SAPIENZA
UNIVERSITÀ DI ROMA

"Sapienza" University of Rome
Faculty of Information Engineering,
Informatics and Statistics
Department of Computer Science

Machine Learning

Lecture notes integrated with the book "Machine Learning", Tom Mitchell

Author
Simone Bianco

January 11, 2025

Contents

Information and Contacts

Personal notes and summaries collected as part of the *Machine Learning* course offered by the degree in Computer Science of the University of Rome "La Sapienza".

Further information and notes can be found at the following link:
https://github.com/Exyss/university-notes. Anyone can feel free to report inaccuracies, improvements or requests through the Issue system provided by GitHub itself or by contacting the author privately:

- Email: **bianco.simone@outlook.it**
- LinkedIn: **Simone Bianco**

The notes are constantly being updated, so please check if the changes have already been made in the most recent version.

Suggested prerequisites:

Sufficient knowledge of calculus, probability and algorithm design

Licence:

1
Introduction on Machine Learning

1.1 What is machine learning?

While many common tasks can be easily solved by computers through an algorithm, some are hard to formalize as a series of steps to be executed in a deterministic way. As an analogy, consider how language is made of syntax and semantics. Syntax can easily be formalized as a sequence of sub-structures that make up a phrase. If a sentence is slightly malformed, the machine can have an hard time trying to reconstruct the correct syntax, but in most cases this can be achieved. For semantics, instead, we have a whole different problem: some words could have more meanings, giving sentences different interpretations depending on the context of the conversation. This task is clearly harder for a machine. Sometimes, not even humans are capable of solving it!

In the past, these type of tasks where solved through *expert systems*, that being any system programmed by an human expert to solve a specific task. Expert systems can be viewed as a sequence of if-else conditions: if the task requires x then do y, and so on. Not all tasks can be solved through expert systems. In particular, some tasks need different solutions for many cases, making these primitive systems useless due to all cases being impossible to program.

To solve this type of complex and variable tasks, we use **machine learning**, which slowly teaches the machine how to solve the problem in the best way possible. The idea here is to program computers in a way that improves a specific *performance criterion* through *example data* and *past experiences.*

Machine learning uses **data mining** — the act of producing knowledge from known data — to increase the experience of the machine in solving the designed problem. In general, machine learning comes in handy when one of the following conditions holds:

- There is no human expertise on the task
- Human experts are unable to explain their expertise
- The solution needs to adapt to particular cases

The field of machine learning had an exponential growth in recent years due to the growing flood of online data — the so called *big data phenomenon* — and the increase of computational power to process such data through advanced algorithms based on theoretical results. First, we give a formal definition of a *learning problem*.

Definition 1: Learning problem

A **learning problem** is the improvement over a task T with respect to a performance measure P based on experience E.

For example, suppose that we want to program a machine that learns how to play checkers. We define the learning problem as:

- The task T is to play checkers
- The performance measure P is the percentage of games won in a tournament
- The experience E is the opportunity to play against self

But how can we improve such performance measure? What *exactly* should the machine learn? These questions reduce the learning problem to finding a valid mathematical representation of T, P and E. The training process can be described by four phases:

1. The human expert suggests what is an optimal move for each configuration of the board
2. The human expert evaluates each configuration, ranking them by optimality
3. The computer plays against an human an automatically detects with configurations lead to a win, a loss or a draw
4. The computer plays against itself to improve performance

Formally, this whole process can be expressed as a simple mathematical function called **target function**. In particular, we want to choose a target function that represents the learning problem in the best way possible and that can be computed by a machine.

For instance, consider the function $V : \text{Board} \to \mathbb{R}$, defined as follows:

- If b is a final board state and it is a win then $V(b) = 100$
- If b is a final board state and it is a loss then $V(b) = -100$
- If b is a final board state and it is a draw then $V(b) = 0$
- If b is not a final board state then $V(b) = V(b^*)$, where b^* is the best final board state that can be achieved starting from the board b playing the optimal moves

This function perfectly models our learning problem. However, it cannot be computed by any program since we haven't defined what an optimal set of moves is. We need a new definition that encodes this concept of optimal strategy for a checkers game.

For example, we can re-define V as follows:

$$V(b) = w_0 + w_1 \cdot \text{bp}(b) + w_2 \cdot \text{rp}(b) + w_3 \cdot \text{bk}(b) + w_4 \cdot \text{rk}(b) + w_5 \cdot \text{bt}(b) + w_6 \cdot \text{rt}(b)$$

where:

- $\text{bp}(b)$ is the number of black pieces
- $\text{rp}(b)$ is the number of red pieces
- $\text{bk}(b)$ is the number of black kings
- $\text{rk}(b)$ is the number of red kings
- $\text{bt}(b)$ is the number of red pieces threatened by black pieces
- $\text{rt}(b)$ is the number of black pieces threatened by red pieces

With this formulation, we have reduced the concept of leaning checkers to estimating the best possible values of the coefficients $w_1, \dots, w_6$, which care called **weights**, that maximize the value of $V(b)$ for any board state b. This estimation process is referred to as *learning the function V*.

Definition 2: Learned function

Given a target function $f : X \to Y$ with weights $w_1, \dots, w_k$, we define the **learned function** $\hat{f} : X \to Y$ as the current approximation of f computed by a learning algorithm.

By definition, the learned function $\hat{f}$ will never be equal to the target function f: the target function's weights are always unknown by definition. The idea here is to approximate f by repeatedly applying small changes to the weights $\hat{w_1}, \dots, \hat{w_k}$ of $\hat{f}$ in order to estimate the weights of f. To learn a function f, we need a *dataset*. A dataset is a set of instances that can be used by a learning algorithm to improve the performance of the learned function.

Definition 3: Dataset

Let $f : X \to Y$ be a target function and let $f_{\text{train}}(x)$ be the training value obtained by x in the training data. Given a set of n training inputs $X_D = \{x_1, \dots, x_n\}$, the **dataset** of the learning problem is the set of samples defined as:

$$D = \{(x_i, f_{\text{train}}(x_i)) \mid x_i \in X_D\}$$

After training, the learned function $\hat{f}$ will have learned the values of the inputs in the dataset, returning a value as close as possible to the one in the dataset (in some cases the returned value is exactly the same). However, we are interested in estimating the *other* possible inputs, i.e. those that aren't in the dataset.

In summary, a machine learning problem is the task of learning a target function $f : X \to Y$ through a dataset D for a set X_D of n inputs. To learn a function f means computing an approximating function $\hat{f}$ that returns values as close as possible to f, especially for values outside of the dataset D, implying that $\forall x \in X - X_D$ it should hold that $f(x) \approx \hat{f}(x)$. In order for the learned function to be *good*, the set of training inputs X_D must be very very small compared to the set of total inputs, meaning that $|X_D| \lll |X|$.

There are distinct types of machine learning problems based on:

- The type of dataset used:
  1. **Supervised learning**: problems where the model learns patterns from labeled data. Here, the dataset corresponds to $D = \{(x_y, y_i) \mid i \in X_D\}$, where y_i is the sample of the function value for x_i
  2. **Unsupervised learning**: problems where the model learns patterns from unlabeled data. Here, the dataset corresponds to $D = \{x_y \mid i \in X_D\}$
  3. **Reinforcement learning**: problems in which an agent learns to make decisions by interacting with an environment and receiving rewards or penalties based on its actions.
- The type of function to be learned:
  1. **Discrete Classification**: the input set is $X = A_1 \times \ldots \times A_n$, where A_i is a finite set, and the output set is $Y = \{c_1, \ldots, c_k\}$. When $k = 2$, i.e. we have only two classes, we say that the problem is a **Concept Learning** problem
  2. **Discrete Regression**: the input set is $X = A_1 \times \ldots \times A_n$, where A_i is a finite set, and the output set is $Y = \mathbb{R}^m$.
  3. **Continuous Classification**: the input set is $X = \mathbb{R}^n$ and the output set is $Y = \{c_1, \ldots, c_k\}$.
  4. **Continuous Regression**: the input set is $X = \mathbb{R}^n$ and the output set is $Y = \mathbb{R}^m$.

Classification problems are based on the classification of inputs into predetermined categories, while regression problems involve the approximation of functions defined over $\mathbb{R}$. Reinforcement learning, instead, is used for dynamic systems with unknown or partially known evolution model, usually robotic tasks and game playing.

1.2 Hypotheses, Consistency and Representation

After discussing the basic notation and terminology, we are ready to deepen our understanding on how to learn a problem. First, we define the notion of **hypothesis space**.

Definition 4: Hypothesis space

Given a target function f, an **hypothesis space** H for f is a set of functions $h \in H$, where h is called hypothesis, that can be learned in order to reach an approximation of f.

The representation of an hypothesis space highly depends on the type of problem. For instance, consider the problem of classifying natural numbers into primes numbers and composite numbers. This corresponds to a discrete classification problem with two classes $\mathbb{P}$ and $\mathbb{N} - \mathbb{P}$, i.e. a concept learning problem where we want to learn the concept of prime number. The target function of the problem is thus described as $f : \mathbb{N} \to \{\mathbb{P}, \mathbb{N} - \mathbb{P}\}$. Here, the simplest hypothesis space is the set $H = \{h_A : \mathbb{N} \to \{A, \mathbb{N} - A\} \mid A \subseteq \mathbb{N}\}$.

Given a performance measure P over a dataset D and an hypothesis space H, the learning task is to find the best approximation $h^* \in H$ of the function f using the dataset D.

$$h^* \in \arg\max_{h \in H} P(h, D)$$

Finding such optimal hypothesis is the core of a learning problem. However, by definition, an hypothesis space may also contain hypotheses that are clearly a wrong approximation of f over D. For instance, given a dataset D for a target function f, for any hypothesis $h \in H$ we can check wether $h(x) = f(x)$ only for instances $x \in X_D$ since the other values are unknown in D. This means that some hypothesis may have some values that are *inconsistent* with the dataset itself, making them useless in the learning process. An hypothesis $h \in H$ is said to be **consistent** with a dataset D of a target function f if and only if $h(x) = f(x)$ for all $x \in X_D$. The subset of hypothesis that are consistent with a dataset is called **version space**.

Definition 5: Version space

The **version space** of a target function f with respect to the hypothesis space H and the dataset D, written as $\mathrm{VS}_{H,D}$, is the subset of H that contains all the hypotheses that are consistent with D.

$$\mathrm{VS}_{H,D} = \{h \in H \mid \forall x \in X_D \;\; h(x) = f(x)\}$$

Consider again the previous example. Suppose that we're working with the following dataset $D = \{(1, \mathbb{N} - \mathbb{P}), (3, \mathbb{P}), (5, \mathbb{P}), (6, \mathbb{N} - \mathbb{P}), (7, \mathbb{P}), (10, \mathbb{N} - \mathbb{P})\}$. Here, the version space would restrict our interest to all those functions $h_A : \mathbb{N} \to \{A, \mathbb{N} - A\}$ where the elements $3, 5, 7$ lie inside A and the elements $1, 6, 10$ lie outside of A.

$$VS_{H,D} = \{h_A : \mathbb{N} \to \{A, \mathbb{N} - A\} \mid \{3,5,7\} \subseteq A \subseteq \mathbb{N} \text{ and } \{1,6,10\} \subseteq \mathbb{N} - A\}$$

By definition, the best approximation for a performance measure P over a dataset D and an hypothesis space H must clearly lie inside the version space $VS_{H,D}$. Hence, we can restrict our interest to the version space itself.

$$h^* \in \arg\max_{h \in VS_{H,D}} P(h, D)$$

The concept of version space is based on the **inductive learning hypothesis**: any hypothesis that is consistent with the target function over a dataset of *adequate size* will also approximate the target function well over other unobserved examples. In other words, if we consider a dataset of adequate size then every hypotheses inside the version space will be a nice approximation of the target function.

The simplest way to compute the version space is through the **List-Then-Eliminate** algorithm, a bruteforce algorithm that enumerates all the hypothesis space and then test the consistency of each hypothesis, discarding the invalid ones. Even though this algorithm is correct, it is also clearly *infeasible* since enumerating all the different hypothesis would require an exponential amount of time. We'll see improved ways to compute the version space given by a dataset.

Algorithm 1: List-Then-Eliminate

Given an hypothesis space H and a dataset D, the algorithm returns $VS_{H,D}$

```
function listThenEliminate($H$, $D$)
    $VS_{H,D} := H$
    for $(x, f(x)) \in D$ do
        $H' := \{h \in H \mid h(x) \neq f(x)\}$
        $VS_{H,D} = VS_{H,D} - H'$
    end for
    return $VS_{H,D}$
end function
```

In order for the inductive learning hypothesis to hold, the size of the dataset is a **critical** factor: if the hypothesis space is too *powerful* and the search is complete, then the system won't be able to classify new instances, meaning that we have no generalization power.

For instance, consider a generic concept learning problem described by the target function $c : X \to \{0, 1\}$. Let D be the chosen dataset. For any hypothesis space H, it's easy to see that H is actually *associated* with a particular set of instances, that being all instances that are classified as positive by such hypothesis. In fact, we have a mapping ϕ between the hypothesis space H and the power set $\mathcal{P}(X)$.

$$\phi_H : H \to \mathcal{P}(X) : h_A \mapsto A = \{x \in X \mid h_A(x) = 1\}$$

In general, this mapping is not surjective, meaning that there is a subset A of $\mathcal{P}(X)$ that is is not covered by an hypothesis inside H. In fact, we prefer cases where such mapping is not surjective. This is known as the **hypothesis space representation issue**: some hypothesis spaces may be useless even when we restrict our interest to the version space.

For example, Suppose that there is an hypothesis spaces H_1, H_2 such that H_1 cannot represent $\mathcal{P}(X)$, i.e. ϕ_{H_1} is not surjective, while H_2 can:

- In $VS_{H_1,D}$ we have that $\exists x \in X - X_D$ for which there are two hypotheses $h, h' \in \mathrm{VS}_{H_1,D}$ such that $h(x) = 0$ and $h'(x) = 1$.
- In $VS_{H_2,D}$ we have that $\forall x \in X - X_D$ there are two hypotheses $h, h' \in \mathrm{VS}_{H_2,D}$ such that $h(x) = 0$ and $h'(x) = 1$.

The small difference in the quantifier has enormous impacts on the usefulness of these two spaces. If we use H_1 then we expect that the approximation found by any algorithm will give the wrong value for some unlabeled inputs $x \in X - X_D$ due to the presence of two functions that can be chosen for x. If we use H_2, instead, we expect that every unlabeled input will have a wrong value.

These observations imply that the *more information* the hypothesis space encapsulates about the values in X_D, the *harder* it becomes to **generalize** and predict values for samples outside X_D. In other words, a more expressive hypothesis space can **overfit** to the data, making it more difficult to make accurate predictions on unsampled data. We'll return on the problem of overfitting the dataset in following sections.

The process of reducing the *representation power* of the hypothesis space in favor of *generalization power* – as in reducing the hypothesis space from H' to H in the previous example – is called **language bias**. Ideally, we want our hypothesis space to be as good as possible. Clearly, the best possible hypothesis space contains only the optimal approximating function $h^* \in H$. In this case, the previous observations regarding the hypothesis space representation issue are solved: every unlabeled data will have only one value inside the function. The process of selecting one particular hypothesis among the set of possible ones – i.e., choosing $h^* \in H$ – is called **search bias**.

In machine learning, the concept of learning bias is crucial for improving a model's ability to generalize. A good learning bias helps guide the learning algorithm towards patterns in the data that are useful for predicting unseen samples, increasing the system's generalization power. This bias allows the model to make accurate predictions on new data that wasn't part of the training set. Without such a bias, a system would simply **memorize** the dataset, failing to predict values for samples outside the training set, rendering it *ineffective* in real-world applications. Systems lacking generalization capabilities would be of little use, as they wouldn't be able to provide meaningful predictions beyond the data they were trained on.

In real-world applications, datasets often contain **noise**, which refers to irrelevant or erroneous information that can distort the true underlying patterns in the data. Noise can come from a variety of sources, including measurement errors, data entry mistakes, incomplete data or random fluctuations in the system being studied. This noise complicates the *learning process*, as machine learning models may *struggle* to distinguish between true *signal* and *noise*.

A noisy data-point in a dataset D for a function f can be formulated as a pair $(x_i, y_i) \in D$, where $y_i \neq f(x_i)$. This means that there may be *no consistent hypothesis* with noisy data, i.e. $\mathrm{VS}_{H,D} = \varnothing$. In these scenarios, **statistical methods** must be employed to implement robust algorithms, in order to reduce the noise in the data.

1.3 Performance evaluation

After discussing which hypotheses we're interested in, we'll now focus on evaluating the performance of an hypothesis. To give an intuition, we'll focus on classification problems. Let $f : X \to Y$ be a classification problem. Let $\mathcal{D}$ be a probability distribution over X, meaning that each element in X has an associated probability of being drawn. The sum of all the probabilities in $\mathcal{D}$ must be 1. Let S be a set of n samples drawn from X according to $\mathcal{D}$ and for which we know the value $f(x)$. By definition, S corresponds to a dataset of n random elements. For any possible hypothesis h returned by a learning algorithm obtained through S, we want to know what is the best estimate of the *accuracy* of h over future instances drawn from the same distribution and what is the *probable error* of this accuracy estimate. First, we have to define two error measures.

Definition 6: True error

Let $f : X \to Y$ be a classification problem. Given an hypothesis h, the **true error** of h with respect to f and a probability distribution $\mathcal{D}$ over X, written as $\mathrm{error}_{\mathcal{D}}(h)$, is defined as:

$$\mathrm{error}_{\mathcal{D}}(h) = \Pr_{x \in \mathcal{D}}[h(x) \neq f(x)]$$

The *true accuracy* of h is defined as $\mathrm{accuracy}_{\mathcal{D}}(h) = 1 - \mathrm{error}_{\mathcal{D}}(h)$.

The true error of an hypothesis is clearly a perfect error measure. However, it's easy to see that computing the true error of h is infeasible since we would have to compute such probability over all the inputs. To fix this, we consider another type of error measure that is less precise but computable.

Definition 7: Sample error

Let $f : X \to Y$ be a classification problem. Given an hypothesis h, the **sample error** of h with respect to f and a data sample S over $\mathcal{D}$, written as $\mathrm{error}_S(h)$, is defined as:

$$\mathrm{error}_S(h) = \frac{1}{n} \sum_{x \in S} \delta(x)$$

where $\delta(x) = 1$ if $h(x) \neq f(x)$ and $\delta(x) = 0$ otherwise. The *sample accuracy* is defined as $\mathrm{accuracy}_S(h) = 1 - \mathrm{error}_S(h)$.

The sample error can be efficiently computed over a small data sample. The goal of a learning system is to be accurate in $h(x)$. However, sample accuracy may often fool us into thinking that our hypothesis is good: if $\text{accuracy}_S(h)$ is very high but but $\text{accuracy}_{\mathcal{D}}(h)$ is poor, our system is not be very useful.

We also notice that, by definition, $\text{error}_S(h)$ is actually a random variable depending on S. Sampling two different sets S and S' from $\mathcal{D}$ may different values for $\text{error}_S(h)$ and $\text{error}_{S'}(h)$. For this reason, we're also interested in the expected value $\mathbb{E}_{S \text{ over } \mathcal{D}}[\text{error}_S(h)]$ of the sample error, i.e. the weighted average over all the possible samples S. The **estimation bias** for $\mathcal{D}$ is defined as the difference between the expected sample error and the true error.

$$\text{bias}_{\mathcal{D}} = \mathop{\mathbb{E}}_{S \text{ over } \mathcal{D}}[\text{error}_S(h)] - \text{error}_{\mathcal{D}}$$

We notice that the estimation bias is equal to 0 if and only if $\mathbb{E}_{S \text{ over } \mathcal{D}}[\text{error}_S(h)] = \text{error}_{\mathcal{D}}$. However, for any learning algorithm it is *impossible* to prove that the bias is exactly 0. This derives from the very concept of learning function. Hence, in order for an hypothesis to be as good as possible, we want an estimation bias that is as close as possible to 0.

If S is the training set used to compute h through some learning algorithm then $\text{error}_{\mathrm{S}}(h)$ will always be have some bias. In order to get an unbiased estimate, The training set S used to compute h and the evaluation set S' must be chosen independently. However, even with an unbiased sample, $\text{error}_{S'}(h)$ may still vary from $\text{error}_{\mathcal{D}}(h)$ – the smaller the set S', the greater the expected variance.

But how good is an estimate of $\text{error}_{\mathcal{D}}(h)$ provided by $\text{errors}_D(h)$? From the theory of statistical analysis, we can derive the concept of **confidence interval**. If S contains n samples drawn independently of one another over $\mathcal{D}$, S is independent of h and $n \geq 30$ then with approximately $N\%$ of probability we have that:

$$|\text{error}_S(h) - \text{error}_{\mathcal{D}}(h)| \leq z_N \sqrt{\frac{\text{error}_S(h)(1 - \text{error}_S(h))}{n}}$$

where the value z_n is a constant given by the *confidence level* $N\%$.

$N\%$	50%	68%	80%	90%	95%	98%	99%
z_n	0.67	1.00	1.28	1.64	1.96	2.33	2.58

Figure 1.1: Relation between each confidence level and its constant

This result shows that the sample error (when unbiased) is indeed a good estimation measure for the true error, up to some confidence level. However, in order to get a good approximation we must make some trade offs between *training* and *testing*:

1. Using more samples for training and less for testing gives a better approximating function h, but $\text{error}_S(h)$ may not be a good approximation of $\text{error}_D(h)$.
2. Using less samples for training and more for testing gives a worse approximating function h', but $\text{error}_S(h')$ is guaranteed to be a good approximation of $\text{error}_D(h)$.

Usually, a good trade off for medium sized datasets is a training set of size $\frac{2}{3}|X|$ and a testing set of size $\frac{1}{3}|X|$. However, computing $\text{error}_S(h)$ is not enough: the estimation bias is defined through $\mathbb{E}_{S \text{ over } \mathcal{D}}[\text{error}_S(h)]$. Computing the expected sample error is a task as hard as computing the true error. Nonetheless, thanks to the *central limit theorem*, the expected sample error can be easily approximated by averaging the computed values of $\text{error}_{S_1}(h), \ldots, \text{error}_{S_k}$, where $S_1, \ldots, S_k$ are independent from each other.

$$\lim_{k \to +\infty} \frac{1}{k} \sum_{i \in [k]} \text{error}_{S_i}(h) - \mathop{\mathbb{E}}_{S \text{ over } \mathcal{D}}[\text{error}_S(h)] = 0 \implies \frac{1}{k} \sum_{i \in [k]} \text{error}_{S_i}(h) \approx \mathop{\mathbb{E}}_{S \text{ over } \mathcal{D}}[\text{error}_S(h)]$$

With all the tools that we have discussed, we're now ready to give the first unbiased estimator: the **K-Fold Cross Validation** algorithm.

Algorithm 2: K-Fold Cross Validation

Given a dataset D, a learning algorithm L for a function f and a value $k > 0$, the K-Fold Cross Validation returns an estimation $\text{error}_{L,D}$ of the expected sample error of L over D.

```
function kFoldCrossValidation($D, L, k$)
    Partition $D$ into $k$ sets $S_1, \ldots, S_k$ where $|S_i| > 30$
    for $i = 1, \ldots, k$ do
        $T_i = D - S_i$                                  ▷ $T_i$ is the training set
        $h_i = L(T_i)$
    end for
    return $\frac{1}{k} \sum_{i \in [k]} \text{error}_{S_i}(h_i)$
end function
```