the dual problem, where we no longer have $\phi(\cdot)$, only $k(\cdot,\cdot)$:

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

The resulting discriminant function is:

$$y(\mathbf{x})=\sum_{n=1}^N \alpha_n t_n\, k(\mathbf{x},\mathbf{x}_n)+b \qquad \text{where:} \qquad b=\frac{1}{|\mathcal{S}|}\sum_{\mathbf{x}_n\in\mathcal{S}}\Big(t_n-\sum_{\mathbf{x}_m\in\mathcal{S}}\alpha_m t_m\, k(\mathbf{x}_n,\mathbf{x}_m)\Big)$$

where $a_n\neq 0$ only for the $\mathbf{x}_n\in\mathcal{S}$ (only the support vectors contribute to the discriminant function).

- **Non-separable problems**

If samples are not linearly separable in the feature space we may allow for "errors" ($\xi$) in classification. Soft-margin optimization problem:

$$\begin{aligned}&\text{Minimize:} && \tfrac{1}{2}||\mathbf{w}||_2^2 + C\textstyle\sum_{n=1}^N \xi_n\\ &\text{Subject to:} && t_n\,(\mathbf{w}^T\phi(\mathbf{x}_n)+b)\geq 1-\xi_n \qquad \forall n\\ &&& \xi_n\geq 0 \qquad \forall n\end{aligned}$$

where $\xi_n$ are called slack variables and represents penalties to margin violation and $C$ is a trade-off parameter between error and margin (the bigger $C$ and the less the violation will be allowed).

The dual formulation (no feature space direct involvement):

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

In this case the supoprt vectors are all the points either on the margin or inside the margin. For support vectors we have $\alpha_n>0$, moreover if $\alpha_n<C$ then $\xi_n=0$ and the sample is on the margin, otherwise, if $\alpha_n=C$, the point is inside the margin and it's either correctly or wrongly classified.

An alternative formulation for the dual problem that allows to get rid of $C$ (since $C$ is not related to anything in the problem and to tune it we should make a guess):

$$\begin{aligned}&\text{Maximize:} && -\frac{1}{2}\textstyle\sum_{n=1}^N\sum_{m=1}^N \alpha_n\alpha_m t_n t_m\, k(\mathbf{x}_n,\mathbf{x}_m)\\ &\text{Subject to:} && 0\leq\alpha_n\leq\frac{1}{N} \qquad \text{for } n=1,..,N\\ &&& \textstyle\sum_{n=1}^N \alpha_n t_n = 0 \qquad \text{for } n=1,..,N\\ &&& \textstyle\sum_{n=1}^N \alpha_n \geq \nu \qquad \text{for } n=1,..,N\end{aligned}$$

where $0\leq\nu<1$ is a parameter that controls both the margin errors and the number of support vectors:

fraction of margin errors $\leq\nu\leq$ fraction of SVs

For example: suppose $N=1000$ and $\nu=0.1\rightarrow$ the number of support vectors that we'll find among the 1000 points will be $\geq N\nu=100$. Moreover, $N\nu$ is also the upper bound of the misclassified points.

# Computational Learning Theory

- **No Free Lunch Theorem**
  We would like to have something that allows us to bound the test error based on the training error (under some assumptions). A learner $L$ wants to learn a concept $c \in \mathcal{H}$ that maps the data in the input space $X$ to a target $t$. Let assume $L$ found an hypothesis $h^*$ with no errors on the training data: how many training samples of $X$ are necessary to be sure that $L$ actually learned the true concept (i.e. $h^* = c$)? With no assumption, unless we observe all the points, we cannot tell if we're learning the true concept or no. This is the basis of the *No Free Lunch theorem*, which states that in absence of any assumption about the data, there is no reason to prefer one model over another, there is no apriori system for determining the best model for a specific dataset.
  If we define with $ACC_G(L)$ the generalization accuracy of $L$ (accuracy on the samples $\notin$ training set) and with $\mathcal{F}$ the set of all the possible concepts $y = f(\mathbf{x})$ then, for any learner $L$ and any possible training set we have:

  $$\frac{1}{|\mathcal{F}|}\sum_{\mathcal{F}} ACC_G(L) = \frac{1}{2}$$

  i.e. on average w.r.t. all the possible samples, the generalized accuracy will be $\frac{1}{2}$.

- **Probably Learning an Approximately Correct Hypothesis**
  Consider a binary classification setting. Let $X$ be the input space, $H$ the space of all the possible learnable functions that map a point $\in X$ to a target and $C$ the space of all the functions that map a point $\in X$ to a target that we may want to learn ($C \neq H$ because $H$ are the learnable functions, $C$ are the functions that we want to learn, there is no guarantee that there is a way to learn them). A learner $L$ outputs a hypothersis $h^* \in \mathcal{H}$ such that:

  $$h^* = \arg\min_h error_{train}(h)$$

  We compute the error of an hypothesis as the probability of misclassifying a sample:

  $$error_{\mathcal{D}}(h) = \mathbb{P}_{\mathbf{x}\in\mathcal{D}}(h(\mathbf{x}) \neq c(\mathbf{x})) = \frac{1}{|\mathcal{D}|}\sum_{\mathbf{x}\in\mathcal{D}} \mathbb{I}_{h(\mathbf{x})\neq c(\mathbf{x})}$$

  where $\mathcal{D}$ is the training data. This is the training error, instead, we're interested in the true error of $h$:

  $$error_{true}(h) = \mathbb{P}_{\mathbf{x}\sim P(X)}(h(\mathbf{x}) \neq c(\mathbf{x}))$$

  where $P(X)$ is the input space distribution.

  We say that $h$ overfits the training data if $error_{true} > error_{\mathcal{D}}$. Can we boudn $error_{true}$ using $error_{\mathcal{D}}$? Since $error_{true}$ is the probability of making a mistake on a sample and, since $error_{\mathcal{D}}$ is the average error probability on $\mathcal{D}$, we may assume a Bernoulli distribution for the error and obtain a 95% CI:

  $$error_{true}(h) = error_{\mathcal{D}}(h) \pm 1.96\sqrt{\frac{error_{\mathcal{D}}(h)(1-error_{\mathcal{D}}(h))}{n}}$$

  However, this is **not** correct because we're introducing a bias: the probability of making an error on a point of $\mathcal{D}$ is $\neq$ from the probability of making an error on a point $\notin \mathcal{D}$.

  **Bound for consistent learners**
  We introduce the concept of consistency: a hypothesis $h$ is consistent with a training dataset $\mathcal{D}$ of the concept $c$ if and only if $h(\mathbf{x}) = c(\mathbf{x})$ for each training sample in $\mathcal{D}$ (we write: $Consistent(h, \mathcal{D})$). We define the version space $VS_{H,\mathcal{D}}$ w.r.t. the hypothesis space $H$ and the dataset $\mathcal{D}$ as the subset of hypothesis of $H$ consistent with $\mathcal{D}$ (the version space is the set of all the hypothesis with zero training errors). Can we (properly) bound $error_{true}$ if we consider only consistent learners? (learners that outputs consistent hypothesis, always $error_{\mathcal{D}} = 0$)
  If $H$ is finite and $\mathcal{D}$ is a sequence of $N \geq 1$ iid random samples 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 than $\epsilon$ is $\leq |H|\, e^{-\epsilon N}$:

  $$\mathbb{P}(\exists h \in H : error_{\mathcal{D}}(h) = 0 \wedge error_{true}(h) \geq \epsilon) \leq |H|\, e^{-\epsilon N}$$

  We're bounding the probability that the version space contains an hypothesis with a true error $\geq \epsilon$.
  Let's call $\delta$ the probability to have $error_{true} \geq \epsilon$ for a consistent hypothesis: $|H|\, e^{-\epsilon N} \leq \delta$. We can bound $N$ after setting $\epsilon$ and $\delta$ and we can bound $\epsilon$ after setting $N$ and $\delta$:

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

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

  **Bound for general learners**
  In general (agnostic) learners will output hypothesis $h$ such that $error_{\mathcal{D}} > 0$. If $H$ is finite and $\mathcal{D}$ is a sequence of $N \geq 1$ iid samples of some target concept $c$, then for any $0 \leq \epsilon \leq 1$ and for any learned hypothesis $h$, the probability that $error_{true}(h) - error_{\mathcal{D}}(h) > \epsilon$ is $\leq |H|\, e^{-2N\epsilon^2}$:

  $$\mathbb{P}(\exists h \in H : error_{true}(h) > error_{\mathcal{D}}(h) + \epsilon) \leq |H|\, e^{-2N\epsilon^2}$$

  Again, from this formulation we can obtain some bounds:

  $$N \geq \frac{1}{2\epsilon^2}(\ln|H| + \ln(\frac{1}{\delta}))$$

  $$error_{true}(h) \leq error_{\mathcal{D}}(h) + \sqrt{\frac{\ln|H| + \ln(\frac{1}{\delta})}{2N}}$$

  All these results are based on the fact that $|H|$ is finite. What if it is not?

**VC Dimension**

We define a dichotomy of a set $S$ of instances as a partition of $S$ into two disjoint subsets (negative and positive). We say that a set of instances $S$ is shattered by the hypothesis space $H$ if and only if $\forall$ dichotomy of $S$ $\exists h \in H$ consistent with this dichotomy ($\forall$ labelling can be learned in $H$). We define the VC($H$) dimension of the hypothesis space $H$ over the instance space $X$ as the largest finite subset of $X$ shattered by $H$.
For example: a linear classifier in $M$-dim space has VC($H$) $= M + 1$.

Given this definition, we can rewrite the previous bounds:
→ how many randomly drawn examples suffice to guarantee that any hypothesis that perfectly fits the training data (consistent) is probabily $(1-\delta)$ approximately $(\epsilon)$ correct?

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

→ with probability $\geq (1-\delta)$ every $h \in H$ satisfies the following:

$$error_{true}(h) \leq error_{\mathcal{D}}(h) + \sqrt{\frac{VC(H)\,(\ln\frac{2N}{VC(H)}+1)+\ln\frac{4}{\delta}}{N}}$$

**PAC-learnability**

Considering a class $C$ of possible target concepts, an input space $X$ with $dim(\mathbf{x} \in X) = M$, a hypothesis space $H$ and a learner $L$ we define: $C$ is PAC-learnable by $L$ using $H$ if $\forall c \in C$, $\forall$ distributions $P(X)$, $\epsilon \in (0, \frac{1}{2})$ and $\delta \in (0, \frac{1}{2})$, the learner $L$ will, with probability $\geq (1-\delta)$, output a hypothesis $h \in H$ such that $error_{true} \leq \epsilon$ in time that is polynomial in $1/\epsilon$, $1/\delta$, $M$, and $size(c)$. A sufficient condition to prove PAC-learnability is proving that $L$ requires only a polynomial number of training samples and that processing per sample is polynomial.

# Markov Decision Processes

- **Agent-Enviroment Interface**

  In sequential decision making an agent takes a sequence of decisions/actions to reach the goal. The optimal actions are context-dependent and we generally do not have examples of correct actions. What we have instead is a reward guidance that tells if the agent is doing good or not. Thus, the agent-enviroment interface is:

  - the agent observe the state $S_t$ (current situation of the enviroment) $\in \mathcal{S}$
  - the agent takes and action $A_t \in \mathcal{A}(\mathcal{S}_t)$ which changes the enviroment and brings it to new state $S_{t+1}$
  - because of the enviroment-change, the agent gets a reward $R_{t+1} \in \mathcal{R}$ (the reward accounts only for the very last action that the agent took, it's not a feedback on how good the action is w.r.t. the ultimate goal)

  where $\mathcal{S}$ is the state space (space of all the possible configuration of the enviroment), $\mathcal{A}(\mathcal{S}_t)$ is the action space (space of all the possible actions that the agent can take as a function of the current state), $\mathcal{R}$ is the reward space (space of all the posible rewards that an action can generate).

- **Markov Decision Process (MDP)**

  We introduce the Markov property: future state $s'$ and reward $r$ only depends on current state $s$ and action $a$. Hence, we obtain a Markov Decision Process (MDP) for which we can describe the one-step dynamic as:

  $$p(s', r|s, a)$$

  When the Markov property holds and the state and action sets are finite, the problem is a finite MDP. To completely characterize a finite MDP we need to define: state and action sets and one step dynamics

  $$p(s', r|s, a) = \mathbb{P}(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$$

  We can also derive the next state distribution based on the action that the agent takes and its current state:

  $$p(s'|s, a) = \mathbb{P}(S_{t+1} = s'|S_t = s, A_t = a) = \sum_{r\in\mathcal{R}} p(s', r|s, a)$$

  and the expected reward for taking the action $a$ in the state $s$:

  $$r(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r|s, a)$$

  The agent should not choose actions on the basis of immediate rewards: long-term consequences are more imoprtant. We need to take into account the sequence of future rewards. We define the return $G_t$ as a function of the sequence of future rewards (usually the return is simply the sum of future rewards):

  $$G_t = f(R_{t+1} + R_{t+2} + R_{t+3} + ..)$$

  To succeed, the agent has to maximize the expected return $\mathbb{E}[G_t]$. There can be two situations:

  - *Episodic tasks*: the agent-enviroment interaction naturally breaks into chunks called episodes.
    $$\mathbb{E}[G_t] = \mathbb{E}[R_{t+1} + R_{t+2} + .. + R_T] \qquad \text{where } T \text{ is the terminal state}$$
  - *Continuing tasks*: the agent-enviroment interaction goes on continually and there are no terminal states. The total reward is a sum over an infinite sequence and might not be finite. To solve this issue we can discount the future rewards by a factor $\gamma \in (0, 1)$:
    $$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + .. + \gamma^{k-1} R_{t+k} + .. < \infty \qquad \Rightarrow \qquad \mathbb{E}[G_t] = \mathbb{E}[\textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}]$$

We can design the terminal states in episodic tasks as absorbing states that always produce zero rewards. Hence, we can now use the same definition of expected reward for episodic and continuing tasks:

$$\mathbb{E}[G_t] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}]$$

where $\gamma = 1$ can be used if an absorbing state is always reached. If $\gamma = 0$ the agent takes care about immediate rewards only and as $\gamma \to 1$ the agent takes into account more and more the future rewards.

The reward hypothesis is that through the reward we can formalize an objective for the agent such that it's possible for the agent to learn the optimal strategy to reach the goal.

- **MDP: Policy**

A policy ($\pi(\cdot)$), at any given point in time, decides which action the agent selects. A policy fully defines the behavior of an agent. Policies can be deterministic ($\pi(s) = a$) or stochastic ($\pi(a|s)$): a deterministic policy is a function that maps each state in the state space into one and only one action, a stochastic policy maps each state to a probability distribution over the actions (s.t. $\sum_a \pi(a|s) = 1$). Policies can be Markovian, if the distribution of the actions depends only on the current state, or non Markovian, if the distribution of the actions depends on the current state and on the past. Policies can also be stationary or non-stationary.

- **MDP: Value Functions**

We need a way to evaluate how good a policy is. Given a policy $\pi$ we can compute the state-value function as:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]$$

which represents the expected return from a given state $s$, following the policy $\pi$.

We can also compute the action-value function as:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a]$$

which represents the expected return from a given state $s$, when a given action $a$ is selected and then policy $\pi$ is followed. In this way we make explicit the contribution of an action. Why is this interesting? Because we can try to change the policy to improve it. In doing so, $Q_\pi(s,a)$ is convenient to use because it tells how would be the return if instead of doing the action of the policy we take another action ($a$).

**Bellman Expectation Equations**

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

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1})|S_t = s] \\ &= \sum_{a\in\mathcal{A}} \pi(a|s)\ [r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a) V_\pi(s')] \end{aligned}$$

The action-value function can be similarly decomposed:

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1})|S_t = s, A_t = a] \\ &= r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a) V_\pi(s') \\ &= r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a) \sum_{a\in\mathcal{A}} \pi(a'|s') Q_\pi(s',a') \end{aligned}$$

In both cases we get a recursive expression.

- **MDP: Optimality**

How can we compare policies through value functions? Value functions are states functions. If two policies, $\pi_1$ and $\pi_2$, are such that $V_{\pi_1}(s) > V_{\pi_2}(s)\ \forall s \in \mathcal{S}$ then we can conclude that $\pi_1$ is better than $\pi_2$. What if, instead, in some states $\pi_1 > \pi_2$ and in others $\pi_1 < \pi_2$?

For any MDP there exists always at least one optimal deterministic policy $\pi^*$ that is better or equal to all the others ($\pi^* \geq \pi\ \forall\pi$). This means that if $\pi_1 > \pi_2$ in a set of states $A$ and $\pi_1 < \pi_2$ in a set of states $B$, we can create a better policy $\pi_3$ by taking $\pi_1$ in $A$ and $\pi_2$ in $B$.

Note: since the state-value function involves $\gamma$, as $\gamma$ changes also the optimality definition changes (if $\pi^*$ is the optimal policy with a particular $\gamma$, it doesn't mean that $\pi^*$ is still the best with another $\gamma$).

The optimal state-value function and action-value function are unique and determined as:

$$\begin{aligned} V^*(s) &= \max_\pi V_\pi(s) && \forall s \in \mathcal{S} \\ Q^*(s,a) &= \max_\pi Q_\pi(s,a) && \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \end{aligned}$$

We can have different optimal policies but they all have the same state-value and action-value functions.

**Bellman Optimality Equations**

Bellman optimality equation for $V^*$: (this time we have an unknown part: $\pi^*(a|s)$)

$$\begin{aligned} V^*(s) &= \sum_{a\in\mathcal{A}} \pi^*(a|s)\ [r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a) V^*(s')] \\ &= \max_a [r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a) V^*(s')] \end{aligned}$$

Bellman optimality equation for $Q^*$:

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

Both are recursive and both don't depend on unknown variables.

From $V^*$ and $Q^*$ we can easily compute the optimal policy $\pi^*$:

$$\pi^*(s) = \arg\max_a [\, r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*(s') \,] = \arg\max_a Q^*(s,a)$$

- **MPD: Conclusions**
  We have defined optimal value functions and optimal policies, however in practice optimal policies can be generated only with extreme computational cost. This is the reason why we introduce *dynamic programming* (policy iteration, value iteration) and *reinforcement learning algorithms* (Q-learning, SARSA) which are both iterative solutions that approximate what we're looking for.

## Dynamic Programming

- **Why Dynamic Programming?**
  To solve an MDP we need to find the optimal policy. Because of the size of the state space we cannot procede using linear systems solver (for Bellman equations). Dynamic Programming (DP) is a method that allows to solve a complex problem by breaking it down into simpler sub-problems in a recursive manner (we start from a policy and we iteratively improve it till we converge to the optimal one). To work on a policy we first need to evaluate it and then to improve it.

- **Policy Evaluation**
  DP solves the evaluation task through an iterative application of the Bellman equation:

  $$V_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \; [\, r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s') \,]$$

  At each iteration $k$ the value function $V_k$ is updated for all the states $s \in \mathcal{S}$.

  - Given the policy $\pi$, to be evaluated
  - Repeat until convergence :
    1. $\Delta \leftarrow 0$
    2. Loop for every $s \in \mathcal{S}$:
       2.1 $v \leftarrow V(s)$
       2.2 $V(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} p(s', r|s, a)[r + \gamma V(s')]$
       2.3 $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

  This algorithm, differently from the first equation, uses the in-place policy iteration.

- **Policy Improvement**
  To improve the policy we can act greedy w.r.t. a non optimal value function:

  $$\pi'(s) = \arg\max_a [\, r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s') \,] = \arg\max_a Q_\pi(s,a) \qquad \forall s \in \mathcal{S}$$

  The policy improvement theorem says that for any pair of deterministic policies $\pi'$ and $\pi$ such that $Q_\pi(s, \pi'(s)) \geq Q_\pi(s, \pi(s)) \, \forall s \in \mathcal{S}$ we have that $\pi' \geq \pi$. If $\exists \, s \in \mathcal{S}$ such that $Q_\pi(s, \pi'(s)) > Q_\pi(s, \pi(s))$ then $\pi' > \pi$. Because of this, we have that the above iterative procedure leads to $\pi' = \pi$ only if $\pi$ is already the optimal policy.

- **Policy Iteration**
  The overall process goes under the name of policy iteration. We start from a random policy $\pi_0$, we evaluate it and we get its value function $V_{\pi_0}$. Then we apply policy improvement and we get a new policy, $\pi_1$. For the policy improvement theorem we know that $\pi_1 \geq \pi_0$. We do the evaluation of $\pi_1$, we get $V_{\pi_1}$, we perform the improvement and we go on repeating until we reach the optimal policy.

- **Generalized Policy Iteration (GPI) - Value Iteration**
  Policy iteration alternates complete policy evaluation and improvement up to the convergence. It can also be possible to find the optimal policy interleaving partial evaluation and improvement steps. Value iteration is one of the most popular GPI methods. In particular, it exploits the improvement ($\pi'$) directly in the evaluation:

  Improvement: $\pi'(s) = \arg\max_a [\, r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s') \,] \qquad \forall s \in \mathcal{S}$

  Evaluation: $V_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi'(a|s) \, [\, r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s') \,] \qquad \forall s \in \mathcal{S}$

  Combining them, we only need to iterate the update of the value function using the Bellman optimality equation:

  $$V_{k+1}(s) \leftarrow \max_a [\, r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s') \,] \qquad \forall s \in \mathcal{S}$$

  This does not try to compute explicitly the policy, only the value function. At the very end we derive the optimal policy from the optimal value function.

# Monte Carlo Methods

- **Why sample-based methods?**
  With dynamic programming we are able to find the optimal value function and the corresponding optimal policy, however we generally do not know the problem dynamics (expected reward for each action, the probability of the next states, etc.). We need to learn all from data. What is data in this context? A direct interaction with the enviroment, i.e. experience: sequence of actions, states and rewards that we observe by a direct interaction with the enviroment. Monte Carlo methods relies only on the experience (data) to learn value functions and policy. Monte Carlo works on complete sample returns, which leads to the fact that Monte Carlo is defined only for episodic tasks. There are two approaches: *model-free* (no model necessary and still reaches optimality), *simulated* (based on simulations, not a full model).

- **Policy Evaluation**
  The goal is to learn $V_\pi(s)$ given some number of episodes under $\pi$ which contains $s$. The idea is to average the returns observed after visiting $s$:

  $$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \approx average[G_t|S_t = s]$$

  What if we encounter the same state multiple times in an episoe? There are two approaches:

  - every-visit MC: average returns for every time $s$ is visited in an episode
  - first-visit MC: average returns only for the first time $s$ is visited in an episode

  Both converges asymptotically.

- **Policy Iteration**
  Following the dynamic programming approach, once we evaluated a policy we should proceed with the improvement. However, we don't have the knowledge of $r(s,a)$ or $p(s'|s,a)$, hence we cannot apply:

  $$\pi'(s) = \arg\max_a [r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s')]$$

  That's the reason why we rely on another improvement formulation:

  $$\pi'(s) = \arg\max_a Q_\pi(s,a)$$

  Instead of doing the evaluation of the state-value function $V_\pi$ (useless for the improvement), we do the evaluation of the action-value function $Q_\pi$. We average the returns starting from a state $s$ and action $a$, then following $\pi$:

  $$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \approx average[G_t|S_t = s, A_t = a]$$

  This time the settings are different: the convergence is guaranteed only if every state-action pair is visited. In the evaluation of $V_\pi$ we only needed episodes of experiences following the policy $\pi$. This means that, if we have a state $s_1$ for which there are two possible actions $A_1$ and $A_2$ and a policy $\pi$ says to choose always $A_1$ in $s_1$, we'll be able to estimate $Q_\pi(s_1, A_1)$ but not $Q_\pi(s_1, A_2)$ (since we'll never going to experience it). This is why we introduce *exploration*: we have to take actions that are not necessarily suggested by $\pi$.

  A method for exploration is *exploring starts*: follow a policy $\pi$ to generate the data, but at least in the first step move completely randomly (start from a random state and choose a random action). If we do it consistently we obtain the minimum amount of exploration that allows to observe all the possible state-action pairs.

- **Policy Iteration with $\epsilon$-soft**
  Exploring starts is a good idea but it is not always possible: we might not have a full control on the enviroment in order to start from a random state making a random action. But, there is still the need to keep exploring during the learning process. This leads to a key problem: the *exploration – exploitation dilemma*. We need to explore to learn new things, avoid repeating and we need to expose to different actions. However, exploring may involve some costs, so we may choose to focus on what is already good and minimize the new actions.

  We intorduce the $\epsilon$-greedy exploration. Instead of searching the optimal deterministic policy we search the optimal $\epsilon$-soft policy, i.e. a policy that selects each action with a probability $\geq \epsilon/|\mathcal{A}|$, in particular:

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

  The idea is to focus on the (current) optimal action but leaving a small chance also to the other actions.
  Any $\epsilon$-greedy policy $\pi'$ with respect to $Q_\pi$ is an improvement over any $\epsilon$-soft policy $\pi$ (if we update the policy by making it greedy w.r.t. the current action-value function, we obtain a new value function that is better or equal to the previous policy's). The algorithm will converge to the optimal $\epsilon$-soft policy possible. Is it okay to learn a non-deterministic policy? Yes, eventually we can $\epsilon \to 0$.

- **Off-policy Learning**
  So far we assumed that the policy that we are following is the policy that we are learning. Can we follow a policy different from the one we are learning? Can we learn the action-value function of a policy different from the one that we're following to generate the episodes? Yes, it's called off-policy learning.

In off-policy learning the agent selects an action using a behavior policy $b(a|s)$. The data is used to learn the value functions of a different target policy $\pi(a|s)$. With this approach, we can only learn a target policy that is covered by the behavior policy: we cannot choose an action with probability $> 0$ if we never experienced taking that action in that state:

$$\pi(a|s) > 0 \qquad where \qquad b(a|s) > 0$$

How is this possible? The *importance sampling* technique allows to estimate the expectation of a distribution different from the one used to drawn samples. Suppose that we want to estimate $\mathbb{E}_p[x]$ but sampling from $p(\cdot)$ is very difficult. If $\exists\, q(\cdot)$ from which sampling is easier, we can do:

$$\mathbb{E}_p[x] = \sum_x x\, p(x) = \sum_x x\, \frac{p(x)}{q(x)} q(x) := \sum_x x\, \tilde{p}(x) q(x) = \mathbb{E}_q[x\, \tilde{p}(x)]$$

A sample-based formulation of the above is:

$$\mathbb{E}_p[x] \approx \frac{1}{N} \sum_{n=1}^{N} x_n \quad \text{with: } x_n \sim p(x) \quad \Rightarrow \quad \mathbb{E}_p[x] \approx \frac{1}{N} \sum_{n=1}^{N} x_n\, \tilde{p}(x_n) \quad \text{with: } x_n \sim q(x)$$

In the case of policy evaluation we have that following a policy $\pi$ we compute the state-value function as:

$$V_\pi(s) \approx average(Returns[0], Returns[1], Returns[2], ..)$$

but when we follow a policy $b$, the value function becomes:

$$V_\pi(s) \approx average(\rho_0\, Returns[0], \rho_1\, Returns[1], \rho_2\, Returns[2], ..)$$

where $\rho_j$ is the probability of performing the trajectory observed in episode $j$ while following policy $\pi$ over the probability of observing the same trajectory by following the policy $b$:

$$\rho_j = \frac{\mathbb{P}(trajectory\ under\ \pi)}{\mathbb{P}(trajectory\ under\ b)} = \prod_{k=j}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \qquad \text{where } T \text{ is the terminal state}$$

For example: if we follow the behavior policy $b$ and we observe:

$$(state,\ action,\ return) : (S_0, A_0, G_0) \to (S_1, A_1, G_1) \to (S_2, A_2, G_2) \to (S_T)$$

To evaluate $V_\pi$ we need to correct the returns with:

$$G_0 \to \rho_0 = \frac{\pi(A_0|S_0)\, \pi(A_1|S_1)\, \pi(A_2|S_2)}{b(A_0|S_0)\, b(A_1|S_1)\, b(A_2|S_2)} \qquad G_1 \to \rho_1 = \frac{\pi(A_1|S_1)\, \pi(A_2|S_2)}{b(A_1|S_1)\, b(A_2|S_2)} \qquad G_2 \to \rho_2 = \frac{\pi(A_2|S_2)}{b(A_2|S_2)}$$

For the ultimate evaluation of $V_\pi$ with this method, we can procede in two ways:

ordinary sampling: $V_\pi(s) \approx \frac{\sum_i \rho_i\, Return_i}{N(s)}$

weighted sampling: $V_\pi(s) \approx \frac{\sum_i \rho_i\, Return_i}{\sum_i \rho_i}$

In *ordinary sampling* we have an unbiased result but higher variance. We can observe a trajectory (using $b$) that is extremely unlikely to achieve with policy $\pi$. Moreover it requires a lot of data for a good approximation. In *weighted sampling* we have a biased result but lower variance. If we initially observe some trajectories that are very unlikely according to the target policy $\pi$, we won't weight them a lot (we weight more the trajectories that are likely to $\pi$).

## Temporal-Difference Learning

- **Why temporal-difference?**
  Dynamic programming was not enough because it required us to know the one step dynamics, which we typically don't know. Monte Carlo can learn from experience but it has some limitations: it can be applied only to episodic tasks (MC needs the whole episode to even start).
- **TD(0)**
  Temporal-difference (TD(0)) combines together Monte Carlo and dynamic programming It combines *sampling* (from MC): we don't need the knowledge of the dynamics, we need experience, and *bootstrapping* (from DP): we exploit the recursive structures of the problem to learn/update the value functions in a state $s$ based on the value functions of the states next to $s$. TD(0) is a model-free and bootrsapping method. We start from:

  $$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

  MC policy evaluation

  To obtain $G_t$ we need the episode to be end. To avoid this, we exploit the recursive structure of the return:

  $$G_t \approx R_{t+1} + \gamma V(S_{t+1}) \qquad \textit{(we should have used } G_{t+1}\textit{, that's why it is an approximation)}$$

  and so, substituing we obtain:

  $$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

  Since $G_t$ disappeared, it possible to learn after each step, so we don't need the episode to end to start learning.

  **Comparison**
  TD can learn before knowing the final outcome, MC must wait until the end of the episode to know the return. TD can learn without the final outcome, it can learn from incomplete sequences, MC can only learn from complete sequences. TD works in continuing enviroments, MC only works for episodic enviroments. MC target is an unbiased estimate of $V_\pi(s)$, TD has a bias. TD target has a lower variance, since MC has multiple sources

of stochasticity $(R_{t+1}, R_{t+2}, ..)$, while TD has only two sources of stochasticity $(R_{t+1}, S_{t+1})$. Overall, MC works well with function approximation and it is not very sensitive to initial values, TD has problems with function approximation and it is more sensitive to initial values.

- **SARSA**
  TD(0) solves the evaluation policy problem, can it also learn a policy? Let's go back to MC policy iteration:

  Evaluation: $Q_\pi(s,a) \approx average[G_t|S_t = s, A_t = a]$

  Improvement: $\pi'(s) = \arg\max_a Q_\pi(s,a)$

  The SARSA algorithm idea is to replace the return with an approximation:

  $G_t \approx R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

  The on-policy control with SARSA is:

  Evaluation: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$

  Improvement: $\epsilon$-greedy policy improvement

- **Q-Learning**
  The Q-learning algorithm modifies the SARSA algoritm by going from an on-policy to a off-policy. SARSA, as policy iteration, is based on Bellman expectation equation. Q-learning instead, as value iteration in DP, is based on Bellman optimality equation:

  $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$

# Multi-Armed Bandit

- **Exploration vs. exploitation**
  Online decision making make us face a fundamental choice: exploration vs. exploitation. If we choose only one of the two options we are never going to reach an optimal solution: if we always explore then we never set on a single choice (we always try something new), while if we always exploit we have limited informations (we end up with a suboptimal solution). Algorithms like $\epsilon$-greedy converge to the optimal choice, however we don't know how much we lose during the learning process.

- **Multi-Armed Bandit (MAB)**
  We can see multi-armed bandit as a specific case of an MDP: we have a single state $\mathcal{S} = \{s\}$, a set of actions (arms) $\mathcal{A} = \{a_1, .., a_N\}$, a state transition probability $p(s|a_i, s) = 1 \, \forall a_i$, a reward function $\mathcal{R}(s, a_i) = \mathcal{R}(a_i)$, a finite time horizon (discount factor $\gamma = 1$) and a set of initial probabilities $\mu^0(s) = 1$. The only thing that we need to have a full definition of the problem is the characterization of the reward $\mathcal{R})(a_i)$. The reward can be *deterministic* (single reward value for each arm), *stochastic* (the reward of an arm is drawn from a (usually unknown) distribution which is stationary over time), *adversarial* (an adversary chooses the reward we get from an arm at a specific round, knowing the algorithm we are using to solve the problem). In the adversarial case, the adversary knows the strategy, however he doesn't know the realization of the strategy.

- **Stochastic MAB**
  A multi-armed bandit problem is a touple $\langle \mathcal{A}, \mathcal{R} \rangle$, where $\mathcal{A}$ is a set of $N$ possible arms and $\mathcal{R}$ is a set of unknown random variable $\mathcal{R}(a_i)$ (with $\mathbb{E}[\mathcal{R}(a_i)] = R(a_i)$). At each round $t$ the agent selects a single arm $a_{i_t}$. The enviroment generates a reward $r_{i_t,t}$ drawn from $\mathcal{R}(a_{i_t})$. The agent updates the information by means of a history $h_t$. The final objective is to maximize the cumulative reward over a given time horizon $T$:

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

  Otherwise, the objective can be formulated in terms of regret (of not having taken the optimal action). We define the expected reward of the optimal arm $a^*$:

  $R^* = R(a^*) = \max_{a \in \mathcal{A}} \mathbb{E}[\mathcal{R}(a)]$

  At a given time step $t$, we select the action $a_{i_t}$ and we incur in a loss of:

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

  On average the algorithm loses:

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

  We want to minimize the expected regret suffered over a finite time horizon of $T$ rounds.
  We define the expected pseudo regret as:

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

  If we define $\Delta_i = R^* - R(a_i)$ and $N_t(a_i)$ as the number of times and arm $a_i$ has been pulled after a total of $t$ steps, we can rewrite the expected pseudo regret as:

  $L_T = T \cdot R^* - \mathbb{E}[\sum_{t=1}^{T} R(a_{i_t}) = \mathbb{E}[\sum_{t=1}^{T} R^* - R(a_{i_t})] = \sum_{a \in \mathcal{A}} \mathbb{E}[N_T(a_i)](R^* - R(a_i)) = \sum_{a \in \mathcal{A}} \mathbb{E}[N_T(a_i)]\Delta_i$

  *Theorem* (MAB lower bound): given a MAB stochastic problem, any algorithm satisfies:

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

where $KL(R(a_i), R(a^*))$ is the Kullback-Leibler divergence (measure of similarity, the smaller the more similar the two entities) between the two Bernoulli distributions $\mathcal{R}(a_i)$ and $\mathcal{R}(a^*)$.

A pure exploitation algorithm leads to select the action such that $a_{i_t} = \arg\max_a \hat{R}_t(a)$, where:

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

is the expected reward for an arm. This approach says to select the action for which the sampled mean of the rewards computed so far is maximal. This strategy tries to minimize the cumulated regred in a straightforward way. However, it might not converge to the optimal action. We need to perform also exploration.

- **Stochastic MAB: *frequentist* approach**

In the frequentist formulation $R(a_1), .., R(a_N)$ are unknown parameters. A policy selects at each time an arm based on the observation history. Instead of using the empirical estimate, we consider an upper bound $U(a_i)$ over the expected value $R(a_i)$. We need to compute:

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

The bound length $B_t(a_i)$ depends on how much information we have on arm, i.e. the number of times we pulled that arm so far ($N_t(a_i)$). The idea is: if we choose an arm many times then we want to rely almost on the empirical mean, if we hoose an arm few times then we want the bound to be big.

Applying the Hoeffding bound ($\mathbb{P}(X > \bar{X}_n + u) \leq e^{-2tu^2}$ for all $X_i$ iid random variables with support in $[0, 1]$ with mean $X$) to the upper bounds corresponding to each arm, we obtain:

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

Picking a probability $p = e^{-2N_t(a_i)B_t(a_i)^2}$, we obtain:

$B_t(a_i) = \sqrt{\frac{-\log p}{2N_t(a_i)}} \quad \Rightarrow \quad p = t^{-4} \quad \Rightarrow \quad \sqrt{\frac{2\log t}{N_t(a_i)}}$

*Upper Confidence Bound 1 (UCB1)* algorithm

For each time step $t$:

Compute : $\hat{R}_t(a_i) = \frac{\sum_{i=1}^{t} r_{i,t}\mathbb{I}_{a_i=a_{i_t}}}{N_t(a_i)} \quad \forall a_i$

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

Play arm : $a_{i_t} = \arg\max_{a_i \in \mathcal{A}} (\hat{R}_t(a_i) + B_t(a_i))$

In this way we're not choosing the best arm so far, but the arm with the best upper confidence bound. At finite time $T$, the expected total regret of the UCB1 algorithm applied to a stochastic MAB problem is:

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

- **Stochastic MAB: *bayesian* approach**

In the bayesian formulation $R(a_1), .., R(a_N)$ are random variable with prior distributions. A policy selects at each time an arm based on the observation history and on the provided priors.

The *Thomson sampling* consider a bayesian prior for each arm $f_1, .., f_N$ as a starting point. At each round $t$ we sample from each one of the distributions, obtaining $\hat{r}_1, .., \hat{r}_N$. We pull the arm $a_{i_t}$ with the highest sampled value $i_t = \arg\max_i \hat{r}_i$. Then we update the prior incorporating the new information.

In the case of Thomson sampling for Bernoulli rewards we use as prior conjugate distributions the $Beta(\alpha, \beta)$ and the *Bernoulli*. We start from all equal priors for all arms: $f_i(0) = Beta(\alpha_0 = 1, \beta_0 = 1) = \mathcal{U}([0, 1])$. Then, when we pull an arm $i$, if we obtain a success we update $f_i(t + 1) = Beta(\alpha_t + 1, \beta_t)$, if instead we obtain a failure we update $f_i(t + 1) = Beta(\alpha_t, \beta_t + 1)$.

The exploration/exploitation dilemma is represented into the updating and sampling from the priors/posteriors. At time $T$, the expected total regret of Thomson sampling algorithm applied to a stochastic MAB problem is:

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

- **Adversarial MAB**

In adversarial settings $\mathcal{R}$ is a reward vector for which the realization $r_{i,t}$ is decided by an adversary player at each turn. At each time step $t$ the agent selects an arm $a_{i_t}$. At the same time the adversary chooses rewards $r_{i,t}\ \forall i$. The agent gets the reward $r_{i_t,t}$. The final is to maximize the cumulative reward over a time horizon T:

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

An adversary choosing the rewards/regrets excludes deterministic algorithms. We define the weak regret as:

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

A lower bound is given by the *theorem* of minmax lower bound:

$\inf\sup \mathbb{E}[L_T] \geq \frac{1}{20}\sqrt{T \cdot N}$ *(we minimize (inf) the regret, while the adversary maximizes it (sup))*

The *EXP3* algorithm chooses an arm with probability:

$$\pi_t(a_i) = (1-\beta)\frac{w_t(a_i)}{\sum_j w_t(a_j)} + \frac{\beta}{N} \quad \text{where:} \quad w_{t+1}(a_i) = \begin{cases} w_t(a_i)e^{\eta \frac{r_{i,t}}{\pi_t(a_i)}} & \text{if } a_i \text{ has been pulled} \\ w_t(a_i) & \text{if else} \end{cases}$$

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

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