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 13, 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 $w_1, \ldots, w_6$, 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 $w_1, \ldots, w_k$, we define the **learned function** $\widehat{f} : X \to Y$ as the current approximation of f computed by a learning algorithm.

By definition, the learned function $\widehat{f}$ 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 $\widehat{w_1}, \ldots, \widehat{w_k}$ of $\widehat{f}$ 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 $X_D = \{x_1, \ldots, x_n\}$, 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 $\widehat{f}$ 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 $\widehat{f}$ 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 $f(x) \approx \widehat{f}(x)$. 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)) ∈ D do
        H' := {h ∈ H | h(x) ≠ f(x)}
        VS_{H,D} = VS_{H,D} − H'
    end for
    return VS_{H,D}
end function
```

1.3 Representation power and generalization power

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 $h, h' \in \text{VS}_{H_1,D}$ 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 $h, h' \in \text{VS}_{H_2,D}$ 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. $\text{VS}_{H,D} = \varnothing$. In these scenarios, **statistical methods** must be employed to implement robust algorithms, in order to reduce the noise in the data.

1.4 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 $\text{error}_{\mathcal{D}}(h)$, is defined as:

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

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

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 $\text{error}_S(h)$, is defined as:

$$\text{error}_{\mathcal{S}}(h) = \frac{1}{n} \sum_{x \in S} \delta(x)$$

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

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 $\mathbb{E}_{S \subseteq \mathcal{D}}[\text{error}_S(h)]$ 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.

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

We notice that the estimation bias is equal to 0 if and only if $\mathbb{E}_{S \subseteq \mathcal{D}}[\text{error}_S(h)] = \text{error}_{\mathcal{D}}$. 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 $\text{error}_S(h)$ 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 a single $\text{errors}_S(h)$? 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 a single sample error (when unbiased) is also 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 $\mathbb{E}_{S\subseteq\mathcal{D}}[\text{error}_S(h)]$. 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.

$$\lim_{k\to+\infty} \overline{\varepsilon}_k - \mathop{\mathbb{E}}_{S\subseteq\mathcal{D}}[\text{error}_S(h)] = 0 \implies \lim_{k\to+\infty} \overline{\varepsilon}_k = \mathop{\mathbb{E}}_{S\subseteq\mathcal{D}}[\text{error}_S(h)]$$

where $\overline{\varepsilon}_k = \frac{1}{k}\sum_{i\in[k]} \text{error}_{S_i}(h)$. With all the tools that we have discussed, we're now ready to discuss our first unbiased estimator: the **K-Fold Cross Validation** algorithm.

Algorithm 2: K-Fold Cross Validation for Sample Error

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
    $h_i = L(T_i)$
    $\delta_i = \text{error}_{S_i}(h_i)$
  end for
  return $\frac{1}{k}\sum_{i\in[k]} \delta_i$
end function

1.5 Hypothesis comparison

Consider the case where we have two independent hypotheses h_1 and h_2 for some discrete-valued target function. Hypothesis h_1 has been tested on a sample S_1 and h_2 has been tested on an sample S_2. S_1 and S_2 are independent from each other and they are drawn from the same distribution $\mathcal{D}$. The difference d between the true errors of these two hypotheses is given by

$$d = \text{error}_{\mathcal{D}}(h_1) - \text{error}_{\mathcal{D}}(h_2)$$

The obvious choice for the estimator $\widehat{d}$ of d is given by

$$\widehat{d} = \text{error}_{S_1}(h_1) - \text{error}_{S_2}(h_2)$$

In fact, it's very easy to prove that this is indeed an unbiased estimator.

$$\mathop{\mathbb{E}}_{S \subseteq \mathcal{D}}[\text{error}_S(h_1) - \text{error}_S(h_2)] = \mathop{\mathbb{E}}_{S \subseteq \mathcal{D}}[\text{error}_S(h_1)] - \mathop{\mathbb{E}}_{S \subseteq \mathcal{D}}[\text{error}_S(h)] = d$$

This definition of error comparison allows us to mathematically describe the concept of **overfitting**. In previous sections, we described this phenomenon as the property of an hypothesis of being too much expressive, i.e. it performs pretty good on the training data but it makes poor predictions on unsampled data.

Definition 8: Overfitting

Let $h \in H$ be an hypothesis for a function f obtained through the training data S, where H is the chosen hypothesis space. We say that h **overfits** the set S if there is an another hypothesis $h' \in H$ obtainable through S that has a sample error higher than h but a true error lower than h.

$$\text{error}_S(h) < \text{error}_S(h') \qquad \text{error}_{\mathcal{D}}(h) > \text{error}_{\mathcal{D}}(h')$$

Suppose that we have two learning algorithms L_A and L_B. We want to determine which of them produces the best approximation of f on average. The K-Fold Cross Validation algorithm that we discussed in the previous section can be easily adapted to compare L_A and L_B.

Algorithm 3: K-Fold Cross Validation for Comparison

Given a dataset D, two learning algorithms L_A, L_B for a function f and a value $k > 0$, if the algorithm returns True then we expect that L_A is better than L_B

function kFoldCrossComparison($D, L_A, L_B k$)
  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
    $h_i = L_A(T_i)$
    $h'_i = L_B(T_i)$
    $\delta_i = \text{error}_{S_i}(h_i) - \text{error}_{S_i}(h'_i)$
  end for
  $\varepsilon = \frac{1}{k} \sum_{i \in [k]} \delta_i$
  return $\varepsilon < 0$
end function

1.6 Performance metrics

Until now, we have focused on error and accuracy. Intuitively, we expect that if the accuracy is high, i.e. the error is low, then our hypothesis must be a good approximation. However, this is not always the case.

For instance, consider a concept learning problem $f : X \to \{+, -\}$ with a training set where 90% of the samples are negative, meaning that they are labeled with $-$. Consider the hypothesis h that labels every single element $x \in X_D$ as negative. Even if this hypothesis is clearly bad, the accuracy would still be 90%. Of course, we're usually interested in the accuracy of approximations yield by learning models. In some cases, accuracy only is not enough to assess the performance of a classification method.

In a more statistical sense, error and accuracy can be also described through the concept of positives and negatives:

- A **true positive** is a positive element that gets classified as positive
- A **true negative** is a negative element that gets classified as negative
- A **false positive** is a negative element that gets classified as positive
- A **false negative** is a positive element that gets classified as negative

Here, the error rate is described as the ratio between all the errors (false positives and false negatives) and all the samples:

$$\text{error} = \frac{\text{FP} + \text{FN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

while the accuracy is still defined as the complement of the error.

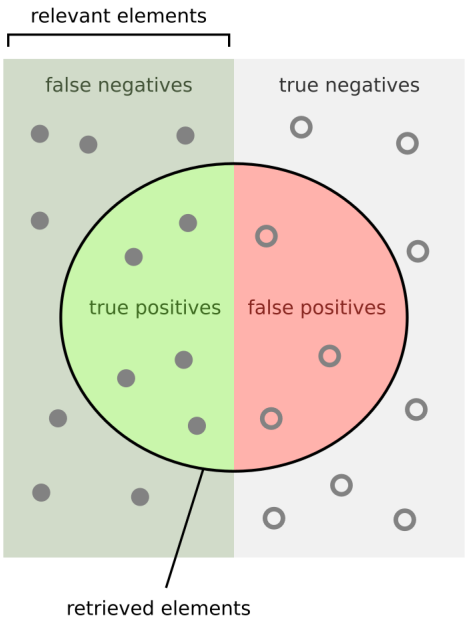

Figure 1.2: The relevant elements are the elements that are really positive, while the retrieved elements are the ones that have been labeled as positive

As discussed in the example above, these two measures may be inappropriate when the datasets are unbalanced. For this reason, we also consider two additional measures. The first measure is the **recall** (also called *true positive rate*), defined as the ratio between the true positives and the relevant elements. The recall measures the ability of the model to avoid false negatives.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

The second measure is the **precision** (also called *true positive rate*), defined as the ratio between the true positives and the retrieved elements. The recall measures the ability of the model to avoid false positives.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

The recall and the precision are usually used to compute the **F-score**, which measures the predictive performance of a model. More formally, the F_1-score is the harmonic mean of the precision and recall. Thus, it symmetrically represents both precision and recall in one single metric.

$$\text{F}_1 = \frac{2 \cdot \text{recall} \cdot \text{precision}}{\text{recall} + \text{precision}}$$

Another very easy to read measure linked to classification errors is the **confusion matrix**. A confusion matrix reports the percentage of instances of class C_i that have been classified in the class C_j. When the confusion matrix produced by the analysis of the performance of a solution retrieved through an algorithm is "accumulated" towards the central diagonal, the solution is a good model for the problem. Otherwise, the model is considered bad.

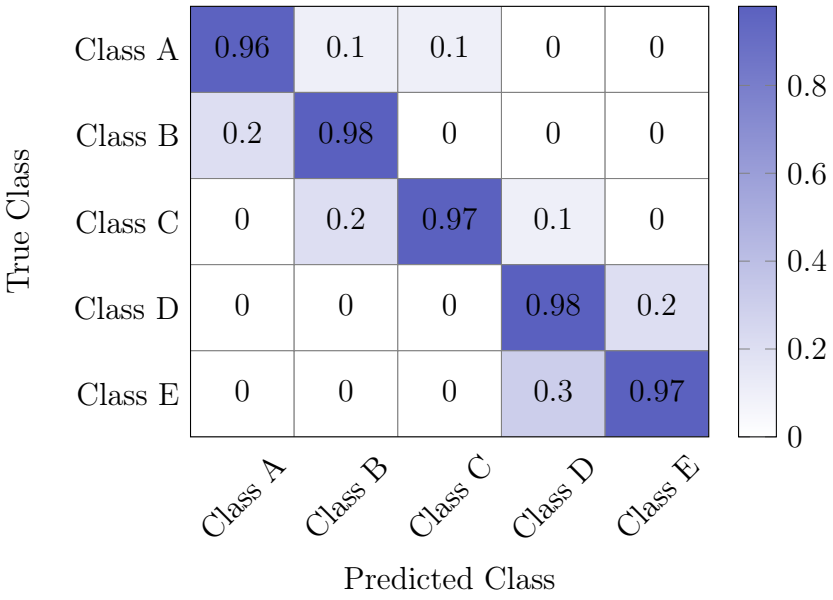

Figure 1.3: Example of a good confusion matrix

For regression problems $f : X \to \mathbb{R}^k$, we need a different type of metrics. In particular, given a test set $S = \{(x_i, t_i)\}_{i \in [n]}$, performance can be measured in various ways:

- **Mean Absolute Error (MAE)**

$$\frac{1}{n} \sum_{i=1}^{n} \left| \widehat{f}(x_i) - t_i \right|$$

- **Mean Squared Error (MSE)**

$$\frac{1}{n} \sum_{i=1}^{n} (\widehat{f}(x_i) - t_i)^2$$

- **Root Mean Squared Error (RMSE)**

$$\sqrt{\frac{1}{n} \sum_{i=1}^{n} (\widehat{f}(x_i) - t_i)^2}$$

The K-Fold Cross Validation algorithm can also be adapted to compute these performance measures.

Algorithm 4: K-Fold Cross Validation for MAE

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 $\text{MAE}_{L,D}$ of the expected mean squared 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
    $h_i = L(T_i)$
    $\delta_i = \text{MAE}_{S_i}(h_i)$
  end for
  return $\frac{1}{k} \sum_{i \in [k]} \delta_i$
end function

2

Decision Tree Learning

2.1 Decision trees

After discussing the general idea behind machine learning problems and how to evaluate the solution retrieved by an algorithm, we're now ready to focus on how these algorithms work. In particular, we'll start by discussing decision trees.

Given a discrete input space described by m attributes $X = A_1 \times \ldots \times A_m$, where each A_i is a finite set, and a problem $f : X \to Y$, a **decision tree** is a n-ary tree where:

- Each internal node is labeled by an attribute A_i
- Each branch outgoing from a node labeled with A_i denotes a possible value of an attribute $v \in \text{Values}(A_i)$, where the latter is the set of possible values for A_i
- Each leaf node of the tree is labeled with a class $j \in Y$

Given an input, a decision tree computes by querying the internal nodes (starting from the root) and always proceeds on the edge corresponding to the attribute value assumed by the input. In other words, a decision tree is nothing more than a set of rules that classifies any given input. Decision trees are a very easy computational model and they are capable of computing any discrete function. For example, suppose that we want to solve the concept learning problem $\text{PlayTennis} : X \to \{\text{Yes}, \text{No}\}$ relative to deciding if good conditions are met in order to play tennis. The input set that we'll be working with is the following:

$$X = \{\text{Outlook} \times \text{Temperature} \times \text{Humidity} \times \text{Wind}\}$$

where:

$$\text{Outlook} = \{\text{Sunny}, \text{Overcast}, \text{Rain}\} \qquad \text{Temperature} = \{\text{Hot}, \text{Mild}, \text{Cold}\}$$

$$\text{Humidity} = \{\text{Normal}, \text{High}\} \qquad \text{Wind} = \{\text{Weak}, \text{Strong}\}$$

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Figure 2.1: Example dataset for *PlayTennis*

A decision tree for this problem would look like the following:

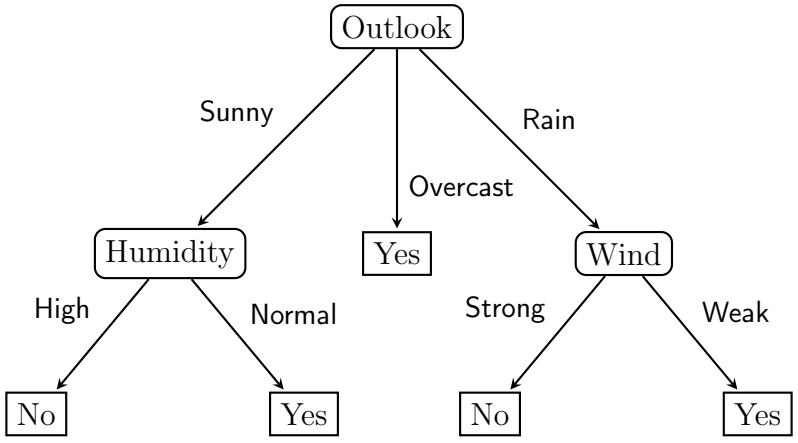

Figure 2.2: Example of decision tree for *PlayTennis*.

In concept learning problems, each decision tree can be described as a disjunction of conjunction of the attributes tested by the tree. In particular, we consider only paths that end up with a positive leaf. For instance, the tree described above can also be represented as following:

$$(\text{Outlook} = \text{Sunny} \wedge \text{Humidity} = \text{Normal})$$

$$\vee(\text{Outlook} = \text{Overcast})\vee$$

$$(\text{Outlook} = \text{Rain} \wedge \text{Wind} = \text{Weak})$$

2.2 Entropy and the ID3 algorithm

In this context, the hypothesis space is the *set of all possible decision trees*. We'll now define the algorithm that we'll be using to derive an optimal decision tree from a given dataset: the **Iterative Dichotomiser 3 (ID3)** algorithm.

Algorithm 5: ID3 algorithm

Given a dataset D for a concept learning problem $f : X \to \{+, -\}$, an attribute list L containing the attributes of X and a target attribute A, the algorithm returns a decision tree for f.

function ID3(D, L, A)
    Initialize an empty decision tree T
    Create a root node in T
    if All the examples in D are positive **then**
        Label the root with $+$
    else if All the examples in D are negative **then**
        Label the root with $-$
    else if $L = \varnothing$ **then**
        Label the root with the most common value for A in D
    else
        Let A be the *best decision attribute* in L for D
        Label the root with A
        for $v \in$ Values(A) **do**
            Let D_a be the subset of D whose tuples have A set to v
            if $D_a = \varnothing$ **then**
                Create a leaf node labeled with the most common value for A in D
                Add an edge labeled with $A = v$ from the root to the leaf node
            else
                Compute the subtree $T_A =$ ID3($D_a, A, L - \{A\}$)
                Add an edge labeled with $A = v$ from the root to T_A
            end if
        end for
    end if
    Return T
end function

In the ID3 algorithm described above, the *best decision attribute* is an attribute whose optimality depends on a pre-defined criteria. Based on the criteria chosen, different attributes may be selected. Trying to define an optimal criteria is crucial. In fact, based on the chosen attribute order, we may get a completely different decision tree.

The commonly used measure for this task is the **information gain**, which measures how well a given attribute *separates* the training examples according to their target classification. The information gain of an attribute is measured as reduction in **entropy**, which measures the impurity of a sample.

Definition 9: Entropy

Let S be a sample set for a classification problem $f : X \to Y$. For each $i \in Y$, we denote with p_i the proportion of elements of S that are classified as y_i. The **entropy** of S is defined as:

$$\text{Entropy}(S) = -\sum_{i \in Y} p_i \log_2 p_i$$

Note: we assume that $0 \log_2 0 = 0$

In the special case of boolean classification problems – since we have only two classes – we denote with $p_\oplus$ and $p_\ominus$ be the proportions of positive and negative samples in S ($p_\ominus = 1 - p_\oplus$). In this case, the entropy of S is defined as:

$$\text{Entropy} = -p_\oplus \log_2 p_\oplus - p_\ominus \log_2 p_\ominus$$

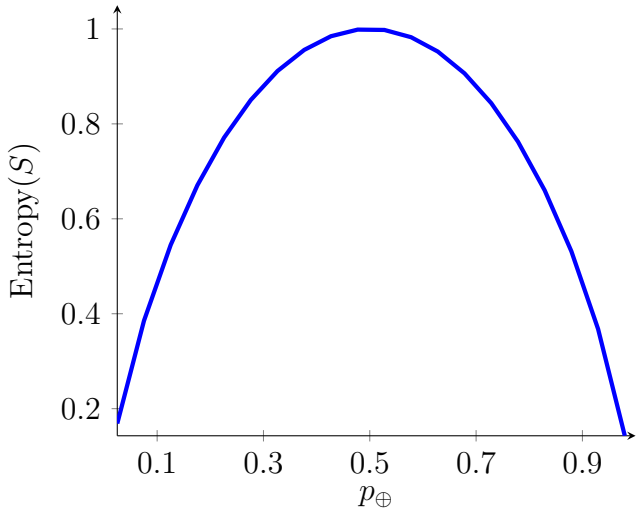

Figure 2.3: The entropy function relative to a boolean classification

Through the graph above, it's easy to see that in boolean classification problems the entropy reaches its maximum value when $p_\oplus = 0.5$, i.e. when the two classes are perfectly balanced, while it reaches its minimum when $p_\oplus = 0$ or $p_\oplus = 1$.

Definition 10: Information gain

The **information gain** for a sample set S is defined as the expected reduction in entropy of S caused by the selection of a value for the attribute A.

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{S} \text{Entropy}(S_v)$$

where $S_v = \{s \in S \mid A(s) = v\}$

Suppose that we have a sample set $S = [9+, 5-]$ (where this notation implies that we have 9 positives and 5 negatives) and that we are testing the information gain of the attribute Wind $= \{\text{Weak}, \text{Strong}\}$. First, we compute the entropy of S:

$$\text{Entropy}(S) = -\frac{9}{14}\log_2\left(\frac{9}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) \approx 0.940$$

After looking at the sample set, we get that $S_{\text{Weak}} = [6+, 2-]$ and $S_{\text{Strong}} = [3+, 3-]$. The entropy of these two subsets corresponds to:

$$\text{Entropy}(S_{\text{Weak}}) = -\frac{6}{8}\log_2\left(\frac{6}{8}\right) - \frac{2}{8}\log_2\left(\frac{2}{8}\right) \approx 0.811$$

$$\text{Entropy}(S_{\text{Strong}}) = -\frac{3}{6}\log_2\left(\frac{3}{6}\right) - \frac{3}{6}\log_2\left(\frac{3}{6}\right) = 1$$

Hence, the information gain on S knowing the attribute Wind corresponds to:

$$\text{Gain}(S, \text{Wind}) = \text{Entropy}(S) - \frac{8}{14}\text{Entropy}(S_{\text{Weak}}) - \frac{6}{14}\text{Entropy}(S_{\text{Strng}}) \approx 0.048$$

Each time the ID3 algorithm has to select the best attribute, it always chooses the one with the highest information gain, corresponding to the one that most reduces the entropy. Eventually, the entropy of the "implicitely partitioned" dataset will reach 1 or 0, meaning that the associated portion of the dataset contains only positive or negative values.

Consider again the dataset S shown in Figure 2.1. This dataset contains 9 positive entries and 5 negative entries. First, the ID3 algorithm computes the information gain of all of the four attributes, finding that Outlook is the attribute that yields the highest gain.

Outlook	Temperature	Humidity	Wind
0.246	0.151	0.048	0.029

Figure 2.4: Gain on S for each attribute

After setting Outlook as the root node, the algorithm now computes the three subtrees, one for each possible value of Outlook. Since the dataset S_{Overcast} contains only positive samples, the leaf node is labeled with Yes. The two other branches, instead, have to recursively compute their subtrees using information gain.

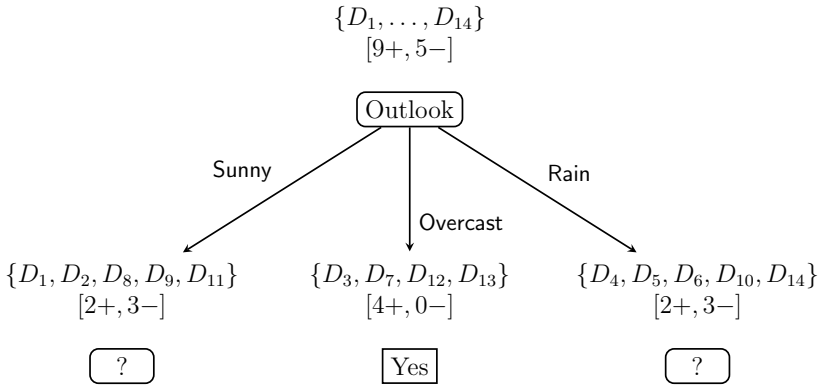

Figure 2.5: The partially learned decision tree after the first level of recursion

After repeating the process for each level of recursion, we get the decision tree shown in Figure 2.2. On each level of recursion, even though such hypothesis space is complete, meaning that every target function lies inside, the ID3 algorithm moves through such space in a greedy manner, returning only a **local minima**, while the target function is a *global minima*. Moreover, each step of the algorithm requires to analyze all the training examples, making this approach **not incremental** – if we want to add more data, the whole tree must be recomputed. However, this also ensures that our statistically-based search choices are robust to noisy data.

2.3 Overfitting in decision trees

A common issue in decision tree learning is the **size** of the decision tree yield by the algorithm, that is the number of nodes int the output tree on average. The importance of size comes from the nature of the algorithm itself: since the hypothesis space is complete, if allow the tree to have an huge number of nodes then it will eventually become a perfect approximation of the dataset.

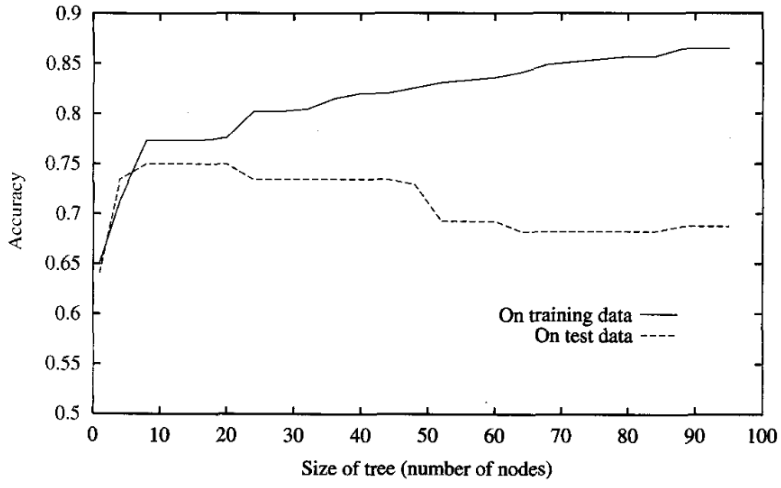

Figure 2.6: Overfitting in decision trees

In the worst case, each entry of the dataset is a leaf of the decision tree. When this happens, the solution found by the algorithm is clearly overfitting the data. In decision tree learning, overfitting is unavoidable. However, it can be reduced through some techniques. The first common technique involves enforcing a maximum growth on the tree by avoiding the recursive process when splitting the data wouldn't give a statistically significant improvement. The second technique grows the full decision tree and then prunes nodes that aren't significant. We'll focus on the second technique.

To determine the optimal tree size, we use a separate set of examples (distinct from the training examples) to evaluate the utility of post-pruning, then apply a statistical test to estimate accuracy of a tree on the entire data distribution.

In **reduced-error pruning**, the data is split into a training and a validation set.The easiest way to achieve each pruning step is to greedily select the subtree that is cut.

1. We copy the decision tree and randomly select a subtree
2. We replace the whole subtree it with a single leaf node labeled with the most common leaf label in the subtree.
3. We test the accuracy on the new subtree through the validation set

After evaluating the accuracy of every possible cut, we choose the one that increases the accuracy the most. Eventually, we'll reach a point where any additional cut will decrease the accuracy, concluding the pruning procedure.

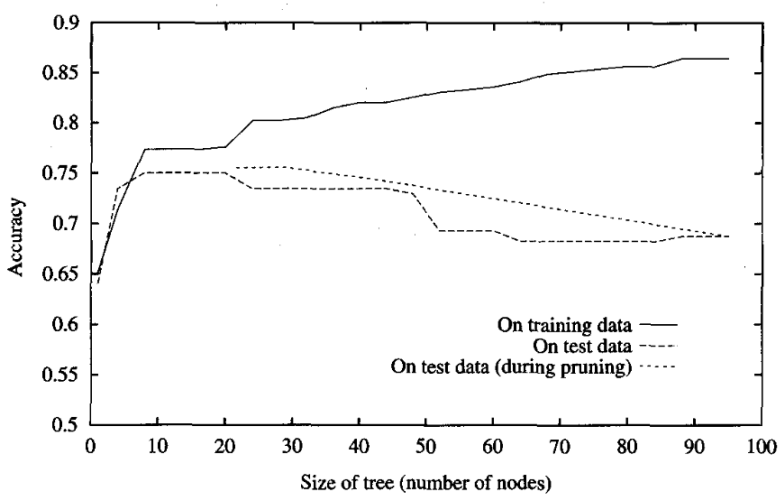

Figure 2.7: Performance after reduced-error pruning.

One more successful method for finding high accuracy hypotheses is the **rule post-pruning** technique. Variants of this technique are used by C4.5, another decision tree learning algorithm.

1. We first infer the decision tree allowing it to overfit the dataset
2. We convert the decision tree into an equivalent set of rules as shown in previous sections

3. We generalize each rule as much as possible independently of others
4. We remove multiple instances of the same generalized rule
5. We sort the final set of rules into a desired sequence
6. We convert the sequence back to a decision tree

Decision trees can also be used for classification of continuous-valued attributes. For instance, given the continuous-valued attribute Temperature, we can create a boolean variable that works with this attribute, such as "Temperature > 72.3". To work with such variables, the best approach is to pre-define the various splits of the set of continuous values.

Moreover, decision trees can also use multi-valued attributes, such as dates. However, the information gain of these variables is usually too high. To relax this issue, one approach is to use the **Gain Ratio** instead of the gain.

$$\text{GainRatio}(S, A) = \frac{\text{Gain}(S, A)}{\text{SplitInformation}(S, A)}$$

where $\text{SplitInformation}(S, A) = -\sum_{i \in Y} \frac{|S_i|}{|S|} \log_2 \left(\frac{|S_i|}{|S|} \right)$ and $S_v = \{s \in S \mid A(s) = v\}$

In some instances, we may want to give more importance to some particular attributes. To do so, we can define the **cost** for each attribute and replace the gain with one of the following values (they have no standard name)

$$\frac{\text{Gain}^2(S, A)}{\text{Cost}(A)} \qquad \frac{2^{\text{Gain}(S,A)} - 1}{(\text{Cost}(A) + 1)^w} \text{ for some } w \in [0, 1]$$

If some examples in the dataset have no value for the attribute selected by ID3, we can still use it through one of the following methods:

1. If node u tests A, assign most common value of A among other examples sorted to node u
2. Assign most common value of A among other examples with same target value
3. Assign a probability p_i to each possible value $v_i \in \text{Value}(A)$, assigning fractions of p_i to each descendant in tree

3

Bayesian learning

3.1 Uncertainty and probability

When the problem that we want to learn is based on actions that may assume continuous values, defining a sequence discrete decision becomes hard. For instance, consider the action A_t representing that Alice leaves for the airport t minutes before her flight. In order to know which t will satisfy the problem, we have to sort out many sub-problems, such as partial observability (road state, other drivers' plans, ...), noisy sensors (traffic reports), uncertainty in action outcomes (flat tire, ...), complexity of modelling and predicting traffic, ...

Hence an approach purely based on logical deduction will either risk falsehood, leads to conclusions that are too weak for decision making due to them requiring way too many conditions to be met ("A_{25} get me there on time if there's no accident on the bridge and it doesn't rain and my tires remain intact ...") or may lead to non-optimal decisions that ensure the outcome ("A_{1440} will surely suffice, but I have to stay overnight in the airport"). When this is the case, the best option is to just accept **uncertainties** and find a way to work with them. In particular, this implies using a probabilistic approach.

Definition 11: Sample space and probability space

A **sample space** Ω is a set of finite or infinite outcomes. The elements $\omega \in \Omega$ are usually called *atomic event* or *outcome of a random process.*

A **probability space** is a function $P : \Omega \to [0, 1]$ defined on a sample space where the sum of all the values equals 1

$$\sum_{\omega \in \Omega} P(\omega) = 1$$

Given a sample space Ω, a probability space associates a probability in the range $[0, 1]$ to each element of Ω. Suppose that we want to model a probability space that represents

the outcomes of a dice roll. The sample space can be defined as:

$$\Omega = \{1, 2, 3, 4, 5, 6\}$$

If we're using a standard dice, the probability space will be defined as:

$$P(\omega) = \left\langle \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6} \right\rangle$$

If the dice is loaded, instead, some outcomes may be more-likely. For instance, we could have the following probability space:

$$P(\omega) = \left\langle \frac{1}{2}, 0, \frac{1}{6}, 0, \frac{1}{6}, \frac{1}{6} \right\rangle$$

Definition 12: Event

Given a probability space $P : \Omega \to [0, 1]$, an **event** is a subset of Ω. The probability of an event $A \subseteq \Omega$ is given by the sum of the probabilities of all of the outcomes in the event.

$$P(A) = \sum_{\omega \in A} P(\omega)$$

Using the previous loaded dice example, the event "roll a value lower than 4" is described by the event $A = \{1, 2, 3\}$ and its probability is $P(A) = \frac{5}{6}$. Events are a simple way to reason about probability. However, sometimes they aren't as intuitive as they look. For this reason, we usually prefer to work with *random variables*.

Definition 13: Random variable

Given a probability space $P : \Omega \to [0, 1]$, a **random variable** is a function $X : \Omega \to B$, where B is an arbitrary set of values.

A random variable associates one of the values in B to every single element of the sample space. To work with a random variable $X : \Omega \to B$, we often consider the event "$X = x_i$", where $x_i \in B$. This event is equivalent to the set $\{\omega \in \Omega \mid X(\omega) = x_i\}$. Due to this, a random variable can be viewed both as a function and a variable.

$$P(X = x_i) = P(\{\omega \in \Omega \mid X(\omega) = x_i\}) = \sum_{\substack{\omega \in \Omega : \\ X(\omega) = x_i}} P(\omega)$$

When we're working with a boolean random variable, i.e. when the value set B is $\{0, 1\}$, we often denote the event "$X = 1$" with X, while the event "$X = 1$" is denoted with $\neg X$. This allows us to write intersections and unions of events as simple conjunctions and disjunctions of boolean random variables.

$$P(\neg A \wedge B) = \sum_{\substack{\omega \in \Omega : \\ A(\omega) = 0, B(\omega) = 1}} P(\omega)$$

Instead of working with a probability space in order to compute the probabilities of each value assumable by a random variable, we often directly consider a **probability distribution**, a function assigning a probability value to all possible assignments of a random variable. The *joint probability distribution* for a set of random variables gives the probability of every atomic joint event on those random variables.

$P(\text{PlayTennis} \wedge \text{Weather})$		Weather			
		Sunny	Rainy	Cloudy	Snowy
PlayTennis	True	0.576	0.02	0.064	0.01
	False	0.144	0.08	0.016	0.09

Figure 3.1: Example of joint probability distribution

We notice that, by definition, the probability of the conjunction of two events is not always equal to the product of the two probabilities of the events.

$$P(A \wedge B) \stackrel{?}{=} P(A) \cdot P(B)$$

When the equality holds, we say that the two events are **independent** from each other. For the probability of the disjunction of two events, instead, we can always use the following formula derived from the *inclusion-exclusion principle*:

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

Random variables can also influence each others. For instance, if we know that some outcomes of an event A are more likely to happen when some conditions are met, we can restrict our interest to such cases. The probability of an event A given that an event B happened is written $P(A \mid B)$. This is also known as **conditional** (or *posterior*) probability and its defined as:

$$P(A \mid B) = \frac{P(A, B)}{P(B)}$$

where $P(A, B) = P(A \wedge B)$. We notice that the events A and B are independent if and only if $P(A \mid B) = P(A)$. Sometimes, we may be also be given *conditional probability distributions*.

$P(\text{PlayTennis} \mid \text{Weather})$		Weather			
		Sunny	Rainy	Cloudy	Snowy
PlayTennis	True	0.8	0.2	0.8	0.1
	False	0.2	0.8	0.2	0.9

Figure 3.2: Example of conditional probability distribution

When this is the case, conditional probability can be used to compute joint probability. This is also known as the *product rule*.

$$P(A, B) = P(A \mid B) \cdot P(B)$$

When the values of a random variable Y are mutually exclusive, we may compute the total probability of another random variable X through joint probability or conditional probability:

$$\begin{aligned} P(X = x) &= P((X = x, Y = y_1) \vee \ldots \vee (X = X, Y_k)) \\ &= \sum_{i=1}^{k} P(X = x, Y = y_i) \\ &= \sum_{i=1}^{k} P(X = x \mid Y = y_i) \cdot P(Y = y_i) \end{aligned}$$

Conditional probability can clearly be viewed as a *normalization factor* α applied to a joint probability. For instance, given the conditional probability, we have that:

$$P(A \mid B) = \frac{P(A, B)}{P(B)} = \alpha P(A, B)$$

where the normalization factor is $\alpha = \frac{1}{P(B)}$. Viewing such probability as nothing more than a normalized instance of the joint one allows us to reason about maximizing and minimizing values in a simpler way: since each element is afflicted by this constant factor, we can just ignore.

$$\underset{x \in \text{Values}(X)}{\arg\max}\ P(X = x \mid Y = y) = \underset{x \in \text{Values}(X)}{\arg\max}\ \alpha P(X = x, Y = y) = \underset{x \in \text{Values}(X)}{\arg\max}\ P(X = x \mid Y)$$

The most important rule derived from the very definition of conditional probability is **Bayes' rule**. This rule allows us to invert the order of the events: to compute the probability of A given B, we can use the probability of B given A

Proposition 1: Bayes' rule

Given two events A and B, it holds that:

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$

Bayes' rule becomes even stronger in the context of maximizing and minimizing values thanks to normalization:

$$\underset{x \in \text{Values}(X)}{\arg\max}\ P(X = x \mid Y = y) = \underset{x \in \text{Values}(X)}{\arg\max}\ P(Y = y \mid X = x) \cdot P(X = x)$$

Moreover, when the conditions $Y_1, \ldots, Y_k$ are independent from each other, we also get that:

$$P(X \mid Y_1, \ldots, Y_k) = \alpha P(Y_1, \ldots, Y_k \mid X) = \alpha P(Y_1 \mid X) \cdot \ldots \cdot P(Y_k \mid X)$$

Another interesting consequence of the product rule is the *chain rule*, where we repeatedly apply the product rule on each joint probability:

$$\begin{aligned} P(X_1, \ldots, X_{n-1}, X_n) &= P(X_1, \ldots, X_{n-1}) \cdot P(X_n \mid X_1, \ldots, X_n) \\ &= P(X_1, \ldots, X_{n-2}) \cdot P(X_{n-1} \mid X_1, \ldots, X_{n-2}) \cdot P(X_n \mid X_1, \ldots, X_{n-1}) \\ &= \prod_{i=1}^{n} P(X_i \mid X_1, \ldots, X_{i-1}) \end{aligned}$$

The chain rule is usually used with **Bayesian networks**, a graphical notation for conditional independence assertions and hence for compact specification of full joint distributions.

Given a set of variables, a Bayesian networks is a directed acyclic graph containing one node for each variable. The directed edges of the graph describe influences between variables: an edge (Y, X) implies that Y influences X. Each influence (Y, X) is associated with a conditional probability $P(X \mid Y)$. In the simplest case, the conditional distribution is represented as a **Conditional Probability Table (CPT)** giving the distribution over X for each combination of its parent values.

For instance, consider the following situation. Suppose, that while we're working, our neighbor John calls to say that our alarm is ringing, but our other neighbor Mary doesn't call. However, we know that the alarm can be set off by minor earthquakes. Can we use probabilities to decide if there is a burglar?

First, we define five variables: Burglar, Earthquake, Alarm, JohnCalls, MaryCalls. Then, we consider the relations between such variables based on what we know:

- A burglar can set the alarm, hence Burglar $\in$ Parent(Alarm)
- An earthquake can set the alarm, hence Earthquake $\in$ Parent(Alarm)
- The alarm can cause John to call, hence Alarm $\in$ Parent(JohnCalls)
- The alarm can cause Mary to call, hence Alarm $\in$ Parent(MaryCalls)

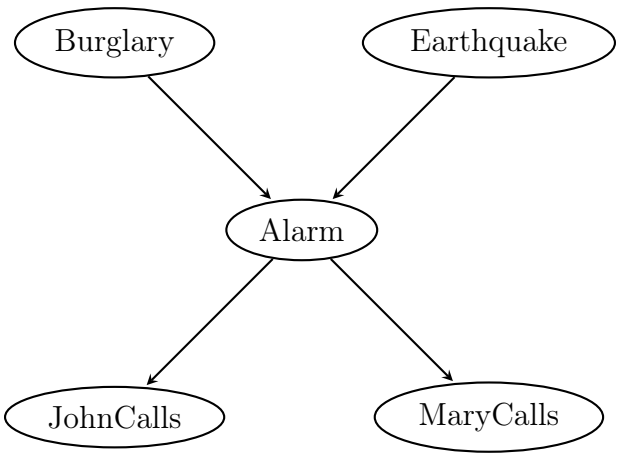

Figure 3.3: The Bayesian network representing the *BurlaryProblem*

For each variable, we construct a CPT representing the conditional probabilities for each of the parent attributes.

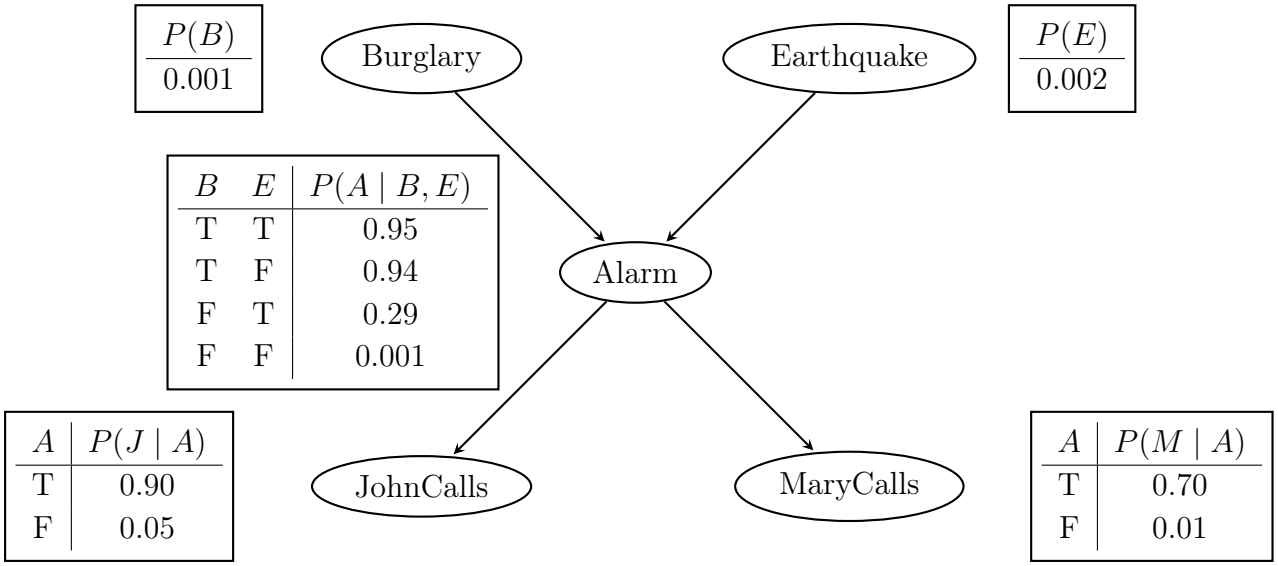

Figure 3.4: The Bayesian network representing the *BurlaryProblem*

Using the chain rule, the probability that both Mary and John call when the alarm rings while there is no burlar or earthquake is given by:

$$P(\neg \mathrm{B}, \neg \mathrm{E}, \mathrm{A}, \mathrm{J}, \mathrm{M}) = P(\neg B) \cdot P(\neg E) \cdot P(A \mid \neg B, \neg E) \cdot P(J \mid A) \cdot P(M \mid A) \approx 0.00063$$