## 0.1 Machine Learning Theory

*(Lezione 2 del 08/10/19) Compilata: 9 ottobre 2019*

Let's define the main elements of a machine learning network.

- **Domain set** (or *instance space*), denoted with $\mathcal{X} = \{\boldsymbol{x}\}$. It consists of the set of all *objects* (in an abstract sense) to make predictions about. Each sample is usually a vector $\boldsymbol{x} \in \mathbb{R}^d$, called the *features vector*, that contains a set of functions of input data. For example, we can consider a set $\mathcal{X}$ of *houses*, represented with vectors containing their price, location (longitude/latitude), number of rooms...

- **Label set** ($\mathcal{Y}$), i.e. the set of possible labels associated with each domain sample. In a problem of classification, the labels are merely the relative classes of input samples. For example, if the domain is a set of images of digits, the labels will be $\in \{0 \dots 9\}$.

- **Training set** $S = ((x_1, y_1), \dots, (x_m, y_m))$. It is a finite sequence that associates *domain samples* with their *labels* ($S \subseteq \mathcal{X} \times \mathcal{Y}$).

- **Prediction rule** ($h_S \colon \mathcal{X} \to \mathcal{Y}$, also denoted with $\hat{f}$, as an estimate/predictor - given the information contained in the training dataset $S$ - of the underlying, unknown, function $\hat{f}$ relates domain points to labels). This represents the algorithm's output. More precisely, one can use the notation $A(S)$ to indicate the $\hat{f}$ learned by the algorithm ($A$) using the specific set $S$ as *training set*. This is important, because the general performance of the algorithm is tied to the quality of $S$, which needs to be a fair sampling of the real-world data (reproducing the intrinsic distribution: for example, if we consider an algorithm to distinguish between males and females from photos, a skewed $S$, containing e.g. much more males than females, will result in an algorithm that doesn't really generalize well in a everyday situation).

- **Data generation**: parameters needed to produce domain instances and their labels. This is for example a sampling function with a predefined probability distribution $D$, and a labelling function $f$. Of course, both $D$ ad $f$ are not known by the ML algorithm. We will denote with $D(A) \in \mathbb{R}$ the probability to sample the set $A$ from $D$. For example, if $D$ is a distribution of heights in a certain population, $D(\{x | 1.6 < x < 1.8\})$ is the probability that a random person will have an height between 1.6 and 1.8.
Of course, if domain instances are result of experiments/real world phenomena (as it happens most of the time), $D$ and $f$ may not be known at all - but we still postulate their existence.

- **Measure of success**: a function to compute the performance of the algorithm. Usually one computes the *error*, that is the probability that the algorithm does not predict the correct label on a random data point sampled by $D$.

### 0.1.1 Measure of success: Loss function

Given a domain subset $A subset \mathcal{X}$, denote with $D(A)$ the probability of observing a point $x \in A$. In many cases, we refer to $A$ as *event* and express it using a *characteristic function* $\pi\colon \mathcal{X} \to \{0,1\}$, that is:

$$A = \{x \in \mathcal{X}\colon \pi(x) = 1\}$$

In this case we have $\mathbb{P}_{x \sim D}[\pi(x)] = D(A)$, that is the probability $\mathbb{P}$ that a point $x$, sampled from distribution $x \sim D$ is such that $\pi(x) = 1$ ($\pi(x)$ is "true").
We define the **error of prediction rule** $h_S\colon \mathcal{X} \to \mathcal{Y}$ is:

$$L_{D,f}(h) \overset{\text{def}}{=} \mathbb{P}_{x \sim D}[h(x) \neq f(x)] \overset{\text{def}}{=} D(\{x\colon h(x) \neq f(x)\})$$

Note that this kind of **loss** $L$ is useful only for classification tasks. In fact, if we aim to predict a continuous number, we'd like to consider an estimate of 0.9, wrt a true label of 1.0, as "somewhat close, not totally wrong". However, the loss here defined would consider this case as wrong as that which results in an estimate of $-965$.

### 0.1.2 Empirical Risk Minimization

We now tackle the problem of optimizing $h_S\colon \mathcal{X} \to \mathcal{Y}$ such that it minimizes the **generalization error** (loss) $L_{D,f}(h)$. However, $D$ and $f$ are not known: so we consider the **error on training data**, that is the training error:

$$L_S(h) \triangleq \frac{1}{m} |\{i | h(x_i) \neq y_i, 1 \leq i \leq m\}|$$

that is the ratio of correct predictions to total number of samples $m$ (in the case of a binary classification problem).
We call **empirical risk minimization** (ERM) the task of finding the correct form of $h \in \mathcal{H}$ in a set of possible predictors $\mathcal{H}$ such that it minimizes the training error $L_S(h)$:

$$\text{ERM}_{\mathcal{H}}(S) \in \underset{h \in \mathcal{H}}{\text{argmin}} L_S(h)$$

Note that this problem usually does not have a unique solution.
Of course, there is no guarantee that minimizing the error on the training dataset will also minimize the generalization error. In fact, maybe the training set is not big enough to account of the diversity in the relevant population.
So, it could happen that a *perfect* predictor on the training dataset (that is, a predictor which never commits errors), is very much wrong if tested on a different dataset.
So, training error **is not** a good measure of true error.

### 0.1.3 Overfitting

The problem where training error is not representative of true error is called **overfitting**. There are various ways to deal with it:

- Use a larger training dataset
- Restrict the set of hypothesis $\mathcal{H}$. For example, if we consider $\mathcal{H}$ to be the set of polynomials (for a regression problem), one can restrict the *order* of polynomial, so that the algorithm needs to choose a “simpler” (and hopefully more general) predictor.
  This is equivalent to *insert* into the algorithm some **prior knowledge** about the problem. For example, if we want to build a regression algorithm to predict a physical value, and we know the underlying physical laws, it is clear how to restrict $\mathcal{H}$ to the set of relevant function that we are interested to.

We can further analyse the problem of overfitting if we make some simplifying assumptions. Let's start by denoting:

$$h_S \in \arg\min_{h \in \mathcal{H}} L_S(h)$$

Then we make the following assumptions:

- Assume $\mathcal{H}$ is a finite class, that is a set of finite cardinality $|\mathcal{H}| < \infty$. This is true in practice, as memory is limited, and model's parameters are represented with $n$-bits, $n < \infty$.
- **Realizability**: it exists a perfect solution. This is only true in some ideal cases - as real-world problems are always messy and not clearly defined (in fact, later on we will drop this assumption).
  More precisely, we assume that:
  $$\exists h^* \in \mathcal{H} \text{ such that } L_{D,f}(h) = 0$$
  Generally, unfortunately $h^* \neq h_S$. Note, however, that as $h^*$ works in the general case, it follows that $L_S(h^*) = 0$, that is the best solution obviously work on the training dataset.
- Examples in the training set are independently and identically distributed (i.i.d) according to a certain (unknown) distribution $D$, that is $S \sim D^m$ ($S$ is a set of $m$ samples of $D$).
  In practice, this is difficult to do, because gathering training data always involves some kind of bias. For example, if we survey students, it is impractical to visit every school in the world - and so the training set will over-represent certain classes, and under-represent others.

Given these assumptions, we want to know if we can learn $h^*$, that is make $h_S = h^*$.
The answer is no: in the general case, this cannot be done.
We can however *approximate* $h^*$, following the framework of **Probably Approximately Correct** (PAC) learning.
Let's introduce two parameters to measure how much $h_S$ is close to $h^*$:

- **Accuracy parameter** $\varepsilon$: we are satisfied with a good $h_S$ for which $L_{D,f}(h_S) \leq \varepsilon$.
- **Confidence parameter** $\delta$: we want $h_S$ to give correct predictions with a probability close to 1: $\geq 1 - \delta$ (depending on the choice of $S$). So a high $\delta$ means that, even with a bad choice of $S$, the algorithm is still good.

**Theorem 0.1.1.** *Let $\mathcal{H}$ be a finite hypothesis class. Let $\delta \in [0,1], \varepsilon \in [0,1]$ and $m \in \mathbb{N}$, such that:*

$$m \geq \frac{\log(|\mathcal{H}|/\delta)}{\varepsilon}$$

*where $m$ is the size of the training dataset, i.e. $S$ contains $m$ i.i.d. samples. Then for any $f$ and any $D$ for which the realizability assumption holds, with probability $\geq 1 - \delta$ we have that for every ERM hypothesis $h_S$ it holds that:*

$$L_{D,f}(h_S) \leq \varepsilon$$

Note that if we increase the required probability (lowering $\delta$) and the accuracy (lowering $\varepsilon$), the number of training samples $m$ rises (as expected).