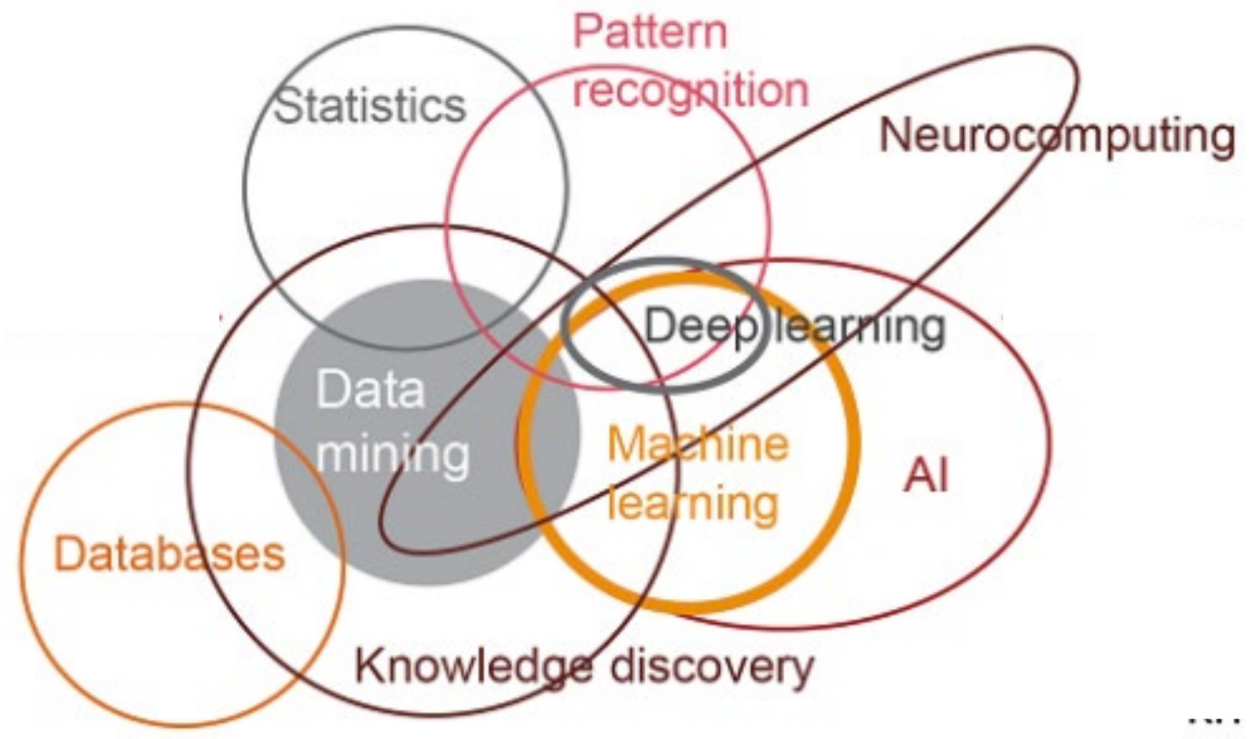
Machine Learning

github.com/martinopiaggi/polimi-notes

Table of Contents

1 Machine Learning

Machine Learning is a fascinating field of study enabling computers to learn and make inferences from data without the need for explicit programming. It involves training computer programs to improve their performance over time by learning from experience, ultimately making them perfect for stuff like predicting things or dealing with complex models that can't be solved exactly.



2 Supervised Learning

It is the most popular and well established learning paradigm. The goal is to learn a good approximation of an unknown function f that maps an input x to an output y with:

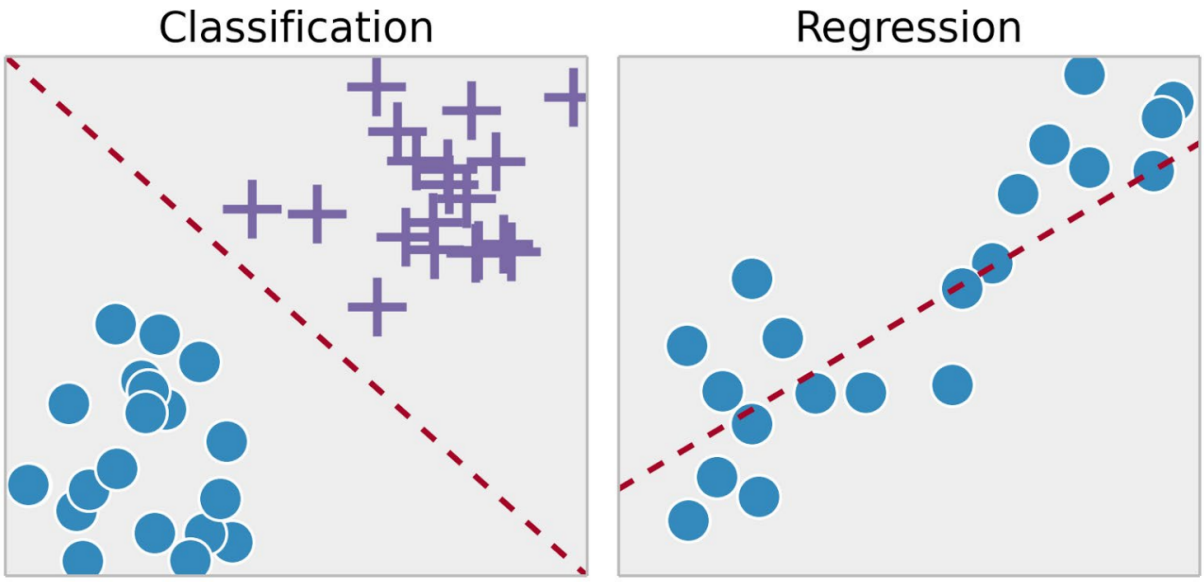
- Loss function: a function which says "how much is good the approximation (called h) of f?"
- Hypothesis space: a subset of the set of all possible function f

Supervised learning can be used every time we can't clearly explain which is this function. Why shouldn't we know what this function looks like?

- human cannot perform the task (DNA analysis)
- human cannot explain a clear algorithm (medical image analysis)
- the task continues to change over time (stocks price prediction)
- the task is user-specific (recommender system)

Supervised Learning is divided into:

- Linear Regression
- Linear Classification



3 Linear Regression

Linear regression is a technique used to model the relationships between observed variables in the context of linear models. A linear model is:

$$\hat{t} = y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{M-1} w_j x_j = \mathbf{w}^\top \mathbf{x}$$

A possible loss function (a way to evaluate the "quality" of my model) is Residual sum of squares :

$$\mathrm{RSS}(\mathbf{w}) = \sum_{n=1}^{N} \left(y\left(\mathbf{x}_n, \mathbf{w}\right) - t_n\right)^2$$

Actually we need a model that is **linear in the parameters** and we can define a model which is linear not on the input variable but linear on the parameter (or feature) vector. Same formula as above but applied on feature vectors:

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

A features vector $\phi(x)$ of a 2d point can be for example:

$\phi([a, b]) = [a, b, ab]^\top$.

3.1 Ordinary Least Squares

Ordinary Least Squares (OLS) is based on the idea of minimizing the sum of squared residuals. Residuals are the differences between the actual and predicted values of the output variable. The **square** is just used to avoid discrimination between positive and negative "residuals".

Compact way:

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

If we compute the second derivative on this we obtain the OLS formula:

$$w_{ols} = (X^T X)^{-1} X^T t$$

In the formula you provided, we have:

- w_{ols}: This represents the vector of estimated coefficients or weights for each feature in our linear equation.
- X: This represents the design matrix, which contains all the input features or independent variables from our dataset. Each row corresponds to one data point and each column corresponds to one feature.
- t: This represents the target variable or dependent variable from our dataset.

This formula allows us to find an optimal solution by minimizing the sum of squared errors between predicted values (obtained using these estimated coefficients) and actual target values from our dataset. Because the OLS is not feasible with large dataset, in practice we will use a sequential learning approach: instead of trying to solve the equation, we compute the gradient on just a single datapoint or a batch (subset) of datapoints.

$$\begin{aligned} & L(\mathbf{x}) = \sum_n L\left(x_n\right) \\ & \Rightarrow \mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \alpha^{(n)} \nabla L\left(x_n\right) \\ & \Rightarrow \mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \alpha^{(n)} \left(\mathbf{w}^{(n)^T} \boldsymbol{\phi}\left(\mathbf{x}_n\right) - t_n\right) \boldsymbol{\phi}\left(\mathbf{x}_n\right) \end{aligned}$$

where α is the learning rate. This is called a Stochastic Gradient Descent. Stochastic gradient descent, often abbreviated as SGD, is **an iterative optimization algorithm used to minimize an objective function**. It is widely employed in machine learning tasks such as regression and training neural networks.

Cool resource to visualize OLS: https://kwichmann.github.io/ml_sandbox/linear_regression_diagnostics/

3.2 Bayesian Linear Regression

While OLS is a frequentist approach, Bayesian Linear Regression is a probabilistic one. We define a model with unknown parameters and specify a prior distribution to account for our uncertainty about them:

$$p(\text{ parameters } \mid \text{ data }) = \frac{p(\text{ data } \mid \text{ parameters })p(\text{ parameters })}{p(\text{ data })}$$

or written in a compact way:

$$p(w \mid D) = \frac{p(D \mid w)p(w)}{p(D)}$$

- $p(D \mid w)$ is the likelihood: the probability of observing the data D given the parameters (w).
- $p(\mathbf{w}|D)$ is the posterior probability of parameters w given training data.
- $p(D)$ is the marginal likelihood and acts as normalizing constant
- $p(w)$ is our "prior knowledge" about the parameters. This is one of the fundamental differences with the OLS approach … it's like having a prior hint about the parameters.

We can model the prior $p(w)$ as a gaussian:

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

and also the likelihood as a gaussian so that the posterior probability $p(\mathbf{w}|D)$ will be gaussian:

$$p\left(\mathbf{w} \mid \mathbf{t}, \mathbf{\Phi}, \sigma^2\right) \propto \mathcal{N}(\mathbf{w} \mid \mathbf{w}_0, \mathbf{S}_0)\, \mathcal{N}\left(\mathbf{t} \mid \mathbf{\Phi}\mathbf{w}, \sigma^2\mathbf{I}\right)$$

In the same way we can use a Beta distribution as prior, a Bernoulli as likelihood so to have at end a posterior probability which is a Beta. This choices are useful to exploit this Bayesian approach for sequential learning: we compute posterior with initial data and later, adding additional data, the posterior becomes the prior (recursion).

4 Linear Classification

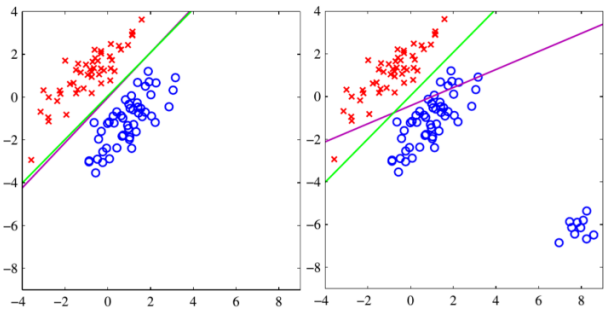
Linear classification is a type of supervised learning where the goal is to assign an input vector x to one of K discrete classes. There are different approaches: - **Discriminant function approach** directly provides the predicted class $f(x) = Ck$: - Ordinary Least square approach - Perceptron - K-NN - **Probabilistic discriminant approach** provides the predicted probability - Logistic Regression - Probabilistic generative approach: - Naïve-Bayes

4.1 OLS for classification

Least squares could be an approach for classification but is problematic in some cases.

Least square is very sensitive to outliers. Least square tries to find a line which is the most close to all points. It does so evaluating the square distance between the samples and the line. It means that an outlier will have a greater impact on the line position because it will be more distant with respect to the probable samples.

This because one of the assumptions of least squares is to have a linear noise which follows a Gaussian distribution: the model generating the data is a linear model plus a noise with gaussian distribution with some unknown standard deviation (hopefully not too big). If we have outliers, it's like the noise is not following any gaussian distribution. So this, in practice, makes least square not able to find good decision boundaries.



4.2 Perceptron

The perceptron is an algorithm for online supervised learning of binary classifiers. The algorithm finds the separating hyperplane by minimizing the distance of misclassified points to the decision boundary. Sum of the distances of misclassified points from the current separating hyperplane:

$$L_P(w) = -\sum_{n \in M} w^T \Phi\left(x_n\right) t_n$$

It's designed to penalize the outliers:

- samples classified correctly do not contribute to L
- each misclassified sample contributes as $w^T \Phi\left(x_i\right) t_i$

The perceptron is an example of linear discriminant model. It has an online linear classification algorithm. The main function to decide if it's in a positive or negative class is:

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

where f is a step function (sign function):

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

In perceptron we apply a Stochastic Gradient Descent: in case of correctly classified samples $\phi(x)$, the loss is zero so we do not do any update on the parameters, while we update the weights when we misclassified samples:

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

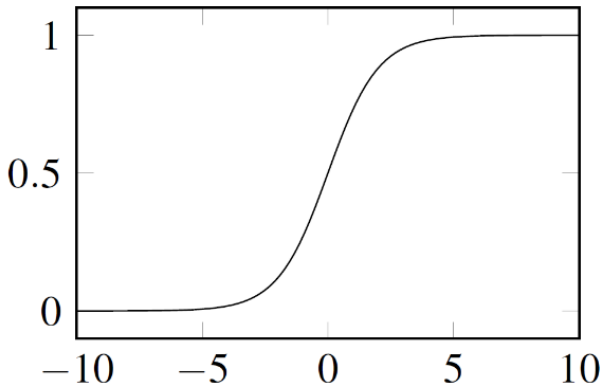
$$w \leftarrow w + \alpha \phi(x)\, t_2$$

Some other details:

- If the dataset fed to a perceptron is not linearly separable, the Perceptron algorithm will not terminate.
- If the dataset is linearly separable, Perceptron will converges in a finite number of steps.
- Many hyperplanes exist and Perceptron converges on one, and it converges depending on the initialization and the order of how the data is processed

4.3 Logistic regression

While Least squares method is highly sensitive to outliers, logistic regression is not. Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary variable. So it is capable of resolve two-class classification. Logistic regression is a discriminative model so we model directly the posterior probability $P(C_k|\Phi)$. In detail we use a logistic sigmoid function:



In logistic regression, we assume that the hypothesis space is distributed as a Bernoulli because we are dealing with binary classification problems. The **Bernoulli distribution**, is the discrete probability distribution of a random variable which takes the value 1 with probability p and the value 0 with probability $q = 1 - p$. In this case, the Bernoulli distribution is used to model the probability of a certain class label being assigned given a set of input features. The sigmoid function plays a crucial role in logistic regression. It is used to map the output of our linear model (which can take any real value) into a range between 0 and 1:

$$P(C_1 \mid \Phi) = \sigma\left(w^T \Phi\right) = \frac{1}{1 + e^{-w^T \Phi}}$$

and for the negative class:

$$P(C_2 \mid \Phi) = 1 - P(C_1 \mid \Phi) = 1 - \frac{1}{1 + e^{-w^T \Phi}}$$

Example:

$$P\left(t_1 = 1 \mid \phi_1, w\right) \quad P\left(t_3 = 1 \mid \phi_3, w\right)$$
$$P\left(t_2 = 0 \mid \phi_2, w\right)$$

So we can take as loss function the product of all the probabilities $P(t_n)$.

$$R_L = \prod_{i=1}^{3} P_i$$

Then we want to maximize this probability. If a prediction is > 0.5 it will be in positive class

$$P(w|X, w) = \prod_{n=1}^{N} y_n^{t_n}(1 - y_n)^{1-t_n}$$

We convert the maximization into a minimization of the negative log-likelihood:

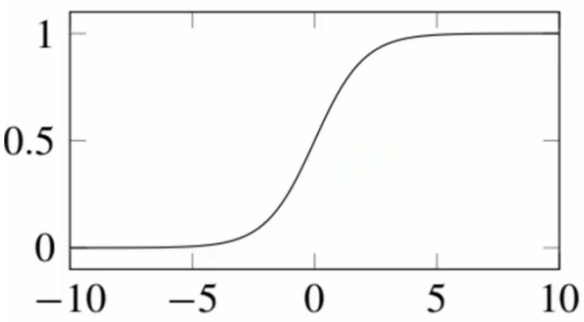
$$L(\mathbf{w}) = -\ln p(\mathbf{t} \mid \mathbf{X}, \mathbf{w}) = -\sum_{n=1}^{N} \left(t_n \ln y_n + (1 - t_n) \ln\left(1 - y_n\right)\right) = \sum_{n=1}^{N} L_n$$

The loss function of LR is convex but LR does not admit a closed-form solution of the optimal weights. A closed-form solution refers to a mathematical solution that can be expressed using a finite number of mathematical operations, such as algebraic equations, integrals, and derivatives. No closed-form solution means: it doesn't allow for the use of analytical methods to find the optimal parameters. We can compute the derivative (gradient).

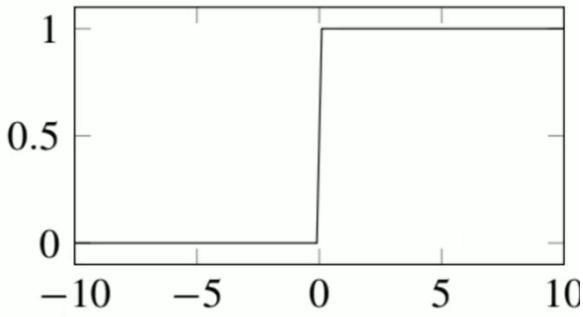
The logistic regression classifier converges to a solution even when the training set is not linearly separable. Indeed, LR always provides the weights minimizing the corresponding loss function, being the objective function convex. If the training set is not linearly separable, the provided weights would not correspond to a separating hyperplane. The more the number of features we are considering, the more the model il complex and prone to overfitting.

4.3.1 Logistic regression as a generalization of perceptron

The perceptron classifier and the logistic regression classifier are both generalized linear models. Indeed, their prediction function can be both written as $f\left(x^T w + w_0\right)$. In particular, for the perceptron we have $f(y) = \begin{cases} +1 & \text{if } y \geq 0 \\ -1 & \text{otherwise} \end{cases}$ and for logistic regression we have $f(y) = \sigma(y)$, where $\sigma(\cdot)$ is the sigmoid function.



$$y(\mathbf{x}, \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \boldsymbol{\phi}(\mathbf{x})}}$$



$$y(\mathbf{x}, \mathbf{w}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \boldsymbol{\phi}(\mathbf{x}) > 0 \\ 0 & \text{otherwise} \end{cases}$$

4.4 Naïve Bayes

The main idea behind Naïve Bayes is to calculate the probability of a certain class label given a set of input features. It assumes that each feature contributes independently and equally to the probability calculation, which is why it's called "naïve". NB calculates the posterior probability $P(A|x_1, x_2, \dots x_n)$ and $P(B|x_1, x_2, \dots x_n)$ using Bayes theorem:

$$P(A|x_1, x_2, \dots x_n) = \frac{(P(x_1|A) * P(x_2|A) * \cdots * P(xn|A) * P(A))}{P(x_1, x_2, \dots x_n)}$$

$$P(B|x_1, x_2, \dots x_n) = \frac{(P(x_1|B) * P(x_2|B) * \cdots * P(xn|B) * P(B))}{P(x_1, x_2, \dots x_n)}$$

Here:

- The terms like $P(A)$ and $P(B)$ are prior probabilities representing our belief about the likelihood of each class occurring.
- The terms like $P(x_i|A)$, represent conditional probabilities indicating how likely each feature x_i occurs given a specific class A.
- The denominator term represents evidence or normalization factor.

During training phase:

- Prior probabilities can be estimated by counting occurrences of each class in the training data.
- Conditional probabilities can be estimated by assuming a specific probability distribution for each feature, such as Gaussian (for continuous features) or multinomial (for discrete features).

During prediction phase:

- Given a new sample with input features $x_1, x_2, ..., x_n$, NB calculates the posterior probabilities for each class using the trained model.
- The class with the highest posterior probability is then assigned as the predicted class label.

$$P(C_k \mid x) = \frac{P(x \mid C_k)P(C_k)}{P(x)}$$

4.5 K-Nearest Neighbor

K-nearest neighbors (KNN), instead of estimating fixed parameters, stores all available training instances in memory during training. During prediction with KNN, it calculates distances between new input samples and all stored instances in order to identify the k nearest neighbors. The predicted output is then determined based on majority voting or averaging among those neighbors. For example in KNN it can be used the Euclidean distance. For each new sample are queried the k nearest points and then predicted the class. Obviously if there is a tie, KNN is unable to provide a prediction, unless a the breaking rule is provided.

KNN 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. So the more the k parameter is small the more it is prone to overfit the original dataset. With $K = 1$ all the data in the training set are perfectly classified. Said in another way, very low k will have high variance and low bias, while a high k will have a low variance but high bias.

4.6 Parametric and non parametric

Two distinction:

- **Non-parametric** models do not involve any weights and require no training. They simply query over the dataset to make predictions for both regression and classification problems.Remember that non parametric method has not training: every time you have constraints on memory or computation you prefer not parametric method.
- A **parametric** method is trained on a dataset and then makes inference with the developed weights. Usually Parametric approaches are faster in the prediction phase.

Scenario	I prefer parametric or not parametric approach?
Large dataset (big data scenario);	Parametric
Embedded system or component with reduced computational capabilities.	Non parametric
Prior information on data distribution	Parametric
Learning in a Real-time scenario	Non parametric

- In the case we have a large dataset it is better to have a model which is able to capture the characteristics through parameters. Indeed, a non-parametric method would require storing the whole dataset and perform queries on it to perform predictions.
- In the case of an embedded system, a parametric approach is possible if trained on another machine, non parametric if there is enough memory.
- Learning in a real-time scenario is impossible: just memorise and queries like KNN for example.

4.7 Performance measures in Binary Classification

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 true positive	fp false positive
Predicted Class: 0	fn false negative	tn true negative

The following metrics can be evaluated:

- Accuracy: $Acc = \frac{(t_p+t_n)}{N}$ is the fraction of predictions our model got right.
- Precision: Pre $= \frac{t_p}{t_p+f_p}$ answers to *"What proportion of positive identifications was actually correct?"*
- Recall: $\mathrm{Rec} = \frac{t_p}{t_p+f_n}$ answers to *"What proportion of actual positives was identified correctly?"*
- F1 score: $F1 = \frac{2 * \text{Pre} * \text{Rec}}{\text{Pre} + \text{Rec}}$

Other metrics to evaluate the model:

$$\mathrm{RSS}(\mathbf{w}) = \sum_{n=1}^{N} \left(\hat{t}_n - t_n\right)^2$$

where $\hat{t}_n = y(x_n, w)$$

$$\mathrm{MSE} = \frac{\mathrm{RSS}(w)}{N}$$

$$\mathrm{RMSE} = \sqrt{\frac{\mathrm{RSS}(\mathbf{w})}{N}}$$

Another important to evaluate the linear model is R-squared: (the coefficient of determination)

$$R^2 = 1 - \frac{\mathrm{RSS}(\mathbf{w})}{\mathrm{TSS}}$$

where TSS is the Total Sum of Squares:

$$\text{TSS} = \sum_{n=1}^{N} (\bar{t} - t_n)^2$$

with

$$\bar{t} = \frac{1}{N} \sum_{n=1}^{N} t_n$$

TSS (Total Sum of Squares) can be seen as a measure of the total variance in the target variable. You happy when R^2 is close to 1 .

4.8 Multiple classes

Different approaches to multiple classes classifiers:

- **one-versus-the-rest**: $K - 1$ classifiers of which solves 2 class problem, separating points of C_k from points not in that class.
- **one-versus-one**: $\frac{K(K-1)}{2}$ classifiers, each classifier for each pair of classes. This introduces ambiguity.
- **k-classifier**: K classifier each one will try to predict only a class, and each time the output will be the class with maximum probability.

Recap on methods seen:

The **perceptron** deals with binary classification problems. It can be employed for K multi-class classification training K one-versus-the-rest classifiers.

Logistic regression classifier deals with binary classification problems. It can be extended to multi-class classification, using the softmax transformation. The Softmax equation is as follows:

$$p(y = j \mid \mathbf{x}) = \frac{e^{\left(\mathbf{w}_j^T \mathbf{x} + b_j\right)}}{\sum_{k \in K} e^{\left(\mathbf{w}_k^T \mathbf{x} + b_k\right)}}$$

Naïve Bayes is able to deal with multi-class classification problems by using a categorical distribution for the class distribution prior $p(C_k)$ and estimating the posterior $P(x|C_k)$ for each class.

KNN is naturally able to deal with multi-class classification problems by using majority voting to decide the class. However, we need to carefully choose the way we are breaking ties since this might be crucial in the case of many classes.

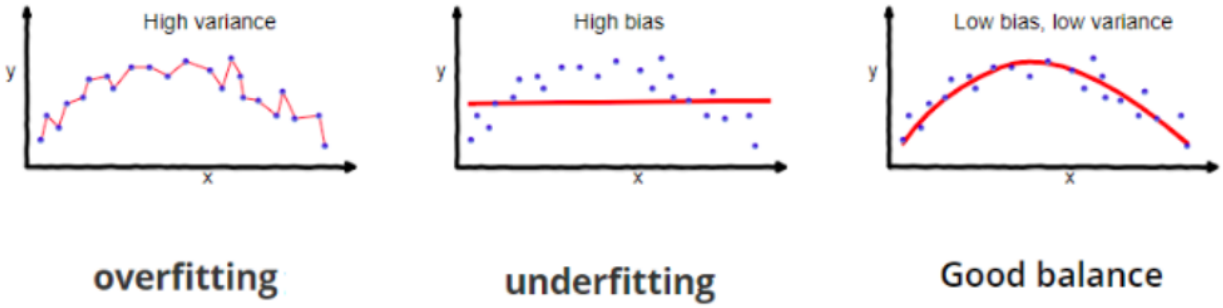
5 Bias-Variance

The Bias-Variance is a framework to analyze the performance of models.

Typically there is a trade-off between bias-variance:

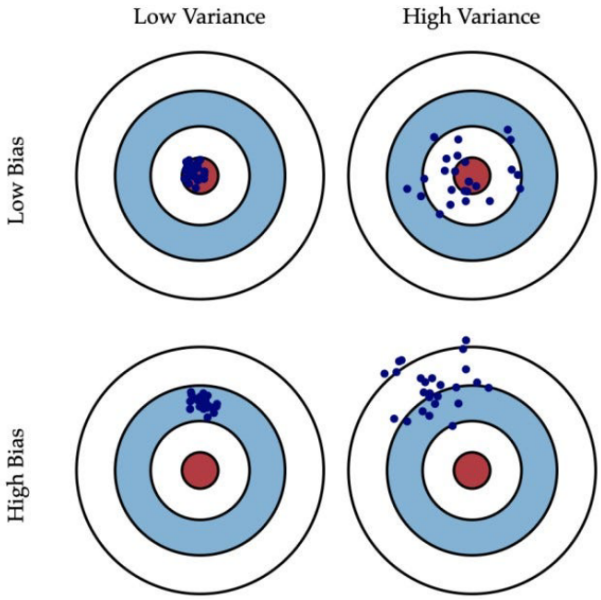
- bias measures the difference between truth and what we expect to learn
- variance in general measures the spread or dispersion of a set of data points around their mean (average) value. It indicates how much it changes according to the dataset used.

The more complex is the model and the higher will be the variance:



{width=50%}

The term “degree of freedom” refers to the number of parameters that can vary in a model without violating any constraints. It represents the flexibility or complexity of a model. Increasing the number of samples doesn’t change the bias of my model: the bias of my model is the property of my model. If I cannot represent the process generating data with my model doesn’t change if I am increasing the data. Increasing the number of samples can instead reduce the variance.
Another visualization:



A hypothesis space H is the set of all possible models that can be learned by an algorithm. It represents the range of functions or mappings that the algorithm can choose from to make predictions based on input data. If H_1 is a smaller subset of H_2:

- the bias of H_2 will be smaller than the bias of H_1.
- the variance of H_1 is smaller than that of H_2.

5.1 Bias-Variance decomposition

The **bias–variance decomposition** is a way of analyzing a learning algorithm's expected generalization error with respect to a particular problem as a sum of three terms:

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

Some considerations:

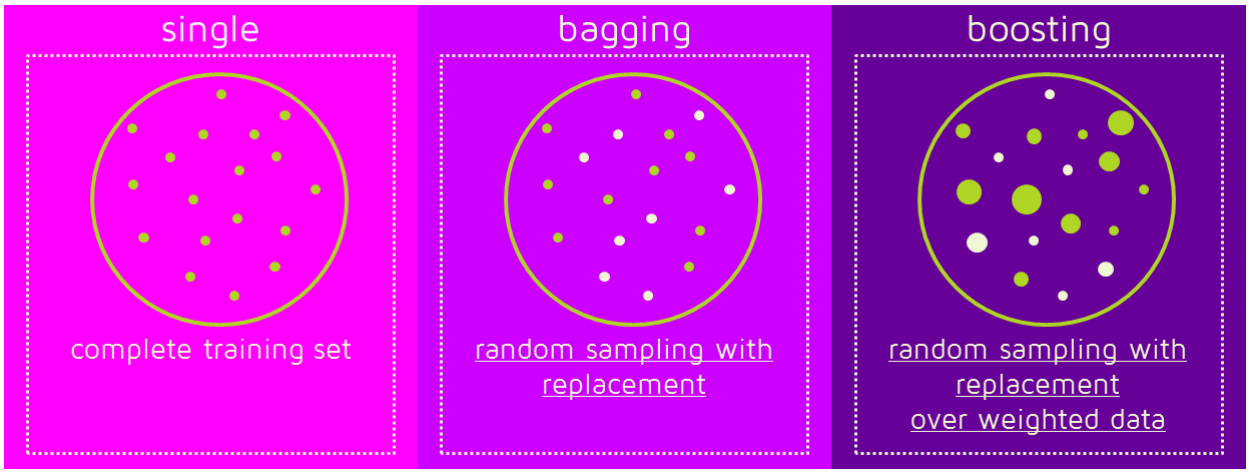
- σ^2 is also called **Irreducible error** while the **reducible error** is composed by the other components.
- It's always a good thing increase the number of samples, since it decreases the variance of the model and therefore the reducible error: since it is composed by the sum of the squared bias.
- Increasing the complexity of the model will decrease your Reducible Error ? It depends… this is the whole point in model selection: balancing between the bias and the variance.

Note that the computation of the bias-variance decomposition is possible only theoretically and it is possible only if we know the true process.

5.2 Ensemble models

Improving the trade off between bias and variance:

- **Bagging**: reduce variance without increasing bias
- **Boosting**: reduce bias without increasing variance



5.2.1 Bagging

Bagging stands for Bootstrap Aggregation: it generates N datasets applying random sampling with replacement.

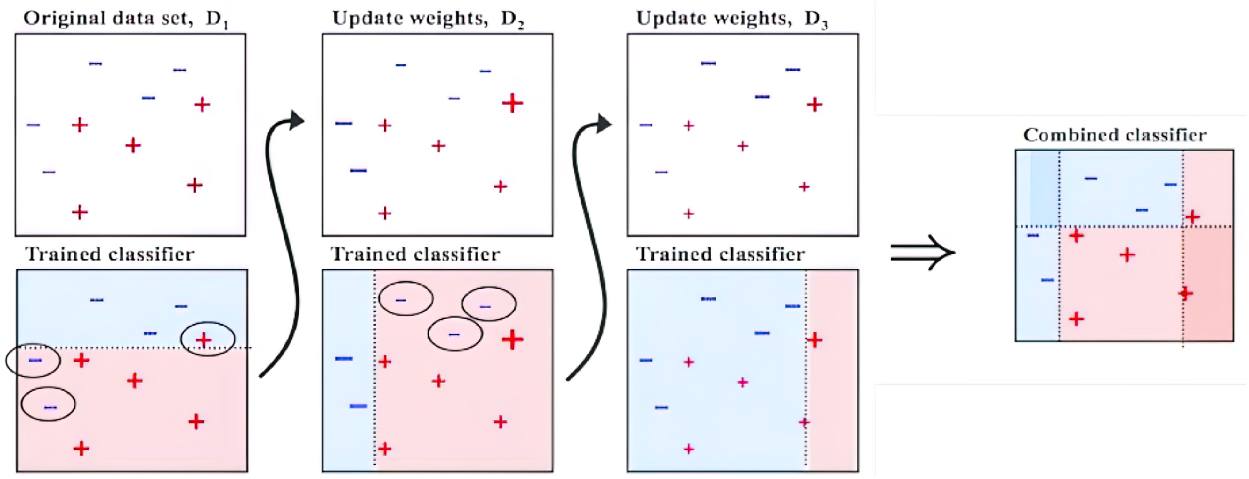
- Reduces variance
- Not good for stable learners (A learning algorithm is stable if changing input slightly won't put effect on the hypothesis)
- Can be applied with noisy data
- Usually helps but the difference might be small
- Parallel by design

5.2.2 Boosting

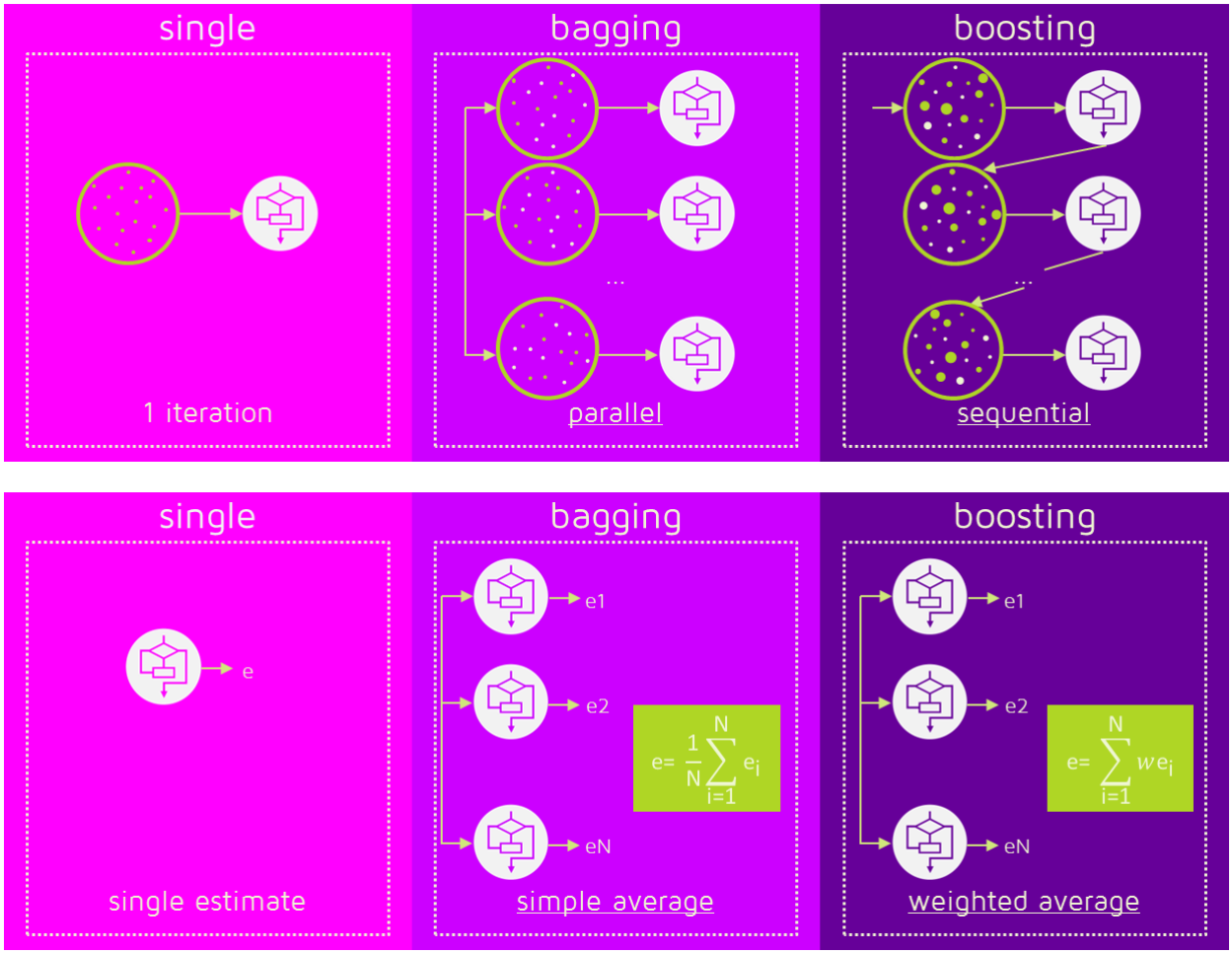
- Reduces bias (generally without overfitting)
- Works with stable learners
- Might have problem with noisy data
- Not always helps but it can makes the difference
- Serial by design

Sequentially means that we train a model based on the prediction of the previous. The steps to perform boosting are the following,

- Give an equal weight to all training samples
- Train a weak model on the training set
- Compute the error of the model on the training set
- For each training samples increase its weight if the model predicted wrong that sample. Doing so we obtain a new training set.
- Iterate the training on the new training set, error computation and samples re-weighting until we are satisfied by the result

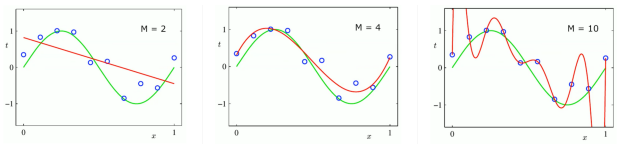


The final prediction is the weighted prediction of every weak learner. In practice we are combining a set of sequential underfitting model. Doing so, we have low variance and the bias is improved by combining the weak learner to form a strong learner. On average, boosting helps more than bagging, but it is also more common for boosting to hurt performance.



6 Model Selection

6.1 Regularization



Increasing the order of the polynomial model reduces the smoothness of the approximation. The weights also increase in size with higher order models. To address this, we can use a **regularization** coefficient, which can be implemented through methods like **ridge** regression or **lasso** regression.

6.1.1 Ridge

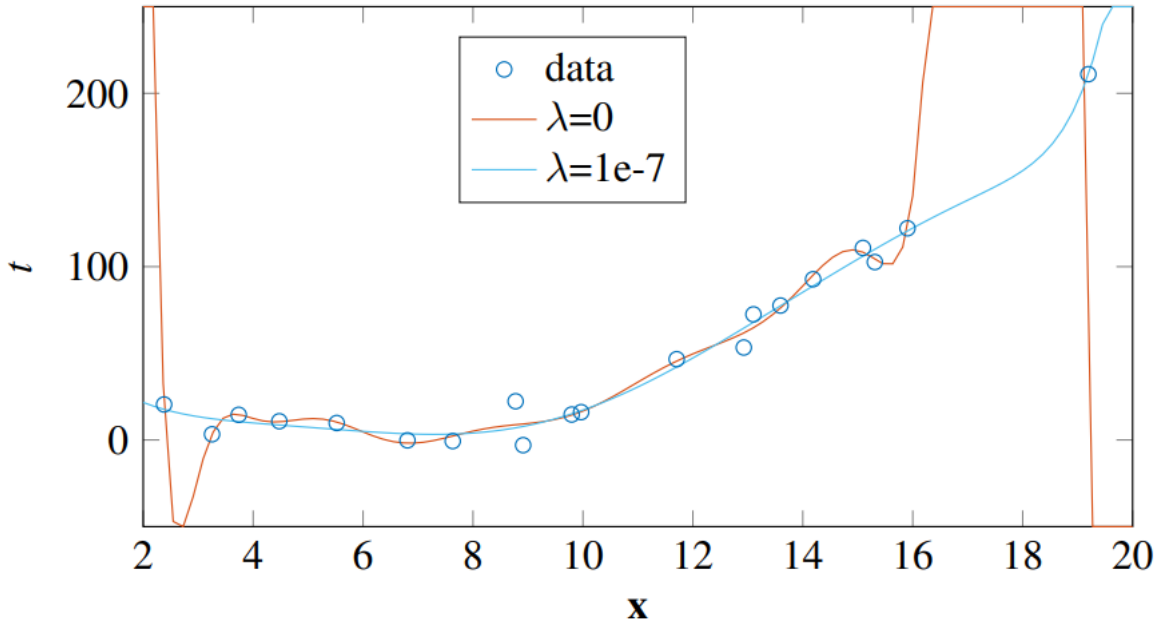
Ridge regression is a technique that adds a penalty term to the ordinary least squares (OLS) method to prevent overfitting. The formula for the Ridge loss function is:

$$L(\mathbf{w}) = \frac{1}{2}\,\mathrm{RSS}(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

In this formula, $\mathrm{RSS}(\mathbf{w})$ represents the residual sum of squares, which measures the difference between predicted and actual values. The second term $\frac{\lambda}{2}\|\mathbf{w}\|_2^2$ is the regularization term that penalizes large coefficients by adding their squared magnitudes multiplied by a tuning parameter λ. This helps to shrink or regularize the coefficients towards zero. When performing Ridge Regression the OLS formula is modified in this way:

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

the magic:



Things to remember:

1) training RSS -> increases !
2) test RSS -> maybe increase/decrease OK (U shape)
3) the variance -> decrease surely
4) square bias -> increase surely
5) irreducible error -> doesn't change

6.1.2 Lasso

Lasso is a regularization method that introduces sparsity constraints by setting many weights to 0. This helps identify the most important features, leading to more **interpretability**. However, Lasso is nonlinear in the t_i and does not have a closed-form solution, making it computationally complex with quadratic complexity. Lasso regression is another regularization technique that also prevents overfitting but has a slightly different penalty term compared to Ridge regression. The formula for Lasso loss function is:

$$L(\mathbf{w}) = \frac{1}{2}\,\mathrm{RSS}(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|_1$$

Similar to Ridge regression, it includes both an RSS term and a regularization term with coefficient λ. However, instead of using squared magnitudes like in Ridge regression ($\|\cdot\|_2$), it uses absolute magnitudes ($\|\cdot\|_1$). The 1-norm $\|\cdot\|_1$ is simply the sum of the absolute values of the columns:

$$\sum_{i=1}^{n} x_i$$

This leads to a sparsity-inducing effect, as it tends to set some coefficients exactly to zero.

The $||\cdot||_1$ notation denotes the L-1 norm or Manhattan norm of vector $\mathbf{w}$.

6.1.3 Elastic Net

Elastic Net regression is a combination of Ridge and Lasso regularization techniques. It includes both penalty terms in its loss function:

$$L(\mathbf{w}) = \frac{1}{2}\,\mathrm{RSS}(\mathbf{w}) + \frac{\lambda_1}{2}\|\mathbf{w}\|_2^2 + \frac{\lambda_2}{2}\|\mathbf{w}\|_1$$

6.2 Features Selection

6.2.1 Filter

The filter method in feature selection is a technique used to select the most relevant features from a dataset. It typically works in this way:

1. For each feature calculate a statistical measure (for example the Pearson correlation coefficient) between x_k and target y
2. Rank the features based on their scores
3. Select the features with higher Pearson correlation coefficient (or the statistical measure choosen)
4. Remove irrelevant features

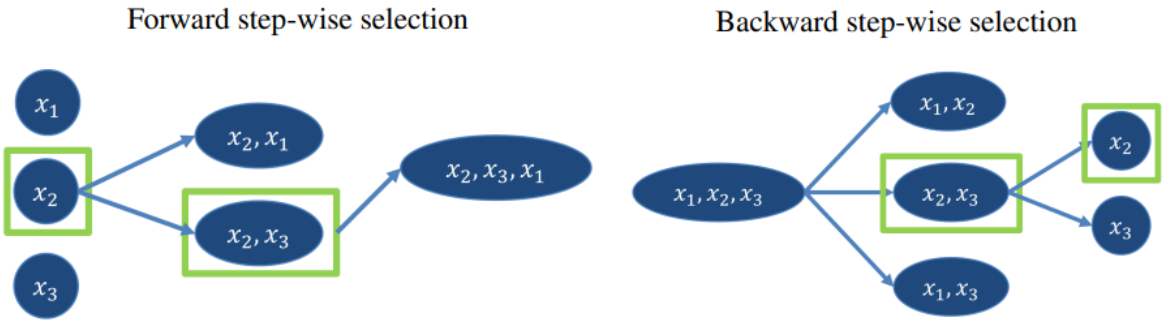
The advantage of using filter methods is that they are computationally efficient and **independent** of any specific machine learning algorithm. Pearson correlation coefficient between x_k and target y :

$$\hat{\rho}(x_j, y) = \frac{\sum_{n=1}^{N}(x_{j,n} - \bar{x}_j)(y_n - \bar{y})}{\sqrt{\sum_{n=1}^{N}(x_{j,n} - \bar{x}_j)^2}\sqrt{\sum_{n=1}^{N}(y_n - \bar{y})^2}}$$

where $\bar{x}_j = \frac{1}{N}\sum_{n=1}^{N} x_{j,n}$ and $\bar{y} = \frac{1}{N}\sum_{n=1}^{N} y_n$ and j is for each feature on M features, while N is the number of samples.

6.2.2 Wrapper

The wrapper method is a technique which "wraps" the used ml algorithm and treats the feature selection process as a **search problem**. This search problem can be solved in two ways: brute-force or step-wise evaluation of a subset of possibilities.

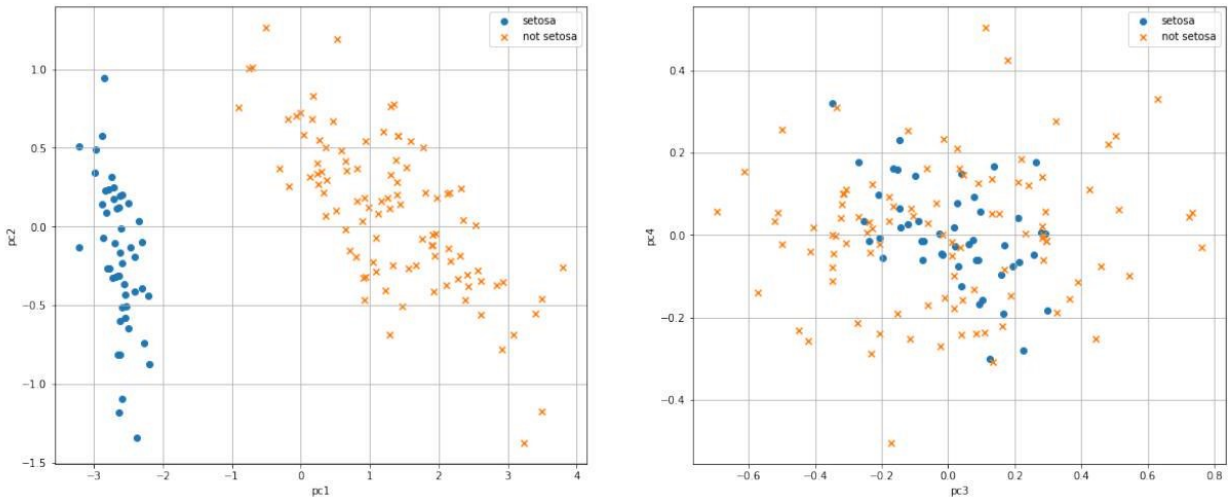


6.2.3 Embedded

Embedded methods refer to algorithms that perform feature selection during the model training process. These methods **incorporate feature selection directly into the learning algorithm**, rather than treating it as a separate preprocessing step. Lasso can be classified as embedded since it identifies and selects the most important features during the training.

6.3 Features Extraction

Feature Extraction reduces the dimensionality of the dataset by selecting only the number of principal components retaining information about the problem. Reducing the dimensionality of the input means maybe to be able to visualize the data:



While all the previous seen methods perform selection, feature **extraction** methods provide novel features that are linear combinations of the original ones.

6.3.1 Principle Components Analysis

In Principal Component Analysis (PCA) is an unsupervised dimensionality reduction technique which extracts some features from a dataset. The steps of PCA (deterministic) are:

1) Translate the original data $\mathbf{X}$ to $\tilde{\mathbf{X}}$ s.t. they have zero mean. Even if all the input features are on very similar scales, we should still perform mean normalization.
2) Compute the covariance matrix of $\tilde{\mathbf{X}}$, $\mathbf{C} = \tilde{\mathbf{X}}^\top \tilde{\mathbf{X}}$. Covariance matrix captures relationships between features. It measures how variables vary together or independently.
3) The **eigenvectors** of $\mathbf{C}$ are the principal components, which explain most of the variance in our data.
4) The eigenvector $\mathbf{e}_1$ corresponds to the largest eigenvalue λ_1. Sorting features by their variance helps identify which features have higher variability and contribute more to capturing patterns or trends in our dataset.
5) k is the number of features to keep (it's the dimension of space where the features are projected).

This is the code:

```
import numpy as np
X_tilde = X - np.mean(X, axis=0)
C = np.dot(X_tilde.T, X_tilde)
eigenvalues, W = np.linalg.eig(C)
T = np.dot(X_tilde, W[:,:2])
```

Nomenclature of PCA stuff:

- the mean normalized sample vector is $\tilde{x}$
- The **loadings** $\mathbf{W} = (\mathbf{e}_1 \, |\mathbf{e}_2| \ldots | \, \mathbf{e}_M)$ is matrix of the principal components.
- The scores are the transformation of the input dataset $t = \tilde{x}\mathbf{W}$

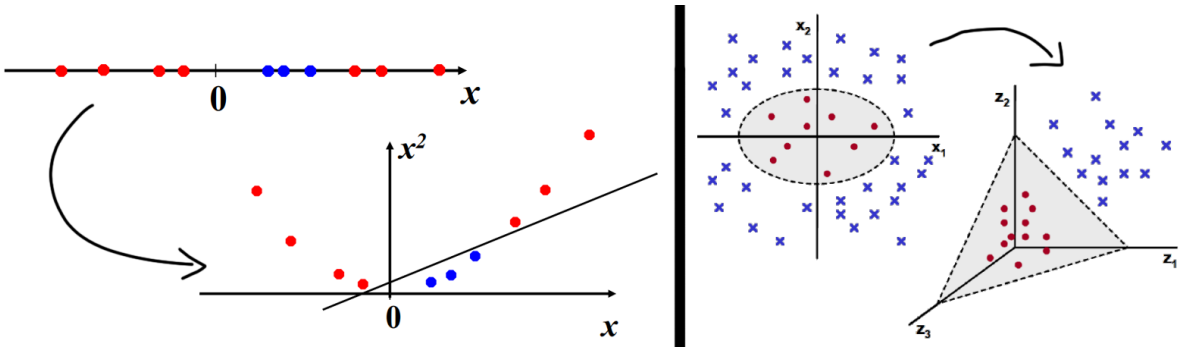
- variance: $(\lambda_1, \ldots, \lambda_M)^\top$ vector of the variance of principal components

There are a few different methods to determine how many feature to choose

- Keep all the principal components until we have a cumulative variance of $90\% - 95\%$
- Keep all the principal components which have more than 5% of variance (discard only those which have low variance)
- Find the elbow in the cumulative variance

7 Kernel Methods

Key idea: sometimes linear separators cannot be found in the input space. **But** it's possible to map the input space to a higher-dimensional feature space to make it easier to separate classes or find patterns.



The clever approach of Kernel methods is that instead of directly mapping the data to a higher-dimensional space and performing computations, kernel methods represent the data through pairwise similarity comparisons between the original data observations (basically we skip the mapping $\phi(x)$ and we only perform the similarity comparisons). This means that instead of explicitly applying transformations and representing the data with transformed coordinates in the higher-dimensional feature space, kernel methods rely on comparing the similarities between data points. By utilizing this approach, kernel methods enable efficient calculations without explicitly mapping to a higher dimension. This "trick" permits us to find the optimal separating hyperplane in this higher dimensional space without having to calculate or in reality even know anything about $\phi(x)$.

Computationally speaking is very very convenient: we can represent features expansion that include billions of elements with very simple kernel which need few operation to be computed.

7.0.1 Constructing Kernels

Theorem Mercer's theorem tells us that any continuous, symmetric, positive semi-definite kernel function $k(x, x_0)$ can be expressed as a dot product in a high-dimensional space:

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

Since demonstrating the positive semi-finite function can be challenging, we can make new kernels using well-known kernels as **building blocks**. 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) $k(\mathbf{x}, \mathbf{x}') = f(\mathbf{x})k_1(\mathbf{x}, \mathbf{x}')f(\mathbf{x}')$ where $f(\cdot)$ is any function
3) $k(\mathbf{x}, \mathbf{x}') = q(k_1(\mathbf{x}, \mathbf{x}'))$, where $q(\cdot)$ is a polynomial with non-negative coefficients
4) $k(\mathbf{x}, \mathbf{x}') = \exp(k_1(\mathbf{x}, \mathbf{x}'))$
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) $k(\mathbf{x}, \mathbf{x}') = k_a(\mathbf{x}_a, \mathbf{x}'_a)k_b(\mathbf{x}_b, \mathbf{x}'_b)$

Note that every kernel has to be symmetric and continuous.

7.0.2 Gaussian Processes

Gaussian processes are versatile algorithms that can be used for both regression and classification. GPs aren't parametric methods: they are memory-based ones and they use a big matrix computed with the samples to predict new samples. A Gaussian process is a probability distribution over possible functions $y(x)$ such that the set of values $y(x)$ evaluated at an arbitrary set of points $x_1, \ldots, x_N$ jointly have a Gaussian distribution. K or C_N measures the similarity between all inputs pairs and so the correlation of the outputs y.

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

This matrix is also called "Gram" since in linear algebra, the Gram matrix represents pairwise inner products between vectors in an inner product space. When we have to predict the target of a new point we use the K matrix:

$$\mathbf{C}_{N+1} = \begin{pmatrix} \mathbf{C}_N & \mathbf{k} \\ \mathbf{k}^\top & c \end{pmatrix}$$

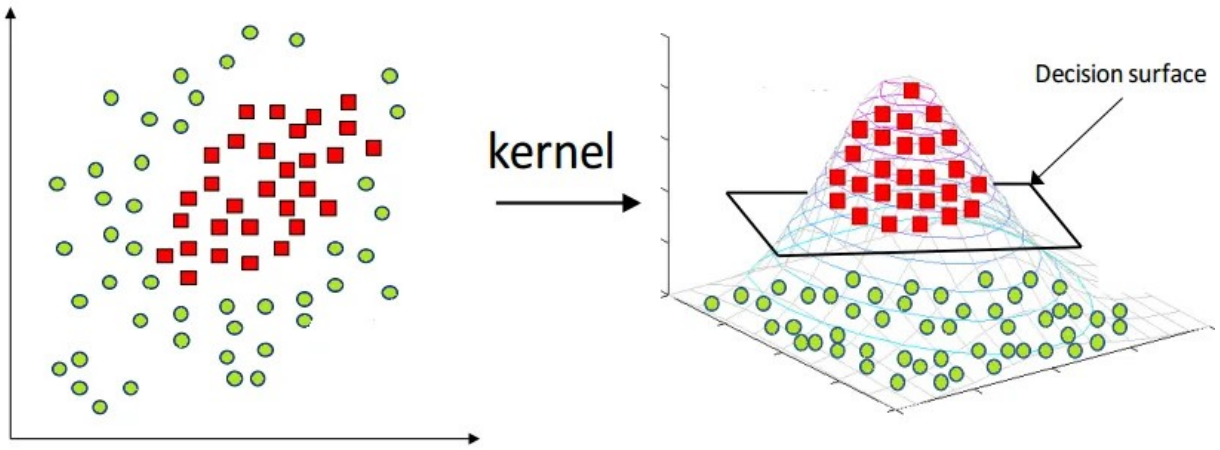
We can compute both the mean and variance of any new target, defining a new probability distribution:

- $m(\mathbf{x}_{N+1}) = \mathbf{k}^\top C_N^{-1}\mathbf{t}$
- $\sigma^2(\mathbf{x}_{N+1}) = c - \mathbf{k}^\top C_N^{-1}\mathbf{k}$

with k is the kernel between the new target and all the training data points (a vector where we keep in memory all the kernels).

7.0.3 SVM Support Vector Machines

SVMs try to find a decision boundary that maximizes the distance between support vectors (data points closest to the decision boundary) from both classes. To handle non-linearly separable datasets, SVMs use kernel functions mentioned earlier such as linear kernels, polynomial kernels, radial basis function (RBF) kernels which implicitly map input features into higher-dimensional spaces where they become linearly separable.



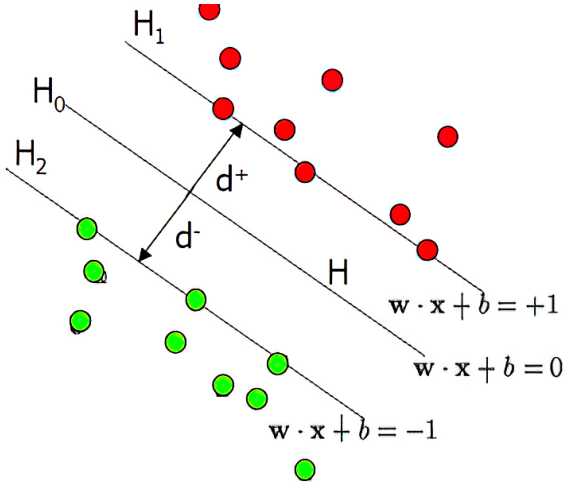
In SVM (Support Vector Machines), the decision boundary is defined by a hyperplane that separates different classes of data points. The support vectors are the data points that lie closest to this decision boundary.

To determine if a particular data point, such as x_1, is a support vector, we need to check if it lies on or within the margin region around the decision boundary. This margin region is defined by two parallel hyperplanes: one on each side of the decision boundary. The equation for these hyperplanes can be written as:

$w^T x + b = 1$ (for the upper hyperplane) $w^T x + b = -1$ (for the lower hyperplane)

Here, w represents the weight vector and b represents the bias term in SVM. The dot product $w^T x_1 + b$ gives us a measure of how far away x_1 is from these hyperplanes.

If $w^T x_1 + b \leq 1$, then it means that x_1 lies inside or on top of the upper margin plane. Similarly, if $w^T x_1 + b \geq -1$, then it means that x_1 lies inside or on top of the lower margin plane.



Therefore, when checking if a data point like x_1 is a support vector, we want to ensure that it satisfies both conditions:

- It should be correctly classified according to its class label.
- It should lie within or on top of either one of these two margin planes $(w^T x + b = +/ - 0ne)$

By satisfying both conditions simultaneously, we can identify whether a given data point belongs to any specific class and also acts as part of defining/supporting our decision boundary in SVM.

7.0.3.1 SVMs and Perceptron

SVMs and Perceptron are similar but:

- the Perceptron adjusts weights iteratively to find a decision boundary, while SVMs aim to maximize the margin between support vectors.
- So Perceptron focus on minimizing misclassifications, while SVMs prioritize maximizing margins.
- Perceptron can only learn linear boundaries, while SVMs can handle non-linear data using kernel functions.

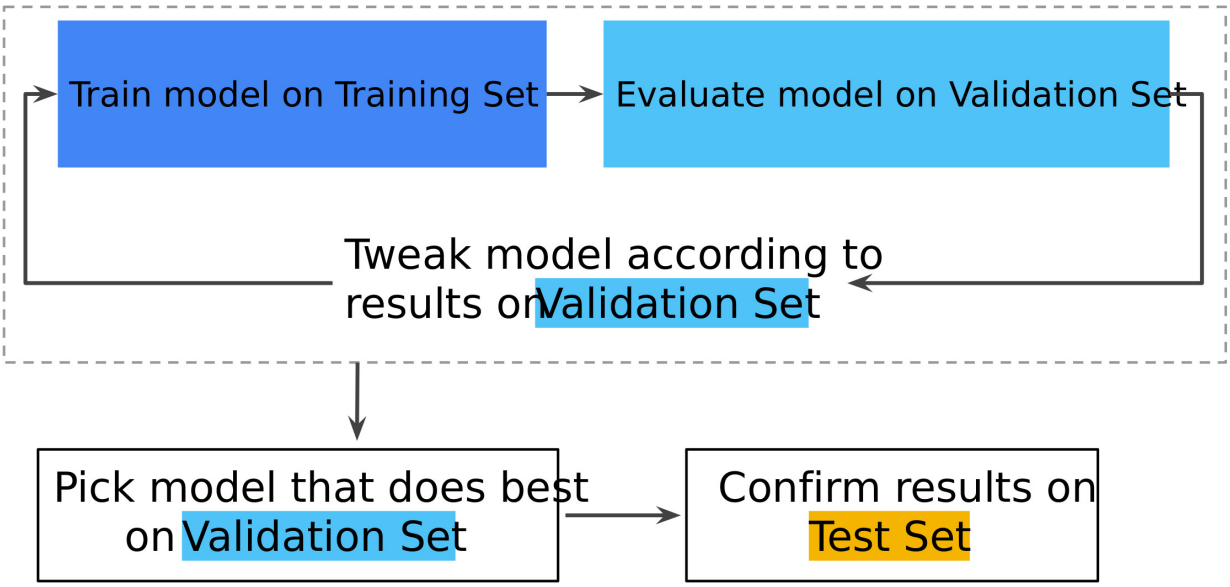
8 Model Evaluation

8.0.1 Training, Validation, Test

We split data in three parts:

- **Training Data**: the sample of data used to fit the model
- **Validation Data**: the sample of data used to provide an unbiased evaluation of a model fit on the training dataset, while tuning model hyperparameters
- **Test Data**: the samples of data used to provide an unbiased evaluation of the final model fit

We use training data to learn model parameters and for each model learned (i.e., different features and hyperparameters) we use validation data to compute the validation error.
We then select the model with the lowest validation error and finally use test data to estimate prediction error.



This separation is important to avoid creating statistical dependency. To further evaluate the model's performance, techniques to obtain a more reliable assessment of the model's generalization ability exist:

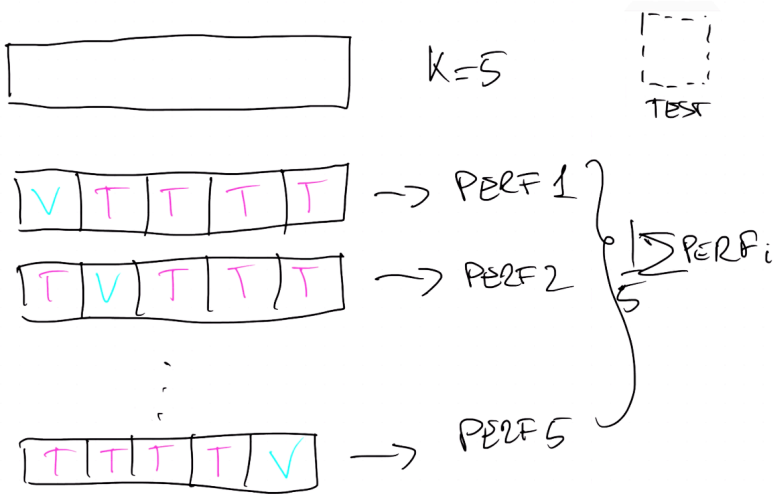
- **cross validation**
- **leave one out** (which is cross validation where `fold_size=n`)
- **adjustment techniques**

8.1 LOO Leave One Out

Leave One Out is the best you can do. It provides an almost unbiased estimate of prediction error (slightly pessimistic) and it's often used when dataset is small and the models are simples (the computational complexity doesn't explode).
Indeed LOO is extremely expensive to compute: you might have problems if you have many data because you have to perform a large number of training.

8.2 K-fold cross validation

K-fold is useful if you want to evaluate the performances of a set of different models in case of **large** dataset.



- The training data is randomly split into k folds.
- For each fold, the model is trained on the rest of the data
- The error is computed on the fold
- The prediction error is estimated by averaging the errors computed for each fold.
- The k-fold cross-validation provides a slightly biased estimate of prediction error but is computationally cheaper.
- Typically, k is around 10.

8.3 Adjustment techniques

Adjustment techniques should be a last resort. It is recommended to avoid using them whenever possible: in cases where you want to avoid reserving any data for validation, adjustment techniques can be employed. This is particularly useful when dealing with a small dataset and complex models that you do not want to train. There are several other adjustment techniques that can be employed.

$$C_p = \frac{1}{N}(RSS + 2d\tilde{\sigma})$$
$$AIC = -2\log L + 2d$$
$$BIC = \frac{1}{N}(RSS + \log(N)d\tilde{\sigma})$$
$$R^2 R^2_{ad} = 1 - \frac{RSS/(N-d-1)}{TSS/(N-1)}$$

9 Learning Theory

Training error and true error are two different types of errors:

- **Training error**: error rate which measures how well the model **fits the training data**
- **True error**: it refers to the average prediction error over all possible input values in an unseen dataset.

We cannot know the true error because we don't know the true distribution of data that the algorithm will work on. Instead, we can measure the model performance on a known set of training data (the "empirical" risk) and try to make an estimation of the true error.

9.1 No free lunch theorem

No Free Lunch theorems are a large class of theorems. Here we will analyze only the "main" NFL theorem only for the binary classification problem:

> "Independently from the learner, any technique in average will have an accuracy of 0.5, that in a binary classification problem is exactly the same accuracy of random guessing. Any learner cannot perform better than random guessing

Takeaways:

- NFL theorem states that an algorithm can perform poorly and not consistently outperform random guessing **on average**. However, this is based on the assumption that all possible hypotheses are equally likely to occur and there is no regularity in the world. Actually in real-world applications, some samples have lower probabilities of occurring, so specific algorithms tailored to specific tasks are needed and **possible**.
- There is no such thing as a winner-takes-all in ML! In ML we always operate under some assumptions! It is important to understand that there are no guarantees of success in all scenarios.
- While we can find algorithms that perform well on specific tasks, they may not generalize to other tasks.
- It is always possible to find data sets where an algorithm performs poorly.

9.2 VC Dimension

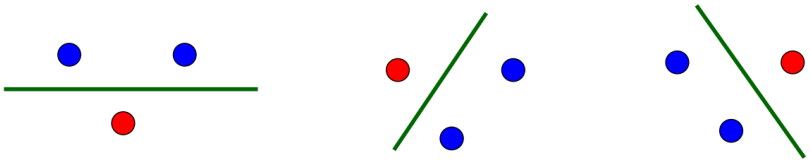
Some concepts:

- **Dichotomy**: given a set S of points a dichotomy is a partition of that set S in 2 disjunct subsets.
- **Shattering**: we say that a set S is shattered by an hypothesis space H if and only if for every dichotomy in S there exist some hypotheses in H consistent with this dichotomy. Basically H is shattering S if it is able to perfectly classify all the examples in S independently of how they are labelled.

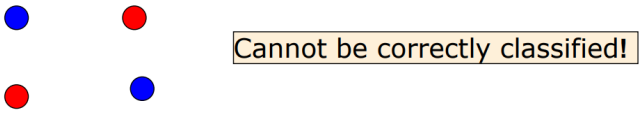
- $VC(H)$: The Vapnik-Chervonenkis dimension is the cardinality of the largest set of instances S shattered by H.

Besides the formalism the more complex is your hypothesis space the more you are likely that you're be able to shatter an higher cardinality set of instances, since "you can draw complex boundaries" to classify the points.

A linear classifier is a model that tries to draw a straight line (or plane, or hyperplane) to separate two groups of data points. In a 2D input space, this means we're trying to draw a line that separates one group of points from another. The VC dimension for this linear classifier in 2D is 3. This means that if we have up to three points in our dataset, we can always find some way to draw a line between them so that all the points on one side are in one group and all the points on the other side are in another group.



It turns out that the VC dimension for a linear classifier in $M - D$ input space is simply $M + 1$.



So if we have an M-dimensional dataset and use a linear classifier model ($y = w_0 + w_1x_1 + w_2x_2 + \cdots + w_m - 1x_m$) then according to VC theory, there will always exist some combination of weights such that any set of up to $M + 1$ data points can be perfectly classified into two groups using this model. Other examples of $VC(H)$ are:

- Neural Networks: $VC(H)$ is the number of parameters
- 1-Nearest Neighbors: $VC(H)$ is ∞

- SVM with Gaussian Kernel: $VC(H)$ is ∞

In a exercitation the prof showed that the $VC(H)$ of a particular hyphotesis space is at least a given number v you have to find at **least one** set of instances S of cardinality v (one possible location of the points) and show that it can be shattered by H, which means to test all possible dichotomies (aka all possible assignements of classes) it's possible to correctly classify the points. Vice versa to prove that $VC(H)$ is lower than v is slightly more difficult since we have to prove that for **all** set of instances S can't be shattered by H .

The VC dimension is at least k if the hypothesis space shatters at least one subset of cardinality k in the instance space.

9.3 PAC-Learning

Probably Approximately Correct is a statistical tool to assess the quality/performance of a learning model given information from the samples (either a test set (simplest scenario) or train set). Some definitions:

- A **concept class** is a set of possible concepts that our model can choose from when making predictions. It represents all possible functions or mappings that could explain the observed data.
- A class C of possible target concepts c is **PAC-learnable** by a learner L using H (the hypothesis space) if for all $c \in C$, for any distribution $P(X), \varepsilon$ (such that $0 < \varepsilon < 1/2$), and δ (such that $0 < \delta < 1/2$), L will with a probability at least $(1-\delta)$ output a hypothesis $h \in H$ such that error $_{\text{true}}(h) \leq \varepsilon$, in time that is polynomial in $1/\varepsilon, 1/\delta, M$, and size(c).
- A sufficient condition to prove **PAC-learnability** is proving that a learner L requires only a polynomial number of training examples and processing time for each example is polynomial. "If we have a concept class that is PAC-learnable it means that is not hard to learn".
- The **version space** is the subset of the hypothesis space H that contains all hypotheses with 0 training error. In practice usually we need to work outside of the version space.
- We will use the **Hoeffding Bound** which is is a tool to build confidence interval: $P(X \leq \bar{X} + \mu) = e^{-2Nu^2}$.
- The empirical risk minimizer $\hat{h} = \arg\min_{h \in \mathcal{H}} \hat{\mathcal{L}}(h) = \frac{1}{N} \sum_{n=1}^{N} \ell\left(h\left(\mathbf{x}_n\right), t_n\right)$ with ℓ the loss function.

We exploit the Hoeffding Bound to evaluate the true loss of the empirical risk minimizer starting from:

- the **test set**: it's independent from the training set so the test loss $\hat{L}(\hat{h})$ is an unbiased estimator for the true loss.
- the **train set**: obviously will provide a **negatively** biased estimator for the true loss.

9.3.1 Using the Test Set

Under the assumption of bounded loss $[0, L]$:

$$L(\hat{h}) \leq \hat{L}(\hat{h}) + L\sqrt{\frac{log(\frac{1}{\delta})}{2J}}$$

with probability $1-\delta$ and where:

- $\mathcal{L}(\hat{h})$ is the true error rate (risk) of our model, which measures how well it performs on new, unseen data points.
- $\tilde{\mathcal{L}}(\hat{h})$ is the empirical error rate (training loss) of our model, which measures how well it fits to a given set of training data.
- L can be thought as an upper bound on how much our predictions can deviate from their true values.

- J represents the size of our test set, i.e., how many samples we have available to evaluate our model's performance on new data points.
- δ represents confidence level or probability threshold we want to achieve with respect to this inequality.

9.3.2 Using the Training Set

We distinguish:

- **consistent learners** which have zero training error
- **agnostic learning** is when: if $c \in H$ and that the learner L will not always output a hypothesis h such that $\text{error}_{\mathcal{D}}(h) = 0$ but L will output a hypothesis h such that $\text{error}_{\mathcal{D}}(h) > 0$.

In case of **binary classification** (finite hypothesis space):

- Finite hypothesis space ($|\mathcal{H}| < +\infty$) and **consistent** learning ($\hat{\mathcal{L}}(\hat{h}) = 0$ always):

$$\mathcal{L}(\hat{h}) \leq \frac{\log|\mathcal{H}| + \log\left(\frac{1}{\delta}\right)}{N} \quad \text{w.p.} \quad 1-\delta$$

- Finite hypothesis space ($|\mathcal{H}| < +\infty$) and **agnostic** learning ($\hat{\mathcal{L}}(\hat{h}) > 0$ possibly):

$$\mathcal{L}(\hat{h}) \leq \hat{\mathcal{L}}(\hat{h}) + \sqrt{\frac{\log|\mathcal{H}| + \log\left(\frac{1}{\delta}\right)}{2N}} \quad \text{w.p.} \quad 1-\delta$$

When we moving to an **infinite** hypothesis space, like for example in a linear regression case (where the output is an real number) we have to use the notion of **VC** dimension, which in some away accounts for the complexity of the hypothesis space. So, in case of infinite hypothesis space ($|\mathcal{H}| = \infty$) and agnostic learning ($\hat{\mathcal{L}}(\hat{h}) > 0$ possibly):

$$\mathcal{L}(\hat{h}) \leq \hat{\mathcal{L}}(\hat{h}) + \sqrt{\frac{\text{VC}(\mathcal{H})\log\left(\frac{2eN}{\text{VC}(\mathcal{H})}\right) + \log\left(\frac{4}{\delta}\right)}{N}} \quad \text{w.p.} \quad 1-\delta$$

9.3.3 PAC takeaways

So basically, this theoretical formula confirms what so far we only said empirically by looking at the different techniques:

- Larger hypotheses space implies larger bound (variance)
- Increasing N implies reduced bound (variance)
- Large $|H|$: low bias, high variance
- Small $|H|$: high bias, low variance
- There is relationship between train error and prediction error (true error):
  - Close relationship between the two when under-fitting

- Relationship lost when over-fitting

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

10 Reinforcement Learning

Reinforcement Learning is a generalization of Machine Learning framework we seen so far which deals with a **sequential decision-making process**. ML problems like classification and regression lack of the **sequentiality** nature. In RL, each decision influences future decisions. The aim is to model process dynamics and choose from different actions in each situation. Two different problems:

- **Prediction**: given a specific behaviour (policy) in each situation, estimate the expected long-term **reward** starting from a specific state.
- **Control**: learn the optimal behaviour to follow in order to maximize the expected long-term **reward** provided by the underlying process.

RL is used in scenarios where the dynamics are unknown or it is difficult to directly model the problem. It is also used when the model is known but too complex to solve exactly or when there is a need to make sequential decisions.

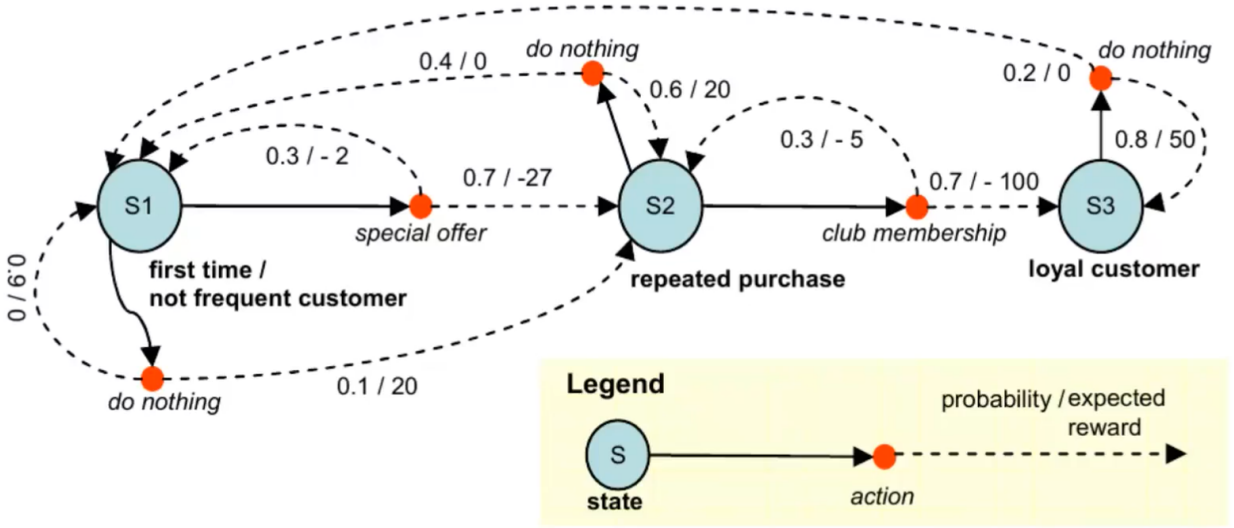
Resources:

https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver

10.1 Markov Decision Processes

MDPs are a mathematical framework used in reinforcement learning to describe an environment fully observable by the agent. MDPs provide a way to represent the agent's interaction and behavior. They operate under the Markov property:

> "The future is independent of the past given the present"



We model the MDP as a tuple of six elements $\mathcal{M} := (\mathcal{S}, \mathcal{A}, P, R, \mu, \gamma)$:

- **States**: S
- **Actions**: A note that not all actions are possible in all the states
- **Transition model**: $P : S \times A \to \Delta(S)$ tells how the environment evolves after each action. It's possible to rapresent it with a matrix.
- **Reward function**: $R : S \times A \to \mathbb{R}$
- **Initial distribution** $\mu \in \Delta(S)$, we need $\dim(\mu) = |S|$ numbers to store it
- **Discount factor**: $\gamma \in (0, 1]$

The agent's behavior is modeled by means of a policy $\pi : S \to \Delta(A)$. The policy can be both deterministic or probabilistic. Other interesting tools are:

- $P^{\pi}(s' \mid s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) P(s' \mid s, a)$ is the probability to reach state s' given we are in state s , so it's a function which maps state.
- $R^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) R(s, a)$ expected reward you will ever get from state s . Formula based on the expected value of the reward function.

10.1.1 Bellman Expectation Equation

The Bellman expectation equation is a mathematical tool which represents the expected value of the sum over an infinite horizon of the reward function, with a discount factor γ regulating future rewards' weight. It can be solved iteratively or recursively.

The **Bellman** expectation equation:

$$V^{\pi}(s) = \mathbb{E}^{\pi}\left[\sum_{t=0}^{+\infty} \gamma^t R(s_t, a_t) \mid s_0 = s\right] = \sum_{a \in \mathcal{A}} \pi(a \mid s) \left[R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V^{\pi}(s')\right]$$
$$= R^{\pi}(s) + \gamma \sum_{s' \in \mathcal{S}} P^{\pi}(s' \mid s) V^{\pi}(s')$$

We can invert the matrix (computationally costly) and say:

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

or we can simply approximate the V^π in an **iterative** and **recursive** way:

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

Both methods have their pros and cons, and the choice between them depends on the specific requirements of the problem.

In code, the first method uses a direct, closed-form solution:

```
V1 = np.linalg.inv(np.eye(nS) - gamma * pi @ P_sas) @ (pi @ R_sa)
```

Remember that: The @ symbol in Python is used for matrix multiplication.

Here, `np.linalg.inv` computes the inverse of the matrix $(I - \gamma P^\pi)$, where I is the identity matrix. This method can be computationally expensive, especially when the number of states or actions is large. Additionally, when $\gamma = 1$, the matrix to be inverted becomes singular, meaning it does not have a unique inverse, leading to potential computation errors.

The second method applies the Bellman expectation equation **iteratively** until the change in the value function between iterations is less than a predefined tolerance:

```
V_old = np.zeros(nS)
tol = 0.0001
V2 = pi @ R_sa
while np.any(np.abs(V_old - V2) > tol):
V_old = V2
V2 = pi @ (R_sa + gamma * P_sas @ V)
```

This iterative method is more computationally efficient and doesn't suffer from the singularity issue when $\gamma = 1$. However, it might take a long time to converge to the true value function if the tolerance is set too low.

10.1.2 Markov Reward Processes and Policies

A Markov reward process is a tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is a set of states, $\mathcal{P}$ is a transition probability matrix, $\mathcal{R}$ is a reward function, and γ is a discount factor. A policy π defines the behavior of an agent by determining actions based on the current state. Policies can be categorized as Markovian or history-dependent, deterministic or stochastic, and stationary or non-stationary.

- Markovian examples: chess, Rubik's cube
- Non-Markovian examples: poker, blackjack (requires information about past cards that have already been played)

A policy fully defines the behavior of the agent by determining which action to take at each step. Policies can be categorized as:

- **Markovian vs history dependent**: A policy is Markovian if it only depends on the current state. A policy is history-dependent if it also depends on the previous states.
- **Deterministic vs stochastic**: A policy is deterministic if, given a state, the action taken is always the same, creating a deterministic mapping between states and actions. A policy is stochastic if, given a state, there is a probability distribution over the actions.
- **Stationary vs non-stationary**: A policy is stationary if it does not depend on time, but **only on the state**. A policy is non-stationary if it depends on both time and states.

Nothing prevents a policy from considering real states and previous actions when determining which action to play. A history-dependent policy, as the name suggests, depends on everything that happened in the past. A Markovian policy, by contrast, depends only on the current state.

For Markov decision processes, Markovian policies are sufficient. There is no need to base decisions on the entire history. However, in **finite** horizon and Markov decision processes, non-stationary policies may be necessary. If the same state is encountered at the beginning and end of a certain number of steps, it is not guaranteed that the same action will be played in both cases. It may be more beneficial to reach certain states at the beginning, while maximizing immediate reward becomes important towards the end. This distinction does not exist in **infinite** horizon scenarios where there is no endpoint. In an **infinite** horizon MDP, Markovian stationary deterministic policies are sufficient.

10.1.3 Bellman operator T^π

The Bellman operator simplifies expectation equations, helping to find the best performance in a Markov Decision Process. Repeatedly using the Bellman operator leads to converging on a solution. The optimal value function shows the highest utility achievable in every state. Choosing the Bellman operator for the Bellman equation is recommended as it offers a computationally efficient way to approximate the value function in an MDP.

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

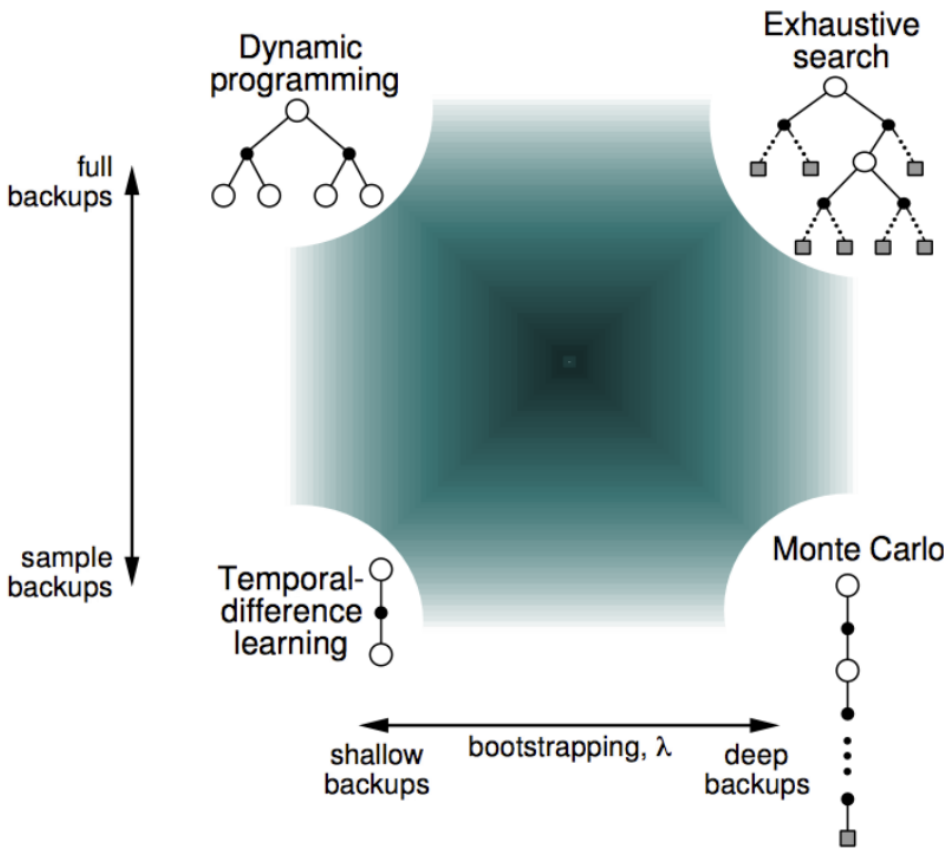
$$V^\pi(s) = E_\pi \left[v_t \mid s_t = s \right]$$

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

The **Bellman operator** T^π for V^π is defined as $T^\pi : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$. This operator takes as an argument a value function and it returns a value function:

$$T^\pi(V) = R^\pi + \gamma P^\pi V$$

11 RL Techniques



11.1 RL Techniques nomenclature

There are many distinctions in the RL techniques to approach the problem:

- **Model-free vs. Model-based**: RL doesn't necessarily require a predefined model of the environment. During the RL process, the collected samples can be used in two main ways:
  - **Model-based**: to estimate the environment's model.
  - **Model-free**: to estimate the value function without explicitly modeling the environment.
- **On-policy vs. Off-policy**: These terms refer to the relationship between the policy being learned and the policy used to collect samples:
  - **On-policy**: Learns the value function of the policy being enacted to collect samples. It involves playing the policy π and learning its Q-function Q^{π}.
  - **Off-policy**: Learns the value function of a policy that is different from the one used to collect samples. It involves playing an exploratory policy while learning the optimal Q-function Q^{*}.
- **Online vs. Offline**: This distinction is based on how the RL system interacts with the environment:

  - **Online**: The system continuously interacts with the environment, updating its policy and collecting new samples in real-time.
  - **Offline**: The system relies on a fixed set of previously collected interaction data, with no further interaction with the environment.

- **Tabular vs. Function Approximation**: This refers to the method of storing the value function:
  - **Tabular**: The value function is stored in a table, with discrete states and actions.
  - **Function Approximation**: The value function is represented using a mathematical function, which can be a simple linear approximator or a complex neural network. This approach is particularly useful for dealing with continuous states and actions.

11.1.1 Prediction & Control

The are 2 concepts which form the foundation for the behavior of agents within an environment:

- **Model-free prediction**: estimate value function of an unknown MRP (MDP + fixed policy π). We will see:
  - Monte Carlo
  - Temporal Difference
- **Model-free control**: optimize value function of an unknown MDP to learn optimal policy. We will see:
  - Monte Carlo Control: Monte Carlo estimation of $Q^\pi(s,a)$ combined with ε-greedy policy improvement
  - SARSA: Temporal Difference $\mathrm{TD}(0)$ estimation of $Q^\pi(s,a)$ combined with ε-greedy policy improvement
  - Q-learning: empirical version of Value Iteration Off-policy: play an **explorative** policy and learn the optimal Q-function $Q*$

11.2 Monte Carlo

Monte Carlo is a simple approach for estimating the value function or `Q(s,a)` (if we are predicting or controlling) directly from experience by taking the mean of the return of observed episodes. However, **it is not suitable for long episodes or infinite horizons.**

11.2.1 Monte Carlo for prediction

Monte Carlo for prediction tasks blueprint:

- wait until the end of the episode
- only episodic problems

- high variance, zero bias
- Good convergence properties
- not very sensitive to initial values
- adjust prediction toward the outcome
- general approach, less efficient

It comes in two flavors (**First-Visit vs. Every-Visit**) where the main difference lies in how they handle repeated visits to states within episodes:

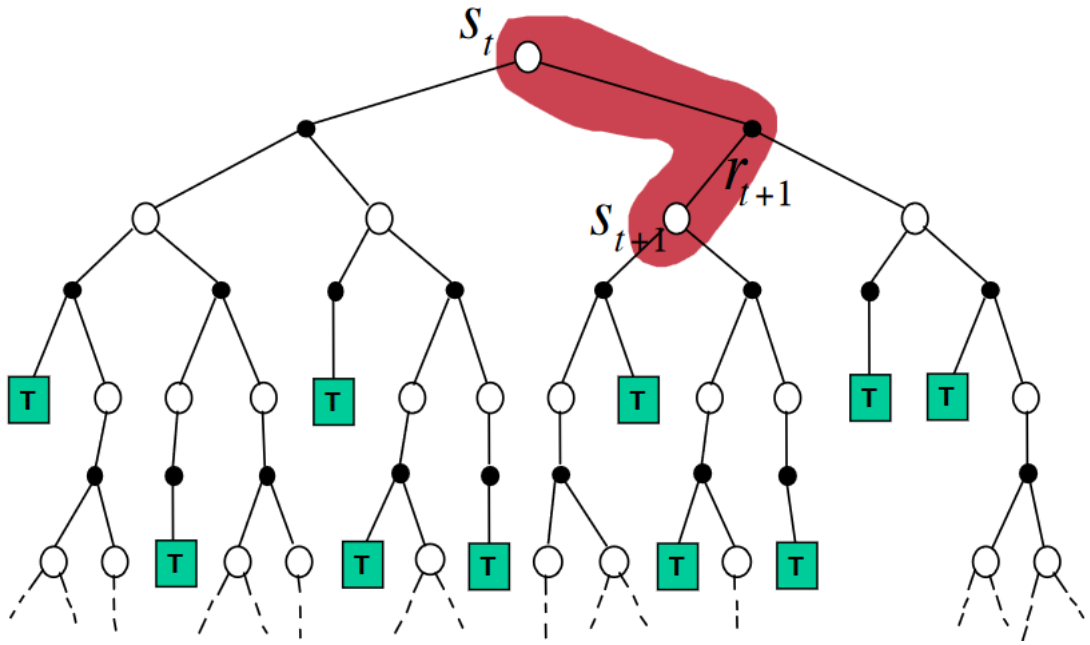
In first-visit Monte Carlo, only the first visit to a state in an episode is used for the value estimate, avoiding bias towards states visited more frequently. Every-visit Monte Carlo, however, includes all visits to a state within an episode, introducing a bias towards more frequently visited states.

11.2.2 Monte Carlo for control

Monte Carlo methods extend to control tasks, where the goal is to optimize the policy:

- **Generalized Policy Iteration**: The control approach follows a similar two-step process of policy evaluation and improvement, but adapted for the RL context where the model is unknown.
- **Policy Evaluation**: Monte Carlo is used to evaluate the `Q(s, a)` function for the current policy, providing a basis for policy improvement.
- **Policy Improvement**: A greedy improvement is made over `Q(s, a)` to enhance the policy. However, a purely greedy policy would lack exploration.
- ϵ**-Greedy Exploration**: The idea is "never give 0 probability to any action". To ensure exploration, the deterministic policy is modified to select a random action with probability ϵ, while choosing the greedy action with probability $1 - \epsilon$. The parameter ϵ regulates the amount of exploration. There is an equivalent policy improvement theorem also for ϵ-greedy policies, so we are sure that the resulting policy is always an improvement.

11.3 Temporal Difference



Temporal Difference (TD) learning method is a crucial technique in RL. At the heart of this method lies the update equation:

$$V(s_t) \leftarrow V(s_t) + \alpha \left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right)$$

- r_{t+1} represents the immediate reward received after transitioning from state s_t to state s_{t+1}.
- Finally, $(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))$ calculates what's known as TD error (Temporal Difference error). It measures the difference between our current estimate of the value of being in state s_t, i.e., $V(s_t)$, and our updated estimate using new information about rewards obtained from transitioning from state s_t to state s_{t+1}.

11.3.1 TD for prediction

RL version of the Bellman expectation equation. TD can *bootstrap* that is it can learn from incomplete episodes. Temporal difference uses its previous estimation to update its estimation (biased but consistent).

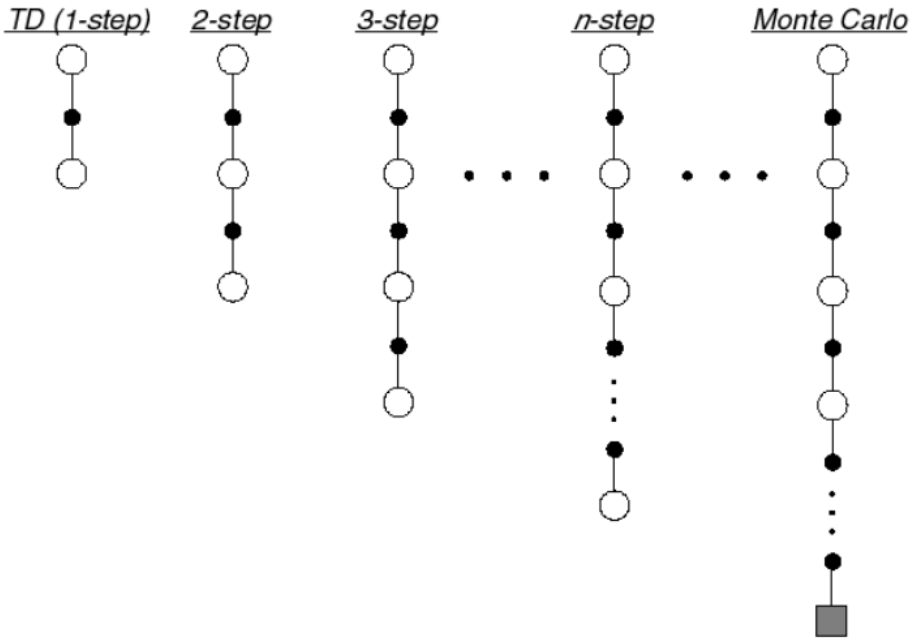
Blueprint of TD:

- Usually more efficient than MC
- learn online at every step
- can work in continuous problems
- low variance, some bias
- worse for function approximation
- more sensitive to initial values

- adjust prediction toward next state
- exploits the Markov properties of the problem

11.3.2 TD(λ)

TD(λ) represents an advanced Temporal Difference (TD) learning which harmonizes the concepts of TD that span from immediate TD updates to the episode-encompassing Monte Carlo ones.



Basically it's an intermediate approach between TD and MC.

$$V\left(s_t\right) \leftarrow V\left(s_t\right)+\alpha\left(v_t^\lambda-V\left(s_t\right)\right)$$

The parameter λ regulates how much we lean towards an approach or the other and the bias-variance trade-off. In MC we look at all the steps while TD ($TD(0)$) looks only at one step. $TD(\lambda)$ looks at some steps into the future before using the approximation.

- `n = 1` is the temporal difference approach ($TD(0)$)
- `n infinite` is the Monte Carlo approach

11.3.3 TD for control

D Control algorithms iteratively adjust the Q-values, which represent the expected returns of taking a particular action in a given state and following a specific policy thereafter. By optimizing these Q-values, the algorithms effectively refine the agent's policy towards the optimal strategy for maximizing rewards over time. The most notable TD Control algorithms include:

- **SARSA (State-Action-Reward-State-Action)**
- **Q-Learning**

11.3.3.1 SARSA algorithm This is an on-policy TD control algorithm where the agent learns the Q-value based on the action taken under the current policy. SARSA updates its Q-values using the equation:

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right)+\alpha\left(r_{t+1}+\gamma Q\left(s_{t+1}, \mathbf{a_{t+1}}\right)-Q\left(s_t, a_t\right)\right) \quad \mathbf{a_{t+1}} \sim \pi\left(\cdot \mid s_{t+1}\right)$$

SARSA λ extends the SARSA algorithm by incorporating the ideas from $TD(\lambda)$, effectively creating a bridge between SARSA(0) and Monte Carlo methods.

SARSA(λ) strikes a balance in information propagation between:

- **SARSA**: Updates only the latest state-action pair, limiting the scope to immediate transitions.
- **Monte Carlo (MC)**: Updates all visited pairs in an episode, considering the full sequence of actions.
- **SARSA(λ)**: Blends both approaches, updating recent and past pairs with diminishing impact via eligibility traces, ensuring a balance between immediate and comprehensive updates.

11.3.4 Q-learning

Q-learning seeks to learn the optimal policy even when the agent is not following it. The Q-value update rule for Q-learning is:

$$(Q(s, a) \leftarrow Q(s, a)+\alpha[r+\gamma \max_{a'} Q(s', a')-Q(s, a)])$$

where $(\max_{a'} Q(s', a'))$ represents the maximum Q-value for the next state s' across all possible actions a'.

Q-learning will learn the optimal policy even if it always plays the random policy. The only requirement is that we need to have a policy that have non zero probability to each action, but there is no constraint on the policy.

Why is this important?

- learn by observing someone else behavior
- reuse experience generated from old policies
- learn about multiple policies while following one policy

So the important difference between Q-learning and SARSA is that SARSA is an on-policy approach and can only learn the best ϵ-greedy policy considering also the exploration.

12 Multi-armed bandit

The multi-armed bandit problem is a sequential decision-making technique where there is a single state and multiple actions to choose from. The reward depends on the action taken, which can be deterministic, stochastic, or adversarial.

The reward distribution for each arm is initially unknown to the decision-maker, and that learning the reward distributions in order to maximize the reward over time is the key part of the problem (finding the best trade-off between exploration and exploitation).

This can be measured by minimizing the loss or expected pseudo-regret, which represents the number of times a suboptimal arm has been chosen.

- A multi-armed bandit is a tuple $\langle \mathcal{A}, \mathcal{R} \rangle$
- $\mathcal{A}$ is a known set of m actions (or "arms")
- $\mathcal{R}^a(r) = \mathbb{P}[r \mid a]$ is an unknown probability distribution over rewards
- At each step t the agent selects an action $a_t \in \mathcal{A}$
- The environment generates a reward $r_t \sim \mathcal{R}^{a_t}$
- The goal is to maximize cumulative reward $\sum_{\tau=1}^{t} r_\tau$
- The action-value is the mean reward for action a,

$$Q(a) = \mathbb{E}[r \mid a]$$

- The optimal value V^* is

$$V^* = Q\left(a^*\right) = \max_{a \in \mathcal{A}} Q(a)$$

- The regret is the opportunity loss for one step: the difference between the optimal value and the expected reward of the chosen action in a single step.

$$I_t = \mathbb{E}\left[V^* - Q\left(a_t\right)\right]$$

- The total regret is the total opportunity loss: the sum of instantaneous regrets over time.

$$L_t = \mathbb{E}\left[\sum_{\tau=1}^{t} V^* - Q\left(a_\tau\right)\right]$$

- Maximise cumulative reward $\equiv$ minimize total regret : while the goal is ultimately to maximize cumulative reward, most multi-armed bandit algorithms are framed in terms of minimizing regret.

12.1 Exploration vs exploitation

To play the optimal policy we need to reach a trade-off between:

- **exploration**: the choice of new unexplored actions, even at random, to increase our knowledge about the problem.

- **exploitation**: use only the current knowledge to make decision, following known-good paths but risking to miss some opportunities.

At the beginning we want to lean more towards exploration to see all the possible oppurtunities and then shift more and more towards exploiting the accumulated knowledge.

12.2 Upper Confidence Bound (UCB)

The Upper Confidence Bound (UCB) algorithm addresses the exploration-exploitation dilemma by embracing "optimism in the face of uncertainty" principle.

It selects actions, or "arms," based on their potential for being optimal, considering both their estimated rewards and the uncertainty surrounding those estimates.

This uncertainty is quantified using upper confidence bounds on the expected rewards, guiding the algorithm away from prematurely converging on suboptimal choices.

Key to UCB's approach is the application of the Hoeffding inequality, which helps calculate these bounds, with variations like UCB1, UCBV, and BayesUCB offering different strategies for computing them.

Consider a set of i.i.d. random variables $X_1, \dots, X_t$ within $[0, 1]$. The sample mean $\bar{X}_t = \frac{1}{t} \sum_{\tau=1}^{t} X_\tau$ approximates the expected value, and the Hoeffding inequality gives us a way to bound the probability of the true mean exceeding this estimate by a margin u:

$$\mathbb{P}\left[\mathbb{E}[X] > \bar{X}_t + u\right] \leq e^{-2tu^2}$$

In practice, the UCB1 algorithm selects the action a_t at time t as follows:

$$a_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left[Q(a) + \sqrt{\frac{2 \log t}{N_t(a)}} \right]$$

Where $Q(a)$ is the estimated reward for action a, and $N_t(a)$ is the number of times action a has been selected prior to time t.

The term $\sqrt{\frac{2 \log t}{N_t(a)}}$ serves as the "uncertainty" or "confidence interval" component, which decreases as the number of times $N_t(a)$ an arm a is played increases, thus reducing the exploration incentive for that arm over time.

This formula ensures a balance between exploiting actions with high estimated rewards and exploring actions with fewer trials.

The UCB approach is known for its logarithmic asymptotic total regret, indicating efficient performance over time.

So the recap is that this algorithm:

1) Iteratively selects the arm with the highest upper confidence bound (calculated as the sum of the estimated payoff and an uncertainty term)

2) observes the reward
3) updates the arm's estimated payoff, and recalculates the bounds.

This loop continues across a specified number of rounds, optimizing action selection based on both past rewards and the potential for discovery.

12.3 Thompson Sampling

Thompson Sampling, a Bayesian method, optimizes decision-making under uncertainty by balancing exploration and exploitation. This approach starts with a prior distribution for each option or "arm," typically assuming all outcomes are equally likely (uniform distribution). In each round, it samples from these priors, selects the arm with the highest value, and updates the priors based on the outcome (success or failure), adjusting the alpha or beta parameters of the distribution.

The core principle of Thompson Sampling is probability matching, guided by Bayes' law to compute posterior distributions $(P[\mathcal{R}|h_t)$ based on historical data (h_t).

Unlike methods that directly estimate reward distributions, Thompson Sampling maintains and updates a belief over these distributions, allowing for a natural integration of exploration and exploitation. By sampling from the posterior distributions, it inherently explores less certain options while exploiting known rewarding ones.

Thompson Sampling's efficiency stems from its ability to match the theoretical upper bound on regret with the lower bound, showcasing optimal performance. The algorithm favors actions with the largest sampled expected reward, $(\hat{r}(a_i))$, at each step.

In the context of Bernoulli trials, where outcomes are binary (success/failure), Thompson Sampling benefits from using a beta distribution as the conjugate prior.

This choice simplifies Bayesian updating, as the posterior remains in the same family (beta distribution), facilitating analytical tractability. The beta distribution, defined by parameters alpha α and beta β, adjusts its shape based on observed outcomes, starting from a uniform Beta(1, 1) prior for each arm.

The updating mechanism is straightforward:

- In case of success: $\phi(t+1) = Beta(\alpha_t + 1, \beta_t)$
- In case of failure: $\phi(t+1) = Beta(\alpha_t, \beta_t + 1)$

This iterative process refines our beliefs about each arm's success probability, guiding the selection towards the most rewarding options over time.