ML Recap

Useful knowledge:

https://machinelearningmastery.com/joint-marginal-and-conditional-probability-for-machine-learning/

Conditional probability: Probability of the occurence of event A given B. The "given" is indicated by the pipe | operator. P(A | B) = P(A, B)/P(B) (inverse formula of joint probability).

Marginal probability: Probability of event X = A given variable Y. P(X = A) = sum of P(X = A | Y = B) for all possible values for B.

Joint probability: Probability of the occurence of event A and B. The "and" can be indicated by the comma or by the operators ^ and ∩. P(A, B) = P(B, A) = P(A | B)P(B) = P(B | A)P(A).

	A=Red	A=Blue	P(B)
B=Red	(2/3)(2/3)=4/9	(1/3)(2/3)=2/9	4/9+2/9=2/3
B=Blue	(2/3)(1/3)=2/9	(1/3)(1/3)=1/9	2/9+1/9=1/3
P(A)	4/9+2/9=2/3	2/9+1/9=1/3	

Statistical Learning

Machine Learning deals with systems that are trained from data rather than being explicitly programmed.

The goal of **supervised learning** is to learn an underlying relationship from the given data. The given data is called **training set** (or **examples**) and is a set of n input-output pairs $S_n = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. And the underlying relationship is a function $f : X \to Y$ so that $f(x_i) \simeq y_i$ per i = 0...n. The important thing to notice is that this implies that if f is applied to a value x_{new}

that wasn't present on the training set, the result will still be similar or equal to the corresponding value of the pair y_{new}.

X, the **input space** is generally a subset of $\mathbb{R}^d$, where d is the number of dimensions or features, while Y, the **output space**, varies according to the problem:

- Scalar regression: $Y \subseteq \mathbb{R}$
- Binary classification: $Y = \{-1, 1\}$
- Multi-category classification: $Y = \{0, 1, 2, \dots, T\}$

$X \times Y$ is called the **data space**.

Our model should take into consideration possible **uncertainty** both in the task and in the data space. Focusing on the last one, we need to make two **assumptions**:

- The samples have been **identically and independently (iid)** generated according to a **fixed and unknown** joint probability distribution p(x, y). The same applies to the **test set**, a set similar to the training set in which we are going to apply our model after training it.
- p(x, y) can be factorized as $p(x, y) = p_X(x)p(y|x)$, where $p(y|x)$ models uncertainty (**noise**) on the output (it represents a distribution that models the probability of a certain y being the output for each x) while the marginal probability $p_X(x)$ (aka, p(x = X)) models uncertaity on the input (the probability of getting that input considering all possibles outputs).

Back to our function f, we use the **loss function** $l : Y \times Y :\rightarrow [0, +\infty)$ to evaluate how much we lose if instead of using the real y_i we use the prediction $f(x_i)$.

We could use the **square loss**, for example: $l(y_i, f(x_i)) = (y_i - f(x_i))^2$. We raise the result of the subtraction to the power of 2 for two reasons: the main reason is that we don't want negative values and the second is that it is better using squares than taking the absolute value because of derivatives. In this case, derivating is useful because by setting the derivative to zero it is possible to find critical points and minimize the loss function (see RLS).

The **expected loss** (or **expected risk**) can be seen as a measure of the error on past and future error:

$$\mathcal{E}(f) = \mathbb{E}[l(y, f(x))] = \int p(x, y) l(y, f(x)) dx dy$$

We want to find the **target function** f^* that **minimizes** the expected loss:

$$f^* = \arg\min_{f \in F} \mathcal{E}(f)$$

Since **we cannot directly compute it by using the integral because p(x, y) is unknown**, the goal of learning algorithms is to find an **estimator** f_n based on training data of n samples S_n that mimics (**generalizes**) the target function, $\mathcal{E}(f_n) \simeq \mathcal{E}(f^*)$.

Now it is possible to give the definition of **empirical loss** (also known as **empirical risk** or **empirical error**), that is basically the discrete version of the expected loss **without the probability distribution** (so it is computable):

$$\mathcal{E}_n(f) = \frac{1}{n} \sum_{i=1}^{n} l(y_i, f(x_i))$$

A basic requirement for the estimator is **consistency**, that basically says that as n gets larger, f_n gets more similar to f^* (their difference approaches zero):

$$\lim_{n \to +\infty} \mathbb{E}[|\mathcal{E}(f^*) - \mathcal{E}(f_n)|] = 0$$

There are also two key concepts in **designing** a good algorithm:

- **Fitting:** an estimator should fit (training) data well.
- **Stability:** an estimator should not change much if the (training) data changes sligthly.

We say that an algorithm **overfits** if it “falls in love with the data” and becomes unstable and **underfits** (oversmooths) when it disregards the data and becomes too stable, basically not changing at all even if the data changes much.

In order to find a trade-off between data-fitting and stability, learning algorithms use one or more **regularization parameters**.

Local methods

Local methods are the class of learning algorithms based on the principle that **nearby points should have similar or the same output**.

- **Nearest Neighbors:** The idea is assign to the new point the output value of the nearest point present in the training data (neighbor).

Given a training set S with n pairs input-output and a new input point x^*, $y^* = \hat{f}(x^*)$ is the predicted output where $y^* = y_j$ and:

$$j = \arg\min_{i=1,\dots,n} ||x^* - x_i||$$

```
def NN(x_new, Xtr, Ytr):
    """
    xnew: new point
    Xtr: input in the training set
    Ytr: output in the training set
    return: y_new, estimated output for x_new
    """
    dist = zeros(1, n) # array n-dim
    for i = 0 ... n-1: # O(n)
        dist(i) = euclidean_dist(x_new, Xtr(i, :)) # Euclidea
    index_min = argmin(dist)
    y_new = Ytr(index_min)
    return y_new
```

The **complexity** is $O(nD)$, because we need to compare x^* with the n points present in the trainining set and each point is of dimension D.

- **K-Nearest Neighbors (K-NN):** The idea is assign to the new point the mean of output values of the K nearest points present in the training data.

Given a training set S with n pairs input-output , $K << n$ and a new input point x^*, $y^* = \hat{f}(x^*)$ is the predicted output where:

$$y^* = \frac{1}{K} \sum_{i \in \{j_1,\dots,j_K\}} y_i$$

And

$$j_1 = \underset{i \in \{1,\dots,n\}}{\arg\min} ||x^* - x_i||$$

$$j_t = \underset{i \in \{1,\dots,n\} \setminus \{j_1,\dots,j_{t-1}\}}{\arg\min} ||x^* - x_i||$$

for $t = 2, \dots, K$

The **complexity** is $O(nD + n\log(n))$. As before, we compare x^* with each of the n points present in the training data and then sort.

```
def KNN(x_new, Xtr, Ytr, K):
    """
    x_new: new point
    Xtr: input in the training set
    Ytr: output in the training set
    K: number of neighbours to consider
    return: y_new, estimated output for x_new
    """
    dist = zeros(1, n) # array n-dim
    for i = 0 ... n-1: # O(n)
        dist(i) = euclidean_dist(x_new - Xtr(i, :)) # Euclide
    [dist_sort, index_sort] = argsort(dist, ascend) # O(nlog(
    # Two situations:
    # Linear regression (average of the values of the neighbo
    y_new = average(Yn(index_sort(0, ..., K-1)))
    # Binary classification (If the result of the sum is posi
    y_new = sign(sum(Yn(index_sort(0, ..., K-1))))
    return y_new
```

The **Euclidean distance** $||x^* - x_i||$ can also be indicated by $d(x^*, x_i)$

When the **hyperparameter** K is small, there is overfitting, since the model takes too much into consideration the current data points (including outliers) and ignores a more significant pattern that can be detected with a higher K. For this reason, it can also be very sensitive to noise. However, pay attention to not make K too high, otherwise there is the opposite problem, underfitting. The model takes into account basically every point, making the model useless.

K-NN puts equal weights on the values of the K selected points. In order to generalize it, we can use **parzen windows**, that provides a way to give more

importance to points that are closer to the new point.

N.B.: We've analyzed KNN for just a single new point. If we have an entire data matrix X^* with m new points to classify, the algorithm changes accordingly (a similar modification can be done to NN). The new complexity is $O(m(nD + n\log(n)))$, because now we have to repeat the same procedure m times.

```
def KNN(Xnew, Xtr, Ytr, K):
    """
    Xnew: input in the test set (in total, m points)
    Xtr: input in the training set (in total, n points)
    Ytr: output in the training set (in total, n points)
    K: number of neighbours to consider
    return: Ynew, estimated output for x_new (in total, m poi
    """
    dist = zeros(m, n) # matrix m x n
    for i = 0 ... m-1: # O(m)
        for j = 0 ... n-1: # O(n)
        dist(i, j) = abs(Xnew(i, :) - Xtr(j, :)) # Euclidean
    [dist_sort, index_sort] = argsort(dist, ascend) # O(nlog(
    # Two situations:
    # Linear regression (average of the values of the neighbo
    Ynew = average(Ytr(index_sort(0, ..., K-1)))
    # Binary classification (If the result of the sum is posi
    Ynew = sign(sum(Ytr(index_sort(0, ..., K-1))))
    return Ynew
```

Cross-validation

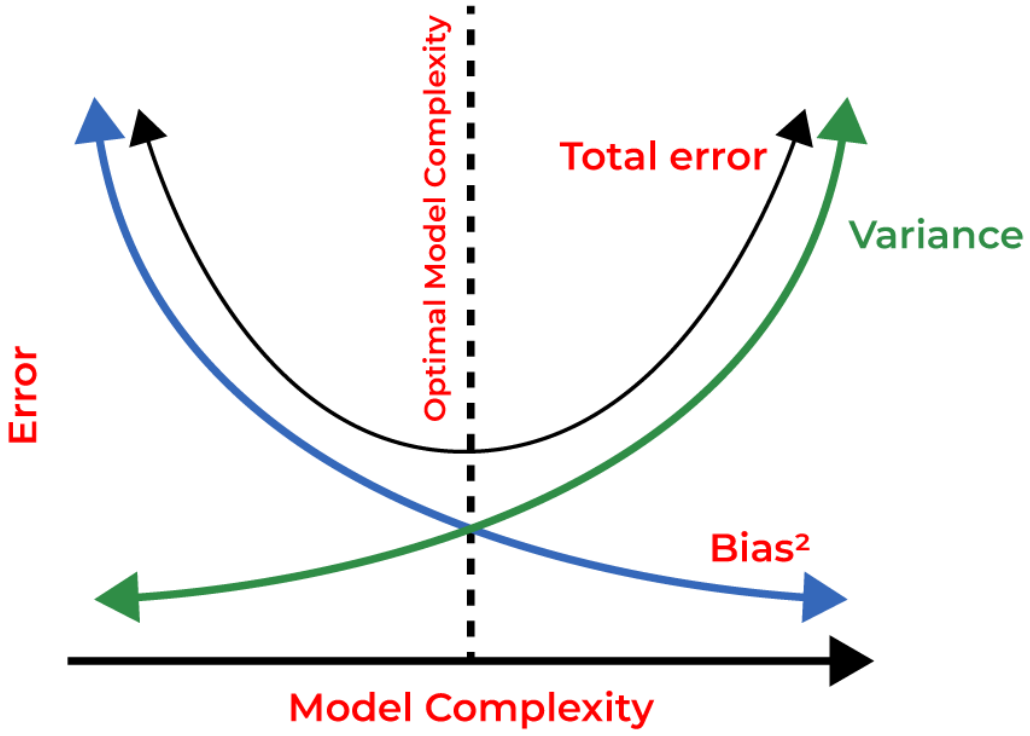
We want to find the optimal hyperparameter K^* that minimizes the expected loss $\mathcal{E}_K$:

$$K^* = \arg\min_{K \in \mathbb{K}} \mathcal{E}_K$$

Considering that $\mathcal{E}_K$ is the sum of the **bias** and the **variance**. We want a model with **low bias**, which means that it is able to fit the training data well, and **low variance**, which means that it is consistent in terms of accuracy of the predictions because it doesn't change much according to the training data. The

problem is that, as K increases, the variance increases as well while the bias decreases, as seen before.

The **bias-variance trade-off** is theoritical (therefore it cannot be computed), but it shows that there is a point K^* (in the x-axis) in which both the bias and the variance meet and minimizes the total error.



A way to find K^* is by using **(hold-out) cross validation**. The idea is to train on some data and then evalute the model's performance on new unseen data.

For each K:

1. shuffle and split S in in T (training) and V (validation)
2. train the algorithm on T and compute the empiracal loss $\mathcal{E}_K$ on V

At the end, select K that minimizes $\mathcal{E}_K$ as K^*.

It is possible to repeat the above procedure to augment stability to K^*

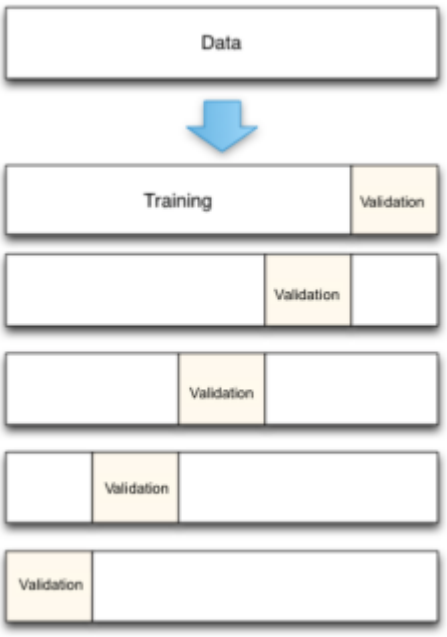
There are other related hyperparameter selection methods, like **k-fold cross validation**, which allows to vary works as follow:

For each K

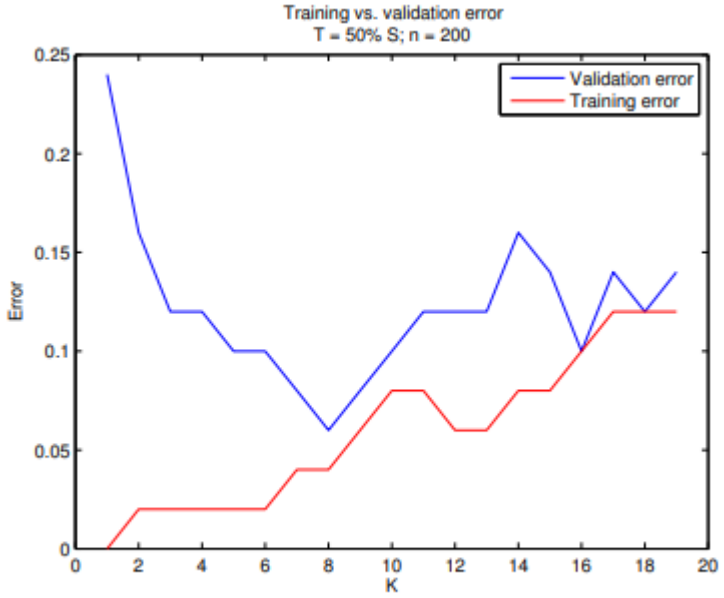
1. Shuffle S randomly.

2. Split S into k groups
3. For each unique group:
    a. Take the group as the validation data set V
    b. Take the remaining groups as the training data set T
    c. Fit a model on the training set and evaluate it on the test set
    d. Retain the evaluation score and discard the model
4. Summarize the skill of the model using the sample of model evaluation scores

At the end, selects as K^* the K that performs better overall.



In the example below, $K^* = 8$ because its the point where the **validation error is smaller** (this means that the model is able to generalize better using this hyperparameter).



$\hat{K} = 8.$

The **pseudocode** for v-fold cross-validation (considering KNN, with hyperparameter k):

```
def VfoldCV_KNN(Xtr, Ytr, V, k_list):
    """
    Xtr: input in the training set
    Ytr: output in the training set
    v: number of folds
    k_list: list of number of neighbour to select the best fr
    return: best_k, the k from k_list that minimizes the mean
    """
    Xs = list(V)
    Ys = list(V)
    E = matrix(V, len(k_list)) # mean error matrix for each v

    [Xs, Ys] = partition(Xtr, Ytr, V) # split the training se
    for i = 1 ... V:
        # choose one partition as validation set
        Vx = Xs(i)
        Vy = Ys(i)

        # and all other partitions together as training set
        Tx = union(Xs - Vx)
```

```
        Ty = union(Ys - Vy)

        for k in k_list:
            Ynew = KNN(Vx, Tx, Ty, k)
            # err calculates the error for each output
            # mean returns a value for the mean error
            E(i, k) = mean(err(Y_new, Vy)) # calculate the me

        # col_mean calculates the mean column-wise for matrix
        # argmin returns the k with smallest mean error assoc
        best_k = argmin(col_mean(E))
        return best_k
```

To change for other methods, just change the KNN function with another function (e.g. RLS), naming the hyperparameters accordingly (e.g., lambda_list and lambda for RLS).

Global vs local methods

When using local methods like KNN, the output y_i is affected only by the K closest points. This **approach based on locality might not work well when points are sparse**, a scenario that is not uncommon in **higher dimensions**. This phenomenon is called **curse of dimensionality**.

For example, suppose the data is spread over a D-dimensional unit cube. If we grow a hyper cube (cube inside the unit cube) until it contains the desired volume of the data points (the volume can be written as a fraction, such as 1% = 0.01), the expected edge length of this cube will be:

$$\frac{e^d}{v} = 1 \rightarrow e = v^{\frac{1}{d}}$$

The first equation is right because the volume of the hypercube is equal to the desidered volume of data points and the second one is right because we pass v to the other side and then take both sides to the power of $\frac{1}{d}$ to find e.

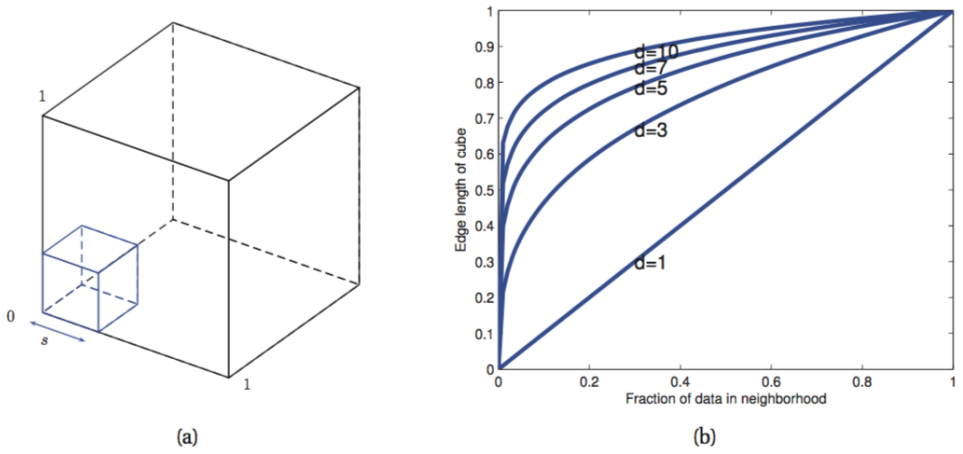


Figure 1.16 Illustration of the curse of dimensionality. (a) We embed a small cube of side s inside a larger unit cube. (b) We plot the edge length of a cube needed to cover a given volume of the unit cube as a function of the number of dimensions. Based on Figure 2.6 from (Hastie et al. 2009). Figure generated by `curseDimensionality`.

Empirial risk minimization (ERM)

Empirical risk minimization (ERM) is one of the most popular approaches to design learning algorithms and its based on the idea of considering the empirical error $\hat{\mathcal{E}}(f)$ (previously written as $\mathcal{E}_n$) as a proxy for the expected error $\mathcal{E}(f)$, that as we've seen cannot be calculated directly since the original probability distribution is unknown.

To turn this idea into an algorithm, we need to fix a suitable hypothesis space H and minimize $\hat{\mathcal{E}}$ over H.

H should allow feasible computations and be rich, since we don't know the complexity of the problem a priori. The simplest example is the space of linear functions:

$$H = \{f : \mathbb{R} \to \mathbb{R}^d : \exists w \in \mathbb{R}^d \text{ such that } f(x) = x^T w, \forall x \in \mathbb{R}^d\}$$

Basically, the above set is the set of all functions that receive a vector x as an input and can be written as a multiplication between x and a vector w. Each function therefore is defined by a vector w:

$$f_w(x) = w^T x = \sum_{i=0}^{d} x_i w_i$$

The problem is, if H is rich enough, solving ERM may cause overfitting. In fact, we may end up finding a function that minimizes the empirical risk because it is highly dependant on the training data. For solving this problem, we use **regualrization techniques.**

One example is the **Tikhonov regularization scheme**, that adds a regularization term to the functional (function of a function to improve the stability of the model:

$$\min_{w \in \mathbb{R}^d} \hat{\mathcal{E}}(w) + \lambda ||w||^2$$

where $||w||^2$ is the **regularizer** (or **penalty**), which controls stability and generalization of the solution preventing overfitting, and λ is the **regularization parameter**, which balances the error term and the regularizer, determining how severe is the penalty. When $\lambda \to 0$, we are not penalizing anything.

There exists no general computational scheme to solve Tikhonov Regularization since the solution depends on the considered loss function. In fact, **different loss functions induce different classes of methods**.

Regularized Least Squares

Considering the Tikhonov regularization scheme using the square loss as loss function (which is called **ridge regression**), we obtain the RLS optimization problem (linear model):

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} (y_i - w^T x_i)^2 + \lambda w^T w$$

With the matrix notation, where the matrix X_n with dimension $n \times d$ represent the n input points with d features and the vector Y_n with dimension $n \times 1$ represents the n corresponding output values, we can rewrite part of the formula as:

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} (y_i - w^T x_i)^2 = \frac{1}{n} ||Y_n - X_n w||^2$$

After computing the gradient of the empirical risk, we set it to zero in order to find the w that minimizes it:

$$\frac{2}{n}X_n^T(Y_n - X_n w) = 0$$

The solution of RLS solves the linear system below (to get the following equation from the above equation, apply the distributive property and consider that $\frac{2}{n}$ can't be zero):

$$(X_n^T X_n)w = X_n^T Y_n$$

For simplicity, let's call $A = (X^T X_n)$ and $b = X_n^T Y_n$, remembering that A is a $d \times d$ matrix (X_n^T is $d \times n$ and X_n is $n \times d$).

In order to find w, let's rewrite the system using the inverse of A:

$$w = A^{-1}b$$

There are two ways to solve this system:

- By using direct methods. For example, we can use **Cholesky decomposition** to calculate the inverse, since A is symmetric and positive definite.
- By using iterative methods, like conjugate gradient.

But what if A is not invertible? If A is a diagonal matrix, it can be written as:

$$A = \begin{bmatrix} \sigma_1 & & \\ & \dots & \\ & & \sigma_d \end{bmatrix}$$

And its inverse can be calculated as:

$$A^{-1} = \begin{bmatrix} \frac{1}{\sigma_1} & & \\ & \dots & \\ & & \frac{1}{\sigma_d} \end{bmatrix}$$

The same applies to A if it is not diagonal: in fact, since it is symmetric and positive definite, we can write A as its singular value decomposition $V\Sigma V^T$. The matrix Σ is diagonal in this case and we can apply the same reasoning above (pseudoinverse).

The problem arises when one of the sigmas is 0, because the term $\frac{1}{\sigma_i} \to \infty$. We can solve that by considering the Tikhonov regularization. In fact, the

regularization parameter λ can be added to the sigmas in the diagonal of the inverse matrix, in order to make it invertible.

$$A^{-1} = \begin{bmatrix} \frac{1}{\sigma_1+\lambda} & & \\ & \dots & \\ & & \frac{1}{\sigma_d+\lambda} \end{bmatrix}$$

- If $\lambda = 0$, the values in the diagonal will be $\frac{1}{\sigma_i}$
- If $\lambda > 0$, the values in the diagonal will be $\frac{1}{\sigma_i+\lambda} \approx \frac{1}{\lambda}$. In this case, if λ is too big, the values will be close to 0.

Since the λ also controls the trade-off between fitting and stability (bias vs variance), increasing stability if larger or increasing data fitting if smaller, we use cross-validation (usually 10-fold) to choose the optimal value for this variable.

The algorithm for solving RLS distinguishes two cases:

- **Underparameterized (or overdetermined) setting:** The setting considered until this point, where $n > d$ and w is calculated in the following way:

$$w = (X_n^T X_n + n\lambda I)^{-1} X_n^T Y_n$$

N.B.: The term $n\lambda I$ is the regularization term. The n is optional, it just helps in the case λ is too small in comparison to A.

- **Overparameterized (or underdetermined) setting:** The number of parameters is grater than the number of data points ($d > n$). In this case, the rank of A is at most n (since the rank is equal to min(n, d)), and this means that some of the singular values σ_i are equal to zero, so the A is not invertible and the system has infinite solutions. The idea is to use **inductive bias**, that is, choosing the simplest solution ($\min_{w \in \mathbb{R}^d} ||w||^2$). For that, we first calculate c and then find w:

$$c = (X_n X_n^T + n\lambda I)^{-1} Y_n$$

$$w = X_n^T c$$

The compleity in the first case is $O(d^2 n + d^3)$ and in the second case $O(n^2 d + n^3)$.

For linear models (especially in lower dimensions), it is useful to consider an offset b, $w^T x + b$. In order to simplify the problem, we center the data using the mean of the sample points to make $b = 0$.

Logistic Regression

Logistic regression is used to **classify data**. In this case, minimizing the expected loss is defined as:

$$\min_f L(f) = \int l(y, f(x))p(x, y)dx = \iint [l(y, f(x))p(y|x)dy]p(x)dx$$

The step with the double integral is achieved because the joint probability is defined as: $p(x, y) = p(y|x)p(x)$. The inside integral is called **inner risk** or **inner loss** and it is indicated with $L_x(a)$, where $a = f(x)$.

We'll first try to minimize it by using both the **square loss** and the **logistic loss** as a loss function, considering **binary classification**:

$$L_x(a) = \int l(y, f(x))p(y|x)dy = l(1, a)p(1|x) + l(-1, a)p(-1|x)$$

As usual, we'll take the derivative to minimize it:

$$\frac{dL_x}{da} = 0$$

By using the square loss, we find the following solution for a:

$$\begin{gathered} -a + p(1|x) - p(-1|x) = 0 \\ a = p(1|x) - p(-1|x) \end{gathered}$$

In this case, if $p(1|x) > p(-1|x)$, a will be positive (> 0) and the prediction will be class 1. Otherwise, it'll be negative (< 0) and the prediction will be class -1. If it's 0, it means that the model predicts the two classes with equal probability, so it doesn't perform well. Conversely, if $|a| = 1$, it means that the model is 100% sure about a certain class.

Now let's consider as loss function the **logistic loss**:

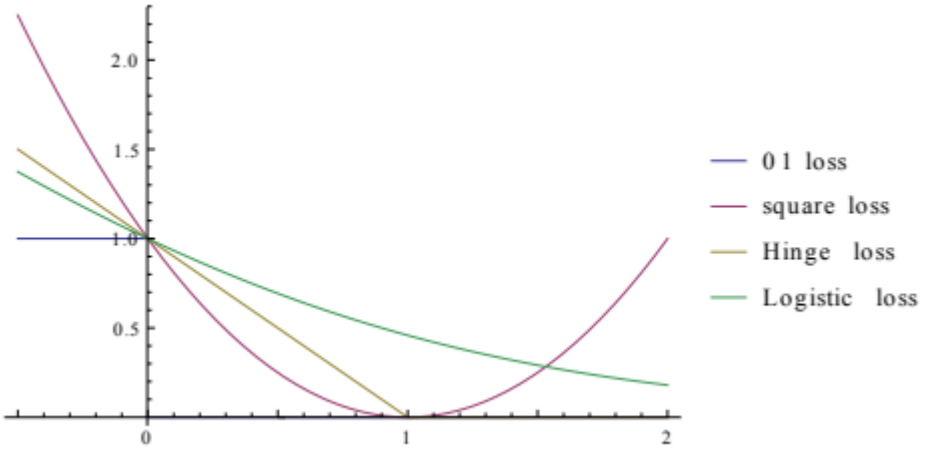
$$l(y, f_w(x)) = \log(1 + e^{-yf_w(x)})$$

In this case, the solution for a is:

$$a = \log\left(\frac{p(1|x)}{p(-1|x)}\right)$$

If the two probabilities are the same, the argument of the logarithm will be 1, so $a = 0$. If $p(1|x) > p(-1|x)$, the argument will be greater than 1, so $a > 0$ and the prediction will be class 1. Otherwise, the argument will be smaller than 1, so $a < 0$ and again prediction will be class -1.

In synthesis, even thought the square loss and logistic loss are different formulas, they behave the same in this case. The main difference between the two is that it is possible to find the minimimum of the square loss, what can't be done with the logistic loss, as can observed in the graphs below (above we've considering the expected loss, not the loss function):



It's not a problem though when using Tikhonov regularization where the loss function is the logistic loss, because the term $\lambda||w||^2$ is actually a parabola (in the same way as $||y - f(x)||^2$, think about it as applying the function that w represents to the vector itself and then multipying by λ instead of using y).

In this case, we can calculate the minimum using **gradient descent (GD)**.

The idea is to first calculate the gradient of $\hat{L}(w)$ for a certain w_0:

$$\nabla \hat{L}_\lambda(w) = \nabla G(w)$$

$$\hat{L}_\lambda(w) = \log(1 + e^{-y f_w(x)}) + \lambda||w||^2$$

And then finding w_t using the following formula:

$$w_t = w_{t-1} - \gamma \nabla G(w_{t-1})$$

Where the gradient $\nabla g(w_{t-1})$ indicates the **direction** and γ is the **step** that regulates how fast we want to find the minimum (if the function is “curvy”, we want to be careful, making small steps to not get past the minium, otherwise if it is “flat” we want to go faster).

By using **stochastic gradient descent (SGD)**, instead of computing the gradient for all the points once, which is computationally expensive, you just use a single point to calculate part of the gradient and use it as if it were the gradient.

$$w_t = w_t - \gamma \nabla g_i(w_t)$$

Thanks to the linearity of the gradient, we know that it can be written as the sum of the gradient calculate on each individual data point:

$$\nabla G(w) = \sum_{i=0}^{n} \nabla g_i(w)$$

N.B.: This sum is just a property when w is fixed, but in the SGD algorithm, the $\nabla g_i(w_t)$ varies accordingly to w_t.

In the mini-batch version, instead of using all the data points or just one, we use a certain number of points (e.g., 4) and calculate the gradient using it.

Another alternative is to use **Newton’s method**: $w_t = w_t - H_f(w_t)^{-1} \nabla f(w^t)$

Support Vector Machines (SVMs)

Now let’s consider as **loss function** the **hinge loss**, always keeping in mind Tikhonov regularization:

$$l(y, f_w(x)) = \max(0, 1 - y_i f_w(x_i))$$

SVMs aim to find a hyperplane that best separates different classes in the feature space. For a binary classification problem, this hyperplane ideally maximizes the margin between the classes. In a 2D space, a hyperplane is a line; in 3D, it's a plane, and so on.

The margin is the distance between the hyperplane and the nearest data point from either class (support vectors). SVM seeks to maximize this margin:

$$\max \beta$$
$$\beta = (w^t x_i + b) y_i$$

Kernels and features spaces

To deal with high dimensional non linear problems, we use a function called **feature map** $\Phi : \mathbb{R}^d \to \mathbb{R}^D$, where in general $D >> d$. It is defined as:

$$\phi(x) = (\varphi_1(x), ..., \varphi_D(x))$$

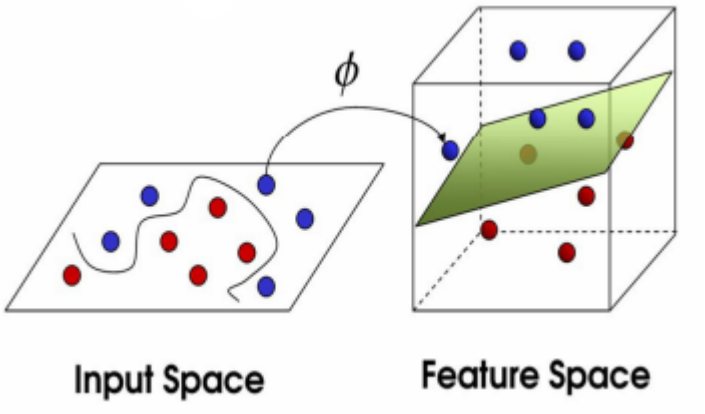
where $\varphi_j(x) : \mathbb{R}^d \to \mathbb{R}$ with $j \in 1, \ldots, D$.

In this way, we can write the non-linear model as:

$$f_w(x) = x^T \phi(x)$$

where $w \in \mathbb{R}^D$.

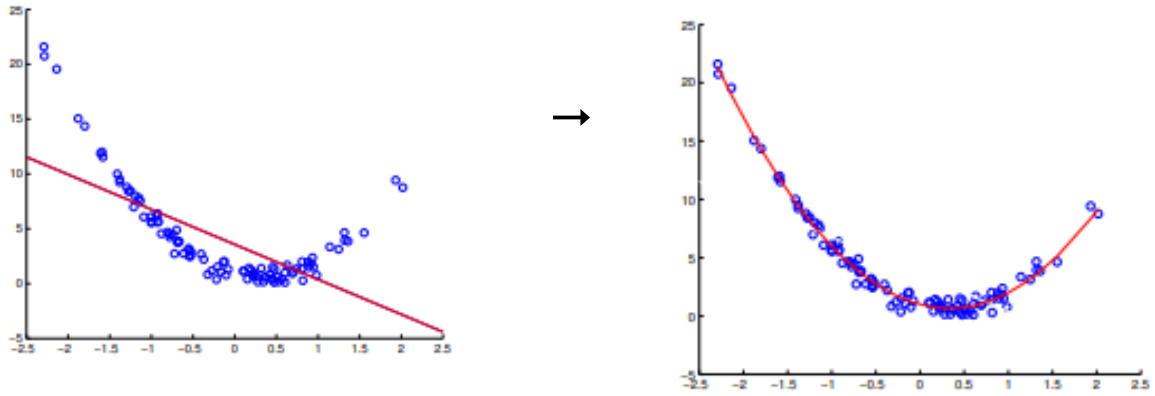
In more pratical terms, the feature map $\phi : X \to F$ therefore **maps** the data in the space X into another space F in which it is **linear**. When using feature maps, points which are not easily classified by a linear model, can be easily classified by a linear model in the feature space.



For example, let's consider a data that follows a quadratic trend and compare the performance of the linear model and the one using the feature map:

$$f_w(x) = w^T x \to f_w(x) = w^t \phi(x) = w_1 x^2 + w_2 x + w_3$$

$$\phi(x) = (x^2, x, 1)$$
$$w = (w_1, w_2, w_3)$$



For RLS, we simply need to replace the $n \times d$ matrix X_n with a new $n \times D$ matrix Φ_n, where each row is the image of an input point in the feature space, as defined by the feature map.

Let $\Phi = (\phi(x_1), \ldots, \phi(x_n))^T \in \mathbb{R}^{nxD}$ be the **data matrix** in the feature space (simply X_n if ϕ is the identity). For RLS, the model will become (from linear to non-linear):

$$w = (X_n^T X_n + \lambda n I)^{-1} X_n^T Y_n \rightarrow w = (\Phi^T \Phi + \lambda n I)^{-1} (\Phi^T Y)$$

When D is huge, $\Phi^T \Phi \in \mathbb{R}^{DxD}$ is not computable.

Unfortunately, selecting the appropriate type and size D of the feature space is not straightforward and a usual choice is to choose D very large ($\rightarrow \infty$)

It can be useful to use the **representer theorem** in this case, which key result is that the solution of regularization problems can be written as:

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} (y_i - w^T x_i)^2 + \lambda ||w|| \rightarrow w = \sum_{i=1}^{n} x_i c_i$$

Keep in mind that we can use features by substituting x_i with $\phi(x_i)$

In the case of RLS:

$$c = (X_n X_n^T + n\lambda I)^{-1} Y_n$$

–

$$w = X_n^T c$$

To be more generic, we can use the **kernel matrix**:

$$c = (K_n + n\lambda I)^{-1} Y_n$$

Where $K_{i,j} = x_i^t x_j$ in the example seen before.

One of the main advantages of using the representer theorem is that the solution to the problem
depends on the input points only through inner products.

Kernel methods can be seen as replacing the inner product with a more general **kernel function** $K(x, x_i)$, that **behaves like an inner product** (symmetric and positive definite). In this case, the representer's theorem becomes:

$$\hat{f}(x) = \sum_{i=1}^{n} K(x, x_i) c_i$$

The main idea is that we don't need to first find w by minimizing the empirical loss and then calculate $f_w(x) = w^t \Phi(x)$, which is computationally expensive. Instead, we first calculate the c using the kernel matrix/function and then find the function $\hat{f}$ by using the representer theorem.

The Kernel functions seen during the course are:

- **Linear Kernel**: $K(x, x') = x^T x'$ (inner product, scalar result)
- **Polynomial Kernel**: $K(x, x') = (x^T x' + 1)^d$
- **Gaussian Kernel**: $K(x, x') = e^{-\frac{||x - x'||^2}{2\sigma^2}}$

N.B.: Pay attention to the c, it often changes meaning accordingly to the context, so it might look confusing sometimes for that reason.

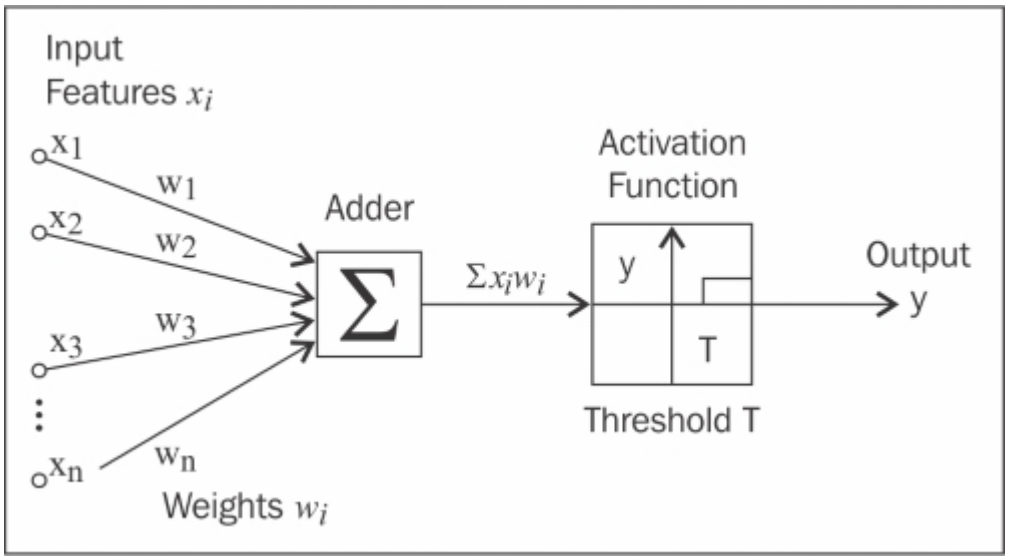
Neural Networks (NNs)

Deep learning is a family of machine learning methods based on artificial neural
networks (ANNs) that use multiple layers to progressively extract higher level features from raw input. ANNs are computing systems inspired by the biological
neural networks, based on a collection of units/nodes, the artificial neurons.

NNs are an alternative way of deriving non-linear functions with non parameterisation.

Single layer perceptron

McCulloch & Pitts Neuron Model:



We first calculate the **weighted sum** $G(X)$ of the inputs $x_i \in X$:

$$g(X) = \sum_{i=0}^{m} x_i w_i$$

The weights allow us to make some neurons in the networks have more influence than others.

After that, we evaluate the total result using the **activation function** σ. If it is higher than the **activation threshold** Θ, we set the binary output Y to 1 (which indicates that the neuron fires), otherwise we set it 0 (which indicates that it is at rest).

$$f(X) = \sigma(g(X)) = \begin{cases} 1 \text{ if } g(X) > \Theta \\ 0 \text{ otherwise} \end{cases}$$

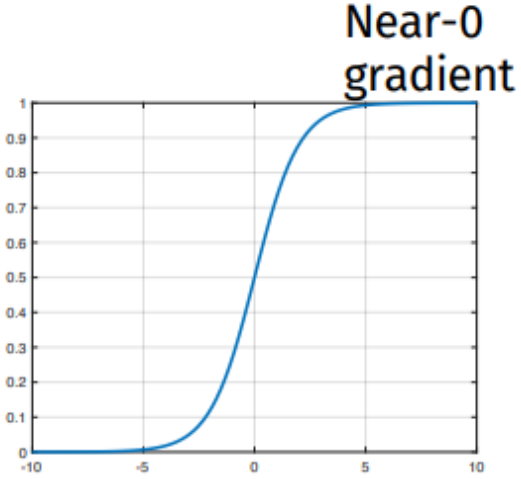
We can also have a **bias term** w_0 (w_0 = 1, for example) included inside the sum in $g(X)$:

$$g(X) = \sum_{i=1}^{m} x_i w_i + w_0$$

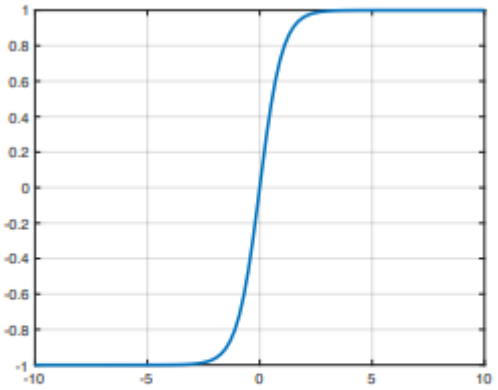
By using this particular activation function, we can't handle non-linearities. In fact, we can represent the operators **and** and **or** by setting the threshold to 1.5 and 0.5 respectively (0 + 0 = 0, 1 + 0 = 0 + 1 = 1, 1 + 1 = 2) , but we can't represent the non-linear function **xor** (fires when the result is 1, but not when it is smaller, 0, or greater, 2).

Since **nonlinear activation** is key to achieving good function approximation, let's see some popular activations functions to use as σ:

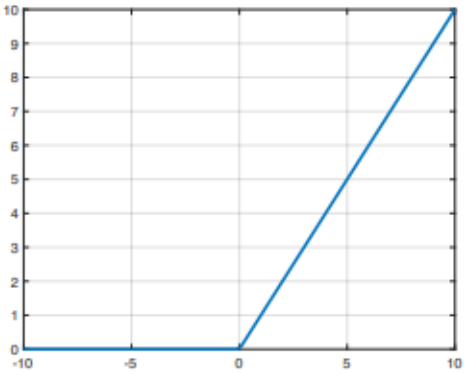
- **Sigmoid:** The most used for **binary classification**, corresponds to the logistic regression. It suffers from the vanishing gradient and has a non-zero centered output that may cause zig-zagging.
- **Tanh**: It is a **scaled version of the sigmoid**, that still suffers from the vanishing gradient, but has a zero-centered output.
- **ReLU**: It thresholds values below 0 and allows for fast convergence of the optimization function, but the weight may irreversibly die.
- **Leaky ReLU**: It is aimed to fix the dying ReLU problem.



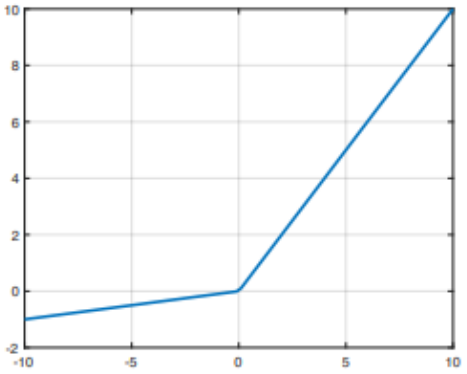
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



$$f(x) = max(0, x)$$



$$f(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

$$\alpha = 0.1$$

Multi-layer perceptron

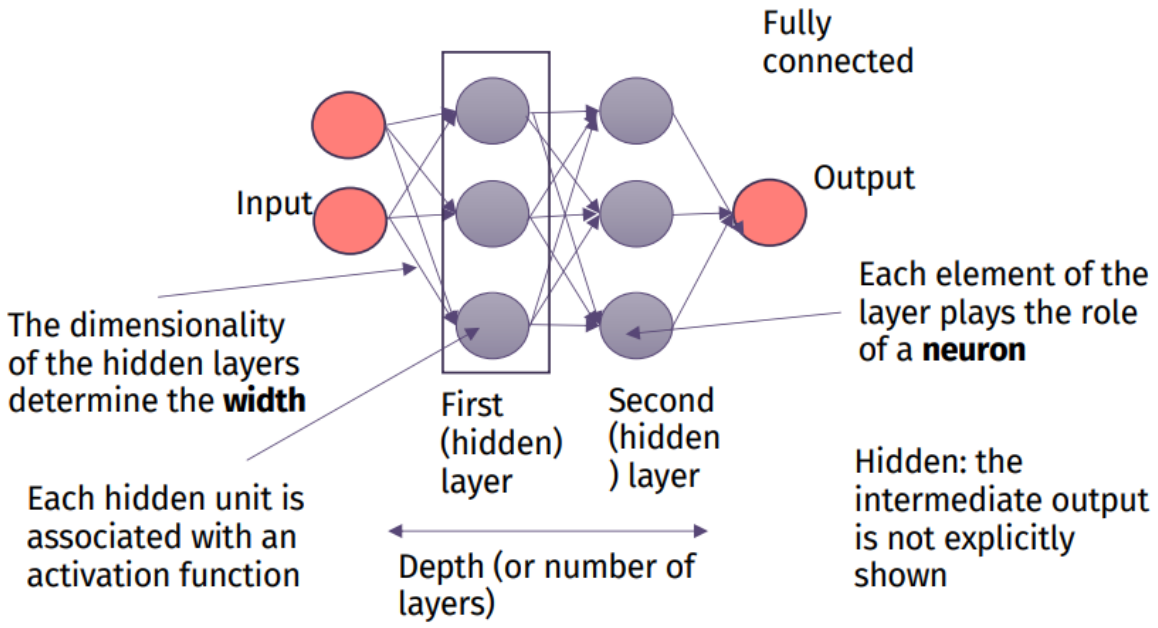
The **deep feedforward networks** (aka **multilayer perceptrons MLPs**) are represented by composing together many different functions. The are composed as chains of layers that may be of three different types:

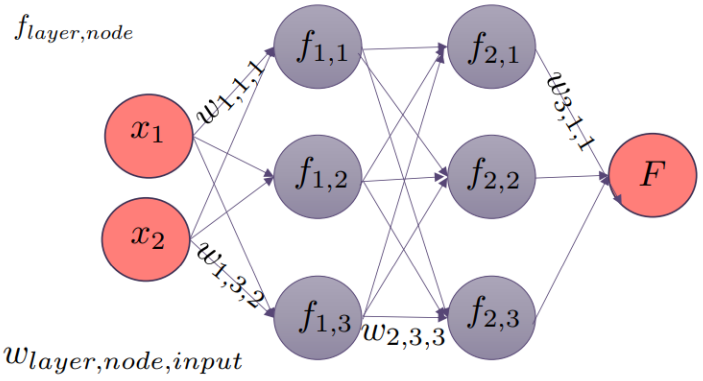
- Input layer (pink left)
- (Multiple) Hidden layer(s) (violet middle)
- Output layer (pink right)

The **width** of a neural network is the the number of hidden units of the hidden layers (dimensionality), while the **depth** is the total number of layers.

The hidden layers are called this way because their output is not shown explicitly and each unit is associated with an activation function. In general, each unit of the network works as a neuron.

In this case, the network is **fully connected** because the output of a unit becomes the input of each unit of the following later.





In the neural network beside, the number of parameters is the sum of the number of weights (one for each arch) and the number of bias (one for each function). In this case, the total is 6 + 9 + 3 (the number of weights for layer) + 7 (the number of activation functions) = 25 in total.

The **universal approximation theorem** tells that a infinitely wide single hidden layer can represent any function. This is **infeasible** in practice, and the resulting model may not generalize well (overfit).

Training a DNN means (as usual) learning the values for the model parameters (weights, bias terms) from the training set. We want them to minimize the **cost function** J (we normally use as **loss function** the **square loss for regression** and **cross-entroty log for classification**)

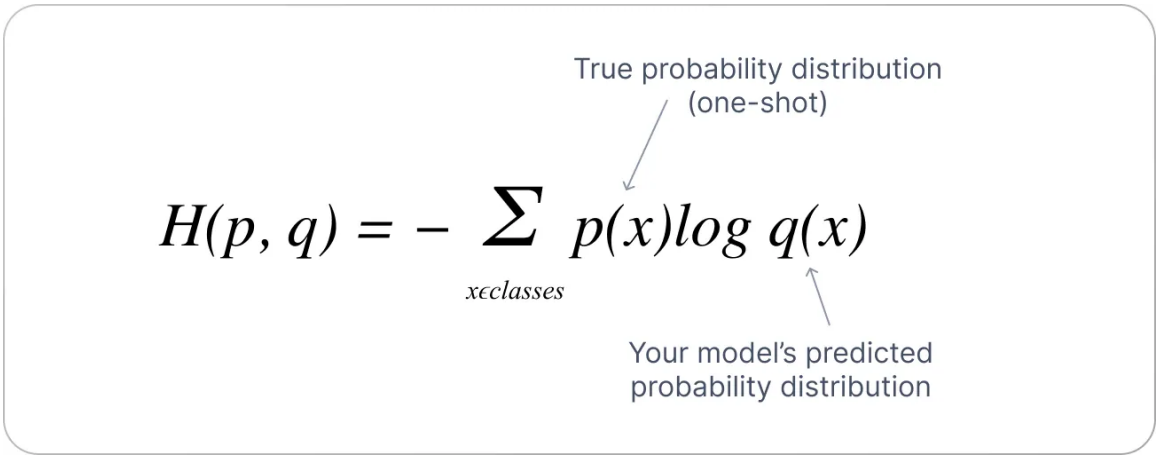
$$L_1 = \sum_{i=1}^{n} |y_{gt} - y_{pred}|$$

$$L_2 = \sum_{i=1}^{n} (y_{gt} - y_{pred})^2$$

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |y_{gt} - y_{pred}|$$

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_{gt} - y_{pred})^2$$

L1 and L2 regualrization and (mean) squared loss (MSE)



Cross-entropy loss

$$W^* = \arg\min_W \frac{1}{n} \sum_{i=1}^{N} \mathcal{L}(F(X^{(i)}; W), y^{(i)})$$

$$W^* = \arg\min_W J(W)$$

Forward propagation is where input data is fed through a network, in a forward direction, to generate an output. The data is accepted by hidden layers and processed, as per the activation function, and moves to the successive layer.

Back propagation aims to miminize the cost function by adjusting the parameters of the networks by **calculating the gradient**. Thanks to the **chain rule**, it is possible to calculate the gradient going backwards, taking the partial derivative of the cost function. Based on the calculations, neural network nodes

with high error rates are given less weight than nodes with lower error rates, which are given more weight.

$$\frac{\partial J^i(\mathbf{w})}{\partial w_k}$$

Gradient Descent uses the gradients calculated previously to **update the weights** in order to minimize the cost function. The parameter updates are performed by moving the parameters in the opposite direction of the gradient.

$$w_0 = 0$$

$$w_t = w_{t-1} + \gamma \nabla J(w_{t-1})$$

For output units, the **sigmoid function is a good choice for binary classification** (if it returns a value greater than 0.5, we predict 1, otherwise 0), while the **Softmax operator can be used for multi-class classification**.

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Softmax: real value to probability

Since DNN models have a huge number of parameters, this leads to **high chance of overfitting** with models that do not generalize, so we need to use regularization. Overfitted DNN models tend to suffer from a problem of co-adaptation, which means that models weights are adjusted co-linearly to learn the model training data.

To solve this problem, we can use two techniques: **bagging** and **dropout**. Bagging involves training multiple models on different subsets of the training

data, while dropout randomly drops units (along with their connections) from the neural network during training. Since bagging is comptuationally expensive, we use dropout to approximate it. Another solution is adding **weight regularization** (L1 or L2 regularization) or **early stopping** (because the more we train in terms of epochs, the more the model will overfit).

Considering gradient descent and variations, the terminology used is the following:

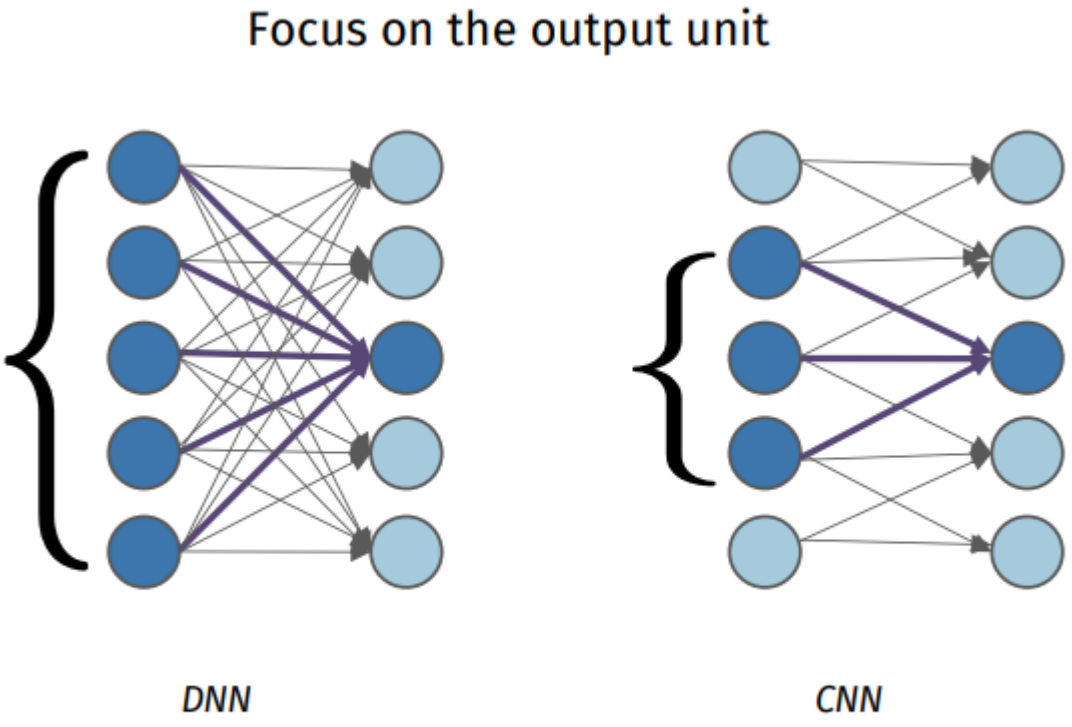
- **Epoch**: is when the **entire dataset** is passed forward and backward through the neural network only once (multiple epochs are usually needed)
- **Batch size**: is the number of training examples in a mini-batch (see GD)
- **Iteration**: is the number of batches needed to complete one epoch

Convolutional Neural Networks (CNNs)

A specialized kind of neural network for processing data with a known **grid-like topology** (1D, like time-series, or 2D, like images).

CNNs leverage two important principles that differ from Dense NNs:

- **Sparse interaction:** In traditional DNNs every output unit interacts with every input unit, while in CNNs it is often not the case. The pros is that we have to learn **fewer parameters**, with improvements in memory requriements, computations and statistical efficiency. In fact, dense networks as we've seen assume that everything works idependently, so they are not good for images and time series in which there are relations over space and time.



- **Parameter sharing:** In traditional DNNs each weight is used exactly once when
  computing the output, while in CNNs instead of having a separate weight for each connection between input pixels and neurons in the next layer, a **single set of weights (kernel)** is used to convolve across the entire input.

The convolution/cross-correlation operation:

A "feature detector" (kernel) slides over the inputs to generate a feature map:

$$s(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$

X_{11}	X_{12}	X_{13}	X_{14}
X_{21}	X_{22}	X_{23}	X_{24}
X_{31}	X_{32}	X_{33}	X_{34}
X_{41}	X_{42}	X_{43}	X_{44}

*

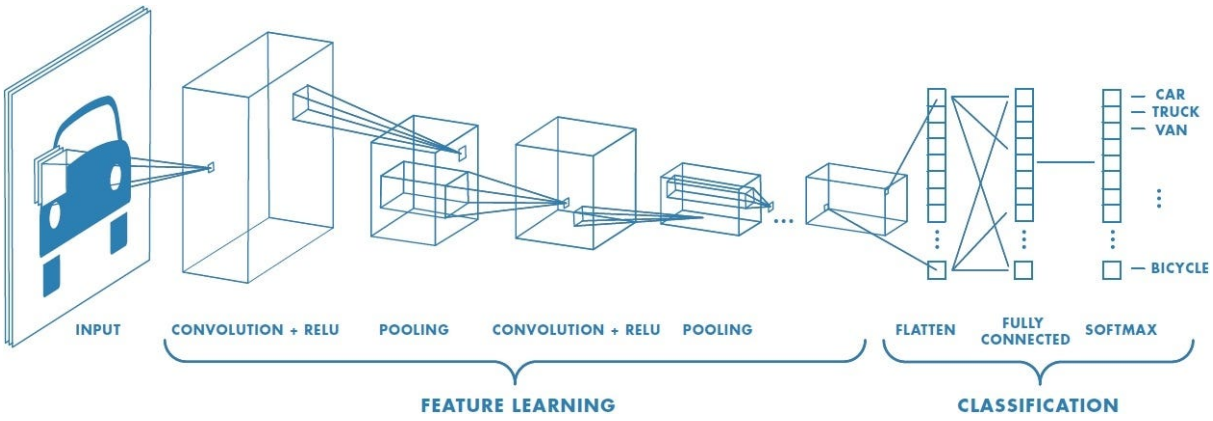
W_{11}	W_{12}
W_{21}	W_{22}

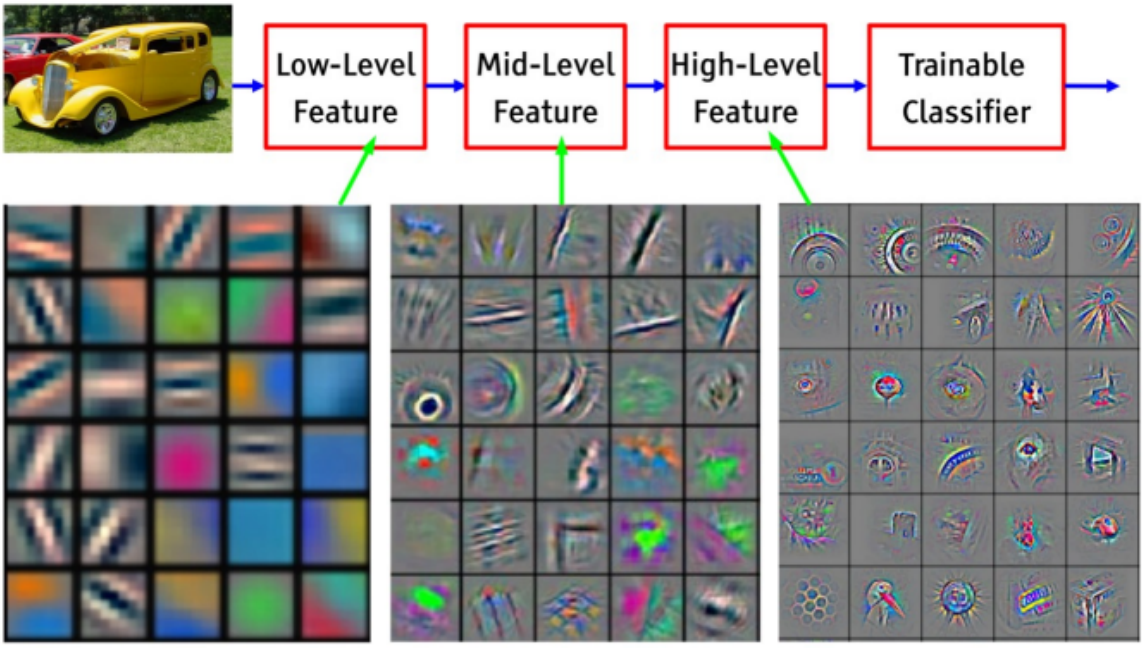
=

Y_{11}	Y_{12}	Y_{13}
Y_{21}	Y_{22}	Y_{23}
Y_{31}	Y_{32}	Y_{33}

$Y_{11} = X_{11}W_{11} + X_{12}\,W_{12} + X_{21}\,W_{21} + X_{22}\,W_{22}$
$Y_{12} = X_{12}W_{11} + X_{13}\,W_{12} + X_{22}\,W_{21} + X_{23}\,W_{22}$
$Y_{13} = X_{13}W_{11} + X_{14}\,W_{12} + X_{23}\,W_{21} + X_{24}\,W_{22}$
.....

From shallow to deep models: shallow models don't extract features from the data (it is a human that needs to do so), only classification, while deep models do both things (**feature learning** + **classification**).





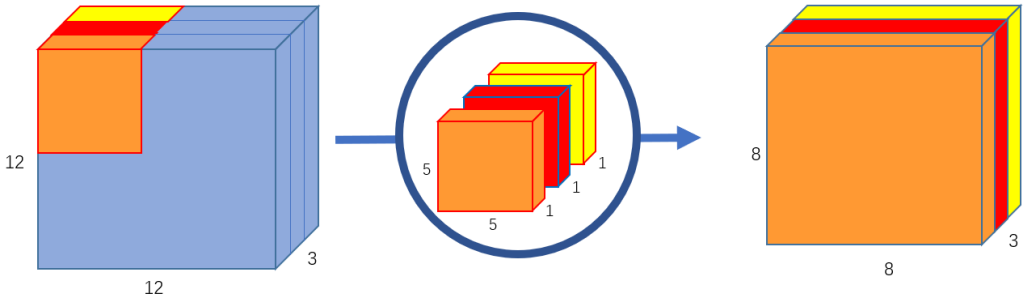
Firstly, we extract low-level features, then high-level

A typical CNN layer replaces the output at a certain location with a summary statistic of nearby outputs (**detector stage**), producing a set of linear activations.

As the kernel slides on the image, it is able to capture the same property in different image regions. Multiple feature detectors can be used to capture different image properties.

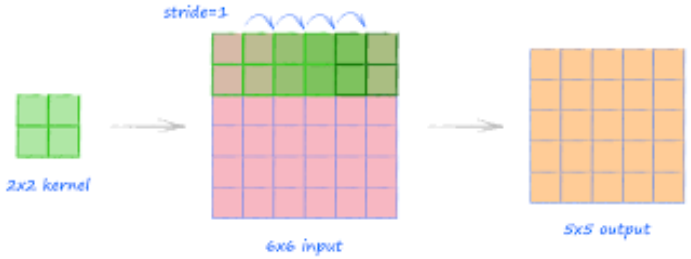
Three parameters control the size of the output of a layer:

- **Depth**: the number of filters (kernels) of the layer

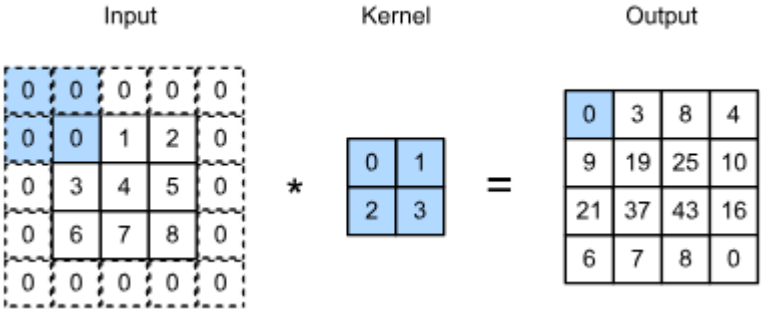


- **Stride**: the step used to slide the filter on the input. When stride > 1 we are down-sampling the

input data and the term
tiling refers to the special case where stride = kernel span.

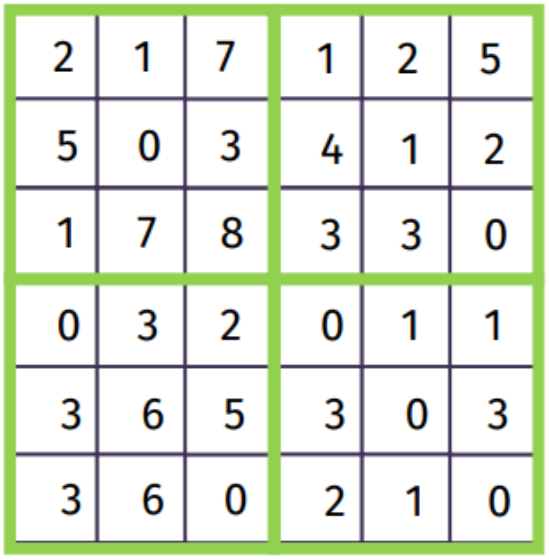


- **Padding**: it is used to enlarge the input and allow for kernels application in each one of the (original) point.



Pooling is a way to further reduce the dimensionality of the description and provides invariance to small shifts of the inputs. The pooling functions are:

- **Average pooling:** average activation of the convolutional layer
- **Max pooling:** max activation of the convolutional layer

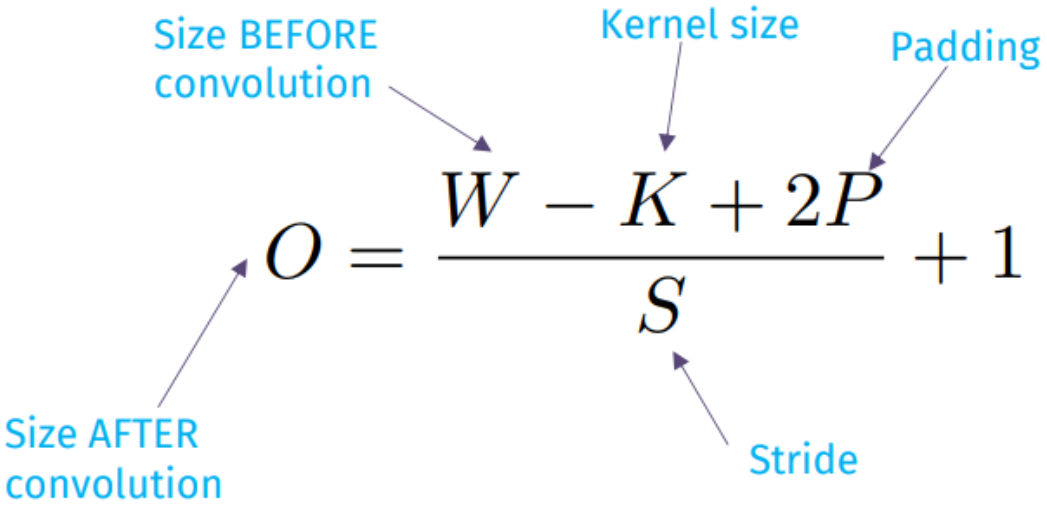


Max pooling

8	5
6	3

Average pooling

3.8	2.3
3	1.2



Bayesian models

The basic idea of **Maximum Likelihood Estimation (MLE)** to assume a parametric model for the unknown the distribution and then estimate the parameter which mostly likely generated the data, instead of focusing only the quantity of interest as in the methods seen previously.

If the previous methods (ERM), the idea was to find w that minimizes the loss function:

$$\text{ERM: } \min_{w \in \mathbb{R}^d} l(f_w(x), y_i)$$

Here we want to find the parameters θ that **maximizes** the joint distributions of the data (**likelihood**). Since the data are independent, their joint distribution is product of the individual distributions:

$$\text{MLE: } \max_{\theta \in \mathbb{R}^d} \prod_{i=1}^{n} p_\theta(z_i)$$

For example, considering the Gaussian distribution so that $z \sim \mathcal{N}(\theta, 1)$:

$$p_\theta(z) = \frac{1}{\sqrt{2\pi}} e^{\frac{-|z-\theta|^2}{2}}$$

We can do the same considering also **linear regression with Gaussian noise** $y_i = w^t x_i + \epsilon_i$, with $\epsilon \sim \mathcal{N}(0, \sigma)$. The formula compare y_i with $w_t x_i$ because we are considering the error, not the input like before.

$$\text{ERM: } \min_{w \in \mathbb{R}^n} \sum_{i=1}^{n} \frac{(y_i - w^T x_i)^2}{2\sigma^2}$$

$$\text{MLE: } \max_{w \in \mathbb{R}^n} e^{-\sum_{i=1}^{n} \frac{(y_i - w^T x_i)^2}{2\sigma^2}}$$

And **logistic regression**, and so on.

The basic idea of **Bayesian estimation** is that the parameter of interest follow a prior unknown distribution reflecting our prior knowledge/beliefs on the problem. The idea is then to derive the
so called posterior distribution, obtained conditioning the prior to the observed data. The
basic tool is the Bayes rule:

$$P(U|V) = \frac{P(V|U)P(U)}{P(V)}$$

In the case of **linear regression**, the unknown parameter of interest is the coefficient vector w. We assume it is distributed according to a standard multivariate Gaussian $\mathcal{N}(0, 1)$. The idea is to apply the Bayes rule to the data and w seen as random quantities:

$$P(w|X_n, Y_n) = \frac{P(Y_n|X_n, w)P(w)}{P(Y_n|X_n)}$$

Where $P(w|X_n, Y_n)$ is the **posteriori,** $P(Y_n|X_n, w)$ is the **likelihood** and $P(w)$ is the **prior** and $P(Y_n, X_n)$ is the **marginal likelihood**, that is only a normalizing factor since it doesn't depend on w.

The above equality can be seen as how our prior beliefs are updated after observing the data (posteriori) and we can get rid of the normalizing factor:

$$P(w|X_n, Y_n) = P(Y_n|X_n, w)P(w)$$

In the above equality, the posterior is given by the product of two Gaussian distributions:

$$P(Y_n|X_n, w) = e^{-\sum_{i=1}^{n} \frac{(y_i - w^t x_i)^2}{2\sigma^2}}$$
$$P(w) = e^{-||w||^2}$$

A possible parameter estimate is given by computing its maximum (which is also equal to its mean).
This parameter estimate is called
Maximum A Posteriori (MAP):

$$\text{MAP: } \max_{m \in \mathbb{R}^d} e^{-\sum_{i=1}^{n} \frac{(y_i - w^t x_i)^2}{2\sigma^2} + ||w||^2}$$

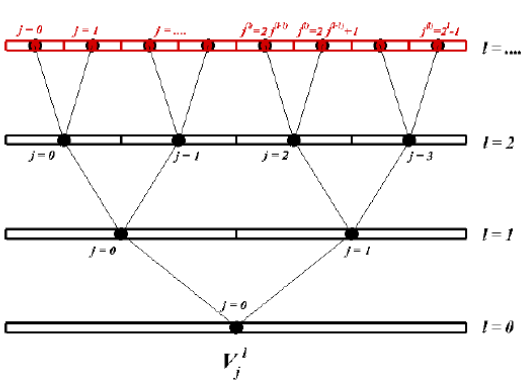
$$\text{RLS } \min_{w \in \mathbb{R}^d} \sum_{i=1}^{n} \frac{(y_i - w^t x_i)^2}{2\sigma^2} + ||w||^2$$

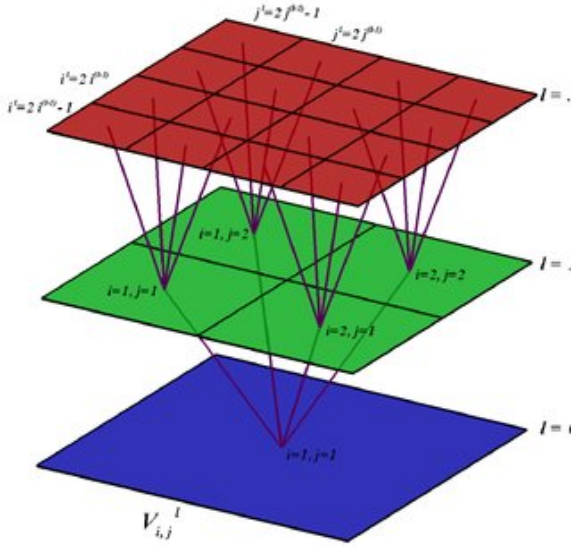
Binary decision trees

A **partition** of the input space X_n is a family of disjoint sets (**cells**) whose union is the whole input space:

$$\mathcal{A} = \{\mathcal{A}_j \subseteq X, j = 1, ..., J | X = \bigcup_{j=1}^{J} \mathcal{A}_j, \mathcal{A}_j \cap \mathcal{A}_j = \emptyset, i \neq j\}$$

A basic example is the **dyadic partition**, where each cell A_j is a cube of sidelength 2^{-k} for some integer k.





Given a partition $\mathcal{A}$, the simplest **partition based estimator** learns a constant function $\hat{c}_j$
approximation in each cell
$\mathcal{A}_j$ of the partition. A natural way to estimate them is to consider the following problem for each cell:

$$\min_{c_j \in \mathbb{R}} \sum_{x_i \in \mathcal{A}_j} (y_i - c_j)^2$$

That can be approximated by considering the mean of all values inside the cell:

$$\hat{c}_j = \frac{1}{n_j} \sum_{x_i \in \mathcal{A}_j} y_i$$

We can equivalently derive the above estimator introducing the class of piecewise constant functions:

$$\mathcal{H} = \{f : X \to \mathbb{R} | \sum_{j=1}^{J} c_j \hat{1}_{x \in \mathcal{A}_j}, \forall x \in X, c_1, ..., c_j \in \mathbb{R}\}$$

Where $\hat{1}$ is the characteristic function that returns 1 if $x \in \mathcal{A}_j$ and 0 otherwise.

And considering the problem:

$$\min_{c_j \in \mathbb{R}} \sum_{x_i \in \mathcal{A}_j} (y_i - f(x_i))^2$$

The above approach is just a particular instance of ERM. To generalize it even further, we can substitute c_j with a generic function $g(x)$ (such as $w_j^T x + b_j$, for example).

Sparsity

In many practical situations, beyond predictions it is important to obtain interpretable
results (
x-plainable AI). Interpretability is often related to detecting which factors have determined our prediction.

Here we can think of the components x_j of an input as of specific measurements (e.g. pixel values)

Given a training set, the goal of variable selection is to detect which variables are important for prediction. The key assumption is that the best possible prediction rule is **sparse (**only few of the coefficients in are different from zero).

A **brute force approach** would be to consider all the training sets obtained considering all the
possible subsets of variables. For each training set one would solve the learning problem and eventually end selecting the variables for which the corresponding training set achieves the best performance. The described approach has an exponential complexity and becomes unfeasible even for relatively small
d.

If we consider the square loss, it can be shown that the corresponding problem could be written as:

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} (y_i - f_w(x_i))^2 + \lambda ||w||_0$$

where $||w||_0 = |\{j | w^j \neq 0\}|$ is called l_0 norm and counts the number of non-zero components in w.

In the following, we'll consider different approaches to find approximate solution to the above problem

- **Greedy methods**

This class of approaches to variable selection generally encompasses the following steps:

1. initialize the residual, the coefficient vector, and the index set
2. find the variable most correlated with the residual
3. update the index set to include the index of such variable
4. update/compute coefficient vector
5. update residual

The simplest such procedure is called forward stage-wise regression in statistics and **matching pursuit (MP)** in signal processing. Let denote the residual $r \in \mathbb{R}^n$, the coefficient vector $w \in \mathbb{R}^d$ and the index set $I \subseteq \{1, \dots, d\}$. X^j indicates the column j of the input training data matrix X and $i = 0, \dots, T$, where T is the maximum number of iterations

1. $r_0 = Y_n$, $w_0 = 0$, $I_0 = \emptyset$: Since the coefficient vector is all zeros, the residual is every part of the output. In this case, t = 0.
2. We select the variable such that the projection of the output on the corresponding column k is larger, or, equivalently, we select the variable such that the corresponding column best explains the the output vector in a least squares sense (such as below):

$$j^* = \underset{j \in 1, \dots, d}{\arg\min} ||X^j v_j - r_{i-1}||^2$$
$$v_j = \underset{v \in \mathbb{R}}{\arg\min} ||X^j v - r_{i-1}||^2$$

3. $I_i = I_{i-1} \cup \{j^*\}$
4. $w_i = w_{i-1} + w_{j^*}$, where $w_{j^*} = v_{j^*} e_{j^*}$: Remember that v is a vector and e_{j^*} is the element of the canonical basis in $\mathbb{R}^d$, with k-th component different from zero.
5. $r_i = r_{i-1} - X w_{j^*}$: The subtraction will get the residual closer to zero as the coefficient vector gets populated. We use w_{j^*} instead of w_{t+1} because we just want to update considering the new component discovered in this iteration.

A variant of the above procedure, called **Orthogonal Matching Pursuit (OMP)**, is also often consid-
ered. The corresponding iteration is analogous to that of MP, the differences are at the following steps:

4. $w_i = \arg\min_{w \in \mathbb{R}^d} ||Y_n - X_n M_i w||^2$, where M_i is a dxd matrix where $(M_i w)^j = w^j$ if $j \in I$ and 0 otherwise: here, since we are considering directly Y_n not r_{i-1}, we want to pick the others w^j used before to select the new one.
5. $r_i = Y_n - X_n w_i$: There we use w_i directly because we are subtracting every time directly from Y_n instead of r_{i-1}.

- **Convex relaxation**

We replace the l_0 norm with $l_1 = \sum_{j=1}^{d} |w_j|$, obtaining:

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} (y_i - f_w(x_i))^2 + \lambda ||w||_1$$

The above problem is called **LASSO** in statistics and Basis Pursuit in signal processing. A simple yet powerful procedure to compute a solution is based on the so called **Iterative Soft Thresholding Algorithm (ISTA)**

On the one hand, while Tikhonov allows to compute a stable solution, in general its solution is not sparse. On the other hand, the solution of LASSO might not be stable. The elastic net algorithm is a trade-off between the two.

Clustering

In **supervised learning**, it's like learning with a teacher because each input x_i of the training data S_n is paired with a label y_i. In **unsupervised learning**, the training set S_n is composed only by inputs from x_0 to x_n.

In addition, in supervised learning we have a measure of success based on a loss function and on a model selection procedure such as cross validation, while in unsupervised learning we don't.

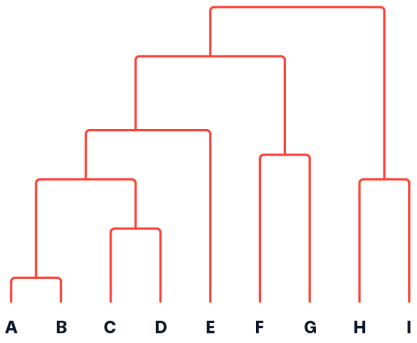
We are going to see the **clustering** technique, which main goal is to group a collection of objects into subsets (clusters). It can also be used to arrange clusters into a natural hierarchy. Let's see some algorithms for it:

- Algorithm DB-Scan:

1. Density estimation
2. Thresholding
3. Find connected regions

- Agglomerative-HC (Hierarchical Clustering):
  1. All nodes are considered one cluster (which is implemented as a set).
  2. The two closest clusters merge (if the cluster has more than one node, you take the average)
  3. Repeat step 2 until all nodes are in the same cluster

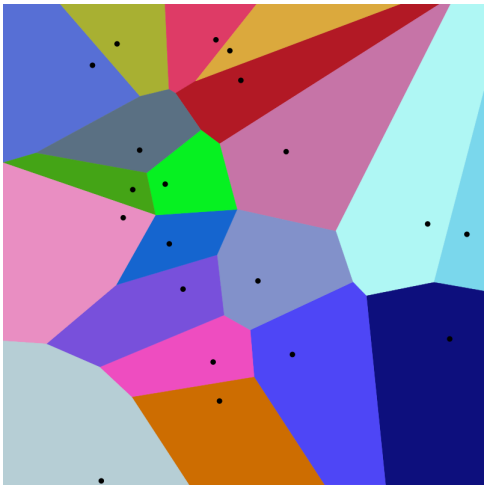
  The output is a **dendogram**, where the x-axis contains the values for the nodes and each depth is a new cluster (in the example below, A and B are the closest two points):



- **K-means clustering**: It divides the data in X in clusters C_i to C_k so that $\bigcup_{i=1}^{k} C_i = X$ and $C_i \bigcap C_j = \emptyset$ for every $i \neq j$. To define which point belongs to which clusters, we use **Voronoi regions**, that are defined as:

$$C_i = \{x \in X | \text{dist}(x, m_i) \leq \text{dist}(x, m_j) \forall j \neq i\}$$

Where $\text{dist}(x, m) = ||x - m||$ (Euclidean distance) and m_i is the **centroid** (or **cluster center**) of the cluster C_i.



The algorithm for K-means clustering consists of:

1. **Initialize** the k cluster centroids
2. **Assign** x_i to the cluster which centroid is closest (as in the formula above).
3. **Update** the centroids according to the mean of the points assigned to the corresponding cluster.
4. Repeat step 2 and 3 until **convergence**

Steps 2, 3 and 4 are the **Lloyd's algorithm**.

For the initialization, we can go random or use **K-means++ algorithm**, that works by selecting the cluster centers as the points within the dataset which have the **maximum distance** between each other. This last approach often works better beause the K-means algorithm performance strongly depends on how the centers are initialized.

An approach to identify the number of clusters can be based on computing and comparing the solutions obtained on different data splits.

Another unsupervised algorithm is **Principal component analysis (PCA)** for dimensionality reduction.

Useful tips for the exam:

Learning problems:

- Regression: Predict a number

- Classification: Classify into categories

Describe data: dimension

Scheme for training and evaluating an estimated function (assuming to have train, validation and test set): Estimating different instances of the solution from the training set (for example, trying different hyperparameters) and picking the best performing on the validation set. Just at the end verify how it goes on the test set

Things I've not covered:

- More details about parzen windows
- The details on the formula for bias and variance

https://stats.stackexchange.com/questions/159070/curse-of-dimensionality-knn-classifier

- How the additional step with c works in RLS + complexity
- Regularized Logistic Regression (page 23 on)
- ERM in the case of RLS
- Offset
- I know the notes I've many notation inconsistencies, but in general if a letter has not a symbol on it, it means that it is the actual thing. If it has a star *, it means it is the optmal value for the varable. If it has any other symbol, it is just an approximation. $\mathcal{E}_n$ was $\hat{L}$ during the lessons.
- Newton's method

https://jrodthoughts.medium.com/bagging-and-dropout-learning-ae484023b0da

https://h2o.ai/wiki/backpropagation/

https://h2o.ai/wiki/forward-propagation/